SYNACKTIV
DIGITAL SECURITY

Rapport technique d'évaluation

Olvid Android

- DIFFUSION PUBLIQUE -

Version 1.2

Olvid

02/06/2021

Identification du document

Caractéristiques

Objet	Rapport technique d'évaluation - Olvid Android
Nombre de pages	91
Diffusion	DIFFUSION PUBLIQUE

Historique

Version	Date	État
1.0	24/03/2021	Première version
1.1	17/05/2021	Suppression de la vulnérabilité V-03 puisqu'une contre mesure existe
1.2	02/06/2021	Anonymisation du rapport et passage en classification DIFFUSION PUBLIQUE

Table des matières

1. Identification du produit évalué et de la cible

1.1. Références et versions de la cible d'évaluation

Nom de l'éditeur	Olvid
Nom du produit	Olvid
N° de version analysée	0.9.2
Correctifs éventuels appliqués	N/A
Cible de sécurité	Cible de sécurité CSPN – Olvid Android | version 1.1 (2020-12-16)
Domaine technique CSPN	Communications sécurisées
Divers	APK: io.olvid.messenger.apk (SHA256: 7f0f653b7620d2c07135f2f794dc9e7e79ce83f51eb0af228973b4bf6ded7ad1)

1.2. Procédure d'identification du produit évalué

Le numéro de version de l'application Olvid est disponible sur le Play Store de Google dans la page « Détails » :

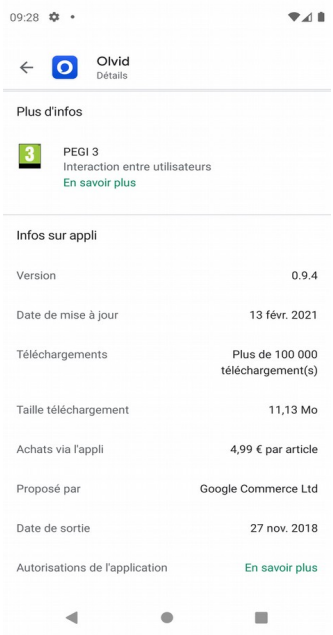

Illustration 1: Détails de l'application Olvid depuis

Le numéro de version est également disponible directement depuis l'application dans le menu « À propos » :

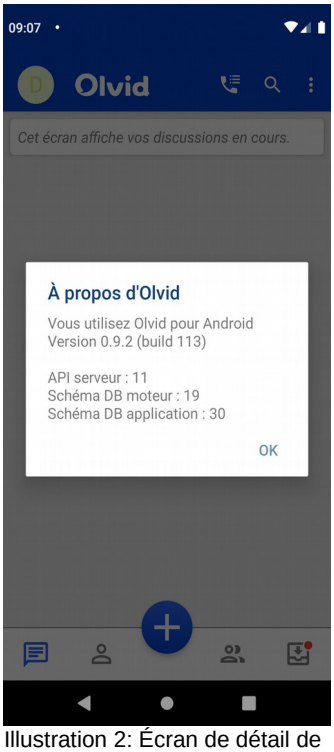

Illustration 2: Écran de détail de la version d'Olvid

Bien que seule la distribution binaire soit l'objet de l'évaluation, l'éditeur a livré le code source de l'application.

Le SHA256 de l'archive fournie est le suivant :

```
SHA256 (android-client.zip) =
a657c080459be02efce6b2c8b4cb6666911a3e6f0c7d6eaa295fc0aa6fc8ad85
```

La hiérarchie (à deux niveaux de profondeur) est la suivante :

```
obv_engine
├── build.gradle
├── engine
│   ├── build.gradle
│   ├── libs
│   ├── local.properties
│   ├── proguard-rules.pro
│   └── src
├── gradle
│   └── wrapper
├── gradle.properties
├── gradlew
├── gradlew.bat
├── local.properties
└── settings.gradle
obv_messenger
├── app
│   ├── build.gradle
│   ├── google-services.json
```

```
│   ├── proguard-rules.pro
│   ├── schemas
│   └── src
├── build.gradle
├── gradle
│   └── wrapper
├── gradle.properties
├── gradlew
├── gradlew.bat
├── libwebrtc
│   ├── build
│   ├── build.gradle
│   └── libwebrtc.aar
├── local.properties
└── settings.gradle

12 directories, 20 files
```

2. Détail des travaux d'évaluation

2.1. Analyse de conformité et de robustesse

2.1.1. Problème de sécurité et environnement

2.1.1.1. Spécification de besoin et problème de sécurité

Conforme au chapitre 2.1 Description générale du produit de **[CIBLE]**.

2.1.1.2. Utilisation et environnement / Argumentaire du produit

Conforme au chapitre 2.2. Description de la manière d'utiliser le produit de **[CIBLE]**.

2.1.1.3. Avis d'expert et vulnérabilités potentielles identifiées

Le générateur pseudo-aléatoire implémenté par la TOE et décrit dans la section Générateur de nombres pseudo-aléatoires page 23 est initialisé avec une graine issue de *java.security.SecureRandom,* qui est une librairie fournie par le système *Android*.

La **[CIBLE]** suppose que le système est sain et au regard de la difficulté d'implémenter un générateur aléatoire sans un minimum de confiance en la plateforme, l'ajout d'une hypothèse couvrant la possibilité que ce générateur ne soit pas défectueux ou malveillant semble de fait raisonnable.

L'hypothèse *H.SecureRandom* est donc ajoutée aux hypothèses de l'environnement déjà présentes dans la cible et est formulée comme suit : il est supposé que le générateur aléatoire exposé par la classe *java.security.SecureRandom* de la plateforme mobile est conforme aux règles et recommandations de l'ANSSI dans le RGS.

De plus, le chiffrement AES et la génération de HMAC-SHA256 est aussi délégué au système via la bibliothèque *javax.crypto*, respectivement *javax.crypto.Cipher* et *javax.crypto.MAC*. La cible n'émet aucune hypothèse quant à la validité de l'implémentation de ces bibliothèques.

L'hypothèse *H.CryptoLib* est donc ajoutée aux hypothèses de l'environnement déjà présentes dans la cible et est formulée comme suit : il est supposé que la bibliothèque *javax.crypto* fournie par le système est correctement implémentée et se comporte conformément à sa spécification.

2.1.2. Mise en œuvre du produit

2.1.2.1. Installation

Dans le cadre de l'évaluation, l'application a été installée via **Google Play** depuis le téléphone. Cela correspond à une utilisation normale. L'utilisateur utilisera la même méthode d'installation. Pour les besoins de l'étude, les modèles suivants de téléphones ont été utilisés :

- Google Pixel 4 avec Android 10 ;
- Émulateur AVD avec Android 9.

Le type de modèle ainsi que la version précise d'Android n'entraînent pas de différence majeure pour la mise en œuvre du produit. Le choix d'utiliser un émulateur en version d'API 28 (Android 9) a permis de simplifier l'analyse en disposant non seulement d'un accès *root* au téléphone ainsi que d'un accès en écriture à l'ensemble du système de fichier.

Une fois l'application installée, plusieurs étapes de configuration sont proposées lors du premier lancement. Tout d'abord un écran de bienvenue permet de restaurer une sauvegarde antérieure. Pour l'installation initiale, l'option « Continuer en tant que nouvel utilisateur » est utilisée.

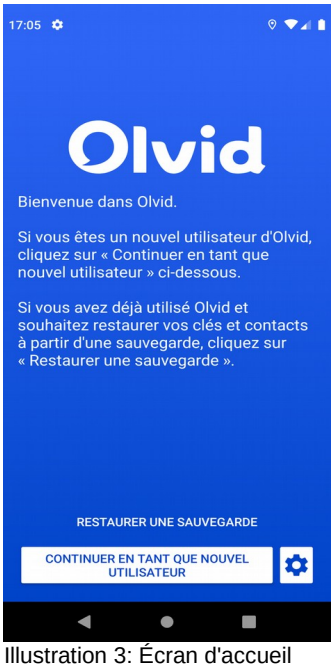

Illustration 3: Écran d'accueil

L'écran suivant permet à l'utilisateur de préciser son identité. Nom, Prénom, Société et Poste peuvent être renseignés. Seul le prénom est un champ obligatoire.

Illustration 4: Création de l'identité

Une fois l'identité renseignée les étapes de configuration de l'application sont terminées. L'utilisateur découvre alors un premier écran récapitulatif.

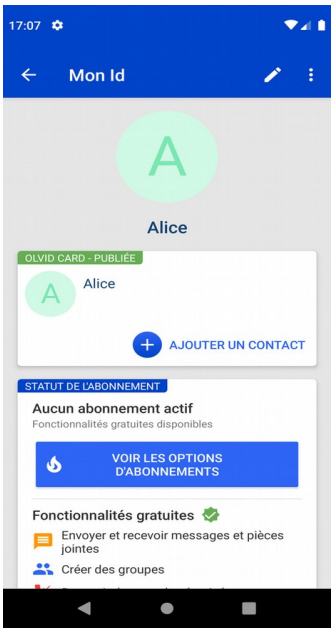

Illustration 5: Récapitulatif de l'identité créée et du statut

Il est alors possible d'ajouter un contact. L'utilisateur se voit présenter un QR code représentant son identité ainsi qu'un bouton permettant de scanner le QR code d'un autre utilisateur.

Illustration 6: Écran d'ajout d'un contact

Il est aussi possible de partager un lien d'invitation via n'importe quel autre canal.

Une fois la demande de contact envoyé l'utilisateur reçoit une notification de demande d'ajout de contact.

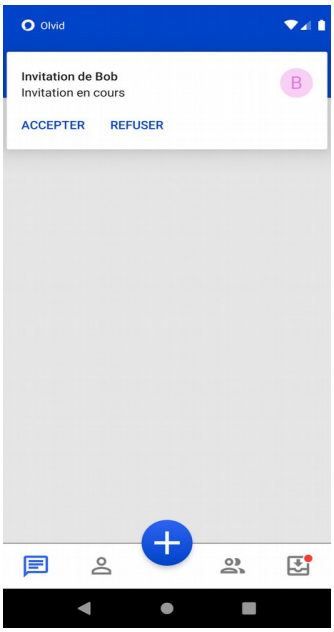

Illustration 7: Notification de demande d'ajout de contact

Lorsque la demande est acceptée, les deux utilisateurs reçoivent une notification permettant l'échange du code court d'authentification.

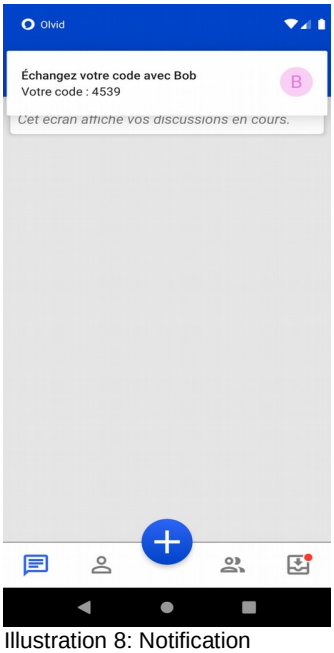

Illustration 8: Notification d'échange de code court

En cliquant sur cette notification l'utilisateur se voit présenter un écran permettant l'échange du code court. Un court texte explicatif permet de sensibiliser l'utilisateur aux contraintes liées à l'échange du code d'authentification.

Le message explicatif est relativement succin et suffisamment clair pour être facilement compris par un utilisateur.

Illustration 9: Écran d'échange du code d'authentification

En cas d'erreur lors de la saisie du code d'authentification un message rouge apparaît.

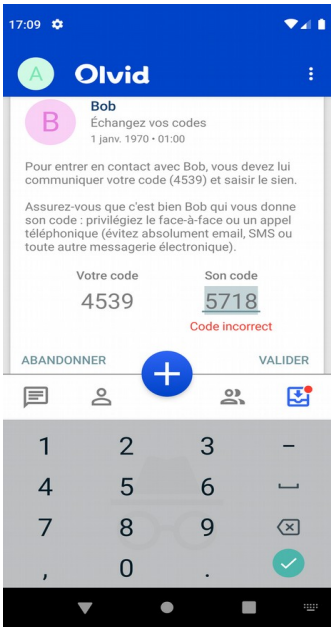

Illustration 10: Message d'erreur pour saisie d'un code

Lorsque le code est correct, le message explicatif est mis à jour et une tique bleue apparaît.

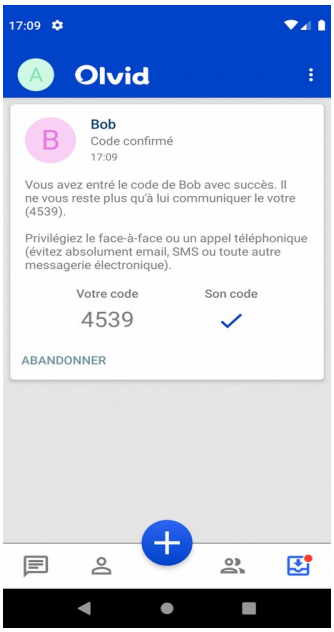

Illustration 11: Validation du code d'authentification

Lorsque les deux utilisateurs ont correctement saisi les codes d'authentification, un message explique qu'un canal de communication est en cours de création.

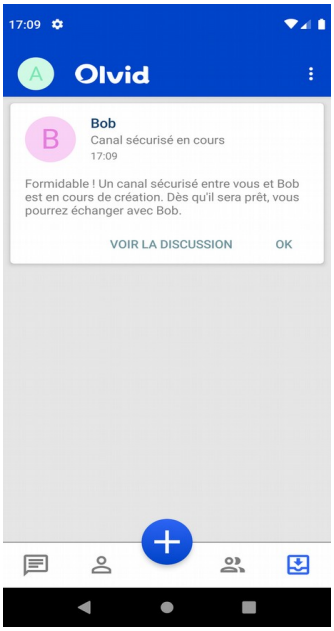

Illustration 12: Message expliquant la création d'un canal

Il est immédiatement possible d'ouvrir la conversation correspondant au contact ajouté. Un message précise que le canal de communication est encore en cours de création et que l'envoi de message peut être retardé.

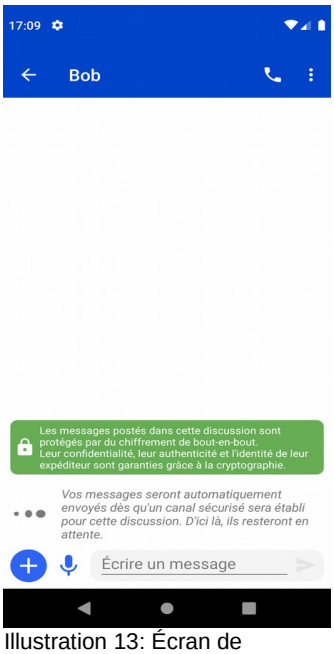

Illustration 13: Écran de conversation

Une fois le canal de communication créé, il est possible de recevoir un nouveau message.

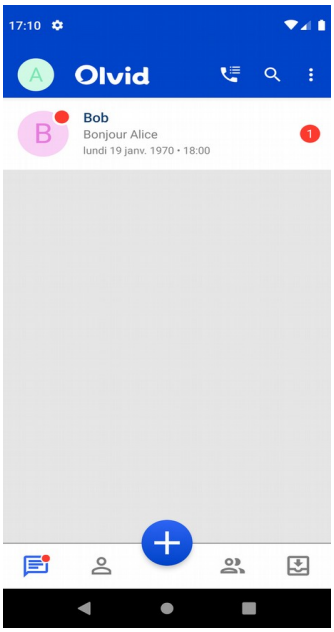

Illustration 14: Réception d'un nouveau message

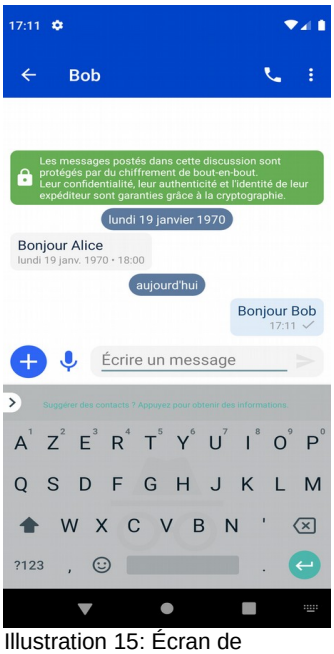

Illustration 15: Écran de conversation avec des

2.1.2.2. Facilité d'emploi

L'application est simple d'utilisation et intuitive. Chaque écran nécessitant une action est pourvu d'un court texte explicatif permettant à l'utilisateur de comprendre ce qui lui est demandé. Aucune action incomprise ou mal réalisée par un utilisateur légitime ne semble mettre en péril la sécurité de l'application.

2.1.2.3. Avis d'expert et vulnérabilités potentielles identifiées

Lors de l'introduction de contacts homonymes via un contact tiers, un comportement de l'interface graphique pouvant porter à confusion a été remarqué. Ce dernier est remonté dans V-02 : CONFUSION-HOMONYMES page 86. Il ne constitue pas une vulnérabilité et est remonté à titre indicatif sans remettre en cause la sécurité d'Olvid.

2.1.3. Conception et développement

2.1.3.1. Documentation et fournitures

Référence	Nom du fichier	Titre	Description	Version	Date
[CIBLE]	2021-03-19 CSPN_cible_Olvid_Android_v1.2.pdf	Cible de Sécurité CSPN – Olvid	Cible de sécurité de l'application mobile Olvid	1.2	19/03/2021
[DOC_SOURCE]	android-client.zip	N/A	Code source de l'application Android Olvid	0.9.2 build 113	N/A
[DOC_CRYPTO]	2020-12-15 Olvid - Spécifications cryptographiques.pdf	Olvid – Specification of Olvid – Application and Server	Spécifications d'implémentation de l'application et du serveur Olvid	N/A	15/12/2020
[SAS_PROOF]	Olvid - Preuve du protocole de Trust	Security Analysis of Olvid's Trust	Publication de M.Abdalla sur la	N/A	N/A

	Establishment.pdf	Establishment Protocol	preuve de la sécurité du protocole de Trust Establishment utilisé dans l'application Olvid		
[RTE_IOS]	Synacktiv-Olvid-CSPN-Olvid-0.8.2-RTE-v1.1.pdf	Rapport d'évaluation technique – Olvid	Rapport d'évaluation technique pour la CSPN concernant le produit Olvid pour iOS	1.1	09/06/2020

2.1.3.2. Analyse de la spécification cryptographique

L'ensemble des mécanismes cryptographiques est décrit en détail dans [DOC_CRYPTO]. La spécification cryptographique contient quelques différences par rapport à celle fournie pour la CSPN Olvid iOS. Néanmoins les chapitres suivants *Part I – Notations and Conventions*, *Part II – Cryptographic Primitives*, *Part III – Encodings, Part IV – Message Structure and Communication Channels*, *Part V – Cryptographic Protocols* et *Part VI – Keys and Contacts Backup* sont identiques aux exceptions suivantes près :

- un décalage dans la numérotation des références ;
- une correction orthographique ;
- l'ajout de champs permettant la gestion des appels dans la structure des messages ;
- l'ajout d'un protocole de gestion des groupes ;
- l'ajout d'un paragraphe précisant les détails de la vulnérabilité **V-03 REJEU-COMMITMENT** de **[RTE_IOS]** dans la partie *24 Trust Establishment Protocol with SAS ;*
- l'ajout de précision sur l'implémentation cryptographique de certains protocoles. Néanmoins, ces détails étaient déjà connus via l'analyse statique lors de la CSPN Olvid iOS.

L'analyse de la spécification cryptographique mène donc aux mêmes conclusions que l'analyse réalisée pour la CSPN Olvid iOS pour ce qui est de :

- l'authentification des utilisateurs ;
- les messages courts d'authentification ;
- l'échange de messages ;
- la communication entre application et serveur ;
- la gestion des clés.

L'analyse originale est disponible au chapitre 2.1.3.2 de **[RTE_IOS]**. Par souci de praticité pour le lecteur, l'analyse est reprise ci-après dans les chapitres :

- Authentification des utilisateurs page 16 ;
- Message court d'authentification page 17 ;
- Échange de messages page 18 ;
- Communication entre application et serveur page 20 ;
- Gestion des clés page 21.

Une analyse complémentaire du chiffrement des pièces jointes est disponible dans Chiffrement des pièces jointes page 24.

Authentification des utilisateurs

L'authentification des utilisateurs d'Olvid est réalisée par un échange de clés publiques et par confirmation de l'échange via un canal authentique extérieur à l'application. Le protocole d'établissement de confiance est décrit en détail dans la section 24 de **[DOC_CRYPTO]**.

Une fois l'échange de clés publiques réalisé, un secret partagé est créé entre deux utilisateurs d'Olvid. Les clés publiques n'entrent alors plus en jeu dans les échanges entre ces deux utilisateurs. L'authentification est réalisée en utilisant une primitive de chiffrement authentifié.

Cryptographie asymétrique

Curve25519

La cryptographie asymétrique utilisée par Olvid pour la création de secrets partagés est basée sur la courbe elliptique *Curve25519*.

Les paramètres choisis pour la courbe sont disponibles dans la section 12.1 de **[DOC_CRYPTO]**. Les voici, exprimés dans le format des courbes Edwards :

```
x2 + y2 = 1 + dx2y2

d = 20800338683988658368647408995589388737092878452977063003340006470870624536394
G.x = 9771384041963202563870679428059935816164187996444183106833894008023910952347
G.y = 463168356949264781694283940034751631413079938662562256157830336031652518559 60
```

La correspondance avec les paramètres tels que présentés habituellement au format Weierstrass est la suivante :

```
Bv2 = u3 + Au2 + u

A = 2 × (1 + d) ÷ (1 - d)
B = 4 ÷ (1 - d)
x = u ÷ v
y = (u - 1) ÷ (u + 1)
```

Règle	Conformité	Justification
RègleECp	Conforme	L'ordre est de 2^{252}+27742317777372353535851937790883648493.
RecommandationECp	Conforme	L'ordre du sous-groupe correspondant à Curve25519 est premier.

Million Dollars Curve

La cryptographie asymétrique utilisée par Olvid pour la signature de messages est basée sur la courbe elliptique MDC.

Les paramètres choisis pour la courbe sont disponibles dans la section 12.2 de **[DOC_CRYPTO]**. Les voici, exprimés dans le format des courbes Edwards.

```
x2 + y2 = 1 + dx2y2

p = 109112363276961190442711090369149551676330307646118204517771511330536253156371
d = 39384817741350628573161184301225915800358770588933756071948264625804612259721
G.x = 8254980322220239934002446203296494251202585681870041425472636420509673142431 5
G.y = 9154954563741573442265828879911904175637825952309714780781339691512593281144 5
```

Règle	Conformité	Justification
RègleECp	Conforme	L'ordre est de 2^{253}+1280407966457577318273139946620139943727176149084299838269878029219638076825 1.
RecommandationECp	Conforme	L'ordre du sous-groupe correspondant à MDC est premier.

Génération des clés

Les clés asymétriques utilisées pour représenter un utilisateur sont générées via le mécanisme présenté en section 14 de **[DOC_CRYPTO]**.

La génération des clés asymétriques s'appuie sur le générateur de nombre pseudo aléatoire présenté en Générateur de nombres pseudo-aléatoires page 23. À partir d'un nombre pseudo aléatoire, un point sur la courbe elliptique utilisée est choisi en faisant une multiplication scalaire du générateur associé à la courbe.

Règle	Conformité	Justification
RègleAléaLocal-1	Conforme	Le mécanisme de génération de clé publique utilise un générateur de nombre pseudo aléatoire conforme à la **[RGS_B]**.

Message court d'authentification

L'authentification des clés publiques échangées entre les contacts se fait via l'échange d'un message court d'authentification (SAS).

Le canal d'échange de ce message doit être authentifié (de vive voix, par téléphone) mais ne doit pas forcément être confidentiel.

La garantie cryptographique d'authentification est obtenue via le protocole présenté en figure 2 de **[DOC_CRYPTO]**. Le protocole utilisé pour l'authentification des clés publiques dispose d'une preuve de sécurité détaillée dans **[PROOF_SAS]**.

Chiffrement authentifié

Les messages applicatifs et protocolaires sont échangés via un canal chiffré et authentifié.

Chaque message est chiffré avec AES-256 en mode CTR. L'authentification retenue est HMAC basé sur SHA-256. Deux clés temporaires sont générées pour chaque message. La première pour chiffrer le message envoyé et la seconde est utilisée pour la production du HMAC.

Règle	Conformité	Justification
RègleHash	Conforme	SHA-256 produit des hachés de 256 bits. Il n'existe pas d'attaque connue permettant de trouver des collisions de hash plus rapidement qu'avec le paradoxe des anniversaires sur SHA-256.
RègleIntegSym	Conforme	La primitive de hachage SHA-256 utilisé dans HMAC SHA-256 est conforme à la **[RGS_B]**. Il n'existe pas d'attaque plus efficace que le paradoxe des anniversaires sur

		HMAC utilisé avec SHA-256.
RecommandationIntegSym	Conforme	HMAC dispose de plusieurs preuves de sécurité.

Échange de messages

Les messages applicatifs et protocolaires sont échangés via un canal chiffré. Le canal chiffré est décomposé en deux sous canaux. Un canal d'envoi et un canal de réception.

Chaque canal dispose d'une clé de chiffrement symétrique propre. Ces clés sont dérivées à partir d'une clé commune en se basant sur les *deviceUID* de chacun des participants.

L'échange de clé initial est présenté en section Échange de clé symétrique page 18.

Échange de clé symétrique

Une fois les clés publiques échangées et leur authenticité confirmées, une clé symétrique est générée et échangée. Cette clé sera utilisée par la suite pour chiffrer l'ensemble des messages applicatifs et protocolaires entre deux utilisateurs d'Olvid.

L'échange de clé est authentifié en utilisant le mécanisme de signature *Schnorr* présenté en section 14 de **[DOC_CRYPTO]**. Le mécanisme *Schnorr* est utilisé sur la courbe elliptique *MDC* présentée en Cryptographie asymétrique page 16.

Règle	Conformité	Justification
RecomSignAsym	Conforme	*Schnorr* dispose d'une preuve de sécurité.

La génération et l'échange de clés symétriques se font via le *mécanisme d'échange de clé* (KEM) présenté en section 16 de **[DOC_CRYPTO]**. Le KEM repose sur une implémentation standard de ECIES. ECIES dispose de plusieurs preuves de sécurité. L'implémentation de ECIES dans Olvid repose sur la cryptographie asymétrique présentée en section Cryptographie asymétrique page 16 ainsi que sur le générateur de clé présenté en section Dérivation de clé page 18. Ces deux composants sont eux même conformes au **[RGS_B]**.

L'échange de clés symétriques est donc conforme au **[RGS_B]**.

Dérivation de clé

La dérivation des clés symétriques est utilisée dans l'application pour générer une première clé de chiffrement symétrique lors d'une mise en relation entre deux contacts (section Échange de clé symétrique page 18) ainsi que dans le cadre du mécanisme de cliquet participant à la garantie de Perfect Forward Secrecy.

Le mécanisme de dérivation de clé choisi est basé sur le générateur de nombre pseudo aléatoire présenté en section Générateur de nombres pseudo-aléatoires page 23.

Règle	Conformité	Justification
RègleAléaLocal-1	Conforme	Le PRNG utilisé est conforme à la **[RGS_B]**.

Chiffrement des messages

Les messages applicatifs et protocolaires sont échangés via un canal chiffré. L'algorithme de chiffrement retenu est AES256 en mode CTR.

Règle	Conformité	Justification
RègleCléSym RecommandationCléSym	Conforme	La taille des clés choisie pour le chiffrement symétrique est de 256 bits.
RègleBlocSym	Conforme	La taille des blocs pour AES-256 est de

RecommandationBlocSym		128 bits.
RègleAlgoBloc	Conforme	La meilleure attaque connue à ce jour à l'encontre de AES-256 nécessite $2^{254.3}$ opérations.
RecommandationAlgoBloc	Conforme	AES est probablement l'algorithme de chiffrement par bloc qui a été le plus éprouvé dans le milieu académique.
RègleModeChiff	Conforme	Il n'existe pas d'attaque plus efficace que le paradoxe des anniversaires sur le mode de chiffrement CTR utilisé avec des Nonces aléatoires
RecommandationModeChiff	Conforme	AES-256 CTR utilise un mécanisme de Nonce et est donc non-déterministe. AES-256 CTR est utilisé conjointement à HMAC SHA-256. Le mode de chiffrement CTR dispose de plusieurs preuves de sécurité.

Le chiffrement utilise AES256 en mode CTR. L'implémentation de AES est ici propre à Android et dépend totalement des modifications apportées par le constructeur du téléphone aux sources d'Android (AOSP). Cette implémentation est supposée correcte pour les téléphones testés, hypothèse soutenue par les tests menés par le CESTI grâce aux vecteurs de tests du projet Olvid. AES256 est utilisé avec un vecteur d'initialisation. Ce dernier est construit à partir d'un nonce de 8 octets. Ces 8 octets sont obtenus à l'aide du *prng* analysé dans Générateur de nombres pseudo-aléatoires page 23. *C*et aléa est régénéré à chaque message et est indépendant du contenu. Il est supposé que ces 8 octets aléatoires sont concaténés avec 8 bytes du compteur du mode CTR pour obtenir un vecteur de 128 bits. Cette hypothèse sur la concaténation est soutenue par le fait que la réimplémentation du CESTI qui fonctionne de cette manière permet le déchiffrement des messages envoyés. Ce résultat est détaillé dans les travaux de Analyse dynamique du chiffrement des messages applicatifs page 68.

Sous ces hypothèses, générer une collision d'*IV* pour un même clair nécessiterait soit un débordement du compteur, c'est-à-dire l'envoi d'un message de 16 x 2^{64} octets, ce qui n'est pas réaliste, soit une collision sur les 8 octets d'aléas. Avec une attaque sur les anniversaires, il faudrait générer environ $5x10^{9}$ messages en chiffrant le même clair. Il semble peu probable qu'un attaquant soit en mesure de forcer une victime à envoyer autant de messages, tout en connaissant le clair et le chiffré. Ceci est d'autant plus improbable que les clés de chiffrements sont rafraîchies tous les 100 messages à l'aide de mécanismes de cliquets.

En effet, deux mécanismes de cliquet permettent de garantir la Perfect Forward Secrecy. Le premier est un cliquet simple, le second un cliquet complet.

Le cliquet simple est effectué à chaque envoi d'un message. La clé de chiffrement symétrique est dérivée en utilisant le générateur de clé présenté en section Génération des clés page 17. À chaque envoi, la clé utilisée est supprimée. Du côté de la réception, un ensemble de 100 clés est pré-généré, celle correspondant au message reçu est supprimée après déchiffrement du message. Le fonctionnement du cliquet est expliqué ci-après.

