SYNACKTIV
DIGITAL SECURITY

Rapport technique d'évaluation

OLVID

- DIFFUSION PUBLIQUE -

Version 1.2

Olvid

01/10/2020

Identification du document

Caractéristiques

Objet	Rapport technique d'évaluation - OLVID
Nombre de pages	86
Diffusion	DIFFUSION PUBLIQUE

Historique

Version	Date	État
1.0	09/06/2020	Première version
1.1	24/08/2020	Modifications suite à la FICHE DE REVUE DE RAPPORT du 3 août 2020- Référence : OLVID.01
1.2	01/10/2020	Suppression des numéros de mobile de l'équipe et passage en classification DIFFUSION PUBLIQUE

Équipe projet

Nom	Fonction	Coordonnées
Renaud Feil	Président	renaud.feil@synacktiv.com
Eloi Benoist-Vanderbeken	Coordinateur pôle ingénierie inverse Expert sécurité	eloi.benoist-vanderbeken@synacktiv.com
Tiphaine Romand-Latapie	Coordinatrice pôle ingénierie inverse Expert sécurité	tiphaine.romand-latapie@synacktiv.com
Thomas Etrillard	Responsable CSPN	thomas.etrillard@synacktiv.com
Paul Fariello	Expert sécurité	paul.fariello@synacktiv.com
Luca Moro	Expert sécurité	luca.moro@synacktiv.com

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
Nom de la cible d'évaluation	Olvid
N° de version analysée	Olvid en version 0.8.2 pour iOS de l'App Store au 27/04/2020
Correctifs éventuels appliqués	Aucun
Cible de sécurité	Cible de Sécurité CSPN - Olvid, version 2.2 du 05/04/2020
Domaine technique CSPN	Communications sécurisées

1.2. Procédure d'identification du produit évalué

La version de l'application est visible dans les paramètres de l'application, section « À propos ».

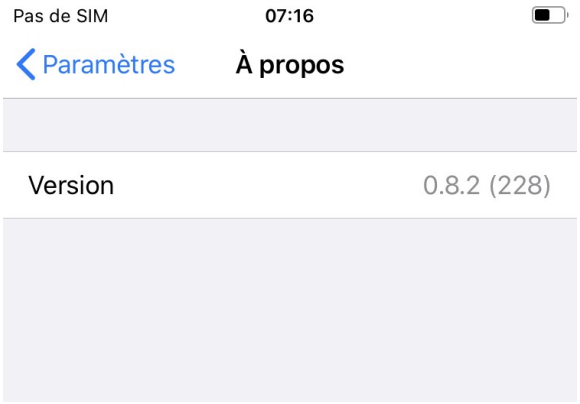

Illustration 1: Affichage du numéro de version de l'application

Le numéro de version de l'application est également disponible depuis les réglages du téléphone.

Illustration 2: Affichage du numéro de version de l'application depuis les Réglages globaux de iOS

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 :

```
36de819974d9238981456ed708a0e89a34b7ee74c41d62f9f5305fd7cec263d2  ios_client_0.8.2.zip
```

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

```
iOS
├── CoreDataStack
│   ├── CoreDataStack
│   └── CoreDataStack.xcodeproj
├── Engine
│   ├── BigInt
│   ├── FakeDelegates
│   ├── ObvBackupManager
│   ├── ObvChannelManager
│   ├── ObvCrypto
│   ├── ObvDatabaseManager
│   ├── ObvEncoder
│   ├── ObvEngine
│   ├── ObvFlowManager
│   ├── ObvIdentityManager
│   ├── ObvMetaManager
│   ├── ObvNetworkFetchManager
│   ├── ObvNetworkSendManager
│   ├── ObvNotificationCenter
│   ├── ObvOperation
│   ├── ObvProtocolManager
│   ├── ObvServerInterface
│   ├── ObvTypes
│   └── Tests
└── iOSClient
    ├── changelogs
    ├── graphics
    └── ObvMessenger
```

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

Concernant la fonctionnalité de sauvegarde du carnet de contact, **[CIBLE]** précise que le chiffrement mis en place par **[FS4]** couvre la menace **[M4]** et empêche un attaquant de récupérer le contenu en clair du fichier en cas d'interception. La fonction de sécurité semble couvrir une autre menace réaliste: **[M2]** en empêchant un attaquant de modifier un fichier de sauvegarde qu'il aurait intercepté. Les travaux d'études effectués considèrent que **[FS4]** couvre **[M2]** et **[M4]**

Le reste est 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

Aucune vulnérabilité potentielle n'a été identifiée.

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 l'**App Store** d'Apple 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 :

- iPhone 11 avec iOS 13.4.1
- iPhone 8 avec iOS 13.3
- iPhone 7 avec iOS 13.2.3

Le type de modèle ainsi que la version précise d'iOS n'entraînent pas de différence majeure pour la mise en œuvre du produit. Pour l'étude, ce choix de téléphones est motivé par le fait que les deux derniers listés sont compatibles avec le *jailbreak* **[checkra1n]**, laissant à l'auditeur la possibilité de désactiver temporairement les sécurités d'iOS pour les besoins d'analyse.

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 « Nouvel utilisateur » est utilisée.

Illustration 2: Écran d'accueil de l'application

Les vues suivantes demandent l’autorisation pour utiliser des notifications en cas de réception de nouveau message.

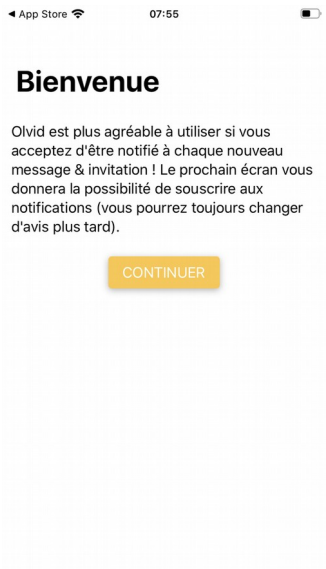

Illustration 3: Écran expliquant la nécessité de donner l'autorisation de notification à l'application

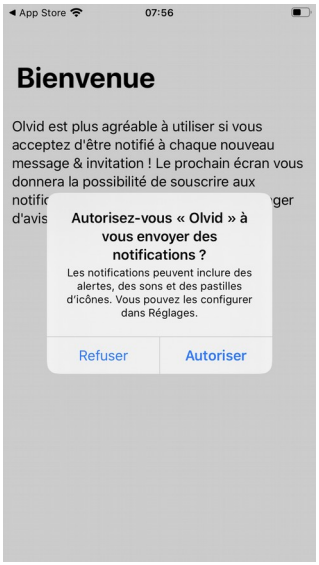

Illustration 4: Demande d'autorisation par le système

L'étape suivante consiste à créer son identité au sein de l'application Olvid. Un prénom et un nom sont requis. Le nom d'entreprise et un intitulé de poste sont optionnelles.

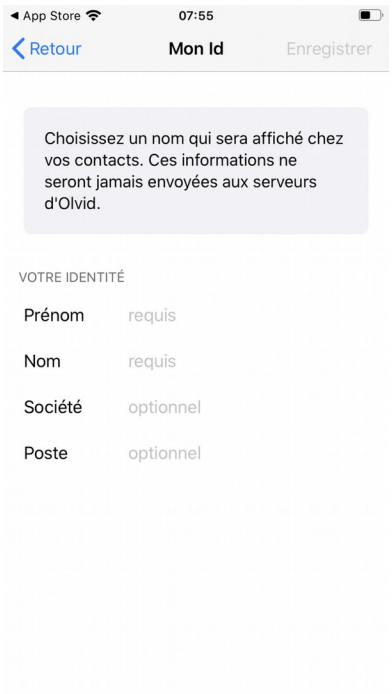

Illustration 5: Écran de création d'identité

Le dernier écran d'installation correspond à la présentation du QR code permettant de partager son identité avec un contact.

Illustration 6: Présentation du QR code permettant le partage d'identité

Une fois l'application installée et configurée, il est nécessaire d'ajouter un premier contact afin de pouvoir échanger des messages. Pour cela il suffit de se rendre dans l'écran *contacts* et de cliquer sur le bouton *ajouter*. Plusieurs options sont alors proposées.

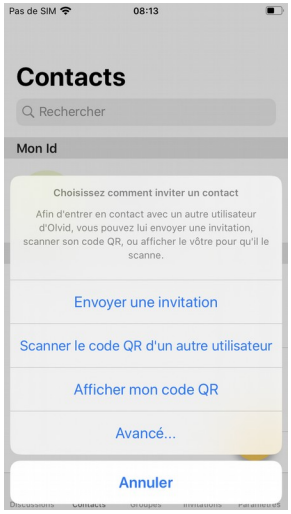

Illustration 7: Choix du mode d'ajout de contact

Il est possible de scanner le QR code d'identité du contact à ajouter ou de lui présenter son propre QR code.

Illustration 8: Scan du QR code d'un futur contact

Une fois le QR code scanné, un écran de confirmation permet de confirmer le nom de la personne à inviter.

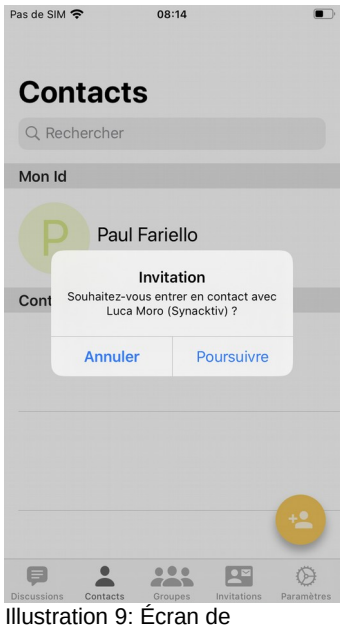

Illustration 9: Écran de confirmation d'invitation

Une fois l'invitation envoyée le contact reçoit l'invitation dans l’application Olvid et peut l'accepter.

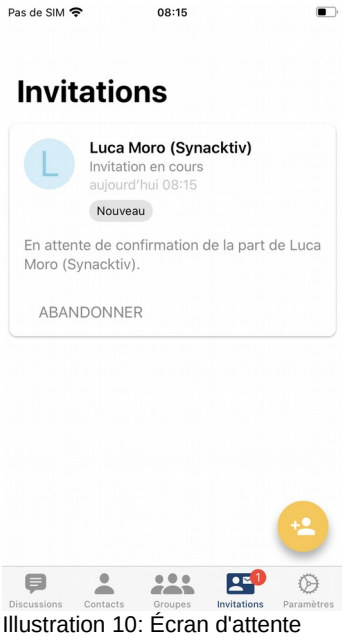

Illustration 10: Écran d'attente d'acception d'invitation

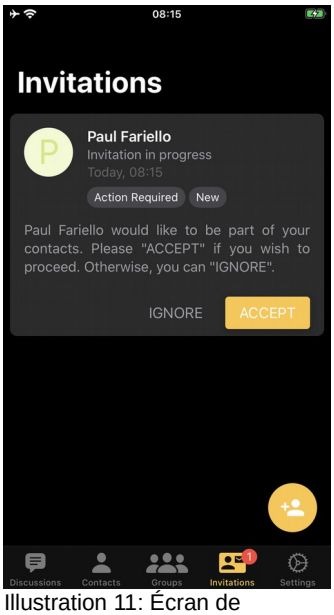

Illustration 11: Écran de réception d'invitation

Une fois l'invitation acceptée, un écran permet la création d'un canal sécurisé. Cet écran demande l'échange du code de confirmation à 4 chiffres. Les deux utilisateurs sont invités à partager ce dernier par le biais d'un canal authentique (face-à-face ou appel téléphonique).

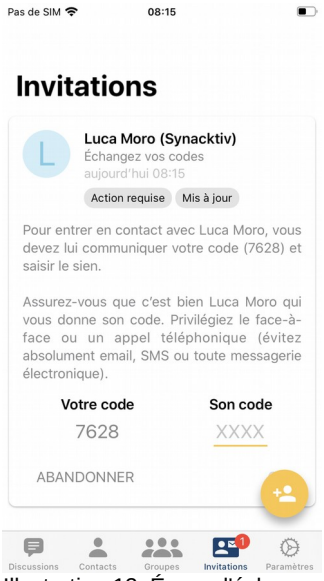

Illustration 12: Écran d'échange de code de confirmation

Une fois les deux codes échangés, le canal sécurisé est créé et le contact apparaît dans l'écran contact. Si le code échangé ne convient pas, un message d'erreur apparaît.