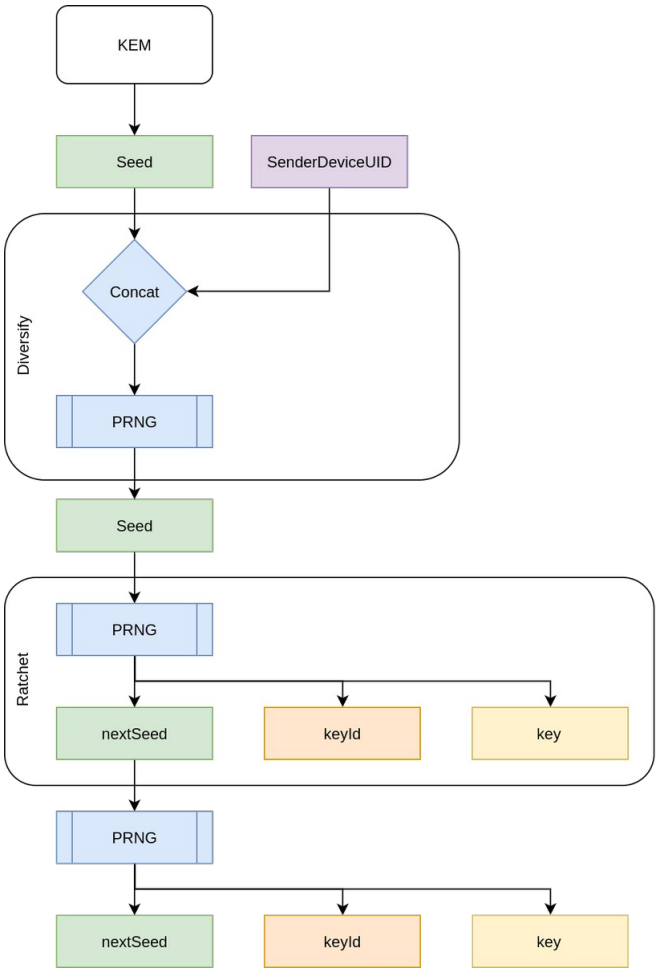

Illustration 16: Fonctionnement du cliquet simple

Le cliquet complet est, quant à lui, initié à intervalles réguliers. Les critères de renouvellement sont les suivants :

1. Nombre de messages chiffrés > 100
2. Temps depuis le dernier échange de clé > 7 jours

Ce mécanisme consiste simplement à relancer un échange de clé tel que présenté en section Échange de clé symétrique page 18. Néanmoins, l'authentification des correspondants n'est pas réalisée via leur clé publique, mais via des clés asymétriques éphémères échangées en utilisant l'ancien canal. L'authentification des nouvelles clés repose donc sur la validité des anciennes clés.

Communication entre application et serveur

Pour communiquer avec le ou les serveurs liés aux identités des correspondants, Olvid utilise l'API *HTTPUrlConnection.* Cette API gère de manière transparente les flux TLS ou en clair.

Néanmoins, la sécurité des échanges ne repose pas sur le chiffrement fourni par la couche TLS. Les mécanismes d'authentification des utilisateurs détaillés dans Authentification des utilisateurs page 16 et Échange de messages page 18 permettent d'assurer l'authenticité, l'intégrité et la confidentialité des données.

L'étude de ces flux est en dehors de la TOE.

Gestion des clés

Différentes clés sont utilisées par l'application Olvid. Le tableau suivant en donne une liste exhaustive.

Clé	Type	Contexte	Durée de vie
Clé d'identification	Clé asymétrique	Clé unique identifiant un contact	Durée de vie du compte utilisateur
Clé d'établissement de canal	Clé asymétrique	Clé éphémère utilisée pour réaliser un échange de secret entre deux utilisateurs afin de créer un canal sécurisé.	Temps de réalisation du protocole d'échange de clé
Clé de canal	Clé symétrique	Clé secrète utilisée pour chiffrer la clé de message.	Temps entre deux envois de messages
Clé de message	Clé symétrique	Clé éphémère utilisée pour chiffrer le corps du message.	Temps de vie du message
Clé de cliquet complet	Clé asymétrique	Clé éphémère utilisée pour réaliser un échange de secret entre deux utilisateurs afin de renouveler la clé de canal.	Temps de réalisation du protocole d'échange de clé
Clés de chiffrement de la sauvegarde du carnet de contact	Clé asymétrique	Paire de clés de dé/chiffrement ECIES pour la sauvegarde du carnet de contact (*backup*). Initialisées à partir du PRNG instancié avec une graine aléatoire.	Temps de chiffrement et du déchiffrement de la sauvegarde.
Clé de chiffrement des pièces jointes	Clé symétrique	Clé utilisée pour chiffrer une unique pièce jointe.	Temps de vie du message contenant la pièce jointe sur le serveur

Aucune clé cryptographique gérée par Olvid n'est réutilisée en dehors de son contexte initial. Les auteurs d'Olvid ont attaché un soin particulier à utiliser les clés au minimum et à générer de nouvelles clés dès que possible.

Règle	Conformité	Justification
RègleGestSym	Conforme	Chaque clé est utilisée pour un seul et unique usage. Les clés sont différenciées ou dérivées en utilisant le mécanisme présenté en section Dérivation de clé page 18. Ce mécanisme est conforme au RGS.
RègleGestAsym	Conforme	Les bi-clés sont employées pour un usage unique. Les clés hiérarchiquement importantes sont générées en utilisant le mécanisme présenté en section Génération des clés page 17. Ce mécanisme est conforme au RGS.

Stockage des clés

Stockage local

Il est nécessaire pour l'application Olvid de stocker les clés privées de manière persistante. L'application utilise le mécanisme de base de données SQLite fourni par Android à cette fin. On retrouve donc la définition d'une classe

représentant une identité privée *io.olvid.engine.identity.databases.OwnedIdentity.* Cette classe permet la création d'une table SQL nommée *owned_identity* dans la base de données gérée par l'application.

On retrouve cette base dans */data/data/io.olvid.messenger/no_backup/engine_db.sqlite*.

```
$ adb pull /data/data/io.olvid.messenger/no_backup/engine_db.sqlite
$ sqlite3 engine_db.sqlite "SELECT * FROM owned_identity"
https://server.olvid.io||1|1|00000177-bfa5-6b63-6aa2-153e23b35468|1
```

Le deuxième champ correspond à la sérialisation d'une instance de *io.olvid.engine.datatypes.PrivateIdentity*.

```
public class OwnedIdentity implements ObvDatabase {
    static final String TABLE_NAME = "owned_identity";

    private final IdentityManagerSession identityManagerSession;

    private Identity ownedIdentity;
    static final String OWNED_IDENTITY = "identity";
    private PrivateIdentity privateIdentity;
    static final String PRIVATE_IDENTITY = "private_identity";
    [...]
}
```

Il est donc possible d'extraire la clé via la commande suivante:

```
$ sqlite3 engine_db.sqlite "SELECT quote(private_identity) FROM owned_identity LIMIT 1;"
X'0300000118000000005A68747470733A2F2F7365727665722E6F6C7669642E696F0000BFE8EB98A77F8E73849
6AD620284FCCB746B51FF6745F512F0CADA3E2C05F2C801372F70DCD058F6380CB949B007A089148D2C00BFF256
87AEE1330069E8444C9492000000370000000002140004000002B00000000016E80000000201561DD65735CD87
C89CABF44D5AB2AB8AA8C84AB642F58E3D197FB4A82FB0845920000003700000000021201040000002B00000000
016E80000000200F98B782DCBD1B2C81EABE4BB011218181F3CF63E9C3EB6E09E5DAEC70192F34900000003C000
0000020100040000003000000000066D61636B6579000000002055A8EEEBFA8F88CBD3824788723CCD0A5285F6
69ABC91939AF8B05B5409FA754'
```

C'est la classe *io.olvid.engine.datatypes.PrivateIdentity* qui contient le matériel cryptographique.

```
public class PrivateIdentity {
    private final Identity publicIdentity;
    private final ServerAuthenticationPrivateKey serverAuthenticationPrivateKey;
    private final EncryptionPrivateKey encryptionPrivateKey;
    private final MACKey macKey;
    [...]
}
```

Il est intéressant de noter qu'Olvid n'utilise pas le mécanisme de stockage de matériel cryptographique proposé nativement par Android (*Android keystore system*). Bien qu'une telle mesure serait un réel plus pour la sécurité des données de l'application Olvid cela ne peut être relevé comme une vulnérabilité. En effet, cette mesure permet de protéger la confidentialité des clés cryptographique contre un attaquant disposant d'un accès au téléphone. Or, cette menace est en dehors de la TOE.

Exclusion des backup du téléphone

Afin d'éviter la sauvegarde automatique des clés cryptographiques sur des médias non maîtrisés, Olvid a pris soin d'utiliser la fonction *getNoBackupFilesDir* d'Android.

```
this.engine = new Engine(App.getContext().getNoBackupFilesDir(), null, [...]);
```

Le chemin associé est ensuite utilisé pour créer le fichier SQLite contenant les clés privées.

```
public class Session implements Connection {
    [...]
    private Session(String dbPath, String dbKey, boolean sessionIsForUpgradeTables) throws
SQLException {
        if (dbPath == null) {
            throw new SQLException("dbPath is null, unable to createCurrentDevice a
Session.");
        }
        this.dbPath = dbPath;
        [...]
    }
}
```

Générateur de nombres pseudo-aléatoires

Le générateur de nombre pseudo aléatoire utilisé est HMAC DRBG Based on SHA-256 publié dans **[NIST.SP.800-90Ar1]**. Ce générateur se base sur HMAC-SHA256 et repose sur les propriétés cryptographiques de celui-ci. Deux variables d'état sont présentes, K et V, chacune de 256 bits. Cela est donc conforme à *RègleArchiGDA-3* de **[RGS_B]**.

```
init(seed):
  K = array(size=32, value=0)
  V = array(size=32, value=1)
  update(seed)

update(data):
  K = HMAC(K, V || 0x00 || data)
  V = HMAC(K,V)
  if length(data) > 0:
    K = HMAC(K, V || 0x01 || data)
    V = HMAC(K,V)

bytes(N):
  output = ""
  while len(output) < N:
    V = HMAC(K,V)
    output = output || V
  update([])
  return output[0:N]

bigint(max:
  l = bitLength(max-1)
  ell = l + (l-1) / 8
  mask = (1<<(l-8*(ell-1))) - 1
  while true:
    rand = bytes(ell)
    rand[0] &= mask
    r = BigInteger(sign=1, data=rand)
    if r < max:
      retur r
```

Le générateur pseudo-aléatoire est initialisé avec une graine issue de **[SecureRandom]** (*java.security.SecureRandom*)

Conformité de l'implémentation

L'implémentation actuelle ne supporte pas ou limite les fonctionnalités suivantes (décrites dans **[NIST.SP.800-90Ar1]**):

- **Reseeding** (chapitres 8.6.8 et 10.1.2.4): le reseeding n'est pas implémenté par le générateur d'aléa (pas de compteur incrémenté) et la fonction *reseed* crée un nouveau générateur à partir de la graine plutôt que d'utiliser et

dériver son état interne actuel. Le chapitre 10.1 préconise un *reseed* au-delà de 2^{48} requêtes, non atteignable dans le cas d'utilisation actuel.

- **Uninstantiate_function** *(chapitres 9.4)*: la fonction de réinitialisation (mise à zéro de façon sécurisée de l'état du générateur) n'est pas implémentée et est déléguée au *garbage collector* de Java.

- ***EntropyInput***, ***Nonce***, et ***PersonalizationString*** (chapitre 10.1.2) ne sont pas distingués (optionnel), mais pré-générés dans une unique graine (Seed = EntropyInput || Nonce || PersonalizationString).

- La méthode ***HMAC_DRBG_Update*** est privée et n'est pas appelable hors de la classe. Les vecteurs de test contenant un *AdditionalInput* ne sont pas applicables.

Des tests unitaires Java sont présents dans le code source de l'application pour valider l'implémentation du générateur pseudo-aléatoire. Les vecteurs de test du NIST (**[DRBG_TESTVECTORS]**) ont été recompilés en une version compatible avec ces tests unitaires pour être validés. De plus une implémentation en Python a été redéveloppée pour valider une seconde fois les tests unitaires de Olvid ainsi que ceux du NIST. Les vecteurs de test du NIST utilisés sont un sous-ensemble restreint pour les raisons suivantes :

- Le fichier *drbgvectors_no_reseed/HMAC_DRBG.rsp* a été utilisé, les autres vecteurs nécessitant l'utilisation du *reseeding,* non implémentés dans l'application ;

- Dans ce fichier, les vecteurs utilisant un *AdditionalInput* (120 vecteurs recensés) sont ignorés en raison de l'inaccessibilité de *HMAC_DRBG_Update* (méthode privée) ;

- Les 120 vecteurs restants ont été testés conformes par l'implémentation disponible en annexe.

Chiffrement des pièces jointes

Afin d'être envoyées, les pièces jointes sont chiffrées via le mécanisme de chiffrement authentifié présenté en Chiffrement authentifié page 17. Une clé est spécifiquement générée pour chaque pièce jointe et est associée au message. La clé utilisée pour le chiffrement authentifié est donc elle-même chiffrée au sein du message via les mécanismes présentés dans Échange de messages page 18.

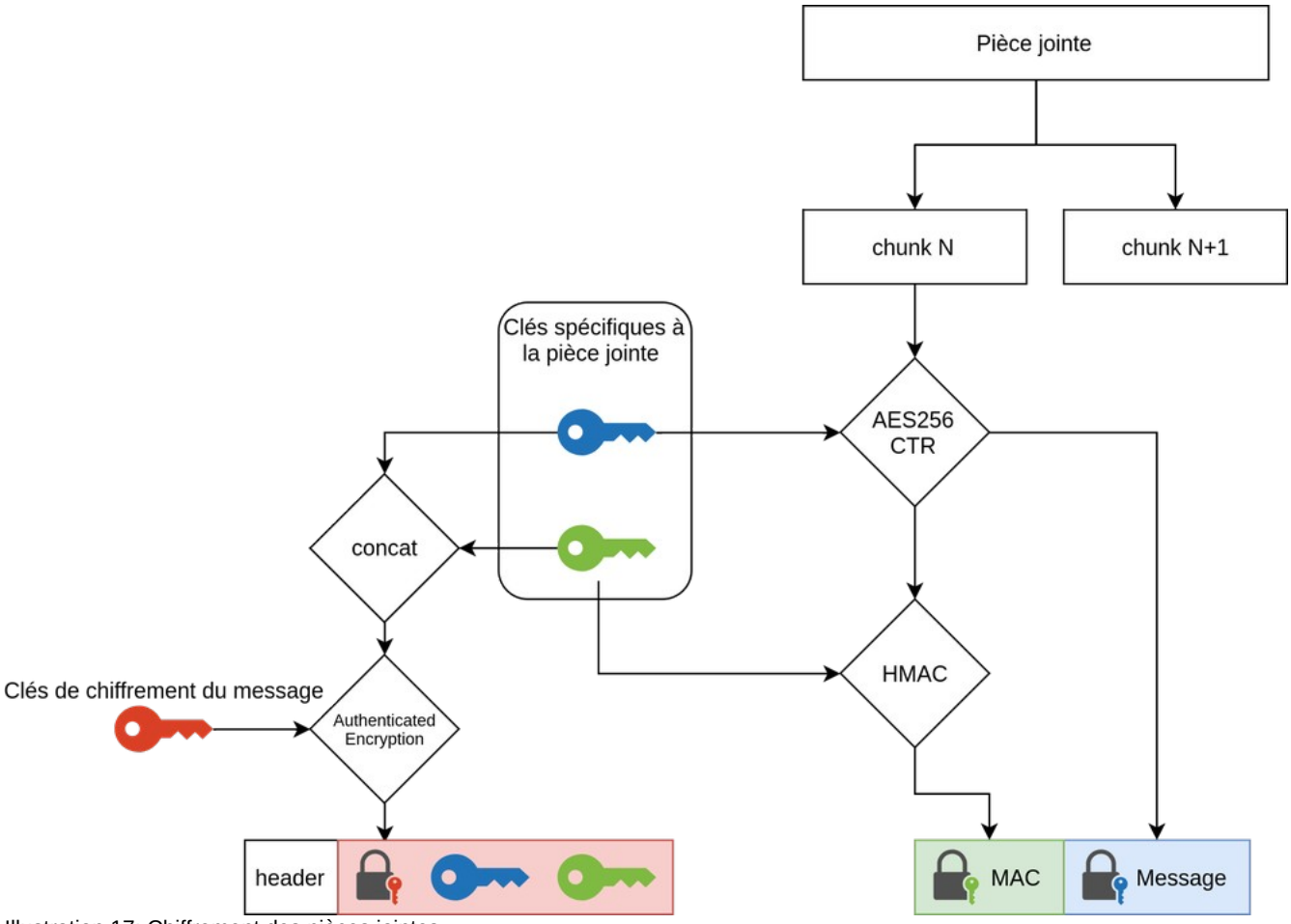

Illustration 17: Chiffrement des pièces jointes

2.1.3.3. Entretien avec les développeurs

Plusieurs entretiens par visioconférence ont eu lieu entre l'éditeur et le CESTI, au début de l'audit, puis pendant, afin d'aborder les spécifications cryptographiques et de répondre aux questions sur l'application. Il ressort de ces échanges que l'éditeur a une excellente maîtrise de son produit et des concepts cryptographiques utilisés. Il apparaît clair que les choix de design et d'implémentation d'Olvid sont le fruit de réflexions mûries et que la sécurité du produit est au centre des préoccupations.

On notera par ailleurs la bonne disponibilité de l'éditeur pour répondre aux questions techniques rencontrées lors de l'évaluation.

2.1.3.4. Avis d'expert et vulnérabilités potentielles identifiées

Une vulnérabilité potentielle a été révélée par l'étude de la spécification cryptographique ; elle est analysée plus en détail dans la section V-01 : USURPATION-TIERS page 85.

L'attaque associée à la vulnérabilité potentielle vise l'établissement d'une relation avec l'usurpation d'une identité auprès de l'un des correspondants. Cette attaque a une probabilité de réussite relativement élevée à 10^{-4} qui est inhérente au design du protocole qui ne peut garantir qu'une sûreté limitée à la quantité d'information échangée sur le canal authentique (2 fois 4 chiffres).

Cependant cette attaque requiert un investissement important d'un attaquant qui doit contrôler le serveur applicatif mais aussi le canal de communication utilisé pour la demande de mise en relation (SMS, e-mail). De plus, en cas de succès de l'attaque, il est probable que l'un des utilisateurs utilise le canal authentique pour prévenir du problème étant donné que le protocole échoue chez l'un des participants.

2.1.4. Conformité et résistance des mécanismes et fonctions

2.1.4.1. Synthèse des fonctionnalités analysées / non analysées

Le tableau de synthèse suivant liste les fonctions de sécurité indiquées dans la cible de sécurité.

Fonction	Analysée	Conformité à la cible	Conformité à l'état de l'art
FS1 : Authentification des utilisateurs	Oui	Oui	Oui
FS2 : Authentification des échanges	Oui	Oui	Oui
FS3 : Chiffrement des messages et des pièces jointes	Oui	Oui	Oui
FS4 : Chiffrement des sauvegardes du carnet de contact	Oui	Oui	Oui
FS5 : Chiffrement des appels VoIp	Oui	Oui	Oui

2.1.4.2. Détails des travaux d'analyse de la conformité des fonctions de sécurité

L'ensemble des protocoles cryptographiques présents dans l'application sont définis par des machines à états. L'analyse des protocoles prenant part aux fonctions de sécurité définies ci-après passe par une revue du code source. Il est proposé de faire dans un premier temps la revue des mécanismes génériques de machine à état puis de compléter l'analyse spécifique de chaque protocole dans les chapitres suivants.

Machine à état

Chaque protocole cryptographique est défini par une machine à état. On retrouve donc un ensemble d'états, d'étapes permettant de transiter entre les états et de messages permettant d'une part d'échanger les informations nécessaires entre les deux participants et d'autre part de représenter les interactions utilisateurs.

L'implémentation des protocoles est regroupée au sein du namespace *io.olvid.engine.protocol.protocols*. Chaque protocole est défini au sein d'une classe dédiée qui hérite de *io.olvid.engine.protocol.protocol_engine.ConcreteProtocol*:

```
public class <ProtocolName>Protocol extends ConcreteProtocol {
    [...]
}
```

Chaque état du protocole est représenté par une classe héritée de *io.olvid.engine.protocol.protocol_engine.ConcreteProtocolState*. Chaque instance existante est stockée en base de données via l'utilisation de la classe *io.olvid.engine.protocol.databases.ProtocolInstance*.

Chaque étape permettant de transiter entre deux états est représentée par une classe héritée de *io.olvid.engine.protocol.protocol_engine.ProtocolStep*. Ces étapes permettent de réaliser les opérations nécessaires à l'avancement du protocole.

Enfin, chaque message échangé entre les deux participants du protocole est représenté par une classe héritée de *io.olvid.engine.protocol.protocol_engine.ConcreteProtocolMessage*. Ces messages permettent de mettre en relation les deux participants.

La réception d'un message provoque l'exécution d'une étape du protocole via l'appel à la fonction *doExecute* de la classe *io.olvid.engine.protocol.protocol_engine.ProtocolOperation*. Il est d'abord nécessaire de trouver l'instance du protocole correspondant au message reçu.

```
protocolInstance = ProtocolInstance.get(protocolManagerSession, protocolInstanceUid,
protocolOwnedIdentity);
if (protocolInstance == null) {
    protocolInstance = ProtocolInstance.create(protocolManagerSession, protocolInstanceUid,
protocolOwnedIdentity, protocolId, new InitialProtocolState());
    if (protocolInstance != null) {
        protocol =
ConcreteProtocol.getConcreteProtocolInInitialState(protocolManagerSession, protocolId,
protocolInstanceUid, protocolOwnedIdentity, prng, jsonObjectMapper);
    }
} else {
    protocol = ConcreteProtocol.getConcreteProtocol(protocolInstance, prng,
jsonObjectMapper);
}
```

L'initialisation d'un nouveau protocole est réalisée lorsqu'aucune instance existante n'est trouvée. Le protocole est alors dans l'état *INITIAL_STATE_ID*.

Il est ensuite nécessaire de trouver l'étape correspondant à ce message. Cette opération est réalisée au sein de la fonction *getStepToExecute*.

```
public final ProtocolStep getStepToExecute(ConcreteProtocolMessage concreteProtocolMessage)
{
    try {
        int matches = 0;
        Constructor<?> constructor = null;
        Class<?>[] classes = getPossibleStepClasses(currentState.id);
        for (Class<?> clazz: classes) {
            try {
                constructor = clazz.getConstructor(currentState.getClass(),
concreteProtocolMessage.getClass(), this.getClass());
                matches++;
            } catch (NoSuchMethodException e) {}
        }
        if (matches != 1) {
            Logger.d("Found " + matches + " protocolStep to execute in " + this.getClass()
+ " for state " + currentState.getClass() + " and message " +
concreteProtocolMessage.getClass());
            return null;
        }
        return (ProtocolStep) constructor.newInstance(currentState,
concreteProtocolMessage, this);
    } catch (Exception e) {
        return null;
    }
}
```

Afin de trouver l'étape à exécuter, deux actions sont réalisées, premièrement trouver l'ensemble des étapes possible à partir de l'état courant du protocole *getPossibleStepClasses(currentState.id);*, deuxièmement trouver l'étape qui est déclenchée par le message reçu. Cette partie est réalisée en s'appuyant sur le typage fort proposé par Java ainsi que sur les fonctionnalités d'introspection (*Class.getConstructor()*).

Une fois l'étape obtenue elle peut être exécutée.

```
// run the step
Logger.d("Executing step " + stepToExecute.getClass());
OperationQueue queue = new OperationQueue();
queue.queue(stepToExecute);
queue.execute(1, "Engine-ProtocolOperation");
queue.join();
if (stepToExecute.isCancelled() || (stepToExecute.getEndState() == null)) {
    Logger.i("Step " + stepToExecute.getClass() + " failed");
    cancel(RFC_THE_STEP_TO_EXECUTE_FAILED);
    return;
}
Logger.d("Finished step " + stepToExecute.getClass() + ". It reached state " +
stepToExecute.getEndState().getClass());
```

Décodage des messages

Les messages protocolaires et applicatifs sont encodés avec un encodage de type *Type Value Length* (TLV) tel que décrit dans la partie III de **[DOC_CRYPTO]**.

L'analyse du décodage des valeurs encodées ainsi est importante puisqu'il fait partie de la surface d'attaque.

Conformément à **[DOC_CRYPTO]** les types supportés sont les suivants:

```
private static final byte BYTE_IDS_BYTE_ARRAY = (byte) 0x00;
private static final byte BYTE_IDS_INT        = (byte) 0x01;
private static final byte BYTE_IDS_BOOLEAN    = (byte) 0x02;
private static final byte BYTE_IDS_LIST       = (byte) 0x03;
private static final byte BYTE_IDS_DICTIONARY = (byte) 0x04;
private static final byte BYTE_IDS_BIG_UINT   = (byte) 0x80;
private static final byte BYTE_IDS_SYM_KEY    = (byte) 0x90;
private static final byte BYTE_IDS_PUB_KEY    = (byte) 0x91;
private static final byte BYTE_IDS_PRV_KEY    = (byte) 0x92;
```

Les données encodées au format TLV sont représentées par la classe *io.olvid.engine.encoder.Encoded*.

Il est important de noter que le décodage est réalisé au moment où la structure du message attendu est connue. Ainsi au moment du décodage, les valeurs encodées sont décodées via les méthodes adéquates. Par exemple lors de la réception d'un message protocolaire la récupération des champs correspondant à l'entête telle que décrite dans **[DOC_CRYPTO]** au chapitre 22.1 – Protocol Message Structure se fait manuellement en décodant les types attendus via les méthodes de la classe *io.olvid.engine.encoder.Encoded*.

```
public static GenericReceivedProtocolMessage of(ProtocolReceivedMessage
protocolReceivedMessage, Identity associatedOwnedIdentity) {
    try {
        Encoded[] listOfEncoded =
protocolReceivedMessage.getEncodedElements().decodeList();
        if (listOfEncoded.length != 4) {
            return null;
        }
        int protocolId = (int) listOfEncoded[0].decodeLong();
        UID protocolInstanceUid = listOfEncoded[1].decodeUid();
```

```
        int protocolMessageId = (int) listOfEncoded[2].decodeLong();
        Encoded[] inputs = listOfEncoded[3].decodeList();
        return new GenericReceivedProtocolMessage(
                protocolReceivedMessage.getOwnedIdentity(),
                inputs,
                null,
                null,
                protocolInstanceUid,
                protocolMessageId,
                protocolId,
                protocolReceivedMessage.getReceptionChannelInfo(),
                associatedOwnedIdentity,
                protocolReceivedMessage.getServerTimestamp());
    } catch (DecodingException e) {
        return null;
    }
}
```

Une vérification de conformité de type est effectuée pour chaque type supporté nativement au sein des méthodes de la classe *io.olvid.engine.encoder.Encoded*.

```
public byte[] decodeBytes() throws DecodingException {
       if (data[0] != BYTE_IDS_BYTE_ARRAY) {
           throw new DecodingException();
       }
       [...]
   }
```

Une fois l'entête du message protocolaire décodé au sein d'un *io.olvid.engine.protocol.datatypes.GenericReceivedProtocolMessage* il est utilisé pour faire avancer la machine à état du protocole visé. Le type de message attendu est représenté par des classes dérivant de *io.olvid.engine.protocol.protocol_engine.ConcreteProtocolMessage*. Le contenu du message est alors décodé dans le constructeur de chacune de ces classes. Par exemple, lors de la réception d'un message *Ping* au sein du protocole *ChannelCreationWithContactDeviceProtocol*.

```
public PingMessage(ReceivedMessage receivedMessage) throws Exception {
    super(new CoreProtocolMessage(receivedMessage));
    if (receivedMessage.getInputs().length != 3) {
        throw new Exception();
    }
    this.contactIdentity = receivedMessage.getInputs()[0].decodeIdentity();
    this.contactDeviceUid = receivedMessage.getInputs()[1].decodeUid();
    this.signature = receivedMessage.getInputs()[2].decodeBytes();
}
```

La revue systématique des différents messages et des types dérivant de *ConcreteProtocolMessage* ne montre aucun problème d'implémentation.

Primitives cryptographiques

Chiffrement asymétrique

Le chiffrement asymétrique est réalisé par l'interface *PublicKeyEncryption*. Il existe deux implémentations pour cette interface :

- *io.olvid.engine.crypto.PublicKeyEncryptionEciesCurve25519*
- *io.olvid.engine.crypto.PublicKeyEncryptionEciesMDC*

L'implémentation est choisie en fonction de la clé fournie par l'utilisateur contacté. La génération d'une clé lors de l'initialisation de l'application est présentée dans Génération des clés page 17. Le chiffrement par défaut est fourni par *io.olvid.engine.crypto.Suite#getDefaultEncryptionAlgoImplByte* et correspond à ECIES avec Curve25519.