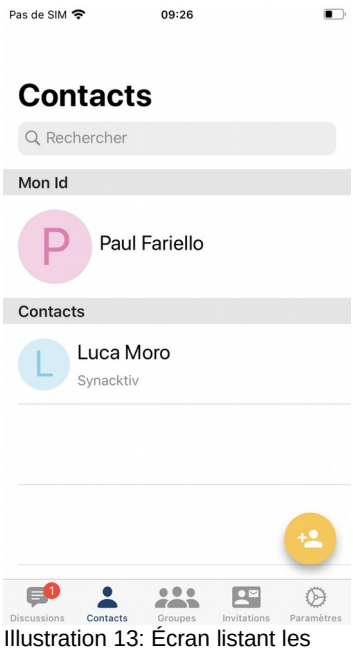

Illustration 13: Écran listant les contacts

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 contact 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 78. 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 des fournitures

Référence	Nom du fichier	Titre	Description	Version	Date
[CIBLE]	2020-04-05 CSPN_cible_Olvid_v2.2.pdf	Cible de Sécurité CSPN – Olvid	Cible de sécurité de l'application mobile Olvid	2.2	05/04/2020
[DOC_SOURCE]	ios_client_0.8.2.zip	N/A	Code source de l'application iOS Olvid	0.8.2	N/A

[DOC_CRYPTO]	Olvid - doc crypto.pdf	Olvid – Specification of Olvid – Application and Server	Spécifications d'implémentation de l'application et du serveur Olvid	N/A	15/04/2020
[SAS_PROOF]	Olvid - Preuve du protocole de Trust Establishment.pdf	Security Analysis of Olvid's Trust Establishment Protocol	Publication de M.Abdalla sur la preuve de la sécurité du protocole de Trust Establishment utilisé dans l'application Olvid	N/A	N/A

2.1.3.2. Analyse de la spécification cryptographique

L'ensemble des mécanismes cryptographiques est décrit en détail dans **[DOC_CRYPTO]**.

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 :

$$x^2 + y^2 = 1 + dx^2y^2$$

$$d = 20800338683988658368647408995589388737092878452977063003340006470870624536394$$
$$G.x = 9771384041963202563870679428059935816164187996444183106833894008023910952347$$
$$G.y = 4631683569492647816942839400347516314130799386625622561578303360316525185596 0$$

La correspondance avec les paramètres tel que présentés habituellement au format Weiestrass est la suivante:

$$Bv^2 = u^3 + Au^2 + u$$

$$A = 2 \times (1 + d) \div (1 - d)$$
$$B = 4 \div (1 - d)$$
$$x = u \div v$$
$$y = (u - 1) \div (u + 1)$$

Règle	Conformité	Justification
RègleECp-1	Conforme	L'ordre est de 2^{252}+27742317777372353535851937790883648493.
RègleECp-2	Non-Conforme	L'ordre de Curve25519 est inférieur à 2^{256} d'un facteur 16, le **[RGS_B]** ne recommande pas l'utilisation de cette primitive au-delà de 2020, cependant rien

		de laisse supposer que l'utilisation de cette courbe diminue la sécurité de l'application.
RègleECp-3	N/A	
RecomECp-1	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.

$x^2 + y^2 = 1 + dx^2y^2$

```
p = 109112363276961190442711090369149551676330307646118204517771511330536253156371
d = 39384817741350628573161184301225915800358770588933756071948264625804612259721
G.x = 82549803222020399340024462032964942512025856818700414254726364205096731424315
G.y = 91549545637415734422658288799119041756378259523097147807813396915125932811445
```

Règle	Conformité	Justification
RègleECp-1	Conforme	L'ordre est de 2^{253}+12804079664575773182731399466201399437271761490842998382698780292196380768251.
RègleECp-2	Non-Conforme	L'ordre de MDC est inférieur à 2^{256} d'un facteur 8, le **[RGS_B]** ne recommande pas l'utilisation de cette primitive au-delà de 2020, cependant rien de laisse supposer que l'utilisation de cette courbe diminue la sécurité de l'application.
RègleECp-3	N/A	
RecomECp-1	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 22. 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-1 RègleHash-2	Conforme	SHA-256 produit des hachés de 256 bits.
RègleHash-3	Conforme	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-1	Conforme	La primitive de hachage SHA-256 utilisé dans HMAC SHA-256 est conforme à la **[RGS_B]**.
RègleIntegSym-2	Conforme	Il n'existe pas d'attaque plus efficace que le paradoxe des anniversaires sur HMAC utilisé avec SHA-256.
RecomIntegSym-1	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 page 16.

Échange de clé symétrique

Une fois les clés publiques échangées et leur authenticité confirmée, 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 14.

Règle	Conformité	Justification
RecomSignAsym-1	Conforme	*Schnorr* dispose d'une preuve de sécurité.

La génération et l'échange de clés symétriques se fait 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 14 ainsi que sur le générateur de clé présenté en section Dérivation de clé page 16. Ces deux composants sont eux même conforme 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 page 15) 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 22.

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-1 RègleCléSym-2 RecomCléSym-1	Conforme	La taille des clés choisie pour le chiffrement symétrique est de 256 bits.
RègleBlocSym-1 RègleBlocSym-2 RecomBlocSym-1	Conforme	La taille des blocs pour AES-256 est de 128 bits.
RègleAlgoBloc-1 RègleAlgoBloc-2	Conforme	La meilleure attaque connu à ce jour à l'encontre de AES-256 nécessite $2^{254.3}$ opérations.
RecomAlgoBloc-1	Conforme	AES est probablement l'algorithme de chiffrement par bloc qui a été le plus éprouvé dans le milieu académique.
RègleModeChiff-1	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
RecomModeChiff-1.	Conforme	AES-256 CTR utilise un mécanisme de Nonce et est donc non-déterministe.
RecomModeChiff-2	Conforme	AES-256 CTR est utilisé conjointement à HMAC SHA-256.
RecomModeChiff-3	Conforme	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 *Apple* et n'est pas disponible publiquement. Cette implémentation est supposée correcte, 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 p.22. Cet 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 ré-implémentation 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 p.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 clefs 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 15. A chaque envoi, la clé utilisée est supprimée. Du côté de la réception, un ensemble de 100 clés sont pré-générées, celle correspondant au message reçu est supprimée après déchiffrement du message. Le fonctionnement du cliquet est expliqué ci-après.

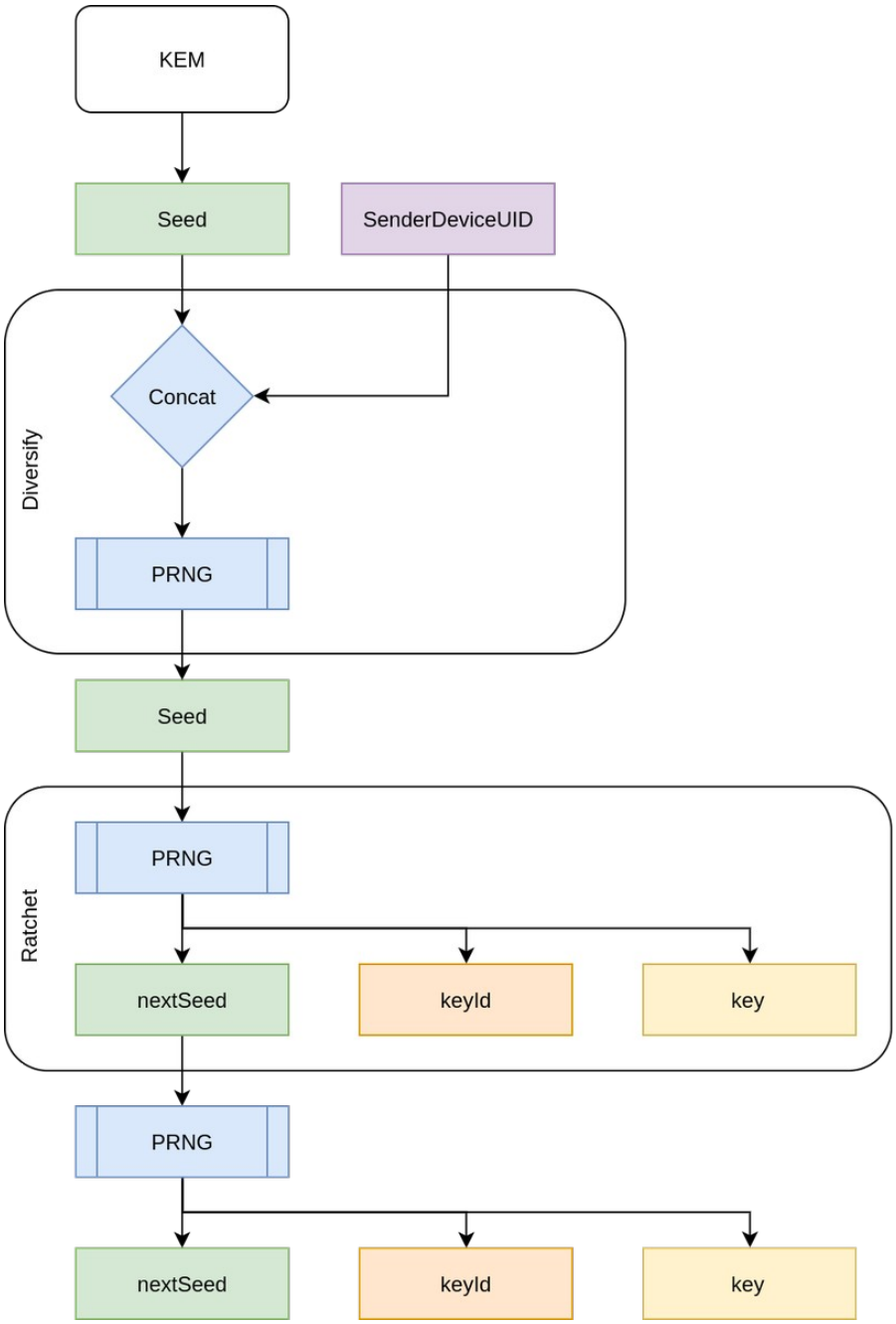

Illustration 14: Fonctionnement du cliquet simple

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 page 16. 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 *URLSession.* 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 14 et Échange de messages page 16 permettent d'assurer l'authenticité, l'intégrité et la confidentialité des données.

L'étude de ces flux est hors périmètre de **[CIBLE]**.

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-1	Conforme	Chaque clé est utilisée pour un seul et unique usage.
RègleGestSym-2	Conforme	Les clés sont différenciées en utilisant le mécanisme présenté en section Dérivation de clé page 16. Ce mécanisme est conforme au RGS.
RègleGestSym-3	Conforme	Les clés sont dérivées en utilisant le mécanisme présenté en section Dérivation de clé page 16. Ce mécanisme est conforme au RGS.
RègleGestAsym-1	Conforme	Les bi-clés sont employées pour un usage unique.
RègleGestAsym-2	Conforme	Les clés hiérarchiquement importantes sont générées en utilisant le mécanisme

		présenté en sectionGénération des clés page 15. Ce mécanisme est conforme au RGS.

Stockage des clés

Sauvegarde locale dans un container partagé

Il est nécessaire pour l'application Olvid de sauvegarder le matériel cryptographique utilisé et en particulier les clés de chiffrement. Ces clés, ainsi que d'autres données utilisées par Olvid sont stockées dans une base de données SQLite, située dans un conteneur partagé sur le système de fichiers. La base de données a comme chemin relatif */Engine/database/ObvEngine.sqlite*. Le conteneur est un dossier dont le chemin est */var/mobile/Containers/Shared/AppGroup/<UUID>/* .

Ce conteneur est accessible aux applications et plug-in de l'éditeur qui sont dans le groupe *group.io.olvid.messenger.* Dans ce groupe, on retrouve les exécutables suivant:

- ObvMessenger.app/ObvMessenger - L'exécutable principal de l'application
- ObvMessenger.app//PlugIns/ObvMessengerShareExtension.appex/ObvMessengerShareExtension – Plug-in en charge de divers fonctions propre à l'application
- ObvMessenger.app/PlugIns/ObvMessengerNotificationServiceExtension.appex/ ObvMessengerNotificationServiceExtension - Le plug-in en charge de la gestion des notifications

Les mécanismes internes du système d'exploitation iOS assurent qu'une application qui n'est pas dans ce groupe ne pourra pas accéder à la base de donnée.