Le chiffrement réalisé dans la fonction *internalEncrypt* de *io.olvid.engine.crypto.PublicKeyEncryptionEciesCurve25519* est conforme à **[DOC_CRYPTO]**.

```
CiphertextAndKey internalEncrypt(EncryptionPublicKey publicKey, PRNGService prng) {
    BigInteger Ay = ((EncryptionEciesPublicKey) publicKey).getAy();
    int l = curve.byteLength;
    BigInteger r;
    do {
        r = prng.bigInt(curve.q);
    } while (r.equals(BigInteger.ZERO));
    BigInteger Gy = curve.G.getY();
    BigInteger By = curve.scalarMultiplication(r, Gy);
    BigInteger Dy = curve.scalarMultiplication(r, Ay);
    try {
        byte[] ciphertext = Encoded.bytesFromBigUInt(By, l);
        byte[] seedBytes = new byte[2 * l];
        System.arraycopy(ciphertext, 0, seedBytes, 0, l);
        System.arraycopy(Encoded.bytesFromBigUInt(Dy, l), 0, seedBytes, l, l);
        AuthEncKey key = (AuthEncKey) new KDFSha256().gen(new Seed(seedBytes), new
KDFDelegateForAuthEncAES256ThenSHA256())[0];
        return new CiphertextAndKey(key, new EncryptedBytes(ciphertext));
    } catch (EncodingException e) {
        return null;
    }
}
```

Chiffrement authentifié

Conformément à **[DOC_CRYPTO]** le chiffrement authentifié est réalisé via CTR AES256 avec HMAC SHA256.

```
//io.olvid.engine.crypto.AuthEncAES256ThenSHA256#encrypt
public EncryptedBytes encrypt(AuthEncKey key, byte[] plaintext, PRNG prng) throws
InvalidKeyException {
    if (! (key instanceof AuthEncAES256ThenSHA256Key)) {
        throw new InvalidKeyException();
    }
    MACHmacSha256Key macKey = ((AuthEncAES256ThenSHA256Key) key).getMacKey();
    SymEncCTRAES256Key encKey = ((AuthEncAES256ThenSHA256Key) key).getEncKey();
    MACHmacSha256 mac = new MACHmacSha256();
    SymEncCtrAES256 enc = new SymEncCtrAES256(encKey);
    byte[] ciphertext = new byte[ciphertextLengthFromPlaintextLength(plaintext.length)];
    byte[] iv = prng.bytes(SymEncCtrAES256.IV_BYTE_LENGTH);
    EncryptedBytes encrypted = enc.encrypt(iv, plaintext);
    byte[] encryptedBytes = encrypted.getBytes();
    System.arraycopy(encryptedBytes, 0, ciphertext, 0, encryptedBytes.length);
    byte[] hash = mac.digest(macKey, encryptedBytes);
    System.arraycopy(hash, 0, ciphertext,
enc.ciphertextLengthFromPlaintextLength(plaintext.length), hash.length);
    return new EncryptedBytes(ciphertext);
}
```

Le chiffrement CTR AES256 est réalisé par *io.olvid.engine.crypto.SymEncCtrAES256#encrypt*.

```
public EncryptedBytes encrypt(byte[] iv, byte[] plaintext) throws InvalidKeyException {
```

```
    if (iv.length != IV_BYTE_LENGTH) {
        throw new InvalidKeyException();
    }
    byte[] ciphertext = new byte[ciphertextLengthFromPlaintextLength(plaintext.length)];
    System.arraycopy(iv, 0, ciphertext, 0, IV_BYTE_LENGTH);
    byte[] fullIV = new byte[AES_BLOCK_BYTE_LENGTH];
    System.arraycopy(iv, 0, fullIV, 0, IV_BYTE_LENGTH);
    try {
        aes.init(Cipher.ENCRYPT_MODE, new SecretKeySpec(key.getKeyBytes(), "AES"), new
IvParameterSpec(fullIV));
        byte[] output = aes.doFinal(plaintext);
        System.arraycopy(output, 0, ciphertext, IV_BYTE_LENGTH, output.length);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return new EncryptedBytes(ciphertext);
}
```

Le chiffrement AES est réalisé par la librairie *javax.crypto* fournie par Android, considérée comme conforme à sa spécification ainsi qu'aux règles et recommandations du RGS selon l'hypothèse *H.BouncyCastle* introduite dans la section Problème de sécurité et environnement page 7.

Key encapsulation mechanism (KEM)

Le mécanisme d'encapsulation de clé utilise le schéma **ECIES256-KEM512** basé sur la courbe **Curve25519**.

L'application implémente le **KEM** dans *io.olvid.engine.crypto.KemEcies256Kem512#internalEncrypt*. L'implémentation est conforme à **[DOC_CRYPTO]**. Cette fonction génère la clé publique éphémère via le PRNG fourni en paramètres puis calcule un secret partagé et en dérive les clés de session AES et MAC. La fonction renvoie un objet **CiphertextAndKey** contenant ces éléments.

```
CiphertextAndKey internalEncrypt(EncryptionPublicKey publicKey, PRNGService prng) {
    BigInteger Ay = ((EncryptionEciesPublicKey) publicKey).getAy();
    int l = curve.byteLength;
    BigInteger r;
    do {
        r = prng.bigInt(curve.q);
    } while (r.equals(BigInteger.ZERO));
    BigInteger Gy = curve.G.getY();
    BigInteger By = curve.scalarMultiplication(r, Gy);
    BigInteger Dy = curve.scalarMultiplication(r, Ay);
    try {
        byte[] ciphertext = Encoded.bytesFromBigUInt(By, l);
        byte[] seedBytes = new byte[2 * l];
        System.arraycopy(ciphertext, 0, seedBytes, 0, l);
        System.arraycopy(Encoded.bytesFromBigUInt(Dy, l), 0, seedBytes, l, l);
        AuthEncKey key = (AuthEncKey) new KDFSha256().gen(new Seed(seedBytes), new
KDFDelegateForAuthEncAES256ThenSHA256())[0];
        return new CiphertextAndKey(key, new EncryptedBytes(ciphertext));
    } catch (EncodingException e) {
        return null;
    }
}
```

Le déchiffrement des clés se fait par *io.olvid.engine.crypto.KemEcies256Kem512#internalDecrypt*. Cette implémentation est conforme à **[DOC_CRYPTO]**. À partir de la clé publique éphémère (**c**) et la clé privée de l'utilisateur, Olvid recalcule le secret partagé et dérive les clés de session, renvoyées dans un objet **AuthEncKey** :

```
AuthEncKey internalDecrypt(EncryptionPrivateKey privateKey, byte[] c) {
    BigInteger a = ((EncryptionEciesPrivateKey) privateKey).getA();
    int l = curve.byteLength;
    if (c.length != l) {
        return null;
    }
    BigInteger By = Encoded.bigUIntFromBytes(c);
    By = curve.scalarMultiplication(curve.nu, By);
    if (By.equals(BigInteger.ONE)) {
        return null;
    }
    a = a.multiply(curve.nu.modInverse(curve.q)).mod(curve.q);
    BigInteger Dy = curve.scalarMultiplication(a, By);
    try {
        byte[] seedBytes = new byte[2 * l];
        System.arraycopy(c, 0, seedBytes, 0, l);
        System.arraycopy(Encoded.bytesFromBigUInt(Dy, l), 0, seedBytes, l, l);
        AuthEncKey k = (AuthEncKey) new KDFSha256().gen(new Seed(seedBytes), new
KDFDelegateForAuthEncAES256ThenSHA256())[0];
        return k;
    } catch (EncodingException e) {
        return null;
    }
}
```

Data Encapsulation Mecanism (DEM)

Le chiffrement des données est fait en AES256-CTR:

- Les clés de session AES et MAC sont calculées dans le KEM ;
- un nonce de 64bits est tiré du PRNG et étendu à 128bits (concaténation de zéro) pour former l'IV ;
- Les données sont chiffrées en AES256-CTR puis un MAC est calculé à partir du chiffré et de la clé MAC précédemment générée ;
- La clé publique éphémère, le nonce, le chiffré et le mac sont écrits dans le fichier final.

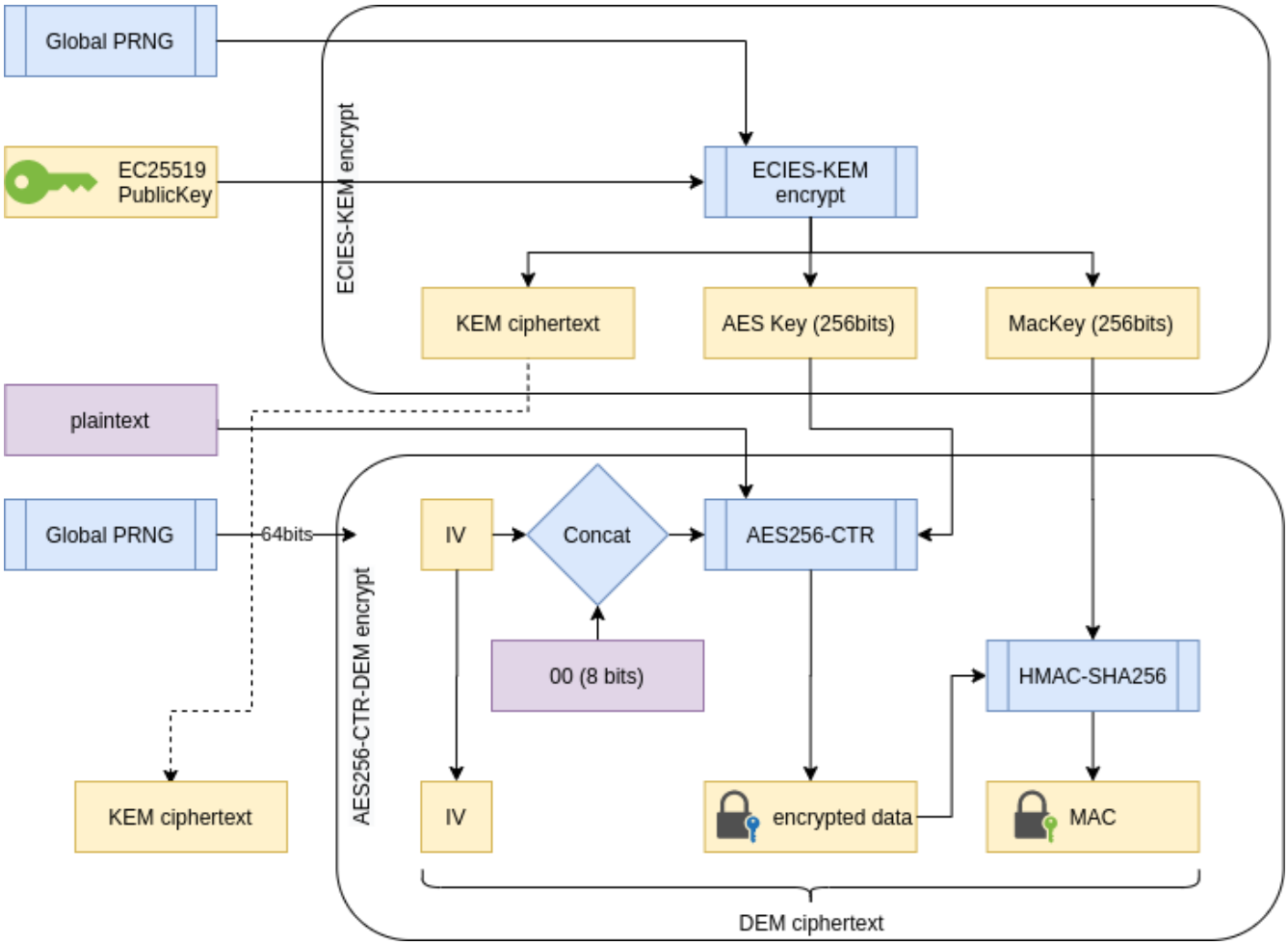

Illustration 18: Chiffrement des données

Le déchiffrement se fait de façon similaire :

- Les clés de session AES et MAC sont calculées dans le KEM à partir de la clé publique éphémère présente dans le fichier ;
- Le MAC du chiffré est vérifié ;
- Le nonce est étendu puis utilisé comme IV avec la clé de session pour déchiffrer la donnée en AES256-CTR.

L'application implémente le DEM dans **AuthEncAES256ThenSHA256**.

Olvid n'implémente pas le chiffrement AES256-CTR ni HMAC-SHA256 mais utilise l'API cryptographique Java sous Android:

- *javax.crypto.Cipher.getInstance("AES/CTR/NoPadding")* pour AES256-CTR;
- *javax.crypto.Mac.getInstance("HmacSHA256")* pour HMAC-SHA256.

L'implémentation de ces deux primitives respecte **[DOC_CRYPTO]**.

FS1 : Authentification des utilisateurs

FS1 assure que deux utilisateurs disposant d'un canal authentique tiers peuvent se mettre en relation dans Olvid de façon authentique. FS1 garantit qu'un attaquant contrôlant totalement les serveurs d'Olvid n'est pas en mesure d'usurper l'identité d'un utilisateur.

L'architecture de cette fonction de sécurité est documentée dans le chapitre *Trust Establishment Protocol with SAS* de **[DOC_CRYPTO]**.

Méthodologie

Pour vérifier que l'application implémente correctement le protocole décrit dans **[DOC_CRYPTO]** et s'assurer que le design ne présente pas de faiblesse non identifiée dans Authentification des utilisateurs page 16 l'analyse est menée en suivant deux approches complémentaires :

- Par l'étude du code source des fonctions de gestion du protocole cryptographique ;
- Par l'implémentation d'un serveur Olvid, permettant d'extraire l'ensemble des messages échangés lors de la réalisation du protocole.

L'objectif est de contrôler que l'implémentation est bien conforme à **[DOC_CRYPTO]** et de vérifier qu'un attaquant contrôlant le serveur ne voit passer aucune information sensible lors de la réalisation du protocole.

L'ensemble des scripts issus de ces travaux est disponible dans Annexes page 91.

Analyse statique

Le protocole *Trust Establishment Protocol with SAS* est implémenté sous forme de machine à état. L'implémentation des machines à état a été revue dans Machine à état page 26.

Le protocole *Trust Establishment Protocol with SAS* est implémenté dans *io.olvid.engine.protocol.protocols.TrustEstablishmentWithSasProtocol*.

Conformément à **[DOC_CRYPTO]** on retrouve les états suivants :

```
public static class WaitingForSeedState extends ConcreteProtocolState { ... }
public static class WaitingForConfirmationState extends ConcreteProtocolState { ... }
public static class CancelledState extends ConcreteProtocolState { ... }
public static class WaitingForDecommitmentState extends ConcreteProtocolState { ... }
public static class WaitingForUserSasState extends ConcreteProtocolState { ... }
public static class ContactIdentityTrustedLegacyState extends ConcreteProtocolState { ... }
public static class MutualTrustConfirmedState extends ConcreteProtocolState { ... }
public static class ContactSasCheckedState extends ConcreteProtocolState { ... }
```

Conformément à **[DOC_CRYPTO]** on retrouve les étapes suivantes :

```
public static class SendCommitmentStep extends ProtocolStep { ... }
public static class StoreDecommitmentStep extends ProtocolStep { ... }
public static class StoreAndPropagateCommitmentAndAskForConfirmationStep extends
ProtocolStep { ... }
public static class StoreCommitmentAndAskForConfirmationStep extends ProtocolStep { ... }
public static class SendSeedAndPropagateConfirmationStep extends ProtocolStep { ... }
public static class ReceivedConfirmationFromOtherDeviceStep extends ProtocolStep { ... }
public static class ShowSasDialogAndSendDecommitmentStep extends ProtocolStep { ... }
public static class ShowSasDialogStep extends ProtocolStep { ... }
public static class CheckSasStep extends ProtocolStep { ... }
public static class CheckPropagatedSasStep extends ProtocolStep { ... }
public static class NotifiedMutualTrustEstablishedLegacyStep extends ProtocolStep { ... }
public static class AddTrustStep extends ProtocolStep { ... }
```

Conformément à **[DOC_CRYPTO]** on retrouve les messages suivants :

```
public static class InitialMessage extends ConcreteProtocolMessage { ... }
public static class SendCommitmentMessage extends ConcreteProtocolMessage { ... }
public static class PropagateInvitationToAliceDevicesMessage extends
ConcreteProtocolMessage { ... }
public static class PropagateCommitmentToBobDevicesMessage extends ConcreteProtocolMessage
{ ... }
public static class BobDialogInvitationConfirmationMessage extends ConcreteProtocolMessage
{ ... }
public static class PropagateConfirmationToBobDevicesMessage extends
ConcreteProtocolMessage { ... }
public static class SendBobSeedMessage extends ConcreteProtocolMessage { ... }
public static class SendDecommitmentMessage extends ConcreteProtocolMessage { ... }
public static class DialogForSasExchangeMessage extends ConcreteProtocolMessage { ... }
public static class PropagateEnteredSasToOtherDevicesMessage extends
ConcreteProtocolMessage { ... }
public static class MutualTrustConfirmationMessage extends ConcreteProtocolMessage { ... }
```

Les différentes étapes définies dans le cas du protocole *Trust Establishment Protocol with SAS* ont les constructeurs suivants :

```
public SendCommitmentStep(InitialProtocolState startState, InitialMessage receivedMessage,
TrustEstablishmentWithSasProtocol protocol) throws Exception { ... }
public StoreDecommitmentStep(InitialProtocolState startState,
PropagateInvitationToAliceDevicesMessage receivedMessage, TrustEstablishmentWithSasProtocol
protocol) throws Exception { ... }
public StoreAndPropagateCommitmentAndAskForConfirmationStep(InitialProtocolState
startState, SendCommitmentMessage receivedMessage, TrustEstablishmentWithSasProtocol
protocol) throws Exception { ... }
public StoreCommitmentAndAskForConfirmationStep(InitialProtocolState startState,
PropagateCommitmentToBobDevicesMessage receivedMessage, TrustEstablishmentWithSasProtocol
protocol) throws Exception { ... }
public SendSeedAndPropagateConfirmationStep(WaitingForConfirmationState startState,
BobDialogInvitationConfirmationMessage receivedMessage, TrustEstablishmentWithSasProtocol
protocol) throws Exception { ... }
public ReceivedConfirmationFromOtherDeviceStep(WaitingForConfirmationState startState,
PropagateConfirmationToBobDevicesMessage receivedMessage, TrustEstablishmentWithSasProtocol
protocol) throws Exception { ... }
public ShowSasDialogAndSendDecommitmentStep(WaitingForSeedState startState,
SendBobSeedMessage receivedMessage, TrustEstablishmentWithSasProtocol protocol) throws
Exception { ... }
public ShowSasDialogStep(WaitingForDecommitmentState startState, SendDecommitmentMessage
receivedMessage, TrustEstablishmentWithSasProtocol protocol) throws Exception { ... }
public CheckSasStep(WaitingForUserSasState startState, DialogForSasExchangeMessage
receivedMessage, TrustEstablishmentWithSasProtocol protocol) throws Exception { ... }
public CheckPropagatedSasStep(WaitingForUserSasState startState,
PropagateEnteredSasToOtherDevicesMessage receivedMessage, TrustEstablishmentWithSasProtocol
protocol) throws Exception { ... }
public NotifiedMutualTrustEstablishedLegacyStep(ContactIdentityTrustedLegacyState
startState, MutualTrustConfirmationMessage receivedMessage,
TrustEstablishmentWithSasProtocol protocol) throws Exception { ... }
public AddTrustStep(ContactSasCheckedState startState, MutualTrustConfirmationMessage
receivedMessage, TrustEstablishmentWithSasProtocol protocol) throws Exception { ... }
```

Le passage d'un état à l'autre est réalisé au sein des différents *Steps*. L'état résultant de l'exécution d'une étape est retourné par la fonction *executeStep* de l'étape.

On peut donc suivre l'enchaînement de l'ensemble des états possibles du protocole.

```
SendCommitmentStep -> WaitingForSeedState
StoreDecommitmentStep -> WaitingForSeedState
StoreAndPropagateCommitmentAndAskForConfirmationStep -> WaitingForConfirmationState
StoreCommitmentAndAskForConfirmationStep -> WaitingForConfirmationState
SendSeedAndPropagateConfirmationStep -> CancelledState, WaitingForDecommitmentState
ReceivedConfirmationFromOtherDeviceStep -> CancelledState, WaitingForDecommitmentState
ShowSasDialogAndSendDecommitmentStep -> WaitingForUserSasState
ShowSasDialogStep -> WaitingForUserSasState
CheckSasStep -> WaitingForUserSasState, ContactSasCheckedState
CheckPropagatedSasStep -> CancelledState, ContactSasCheckedState
NotifiedMutualTrustEstablishedLegacyStep -> MutualTrustConfirmedState
AddTrustStep -> MutualTrustConfirmedState
```

Conformément à **[DOC_CRYPTO]** la première étape du protocole *Trust Establishment Protocol with SAS* consiste pour Alice à envoyer un *commitment* composé d'un *seed* aléatoire, de son identité et d'un nonce.

```
public ConcreteProtocolState executeStep() throws Exception {
    [...]
    Seed seedAliceForSas = new Seed(getPrng());
    Commitment commitmentScheme = Suite.getDefaultCommitment(0);
    Commitment.CommitmentOutput commitmentOutput = commitmentScheme.commit(
            getOwnedIdentity().getBytes(),
            seedAliceForSas.getBytes(),
            getPrng()
    );
    [...]
    // Broadcast commitment to Bob
    CoreProtocolMessage coreProtocolMessage =
buildCoreProtocolMessage(SendChannelInfo.createAsymmetricBroadcastChannelInfo(receivedMessa
ge.contactIdentity, getOwnedIdentity()));
    ChannelMessageToSend messageToSend = new SendCommitmentMessage(coreProtocolMessage,
getOwnedIdentity(), receivedMessage.ownSerializedDetails, ownedDeviceUids,
commitmentOutput.commitment).generateChannelProtocolMessageToSend();
    protocolManagerSession.channelDelegate.post(protocolManagerSession.session,
messageToSend, getPrng());
    [...]
}
```

La génération du *seed* via le PRNG à été revue dans Générateur de nombres pseudo-aléatoires page 23. La génération du *commitment* est réalisée conformément à **[DOC_CRYPTO]** par *commitmentScheme.commit*.

```
public CommitmentOutput commit(byte[] tag, byte[] value, PRNGService prng) {
    HashSHA256 h = new HashSHA256();
    byte[] e = prng.bytes(COMMITMENT_RANDOM_LENGTH);
    byte[] decommitment = new byte[value.length + COMMITMENT_RANDOM_LENGTH];
    System.arraycopy(value, 0, decommitment, 0, value.length);
    System.arraycopy(e, 0, decommitment, value.length, COMMITMENT_RANDOM_LENGTH);
    byte[] input = new byte[tag.length + value.length + COMMITMENT_RANDOM_LENGTH];
    System.arraycopy(tag, 0, input, 0, tag.length);
    System.arraycopy(decommitment, 0, input, tag.length, decommitment.length);
    byte[] commitment = h.digest(input);
    return new CommitmentOutput(commitment, decommitment);
}
```

L'étape suivante du protocole consiste pour Bob à envoyer un *seed* pour le calcul du SAS final.

```
public ConcreteProtocolState executeStep() throws Exception {
    [...]
```

```
    Seed seedBobForSas =
protocolManagerSession.identityDelegate.getDeterministicSeedForOwnedIdentity(getOwnedIdenti
ty(), startState.commitment);
    UID[] ownedDeviceUids =
protocolManagerSession.identityDelegate.getDeviceUidsOfOwnedIdentity(protocolManagerSession
.session, getOwnedIdentity());
    String ownSerializedDetails =
protocolManagerSession.identityDelegate.getSerializedPublishedDetailsOfOwnedIdentity(protoc
olManagerSession.session, getOwnedIdentity());
    {
        // send the seed to Alice
        CoreProtocolMessage coreProtocolMessage =
buildCoreProtocolMessage(SendChannelInfo.createAsymmetricChannelInfo(startState.contactIden
tity, getOwnedIdentity(), startState.contactDeviceUids));
        ChannelMessageToSend messageToSend = new SendBobSeedMessage(coreProtocolMessage,
seedBobForSas, ownedDeviceUids,
ownSerializedDetails).generateChannelProtocolMessageToSend();
        protocolManagerSession.channelDelegate.post(protocolManagerSession.session,
messageToSend, getPrng());
    }
    [...]
}
```

La génération du *seed* est réalisée de manière déterministe. Ce comportement est prévu et analysé dans **[DOC_CRYPTO]** au chapitre 24.2. Afin d'éviter le rejeux de commitment une protection est mise en place lors de la réception du commitment, dans l'étape *StoreAndPropagateCommitmentAndAskForConfirmationStep*.

```
public static class StoreAndPropagateCommitmentAndAskForConfirmationStep extends
ProtocolStep {
    [...]
    public ConcreteProtocolState executeStep() throws Exception {
        ProtocolManagerSession protocolManagerSession = getProtocolManagerSession();

        if (TrustEstablishmentCommitmentReceived.exists(protocolManagerSession,
getOwnedIdentity(), receivedMessage.commitment)) {
            // we already received this commitment
            return new CancelledState();
        } else {
            // store the commitment to prevent future replay
            TrustEstablishmentCommitmentReceived.create(protocolManagerSession,
getOwnedIdentity(), receivedMessage.commitment);
        }
        [...]
    }
}
```

L'étape suivante consiste pour Alice à envoyer un *decommitment* et à calculer la valeur finale du SAS.

```
public ConcreteProtocolState executeStep() throws Exception {
    [...]
    {
        // send decommitment to Bob's devices
        CoreProtocolMessage coreProtocolMessage =
buildCoreProtocolMessage(SendChannelInfo.createAsymmetricChannelInfo(startState.contactIden
tity, getOwnedIdentity(), receivedMessage.contactDeviceUids));
        ChannelMessageToSend messageToSend = new
SendDecommitmentMessage(coreProtocolMessage,
startState.decommitment).generateChannelProtocolMessageToSend();
        protocolManagerSession.channelDelegate.post(protocolManagerSession.session,
messageToSend, getPrng());
```

```
    }
    byte[] fullSas = SAS.computeDouble(startState.seedAliceForSas,
receivedMessage.seedBobForSas, startState.contactIdentity,
Constants.DEFAULT_NUMBER_OF_DIGITS_FOR_SAS);
    byte[] sasToDisplay = Arrays.copyOfRange(fullSas, 0,
Constants.DEFAULT_NUMBER_OF_DIGITS_FOR_SAS);
    [...]
}
```

Lorsque Bob reçoit le *decommitment* il calcule lui aussi la valeur du SAS.

```
public ConcreteProtocolState executeStep() throws Exception {
    [...]
    Commitment commitmentScheme = Suite.getDefaultCommitment(0);
    byte[] opened = commitmentScheme.open(startState.contactIdentity.getBytes(),
startState.commitment, receivedMessage.decommitment);
    if (opened == null) {
        Logger.e("Unable to open commitment.");
        return null;
    }
    Seed contactSeedForSas = new Seed(opened);
    byte[] fullSas = SAS.computeDouble(contactSeedForSas, startState.seedBobForSas,
getOwnedIdentity(), Constants.DEFAULT_NUMBER_OF_DIGITS_FOR_SAS);
    byte[] sasToDisplay = Arrays.copyOfRange(fullSas,
Constants.DEFAULT_NUMBER_OF_DIGITS_FOR_SAS, 2 *
Constants.DEFAULT_NUMBER_OF_DIGITS_FOR_SAS);
    [...]
}
```

L'ouverture du *commitment* procède correctement à la vérification de la valeur *commitée*.

```
public byte[] open(byte[] tag, byte[] commitment, byte[] decommitment) {
    HashSHA256 h = new HashSHA256();
    byte[] input = new byte[tag.length + decommitment.length];
    System.arraycopy(tag, 0, input, 0, tag.length);
    System.arraycopy(decommitment, 0, input, tag.length, decommitment.length);
    byte[] commitment2 = h.digest(input);
    if (Arrays.equals(commitment, commitment2)) {
        return Arrays.copyOfRange(decommitment, 0, decommitment.length -
CommitmentWithSHA256.COMMITMENT_RANDOM_LENGTH);
    } else {
        return null;
    }
}
```

Le calcul du SAS est réalisé conformément à **[DOC_CRYPTO]** par la fonction *SAS.computeDouble*.

```
 public static byte[] computeDouble(Seed seedAlice, Seed seedBob, Identity identityBob, int
numberOfDigits) {
    Hash sha256 = Suite.getHash(Hash.SHA256);
    byte[] bytesIdentity = identityBob.getBytes();
    byte[] toHash = new byte[bytesIdentity.length + seedAlice.length];
    System.arraycopy(bytesIdentity, 0, toHash, 0, bytesIdentity.length);
    System.arraycopy(seedAlice.getBytes(), 0, toHash, bytesIdentity.length,
seedAlice.length);
    byte[] hash = sha256.digest(toHash);
    byte[] xor = new byte[Math.min(hash.length, seedBob.length)];
    for (int i=0; i<xor.length; i++) {
        xor[i] = (byte) (seedBob.getBytes()[i] ^ hash[i]);
```

```
    }
    Seed seed = new Seed(xor);
    PRNG prng = Suite.getPRNG(PRNG.PRNG_HMAC_SHA256, seed);
    BigInteger max = BigInteger.valueOf(10).pow(2*numberOfDigits);
    BigInteger sas = prng.bigInt(max).add(max); // We add max to the sas to be able to get
the 0 padding by simply removing the first character of the String
    return sas.toString(10).substring(1).getBytes(StandardCharsets.UTF_8);
}
```

Enfin, lorsque les deux parties du SAS ont été échangées via un canal authentique, la dernière étape consiste à vérifier la validité du SAS et à envoyer un message de confirmation.

```
public ConcreteProtocolState executeStep() throws Exception {
    [...]
    byte[] sasToDisplay;
    byte[] computedSas;
    if (startState.isAlice) {
        byte[] fullSas = SAS.computeDouble(startState.seedForSas,
startState.contactSeedForSas, startState.contactIdentity,
Constants.DEFAULT_NUMBER_OF_DIGITS_FOR_SAS);
        sasToDisplay = Arrays.copyOfRange(fullSas, 0,
Constants.DEFAULT_NUMBER_OF_DIGITS_FOR_SAS);
        computedSas = Arrays.copyOfRange(fullSas,
Constants.DEFAULT_NUMBER_OF_DIGITS_FOR_SAS, 2 *
Constants.DEFAULT_NUMBER_OF_DIGITS_FOR_SAS);
    } else {
        byte[] fullSas = SAS.computeDouble(startState.contactSeedForSas,
startState.seedForSas, getOwnedIdentity(), Constants.DEFAULT_NUMBER_OF_DIGITS_FOR_SAS);
        sasToDisplay = Arrays.copyOfRange(fullSas,
Constants.DEFAULT_NUMBER_OF_DIGITS_FOR_SAS, 2 *
Constants.DEFAULT_NUMBER_OF_DIGITS_FOR_SAS);
        computedSas = Arrays.copyOfRange(fullSas, 0,
Constants.DEFAULT_NUMBER_OF_DIGITS_FOR_SAS);
    }
    if (! Arrays.equals(computedSas, receivedMessage.sasEnteredByUser)) {
        Logger.e("The propagated SAS does not match the computed SAS.");
        // remove the dialog
        CoreProtocolMessage coreProtocolMessage =
buildCoreProtocolMessage(SendChannelInfo.createUserInterfaceChannelInfo(getOwnedIdentity(),
DialogType.createDeleteDialog(), startState.dialogUuid));
        ChannelMessageToSend messageToSend = new
OneWayDialogProtocolMessage(coreProtocolMessage).generateChannelDialogMessageToSend();
        protocolManagerSession.channelDelegate.post(protocolManagerSession.session,
messageToSend, getPrng());
        return new CancelledState();
    }
    [...]
    {
        // display the sas confirmed dialog
        CoreProtocolMessage coreProtocolMessage =
buildCoreProtocolMessage(SendChannelInfo.createUserInterfaceChannelInfo(getOwnedIdentity(),
DialogType.createSasConfirmedDialog(startState.contactSerializedDetails,
startState.contactIdentity, sasToDisplay, receivedMessage.sasEnteredByUser),
startState.dialogUuid));
        ChannelMessageToSend messageToSend = new
OneWayDialogProtocolMessage(coreProtocolMessage).generateChannelDialogMessageToSend();
        protocolManagerSession.channelDelegate.post(protocolManagerSession.session,
messageToSend, getPrng());
    }
```

```
    return new ContactSasCheckedState(startState.contactSerializedDetails,
startState.contactIdentity, startState.dialogUuid, startState.contactDeviceUids);
}
```

La revue systématique du code gérant le protocole d'authentification des utilisateurs ne montre aucun problème particulier.