La base de donnée SQLite est stockée en mémoire sans être sur-chiffrée, iOS assurant déjà nativement le chiffrement de fichier avec l'aide de sa *Secure Enclave*. Sous l'hypothèse d'un système iOS sain, ce sur-chiffrement n'est pas nécessaire. Toutefois, l'application Olvid pourrait gagner à l'implémenter tout en stockant la clé de chiffrement dans la *Keychain*. Cela permettrait à l'application de garder la confidentialité de ses données en cas de vulnérabilité de type *contournement de règles spécifiques de sandboxing.* Cela dit, les menaces habituelles sur les systèmes iOS passent par l'utilisation de *jailbreak* et vulnérabilités noyau. Cela permet le plus souvent l'injection de code dans les processus et de passer outre les protections apportées par le chiffrement. Cela réduit l'intérêt d'une telle contre-mesure. En conclusion et au vu des hypothèses, le non sur-chiffrement n'est pas retenu comme un problème de sécurité.

Le stockage du matériel cryptographique est donc adapté, si l'on considère l'hypothèse d'un système sain telle que présentée dans la section 2.4 de **[CIBLE]**.

Exclusion des backup du téléphone

On remarque que l'éditeur a pris le soin d'exclure le dossier *database* des *backup* système (au sens iOS) en positionnant l'attribut *isExcludedFromBackup* dans la fonction de création du dossier *createAndConfigureBox*.

```
static func createAndConfigureBox(_ nameOfDirectory: String, mainContainerURL: URL) -> URL?
{

        // [...]

        let box = mainContainerURL.appendingPathComponent(nameOfDirectory, isDirectory:
true)
        do {
            try FileManager.default.createDirectory(at: box, withIntermediateDirectories:
true, attributes: nil)
        } // [...]

        var resourceValues = URLResourceValues()
```

```
        resourceValues.isExcludedFromBackup = true
        do {
            var mutableBox = box
            try mutableBox.setResourceValues(resourceValues)
        } // [...]

    }
```

Ceci assure que les sauvegardes journalières *iCloud* ainsi que les sauvegardes ponctuelles réalisables avec *iTunes* ne feront pas de copie de la base de données Olvid. Cette propriété a été vérifiée dynamiquement en réalisant un *backup* manuellement avec l'outil **[idevicebackup2]** puis en constatant l'absence de la base de données.

Représentation des données dans la base SQLite

Le matériel cryptographique est stocké dans la table *ZOWNEDIDENTITY*

```
CREATE TABLE "ZOWNEDIDENTITY" (
    "Z_PK"    INTEGER,
    "Z_ENT"    INTEGER,
    "Z_OPT"    INTEGER,
    "ZISACTIVE"    INTEGER,
    "ZCURRENTDEVICE"    INTEGER,
    "ZLATESTIDENTITYDETAILS"    INTEGER,
    "ZMASKINGUID"    INTEGER,
    "ZPUBLISHEDIDENTITYDETAILS"    INTEGER,
    "ZAPIKEY"    BLOB,
    "ZCRYPTOIDENTITY"    BLOB,
    "ZOWNEDCRYPTOIDENTITY"    BLOB,
    PRIMARY KEY("Z_PK")
);
```

L'identité cryptographique telle que décrite dans **[DOC_CRYPTO]**, est stockée de manière encodée dans le blob binaire *ZOWNEDCRYPTOIDENTITY.*

Ce champ contient la concaténation des valeurs suivantes:

- serverURL
- publicKeyForAuthentication
- publicKeyForPublicKeyEncryption
- privateKeyForAuthentication
- privateKeyForPublicKeyEncryption
- secretMACKey

Il est à noter que l'organisation de la base de données SQLite est en réalité transparente d'un point de vue du code de l'application. En effet, du côté de l'implémentation, la base est accédée à travers un objet de type *NSPersistentStoreCoordinator*. Ce dernier expose une abstraction de la base *SQLite* en dictionnaire et décharge le développeur de manipuler lui-même des requêtes SQL.

Générateur de nombres pseudo-aléatoires

Caractéristiques des sources d'aléas

La fonction utilisée pour initialiser l'état interne du générateur pseudo-aléatoire est *SecRandomCopyBytes.* Sa documentation **[SecRandomCopyBytes]** précise que les octets générés sont cryptographiquement sûrs. Cette source d'aléa est hors périmètre de **[CIBLE]**. Cependant, aucune attaque n'a été publiée à ce jour sur cette source.

Analyse du retraitement cryptographique

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]**.

Les primitives sont décrites par le pseudo-algorithme suivant:

```
gen(N):
    output = ""
    while len(output) < N:
        V = HMAC(K,V)
        output = output || V

    update()
    return output[0:N]

update():
    K = HMAC(K, V || 0x00)
    V = HMAC(K,V)
```

La littérature **[DRBG_HMAC_PROOF]** montre que la génération d'aléa de ce générateur est un *DRBG* si HMAC est une *PRF* (PseudoRandom Function).

Analyse de la conformité de l'implémentation de la cryptographie

L'étude de code montre que PRNG.swift et PRNGService.swift sont conformes aux spécifications de **[NIST.SP.800-90Ar1]**. On notera la non implémentation de la fonction *reseed_counter.* Ce choix s'explique par le fait que le PRNG n'est pas reseedé pour garder son déterminisme. Cette variable est normalement utilisée pour forcer le rafraîchissement de l'état interne quand un nombre de requêtes a été atteint (*reseed_interval*). Pour HMAC-SHA256, **[NIST.SP.800-90Ar1]** préconise de fixer *reseed_interval* au maximum à 2^48. A la vue du cadre d'utilisation (terminal mobile) et du fait que les utilisateurs sont considérés comme non malveillants, il semble hautement invraisemblable d'atteindre ce nombre de requêtes. En conséquence et étant donné que le PRNG est initialisé avec une source supposée sûre, le choix de ne pas implémenter *reseed_counter* n'entraîne pas de réduction du niveau de sécurité.

L'implémentation du PRNG s'accompagne d'un jeu de tests avec des vecteurs (*ObvCryptoTests.swift* et *TestVectorsPRNGWithHMACWithSHA256.swift*). Ces vecteurs de tests correspondent en partie à ceux publié par le NIST **[DRBG_TESTVECTORS]**. Tous ces tests sont validés par l'implémentation de Olvid.

Règle	Conformité	Justification
RègleArchiGDA-1	Conforme	Un retraitement algorithmique est employé: DRBG_HMAC avec HMAC-SHA256
RègleArchiGDA-2	N/A	En cas de mise hors-tension le générateur est réinitialisé avec de l'aléa provenant de *SecRandomCopyBytes* hors cadre d'étude. Le fait que *SecRandomCopyBytes* utilise de la

		mémoire volatile ou non n'est pas établie. Aucune attaque n'est connue à ce jour sur cette fonction.
RègleArchiGDA-3	Conforme	L'état interne est de 256 bit.
RègleArchiGDA-4	N/A	La source d'aléa pour initialiser le générateur provient de *SecRandomCopyBytes* qui est hors du cadre de l'étude. Aucune attaque n'est connue à ce jour sur cette fonction.
RègleAlgoGDA-1	Conforme	Les primitives cryptographiques employées par le retraitement sont HMAC-SHA256.
RègleAlgoGDA-2	Conforme	Il a été prouvé que l'aléa des sorties successives du retraitement de DBRG_HMAC avec HMAC-SHA256 sous l'hypothèse d'un état interne fiable dépend de la robustesse de HMAC-SHA256 conforme au RGS
RègleAlgoGDA-3	Conforme	Retrouver l'état interne précédent à partir d'un état interne connu et d'une sortie revient à inverser SHA256. Aucune attaque raisonnable n'est connue à ce jour.

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:

- V-01-USURPATION-TIERS page 77

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	Oui	Oui	Conforme au **[RGS_B]**
FS2	Oui	Oui	Conforme au **[RGS_B]**
FS3	Oui	Oui	Conforme au **[RGS_B]**
FS4	Oui	Oui	Conforme au **[RGS_B]**

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 d'un répertoire dans *ObvProtocolManager/ObvProtocolManager/Protocols/<ProtocolName>*. 4 fichiers permettent ensuite de définir les états, messages et étapes du protocole.

- *<ProtocolName>*ProtocolMessages.swift
- *<ProtocolName>*ProtocolStates.swift
- *<ProtocolName>*ProtocolSteps.swift
- *<ProtocolName>*Protocol.swift

Chaque état du protocole est représenté par une classe héritée de *TypeConcreteProtocolState*. Ces états sont utilisés pour stocker l'état courant du protocole en base de données.

Chaque étape permettant de transiter entre deux états est représentée par une classe héritée de *ProtocolStep* et de *TypedConcreteProtocolStep*. 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 *ConcreteProtocolMessage*. Ces messages permettent de mettre en relation les deux participants.

L'initialisation d'un protocole est réalisée via l'envoi d'un message *InitialMessage*.

```
public func start<ProtocolName>ProtocolOfRemoteIdentity(with remoteCryptoId: ObvCryptoId,
withFullDisplayName remoteFullDisplayName: String, forOwnedIdentyWith ownedCryptoId:
ObvCryptoId) throws {
    [...]
```

```
    let message = try
protocolDelegate.getInitialMessageFor<ProtocolName>TrustEstablishmentProtocol(of:
remoteCryptoId.cryptoIdentity,

withFullDisplayName: remoteFullDisplayName,

forOwnedIdentity: ownedCryptoId.cryptoIdentity,

withOwnedIdentityCoreDetails: obvOwnedIdentity.currentIdentityDetails.coreDetails,

usingProtocolInstanceUid: protocolInstanceUid)
    [...]
    _ = try channelDelegate.post(message, randomizedWith: prng, within: obvContext)
    [...]
}
```

La réception d'un message provoque l'exécution d'une étape du protocole. Cette opération est réalisée au sein de la fonction *tryToExecuteAnAppropriateCryptoProtocolStep*.

```
/// This method tries to find an appropriate crypto protocol given the received message. If
it manages to do so, it tries to find an appropriate step to execute, and execute it in
order to transition the concrete crypto protocol to a new state. If it manages to do so, it
returns the concrete crypto protocol it obtaine
d after executing the step, that is, in a new state that still requires to be saved in DB.
private func tryToExecuteAnAppropriateCryptoProtocolStep(given message: ReceivedMessage,
within obvContext: ObvContext) -> (concreteCryptoProtocol: ConcreteCryptoProtocol,
eraseReceivedMessagesAfterReachingAFinalState: Bool)? {
    [...]
    guard let concreteCryptoProtocol = getConcreteCryptoProtocol(given: message, prng:
prng, delegateManager: delegateManager, within: obvContext) else {
        [...]
    }
    [...]
    // We reconstructed a concrete crypto protocol, that is, a crypto protocol in a well
defined state. We can now try to turn the (generic) received protocol message into one of
the possible concrete protocol messages of the concrete crypto protocol.
    guard let concreteProtocolMessage =
concreteCryptoProtocol.getConcreteProtocolMessage(from: message) else {
        [...]
    }
    // We constructed a concrete crypto protocol and have a concrete protocol message for
this protocol. We now try to find an appropriate concrete protocol step to execute, given
the current state the protocol is in and the message we received.
    guard let stepToExecute = concreteCryptoProtocol.getConcreteStepToExecute(message:
concreteProtocolMessage) as? ProtocolStep else {
        [...]
    }
    // We have a step to execute.
    stepToExecute.execute()
    [...]
    guard let newProtocolState = stepToExecute.endState else {
        [...]
    }
    // If we reached this point, we have a concrete crypto protocol and a new state for
this protocol.
    let concreteCryptoProtocolInNewState =
concreteCryptoProtocol.transitionedTo(newProtocolState)
    return (concreteCryptoProtocolInNewState,
stepToExecute.eraseReceivedMessagesAfterReachingAFinalState)
}
```

La première étape consiste à récupérer l'état courant du protocole en se basant sur le *ProtocolID* et le *ProtocolUID*. Cette opération est réalisée par la fonction *getConcreteCryptoProtocol*. Si aucune instance courante de protocole n'est trouvée, le protocole est initialisé dans son état initial.

```
/// When receiving a protocol message, we are in one of the following situations :
/// - We can find an instance ProtocolInstance in database that matches both the protocol
id and the protocolInstanceUid: we can construct and return a `ConcreteCryptoProtocol`
based on this instance, with a current state set to the one that was saved in database.
/// - We cannot find such an instance: We return a `ConcreteCryptoProtocol` with a current
state set to `ConcreteProtocolInitialState`
private func getConcreteCryptoProtocol(given message: ReceivedMessage, prng: PRNGService,
delegateManager: ObvProtocolDelegateManager, within obvContext: ObvContext) ->
ConcreteCryptoProtocol? {
    [...]
    if let protocolInstance = ProtocolInstance.get([...]) {
        concreteCryptoProtocol = cryptoProtocolId.getConcreteCryptoProtocol([...])
    } else {
        // We create a protocol instance in DB (note that this checks whether the identity
in the message is indeed an owned identity)
        guard ProtocolInstance([...]) != nil else { [...] }
        concreteCryptoProtocol =
 cryptoProtocolId.getConcreteCryptoProtocolInInitialState([...])
    }
    return concreteCryptoProtocol
}
```

La seconde étape consiste à récupérer l'étape correspondant au message reçu. Cette opération est réalisée par la fonction *getConcreteStepToExecute* du protocole en cours.

```
func getConcreteStepToExecute(message: ConcreteProtocolMessage) -> ConcreteProtocolStep? {
    var candidateSteps = [ConcreteProtocolStep]()
    for stepId in Self.allStepIds {
        if let step = stepId.getConcreteProtocolStep(self, message) {
            candidateSteps.append(step)
        }
    }
    guard candidateSteps.count == 1 else {
        return nil
    }
    return candidateSteps.first
}
```

Cette fonction tente d'instancier chacune des étapes du protocole avec le message reçu. Or, seule une des étapes dispose d'un constructeur prenant en paramètre le message typé.

L'utilisation des *Failable Initializer* de Swift permet de garantir via le typage qu'une étape ne sera jamais déclenchée sur la réception d'un message ne lui correspondant pas.

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:

```
public enum ByteIdOfObvEncoded: UInt8 {
    case bytes = 0x00
    case int = 0x01
    case bool = 0x02
    case unsignedBigInt = 0x80
    case list = 0x03
    case dictionary = 0x04
    case symmetricKey = 0x90
    case publicKey = 0x91
    case privateKey = 0x92
}
```

Les messages reçus sont représentés par la classe *ReceivedMessage*. Toutes les données encodées au format TLV sont représentées par la classe *ObvEncoded*.

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 types attendus sont initialisés avec un *ObvEncoded*. Pour cela, Olvid étend les types de base avec des constructeurs spécifiques.

```
    init?(_ obvEncoded: ObvEncoded) { [...] }
```

Olvid utilise aussi des fonctions *decode* définies sur le type *ObvEncoded* avec des *generics* permettant un *cast* automatique via le système de typage.

```
extension ObvEncoded {
    public func decode<DecodedType: ObvDecodable>() throws -> DecodedType {
        guard let decodedValue: DecodedType = DecodedType(self) else { throw NSError() }
        return decodedValue
    }
    [...]
}
```

Cette méthode est utilisée en particulier sur les types *Array*.

```
public extension Array where Element == ObvEncoded {
    [...]
    func decode<T0: ObvDecodable>() throws -> T0 {
        guard self.count == 1 else { throw NSError() }
        return try self[0].decode()
    }
    func decode<T0: ObvDecodable, T1: ObvDecodable>() throws -> (T0, T1) {
        guard self.count == 2 else { throw NSError() }
        return try (self[0].decode(), self[1].decode())
    }
    [...]
}
```

Tous les messages sont encodés sous la forme d'une liste dont le premier élément est un Byte correspondant au type de message.

```
fileprivate extension ObvChannelMessageToSendWrapper {
    static func generateContent(type: ObvChannelMessageType, encodedElements: ObvEncoded) -
> ObvEncoded {
        return [type.encode(), encodedElements].encode()
    }

    [...]
}
```

```
public enum ObvChannelMessageType: Int, ObvCodable {
    case ProtocolMessage = 0
    case ApplicationMessage = 1
    case DialogMessage = 2
    case DialogResponseMessage = 3
    case ServerQuery = 4
    case ServerResponse = 5
}
```

Lors de la réception d'un message il est décodé sous forme de liste.

```
struct ReceivedMessage {
    [...]
    private static func parse(_ content: ObvEncoded) -> (messageType:
ObvChannelMessageType, encodedElements: ObvEncoded)? {
        guard let listOfEncoded = [ObvEncoded](content) else { return nil }
        guard listOfEncoded.count == 2 else { return nil }
        guard let messageType = ObvChannelMessageType(listOfEncoded[0]) else { return nil }
        let encodedElements = listOfEncoded[1]
        return (messageType, encodedElements)
    }
}
```

Une liste *[ObvEncoded]* est initialisée à partir de la valeur du message. L'initialisation d'une liste de valeurs *ObvEncoded* est définie par une extension du type natif *Array*.

```
public extension Array where Element == ObvEncoded {
    init?(_ obvEncoded: ObvEncoded) {
        guard obvEncoded.byteId == .list else { return nil }
        if let unpackedElements = ObvEncoded.unpack(obvEncoded) {
            self = unpackedElements
        } else {
            return nil
        }
    }

    [...]
}
```

Le type de la valeur encodée est bien vérifié avant de procéder au décodage. Le décodage est ensuite réalisé dans la fonction *unpack*.

```
extension ObvEncoded {
    [...]
```

```
    /// Returns the byte identifier of the encoded value, as well as all the encoded
elements contained within the supposly packed structure.
    ///
    /// - Parameter encodedPack: An encoded element that is expected to contain several
"packed" encoded elements.
    /// - Returns: A list of all the encoded elements contained within the "packed"
structure.
    /// - Throws: An error if the inner data of the structure is not a proper "pack" of
several encoded elements.
    public static func unpack(_ encodedPack: ObvEncoded) -> [ObvEncoded]? {
        var listOfEncodedElements = [ObvEncoded]()
        var remainingInnerData = encodedPack.innerData
        while remainingInnerData.count > 0 {
            guard let (encodedElement, remainingData) =
ObvEncoded.getNextEncodedElementAndRemainingInnerData(fromInnerData: remainingInnerData)
else { return nil }
            listOfEncodedElements.append(encodedElement)
            remainingInnerData = remainingData
        }
        return listOfEncodedElements
    }

    [...]
}
```

Une fois le type de message déterminé, le message est converti dans le bon type.

```
extension NetworkReceivedMessageDecryptor {
    [...]
    private func decryptAndProcess(_ receivedMessage: ObvNetworkReceivedMessageEncrypted,
with messageKey: AuthenticatedEncryptionKey, channelType: ObvProtocolReceptionChannelInfo,
within obvContext: ObvContext) {
        [...]
        guard let obvChannelReceivedMessage = ReceivedMessage(with: receivedMessage,
decryptedWith: messageKey, obtainedUsing: channelType) else {
            os_log("A received message could not be decrypted or parsed", log: log,
type: .error)
            networkFetchDelegate.deleteMessageAndAttachments(messageId:
receivedMessage.messageId, within: obvContext)
            return
        }
        switch obvChannelReceivedMessage.type {
        case .ProtocolMessage:
            os_log("New protocol message", log: log, type: .debug)
            if let receivedProtocolMessage = ReceivedProtocolMessage(with:
obvChannelReceivedMessage) {
                [...]
            }
        case .ApplicationMessage:
            os_log("New application message", log: log, type: .debug)
            if let receivedApplicationMessage = ReceivedApplicationMessage(with:
obvChannelReceivedMessage) {
                [...]
            }
        case .DialogMessage,
             .DialogResponseMessage,
             .ServerQuery,
             .ServerResponse:
            os_log("Dialog/Response/ServerQuery messages are not intended to be decrypted",
log: log, type: .fault)
```

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

Une fois converti, le message est décodé lors d'une conversion vers le type *GenericReceivedProtocolMessage*.

```
struct GenericReceivedProtocolMessage {
    [...]
    // Instantiating a `GenericProtocolMessage` when receiving an
`ObvProtocolReceivedMessage`
    init?(with obvProtocolReceivedMessage: ObvProtocolReceivedMessage) {
        guard let (cryptoProtocolId, protocolInstanceUid, protocolMessageRawId,
encodedInputs) =
GenericReceivedProtocolMessage.decode(obvProtocolReceivedMessage.encodedElements) else {
            return nil
        }
        self.encodedInputs = encodedInputs
        self.receptionChannelInfo = obvProtocolReceivedMessage.receptionChannelInfo
        self.toOwnedIdentity = obvProtocolReceivedMessage.messageId.ownedCryptoIdentity
        self.protocolInstanceUid = protocolInstanceUid
        self.protocolMessageRawId = protocolMessageRawId
        self.cryptoProtocolId = cryptoProtocolId
        self.encodedUserDialogResponse = nil
        self.userDialogUuid = nil
        self.timestamp = obvProtocolReceivedMessage.timestamp
    }

    [...]

    private static func decode(_ encodedElements: ObvEncoded) -> (CryptoProtocolId, UID,
Int, [ObvEncoded])? {
        guard let listOfEncoded = [ObvEncoded](encodedElements, expectedCount: 4) else
{ return nil }
        guard let cryptoProtocolRawId = Int(listOfEncoded[0]) else { return nil }
        guard let cryptoProtocolId = CryptoProtocolId(rawValue: cryptoProtocolRawId) else {
return nil }
        guard let protocolInstanceUid = UID(listOfEncoded[1]) else { return nil }
        guard let protocolMessageRawId = Int(listOfEncoded[2]) else { return nil }
        guard let encodedInputs = [ObvEncoded](listOfEncoded[3]) else { return nil }
        return (cryptoProtocolId, protocolInstanceUid, protocolMessageRawId, encodedInputs)
    }
    [...]
}
```

Le décodage des valeurs encodées dans le message est réalisé par *GenericReceivedProtocolMessage.decode*.

La revue systématique des différents messages et des types implémentant *ObvDecodable* ne montre aucun problème d'implémentation.

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 garantie 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 14 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.

Analyse statique

Le protocole *Trust Establishment Protocol with SAS* est implémenté sous forme de machine à état. L'implémentation des machines à état à été revue dans Machine à état page 24.

Les fichiers suivants définissent l'ensemble des éléments requis à la définition du protocole *Trust Establishment Protocol with SAS*:

- TrustEstablishmentWithSASProtocolMessages.swift
- TrustEstablishmentWithSASProtocolStates.swift
- TrustEstablishmentWithSASProtocolSteps.swift
- TrustEstablishmentWithSASProtocol.swift

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

```
struct WaitingForSeedState: TypeConcreteProtocolState { [...] }
struct WaitingForConfirmationState: TypeConcreteProtocolState { [...] }
struct WaitingForDecommitmentState: TypeConcreteProtocolState { [...] }
struct WaitingForUserSASState: TypeConcreteProtocolState { [...] }
struct ContactIdentityTrustedLegacyState: TypeConcreteProtocolState { [...] }
struct ContactSASCheckedState: TypeConcreteProtocolState { [...] }
struct MutualTrustConfirmedState: TypeConcreteProtocolState { [...] }
struct CancelledState: TypeConcreteProtocolState { [...] }
```

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

```
final class SendCommitmentStep: ProtocolStep, TypedConcreteProtocolStep { [...] }
final class StoreDecommitmentStep: ProtocolStep, TypedConcreteProtocolStep { [...] }
```

```
final class ShowSasDialogAndSendDecommitmentStep: ProtocolStep, TypedConcreteProtocolStep {
[...] }
final class StoreAndPropagateCommitmentAndAskForConfirmationStep: ProtocolStep,
TypedConcreteProtocolStep { [...] }
final class StoreCommitmentAndAskForConfirmationStep: ProtocolStep,
TypedConcreteProtocolStep { [...] }
final class SendSeedAndPropagateConfirmationStep: ProtocolStep, TypedConcreteProtocolStep {
[...] }
final class ReceiveConfirmationFromOtherDeviceStep: ProtocolStep, TypedConcreteProtocolStep
{ [...] }
final class ShowSasDialogStep: ProtocolStep, TypedConcreteProtocolStep { [...] }
final class CheckSasStep: ProtocolStep, TypedConcreteProtocolStep { [...] }
final class CheckPropagatedSasStep: ProtocolStep, TypedConcreteProtocolStep { [...] }
final class NotifiedMutualTrustEstablishedLegacyStep: ProtocolStep,
TypedConcreteProtocolStep { [...] }
final class AddTrustStep: ProtocolStep, TypedConcreteProtocolStep { [...] }
```