Analyse dynamique

L'implémentation d'un serveur Olvid permet d'intercepter les messages échangés lors de la réalisation du protocole *Trust Establishment Protocol with SAS*. Ce protocole chiffre l'ensemble de ses messages en utilisant la cryptographie asymétrique. Il est donc possible d'extraire les clés privées du téléphone des deux participants et de les utiliser pour déchiffrer l'ensemble des messages échangés. Il est proposé ci-après de présenter les messages échangés afin de confirmer qu'ils correspondent bien aux messages attendus et décrits dans **[DOC_CRYPTO]**.

Le premier message correspond à l'envoi du *commitment* par Alice.

```
Protocol Message
  Trust Establishment with SAS
    Protocol UID: db80e14a979ef133a2a71358da20dc31f4d8e67d42fbd18064b6f8d9451477fb
    Alice sends commitment
      Contact identity:
68747470733a2f2f7365727665722e6f6c7669642e696f0000841cc9dd927ebf9640f2ecea5a761e4786903752e
9a5520df98e711546a2a988016a9ea036a2967313eee899a350924e574c55763cdab82e9dda794608ac3c054c
      Encoded Contact identity: 7b2266697273745f6e616d65223a22426f62227d
      Contact device UIDs:
[19d2dae8233e53c8a41c96f310a8941bae75b4d650645f53f906c0fc74226f01]
      Commitment:
25d5a3aa7317337a1576adef32a37ae9756d4a28ad67843cd73df12ec0b6db18
```

Il est suivi par un message de Bob envoyant son *seed*.

```
 Protocol Message
  Trust Establishment with SAS
    Protocol UID: db80e14a979ef133a2a71358da20dc31f4d8e67d42fbd18064b6f8d9451477fb
    Bob sends seed
      seedBobForSas:
72dd7782947e2a2c5b8741f5f71a4667e283c8728be02f800c487e196044bfaa
      Contact device UIDs:
[238f881d0634e1d66593ce6f860eadd28c2a5b6946dc1a590648bc8bbf3186e2]
      Encoded Contact identity: 7b2266697273745f6e616d65223a22416c696365227d
```

Vient ensuite le *decommitment* d'Alice.

```
 Protocol Message
  Trust Establishment with SAS
    Protocol UID: db80e14a979ef133a2a71358da20dc31f4d8e67d42fbd18064b6f8d9451477fb
    Alice sends decommitment
      decommitment:
4385007147492501a172af2501bd95d37d98c918996771a4896838baf58c7206aa7d5a15f6345091e8533600993
23b8194dc43d85c5555e0b22de559b3ba2661
```

Enfin, après validation des deux parties du SAS, vient l'échange de deux messages de confirmation.

```
 To:
68747470733a2f2f7365727665722e6f6c7669642e696f0000841cc9dd927ebf9640f2ecea5a761e4786903752e
9a5520df98e711546a2a988016a9ea036a2967313eee899a350924e574c55763cdab82e9dda794608ac3c054c
(19d2dae8233e53c8a41c96f310a8941bae75b4d650645f53f906c0fc74226f01)
 Protocol Message
  Trust Establishment with SAS
```

```
    Protocol UID: db80e14a979ef133a2a71358da20dc31f4d8e67d42fbd18064b6f8d9451477fb
    Mutual trust confirmation
To:
68747470733a2f2f7365727665722e6f6c7669642e696f0000bfe8eb98a77f8e738496ad620284fccb746b51ff6
745f512f0cada3e2c05f2c801372f70dcd058f6380cb949b007a089148d2c00bff25687aee1330069e8444c94
(238f881d0634e1d66593ce6f860eadd28c2a5b6946dc1a590648bc8bbf3186e2)
Protocol Message
  Trust Establishment with SAS
    Protocol UID: db80e14a979ef133a2a71358da20dc31f4d8e67d42fbd18064b6f8d9451477fb
    Mutual trust confirmation
```

L'analyse dynamique des échanges prenant part dans le protocole *Trust Establishment Protocol with SAS* permet de confirmer que le protocole se déroule conformément à **[DOC_CRYPTO]** et qu'ils sont correctement chiffrés en utilisant les clés asymétriques dédiées.

Conclusion

La fonctionnalité d'authentification des utilisateurs repose sur un protocole spécifique disposant d'une preuve de sécurité. Ce protocole repose lui-même sur des primitives cryptographiques connues.

Le design des fonctionnalités cryptographiques ayant déjà été analysé dans Analyse de la spécification cryptographique page 15, il était nécessaire de vérifier que les autres implémentations sont correctes.

L'audit a permis de confirmer la bonne implémentation des fonctionnalités étudiées grâce aux points suivants :

- L'utilisation de primitives cryptographiques testées et éprouvées ;
- Une réimplémentation de la fonctionnalité à partir de briques logicielles différentes donnant les mêmes résultats que l'application ;
- La revue de l'implémentation du protocole spécifique à l'établissement d'une authentification mutuelle entre deux utilisateurs.

En conséquence, la fonction de sécurité **[FS1]** ne présente pas de faiblesse quant à la menace **[M3]** et empêche un attaquant d'usurper l'identité d'un utilisateur d'Olvid.

L'implémentation de la fonction de sécurité est conforme à la spécification et à l'état de l'art. Aucune vulnérabilité de conception ou d'implémentation n'a été identifiée.

FS2 : Authentification des échanges

FS2 assure que deux utilisateurs, ayant utilisé la mise en relation garantissant l'authentification via FS1, puissent échanger des messages de façon authentifiée. FS2 garantit qu'un attaquant contrôlant totalement les serveurs d'Olvid n'est pas en mesure d'usurper l'identité d'un utilisateur d'Olvid lors d'échange de messages protocolaires.

Chaque message est chiffré avec le chiffrement authentifié présenté dans Chiffrement authentifié page 17. Les clés générées spécifiquement pour chaque message sont à leur tour chiffrées via du chiffrement asymétrique puis ajoutées dans l'entête du message. Le chiffrement asymétrique utilisé pour chiffrer les clés associées au message doit garantir l'authentification des échanges.

Méthodologie

L'analyse de cette fonction de sécurité est réalisée en deux parties.

- L'analyse statique du code source de l'application ;
- L'analyse dynamique via la capture et le déchiffrement des messages protocolaires.

L'architecture de cette fonction de sécurité est documentée dans le chapitre IV de **[DOC_CRYPTO]**.

Analyse statique

Les messages protocolaires sont échangés en étant chiffrés par chiffrement symétrique. La clé utilisée pour chiffrer le message est unique et partagée avec le message. Elle peut ainsi être elle-même chiffrée via un chiffrement asymétrique ou symétrique en utilisant une clé secrète partagée uniquement entre les deux utilisateurs via la réalisation du protocole *ChannelCreationWithContactDevice*. Ce protocole est décrit dans le chapitre *Channel Creation with Contact Device Protocol* de **[DOC_CRYPTO]**.

Afin de garantir l'authentification des participants, il est nécessaire de prendre en compte ces deux cas.

La fonction FS1 garantit que les clés publiques obtenues suite à la réalisation du protocole *Trust Establishment Protocol with SAS* correspondent réellement aux participants de l'échange. La mise en place d'un secret partagé, utilisé pour l'échange des messages applicatifs et des messages protocolaires suivants doit s'assurer de l'authenticité des participants.

Les messages sont envoyés depuis les événements *onClick* de l'activité Android *io.olvid.messenger.activities.DiscussionActivity.*

```
public void onClick(View view) {
   int id = view.getId();
   if (id == R.id.compose_message_send_button) {
       sendMessage();
       hideAttachStuff();
    }
    [...]
}
```

La fonction *sendMessage* de cette même activité démarre simplement la tâche *io.olvid.messenger.databases.tasks.PostMessageInDiscussionTask*. Cette tâche créée quant à elle un *io.olvid.messenger.databases.entity.Message* puis appelle sa méthode *post*. Cette méthode appelle ensuite la méthode *io.olvid.engine.engine.Engine#post*. Cette dernière créé des *io.olvid.engine.datatypes.containers.ChannelApplicationMessageToSend*.

```
public ObvPostMessageOutput post(byte[] messagePayload, ObvOutboundAttachment[]
outboundAttachments, List<byte[]> bytesContactIdentities, byte[] bytesOwnedIdentity,
boolean hasUserContent, boolean isVoipMessage) {
    [...]
    for (String server: contactServersHashMap.keySet()) {
        [...]
        try {
            ChannelApplicationMessageToSend.Attachment[] attachments = new
ChannelApplicationMessageToSend.Attachment[outboundAttachments.length];
            for (int i = 0; i < outboundAttachments.length; i++) {
                attachments[i] = new ChannelApplicationMessageToSend.Attachment(
                        outboundAttachments[i].getPath(),
                        false,
                        outboundAttachments[i].getAttachmentLength(),
                        outboundAttachments[i].getMetadata()
                );
            }
            Identity ownedIdentity = Identity.of(bytesOwnedIdentity);
            ChannelApplicationMessageToSend message = new ChannelApplicationMessageToSend(
                    contactIdentities.toArray(new Identity[0]),
                    ownedIdentity,
                    messagePayload,
                    attachments,
                    hasUserContent,
                    isVoipMessage
            );

            UID messageUid = null;
            try (EngineSession engineSession = getSession()) {
                try {
                    engineSession.session.startTransaction();
                    messageUid = channelManager.post(engineSession.session, message, prng);
                    engineSession.session.commit();
                } catch (Exception e) {
                    engineSession.session.rollback();
                }
            }
            [...]
```

```
        }
    [...]
}
```

Les messages ainsi générés sont envoyés via un appel à *io.olvid.engine.channel.ChannelManager#post.* Cet appel termine par un appel à *io.olvid.engine.channel.datatypes.NetworkChannel#post* dans le cadre d'un message envoyé sur le réseau. Le chiffrement est alors réalisé à cette étape.

```
public static UID post(ChannelManagerSession channelManagerSession, ChannelMessageToSend
message, PRNGService prng) throws Exception {
    [...]
    NetworkChannel[] networkChannels = acceptableChannelsForPosting(channelManagerSession,
message);
    if (networkChannels.length == 0) {
        Logger.i("No acceptable channels were found for posting");
        throw new Exception();
    }
    [...]
    AuthEnc authEnc = Suite.getDefaultAuthEnc(suiteVersion);
    AuthEncKey messageKey = authEnc.generateKey(prng);
    MessageToSend.Header[] headers = new MessageToSend.Header[networkChannels.length];
    boolean partOfFullRatchetProtocol = (message instanceof ChannelProtocolMessageToSend)
&& ((ChannelProtocolMessageToSend) message).isPartOfFullRatchetProtocolOfTheSendSeed();
    for (int i=0; i<networkChannels.length; i++) {
        headers[i] = networkChannels[i].wrapMessageKey(messageKey, prng,
partOfFullRatchetProtocol);
    }
    [...]
    MessageToSend messageToSend;
    UID messageUid = new UID(prng);
    switch (message.getMessageType()) {
        case MessageType.APPLICATION_MESSAGE_TYPE:
            [...]
            ChannelApplicationMessageToSend channelApplicationMessageToSend =
(ChannelApplicationMessageToSend) message;
            ChannelApplicationMessageToSend.Attachment[] attachments =
channelApplicationMessageToSend.getAttachments();
            Encoded[] listOfEncodedAttachments = new Encoded[attachments.length + 1];
            MessageToSend.Attachment[] messageToSendAttachments = new
MessageToSend.Attachment[attachments.length];
            for (int i=0; i<attachments.length; i++) {
                AuthEncKey attachmentKey = authEnc.generateKey(prng);
                listOfEncodedAttachments[i] = Encoded.of(new Encoded[]{
                        Encoded.of(attachmentKey),
                        Encoded.of(attachments[i].getMetadata())
                });
                messageToSendAttachments[i] = new
MessageToSend.Attachment(attachments[i].getUrl(), attachments[i].isDeleteAfterSend(),
attachments[i].getAttachmentLength(), attachmentKey);
            }
            // add the message payload after the attachment keys and metadata
            listOfEncodedAttachments[attachments.length] =
Encoded.of(channelApplicationMessageToSend.getMessagePayload());
            Encoded plaintextContent = Encoded.of(new Encoded[]{
                    Encoded.of(MessageType.APPLICATION_MESSAGE_TYPE),
                    Encoded.of(listOfEncodedAttachments)
            });
            EncryptedBytes encryptedContent = authEnc.encrypt(messageKey,
plaintextContent.getBytes(), prng);
```

```
            messageToSend = new
MessageToSend(message.getSendChannelInfo().getFromIdentity(), messageUid, server,
encryptedContent, headers, messageToSendAttachments,
channelApplicationMessageToSend.hasUserContent(),
channelApplicationMessageToSend.isVoipMessage());
            break;
        case MessageType.PROTOCOL_MESSAGE_TYPE:
            if (!(message instanceof ChannelProtocolMessageToSend)) {
                Logger.w("Trying to post a message of type " + message.getMessageType() + "
that is not a ChannelProtocolMessageToSend.");
                throw new Exception();
            }
            ChannelProtocolMessageToSend channelProtocolMessageToSend =
(ChannelProtocolMessageToSend) message;
            plaintextContent = Encoded.of(new Encoded[]{
                    Encoded.of(MessageType.PROTOCOL_MESSAGE_TYPE),
                    channelProtocolMessageToSend.getEncodedElements()
            });
            encryptedContent = authEnc.encrypt(messageKey, plaintextContent.getBytes(),
prng);
            messageToSend = new
MessageToSend(message.getSendChannelInfo().getFromIdentity(), messageUid, server,
encryptedContent, headers);
            break;
        default:
            Logger.w("Trying to post a message of type " + message.getMessageType() + " on
a network channel.");
            throw new Exception();
    }
    channelManagerSession.networkSendDelegate.post(channelManagerSession.session,
messageToSend);
    return messageUid;
}
```

Cette fonction traite deux types de messages : protocolaire et applicatif. Dans le cas d'un message applicatif, les pièces jointes sont également ajoutées au message. Dans tous les cas, le chiffrement est opéré en deux phases. Premièrement, le message et les pièces jointes sont chiffrés via le mécanisme de chiffrement authentifié décrit dans le chapitre 11 de **[DOC_CRYPTO]**. Ce mécanisme est étudié dans FS2 : Authentification des échanges page 42. Dans un deuxième temps la clé utilisée pour le chiffrement du message doit elle-même être chiffrée via l'utilisation de *io.olvid.engine.channel.datatypes.NetworkChannel#wrapMessageKey*. Deux classes implémentent cette interface :

- io.olvid.engine.channel.datatypes.AsymmetricChannel#wrapMessageKey
- iio.olvid.engine.channel.databases.ObliviousChannel#wrapMessageKey

La première correspond à l'utilisation d'un canal asymétrique. Elle est utilisée principalement pour la mise en place d'un canal de chiffrement symétrique, correspondant au deuxième type de canal.

Chiffrement asymétrique

Le chiffrement asymétrique est réalisé en utilisant *io.olvid.engine.channel.datatypes.AsymmetricChannel*. En particulier lors de la construction d'un message la fonction *wrapMessageKey* associée au canal asymétrique est utilisée.

```
public MessageToSend.Header wrapMessageKey(AuthEncKey messageKey, PRNGService prng, boolean
partOfFullRatchetProtocol) {
    if (encryptionForIdentityDelegate == null) {
        return null;
    }
    EncryptedBytes wrappedKey = encryptionForIdentityDelegate.wrap(messageKey, toIdentity,
prng);
```

```
    return new MessageToSend.Header(toDeviceUid, toIdentity, wrappedKey);
}
```

Le chiffrement est réalisé via l'interface *io.olvid.engine.metamanager.EncryptionForIdentityDelegate#wrap*. La seule implémentation existante est *io.olvid.engine.identity.IdentityManager#wrap*.

```
public EncryptedBytes wrap(AuthEncKey messageKey, Identity toIdentity, PRNGService prng) {
    try {
        PublicKeyEncryption pubEnc =
Suite.getPublicKeyEncryption(toIdentity.getEncryptionPublicKey());
        return pubEnc.encrypt(toIdentity.getEncryptionPublicKey(),
Encoded.of(messageKey).getBytes(), prng);
    } catch (InvalidKeyException e) {
        return null;
    }
}
```

Le chiffrement asymétrique correspond ensuite à celui présenté dans Chiffrement asymétrique page 29.

Établissement d'un secret partagé

L'établissement d'un secret partagé est nécessaire pour utiliser le chiffrement symétrique tel qu'utilisé dans *io.olvid.engine.channel.databases.ObliviousChannel*. Cet établissement passe par l'échange de messages protocolaires via un canal *io.olvid.engine.channel.datatypes.AsymmetricChannel*. Il est donc nécessaire de s'assurer de l'identité du correspondant. Cette garantie est apportée par une signature envoyée dans le message de *ping* du protocole *io.olvid.engine.protocol.protocols.ChannelCreationWithContactDeviceProtocol*.

```
public static class SendPingStep extends ProtocolStep {
    private final InitialProtocolState startState;
    private final InitialMessage receivedMessage;
    [...]
    public ConcreteProtocolState executeStep() throws Exception {
        [...]
        // send a signed ping proving you trust the contact and have no channel with him
        byte[] prefix = new byte[SIGNATURE_CHALLENGE_PREFIX.length + UID.UID_LENGTH*2];
        System.arraycopy(SIGNATURE_CHALLENGE_PREFIX, 0, prefix, 0,
SIGNATURE_CHALLENGE_PREFIX.length);
        System.arraycopy(receivedMessage.contactDeviceUid.getBytes(), 0, prefix,
SIGNATURE_CHALLENGE_PREFIX.length, UID.UID_LENGTH);
        UID currentDeviceUid =
protocolManagerSession.identityDelegate.getCurrentDeviceUidOfOwnedIdentity(protocolManagerS
ession.session, getOwnedIdentity());
        if (currentDeviceUid == null) {
            return new CancelledState();
        }
        System.arraycopy(currentDeviceUid.getBytes(), 0, prefix,
SIGNATURE_CHALLENGE_PREFIX.length+UID.UID_LENGTH, UID.UID_LENGTH);
        byte[] signature = protocolManagerSession.identityDelegate.signIdentities(
                protocolManagerSession.session,
                prefix,
                new Identity[]{ receivedMessage.contactIdentity, getOwnedIdentity()},
                getOwnedIdentity(),
                getPrng()
        );
        // send the ping containing the signature
        CoreProtocolMessage coreProtocolMessage =
buildCoreProtocolMessage(SendChannelInfo.createAsymmetricChannelInfo(receivedMessage.contac
tIdentity, getOwnedIdentity(), new UID[]{receivedMessage.contactDeviceUid}));
```

```
        ChannelMessageToSend messageToSend = new PingMessage(coreProtocolMessage,
getOwnedIdentity(), currentDeviceUid, signature).generateChannelProtocolMessageToSend();
        protocolManagerSession.channelDelegate.post(protocolManagerSession.session,
messageToSend, getPrng());
        return new PingSentState();
    }
}
```

La signature est réalisée par *io.olvid.engine.metamanager.IdentityDelegate#signIdentities*. L'unique implémentation correspondante est *io.olvid.engine.identity.IdentityManager#signIdentities*.

```
public byte[] signIdentities(Session session, byte[] prefix, Identity[] identities,
Identity ownedIdentity, PRNGService prng) throws Exception {
    try {
        IdentityManagerSession identityManagerSession = wrapSession(session);
        OwnedIdentity ownedIdentityObject = OwnedIdentity.get(identityManagerSession,
ownedIdentity);
        if (ownedIdentityObject == null) {
            throw new Exception("Unknown owned identity");
        }
        PrivateIdentity privateIdentity = ownedIdentityObject.getPrivateIdentity();
        [...]
        SignaturePublicKey signaturePublicKey =
ownedIdentity.getServerAuthenticationPublicKey().getSignaturePublicKey();
        SignaturePrivateKey signaturePrivateKey =
privateIdentity.getServerAuthenticationPrivateKey().getSignaturePrivateKey();
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        baos.write(prefix);
        for (Identity identity: identities) {
            baos.write(identity.getBytes());
        }
        byte[] padding = prng.bytes(PADDING_LENGTH);
        baos.write(padding);
        byte[] challenge = baos.toByteArray();
        baos.close();
        Signature signature = Suite.getSignature(signaturePrivateKey);
        byte[] signatureBytes =  signature.sign(signaturePrivateKey, signaturePublicKey,
challenge, prng);
        byte[] output = new byte[PADDING_LENGTH + signatureBytes.length];
        System.arraycopy(padding, 0, output, 0, PADDING_LENGTH);
        System.arraycopy(signatureBytes, 0, output, PADDING_LENGTH, signatureBytes.length);
        return output;
    } catch (InvalidKeyException e) {
        e.printStackTrace();
        return null;
    }
}
```

La signature repose donc sur l'interface *io.olvid.engine.crypto.Signature#sign* dont l'implémentation correspond au choix par défaut pour l'authentification avec les serveurs d'Olvid: *io.olvid.engine.crypto.SignatureECSdsaMDC#sign*.

```
public byte[] internalSign(SignatureECSdsaPrivateKey privateKey, SignatureECSdsaPublicKey
publicKey, byte[] message, PRNGService prng) {
    try {
        int l = curve.byteLength;
        byte[] hashInput = new byte[message.length + 2*l];
        EdwardCurve.ScalarAndPoint aAndaG = curve.generateRandomScalarAndPoint(prng);
        System.arraycopy(Encoded.bytesFromBigUInt(aAndaG.getPoint().getY(), l), 0,
hashInput, 0, l);
```

```
        System.arraycopy(Encoded.bytesFromBigUInt(publicKey.getAy(), l), 0, hashInput, l,
l);
        System.arraycopy(message, 0, hashInput, 2*l, message.length);
        byte[] hash = new HashSHA256().digest(hashInput);
        BigInteger e = Encoded.bigUIntFromBytes(hash);
        BigInteger y =
aAndaG.getScalar().subtract(privateKey.getA().multiply(e)).mod(curve.q);
        byte[] signature = new byte[HashSHA256.OUTPUT_LENGTH + l];
        System.arraycopy(hash, 0, signature, 0, HashSHA256.OUTPUT_LENGTH);
        System.arraycopy(Encoded.bytesFromBigUInt(y, l), 0, signature,
HashSHA256.OUTPUT_LENGTH, l);
        return signature;
    } catch (EncodingException e) {}
    return null;
}
```

Cette implémentation est conforme à **[DOC_CRYPTO]**.

Le message de type *Ping* contenant cette signature est ainsi envoyé au correspondant. Lors de la réception, la signature est vérifiée par la fonction *io.olvid.engine.protocol.protocols.ChannelCreationWithContactDeviceProtocol.SendPingOrEphemeralKeyStep*:

```
public ConcreteProtocolState executeStep() throws Exception {
    ProtocolManagerSession protocolManagerSession = getProtocolManagerSession();
    // check that the contactIdentity in the receivedMessage is indeed trusted by the
ownedIdentity running the protocol
    if (!
protocolManagerSession.identityDelegate.isIdentityAContactIdentityOfOwnedIdentity(protocolM
anagerSession.session, receivedMessage.contactIdentity, getOwnedIdentity())) {
        Logger.w("Received a ping for a ChannelCreationWithContactDeviceProtocol from an
untrusted ContactIdentity");
        return new CancelledState();
    }
    // verify the signature in the PingMessage
    byte[] prefix = new byte[SIGNATURE_CHALLENGE_PREFIX.length + UID.UID_LENGTH*2];
    System.arraycopy(SIGNATURE_CHALLENGE_PREFIX, 0, prefix, 0,
SIGNATURE_CHALLENGE_PREFIX.length);
    UID currentDeviceUid =
protocolManagerSession.identityDelegate.getCurrentDeviceUidOfOwnedIdentity(protocolManagerS
ession.session, getOwnedIdentity());
    if (currentDeviceUid == null) {
        return new CancelledState();
    }
    System.arraycopy(currentDeviceUid.getBytes(), 0, prefix,
SIGNATURE_CHALLENGE_PREFIX.length, UID.UID_LENGTH);
    System.arraycopy(receivedMessage.contactDeviceUid.getBytes(), 0, prefix,
SIGNATURE_CHALLENGE_PREFIX.length+UID.UID_LENGTH, UID.UID_LENGTH);
    boolean signatureIsValid =
protocolManagerSession.identityDelegate.verifyIdentitiesSignature(
            prefix,
            new Identity[]{getOwnedIdentity(), receivedMessage.contactIdentity},
            receivedMessage.contactIdentity,
            receivedMessage.signature
    );
    if (!signatureIsValid) {
        return new CancelledState();
    }
    [...]
}
```

La vérification de la signature passe par *io.olvid.engine.identity.IdentityManager#verifyIdentitiesSignature*:

```
public boolean verifyIdentitiesSignature(byte[] prefix, Identity[] identities, Identity
signerIdentity, byte[] signature) throws Exception {
    try {
        SignaturePublicKey signaturePublicKey =
signerIdentity.getServerAuthenticationPublicKey().getSignaturePublicKey();
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        baos.write(prefix);
        for (Identity identity: identities) {
            baos.write(identity.getBytes());
        }
        baos.write(Arrays.copyOfRange(signature, 0, PADDING_LENGTH));
        byte[] challenge = baos.toByteArray();
        Signature signatureAlgo = Suite.getSignature(signaturePublicKey);
        return signatureAlgo.verify(signaturePublicKey, challenge,
Arrays.copyOfRange(signature, PADDING_LENGTH, signature.length));
    } catch (InvalidKeyException e) {
        e.printStackTrace();
        return false;
    }
}
```

L'interface *io.olvid.engine.crypto.Signature#verify* est chargée de la vérification de la signature. La classe *io.olvid.engine.crypto.SignatureECSdsaMDC#verify* implémente cette interface.

```
public byte[] internalSign(SignatureECSdsaPrivateKey privateKey, SignatureECSdsaPublicKey
publicKey, byte[] message, PRNGService prng) {
    try {
        int l = curve.byteLength;
        byte[] hashInput = new byte[message.length + 2*l];
        EdwardCurve.ScalarAndPoint aAndaG = curve.generateRandomScalarAndPoint(prng);
        System.arraycopy(Encoded.bytesFromBigUInt(aAndaG.getPoint().getY(), l), 0,
hashInput, 0, l);
        System.arraycopy(Encoded.bytesFromBigUInt(publicKey.getAy(), l), 0, hashInput, l,
l);
        System.arraycopy(message, 0, hashInput, 2*l, message.length);
        byte[] hash = new HashSHA256().digest(hashInput);
        BigInteger e = Encoded.bigUIntFromBytes(hash);
        BigInteger y =
aAndaG.getScalar().subtract(privateKey.getA().multiply(e)).mod(curve.q);
        byte[] signature = new byte[HashSHA256.OUTPUT_LENGTH + l];
        System.arraycopy(hash, 0, signature, 0, HashSHA256.OUTPUT_LENGTH);
        System.arraycopy(Encoded.bytesFromBigUInt(y, l), 0, signature,
HashSHA256.OUTPUT_LENGTH, l);
        return signature;
    } catch (EncodingException e) {}
    return null;
}
```

La vérification de signature est conforme à **[DOC_CRYPTO]**.