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

```
struct InitialMessage: ConcreteProtocolMessage { [...] }
struct AliceSendsCommitmentMessage: ConcreteProtocolMessage { [...] }
struct AlicePropagatesHerInviteToOtherDevicesMessage: ConcreteProtocolMessage { [...] }
struct BobPropagatesCommitmentToOtherDevicesMessage: ConcreteProtocolMessage { [...] }
struct BobDialogInvitationConfirmationMessage: ConcreteProtocolMessage { [...] }
struct BobPropagatesConfirmationToOtherDevicesMessage: ConcreteProtocolMessage { [...] }
struct BobSendsSeedMessage: ConcreteProtocolMessage { [...] }
struct AliceSendsDecommitmentMessage: ConcreteProtocolMessage { [...] }
struct DialogSasExchangeMessage: ConcreteProtocolMessage { [...] }
struct PropagateEnteredSasToOtherDevicesMessage: ConcreteProtocolMessage { [...] }
struct MutualTrustConfirmationMessageMessage: ConcreteProtocolMessage { [...] }
struct DialogForMutualTrustConfirmationMessage: ConcreteProtocolMessage { [...] }
struct DialogInformativeMessage: ConcreteProtocolMessage { [...] }
```

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

```
init?(startState: ConcreteProtocolInitialState, receivedMessage: InitialMessage,
concreteCryptoProtocol: ConcreteCryptoProtocol) {
init?(startState: ConcreteProtocolInitialState, receivedMessage:
AlicePropagatesHerInviteToOtherDevicesMessage, concreteCryptoProtocol:
ConcreteCryptoProtocol) {
init?(startState: WaitingForSeedState, receivedMessage: BobSendsSeedMessage,
concreteCryptoProtocol: ConcreteCryptoProtocol) {
init?(startState: ConcreteProtocolInitialState, receivedMessage:
AliceSendsCommitmentMessage, concreteCryptoProtocol: ConcreteCryptoProtocol) {
init?(startState: ConcreteProtocolInitialState, receivedMessage:
BobPropagatesCommitmentToOtherDevicesMessage, concreteCryptoProtocol:
ConcreteCryptoProtocol) {
init?(startState: WaitingForConfirmationState, receivedMessage:
BobDialogInvitationConfirmationMessage, concreteCryptoProtocol: ConcreteCryptoProtocol) {
init?(startState: WaitingForConfirmationState, receivedMessage:
BobPropagatesConfirmationToOtherDevicesMessage, concreteCryptoProtocol:
ConcreteCryptoProtocol) {
init?(startState: WaitingForDecommitmentState, receivedMessage:
AliceSendsDecommitmentMessage, concreteCryptoProtocol: ConcreteCryptoProtocol) {
init?(startState: WaitingForUserSASState, receivedMessage: DialogSasExchangeMessage,
concreteCryptoProtocol: ConcreteCryptoProtocol) {
init?(startState: WaitingForUserSASState, receivedMessage:
PropagateEnteredSasToOtherDevicesMessage, concreteCryptoProtocol: ConcreteCryptoProtocol) {
```

```
init?(startState: ContactIdentityTrustedLegacyState, receivedMessage:
MutualTrustConfirmationMessageMessage, concreteCryptoProtocol: ConcreteCryptoProtocol) {
init?(startState: ContactSASCheckedState, receivedMessage:
MutualTrustConfirmationMessageMessage, concreteCryptoProtocol: ConcreteCryptoProtocol) {
```

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
ShowSasDialogAndSendDecommitmentStep -> WaitingForUserSASState
StoreAndPropagateCommitmentAndAskForConfirmationStep -> WaitingForConfirmationState
StoreCommitmentAndAskForConfirmationStep -> WaitingForConfirmationState
SendSeedAndPropagateConfirmationStep -> WaitingForConfirmationState
ReceiveConfirmationFromOtherDeviceStep -> WaitingForDecommitmentState
ShowSasDialogStep -> WaitingForUserSASState
CheckSasStep -> WaitingForUserSASState, ContactSASCheckedState
CheckPropagatedSasStep -> 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 et de son identité et un nonce.

```
override func executeStep(within obvContext: ObvContext) throws -> ConcreteProtocolState? {
    [...]
    // Generate a seed for the SAS and commit on it
    let seedAliceForSas = prng.genSeed()
    let commitmentScheme = ObvCryptoSuite.sharedInstance.commitmentScheme()
    let (commitment, decommitment) = commitmentScheme.commit(onTag:
ownedIdentity.getIdentity(),
                                                             andValue: seedAliceForSas.raw,
                                                             with: prng)
    [...]
    // Send the commitment on our own seed for SAS, as well as our identity and display
name.
    [...]
    do {
        let coreMessage = getCoreMessage(for: .AsymmetricChannelBroadcast(to:
contactIdentity, fromOwnedIdentity: ownedIdentity))
        let concreteProtocolMessage = AliceSendsCommitmentMessage(coreProtocolMessage:
coreMessage,

contactIdentityCoreDetails: ownIdentityCoreDetails,
                                                                  contactIdentity:
ownedIdentity,
                                                                  contactDeviceUids: [UID]
(ownedDeviceUids),
                                                                  commitment: commitment)
        guard let messageToSend =
concreteProtocolMessage.generateObvChannelProtocolMessageToSend(with: prng) else { throw
NSError() }
        _ = try channelDelegate.post(messageToSend, randomizedWith: prng, within:
obvContext)
    }
    [...]
}
```

La génération du *seed* via le PRNG à été revue dans Générateur de nombres pseudo-aléatoires p.22. La génération du commitment est réalisée conformément à **[DOC_CRYPTO]** par *commitmentScheme.commit*.

```
static func commit(onTag tag: Data, andValue value: Data, with prng: PRNG) -> (commitment:
Data, decommitToken: Data) {
    // Compute d
    var d = Data(value)
    d.append(prng.genBytes(count: CommitmentWithSHA256.randomValueLength))
    // Compute the commitment
    var dataToHash = Data(tag)
    dataToHash.append(d)
    let commitment = SHA256.hash(dataToHash)
    return (commitment, d) // FIXME: A commitment should allow to identify the algorithm to
use to open it
}
```

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

```
override func executeStep(within obvContext: ObvContext) throws -> ConcreteProtocolState? {
    [...]
    // Send a seed for the SAS to Alice
    let seedBobForSas: Seed
    do {
        guard !commitment.isEmpty else { throw NSError() }
        seedBobForSas = try
identityDelegate.getDeterministicSeedForOwnedIdentity(ownedIdentity, diversifiedUsing:
commitment, within: obvContext)
    } catch {
        os_log("Could not compute (deterministic but diversified) seed for sas", log: log,
type: .error)
        return CancelledState()
    }
    [...]
    do {
        let coreMessage = getCoreMessage(for: .AsymmetricChannel(to: contactIdentity,
remoteDeviceUids: contactDeviceUids, fromOwnedIdentity: ownedIdentity))
        let concreteProtocolMessage = BobSendsSeedMessage(coreProtocolMessage: coreMessage,
                                                          seedBobForSas: seedBobForSas,
                                                          contactIdentityCoreDetails:
ownedIdentityCoreDetails,
                                                          contactDeviceUids: [UID]
(ownedDeviceUids))
        guard let messageToSend =
concreteProtocolMessage.generateObvChannelProtocolMessageToSend(with: prng) else { throw
NSError() }
        _ = try channelDelegate.post(messageToSend, randomizedWith: prng, within:
obvContext)
    }
    [...]
}
```

La génération du seed est réalisée de manière déterministe. Ce comportement est analysé en détail dans V-03 REJEU-COMMITMENT page 80.

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

```
override func executeStep(within obvContext: ObvContext) throws -> ConcreteProtocolState? {
    [...]
    // Send the decommitment to Bob
    do {
```

```
        let coreMessage = getCoreMessage(for: .AsymmetricChannel(to: contactIdentity,
remoteDeviceUids: contactDeviceUids, fromOwnedIdentity: ownedIdentity))
        let concreteProtocolMessage = AliceSendsDecommitmentMessage(coreProtocolMessage:
coreMessage, decommitment: decommitment)
        guard let messageToSend =
concreteProtocolMessage.generateObvChannelProtocolMessageToSend(with: prng) else { throw
NSError() }
        _ = try channelDelegate.post(messageToSend, randomizedWith: prng, within:
obvContext)
    }
    // Bob accepted the invitation. We have all the information we need to compute and show
a SAS dialog to Alice.
    let sasToDisplay: Data
    do {
        let fullSAS = try SAS.compute(seedAlice: seedAliceForSas, seedBob: seedBobForSas,
identityBob: contactIdentity, numberOfDigits: ObvConstants.defaultNumberOfDigitsForSAS * 2)
        sasToDisplay = fullSAS.leftHalf
    } catch let error {
        os_log("Could not compute SAS: %{public}@", log: log, type: .fault,
error.localizedDescription)
        return CancelledState()
    }
   [...]
}
```

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

```
override func executeStep(within obvContext: ObvContext) throws -> ConcreteProtocolState? {
    [...]
    // Open the commitment to recover the contact seed for the SAS
    let seedAliceForSas: Seed
    do {
        let commitmentScheme = ObvCryptoSuite.sharedInstance.commitmentScheme()
        guard let rawContactSeedForSAS = commitmentScheme.open(commitment: commitment,
onTag: contactIdentity.getIdentity(), usingDecommitToken: decommitment) else {
            os_log("Could not open the commitment", log: log, type: .error)
            return CancelledState()
        }
        guard let seed = Seed(with: rawContactSeedForSAS) else {
            os_log("Could not recover contact seed", log: log, type: .error)
            return CancelledState()
        }
        seedAliceForSas = seed
    }
    // We have all the information we need to compute and show a SAS dialog to Bob
    let sasToDisplay: Data
    do {
        let fullSAS = try SAS.compute(seedAlice: seedAliceForSas, seedBob: seedBobForSas,
identityBob: ownedIdentity, numberOfDigits: ObvConstants.defaultNumberOfDigitsForSAS * 2)
        sasToDisplay = fullSAS.rightHalf
    } catch let error {
        os_log("Could not compute SAS: %{public}@", log: log, type: .fault,
error.localizedDescription)
        return CancelledState()
    }
    [...]
}
```

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

```
static func open(commitment: Data, onTag tag: Data, usingDecommitToken d: Data) -> Data? {
    var value: Data? = nil
    var dataToHash = Data(tag)
    dataToHash.append(d)
    let computedCommitment = SHA256.hash(dataToHash)
    if commitment == computedCommitment {
        value = Data(d)
        value!.removeLast(CommitmentWithSHA256.randomValueLength)
    }
    return value
}
```

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

```
/// 2019-10-24: This new way of computing a SAS will be used in the new version of the
trust authentication protocol.
public static func compute(seedAlice: Seed, seedBob: Seed, identityBob: ObvCryptoIdentity,
numberOfDigits: Int) throws -> Data {
    let toHash = identityBob.getIdentity() + seedAlice.raw
    let hash = SHA256.hash(toHash)
    let xorLength = min(hash.count, seedBob.length)
    let hashToXor = hash[hash.startIndex..<hash.startIndex+xorLength]
    let seedBobToXor =
seedBob.raw[seedBob.raw.startIndex..<seedBob.raw.startIndex+xorLength]
    let xorResult = try Data.xor(hashToXor, seedBobToXor)
    guard let seed = Seed(with: xorResult) else {
        throw makeError(message: "Could not instantiate a seed on the basis of the xor
between an identity and another seed")
    }
    let prng = PRNGWithHMACWithSHA256(with: seed)
    let zeroes = String(repeating: "0", count: numberOfDigits)
    let maxAsString = "1" + zeroes
    let max: BigInt
    do {
        max = try BigInt(maxAsString, base: .ten)
    } catch {
        throw makeError(message: "Could not instantiate BigInt")
    }
    let sasAsBigInt = prng.genBigInt(smallerThan: max)
    let sasPostfix = String(sasAsBigInt, base: .ten)
    let sasPrefix = String(repeating: "0", count: numberOfDigits - sasPostfix.count)
    let sas = sasPrefix + sasPostfix
    return sas.data(using: .utf8)!
}
```