Lors de la construction du message de réponse au premier message de type *Ping* un mécanisme strictement identique de signature est utilisé.

```
public ConcreteProtocolState executeStep() throws Exception {
    [...]
    // Compute a signature to prove we trust the contact and don't have any channel/ongoing
protocol with him
    // only rewrite the end of the prefix
```

```
    System.arraycopy(receivedMessage.contactDeviceUid.getBytes(), 0, prefix,
SIGNATURE_CHALLENGE_PREFIX.length, UID.UID_LENGTH);
    System.arraycopy(currentDeviceUid.getBytes(), 0, prefix,
SIGNATURE_CHALLENGE_PREFIX.length+UID.UID_LENGTH, UID.UID_LENGTH);
    byte[] signature = protocolManagerSession.identityDelegate.signIdentities(
            protocolManagerSession.session,
            prefix,
            new Identity[]{ receivedMessage.contactIdentity, getOwnedIdentity()},
            getOwnedIdentity(),
            getPrng()
    );
   [...]
}
```

La vérification de cette dernière signature est réalisée lors de l'exécution de l'étape *io.olvid.engine.protocol.protocols.ChannelCreationWithContactDeviceProtocol.SendEphemeralKeyAndK1Step#executeStep* à la réception du message.

```
public ConcreteProtocolState executeStep() throws Exception {
    ProtocolManagerSession protocolManagerSession = getProtocolManagerSession();
    {
        // check that the contactIdentity in the receivedMessage is indeed trusted by the
ownedIdentity running the protocol
        if (!
protocolManagerSession.identityDelegate.isIdentityAContactIdentityOfOwnedIdentity(protocolM
anagerSession.session, receivedMessage.contactIdentity, getOwnedIdentity())) {
            Logger.w("Received a ping for a ChannelCreationWithContactDeviceProtocol from
an untrusted ContactIdentity");
            return new CancelledState();
        }
        // verify the signature in the AliceIdentityAndEphemeralKeyMessage
        byte[] prefix = new byte[SIGNATURE_CHALLENGE_PREFIX.length + UID.UID_LENGTH * 2];
        System.arraycopy(SIGNATURE_CHALLENGE_PREFIX, 0, prefix, 0,
SIGNATURE_CHALLENGE_PREFIX.length);
        UID currentDeviceUid =
protocolManagerSession.identityDelegate.getCurrentDeviceUidOfOwnedIdentity(protocolManagerS
ession.session, getOwnedIdentity());
        if (currentDeviceUid == null) {
            return new CancelledState();
        }
        System.arraycopy(currentDeviceUid.getBytes(), 0, prefix,
SIGNATURE_CHALLENGE_PREFIX.length, UID.UID_LENGTH);
        System.arraycopy(receivedMessage.contactDeviceUid.getBytes(), 0, prefix,
SIGNATURE_CHALLENGE_PREFIX.length + UID.UID_LENGTH, UID.UID_LENGTH);
        boolean signatureIsValid =
protocolManagerSession.identityDelegate.verifyIdentitiesSignature(
                prefix,
                new Identity[]{getOwnedIdentity(), receivedMessage.contactIdentity},
                receivedMessage.contactIdentity,
                receivedMessage.signature
        );
        if (!signatureIsValid) {
            return new CancelledState();
        }
        [...]
    }
}
```

Dans la suite des échanges du protocole *ChannelCreationWithContactDeviceProtocol* les deux participants utilisent des clés asymétriques temporaires pour échanger, via deux KEM, deux clés symétriques. Premièrement, Alice crée une clé

temporaire et l'envoie à Bob lors de l'étape *io.olvid.engine.protocol.protocols.ChannelCreationWithContactDeviceProtocol.SendPingOrEphemeralKeyStep*.

```
public static class SendPingOrEphemeralKeyStep extends ProtocolStep {
    [...]
    public ConcreteProtocolState executeStep() throws Exception {
        [...]
        {
            [...]
            KeyPair keyPair =
Suite.generateEncryptionKeyPair(getOwnedIdentity().getEncryptionPublicKey().getAlgorithmImp
lementation(), getPrng());
            if (keyPair == null) {
                throw new Exception();
            }
            CoreProtocolMessage coreProtocolMessage =
buildCoreProtocolMessage(SendChannelInfo.createAsymmetricChannelInfo(receivedMessage.contac
tIdentity, getOwnedIdentity(), new UID[]{receivedMessage.contactDeviceUid}));
            ChannelMessageToSend messageToSend = new
AliceIdentityAndEphemeralKeyMessage(coreProtocolMessage, getOwnedIdentity(),
currentDeviceUid, signature, (EncryptionPublicKey)
keyPair.getPublicKey()).generateChannelProtocolMessageToSend();
            protocolManagerSession.channelDelegate.post(protocolManagerSession.session,
messageToSend, getPrng());
            return new WaitingForK1State(receivedMessage.contactIdentity,
receivedMessage.contactDeviceUid, (EncryptionPrivateKey) keyPair.getPrivateKey());
        }
    }
}
```

À la réception, Bob crée une clé éphémère et démarre un KEM avec la clé temporaire d'Alice.

```
 public static class SendEphemeralKeyAndK1Step extends ProtocolStep {
    [...]
    public ConcreteProtocolState executeStep() throws Exception {
        [...]
        KeyPair keyPair =
Suite.generateEncryptionKeyPair(getOwnedIdentity().getEncryptionPublicKey().getAlgorithmImp
lementation(), getPrng());
        if (keyPair == null) {
            throw new Exception();
        }
        // compute k1
        PublicKeyEncryption publicKeyEncryption =
Suite.getPublicKeyEncryption(receivedMessage.contactEphemeralPublicKey);
        CiphertextAndKey ciphertextAndKey =
publicKeyEncryption.kemEncrypt(receivedMessage.contactEphemeralPublicKey, getPrng());
        AuthEncKey k1 = ciphertextAndKey.getKey();
        EncryptedBytes c1 = ciphertextAndKey.getCiphertext();
        CoreProtocolMessage coreProtocolMessage =
buildCoreProtocolMessage(SendChannelInfo.createAsymmetricChannelInfo(receivedMessage.contac
tIdentity, getOwnedIdentity(), new UID[]{receivedMessage.contactDeviceUid}));
        ChannelMessageToSend messageToSend = new
BobEphemeralKeyAndK1Message(coreProtocolMessage, (EncryptionPublicKey)
keyPair.getPublicKey(), c1).generateChannelProtocolMessageToSend();
        protocolManagerSession.channelDelegate.post(protocolManagerSession.session,
messageToSend, getPrng());
        return new WaitingForK2State(receivedMessage.contactIdentity,
receivedMessage.contactDeviceUid, (EncryptionPrivateKey) keyPair.getPrivateKey(), k1);
    }
```

```
}
```

Lorsque Alice reçoit le message contenant le KEM, elle peut alors le déchiffrer et ainsi créer le canal symétrique. Lors de la même étape, elle utilise la clé éphémère de Bob pour lancer un KEM.

```
public static class RecoverK1AndSendK2AndCreateChannelStep extends ProtocolStep {
    [...]
    public ConcreteProtocolState executeStep() throws Exception {
        ProtocolManagerSession protocolManagerSession = getProtocolManagerSession();
        PublicKeyEncryption publicKeyEncryption =
Suite.getPublicKeyEncryption(startState.ephemeralPrivateKey);
        AuthEncKey k1 = publicKeyEncryption.kemDecrypt(startState.ephemeralPrivateKey,
receivedMessage.c1);
        if (k1 == null) {
            Logger.e("Could not recover k1.");
            return new CancelledState();
        }
        // compute k2
        publicKeyEncryption =
Suite.getPublicKeyEncryption(receivedMessage.contactEphemeralPublicKey);
        CiphertextAndKey ciphertextAndKey =
publicKeyEncryption.kemEncrypt(receivedMessage.contactEphemeralPublicKey, getPrng());
        AuthEncKey k2 = ciphertextAndKey.getKey();
        EncryptedBytes c2 = ciphertextAndKey.getCiphertext();
        Seed seed = Seed.of(k1, k2);
        [...]
        // create the channel
        protocolManagerSession.channelDelegate.createObliviousChannel(
                protocolManagerSession.session,
                getOwnedIdentity(),
                startState.contactDeviceUid,
                startState.contactIdentity,
                seed,
                0 // TODO: use the actual engine version for this channel!
        );
        CoreProtocolMessage coreProtocolMessage =
buildCoreProtocolMessage(SendChannelInfo.createAsymmetricChannelInfo(startState.contactIden
tity, getOwnedIdentity(), new UID[]{startState.contactDeviceUid}));
        ChannelMessageToSend messageToSend = new K2Message(coreProtocolMessage,
c2).generateChannelProtocolMessageToSend();
        protocolManagerSession.channelDelegate.post(protocolManagerSession.session,
messageToSend, getPrng());
        return new WaitForFirstAckState(startState.contactIdentity,
startState.contactDeviceUid);
    }
}
```

À la réception du KEM initié par Alice, Bob peut déchiffrer la clé et créer sa partie du canal symétrique.

```
public static class RecoverK2CreateChannelAndSendAckStep extends ProtocolStep {
    [...]
    public ConcreteProtocolState executeStep() throws Exception {
        ProtocolManagerSession protocolManagerSession = getProtocolManagerSession();
        PublicKeyEncryption publicKeyEncryption =
Suite.getPublicKeyEncryption(startState.ephemeralPrivateKey);
        AuthEncKey k2 = publicKeyEncryption.kemDecrypt(startState.ephemeralPrivateKey,
receivedMessage.c2);
        if (k2 == null) {
            Logger.e("Could not recover k2.");
            return new CancelledState();
```

```
        }
        [...]
        Seed seed = Seed.of(startState.k1, k2);
        protocolManagerSession.channelDelegate.createObliviousChannel(
                protocolManagerSession.session,
                getOwnedIdentity(),
                startState.contactDeviceUid,
                startState.contactIdentity,
                seed,
                0 // TODO: use the actual engine version for this channel!
        );
        [...]
    }
}
```

Les deux clés échangées via KEM permettent à Alice et Bob de dériver une clé symétrique identique utilisée par la suite pour l'échange de message via un canal symétrique.

L'ensemble des étapes participant à la création d'un canal symétrique est conforme à **[DOC_CRYPTO]**.

Chiffrement symétrique

L'authentification des participants repose sur le fait que seuls les participants ayant accès au secret partagé sont en capacité de calculer et de vérifier le MAC associé aux messages. La garantie que le secret partagé est bien confidentiel et authentifié est fournie par le protocole *io.olvid.engine.protocol.protocols.ChannelCreationWithContactDeviceProtocol* dont l'analyse a été réalisée dans Établissement d'un secret partagé page 46.

Le chiffrement symétrique est réalisé dans le canal *io.olvid.engine.channel.databases.ObliviousChannel*.

```
//io.olvid.engine.channel.databases.ObliviousChannel#wrapMessageKey
public MessageToSend.Header wrapMessageKey(AuthEncKey messageKey, PRNGService prng, boolean
partOfFullRatchetProtocol) {
    RatchetingOutput ratchetingOutput = selfRatchet();
    if (ratchetingOutput == null) {
        return null;
    }
    AuthEnc authEnc = Suite.getAuthEnc(ratchetingOutput.getAuthEncKey());
    EncryptedBytes encryptedMessageKey;
    try {
        encryptedMessageKey = authEnc.encrypt(ratchetingOutput.getAuthEncKey(),
Encoded.of(messageKey).getBytes(), prng);
    } catch (InvalidKeyException e) {
        e.printStackTrace();
        return null;
    }
    [...]
}
```

La fonction *encrypt* de l'interface *io.olvid.engine.crypto.AuthEnc* est responsable du chiffrement et de l'authentification. La classe *io.olvid.engine.crypto.AuthEnc* est la seule implémentation de cette interface disponible.

```
public EncryptedBytes encrypt(AuthEncKey key, byte[] plaintext, PRNG prng) throws
InvalidKeyException {
    if (! (key instanceof AuthEncAES256ThenSHA256Key)) {
        throw new InvalidKeyException();
    }
    MACHmacSha256Key macKey = ((AuthEncAES256ThenSHA256Key) key).getMacKey();
    SymEncCTRAES256Key encKey = ((AuthEncAES256ThenSHA256Key) key).getEncKey();
    MACHmacSha256 mac = new MACHmacSha256();
```

```
    SymEncCtrAES256 enc = new SymEncCtrAES256(encKey);
    byte[] ciphertext = new byte[ciphertextLengthFromPlaintextLength(plaintext.length)];
    byte[] iv = prng.bytes(SymEncCtrAES256.IV_BYTE_LENGTH);
    EncryptedBytes encrypted = enc.encrypt(iv, plaintext);
    byte[] encryptedBytes = encrypted.getBytes();
    System.arraycopy(encryptedBytes, 0, ciphertext, 0, encryptedBytes.length);
    byte[] hash = mac.digest(macKey, encryptedBytes);
    System.arraycopy(hash, 0, ciphertext,
enc.ciphertextLengthFromPlaintextLength(plaintext.length), hash.length);
    return new EncryptedBytes(ciphertext);
}
```

La fonction *io.olvid.engine.crypto.MACHmacSha256#digest* est responsable de la génération du MAC permettant l'authentification des échanges.

```
import javax.crypto.Mac;

public byte[] digest(MACKey key, byte[] bytes) throws InvalidKeyException {
    try {
        Mac h = Mac.getInstance("HmacSHA256");
        h.init(new SecretKeySpec(key.getKeyBytes(), "HmacSHA256"));
        return h.doFinal(bytes);
    } catch (NoSuchAlgorithmException e) {}
    return null;
}
```

Conformément à **[DOC_CRYPTO]**, HMAC SHA256 est utilisé pour la génération du MAC. L'implémentation utilisée est celle fournie par le système : *javax.crypto.Mac*, considérée comme conforme à sa spécification ainsi qu'aux règles et recommandations du RGS selon l'hypothèse *H.BouncyCastle* introduite dans la section Problème de sécurité et environnement page 7.

Analyse dynamique

Chiffrement asymétrique

Le déchiffrement des messages protocolaires échangés via un canal asymétrique permet de confirmer de façon dynamique le fonctionnement du chiffrement asymétrique.

La capture suivante présente un message tel que reçu sur le serveur.

```
header: [ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff,
7c6892143be6bb69db4e65153e51b8cc3c610a5ea99c1656003b516be9a8791c77b25d62508cf84efafaf158cef
77c5c8f807decffe2f352dd86969b7bb6969d4e030a5abd5d02999b74b3470632d03bc81c74b2cd44c5a3b854c3
716d93733ff0b9c6217ec40f25cd931d922bd2f16c18df1eeba367b00f29b658a8f51f4ba9e6473ad2b5ec247fa
fda6d161628db6e080d83d2f207b1c4654cc240eff9f36daa377b2146eb536027743c6105ffd69666448967cc2b
4fd6d9,
68747470733a2f2f7365727665722e6f6c7669642e696f0000bfe8eb98a77f8e738496ad620284fccb746b51ff6
745f512f0cada3e2c05f2c801372f70dcd058f6380cb949b007a089148d2c00bff25687aee1330069e8444c94]
Message:
e6b6ad8cc1920c4a915d25f4ce17ede3b3f4e938b93d20a9725fa084f1248933f56e49532a8cf5472664f8e2932
e359d7cb41e39ce2ff988df482dfa49b8106f98ba885b29e09571d71c76b433ed95e253be86c887060b38ea6431
e5654b82596fd1b3c19cadf456d4cb68cdd3aec4ef438643da3289c3f395ef3c02a13a7d6bc5fa2aeb2ed92ee7d
60fe6a49b87948d78d113b6953f3bfdea710d49e3f74694474733cb3ee3cc37887608525280ba7018a41c593b9f
4cc12996c32ff027162aa2637fc4a009bd4ad6df8a1f7751f3507c8d71d7328f9a27fa6f35b31b84534f9f2a4b9
2e73f4f68048de5d1f9816e4023a1090e6bea3bbd9e82b31f346faa751c8ad9d15502606a6045dbf99a2440e7d7
224ae80f95adf6d7225cbe1f9248e23e97b7346c1c9dfb4f7ea8b1cdf67a78fe13aef0458b8586c2207505ba437
5d0b56dc78b62e77047a5be
```

Le message est décomposable en deux parties, un entête contenant les ID des destinataires et un corps chiffré.

L'entête contient les clés symétriques utilisées pour le chiffrement du corps du message. Ces clés sont chiffrées via le KEM présenté ci-avant.

À titre d'information l'entropie de l'entête contenant le chiffré des clés du message est de 7.38 bit/octet. L'entropie est calculée ici comme **[l'entropie de Shanon]**, soit l'inverse de la somme des probabilités de chaque symbole multipliées par leurs logarithmes binaires. Elle est ici calculée sur les octets de l'ensemble de l'entête.

Il est possible de confirmer que les clés sont correctement chiffrées en récupérant les clés privées des participants directement depuis la base de données *SQLite* de l'application et en réimplémentant le déchiffrement. Le code réalisant cette opération est disponible en annexe.

Pour l'exemple ci-dessus, on récupère donc les clés symétriques chiffrées via le KEM.

```
Symetric key: <SymmetricKey algoClassByteId=2 algoImplemByteId=0 dict={'mackey':
6f87600fc1f6b2e1661f991fcff517f03437c168b86b58f7e2e987c57ea1ab09, 'enckey':
789e0468841c298a51e3ce91f0bf79883faa4257bd651d9329ba869f43c60cf8}>
```

Établissement du chiffrement symétrique

La capture des échanges de messages entre deux participants permet d'obtenir les messages suivants.

```
To:
68747470733a2f2f7365727665722e6f6c7669642e696f0000841cc9dd927ebf9640f2ecea5a761e4786903752e
9a5520df98e711546a2a988016a9ea036a2967313eee899a350924e574c55763cdab82e9dda794608ac3c054c
(19d2dae8233e53c8a41c96f310a8941bae75b4d650645f53f906c0fc74226f01)
Protocol Message
  ChannelCreationWithContactDevice
    Ping
      Contact identity:
68747470733a2f2f7365727665722e6f6c7669642e696f0000bfe8eb98a77f8e738496ad620284fccb746b51ff6
745f512f0cada3e2c05f2c801372f70dcd058f6380cb949b007a089148d2c00bff25687aee1330069e8444c94
      Contact device UIDs: 238f881d0634e1d66593ce6f860eadd28c2a5b6946dc1a590648bc8bbf3186e2
      Signature:
996e01426e50bdc6c16b761f8ee9bca9e2eca9a7aeaa28f5f38d4134ca22e9138193db3cc581ef7027c06798908
fddda19d55d2e3af8c08c256feaa29503e6e118b75db0c00bd07341f88a7c963a9647


To:
68747470733a2f2f7365727665722e6f6c7669642e696f0000bfe8eb98a77f8e738496ad620284fccb746b51ff6
745f512f0cada3e2c05f2c801372f70dcd058f6380cb949b007a089148d2c00bff25687aee1330069e8444c94
(238f881d0634e1d66593ce6f860eadd28c2a5b6946dc1a590648bc8bbf3186e2)
Protocol Message
  ChannelCreationWithContactDevice
    Alice identity and ephemeral key
      Contact identity:
68747470733a2f2f7365727665722e6f6c7669642e696f0000841cc9dd927ebf9640f2ecea5a761e4786903752e
9a5520df98e711546a2a988016a9ea036a2967313eee899a350924e574c55763cdab82e9dda794608ac3c054c
      Contact device UIDs: 19d2dae8233e53c8a41c96f310a8941bae75b4d650645f53f906c0fc74226f01
      Signature:
ecc00396f2e2865731c93002a0a886306e46480ce0f2a998ae98e5de65784524143103caf8d227de2ffb9f75f75
053922d303b5025f807297783dd1fa368598f7f5c85f85fd8975d2449c6a3eb1b422a
      Ephemeral key:       <PublicKey algoClassByteId=18 algoImplemByteId=1 dict={'x':
54432748949623077581758854764032130882863857652809111289206043426724923811723, 'y':
14572361477633097156165858894768271078260229752176485895882798999322794342 82}>


To:
68747470733a2f2f7365727665722e6f6c7669642e696f0000bfe8eb98a77f8e738496ad620284fccb746b51ff6
745f512f0cada3e2c05f2c801372f70dcd058f6380cb949b007a089148d2c00bff25687aee1330069e8444c94
(238f881d0634e1d66593ce6f860eadd28c2a5b6946dc1a590648bc8bbf3186e2)
```

```
Protocol Message
  ChannelCreationWithContactDevice
    Ping
      Contact identity:
68747470733a2f2f7365727665722e6f6c7669642e696f0000841cc9dd927ebf9640f2ecea5a761e4786903752e
9a5520df98e711546a2a988016a9ea036a2967313eee899a350924e574c55763cdab82e9dda794608ac3c054c
      Contact device UIDs: 19d2dae8233e53c8a41c96f310a8941bae75b4d650645f53f906c0fc74226f01
      Signature:
c5dd575ebc9d420001d8cdda56cedaffb79c4083640ad1c90bedacf8c83dd64d5b1143dcae969a064795f97f901
08ad22f4e4936f2bbf4888ab01a84a367885a1a0a5cba4bccec114f39d559f00c6e83


To:
68747470733a2f2f7365727665722e6f6c7669642e696f0000841cc9dd927ebf9640f2ecea5a761e4786903752e
9a5520df98e711546a2a988016a9ea036a2967313eee899a350924e574c55763cdab82e9dda794608ac3c054c
(19d2dae8233e53c8a41c96f310a8941bae75b4d650645f53f906c0fc74226f01)
Protocol Message
  ChannelCreationWithContactDevice
    Bob ephemeral key
      Ephemeral key: <PublicKey algoClassByteId=18 algoImplemByteId=1 dict={'x':
54566980364714661881796356575027150978080970279500062210275445105930544186508, 'y':
23607515692119549490315011278865651341644988532768128874661893880723988424786}>
      c1:            572c219553c49df3e2e86c0864d628b5ba8f4c32b60ef6ffede59fb78e84581c


To:
68747470733a2f2f7365727665722e6f6c7669642e696f0000841cc9dd927ebf9640f2ecea5a761e4786903752e
9a5520df98e711546a2a988016a9ea036a2967313eee899a350924e574c55763cdab82e9dda794608ac3c054c
(19d2dae8233e53c8a41c96f310a8941bae75b4d650645f53f906c0fc74226f01)
Protocol Message
  ChannelCreationWithContactDevice
    Ping
      Contact identity:
68747470733a2f2f7365727665722e6f6c7669642e696f0000bfe8eb98a77f8e738496ad620284fccb746b51ff6
745f512f0cada3e2c05f2c801372f70dcd058f6380cb949b007a089148d2c00bff25687aee1330069e8444c94
      Contact device UIDs: 238f881d0634e1d66593ce6f860eadd28c2a5b6946dc1a590648bc8bbf3186e2
      Signature:
529de690017ef629665b567dad613dae351525adfb0b09975c1c95a89c05c3a83e8434d4a728f479059405b80e9
e3d8709d577da2656c7db2a39dc283b3b673e6f685bccd23ab770b8b7100ffcadf643


To:
68747470733a2f2f7365727665722e6f6c7669642e696f0000bfe8eb98a77f8e738496ad620284fccb746b51ff6
745f512f0cada3e2c05f2c801372f70dcd058f6380cb949b007a089148d2c00bff25687aee1330069e8444c94
(238f881d0634e1d66593ce6f860eadd28c2a5b6946dc1a590648bc8bbf3186e2)
Protocol Message
  ChannelCreationWithContactDevice
    Alice identity and ephemeral key
      Contact identity:
68747470733a2f2f7365727665722e6f6c7669642e696f0000841cc9dd927ebf9640f2ecea5a761e4786903752e
9a5520df98e711546a2a988016a9ea036a2967313eee899a350924e574c55763cdab82e9dda794608ac3c054c
      Contact device UIDs: 19d2dae8233e53c8a41c96f310a8941bae75b4d650645f53f906c0fc74226f01
      Signature:
bcdbc44ff3db26c9c4f26d877a1315ce14ae733acd5799545462bbe5429ad99db25280a97e7499dbfebdd07c95e
2bc6d1e1c11303ec0af60e0bc5bd540579d85da5d346dddc0ca9e54a53633b9e39191
      Ephemeral key:       <PublicKey algoClassByteId=18 algoImplemByteId=1 dict={'x':
31536392597439342486440378783852758477437897398385985941659008942672638844071, 'y':
39969346414434849763317218203719979793089636431760762833307655351035741010940}>
```

```
To:
68747470733a2f2f7365727665722e6f6c7669642e696f0000841cc9dd927ebf9640f2ecea5a761e4786903752e
9a5520df98e711546a2a988016a9ea036a2967313eee899a350924e574c55763cdab82e9dda794608ac3c054c
(19d2dae8233e53c8a41c96f310a8941bae75b4d650645f53f906c0fc74226f01)
Protocol Message
  ChannelCreationWithContactDevice
    Bob ephemeral key
      Ephemeral key: <PublicKey algoClassByteId=18 algoImplemByteId=1 dict={'x':
54740850194015971400688713262028809928194113488161544266078731284122346577432, 'y':
50208357726484954889333806414802002750455071819052033191185228636215158931144}>
      c1:            4e61d583649f5907441391586afcd6eea9d84920736542f72b63dd183aeb0b6d


To:
68747470733a2f2f7365727665722e6f6c7669642e696f0000bfe8eb98a77f8e738496ad620284fccb746b51ff6
745f512f0cada3e2c05f2c801372f70dcd058f6380cb949b007a089148d2c00bff25687aee1330069e8444c94
(238f881d0634e1d66593ce6f860eadd28c2a5b6946dc1a590648bc8bbf3186e2)
Protocol Message
  ChannelCreationWithContactDevice
    K2
      c2:            117a30addad745d980ba9434780b8570663167ecbdeef017a7cbbbd2b65df3e2
```

L'analyse dynamique des messages échangés lors de la réalisation du protocole *ChannelCreationWithContactDevice* montre la conformité avec **[DOC_CRYPTO]**.

Conclusion

La fonctionnalité d'authentification des échanges est basée sur plusieurs mécanismes. Chaque mécanisme entre en jeu à différents moments de la vie de l'application. Néanmoins chacun de ces mécanismes repose sur des implémentations spécifiques de primitives cryptographiques standards ainsi que sur un protocole d'échange de clés.

Lorsqu'un canal de communication est en cours de création, l'authentification des échanges repose sur la cryptographie asymétrique et sur l'utilisation de signatures.

Lorsqu'un canal de communication est établi, l'authentification des échanges repose sur le chiffrement symétrique authentifié présenté en Chiffrement authentifié page 17 et sur la garantie que le secret partagé n'est connu que des deux participants. Cette garantie est apportée par l'utilisation d'un protocole spécifique d'échange de clés.

Le design des fonctionnalités cryptographiques ayant déjà été analysé dans Analyse de la spécification cryptographique page 15, il est nécessaire de vérifier que les autres implémentations sont correctes.

L'audit a permis de confirmer le bon fonctionnement de la fonctionnalité sur plusieurs points:

- L'utilisation de primitives cryptographiques testées et éprouvées ;
- Une réimplémentation de la fonctionnalité à partir de briques logicielles différentes donnant les mêmes résultats que l'application ;
- L'analyse exhaustive du code source lié aux chiffrements symétriques et asymétrique des messages ;
- La revue de l'implémentation du protocole d'échange de clé.

En conséquence, la fonction de sécurité **[FS2]** ne présente pas de faiblesse quant à la menace **[M1]** et empêche un attaquant de modifier des messages échangés via l'application Olvid.

L'implémentation de la fonction de sécurité est conforme à la spécification et à l'état de l'art. Aucune vulnérabilité de conception ou d'implémentation n'a été identifiée.

FS3 : Chiffrement des messages et des pièces jointes

FS3 assure que deux utilisateurs puissent échanger des messages de façon confidentielle. FS3 garantit qu'un attaquant contrôlant totalement les serveurs d'Olvid n'est pas en mesure de lire ou modifier les messages échangés entre deux utilisateurs d'Olvid.

Chaque message est chiffré avec le chiffrement authentifié présenté dans Chiffrement authentifié page 17. Les clés générées spécifiquement pour chaque message sont à leur tour chiffrées via du chiffrement asymétrique puis ajoutées dans l'entête du message. Le schéma suivant présente le fonctionnement du chiffrement d'un message protocolaire.

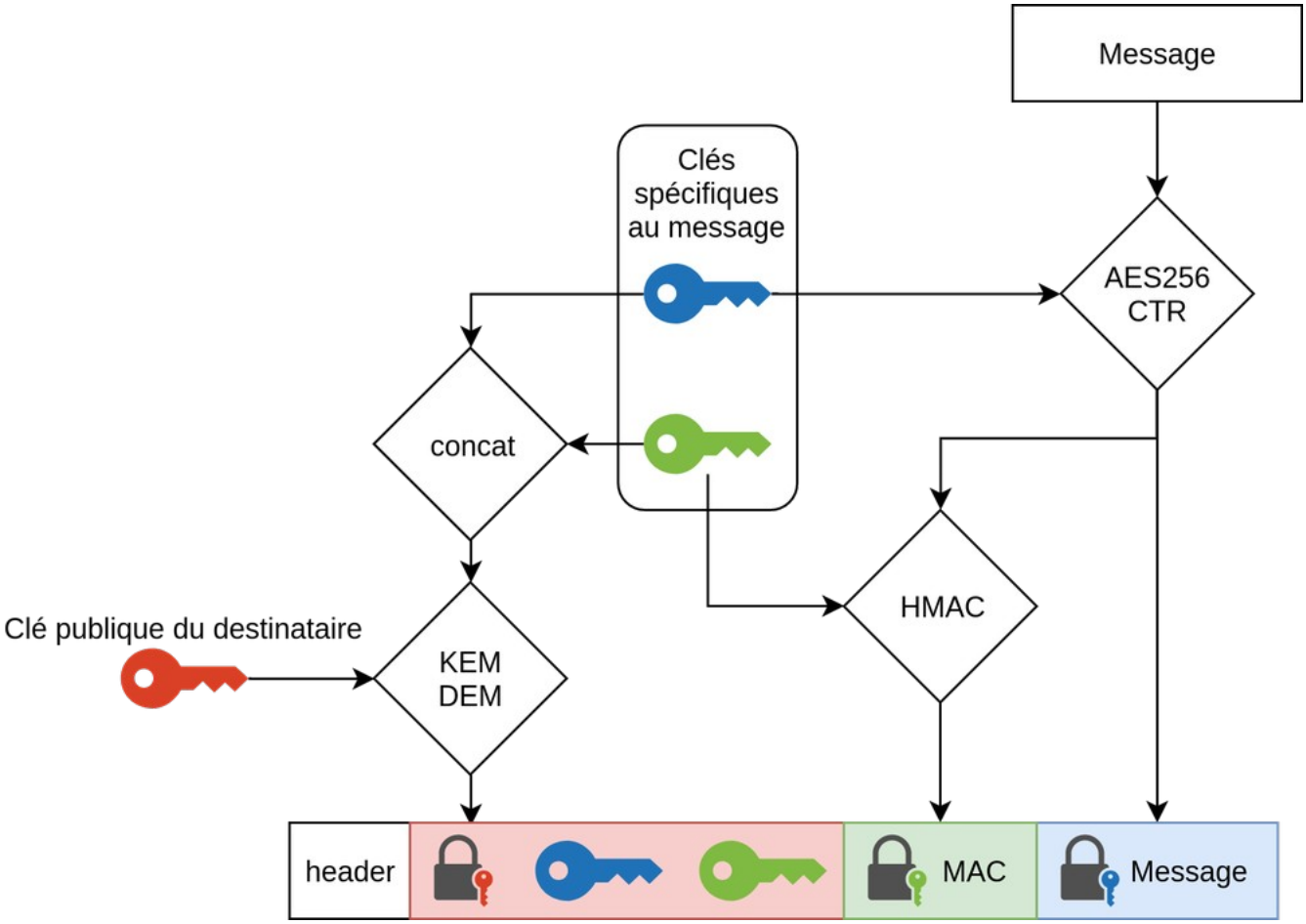

Illustration 19: Chiffrement des messages

Méthodologie

L'analyse de cette fonction de sécurité est réalisée en deux parties :

- Analyse statique du code source de l'application ;
- Analyse dynamique via la capture et le déchiffrement de messages protocolaire.

L'architecture de cette fonction de sécurité est documentée dans le chapitre *23 Encryption* de **[DOC_CRYPTO]**.

Analyse statique

Le chiffrement symétrique du corps du message est réalisé via l'interface *io.olvid.engine.crypto.AuthEnc* dans *io.olvid.engine.channel.datatypes.NetworkChannel#post*.

```
public static UID post(ChannelManagerSession channelManagerSession, ChannelMessageToSend
message, PRNGService prng) throws Exception {
    [...]
    NetworkChannel[] networkChannels = acceptableChannelsForPosting(channelManagerSession,
message);
    [...]
    AuthEnc authEnc = Suite.getDefaultAuthEnc(suiteVersion);
    AuthEncKey messageKey = authEnc.generateKey(prng);
    MessageToSend.Header[] headers = new MessageToSend.Header[networkChannels.length];
    boolean partOfFullRatchetProtocol = (message instanceof ChannelProtocolMessageToSend)
&& ((ChannelProtocolMessageToSend) message).isPartOfFullRatchetProtocolOfTheSendSeed();
    for (int i=0; i<networkChannels.length; i++) {
        headers[i] = networkChannels[i].wrapMessageKey(messageKey, prng,
partOfFullRatchetProtocol);
    }
    [...]
    switch (message.getMessageType()) {
        case MessageType.APPLICATION_MESSAGE_TYPE:
            if (!(message instanceof ChannelApplicationMessageToSend)) {
                Logger.w("Trying to post a message of type " + message.getMessageType() + "
that is not a ChannelApplicationMessageToSend.");
                throw new Exception();
            }
            ChannelApplicationMessageToSend channelApplicationMessageToSend =
(ChannelApplicationMessageToSend) message;
            ChannelApplicationMessageToSend.Attachment[] attachments =
channelApplicationMessageToSend.getAttachments();
            Encoded[] listOfEncodedAttachments = new Encoded[attachments.length + 1];
            MessageToSend.Attachment[] messageToSendAttachments = new
MessageToSend.Attachment[attachments.length];
            for (int i=0; i<attachments.length; i++) {
                AuthEncKey attachmentKey = authEnc.generateKey(prng);
                listOfEncodedAttachments[i] = Encoded.of(new Encoded[]{
                        Encoded.of(attachmentKey),
                        Encoded.of(attachments[i].getMetadata())
                });
                messageToSendAttachments[i] = new
MessageToSend.Attachment(attachments[i].getUrl(), attachments[i].isDeleteAfterSend(),
attachments[i].getAttachmentLength(), attachmentKey);
            }
            // add the message payload after the attachment keys and metadata
            listOfEncodedAttachments[attachments.length] =
Encoded.of(channelApplicationMessageToSend.getMessagePayload());
            Encoded plaintextContent = Encoded.of(new Encoded[]{
                    Encoded.of(MessageType.APPLICATION_MESSAGE_TYPE),
                    Encoded.of(listOfEncodedAttachments)
            });
            EncryptedBytes encryptedContent = authEnc.encrypt(messageKey,
plaintextContent.getBytes(), prng);
            messageToSend = new
MessageToSend(message.getSendChannelInfo().getFromIdentity(), messageUid, server,
encryptedContent, headers, messageToSendAttachments,
channelApplicationMessageToSend.hasUserContent(),
channelApplicationMessageToSend.isVoipMessage());
            break;
}
```

Une clé est générée pour chiffrer le message via un appel a la fonction *io.olvid.engine.crypto.AuthEnc#generateKey*. Puis une clé temporaire est créée via la même fonction pour chaque pièce jointe.

Une seule classe implémente cette interface : *io.olvid.engine.crypto.AuthEncAES256ThenSHA256*.

```
public AuthEncKey generateKey(PRNG prng) {
    KDF kdf = Suite.getKDF(KDF.KDF_SHA256);
    Seed kdfSeed = new Seed(prng);
    try {
        return (AuthEncKey) kdf.gen(kdfSeed, getKDFDelegate())[0];
    } catch (Exception e) {
        return null;
    }
}
```

Cette clé est utilisée pour chiffrer le corps du message. Le chiffrement authentifié utilisé est celui présenté dans Chiffrement authentifié page 30.

La clé utilisée pour le chiffrement authentifié est ensuite elle-même chiffrée via un appel à *io.olvid.engine.channel.datatypes.NetworkChannel#wrapMessageKey*. Les messages applicatifs sont envoyés uniquement via des canaux symétriques *io.olvid.engine.channel.databases.ObliviousChannel.* Cette contrainte est appliquée via les fonctions de choix de canal par type de messages :

```
public ChannelApplicationMessageToSend(Identity[] toIdentities, Identity fromIdentity,
byte[] messagePayload, Attachment[] attachments, boolean hasUserContent, boolean
isVoipMessage) throws Exception {
    SendChannelInfo[] sendChannelInfos =
SendChannelInfo.createAllConfirmedObliviousChannelsInfosForMultipleIdentities(toIdentities,
fromIdentity);
    if (sendChannelInfos.length != 1) {
        Logger.e("Error: trying to create a ChannelApplicationMessageToSend for identities
on different servers");
        throw new Exception();
    }
    this.sendChannelInfo = sendChannelInfos[0];
    [...]
}
```

La fonction implémentant *io.olvid.engine.channel.datatypes.NetworkChannel#wrapMessageKey* est donc *io.olvid.engine.channel.databases.ObliviousChannel#wrapMessageKey*.

```
public MessageToSend.Header wrapMessageKey(AuthEncKey messageKey, PRNGService prng, boolean
partOfFullRatchetProtocol) {
    RatchetingOutput ratchetingOutput = selfRatchet();
    if (ratchetingOutput == null) {
        return null;
    }
    AuthEnc authEnc = Suite.getAuthEnc(ratchetingOutput.getAuthEncKey());
    EncryptedBytes encryptedMessageKey;
    try {
        encryptedMessageKey = authEnc.encrypt(ratchetingOutput.getAuthEncKey(),
Encoded.of(messageKey).getBytes(), prng);
    } catch (InvalidKeyException e) {
        e.printStackTrace();
        return null;
    }
    byte[] headerBytes = new byte[KeyId.KEYID_LENGTH + encryptedMessageKey.length];
    System.arraycopy(ratchetingOutput.getKeyId().getBytes(), 0, headerBytes, 0,
KeyId.KEYID_LENGTH);
```

```
    System.arraycopy(encryptedMessageKey.getBytes(), 0, headerBytes, KeyId.KEYID_LENGTH,
encryptedMessageKey.length);
    MessageToSend.Header header = new MessageToSend.Header(remoteDeviceUid, remoteIdentity,
new EncryptedBytes(headerBytes));
    [...]
    return header;
}
```

Les pièces jointes sont chiffrées via le même mécanisme lors de leurs envois sur le serveur via la classe *io.olvid.engine.networksend.operations.UploadAttachmentOperation*. Les clés temporaires sont stockées dans le message lui-même chiffré.

```
public class UploadAttachmentOperation extends PriorityOperation {
    [...]

    @Override
    public void doExecute() {
        [...]
        Chunk attachmentChunk = new Chunk(outboxAttachment.getAcknowledgedChunkCount(),
Arrays.copyOfRange(buffer, 0, bufferFullness));
        final EncryptedBytes encryptedAttachmentChunk =
authEnc.encrypt(outboxAttachment.getKey(), attachmentChunk.encode().getBytes(), prng)
        [...]
    }

    [...]
}
```

Le chiffrement utilise un mécanisme de cliquet pour assurer une propriété de *forward secrecy*.

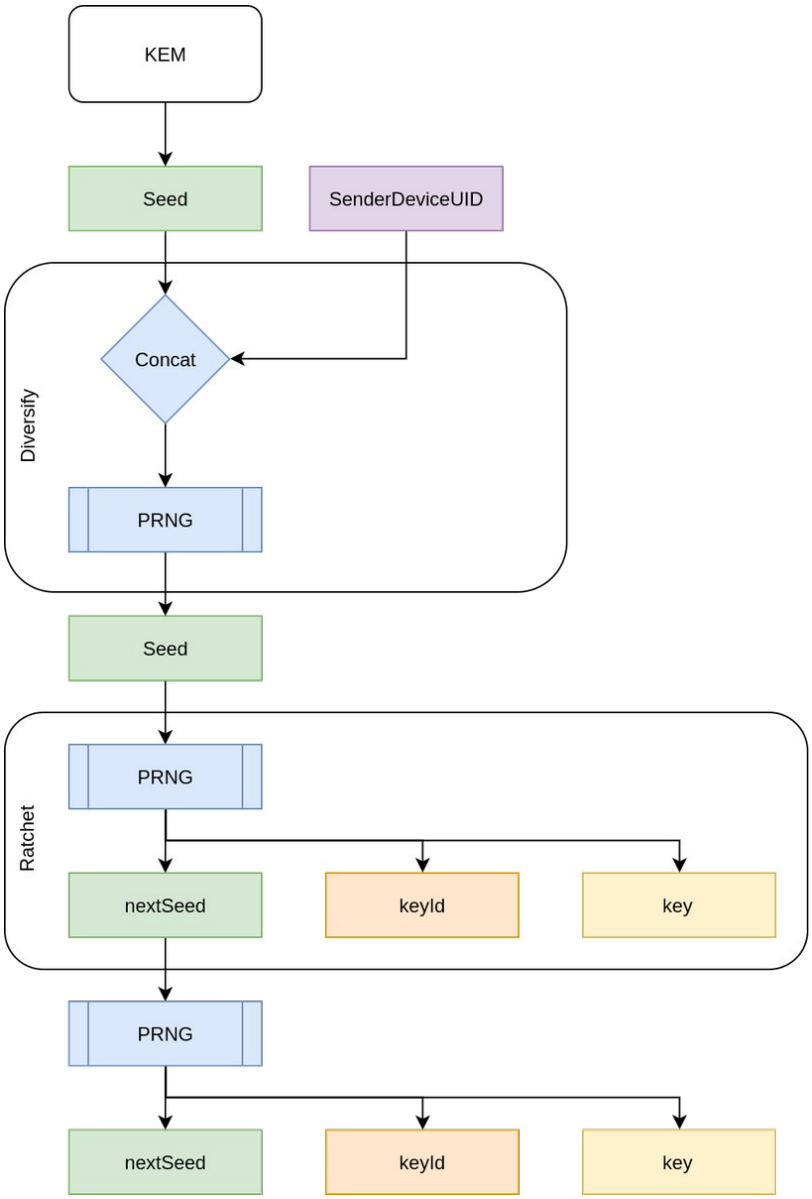

Illustration 20: Mécanisme de cliquet

Le cliquet est avancé à chaque envoi de message via la fonction *io.olvid.engine.channel.databases.ObliviousChannel#selfRatchet*.

```
private RatchetingOutput selfRatchet() {
    RatchetingOutput ratchetingOutput =
ObliviousChannel.computeSelfRatchet(seedForNextSendKey, obliviousEngineVersion);
    if (ratchetingOutput == null) {
        return null;
    }
    try (PreparedStatement statement =
channelManagerSession.session.prepareStatement("UPDATE " + TABLE_NAME + " SET " +
            SEED_FOR_NEXT_SEND_KEY + " = ? " +
            " WHERE " + CURRENT_DEVICE_UID + " = ? AND " + REMOTE_DEVICE_UID + " = ? AND "
+ REMOTE_IDENTITY + " = ?;")) {
```

```
        statement.setBytes(1, ratchetingOutput.getRatchetedSeed().getBytes());
        statement.setBytes(2, currentDeviceUid.getBytes());
        statement.setBytes(3, remoteDeviceUid.getBytes());
        statement.setBytes(4, remoteIdentity.getBytes());
        statement.executeUpdate();
        this.seedForNextSendKey = ratchetingOutput.getRatchetedSeed();
    } catch (SQLException e) {
        e.printStackTrace();
        return null;
    }
    return ratchetingOutput;
}
```

Le calcul du cliquet est réalisé dans *io.olvid.engine.channel.databases.ObliviousChannel#computeSelfRatchet*.

```
static RatchetingOutput computeSelfRatchet(Seed seed, int obliviousEngineVersion) {
    PRNG prng = Suite.getDefaultPRNG(obliviousEngineVersion, seed);
    Seed ratchetedSeed = new Seed(prng);
    KeyId keyId = new KeyId(prng.bytes(KeyId.KEYID_LENGTH));
    AuthEncKey authEncKey;
    KDF kdf = Suite.getDefaultKDF(obliviousEngineVersion);
    Seed kdfSeed = new Seed(prng);
    try {
        authEncKey = (AuthEncKey) kdf.gen(kdfSeed,
Suite.getDefaultAuthEnc(obliviousEngineVersion).getKDFDelegate())[0];
    } catch (Exception e) {
        return null;
    }
    return new RatchetingOutput(ratchetedSeed, keyId, authEncKey);
}
```

Cette fonction utilise la *seed* générée avec le dernier cliquet pour alimenter un PRNG. La sortie de ce PRNG est utilisée pour alimenter la fonction de dérivation de clé décrite dans **[DOC_CRYPTO]**. Ainsi que générer la *seed* pour la prochaine itération du cliquet.

Le cliquet ainsi généré est utilisé pour chiffrer la clé via le chiffrement authentifié déjà analysé dans Chiffrement authentifié page 17 et pour le chiffrement du corps des messages.

[DOC_CRYPTO] précise qu'un mécanisme de cliquet complet doit être utilisé à l'initialisation du canal symétrique ainsi que tous les 100 messages ou toutes les semaines.

La nécessité de renouvellement du cliquet complet est déterminé par *io.olvid.engine.channel.databases.ObliviousChannel#requiresFullRatchet* lors de la persistance de l'état du canal symétrique en base de donnée.

```
public void wasCommitted() {
    if ((commitHookBits & HOOK_BIT_MIGHT_NEED_FULL_RATCHET) != 0) {
        if (requiresFullRatchet()) {
            if (channelManagerSession.fullRatchetProtocolStarterDelegate != null) {
                try {

channelManagerSession.fullRatchetProtocolStarterDelegate.startFullRatchetProtocolForOblivio
usChannel(currentDeviceUid, remoteDeviceUid, remoteIdentity);
                } catch (Exception e) {
                    // no need to do anything, the next message will try to restart the
full ratchet
                    e.printStackTrace();
                }
            } else {
```

```
                Logger.w("Full ratchet required, but no FullRatchetProtocolStarterDelegate
is set.");
            }
        }
    }
    [...]
}
```

Le cliquet complet est bien requis tous les 100 messages ou toutes les semaines :

```
public boolean requiresFullRatchet() {
    if (fullRatchetOfTheSendSeedInProgress) {
        // 1. If we received too many messages since the last full ratchet protocol message
that we sent,
        // it means that the other end of the channel will probably never send an answer to
our last protocol message.
        // In that case, we decide to start the full ratchet protocol all over again.
        if (numberOfDecryptedMessagesSinceLastFullRatchetSentMessage >=
Constants.THRESHOLD_NUMBER_OF_DECRYPTED_MESSAGES_SINCE_LAST_FULL_RATCHET_SENT_MESSAGE) {
            return true;
        }
        // 2. If too much time passed since the time we sent a message related to the full
ratcheting protocol in progress,
        // we decide to start the protocol all over again.
        if (System.currentTimeMillis() - timestampOfLastFullRatchetSentMessage >=
Constants.THRESHOLD_TIME_INTERVAL_SINCE_LAST_FULL_RATCHET_SENT_MESSAGE) {
            return true;
        }
        // 3. If the number of messages sent since the last sent message related to the
full ratcheting protocol
        // is larger than the reprovisioning threshold, we must restart the protocol since
the recipient could end up
        // not being able to decrypt an old message arriving after the end of the full
ratcheting.
        if (numberOfEncryptedMessagesSinceLastFullRatchetSentMessage >=
Constants.REPROVISIONING_THRESHOLD) {
            return true;
        }
    } else {
        // 1. If the number of encrypted messages since the last successful full ratchet is
too high,
        // we must start a new full ratchet
        if (getNumberOfEncryptedMessagesSinceLastFullRatchet() >=
Constants.THRESHOLD_NUMBER_OF_ENCRYPTED_MESSAGES_PER_FULL_RATCHET) {
            return true;
        }
        // 2. If the elapsed time since the last successful full ratchet is too high,
        // we must start a new full ratchet
        if (System.currentTimeMillis() - timestampOfLastFullRatchet >=
Constants.FULL_RATCHET_TIME_INTERVAL_VALIDITY) {
            return true;
        }
    }
    return false;
}
```

Ces valeurs peuvent être plus courtes (2 heures ou 20 messages) dans le cas particulier ou un cliquet complet est déjà en cours. Lorsque les conditions sont réunies le protocole *io.olvid.engine.protocol.protocols.FullRatchetProtocol* est démarré via l'envoi du message *io.olvid.engine.protocol.protocols.FullRatchetProtocol.InitialMessage*.

```
public void startFullRatchetProtocolForObliviousChannel(UID currentDeviceUid, UID
remoteDeviceUid, Identity remoteIdentity) throws Exception {
      Logger.i("Full ratchet is currently deactivated");
    try (ProtocolManagerSession protocolManagerSession = getSession()) {
        Identity ownedIdentity =
identityDelegate.getOwnedIdentityForDeviceUid(protocolManagerSession.session,
currentDeviceUid);
        UID protocolInstanceUid = FullRatchetProtocol.computeProtocolUid(ownedIdentity,
remoteIdentity, currentDeviceUid, remoteDeviceUid);
        CoreProtocolMessage coreProtocolMessage = new
CoreProtocolMessage(SendChannelInfo.createLocalChannelInfo(ownedIdentity),
                ConcreteProtocol.FULL_RATCHET_PROTOCOL_ID,
                protocolInstanceUid,
                true);
        ChannelMessageToSend message = new
FullRatchetProtocol.InitialMessage(coreProtocolMessage, remoteIdentity,
remoteDeviceUid).generateChannelProtocolMessageToSend();
        protocolManagerSession.channelDelegate.post(protocolManagerSession.session,
message, prng);
        protocolManagerSession.session.commit();
    }
}
```

Le protocole suit ensuite les enchaînements d'états via les étapes suivantes :

```
AliceSendEphemeralKeyStep(InitialProtocolState, InitialMessage) ->
AliceEphemeralKeyMessage: AliceWaitingForK1State
BobSendEphemeralKeyAndK1Step(InitialProtocolState, AliceEphemeralKeyMessage) ->
BobEphemeralKeyAndK1Message: CancelledState, BobWaitingForK2State
AliceRecoverK1AndSendK2Step(AliceWaitingForK1State, BobEphemeralKeyAndK1Message) ->
AliceK2Message: CancelledState, AliceWaitingForAckState
BobRecoverK2ToUpdateReceiveSeedAndSendAckStep(BobWaitingForK2State, AliceK2Message) ->
BobAckMessage: CancelledState, FullRatchetDoneState
AliceUpdateSendSeedStep(AliceWaitingForAckState, BobAckMessage) -> FullRatchetDoneState
```

Cette implémentation est conforme à la description donnée dans le chapitre *32 – Full Ratchet Protocol* de **[DOC_CRYPTO]**.

Lors de l'étape *AliceEphemeralKeyMessage*, Alice génère une clé éphémère :

```
public ConcreteProtocolState executeStep() throws Exception {
    ProtocolManagerSession protocolManagerSession = getProtocolManagerSession();
    // generate a random 5-byte nonce as the heqavy weight bits of the restart counter
    //   --> this prevents reusing a message from an old run of the protocol in a newer run
(this is required because the prtocolUid is deterministic)
    byte[] bytes = getPrng().bytes(5);
    long restartCounter = 0;
    for (int i=0; i<5; i++) {
        restartCounter = restartCounter << 8;
        restartCounter += bytes[i] & 0xff;
    }
    restartCounter = restartCounter << 23; // the MSb is 0, 40 bits of nonce, 23 bits for
the actual restartCounter
      Logger.e("Alice restart counter: " + restartCounter);
    KeyPair keyPair =
Suite.generateEncryptionKeyPair(getOwnedIdentity().getEncryptionPublicKey().getAlgorithmImp
lementation(), getPrng());
    if (keyPair == null) {
        throw new Exception();
    }
```

```
    CoreProtocolMessage coreProtocolMessage = new
CoreProtocolMessage(SendChannelInfo.createObliviousChannelInfo(receivedMessage.contactIdent
ity, getOwnedIdentity(), new UID[]{receivedMessage.contactDeviceUid}, true),
getProtocolId(), getProtocolInstanceUid(), true);
    ChannelMessageToSend messageToSend = new AliceEphemeralKeyMessage(coreProtocolMessage,
(EncryptionPublicKey) keyPair.getPublicKey(),
restartCounter).generateChannelProtocolMessageToSend();
    protocolManagerSession.channelDelegate.post(protocolManagerSession.session,
messageToSend, getPrng());
    return new AliceWaitingForK1State(receivedMessage.contactIdentity,
receivedMessage.contactDeviceUid, (EncryptionPrivateKey) keyPair.getPrivateKey(),
restartCounter);
}
```

La clé est générée en utilisant le même mécanisme que celui analysé dans Établissement d'un secret partagé page 46.

L'étape suivante consiste pour Bob à générer lui aussi une clé éphémère ainsi qu'à créer un message KEM avec la clé éphémère d'Alice :

```
public ConcreteProtocolState executeStep() throws Exception {
    [...]
    KeyPair keyPair =
Suite.generateEncryptionKeyPair(getOwnedIdentity().getEncryptionPublicKey().getAlgorithmImp
lementation(), getPrng());
    if (keyPair == null) {
        throw new Exception();
    }
    // compute k1
    PublicKeyEncryption publicKeyEncryption =
Suite.getPublicKeyEncryption(receivedMessage.contactEphemeralPublicKey);
    CiphertextAndKey ciphertextAndKey =
publicKeyEncryption.kemEncrypt(receivedMessage.contactEphemeralPublicKey, getPrng());
    AuthEncKey k1 = ciphertextAndKey.getKey();
    EncryptedBytes c1 = ciphertextAndKey.getCiphertext();
    CoreProtocolMessage coreProtocolMessage =
buildCoreProtocolMessage(SendChannelInfo.createObliviousChannelInfo(receivedMessage.getRece
ptionChannelInfo().getRemoteIdentity(), getOwnedIdentity(), new UID[]
{receivedMessage.getReceptionChannelInfo().getRemoteDeviceUid()}, true));
    ChannelMessageToSend messageToSend = new
BobEphemeralKeyAndK1Message(coreProtocolMessage, (EncryptionPublicKey)
keyPair.getPublicKey(), c1,
receivedMessage.restartCounter).generateChannelProtocolMessageToSend();
    protocolManagerSession.channelDelegate.post(protocolManagerSession.session,
messageToSend, getPrng());
    return new
BobWaitingForK2State(receivedMessage.getReceptionChannelInfo().getRemoteIdentity(),
receivedMessage.getReceptionChannelInfo().getRemoteDeviceUid(), (EncryptionPrivateKey)
keyPair.getPrivateKey(), k1, receivedMessage.restartCounter);
}
```

L'étape suivante consiste pour Alice à déchiffrer le KEM de Bob et à générer elle-même un KEM avec la clé éphémère de Bob.

```
public ConcreteProtocolState executeStep() throws Exception {
    [...]
    // recover k1
    PublicKeyEncryption publicKeyEncryption =
Suite.getPublicKeyEncryption(startState.ephemeralPrivateKey);
    AuthEncKey k1 = publicKeyEncryption.kemDecrypt(startState.ephemeralPrivateKey,
receivedMessage.c1);
```

```
    if (k1 == null) {
        Logger.e("Could not recover k1.");
        return new CancelledState();
    }
    // compute k2
    publicKeyEncryption =
Suite.getPublicKeyEncryption(receivedMessage.contactEphemeralPublicKey);
    // TODO: maybe include entropy from the channel send seed (see this with Michel
Abdalla)
    CiphertextAndKey ciphertextAndKey =
publicKeyEncryption.kemEncrypt(receivedMessage.contactEphemeralPublicKey, getPrng());
    AuthEncKey k2 = ciphertextAndKey.getKey();
    EncryptedBytes c2 = ciphertextAndKey.getCiphertext();
    Seed seed = Seed.of(k1, k2);
    CoreProtocolMessage coreProtocolMessage = new
CoreProtocolMessage(SendChannelInfo.createObliviousChannelInfo(startState.contactIdentity,
getOwnedIdentity(), new UID[]{startState.contactDeviceUid}, true), getProtocolId(),
getProtocolInstanceUid(), true);
    ChannelMessageToSend messageToSend = new AliceK2Message(coreProtocolMessage, c2,
startState.restartCounter).generateChannelProtocolMessageToSend();
    protocolManagerSession.channelDelegate.post(protocolManagerSession.session,
messageToSend, getPrng());
    return new AliceWaitingForAckState(startState.contactIdentity,
startState.contactDeviceUid, seed, startState.restartCounter);
}
```

Enfin, Bob peut déchiffrer le KEM envoyé par Alice.

```
public ConcreteProtocolState executeStep() throws Exception {
    [...]
    // recover k2
    PublicKeyEncryption publicKeyEncryption =
Suite.getPublicKeyEncryption(startState.ephemeralPrivateKey);
    AuthEncKey k2 = publicKeyEncryption.kemDecrypt(startState.ephemeralPrivateKey,
receivedMessage.c2);
    if (k2 == null) {
        Logger.e("Could not recover k2.");
        return new CancelledState();
    }
    Seed seed = Seed.of(startState.k1, k2);

protocolManagerSession.channelDelegate.updateObliviousChannelReceiveSeed(protocolManagerSes
sion.session, getOwnedIdentity(), startState.contactDeviceUid, startState.contactIdentity,
seed, 0); // TODO: use the actual engine ver
    CoreProtocolMessage coreProtocolMessage =
buildCoreProtocolMessage(SendChannelInfo.createObliviousChannelInfo(startState.contactIdent
ity, getOwnedIdentity(), new UID[]{startState.contactDeviceUid}, true));
    ChannelMessageToSend messageToSend = new BobAckMessage(coreProtocolMessage,
startState.restartCounter).generateChannelProtocolMessageToSend();
    protocolManagerSession.channelDelegate.post(protocolManagerSession.session,
messageToSend, getPrng());
    return new FullRatchetDoneState();
}
```

Les deux participants ont alors obtenu deux *seed* dont eux seuls ont la connaissance. Ces *seed* sont utilisées pour mettre à jour les deux directions du canal symétrique.

Analyse dynamique

Analyse dynamique du chiffrement des messages protocolaires

La capture suivante présente un message tel que reçu sur le serveur.

```
header: [238f881d0634e1d66593ce6f860eadd28c2a5b6946dc1a590648bc8bbf3186e2,
349b794e9c93d92ad6dbc8c34342403bab0f10a9c03d59295f73364e2dcbaf980728828fcd19486bb161b1c58ad
9004ae28f887c32d2d610d245adfe7c6b97754b08c7ca7d649f8c9e617301e1d87e411a93dab2b04d29677da422
8fccd82fee5f3dd45e2ede6acfaf0b6764ecf0776c4a2ee2f6b4f513aa2006f52741ac184d6e7e06caface7e04b
4889bbff20b0db75e0d29bc7cad22ca9747adab0296fc2a8807f41af65a9bc093d77802c3a080ec62b94454be02
323969,
68747470733a2f2f7365727665722e6f6c7669642e696f0000bfe8eb98a77f8e738496ad620284fccb746b51ff6
745f512f0cada3e2c05f2c801372f70dcd058f6380cb949b007a089148d2c00bff25687aee1330069e8444c94]
Message:
efaf07f1fbc9f31d3a1b149a5436c1bcd901e491cf907a4b2514892ec56d84a8c1af40d5125b304fe0c95495d3b
8b9bd6041140cd959175688f716e66e4d8eedf4d07772eab3eaceeb44453f5723ba2b75a6ccc9a1e9554dac6033
5580bf58dce8f4cd8b15d0a2101cf7e72bcc899d3c54169cbd6790b1c489085d5c587619b09d2100add18b18a7b
8f494b4da155a55aa28eea89652ede3654b03ddf66a0a86ad783dd189705c79
```

Le message est décomposable en deux parties, un entête contenant les ID des destinataires et un corps chiffré.

À titre d'information l'entropie du corps du message est de 6.91 bit/octet. L'entropie est calculée comme dans la section Chiffrement asymétrique page 54, sur les octets du corps chiffré présenté dans l'extrait précédent.

Il est possible de confirmer que les messages sont correctement chiffrés en récupérant les clés privées des participants directement depuis la base de données *SQLite* de l'application et en réimplémentant le déchiffrement. Le code réalisant cette opération est disponible en annexe.

Pour l'exemple ci-dessus, on obtient donc les clés associées au message.

```
Symetric key: <SymmetricKey algoClassByteId=2 algoImplemByteId=0 dict={'mackey':
af231151402d97bb52a0a04cc551ca5145427fd5fd55c72b8c6bc13061506172, 'enckey':
b91afa465120d8467fa4fab4ad0e2d5c82718033e6876cffab3fcc74c964ddb5}>
```

Il est ensuite possible de déchiffrer le corps du message.

```
Protocol Message
  ChannelCreationWithContactDevice
    K2
      c2:            117a30addad745d980ba9434780b8570663167ecbdeef017a7cbbbd2b65df3e2
```

Analyse dynamique du chiffrement des messages applicatifs

La capture suivante présente un message applicatif tel que reçu sur le serveur.

```
header: [238f881d0634e1d66593ce6f860eadd28c2a5b6946dc1a590648bc8bbf3186e2,
08bf4ea3e8d3eb3ab83b55bcdcaabf40f9e857d2055ae323dd98614b326177f7f23dd8a4263afa0e6de8b59865b
058ac349f17d9b0f4a164e6a803035216515c23109ff559cbf98fcd8b1dd4995bddc7ef39e73c093d84ddd4b554
346cb4cad8f06fca0b1240b622baa6153fef08fa840423edbd01fd79e56ee18220bb04f1409101b75d652f3ad7d
7dcf5ffe944fdf8d88a68ff1e8289ae292f0d00e5cfaf84e835ef6f2a1a662308b3a35d638f20e7166cc20c0b41
6a32a8,
68747470733a2f2f7365727665722e6f6c7669642e696f0000bfe8eb98a77f8e738496ad620284fccb746b51ff6
745f512f0cada3e2c05f2c801372f70dcd058f6380cb949b007a089148d2c00bff25687aee1330069e8444c94]
Message:
e4f4be95325ece22be01ffa6c1197e7ef459e8891f405a40f8cc274b86013269478d1a73388c7c3bdd3186c8c6b
d6769892342f4703ebca85e516d08e754950e0be5bc6a7fbec38e0c300983cae7827ff2bc158cc036a81c8e894c
3b791613accd27e3f5edd7bdc9b39b3d3b98044697a059377fb4f9f93e1136ddeb65299284b364ed176aa6c4b19
b1409a3bccaa30011bb4baa57b6abd7c911fab8f5276d7ed45681701001b80ddcc8aec17545d26e7970e89397ad
9aaa5d00d05886bec7d51261a91f8ab1b25a3279b51239059e6e2acfbd4f681cf342e929c929d349fdee0339aea
5e8e6ee5c65ae3e49a1e95845b097bd0e96200b6c8424b253cd09dcc5f5069fee8c7da67bbf396add40af31559c
```

```
8e1c76a2813970941215b58e784434ea999980dd0a9042eec2298385b35accb3c077c7558c245565e48986155e6
d203f275afe54f1b05d4d7ff92dd592f730d15448044116a2fee8cde2d7891baf60fd475e26
```

Le message est décomposable en deux parties, un entête contenant les ID des destinataires et un corps chiffré.

À titre d'information l'entropie du corps du message est de 7.41 bit/octet. L'entropie est calculée comme dans la section Chiffrement asymétrique page 54, sur les octets du corps chiffré présenté dans l'extrait précédent.

Il est possible de confirmer que le message est correctement chiffré en récupérant la clé secrète de l'un des participants directement depuis la base de données *SQLite* de l'application et en réimplémentant le déchiffrement. Le code réalisant cette opération est disponible en annexe.

Pour l'exemple ci-dessus, on obtient donc les clés associées au message.

```
Symetric key: <SymmetricKey algoClassByteId=2 algoImplemByteId=0 dict={'mackey':
f3afb2013b7d55fd79af6faafe5b0bf867a6b67b0746dcd246a096bca6424cda, 'enckey':
588bb0c9e89b898f3a04ddcdc623878578b8e7658b7bf2601df65acead385be2}>
```

Il est ensuite possible de déchiffrer le corps du message.

```
{
  "message": {
    "body": "Hello Bob",
    "ssn": 1,
    "sti": "7fa0ceb3-adf2-4f51-92b1-25baf338fd9f"
  },
  "rr": {
    "key": "kAAAAGwAAAAAAgIABAAAAGAAAAAABm1hY2tleQAAAAAgMn+QRqLDH5ygi5Ygk90C/
58vhzjoK5sxApB+57zwCbwAAAAABmVuY2tleQAAAAAg5CmnsW8bacFhq2qrSSTcQeHrtS1IQpY7f8jVsIQT+r4=",
    "nonce": "UzxhAYRW93MvQbKqDH47CQ=="
  }
}
```

Conclusion

La fonctionnalité de chiffrement des messages et des pièces jointes repose sur des implémentations spécifiques de primitives cryptographiques standards ainsi que sur deux protocoles d'échange de clés.

Le design des fonctionnalités cryptographiques ayant déjà été analysé dans Analyse de la spécification cryptographique page 15, il était nécessaire de vérifier que les implémentations sont correctes.