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.

```
override func executeStep(within obvContext: ObvContext) throws -> ConcreteProtocolState? {
    [...]
    // Re-compute the SAS and compare it to the SAS entered by the user
    let sasToDisplay: Data
    do {
        let seedAlice = isAlice ? seedForSas : contactSeedForSas
        let seedBob = isAlice ? contactSeedForSas : seedForSas
        let identityBob = isAlice ? contactIdentity : ownedIdentity
        let fullSAS = try SAS.compute(seedAlice: seedAlice, seedBob: seedBob, identityBob:
identityBob, numberOfDigits: ObvConstants.defaultNumberOfDigitsForSAS * 2)
        sasToDisplay = isAlice ? fullSAS.leftHalf : fullSAS.rightHalf
        let sasToCompare = isAlice ? fullSAS.rightHalf : fullSAS.leftHalf
```

```
        guard sasToCompare == sasEnteredByUser else {
            os_log("The SAS entered by the user does not match the expected SAS.", log:
log, type: .error)
            // We re-post the same dialog
            let newNumberOfBadEnteredSas = numberOfBadEnteredSas + 1
            do {
                let contact = CryptoIdentityWithCoreDetails(cryptoIdentity:
contactIdentity, coreDetails: contactIdentityCoreDetails)
                let dialogType = ObvChannelDialogToSendType.sasExchange(contact: contact,
sasToDisplay: sasToDisplay, numberOfBadEnteredSas: newNumberOfBadEnteredSas)
                let coreMessage = getCoreMessage(for: .UserInterface(uuid: dialogUuid,
ownedIdentity: ownedIdentity, dialogType: dialogType))
                let concreteProtocolMessage = DialogSasExchangeMessage(coreProtocolMessage:
coreMessage)
                guard let messageToSend =
concreteProtocolMessage.generateObvChannelDialogMessageToSend() else { throw NSError() }
                _ = try channelDelegate.post(messageToSend, randomizedWith: prng, within:
obvContext)
            }
            // We go back to the WaitingForUserSAS state (only the number of bad entered
sas changes)
            return WaitingForUserSASState(contactIdentity: contactIdentity,
                                          contactIdentityCoreDetails:
contactIdentityCoreDetails,
                                          contactDeviceUids: contactDeviceUids,
                                          seedForSas: seedForSas,
                                          contactSeedForSas: contactSeedForSas,
                                          dialogUuid: dialogUuid,
                                          isAlice: isAlice,
                                          numberOfBadEnteredSas: newNumberOfBadEnteredSas)
        }
    } catch {
        os_log("Could not re-compute the SAS and compare it to the SAS entered by the
user", log: log, type: .fault)
        return CancelledState()
    }
    [...]
    // 2020-03-02 : We used to add the contact identity to the contact database (or simply
add a new trust origin if the contact already exists) and add all the contact device uids
    // We do not do this now. Instead, this is performed within the
AddAndPropagateTrustStep since, at this point, we know for sure that both users checked
their respective SAS.
    // Send a confirmation message
    do {
        let coreMessage = getCoreMessage(for: .AsymmetricChannel(to: contactIdentity,
remoteDeviceUids: contactDeviceUids, fromOwnedIdentity: ownedIdentity))
        let concreteProtocolMessage =
MutualTrustConfirmationMessageMessage(coreProtocolMessage: coreMessage)
        guard let messageToSend =
concreteProtocolMessage.generateObvChannelProtocolMessageToSend(with: prng) else { return
nil }
        _ = try channelDelegate.post(messageToSend, randomizedWith: prng, within:
obvContext)
    }
    [...]
}
```

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: 1f47df688d146a241bdd27547ba5cf4833c800985a94ade9b8a15e8a809adfaa
    Alice sends commitment
      Contact identity:
68747470733a2f2f7365727665722e6f6c7669642e696f00002c56c7c4b256d707b3d258c9266942b9eec363651
2f8546bfc7773c5e65ddef50107b054f42552f293df32561630cf960311b268cd925881fbeee073e1ec06a0e2
      Encoded Contact identity: 7b2266697273745f6e616d65223a22416c696365227d
      Contact device UIDs:
 [f4ee3ae632595d6833b024ae51805fb7403d686560b26246a92a2c8391c37a0d]
      Commitment:
18453f7d65aa8fa83baf92dc0c5dbb5fbf932dfffd28c65a8d4368cd4b9cc2a4
```

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

```
Protocol Message
  Trust Establishment with SAS
    Protocol UID: 1f47df688d146a241bdd27547ba5cf4833c800985a94ade9b8a15e8a809adfaa
    Bob sends seed
      seedBobForSas:
 4341c3e44daf1531438c882dae27ad0628fd9a3ed5b0264c81ef2118b00294b4
      Contact device UIDs:
 [48e7cb75d852123700690e6eb0552b1dd026e7c2b23ecd57c410788010e22ed7]
      Encoded Contact identity: 7b2266697273745f6e616d65223a22426f62227d
```

Vient ensuite le *decommitment* d'Alice.

```
Protocol Message
  Trust Establishment with SAS
    Protocol UID: 1f47df688d146a241bdd27547ba5cf4833c800985a94ade9b8a15e8a809adfaa
    Alice sends decommitment
      decommitment:
 55fad660bf8695bdc490b6ccfa33d8af393b7e542cb7e577344ba439102b77dd8e711e376c7d1b3aa0ed36b309
22a7439acc3eac85a0bfb6a617f3063ba40443
```

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

```
To:
68747470733a2f2f7365727665722e6f6c7669642e696f00002c56c7c4b256d707b3d258c9266942b9eec363651
2f8546bfc7773c5e65ddef50107b054f42552f293df32561630cf960311b268cd925881fbeee073e1ec06a0e2
(f4ee3ae632595d6833b024ae51805fb7403d686560b26246a92a2c8391c37a0d)
Protocol Message
  Trust Establishment with SAS
    Protocol UID: 1f47df688d146a241bdd27547ba5cf4833c800985a94ade9b8a15e8a809adfaa
    Mutual trust confirmation

To:
68747470733a2f2f7365727665722e6f6c7669642e696f000023569981ae071758ec2ef28eae66ae9df5223e2fd
a5919fb911ff26daa9026fa010b3b4561f33b73c67c56a5e9e8a6073897c9a675f071dc2073757472bcb13a95
(48e7cb75d852123700690e6eb0552b1dd026e7c2b23ecd57c410788010e22ed7)
```

```
Protocol Message
  Trust Establishment with SAS
    Protocol UID: 1f47df688d146a241bdd27547ba5cf4833c800985a94ade9b8a15e8a809adfaa
    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 14, il était nécessaire de vérifier que les autres implémentations sont correctes.

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

- L'utilisation de primitives cryptographiques testées et éprouvées;
- Une ré-implémentation 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 conséquences, 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 au RGS 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 garantie 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 15. Les clés générées spécifiquement pour chaque message sont à leur tour chiffrées via du chiffrement asymétrique puis ajoutées dans l'en-tête du message. Le chiffrement asymétrique utilisé pour chiffrer les clés associées au message doit garantir l'authentification des échanges.

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.

Chiffrement asymétrique

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.

Le chiffrement asymétrique est réalisé en utilisant l'*AsymmetricChannel*. En particulier lors de la construction d'un message la fonction *getCoreMessage* associée au canal asymétrique est utilisée.

```
getCoreMessage(for: .AsymmetricChannelBroadcast(to: contactIdentity, fromOwnedIdentity:
ownedIdentity))
getCoreMessage(for: .AsymmetricChannel(to: contactIdentity, remoteDeviceUids:
contactDeviceUids, fromOwnedIdentity: ownedIdentity))
```

Lors de l'envoi du message ainsi généré, un appel à la fonction *ObvNetworkChannel.post* est réalisé.

```
extension ObvNetworkChannel {
    [...]
    private static func generateMessageKeyAndHeaders(using acceptableChannels:
[ObvNetworkChannel], randomizedWith prng: PRNGService) -> (AuthenticatedEncryptionKey,
[ObvNetworkMessageToSend.Header])? {
        let cryptoSuiteVersion =
acceptableChannels.reduce(ObvCryptoSuite.sharedInstance.latestVersion) { min($0,
$1.cryptoSuiteVersion) }
        guard let authEnc =
ObvCryptoSuite.sharedInstance.authenticatedEncryption(forSuiteVersion: cryptoSuiteVersion)
else {
            return nil
        }
```

```
        let messageKey = authEnc.generateKey(with: prng)
        let headers = acceptableChannels.map { $0.wrapMessageKey(messageKey,
randomizedWith: prng) }
        return (messageKey, headers)
    }
    [...]
    static func post(_ message: ObvChannelMessageToSend, randomizedWith prng: PRNGService,
delegateManager: ObvChannelDelegateManager, within obvContext: ObvContext) throws ->
Set<ObvCryptoIdentity> {
        [...]
        guard let (messageKey, headers) = generateMessageKeyAndHeaders(using:
acceptableChannels, randomizedWith: prng) else { throw NSError() }
        guard let networkMessage = generateObvNetworkMessageToSend(from: message,
messageKey: messageKey, headers: headers, randomizedWith: prng) else { throw NSError() }
        try networkPostDelegate.post(networkMessage, within: obvContext)
        [...]
    }
}
```

La clé utilisée pour chiffrer le message est chiffrée de manière asymétrique via un appel à la fonction *wrapMessageKey* de l'*AsymmetricChannel*.

```
func wrapMessageKey(_ messageKey: AuthenticatedEncryptionKey, randomizedWith prng:
PRNGService) -> ObvNetworkMessageToSend.Header {
    let wrappedMessageKey = keyWrapperForIdentityDelegate.wrap(messageKey, for: identity,
randomizedWith: prng)
    let header = ObvNetworkMessageToSend.Header(toIdentity: identity, deviceUid: deviceUid,
wrappedMessageKey: wrappedMessageKey)
    return header
}
```

La fonction réalisant le chiffrement de la clé est définie par le *delegate keyWrapperForIdentityDelegate*.

```
extension ObvIdentityManagerImplementation: ObvKeyWrapperForIdentityDelegate {
    public func wrap(_ key: AuthenticatedEncryptionKey, for identity: ObvCryptoIdentity,
randomizedWith prng: PRNGService) -> EncryptedData {
        return PublicKeyEncryption.encrypt(key.encode().rawData, for: identity,
randomizedWith: prng)
    }
```

Le chiffrement utilisé est donc bien le chiffrement asymétrique lié à la clé publique du destinataire.

```
extension PublicKeyEncryptionCommon {
    public static func encrypt(_ plaintext: Data, for identity: ObvCryptoIdentity,
randomizedWith prng: PRNGService) -> EncryptedData {
        return encrypt(plaintext, using: identity.publicKeyForPublicKeyEncryption, and:
prng)
    }
    [...]
}
[...]
public protocol PublicKeyEncryptionGeneric: PublicKeyEncryptionCommon {
    [...]
}
public final class PublicKeyEncryption: PublicKeyEncryptionGeneric {
    [...]
    public static func encrypt(_ plaintext:Data, using publicKey:
PublicKeyForPublicKeyEncryption, and prng: PRNGService) -> EncryptedData {
```

```
        return
publicKey.algorithmImplementationByteId.algorithmImplementation.encrypt(plaintext, using:
publicKey, and: prng)
    }
    [...]
}
```

L'analyse dynamique confirme que l' *algorithmImplementationByteId* utilisé est bien *KEM_ECIES_Curve25519_and_DEM_CTR_AES_256_then_HMAC_SHA_256* et correspond à l'utilisation de *ECIESwithCurve25519andDEMwithCTRAES256thenHMACSHA256*.

```
public enum PublicKeyEncryptionImplementationByteId: UInt8 {
    case KEM_ECIES_MDC_and_DEM_CTR_AES_256_then_HMAC_SHA_256 = 0x00
    case KEM_ECIES_Curve25519_and_DEM_CTR_AES_256_then_HMAC_SHA_256 = 0x01
    public var algorithmImplementation: PublicKeyEncryptionConcrete.Type {
        switch self {
        case .KEM_ECIES_MDC_and_DEM_CTR_AES_256_then_HMAC_SHA_256:
            return ECIESwithMDCandDEMwithCTRAES256thenHMACSHA256.self as
PublicKeyEncryptionConcrete.Type
        case .KEM_ECIES_Curve25519_and_DEM_CTR_AES_256_then_HMAC_SHA_256:
            return ECIESwithCurve25519andDEMwithCTRAES256thenHMACSHA256.self as
PublicKeyEncryptionConcrete.Type
        }
    }
}
```