L'audit a permis de confirmer le bon fonctionnement de la fonctionnalité sur plusieurs points :

- L'utilisation de primitives cryptographiques testées et éprouvées ;
- Une réimplémentation de la fonctionnalité à partir de briques logicielles différentes donnant les mêmes résultats que l'application ;
- L'analyse exhaustive du code source lié à l'envoi et au chiffrement de messages et pièces jointes.

En conséquence, la fonction de sécurité **[FS3]** ne présente pas de faiblesse quant à la menace **[M2]** et empêche un attaquant d'intercepter et de modifier les messages échangés.

L'implémentation de la fonction de sécurité est conforme au RGS et à l'état de l'art. Aucune vulnérabilité de conception ou d'implémentation n'a été identifiée.

FS4 : Chiffrement des sauvegardes du carnet de contact

FS4 assure qu'un attaquant en mesure de récupérer un fichier de sauvegarde du carnet de contact ne pourra pas le déchiffrer pour récupérer son contenu.

Le design de cette fonction de sécurité est documenté page 18 dans **[CIBLE]** ainsi que dans le chapitre *VI Keys and Contacts Backup* de **[DOC_CRYPTO]**.

Il est précisé qu'un fichier de sauvegarde est un export JSON de l'identité (cryptographique) de l'utilisateur et de son fichier de contacts, compressé puis chiffré via ECIES par des clés dérivées de la *passphrase* donnée par l'application à l'utilisateur.

Méthodologie

Pour vérifier que l'application implémente correctement le chiffrement décrit dans **[CIBLE]** et **[DOC_CRYPTO]** et s'assurer que le design ne présente pas de faiblesse non identifiée dans Analyse de la spécification cryptographique page 15 l'analyse a été menée en suivant deux approches complémentaires :

- Par l'étude du code source des fonctions d'export et d'importation de fichier de sauvegarde ;
- Par la réimplémentation du déchiffrement hors-ligne d'un fichier de sauvegarde.

L'objectif est de contrôler que l'implémentation du chiffrement est conforme et qu'un attaquant ne peut pas obtenir un clair. L'ensemble des scripts issus de ces travaux est disponible dans les annexes.

Contenu de la sauvegarde

Une sauvegarde est sous forme de *Json* sérialisé puis compressé via **java.util.zip** (**DeflaterOutputStream/InflaterInputStream** et *zlib* sans headers). L'application Olvid ne sauvegarde que l'identité cryptographique des contacts et des groupes auxquels l'utilisateur appartient (les messages ne sont pas sauvegardés) comme le montre l'exemple de sauvegarde déchiffrée et mise en forme ci-dessous :


```
{
    "backup_json_version": 0,
    "backup_timestamp": 1234567890,
    "engine": {
        "identity_manager": [
            {
                "api_key": "12345678-1234-1234-1234-123456789ABC",
                "contact_identities": [
                    {
                        "contact_groups": [],
                        "contact_identity": "aHR0cHM [redacted] 4LFWqnVwJIv4S/",
                        "trust_level": "4.0",
                        "trust_origins": [
                            {
                                "timestamp": 1234567890,
                                "trust_type": 0
                            }
                        ],
                        "trusted_details": {
                            "serialized_details": "{\"first_name\":\"BOB\"}",
                            "version": 1
                        }
                    },
                    {
                        "contact_groups": [],
                        "contact_identity": "aHR0cHM[redacted] sgqqClSfylFx52Q",
                        "trust_level": "4.0",
                        "trust_origins": [
```

```
                            {
                                "timestamp": 1234567890,
                                "trust_type": 0
                            }
                        ],
                        "trusted_details": {
                            "serialized_details": {
                                  "first_name": "Charlie"
                            },
                            "version": 1
                        }
                    }
                ],
                "owned_groups": [],
                "owned_identity": "aHR0cHM6 [redacted] OhZsF+Y0g==",
                "private_identity": {
                    "encryption_private_key": "kg [redacted] vQy",
                    "mac_key": "kAA [redacted] v5A=",
                    "server_authentication_private_key": "kg [redacted] rPtG"
                },
                "published_details": {

                    "serialized_details": {
                            "first_name": "Alice"
                    },
                    "version": 1
                }
            }
        ]
    }
}
```

Génération de la graine

L'ensemble du matériel cryptographique est dérivé à partir d'une graine unique de 160bits (*backupseed*). Cette graine est générée à partir du PRNG global (validé dans Analyse de la spécification cryptographique page 15) et de la fonction suivante :

```
// io.olvid.engine.datatypes.BackupSeed#generate
public static BackupSeed generate(PRNG prng) {
    try {
        return new BackupSeed(prng.bytes(BACKUP_SEED_LENGTH /*20*/));
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

La graine obtenue est encodée en *passphrase* lisible pour être sauvegardée par l'utilisateur, à partir d'un encodage en *base5*: chaque bloc de 5bits de la graine est converti en caractère dans l'alphabet "0123456789ABCDEFGHJKLMNPQRTUVWXY". L'alphabet exclue les lettres 'I', 'O' et 'S' pour éviter la confusion avec les caractères 1', '0' et '5'.

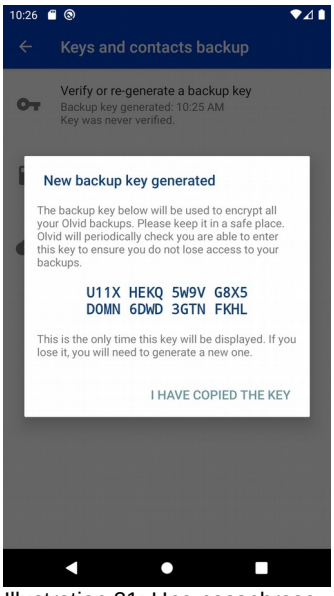

Illustration 21: Une passphrase encodée à partir d'une graine

Dérivation de la graine

La graîne générée permet de dériver un ensemble de clés pour le (dé)chiffrement des données de backup. Dans un premier temps, la graine est étendue de 160 à 256 bits (bits nuls) et permet d'initialiser un PRNG (**HMAC-SHA256 DRBG**).

Le PRNG ainsi initialisé est utilisé pour générer les éléments suivants :

- un **UID** (backupKeyUid) de 256bits pour identifier la clé dans les requêtes à la base de données locale ou le cloud ;
- une paire de clés (privée et publique) pour l'**ECIES** sur la courbe **Curve25519** : utilisé pour encapsuler la clé AES256 utilisée pour le chiffrement du fichier ;
- une clé pour un **HMAC-SHA256** (256bits): permet d'appliquer un mac sur le fichier chiffré afin de garantir son intégrité et empêcher un attaquant de forger un fichier de sauvegarde sans connaissance de la *passphrase* ;

La dérivation complète est effectuée dans la fonction suivante :

```
// io.olvid.engine.datatypes.BackupSeed#deriveKeys
public DerivedKeys deriveKeys() {
    // 1: initialize PRNG
    byte[] fullSeedBytes = new byte[32];
    System.arraycopy(backupSeedBytes, 0, fullSeedBytes, 0, BACKUP_SEED_LENGTH);
    PRNG prng = Suite.getPRNG(PRNG.PRNG_HMAC_SHA256, new Seed(fullSeedBytes));

    // 2: generate crypto material
    UID backupKeyUid = new UID(prng);
    EncryptionEciesCurve25519KeyPair encryptionKeyPair =
EncryptionEciesCurve25519KeyPair.generate(prng);
    MAC mac = Suite.getMAC(MAC.HMAC_SHA256);
    if (mac == null) {
        return null;
    }
    MACKey macKey = mac.generateKey(prng);
    return new DerivedKeys(backupKeyUid, encryptionKeyPair, macKey);
}
```

Pour chiffrer, comme pour déchiffrer le message, les clés sont dérivées de la graine. Le schéma suivant montre la génération ainsi que la dérivation de la graine :

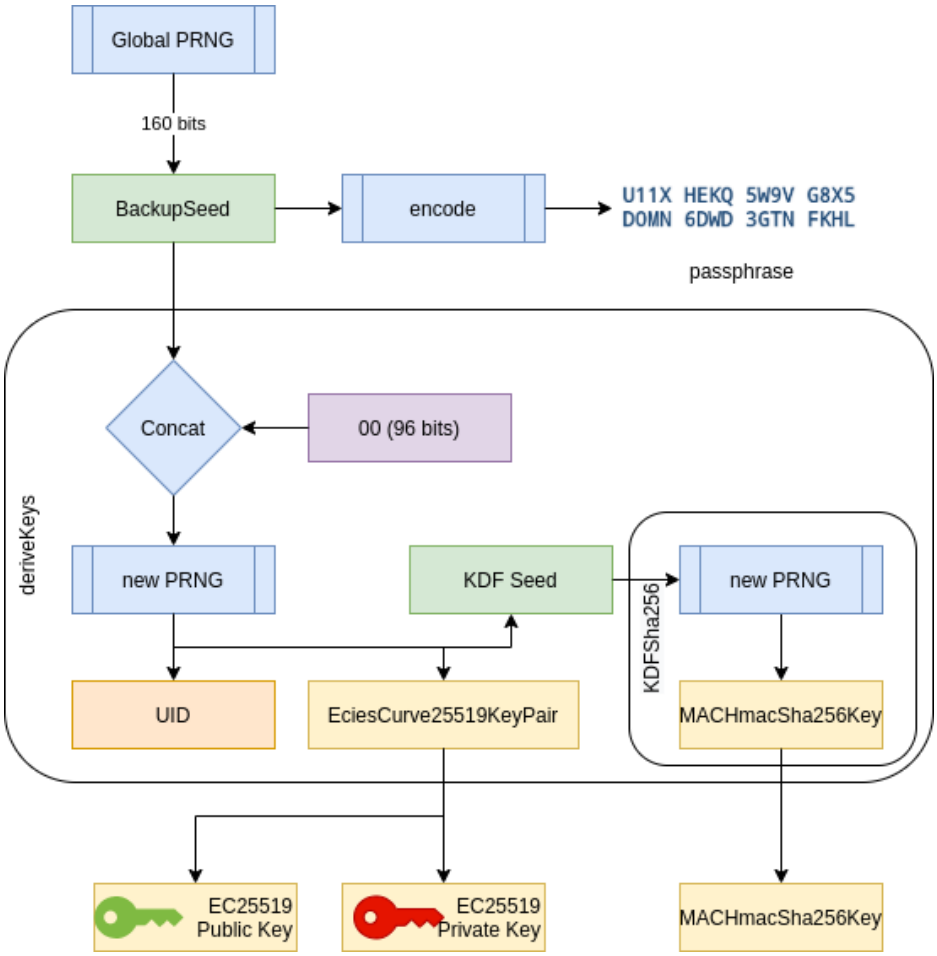

Illustration 22: Dérivation des clés à partir de la graine

Chiffrement de la sauvegarde

Le chiffrement d'une sauvegarde est effectué en quatre étapes:

- L'application génère une graine et dérive les clés de backup (ou prend les clés de backup existantes dans sa base de données);
- Le mécanisme d'encapsulation de clés (**KEM**, décrit dans Key encapsulation mechanism (KEM) page 31) génère des clés de session AES, MAC, ainsi qu'un séquestre (KEM ciphertext);
- Le mécanisme d'encapsulation de données (**DEM**, décrit dans Data Encapsulation Mecanism (DEM) page 32) utilise les clés de session pour chiffrer les données de *backup* pré-compressées et générer un MAC;
- Les éléments générés précédemment sont écrits dans le fichier final, et un second HMAC est appliqué sur l'intégralité;

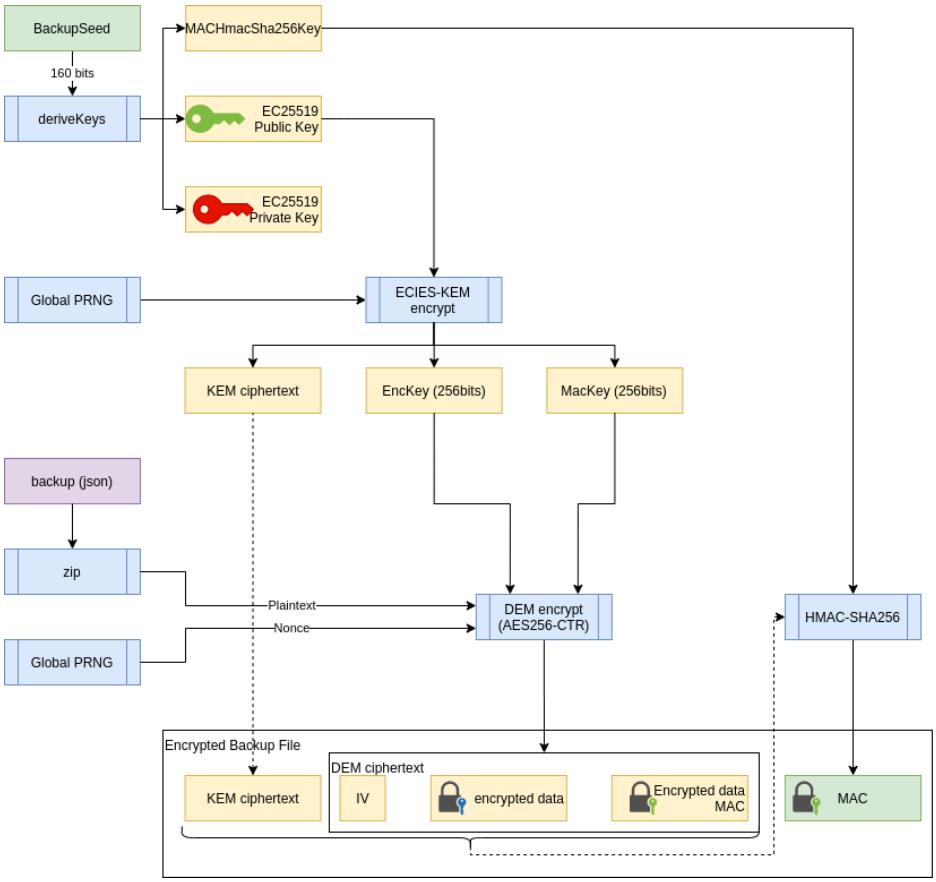

Illustration 23: Mécanisme complet de chiffrement d'une sauvegarde

Dans l'application, ces étapes sont effectuées lorsque les données de backup sont prêtes à être chiffrées :

```
// io.olvid.engine.backup.BackupManager#backupSuccess
public void backupSuccess(String tag, UID backupKeyUid, int version, String backupContent)
{
    /*redacted:
          - get Backup key from backupKeyUid
          - compress backupContent using DeflaterOutputStream
    */
    // KEM encryption + DEM encryption
    EncryptedBytes encryptedBackup =
Suite.getPublicKeyEncryption(encryptionPublicKey).encrypt(encryptionPublicKey,
compressedBackup, prng);
    byte[] mac = Suite.getMAC(macKey).digest(macKey, encryptedBackup.getBytes());

    // Final HMAC
    byte[] macedEncryptedBackup = new byte[encryptedBackup.getBytes().length + mac.length];
    System.arraycopy(encryptedBackup.getBytes(), 0, macedEncryptedBackup, 0,
encryptedBackup.getBytes().length);
    System.arraycopy(mac, 0, macedEncryptedBackup, encryptedBackup.getBytes().length,
mac.length);

    backup.setReady(macedEncryptedBackup);

    // redacted: notify for export to file or cloud
}
```

```
// io.olvid.engine.crypto.PublicKeyEncryptionEcies#encrypt
// kem is defined as KemEcies256Kem512
// dem is defined as AuthEncAES256ThenSHA256
public EncryptedBytes encrypt(EncryptionPublicKey publicKey, byte[] plaintext, PRNGService
prng) throws InvalidKeyException {
    byte[] ciphertextBytes = new byte[kem.ciphertextLength() +
dem.ciphertextLengthFromPlaintextLength(plaintext.length)];

    CiphertextAndKey ciphertextAndKey = kem.encrypt(publicKey, prng);
    System.arraycopy(ciphertextAndKey.getCiphertext().getBytes(), 0, ciphertextBytes, 0,
kem.ciphertextLength());

    EncryptedBytes demCiphertext = dem.encrypt(ciphertextAndKey.getKey(), plaintext, prng);
    System.arraycopy(demCiphertext.getBytes(), 0, ciphertextBytes, kem.ciphertextLength(),
dem.ciphertextLengthFromPlaintextLength(plaintext.length));

    return new EncryptedBytes(ciphertextBytes);
}
```

KEM

Une clé éphémère K_u est générée à partir du PRNG. Un secret partagé est calculé à partir de la clé éphémère privée et la clé publique de l'utilisateur. Ce secret partagé servira à dériver le matériel cryptographique de (dé)chiffrement. La clé éphémère publique est ajoutée dans le fichier chiffré afin de calculer le même secret partagé à partir de la clé privée de l'utilisateur.

Le schéma ci-dessous résume les étapes nécessaires au partage du secret :

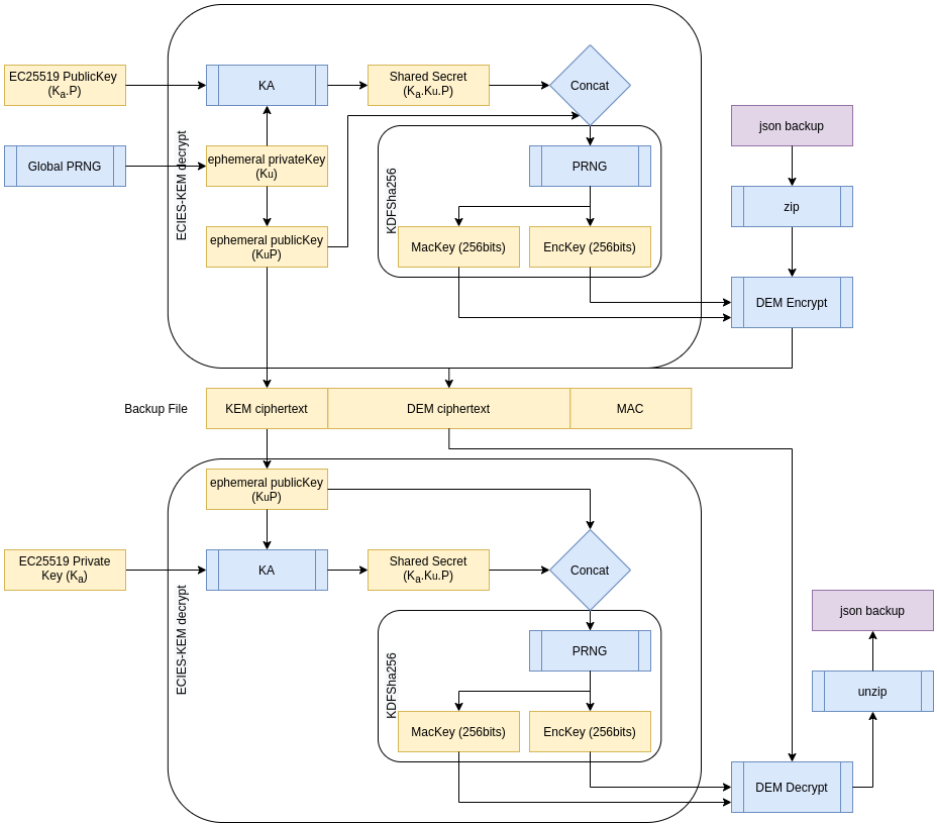

Illustration 24: Schéma de (dé)chiffrement IECS-KEM implémenté par Olvid

Conformité de l'implémentation

- L'encodage de la graine en passphrase, et l'opération inverse n'engendrent pas de biais dans le sens où il s'agit d'une conversion de base pour rendre la graine plus lisible;
- Les propriétés du PRNG utilisé garantissent qu'un attaquant qui ne dispose pas de la graine n'aurait pas de meilleure attaque que le *bruteforce* de cette dernière (160bits) pour retrouver les mêmes clés de chiffrement ;
- Les propriétés du HMAC empêchent un attaquant de forger un fichier de sauvegarde sans connaissance de la *passphrase* (graine);
- Les propriétés ECIES-KEM (clé publique éphémère) et la dérivation du matériel cryptographique à partir d'une graine rendent plus difficile une attaque cryptographique à partir d'une clé publique (non présente dans le fichier);
- Pour un attaquant qui aurait récupéré un fichier de sauvegarde, le recouvrement des données chiffrées nécessiterait :
    - la connaissance de la graine (ou la *passphrase*) sur 160bits pour recalculer les secrets cryptographiques dérivés (attaque par force brute);
    - avoir une faiblesse dans le PRNG (HMAC256-DRBG) permettant de retrouver un état interne à partir du vecteur d'initialisation (64bits) ou de la clé HMAC bruteforcée (256bits);
    - avoir une faiblesse dans AES256-CTR en ayant peu ou pas d'informations sur la clé.

La fonctionnalité de chiffrement du fichier de sauvegarde repose d'un côté sur des API éprouvées et fournies par Android (AES256-CTR, HMAC256, zlib) et de l'autre sur des implémentations spécifiques d'algorithmes connus (courbes 25519, KEM, HMAC-DRBG).

Lors de l'audit plusieurs points vont dans le sens d'une implémentation conforme, en particulier :

- L'utilisation de vecteurs de tests standards par l'éditeur de l'application pour tester le code ;
- Une réimplémentation de la fonctionnalité à partir de briques logicielles différentes donnant les mêmes résultats que l'application ;
- L'analyse exhaustive du code source lié à la sauvegarde et importation du carnet de contact.

En conséquence, la fonction de sécurité **[FS4]** ne présente pas de faiblesse quant aux menaces **[M2]** et **[M4]**, en empêchant un attaquant qui aurait récupéré un fichier de sauvegarde d'exploiter ce dernier. **L'implémentation de la fonction de sécurité est conforme à la spécification et à l'état de l'art. Aucune vulnérabilité de conception ou d'implémentation n'a été identifiée.**

FS5 : Chiffrement des appels VoIp

Introduction

Olvid permet de passer des appels entre deux clients. L'initiation d'un appel est une fonctionnalité payante ou gratuite pour une période d'essai de 30 jours. Les tests ont été réalisés sur la version gratuite avec période d'essai.

Pour passer des appels entre utilisateurs, Olvid se base sur la bibliothèque open source **WebRTC.** Le code source de l'application à auditer est livré avec une version pré-compilée de WebRTC sous la forme d'une bibliothèque : *libjingle_peerconnection_so.so*, version 4324 (correspondant selon [WEBRTC_VERSION] à la version Chrome 88).

WebRTC se base sur une série de protocoles :

- Traversal Using Relays around NAT (TURN): protocole permettant de mettre en relation deux pairs via un serveur central ;
- Interactive Connectivity Establishment (ICE) ;
- Session Description Protocol (SDP) : information sur les capacités actuelles d'un *endpoint* (candidats ICE, codecs audio/video supportés, fingerprint de certificat, ...) ;
- Datagram Transport Layer Security (DTLS) et Secure Real-time Transport Protocol (SRTP) : canal de donnée chiffré WebRTC basé sur OpenSSL.

La connexion sécurisée et l'échange des données VoIP est entièrement effectuée par WebRTC. La sécurité du canal repose sur l'échange d'un descripteur SDP (Session Description Protocol) qui contient entre autres l'empreinte du certificat du pair à authentifier. Cette information doit être transmise sur un canal de signalisation sécurisé. WebRTC laisse à la charge du développeur de mettre en place son propre canal de signalisation.

Analyse statique

Canal de signalisation

Le canal de signalisation est basé sur le canal authentifié classique (**ObliviousChannel**) d'Olvid documenté en FS3 : Chiffrement des messages et des pièces jointes page 58 et considéré comme fiable.

Les messages sont gérés par l'applicatif Java:

- **réception:** messages reçus par *io.olvid.messenger.App#handleWebrtcMessage* et dispatchés au service io.olvid.messenger.webrtc.WebrtcCallService ;
- **envoi**: les messages sont envoyés via *io.olvid.messenger.webrtc.WebrtcCallService#postMessage.*

Les messages sont des objets *JsonWebrtcMessage* ou *JsonWebrtcProtocolMessage* pour la (dé)sérialisation réseau.

	Message Type	JsonWebrtcMessage object	Message attributes	Sending method
0	START_CALL_MESSAGE_TYPE	JsonStartCallMessage	caller SDP type and value, turnUserName, turnPassword	WebrtcCallService.sendStartCallMessage
1	ANSWER_CALL_MESSAGE_TYPE	JsonAnswerCallMessage	responder SDP type and value	WebrtcCallService.sendAnswerCallMessage
2	REJECT_CALL_MESSAGE_TYPE	JsonRejectCallMessage		WebrtcCallService.sendRejectCallMessage
3	HANGED_UP_MESSAGE_TYPE	JsonHangedUpMessage		WebrtcCallService.sendHangedUpMessage
4	RINGING_MESSAGE_TYPE	JsonRingingMessage		WebrtcCallService.sendRingingMessage
5	BUSY_MESSAGE_TYPE	JsonBusyMessage		WebrtcCallService.sendBusyMessage
6	RECONNECT_CALL_MESSAGE_TYPE	JsonReconnectCallMessage	peer SDP type and value	WebrtcCallService.sendReconnectCallMessage

Ces messages sont liés à une machine à état pouvant avoir les états suivants :

```
// io.olvid.messenger.webrtc.WebrtcCallService.State
public enum State {
        INITIAL,
        WAITING_FOR_AUDIO_PERMISSION,
```

```
        GETTING_TURN_CREDENTIALS,
        INITIALIZING_CALL,
        START_CALL_MESSAGE_SENT,
        RINGING,
        BUSY,
        CALL_REJECTED,
        CONNECTING_TO_PEER,
        CALL_IN_PROGRESS,
        RECONNECTING,
        HANGED_UP,
        FAILED
    }
```

Initialisation de l'appel

L'appel est initialisé dans la fonction *io.olvid.messenger.webrtc.WebrtcCallService#callerStartCall* qui prend en paramètre l'identité cryptographique du compte utilisateur et du contact et les associe à une session (UUID aléatoire). Il initialise en suite une connexion (*io.olvid.messenger.webrtc.WebrtcPeerConnectionHolder#initializePeerConnectionFactory*) et une session TURN (getTurnCredentials).

Configuration TURN

La session TURN est initialisée à partir des *credentials* de l'utilisateur, permettant l'enregistrement auprès du serveur TURN :

```
// io.olvid.messenger.webrtc.WebrtcPeerConnectionHolder#getIceServer
private static PeerConnection.IceServer getIceServer(String username, String password) {
      List<String> servers;
      if (SettingsActivity.getScaledTurn()) {
          servers = turnScaledServers;
      } else {
          servers = turnServers;
      }
      return PeerConnection.IceServer.builder(servers)
              .setUsername(username)
              .setPassword(password)
              // Well, let's encrypt root is not correctly recognized by libwebrtc and
there is no way to override the root CAs, so we disable certificate check...
              .setTlsCertPolicy(PeerConnection.TlsCertPolicy.TLS_CERT_POLICY_INSECURE_NO_CH
ECK)
              .createIceServer();
}
```

La désactivation de la vérification du certificat du serveur n'a pas d'impact sur la confidentialité et l'authentification des appels puisque qu'un chiffrement de bout en bout est réalisé.

Configuration WebRTC

La connexion WebRTC est initialisée par l'appel à *WebrtcPeerConnectionHolder.createPeerConnection* et configurée au moyen de l'objet *PeerConnection.RTCConfiguration:*

```
//io.olvid.messenger.webrtc.WebrtcPeerConnectionHolder#createPeerConnection
void createPeerConnection() {
    PeerConnection.IceServer iceServer = getIceServer(turnUsername, turnPassword);
    if (iceServer == null) {

webrtcCallService.peerConnectionHolderFailed(FailReason.ICE_SERVER_CREDENTIALS_CREATION_ERR
OR);
        return;
    }
```

```
    PeerConnection.RTCConfiguration configuration = new
PeerConnection.RTCConfiguration(Collections.singletonList(iceServer));
    configuration.iceTransportsType = PeerConnection.IceTransportsType.RELAY;
    configuration.sdpSemantics = PeerConnection.SdpSemantics.UNIFIED_PLAN;

    peerConnection = peerConnectionFactory.createPeerConnection(configuration,
peerConnectionObserver);
}
```

Certains éléments ne sont pas configurés par l'application Olvid et gérés implicitement par la bibliothèque WebRTC:

- *configuration.keyType:* permet de spécifier le type de clé à utiliser (RSA ou ECDSA)
    - ECDSA (défaut), sur la courbe ANSI X9.62 Prime 256v1
    - RSA 1024 (option non configurée mais qui peut s'avérer dangereuse le cas échéant)
- *configuration.certificate:* le certificat d'authentification au format PEM à présenter à un pair. Dans le cas où le certificat n'est pas configuré, un nouveau certificat de session est généré à la volée par WebRTC

SDP Offer

L'offre SDP est obtenue via appel à **PeerConnection.createOffer.** La bibliothèque native WebRPC initialise la connexion et retourne l'offre SDP locale via la callback **onCreateSuccess**.

```
// io.olvid.messenger.webrtc.WebrtcPeerConnectionHolder#createOffer
void createOffer() {
    createAudioTrack(new MediaConstraints());
    createDataChannel();
//            Use CameraVideoCapturer
//            peerConnection.addTrack(peerConnectionFactory.createVideoTrack("video",
peerConnectionFactory.createVideoSource(false)));

    peerConnection.createOffer(sessionDescriptionObserver, new MediaConstraints());
}

//
io.olvid.messenger.webrtc.WebrtcPeerConnectionHolder.SessionDescriptionObserver#onCreateSuc
cess
private class SessionDescriptionObserver implements SdpObserver {
    @Override
    public void onCreateSuccess(SessionDescription sdp) {
//            Logger.e("Created " + sdp.type + "\n\n" + sdp.description);
        String filteredSdpDescription = filterSdpDescriptionCodec(sdp.description);
        peerConnection.setLocalDescription(this, new SessionDescription(sdp.type,
filteredSdpDescription));
    }

    // [redacted]
}
```

L'offre SDP subit alors un filtrage pour retirer les codecs indésirables. Seuls les codecs à bitrate constants (opus, PCMU, PCMA, telephone-event) sont conservés. Elle contient aussi le fingerprint du certificat de session généré par WebRTC. La description SDP est appliquée via l'appel à **PeerConnection.setLocalDescription** est appelé, qui déclenche la mise en place ICE de WebRTC.

Lorsque l'étape ICE est terminée, *io.olvid.messenger.webrtc.WebrtcPeerConnectionHolder#iceGatheringCompleted* est appelée. Olvid envoie alors sur le canal de signalisation un message de type START_CALL_MESSAGE_TYPE pour l'appel notifier à l'appelant.

SDP Answer

De son côté, l'appelé reçoit la notification d'appel et l'offre SDP. S'il choisit de répondre, Olvid initialise une connexion via io.olvid.messenger.webrtc.WebrtcCallService#recipientAnswerCallInternal. WebRTC génère alors une réponse SDP renvoyée à l'appelant dans un message de type ANSWER_CALL_MESSAGE_TYPE.

Côté appelant, l'appel à **PeerConnection.setRemoteDescription** permet d'appliquer les paramètres du client distant et d'initialiser la connexion WebRTC. Une session DTLS-SRTP est alors initialisée en prenant en compte les *fingerprint* de certificats des deux pairs, présents dans les descriptions SDP.

Certificat SDP

Si WebRTC n'est configuré avec aucun certificat, un certificat temporaire est généré par WebRTC dans *WebRtcSessionDescriptionFactory :*

```
WebRtcSessionDescriptionFactory::WebRtcSessionDescriptionFactory( /*[...]*/) {
if (certificate) {
    // Use |certificate|.
    //[...]
} else {
    // Generate certificate.
    certificate_request_state_ = CERTIFICATE_WAITING;

    rtc::scoped_refptr<WebRtcCertificateGeneratorCallback> callback(
        new rtc::RefCountedObject<WebRtcCertificateGeneratorCallback>());
    callback->SignalRequestFailed.connect(
        this, &WebRtcSessionDescriptionFactory::OnCertificateRequestFailed);
    callback->SignalCertificateReady.connect(
        this, &WebRtcSessionDescriptionFactory::SetCertificate);

    rtc::KeyParams key_params = rtc::KeyParams();
    RTC_LOG(LS_VERBOSE)
        << "DTLS-SRTP enabled; sending DTLS identity request (key type: "
        << key_params.type() << ").";

    // Request certificate. This happens asynchronously, so that the caller gets
    // a chance to connect to |SignalCertificateReady|.
    cert_generator_->GenerateCertificateAsync(key_params, absl::nullopt,
                                              callback);
  }
```

Le type de certificat généré est choisi par défaut, soit ECDSA :

```
enum KeyType { KT_RSA, KT_ECDSA, KT_LAST, KT_DEFAULT = KT_ECDSA };
```

La génération est effectuée via OpenSSL selon le chemin d'appel suivant :

- WebRtcSessionDescriptionFactory::WebRtcSessionDescriptionFactory
- RTCCertificateGenerator::GenerateCertificateAsync
- *RTCCertificateGenerator::GenerateCertificate*
- *SSLIdentity::Create*
- OpenSSLIdentity::CreateWithExpiration
- OpenSSLIdentity::CreateInternal

- OpenSSLKeyPair::Generate / OpenSSLCertificate::Generate
- rtc::MakeCertificate

Ce certificat sera utilisé pour authentifier un pair lors de l'établissement de la session DTLS-SRTP.

DTLS-SRTP

La session DTL-SRTP est initialisée via la fonction native *DtlsTransport::SetupDtls* qui initialise les paramètres cryptographiques utilisés.

```
bool DtlsTransport::SetupDtls() {
  RTC_DCHECK(dtls_role_);
  {
    auto downward = std::make_unique<StreamInterfaceChannel>(ice_transport_);
    StreamInterfaceChannel* downward_ptr = downward.get();

    dtls_ = rtc::SSLStreamAdapter::Create(std::move(downward));
    if (!dtls_) {
      RTC_LOG(LS_ERROR) << ToString() << ": Failed to create DTLS adapter.";
      return false;
    }
    downward_ = downward_ptr;
  }

  dtls_->SetIdentity(local_certificate_->identity()->Clone());
  dtls_->SetMode(rtc::SSL_MODE_DTLS);
  dtls_->SetMaxProtocolVersion(ssl_max_version_);
  dtls_->SetServerRole(*dtls_role_);
  dtls_->SignalEvent.connect(this, &DtlsTransport::OnDtlsEvent);
  dtls_->SignalSSLHandshakeError.connect(this,
                                         &DtlsTransport::OnDtlsHandshakeError);
  if (remote_fingerprint_value_.size() &&
      !dtls_->SetPeerCertificateDigest(
          remote_fingerprint_algorithm_,
          reinterpret_cast<unsigned char*>(remote_fingerprint_value_.data()),
          remote_fingerprint_value_.size())) {
    RTC_LOG(LS_ERROR) << ToString()
                      << ": Couldn't set DTLS certificate digest.";
    return false;
  }

  // Set up DTLS-SRTP, if it's been enabled.
  if (!srtp_ciphers_.empty()) {
    if (!dtls_->SetDtlsSrtpCryptoSuites(srtp_ciphers_)) {
      RTC_LOG(LS_ERROR) << ToString() << ": Couldn't set DTLS-SRTP ciphers.";
      return false;
    }
  } else {
    RTC_LOG(LS_INFO) << ToString() << ": Not using DTLS-SRTP.";
  }

  RTC_LOG(LS_INFO) << ToString() << ": DTLS setup complete.";
```

Ici, la suite cryptographique utilisée (**srtp_ciphers**_) est la suite AES_CM_128_HMAC_SHA1_80, utilisée par défaut (les autres suites ne sont pas activées, et activables par l'API uniquement).

Le certificat pair (**remote_fingerprint_value**_) est le *hash* du certificat distant, permettant une authentification du pair et fourni dans la description SDP reçue via le canal de signalisation.

Analyse Dynamique

Des tests dynamiques ont été effectués par instrumentation de l'application avec l''outil *Frida*. Ces tests consistent entre autres à capturer les messages sur le canal authentifié et les appels aux fonctions natives de la bibliothèque WebRTC pour valider la configuration cryptographique utilisée lors d'un appel.

Lors de l'initialisation d'un appel, WebRCT crée dans un premier temps un certificat de session:

```
[0][tid:    0] WebrtcCallService::callerSetTurnCredentialsAndInitializeCall, info={},
data=None
[38][tid: 6911] OpenSSLKeyPair::Generate, info={'KeyPair': {'type': 'EC', 'value': {'type':
'prime256v1 (P-256)', 'pub_key': {'x': [redacted], 'y': [redacted], 'z': [redacted]},
'priv_key': {'bn': [redacted], 'scalar': [redacted]}}}, 'params': {'type': 'ECDSA',
'ECCurve': 'EC_NIST_P256'}}, data=[redacted]
[39][tid: 6911] OpenSSLCertificate::Generate, info={'KeyPair': {'type': 'EC', 'value':
{'type': 'prime256v1 (P-256)', 'pub_key': {'x': [redacted], 'y': [redacted], 'z':
[redacted]}, 'priv_key': {'bn': [redacted], 'scalar': [redacted]}}}, 'params': {'CN':
'WebRTC', 'NB': '1616444177', 'NA': '1619122577', 'key_params': {'type': 'ECDSA',
'ECCurve': 'EC_NIST_P256'}}}, data=None
```

Ce certificat est utilisé pour construire descripteur SDP local avec son empreinte et les informations TURN/ICE:

```
[76][tid:    0] PeerConnection::setLocalDescription, info={'type': 'offer', 'description':
[redacted]}, data=None
[77][tid: 6913] PeerConnection::OnTransportControllerGatheringState, info={'new_state': 1},
data=None
[redacted]
[84][tid: 6911] OpenSSLAdapter::BeginSSL, info={'ssl_host_name': 'turn.olvid.io'},
data=None
[85][tid: 6911] SSL_CTX_set_verify, info={'mode': '0x1'}, data=None
[86][tid: 6911] SSL_CTX_set_cipher_list, info={'cipher_list': 'ALL:!SHA256:!SHA384:!aPSK:!
ECDSA+SHA1:!ADH:!LOW:!EXP:!MD5'}, data=None
[redacted]
[298][tid: 6913] PeerConnection::OnTransportControllerGatheringState, info={'new_state':
2 /*kIceGatheringComplete*/}, data=None
```

L'application émet alors au destinataire un message de type "rtc" (*JsonWebrtcMessage*). La capture ci-dessous montre les métadonnées qu'il contient :

- "ci": Call Identifier (uid) de l'émetteur ;
- "sd": le contenu SDP compressé et encodé en *Base64* ;
- "sdt": le type de SDP (ici offer) ;
- "tu" (TurnUSername"), "tp" (TurnPassword) et "ts" (TurnServers): les informations relatives au serveur TURN sur lequel se synchroniser.

```
[299][tid:    0] Engine::post, info={'messagePayload': '{"rtc":{"ci":"ffd78171-a1be-4808-
aa75-
7f3c15da56a2","mt":0,"smp":"{\\"sd\\":\\"[redacted]\\",\\"sdt\\":\\"offer\\",\\"tp\\":\\"[r
edacted]\\",\\"ts\\":[\\"turns:turn.olvid.io:5349?
transport=udp\\",\\"turns:turn.olvid.io:443?
transport=tcp\\"],\\"tu\\":\\"1616597692:recipient\\"}"}}'}, data=None
```

L'offre SDP décodée contient entre autres le *hash* du certificat WebRTC de l'émetteur :

```
[redacted]
c=IN IP4 35.180.159.115
a=rtcp:9 IN IP4 0.0.0.0
```

```
a=candidate:3884911299 1 udp 8331007 35.180.159.115 52103 typ relay raddr 0.0.0.0 rport 0
generation 0 network-id 3 network-cost 10
a=candidate:3884911299 1 udp 8331263 35.180.159.115 51571 typ relay raddr 0.0.0.0 rport 0
generation 0 network-id 3 network-cost 10
a=ice-ufrag:lXKy
a=ice-pwd:[redacted]
a=ice-options:trickle renomination
a=fingerprint:sha-256
A4:B1:8D:56:99:46:DC:4C:0D:44:D2:36:11:0D:F9:13:32:FE:CD:75:5D:3C:C1:AB:BC:04:E5:D1:76:CD:B
6:03
a=setup:actpass
a=mid:0
[redacted]
m=application 60644 UDP/DTLS/SCTP webrtc-datachannel
c=IN IP4 35.180.159.115
a=candidate:3884911299 1 udp 8331007 35.180.159.115 60644 typ relay raddr 0.0.0.0 rport 0
generation 0 network-id 3 network-cost 10
a=candidate:3884911299 1 udp 8331263 35.180.159.115 65337 typ relay raddr 0.0.0.0 rport 0
generation 0 network-id 3 network-cost 10
a=ice-ufrag:lXKy
a=ice-pwd:[redacted]
a=ice-options:trickle renomination
a=fingerprint:sha-256
A4:B1:8D:56:99:46:DC:4C:0D:44:D2:36:11:0D:F9:13:32:FE:CD:75:5D:3C:C1:AB:BC:04:E5:D1:76:CD:B
6:03
a=setup:actpass
a=mid:1
a=sctp-port:5000
a=max-message-size:262144
```

Une fois l'appel validé par le destinataire, ce dernier génère une réponse SDP qu'il transmet via le canal de signalisation et reçu par l'appelant comme le montre la capture ci-dessous :

```
[301][tid:    0] App::handleWebrtcMessage, info={'messagePayload':
'{"sd":"[redacted]","sdt":"answer"}'}, data=None
```

La réponse SDT contient aussi l'empreinte certificat du destinataire :

```
[redacted]
c=IN IP4 35.180.159.115
a=rtcp:9 IN IP4 0.0.0.0
a=candidate:3884911299 1 udp 8331263 35.180.159.115 60545 typ relay raddr 0.0.0.0 rport 0
generation 0 network-id 3 network-cost 10
a=candidate:3884911299 1 udp 8331007 35.180.159.115 52075 typ relay raddr 0.0.0.0 rport 0
generation 0 network-id 3 network-cost 10
a=ice-ufrag:POMV
a=ice-pwd:[redacted]
a=ice-options:trickle renomination
a=fingerprint:sha-256
49:1E:F2:94:58:3B:A8:FA:0E:57:F7:BD:9F:4E:0D:F5:9E:10:87:4B:B3:D2:C8:31:F3:40:1B:72:DB:EC:2
4:22
a=setup:active
a=mid:0
[redacted]
m=application 9 UDP/DTLS/SCTP webrtc-datachannel
c=IN IP4 0.0.0.0
a=ice-ufrag:POMV
a=ice-pwd:B9xnAAXd1TRd7Ix5wwmXtEZc
a=ice-options:trickle renomination
```

```
a=fingerprint:sha-256
49:1E:F2:94:58:3B:A8:FA:0E:57:F7:BD:9F:4E:0D:F5:9E:10:87:4B:B3:D2:C8:31:F3:40:1B:72:DB:EC:2
4:22
a=setup:active
a=mid:1
a=sctp-port:5000
a=max-message-size:262144
```

La capture des paramètres de la fonction native *DtlsTransport::SetupDtls* ainsi que l'initialisation SSL montre bien que le certificat de l'appelé est utilisé comme *fingerprint* pour DTLS-SRTP:

```
[313][tid: 6911] DtlsTransport::SetupDtls:, info={'fingerprint': '{}', 'srtp_ciphers_':
'{"1":"SRTP_AES128_CM_SHA1_80"}'}, data=b'I\x1e\xf2\x94X;\xa8\xfa\x0eW\xf7\xbd\x9fN\r\xf5\
x9e\x10\x87K\xb3\xd2\xc81\xf3@\x1br\xdb\xec$"'
[314][tid: 6911] OpenSSLStreamAdapter::SetDtlsSrtpCryptoSuites, info={'crypto_suites':
'{"1":"SRTP_AES128_CM_SHA1_80"}'}, data=None
[redacted]
[355][tid: 6911] SSL_CTX_set_verify, info={'mode': '0x3' /*SSL_VERIFY_PEER*/}, data=None
[356][tid: 6911] SSL_CTX_set_cipher_list, info={'cipher_list': 'DEFAULT:!NULL:!aNULL:!
SHA256:!SHA384:!aECDH:!AESGCM+AES256:!aPSK'}, data=None
[357][tid: 6911] ssl_server_handshake, info={'state': 0, 'value': 'state_start_connect'},
data=None
```

Securité protocolaire

Dans un récent papier (**[WEBRTC_VULN]**), plusieurs applications de messagerie instantanée utilisant WebRTC ont été étudiés. Il en ressort que pour certaines applications, l'API WebRTC n'est pas correctement utilisée et permet à un attaquant d'initier un appel et forcer la victime à décrocher sans intervention de l'utilisateur, transformant le smartphone en micro ambiant.

La vulnérabilité réside dans le fait que, dans certaines applications, le canal audio est activé avant l'échange SDP (via l'appel à PeerConnection.addTrack**)** et que l'origine des messages de signalisation n'est pas vérifiée. Un attaquant (appelant) peut envoyer des messages forgés de façon à faire croire qu'un message provient, en interne, de l'appelé et ainsi mettre à mal la machine a état.

Dans la totalité des cas recensées dans le papier, la vulnérabilité provient du fait les échanges SDP et la connexion pair à pair sont effectués avant interaction de l'utilisateur:

```
For example, some applications do not exchange any SDP until the callee user has interacted
with the application to answer the call, meanwhile others set up the peer-to-peer
connection, and start sending audio and video from caller to callee before the callee is
even notified of the call.
```

Dans l'application Olvid, la piste audio est créée dès la création de l'offre SDP :

```
// io.olvid.messenger.webrtc.WebrtcPeerConnectionHolder#createOffer
void createOffer() {
    createAudioTrack(new MediaConstraints());
    createDataChannel();
    peerConnection.createOffer(sessionDescriptionObserver, new MediaConstraints());
}

// io.olvid.messenger.webrtc.WebrtcPeerConnectionHolder#createAudioTrack
private void createAudioTrack(MediaConstraints mediaConstraints) {
    audioSource = peerConnectionFactory.createAudioSource(mediaConstraints);
    audioTrack = peerConnectionFactory.createAudioTrack("audio0", audioSource);
    audioTrack.setEnabled(!webrtcCallService.microphoneMuted);
    peerConnection.addTrack(audioTrack);
}
```

Toutefois, la réponse SDP n'est envoyée que si l'appelé décroche en validant l'appel. La connexion ne peut avoir lieu que dans ce cas et donc l'application ne se trouve pas impactée par ce type de vulnérabilité.

2.1.4.3. Avis d'expert et vulnérabilités potentielles identifiées

Les fonctions de sécurités sont conformes à leurs spécifications dans la cible.

2.1.5. Identification des vulnérabilités génériques

2.1.5.1. Référentiels utilisés pour l'analyse

Pour la recherche de vulnérabilités génériques, les actions suivantes ont été suivies:

- L'audit statique du code source des éléments non couverts par les fonctions de sécurité.
- La recherche dynamique de comportements suspects.
- Une nouvelle analyse des protocoles cryptographiques sous des hypothèses moins fortes, en particulier leur résistance face à des utilisateurs crédules et peu susceptibles de remarquer des comportements anormaux.

Il est important de préciser que l'étude des fonctions de sécurité a déjà permis de couvrir une grande partie de l'application. En effet, les hypothèses de **[CIBLE]** ne restreignent qu'assez peu la surface d'attaque exposée.

2.1.5.2. Avis d'expert et vulnérabilités potentielles identifiées

Aucune vulnérabilité générique n'a été découverte.

2.2. Analyse des vulnérabilités

Vulnérabilité	Exploitation
V-01 : USURPATION-TIERS	Exploitable
V-02 : CONFUSION-HOMONYMES	Non Exploitable

2.2.1. V-01 : USURPATION-TIERS

Le protocole *Trust Establishment Protocol with SAS*, lors de l'établissement d'une relation de confiance entre deux tiers annonce une probabilité de succès de Man-in-the-Middle imperceptible à 10^{-8}. Ce résultat, prouvé dans **[SAS_PROOF]**, se conçoit assez bien en considérant l'exemple suivant.

Eve, qui contrôle l'ensemble des communications en dehors du canal authentique, souhaite faire de l'interception active entre Alice et Bob lors de l'établissement du canal sécurisé entre ces derniers. Elle aimerait donc se présenter à Alice en tant que Bob en fournissant une fausse identité et inversement pour Bob. Pour cela, Eve joue l'ensemble du protocole avec Alice jusqu'à ce que le mobile d'Alice demande le SAS de Bob, et fait de même avec Bob jusqu'à ce que son mobile attende l'information d'Alice. L'objectif de Eve est qu'Alice et Bob tombent tout de même d'accord sur un *SAS* commun de 8 digits et que chacun échange un demi *SAS* à travers le canal authentique.

L'intérêt du protocole utilisé est de faire en sorte que le SAS généré, quand Eve mentant à Alice et Bob sur leurs identités respectives, ne soit pas égal pour les deux parties. Le SAS faisant 8 chiffres, on comprend alors que, sans supposer de faiblesse dans les primitives cryptographiques, Alice et Bob ont une "mal"chance sur de 10^8 de choisir indépendamment le même SAS et donc autant de probabilité que l'attaque de Eve fonctionne.

Cependant il faut aussi considérer le cas où, de manière fortuite, Alice générerait le même demi-SAS que celui que lui a envoyé Bob via le canal authentique, mais que l'autre demi-SAS, celui qu'elle envoie à Bob, ne corresponde pas. Dans ce cas, pour Alice le protocole peut être complété et la relation validée sans qu'elle ne se doute qu'elle échange avec Eve. Bob de son côté aura une erreur lors de la validation du SAS. Le scénario inverse où seul le demi-SAS envoyée par Alice est

validé chez Bob est symétrique. Bob se trouverait alors en relation avec Eve sans s'en rendre compte alors que Alice aura une erreur d'affichée. Sans biais cryptographique connu par Eve, chacun de ces deux cas a une probabilité de 10^{-4}.

On constate donc qu'un attaquant contrôlant les échanges aurait une chance d'usurper un des participants avec une probabilité d'un 1/10000. Bien que la probabilité reste relativement faible, cette attaque est à considérer dans le cas d'un serveur compromis et utilisé par un grand nombre de participants. L'attaque ne remet cependant pas en cause la probabilité de succès d'un Man-in-the-Middle à 10^{-8} annoncé par Olvid car en cas de succès ou d'échec, l'un des participants s'en rend compte. Elle ne remet non plus pas en cause **[SAS_PROOF]** qui identifie cette probabilité d'usurpation.

Il est aussi important de nuancer en précisant que dans un cas réel, Alice et Bob disposent d'un canal authentique (par exemple un appel téléphonique) qui pourrait servir au participant chez lequel la validation du SAS échoue de prévenir l'autre. De plus pour qu'un attaquant puisse mettre en place cette attaque d'usurpation, il est nécessaire de pouvoir intercepter et modifier le vecteur initial de mise en relation (en général un QR code échangé par e-mail ou SMS). Cela signifie que le contrôle du serveur d'Olvid ne suffirait pas.

Cette vulnérabilité est considérée comme **exploitable** mais ne l'a pas été durant l'audit en raison de sa complexité.

La cotation de la vulnérabilité suivant **[CEM]**:

Facteur	Valeur	Score
Temps mis pour l'exploitation	> 6 mois	19
Expertise de l'attaquant	Expert	6
Connaissance nécessaire à l'attaquant	Information publique	0
Accès au produit par l'attaquant	Difficile	10
Type d'équipement nécessaire pour exploiter la vulnérabilité	Standard	0

Somme des valeurs	Résistant à un attaquant ayant un potentiel d'attaque	Niveau de résistance des fonctions
35	Fort	Fort

2.2.2. V-02 : CONFUSION-HOMONYMES

L'application Olvid permet la mise en relation de contacts à travers un contact tiers commun. Cette fonctionnalité est décrite dans le chapitre 2.2.4. Mise en relation par un tiers de **[CIBLE]**.

Lors d'une mise en place d'une nouvelle relation par un contact de premier niveau, la procédure est automatique et il n'y a pas à accepter le nouveau contact. Par exemple, quand Alice déjà en lien avec Dave et Bob introduit Bob à Dave, Dave obtient automatiquement une nouvelle entrée dans ses contacts sans avoir à la valider. Toutefois, une notification système ainsi qu'une entrée dans la vue *Invitations* apparaissent pour expliquer la situation à Dave. Cela est le comportement normal de l'application et ne constitue pas une vulnérabilité.

Cependant, ce type d'utilisation pourrait prêter à une légère confusion lors l'introduction d'un contact homonyme. En effet, dans ce cas précis un utilisateur pourrait alors voir apparaître dans sa liste de contact deux entrées similaires.

La confusion est d'autant plus grande quand le code couleur des deux contacts se retrouve être le même.

La figure suivante illustre le problème. On retrouve deux contacts Bob dans la liste des contacts.

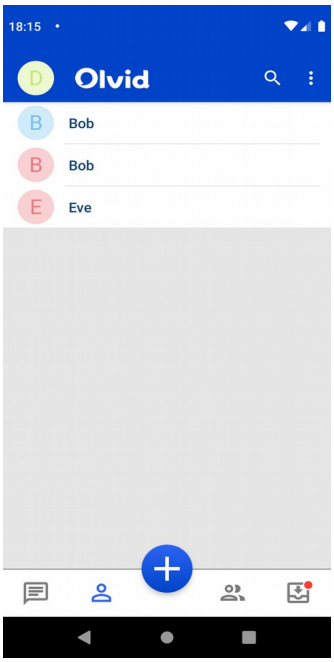

Illustration 25: Confusion d'homonyme dans la liste des

Le doute sur l'origine du contact homonyme peut être levé en regardant ses détails pour vérifier l'origine de la relation de confiance *Trust Origine.* De plus la notification dans l'onglet *Invitations* devrait attirer l'attention de l'utilisateur comme le montre la figure suivante:

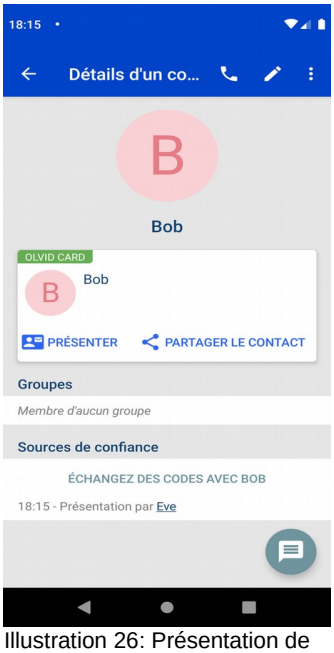

Illustration 26: Présentation de la source de confiance pour un

Il est ensuite possible de renommer le contact afin d'éviter toute confusion postérieure.

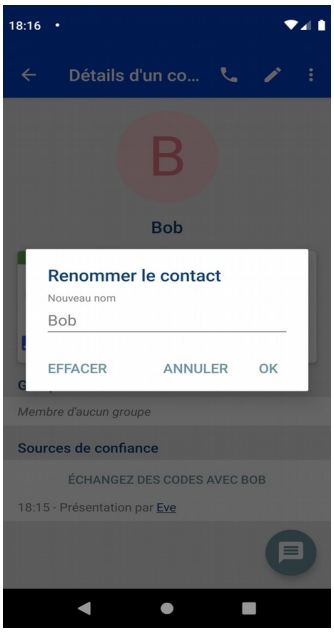

Illustration 27: Renommage d'un contact

Toute ambiguïté pourrait être définitivement levée si un pop-up supplémentaire était implémenté lors de l'ajout pars un tiers d'un contact homonyme.

L'avis du CESTI est que ce comportement n'est pas exploitable car non compatible avec l'hypothèse H2 de **[CIBLE]**, qui suppose les utilisateurs non hostiles. De plus il est probable qu'un utilisateur se rende compte d'une telle attaque. Enfin un correctif est simple à mettre en place.

Ce comportement est donc relevé à titre indicatif et ne relève pas d'une réelle vulnérabilité pour l'évaluation. Il ne fera donc pas l'objet d'une cotation.

3. Synthèse de l'évaluation

3.1. Synthèse de la sécurité du produit

L'application Olvid propose un modèle de messagerie sécurisée pour Android, basée sur l'échange de messages courts (SAS) et sans confiance accordée au serveur.

L'évaluation menée montre que le niveau de sécurité implémenté est élevé, avec un éditeur compétent sur les concepts cryptographiques manipulés. En l'occurrence aucune vulnérabilité critique n'a été découverte sur le périmètre défini dans **[CIBLE]**.

Le choix de l'éditeur d'implémenter lui-même des protocoles dédiés ainsi que certaines primitives cryptographiques (EdwardCurve, HMAC_DBRG, ECIES) était ambitieux. Mais force est de constater que le développement est de qualité. De plus, l'ajout du support des appels audio en se basant sur un standard de l'industrie semble être un choix pertinent. L'utilisation des protocoles d'échanges de messages présents dans l'application pour la mise en place d'une session audio permet de garantir la sécurité des appels.

Les quelques faiblesses identifiées mériteraient d'être corrigées ou du moins connues, mais elles ne réduisent pas sensiblement la sécurité de l'application et ne remettent pas en cause les garanties annoncées par le constructeur.

Malgré cela, il est important de comprendre que la sécurité cryptographique de la mise en relation de deux contacts ne pourra pas être supérieure à la taille du SAS échangé. Cela laisse à un attaquant disposant de beaucoup de moyens une probabilité de 10^{-8} pour réussir une attaque de Man-in-the-Middle et 10^{-4} pour une usurpation.

3.2. Durée des travaux

Phase	Durée des travaux (en jours*H)
Analyse du besoin et de l'environnement	1
Analyse de la mise en œuvre	2
Analyse de la conception/développement	10
Conformité et résistance - Analyse de la conformité, résistance et vulnérabilités	10
Conformité et résistance - Analyse de la cryptographie	5
Exploitation des résultats	2
Synthèse et rédaction du rapport	5
Total	35

3.3. Avis d'expert

En l'état l'application Olvid semble avoir le niveau de sécurité requis pour une CSPN et le CESTI émet un avis **FAVORABLE** quant à l'obtention de cette dernière.

4. Références

Sigle	Référence
[CEM]	Common Methodology for Information Technology Security Evaluation : Evaluation Methodology, version en vigueur.
[CSPN]	Certification de sécurité de premier niveau des produits des technologies de l'information, référence ANSSI-CSPN-CER-P-01, version en vigueur
[CRITERES]	Critères pour l'évaluation en vue d'une certification de sécurité de premier niveau, référence ANSSI-CSPN-CER-I-02, version en vigueur.
[DRBG_TESTVECTORS]	https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Algorithm-Validation-Program/documents/drbg/drbgtestvectors.zip
[NIST.SP.800-90Ar1]	https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90Ar1.pdf
[RGS_B]	Référentiel général de sécurité, annexes B : [RGS_B1] : Règles et recommandations concernant le choix et le dimensionnement des mécanismes cryptographiques. [RGS_B2] : Règles et recommandations concernant la gestion des clés utilisées dans des mécanismes cryptographiques.[RGS_B3] : Règles et recommandations concernant les mécanismes d'authentification.
[SecureRandom]	https://developer.android.com/reference/java/security/SecureRandom
[WEBRTC_VERSION]	version basée sur la version majeure de chrome et référencée dans https://chromiumdash.appspot.com/branches
[WEBRTC_VULN]	https://googleprojectzero.blogspot.com/2021/01/the-state-of-state-machines.html

5. Annexes

L'ensemble des développements, captures et vecteurs de tests utilisés dans la présente évaluation est disponible dans le fichier annexe.zip accompagnant ce rapport.

```
SHA256 (annexes.zip) = a3c5e935f830ca8eb53ae6dc4320d991ceb319d3d2d3322c8b5b510d4aa78d2a
```

Les fichiers présents sont les suivants :

```
annexes
├── BackupSeed.py
├── decrypt_backup.py
├── DerivedKeysForBackup.py
├── dummy_server
│   ├── ca.mk
│   ├── olvid.conf
│   ├── OlvidCrypto
│   ├── OlvidDummyClient
│   ├── OlvidDummyServer
│   ├── scripts
│   └── setup.py
├── fs4_backup
│   ├── __init__.py
│   ├── main.py
│   ├── olvid
│   └── test_prng.py
├── fs5_frida
│   ├── frida_loader.py
│   ├── libjingle.js
│   └── README.md
├── hmac_drbg.py
├── KEM.py
├── PRNG.py
├── Tests.py
├── TestVectorsBackupKeysFromBackupSeedString.json
├── TestVectorsBackupSeedFromString.json
├── TestVectorsPRNGGenBigInt.json
└── TestVectorsPRNGWithHMACWithSHA256.json

8 directories, 21 files
```

Ils sont regroupés en trois catégories :

- *dummy_server* : implémentation en python d'un serveur compatible avec les protocole Olvid, afin de pouvoir recevoir et tester le test de déchiffrement des messages
- *fs4_backup* : implémentation du recouvrement d'une sauvegarde (fonction de sécurité 4). Le script permet à partir d'un fichier *.olvidbackup* et de la passphrase associée, de valider le déchiffrement d'une sauvegarde et de l'afficher. Le sous dossier olvid contient l'ensemble des fonctionnalités cryptographiques réimplémentées en Python pour valider la cryptographie. Il est utilisé pour le déchiffrement des sauvegardes mais aussi pour valider le PRNG (HMAC_DRBG) à partir des vecteurs de test du NIST (**[DRBG_TESTVECTORS]**)
- *fs5_frida* : scripts frida permettant d'instrumenter l'application Olvid sur un téléphone ou un émulateur et tracer les appels aux fonctions Java ou Natives de WebRTC