La classe *ECIESwithCurve25519andDEMwithCTRAES256thenHMACSHA256* réalise le chiffrement asymétrique d'une clé aléatoire via le KEM puis chiffre la clé temporaire via un chiffrement symétrique AES en utilisant la clé aléatoire générée via le KEM.

```
extension ECIESwithEdwardsCurveandDEMwithCTRAES256thenHMACSHA256 {
    private static var KEM: KEM_ECIES256KEM512.Type {
        switch algorithmImplementationByteId {
        case .KEM_ECIES_MDC_and_DEM_CTR_AES_256_then_HMAC_SHA_256:
            return KEM_ECIES256KEM512_WithMDC.self as KEM_ECIES256KEM512.Type
        case .KEM_ECIES_Curve25519_and_DEM_CTR_AES_256_then_HMAC_SHA_256:
            return KEM_ECIES256KEM512_WithCurve25519.self as KEM_ECIES256KEM512.Type
        }
    }
    [...]
    static func encrypt(_ plaintext: Data, using publicKey:
PublicKeyForPublicKeyEncryption, and prng: PRNGService) -> EncryptedData {
        let (c0, key) = KEM.encrypt(using: publicKey, with: prng)
{ AuthenticatedEncryptionWithAES256CTRThenHMACWithSHA256Key(data: $0)! }!
        let c1 = try!
AuthenticatedEncryptionWithAES256CTRThenHMACWithSHA256.encrypt(plaintext, with: key, and:
prng)
        let ciphertext = EncryptedData.byAppending(c1: c0, c2: c1)
        return ciphertext
    }
    [...]
}
[...]
final class ECIESwithCurve25519andDEMwithCTRAES256thenHMACSHA256:
ECIESwithEdwardsCurveandDEMwithCTRAES256thenHMACSHA256 {
    static var curve: EdwardsCurve = Curve25519()
    static let algorithmImplementationByteId =
PublicKeyEncryptionImplementationByteId.KEM_ECIES_Curve25519_and_DEM_CTR_AES_256_then_HMAC_
SHA_256
}
```

La fonction réalisant le chiffrement KEM est bien conforme à **[DOC_CRYPTO]**.

```
extension KEM_ECIES256KEM512 {
    [...]
    static func encrypt<T: SymmetricKey>(using _publicKey: PublicKeyForPublicKeyEncryption,
with _prng: PRNGService?, _ convertBytesToKey: (Data) -> T) -> (EncryptedData, T)? {
        guard let publicKey = _publicKey as? PublicKeyForPublicKeyEncryptionOnEdwardsCurve,
            publicKey.curveByteId == curve.byteId
            else { return nil }
        let prng = _prng ?? ObvCryptoSuite.sharedInstance.prngService()
        let r = BigInt(0)
        while r == BigInt(0) {
            r.set(prng.genBigInt(smallerThan: curve.parameters.q))
        }
        let B = curve.scalarMultiplication(scalar: r, point: curve.parameters.G)! // FIXME:
we assumed that this routine is faster than the one relying only on the y-coordinate
        let Dy: BigInt
        if publicKey.point != nil {
            let D = curve.scalarMultiplication(scalar: r, point: publicKey.point!)!
            Dy = D.y
        } else {
            Dy = curve.scalarMultiplication(scalar: r, yCoordinate: publicKey.yCoordinate)!
        }
        let c = try! Data(B.y, count: length)
        let ciphertext = EncryptedData(data: c)
        var rawSeed = c
        rawSeed.append(try! Data(Dy, count: length))
        guard let seed = Seed(with: rawSeed) else { return nil }
        guard let key = KDFFromPRNGWithHMACWithSHA256.generate(from: seed,
convertBytesToKey) else { return nil }
        return (ciphertext, key)
    }
    [...]
}
[...]
class KEM_ECIES256KEM512_WithCurve25519: KEM_ECIES256KEM512 {
    static let curve: EdwardsCurve = Curve25519()
}
```

La fonction réalisant le chiffrement symétrique est bien conforme à **[DOC_CRYPTO]**.

```
final class AuthenticatedEncryptionWithAES256CTRThenHMACWithSHA256 :
AuthenticatedEncryptionConcrete {
    [...]
    static func encrypt(_ plaintext: Data, with _key: AuthenticatedEncryptionKey, and
_prng: PRNG?) throws -> EncryptedData {
        guard let key = _key as? AuthenticatedEncryptionWithAES256CTRThenHMACWithSHA256Key
else { throw AuthenticatedEncryptionError.incorrectKey }
        let prng = _prng ?? ObvCryptoSuite.sharedInstance.prngService()
        // Encrypt...
        let iv = prng.genBytes(count: SymmetricEncryptionWithAES256CTR.ivLength)
        let ciphertext = try! SymmetricEncryptionWithAES256CTR.encrypt(plaintext, with:
key.aes256CTRKey, andIv: iv)
        // ... then authenticate
        let mac = try! HMACWithSHA256.compute(forData: ciphertext, withKey:
key.hmacWithSHA256Key)
        let authenticatedCiphertext = EncryptedData.byAppending(c1: ciphertext, c2:
EncryptedData(data: mac))
        return authenticatedCiphertext
    }
```

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

Ce chiffrement garantit qu'un message est bien délivré uniquement à son destinataire.

Établissement d'un secret partagé

Lors de l'établissement d'un secret partagé il est néanmoins nécessaire d'assurer de l'identité de l'émetteur. Cette garantie est apportée par une signature envoyée dans le message de ping du protocole *ChannelCreationWithContactDevice*.

```
final class SendPingStep: ProtocolStep, TypedConcreteProtocolStep {
    [...]
    override func executeStep(within obvContext: ObvContext) throws ->
ConcreteProtocolState? {
        [...]
        // Send a signed ping proving you trust the contact and have no channel with him
        let signature: Data
        do {
            let prefix = ChannelCreationWithContactDeviceProtocol.challengePrefix +
contactDeviceUid.raw + currentDeviceUid.raw
            let challenge = contactIdentity.getIdentity() + ownedIdentity.getIdentity()
            guard let res = solveChallengeDelegate.solveChallenge(challenge, prefixedWith:
prefix, for: ownedIdentity, using: prng) else {
                os_log("Could not solve challenge", log: log, type: .fault)
                return CancelledState()
            }
            signature = res
        }
        // Send the ping message containing the signature
        do {
            let coreMessage = getCoreMessage(for: .AsymmetricChannel(to: contactIdentity,
remoteDeviceUids: [contactDeviceUid], fromOwnedIdentity: ownedIdentity))
            let concreteProtocolMessage = PingMessage(coreProtocolMessage: coreMessage,
contactIdentity: ownedIdentity, contactDeviceUid: currentDeviceUid, signature: signature)
            guard let messageToSend =
concreteProtocolMessage.generateObvChannelProtocolMessageToSend(with: prng) else {
                return CancelledState()
            }
            _ = try channelDelegate.post(messageToSend, randomizedWith: prng, within:
obvContext)
        } catch {
            os_log("Could not post message", log: log, type: .fault)
            return CancelledState()
        }
        [...]
    }
}
```

Lors de la réception du message de *ping* la signature est vérifiée.

```
final class SendPingOrEphemeralKeyStep: ProtocolStep, TypedConcreteProtocolStep {
    [...]
    override func executeStep(within obvContext: ObvContext) throws ->
ConcreteProtocolState? {
        [...]
        // Verify the signature
        do {
            let currentDeviceUid = try
identityDelegate.getCurrentDeviceUidOfOwnedIdentity(ownedIdentity, within: obvContext)
```

```
            let prefix = ChannelCreationWithContactDeviceProtocol.challengePrefix +
currentDeviceUid.raw + contactDeviceUid.raw
            let challenge = ownedIdentity.getIdentity() + contactIdentity.getIdentity()
            guard solveChallengeDelegate.checkResponse(signature, toChallenge: challenge,
prefixedWith: prefix, from: contactIdentity) else {
                os_log("The signature is invalid", log: log, type: .error)
                return CancelledState()
            }
        } catch {
            os_log("Could not check the signature", log: log, type: .fault)
            return CancelledState()
        }        [...]
    }
}
```

La signature est calculée dans *solveChallengeDelegate.solveChallenge* et vérifiée dans *solveChallengeDelegate.checkResponse*.

```
extension ObvIdentityManagerImplementation: ObvSolveChallengeDelegate {
    public func solveChallenge(_ challenge: Data, prefixedWith prefix: Data, for identity:
ObvCryptoIdentity, using prng: PRNGService) -> Data? {
        // Fetch the crypto owned identity from the database
        var fetchedOwnedCryptoIdentity: ObvOwnedCryptoIdentity? = nil
        let randomFlowId = FlowIdentifier()
        delegateManager.contextCreator.performBackgroundTaskAndWait(flowId: randomFlowId) {
(obvContext) in
            if let ownedIdentity = OwnedIdentity.get(identity, delegateManager:
delegateManager, within: obvContext) {
                fetchedOwnedCryptoIdentity = ownedIdentity.ownedCryptoIdentity
            } else {
                os_log("Could not find an appropriate owned identity", log: log,
type: .fault)
                return
            }
        }
        guard let ownedCryptoIdentity = fetchedOwnedCryptoIdentity else {
            return nil
        }
        let serverAuth = ObvCryptoSuite.sharedInstance.authentication()
        let response = serverAuth.solve(challenge,
                                        prefixedWith: prefix,
                                        with:
ownedCryptoIdentity.privateKeyForAuthentication,
                                        and:
ownedCryptoIdentity.publicKeyForAuthentication,
                                        using: prng)
        if response == nil {
            os_log("Could not compute the challenge's response", log: log, type: .error)
        }
        return response
    }
    public func checkResponse(_ response: Data, toChallenge challenge: Data, prefixedWith
prefix: Data, from identity: ObvCryptoIdentity) -> Bool {
        let serverAuth = ObvCryptoSuite.sharedInstance.authentication()
        return serverAuth.check(response: response, toChallenge: challenge, prefixedWith:
prefix, using: identity.publicKeyForAuthentication)
    }
    [...]
}
```

La construction de la signature est réalisée par *ObvCryptoSuite.sharedInstance.authentication,* qui vaut par défaut *AuthenticationFromSignatureOnMDC*. Ceci est vérifié par l'analyse dynamique.

```
extension AuthenticationFromSignatureOnEdwardsCurve {
    static func generateKeyPair(with prng: PRNGService) -> (PublicKeyForAuthentication,
PrivateKeyForAuthentication) {
        let (scalar, point) = curve.generateRandomScalarAndPoint(withPRNG: prng)
        let publicKey = PublicKeyForAuthenticationFromSignatureOnEdwardsCurve(point:
point)!
        let privateKey = PrivateKeyForAuthenticationFromSignatureOnEdwardsCurve(scalar:
scalar, curveByteId: curve.byteId)
        return (publicKey, privateKey)
    }
    static func solve(_ challenge: Data, prefixedWith prefix: Data, with _privateKey:
PrivateKeyForAuthentication, and _publicKey: PublicKeyForAuthentication, using prng:
PRNGService) -> Data? {
        guard let publicKey = _publicKey as?
PublicKeyForAuthenticationFromSignatureOnEdwardsCurve else { return nil }
        guard let privateKey = _privateKey as?
PrivateKeyForAuthenticationFromSignatureOnEdwardsCurve else { return nil }
        let randomSuffix = prng.genBytes(count:
AuthenticationFromSignatureOnEdwardsCurveConstants.lengthOfRandomFormattedChallengeSuffix)
        var formattedChallenge = prefix
//AuthenticationFromSignatureOnEdwardsCurveConstants.formattedChallengePrefix
        formattedChallenge.append(challenge) // FIXME: check challenge length?
        formattedChallenge.append(randomSuffix)
        guard let signature = Signature.sign(formattedChallenge, with:
privateKey.privateKeyForSignatureOnEdwardsCurve, and:
publicKey.publicKeyForSignatureOnEdwardsCurve, using: prng) else { return nil }
        var response = randomSuffix
        response.append(signature)
        return response
    }
    static func check(response: Data, toChallenge challenge: Data, prefixedWith prefix:
Data, using _publicKey: PublicKeyForAuthentication) -> Bool {
        guard let publicKey = _publicKey as?
PublicKeyForAuthenticationFromSignatureOnEdwardsCurve else { return false }
        guard response.count >
AuthenticationFromSignatureOnEdwardsCurveConstants.lengthOfRandomFormattedChallengeSuffix
else { return false }
        let randomSuffix = response[response.startIndex..<response.startIndex +
AuthenticationFromSignatureOnEdwardsCurveConstants.lengthOfRandomFormattedChallengeSuffix]
        var formattedChallenge = prefix
        let signature = response[response.startIndex +
AuthenticationFromSignatureOnEdwardsCurveConstants.lengthOfRandomFormattedChallengeSuffix..
<response.endIndex]
        formattedChallenge.append(challenge)
        formattedChallenge.append(randomSuffix)
        return Signature.verify(signature, on: formattedChallenge, with:
publicKey.publicKeyForSignatureOnEdwardsCurve) ?? false
    }
}
final class AuthenticationFromSignatureOnMDC: AuthenticationFromSignatureOnEdwardsCurve {
    static let algorithmImplementationByteId =
AuthenticationImplementationByteId.Signature_with_EC_SDSA_with_MDC
    static let curve: EdwardsCurve = CurveMDC()
}
```

Le calcul de la signature est quant à lui réalisé conformément à **[DOC_CRYPO]** par *SignatureECSDSA256overMDC*.

```
public final class Signature: SignatureGeneric {
    [...]
    static func sign(_ data: Data, with privateKey: PrivateKeyForSignature, and publicKey:
PublicKeyForSignature, using prng: PRNGService) -> Data? {
        let algorithmImplementation =
privateKey.algorithmImplementationByteId.algorithmImplementation
        return algorithmImplementation.sign(data, with: privateKey, and: publicKey, using:
prng)
    }
    static func verify(_ signature: Data, on data: Data, with publicKey:
PublicKeyForSignature) -> Bool? {
        let algorithmImplementation =
publicKey.algorithmImplementationByteId.algorithmImplementation
        return algorithmImplementation.verify(signature, on: data, with: publicKey)
    }
}
protocol SignatureECSDSA256overEdwardsCurve: SignatureConcrete {
    [...]
}
extension SignatureECSDSA256overEdwardsCurve {
    [...]
    static func sign(_ message: Data, with _privateKey: PrivateKeyForSignature, and
_publicKey: PublicKeyForSignature, using prng: PRNGService) -> Data? {
        guard let privateKey = _privateKey as? PrivateKeyForSignatureOnEdwardsCurve else
{ return nil }
        guard let publicKey = _publicKey as? PublicKeyForSignatureOnEdwardsCurve else
{ return nil }
        guard publicKey.curveByteId == privateKey.curveByteId else { return nil }
        guard publicKey.curveByteId == curve.byteId else { return nil }
        let algorithmImplementation =
privateKey.algorithmImplementationByteId.algorithmImplementation
        let (localPublicKey, localPrivateKey) =
algorithmImplementation.generateKeyPair(with: prng) as!
(PublicKeyForSignatureOnEdwardsCurve, PrivateKeyForSignatureOnEdwardsCurve)
        // Construct the data to hash and sign
        let pLength = curve.parameters.p.byteSize()
        var dataToHash = localPublicKey.yCoordinate.encode(withInnerLength:
pLength)!.innerData
        dataToHash.append(publicKey.yCoordinate.encode(withInnerLength:
pLength)!.innerData)
        dataToHash.append(message)
        // Hash and cast as a big integer
        let h = SHA256.hash(dataToHash)
        let e = BigInt(ObvEncoded.init(byteId: .unsignedBigInt, innerData: h))!
        // sign
        let q = BigInt(curve.parameters.q)
        let y = BigInt(localPrivateKey.scalar).sub(BigInt(privateKey.scalar).mul(e, modulo:
q), modulo: q) // y = (r - a*e) % q
        let z = y.encode(withInnerLength: zLength)!.innerData
        var signature = h
        signature.append(z)
        return signature
    }
    static func verify(_ signature: Data, on message: Data, with _publicKey:
PublicKeyForSignature) -> Bool? {
        guard let publicKey = _publicKey as? PublicKeyForSignatureOnEdwardsCurve else
{ return nil }
        guard publicKey.curveByteId == curve.byteId else { return nil }
        guard signature.count == self.hLength + self.zLength else { return false }
        let hRange = signature.startIndex..<signature.startIndex+self.hLength
```

```
        let h = signature[hRange]
        let zRange = signature.startIndex+self.hLength..<signature.endIndex
        let z = signature[zRange]
        guard z.count == self.zLength else { return nil }
        let e = BigInt(h)
        let y = BigInt(z)
        let resultingPoints: (point1: PointOnCurve, point2: PointOnCurve)
        if let point = publicKey.point {
            let resultingPoint = curve.mulAdd(a: y, point1: curve.parameters.G, b: e,
point2: point)!
            resultingPoints = (resultingPoint, resultingPoint)
        } else {
            resultingPoints = curve.mulAdd(a: y, point1: curve.parameters.G, b: e,
yCoordinateOfPoint2: publicKey.yCoordinate)!
        }
        let pLength = curve.parameters.p.byteSize()
        let Ay = publicKey.yCoordinate.encode(withInnerLength: pLength)!.innerData
        let Ay1 = resultingPoints.point1.y.encode(withInnerLength: pLength)!.innerData
        let Ay2 = resultingPoints.point2.y.encode(withInnerLength: pLength)!.innerData
        var dataToHash1 = Ay1
        dataToHash1.append(Ay)
        dataToHash1.append(message)
        var dataToHash2 = Ay2
        dataToHash2.append(Ay)
        dataToHash2.append(message)
        let h1 = SHA256.hash(dataToHash1)
        let h2 = SHA256.hash(dataToHash2)
        return h == h1 || h == h2
    }
}
final class SignatureECSDSA256overMDC: SignatureECSDSA256overEdwardsCurve {
    static let algorithmImplementationByteId =
SignatureImplementationByteId.EC_SDSA_with_MDC
    static let curve: EdwardsCurve = CurveMDC()
}
```

Le même mécanisme de signature est utilisé en réponse au premier message de *ping*.

```
final class SendPingOrEphemeralKeyStep: ProtocolStep, TypedConcreteProtocolStep {
    [...]
    override func executeStep(within obvContext: ObvContext) throws ->
ConcreteProtocolState? {
        [...]
        // Compute a signature to prove we trust the contact and don't have any
channel/ongoing protocol with him
        let ownSignature: Data
        do {
            let prefix = ChannelCreationWithContactDeviceProtocol.challengePrefix +
contactDeviceUid.raw + currentDeviceUid.raw
            let challenge = contactIdentity.getIdentity() + ownedIdentity.getIdentity()
            guard let res = solveChallengeDelegate.solveChallenge(challenge, prefixedWith:
prefix, for: ownedIdentity, using: prng) else {
                os_log("Could not solve challenge (1)", log: log, type: .fault)
                return CancelledState()
            }
            ownSignature = res
        }
        // If we are "in charge" (small device uid), send an ephemeral key.
        // Otherwise, simply send a ping back
        if currentDeviceUid >= contactDeviceUid {
```

```
            [...]
            // Send the ping message containing the signature
            do {
                let coreMessage = getCoreMessage(for: .AsymmetricChannel(to:
contactIdentity, remoteDeviceUids: [contactDeviceUid], fromOwnedIdentity: ownedIdentity))
                let concreteProtocolMessage = PingMessage(coreProtocolMessage: coreMessage,
contactIdentity: ownedIdentity, contactDeviceUid: currentDeviceUid, signature:
ownSignature)
                guard let messageToSend =
concreteProtocolMessage.generateObvChannelProtocolMessageToSend(with: prng) else { return
nil }
                _ = try channelDelegate.post(messageToSend, randomizedWith: prng, within:
obvContext)
            } catch {
                os_log("Could not post message", log: log, type: .fault)
                return CancelledState()
            }
            [...]
        } else {
            [...]
            do {
                let coreMessage = getCoreMessage(for: .AsymmetricChannel(to:
contactIdentity, remoteDeviceUids: [contactDeviceUid], fromOwnedIdentity: ownedIdentity))
                let concreteProtocolMessage =
AliceIdentityAndEphemeralKeyMessage(coreProtocolMessage: coreMessage,

contactIdentity: ownedIdentity,

contactDeviceUid: currentDeviceUid,

signature: ownSignature,

contactEphemeralPublicKey: ephemeralPublicKey)
                guard let messageToSend =
concreteProtocolMessage.generateObvChannelProtocolMessageToSend(with: prng) else { return
nil }
                _ = try channelDelegate.post(messageToSend, randomizedWith: prng, within:
obvContext)
            }
            [...]
        }
    }
}
```

Et la vérification est de nouveau réalisée à la réception de ce deuxième message.

```
final class SendEphemeralKeyAndK1Step: ProtocolStep, TypedConcreteProtocolStep {
    [...]
    override func executeStep(within obvContext: ObvContext) throws ->
ConcreteProtocolState? {
        [...]
        // Verify the signature
        do {
            let currentDeviceUid = try
identityDelegate.getCurrentDeviceUidOfOwnedIdentity(ownedIdentity, within: obvContext)
            let prefix = ChannelCreationWithContactDeviceProtocol.challengePrefix +
currentDeviceUid.raw + contactDeviceUid.raw
            let challenge = ownedIdentity.getIdentity() + contactIdentity.getIdentity()
            guard solveChallengeDelegate.checkResponse(signature, toChallenge: challenge,
prefixedWith: prefix, from: contactIdentity) else {
```

```
                os_log("The signature is invalid", log: log, type: .error)
                return CancelledState()
            }
        } catch {
            os_log("Could not check the signature", log: log, type: .fault)
            return CancelledState()
        }
        [...]
    }
}
```

Dans la suite des échanges du protocole *ChannelCreationWithContactDevice* les deux participants utilisent ensuite des clés asymétriques temporaires pour échanger via deux KEM deux clés symétriques.

```
final class SendEphemeralKeyAndK1Step: ProtocolStep, TypedConcreteProtocolStep {
    [...]
    override func executeStep(within obvContext: ObvContext) throws ->
ConcreteProtocolState? {
        [...]
        // Generate k1
        let (c1, k1) = PublicKeyEncryption.kemEncrypt(using: contactEphemeralPublicKey,
with: prng)
        // Send the ephemeral public key and k1 to Alice
        do {
            let coreMessage = getCoreMessage(for: .AsymmetricChannel(to: contactIdentity,
remoteDeviceUids: [contactDeviceUid], fromOwnedIdentity: ownedIdentity))
            let concreteProtocolMessage = BobEphemeralKeyAndK1Message(coreProtocolMessage:
coreMessage,

contactEphemeralPublicKey: ephemeralPublicKey,
                                                                    c1: c1)
            guard let messageToSend =
concreteProtocolMessage.generateObvChannelProtocolMessageToSend(with: prng) else { return
nil }
            _ = try channelDelegate.post(messageToSend, randomizedWith: prng, within:
obvContext)
        }
        [...]
    }
}
```

```
final class RecoverK1AndSendK2AndCreateChannelStep: ProtocolStep, TypedConcreteProtocolStep
{
    [...]
    override func executeStep(within obvContext: ObvContext) throws ->
ConcreteProtocolState? {
        [...]
        // Recover k1
        guard let k1 = PublicKeyEncryption.kemDecrypt(c1, using: ephemeralPrivateKey) else
{
                os_log("Could not recover k1", log: log, type: .error)
                return CancelledState()
        }
        // Generate k2
        let (c2, k2) = PublicKeyEncryption.kemEncrypt(using: contactEphemeralPublicKey,
with: prng)
        [...]
        // Send the k2 to Bob
```

```
        do {
            let coreMessage = getCoreMessage(for: .AsymmetricChannel(to: contactIdentity,
remoteDeviceUids: [contactDeviceUid], fromOwnedIdentity: ownedIdentity))
            let concreteProtocolMessage = K2Message(coreProtocolMessage: coreMessage, c2:
c2)
            guard let messageToSend =
concreteProtocolMessage.generateObvChannelProtocolMessageToSend(with: prng) else { return
nil }
            _ = try channelDelegate.post(messageToSend, randomizedWith: prng, within:
obvContext)
        }
        [...]
    }
}
```

Le KEM utilisé pour l'échange de ces clés est le même que celui revu précédemment.

Ces deux clés sont utilisées pour dériver une clé symétrique commune.

```
final class RecoverK1AndSendK2AndCreateChannelStep: ProtocolStep, TypedConcreteProtocolStep
{
    [...]
    override func executeStep(within obvContext: ObvContext) throws ->
ConcreteProtocolState? {
        [...]
        // Create the Oblivious Channel using the seed derived from k1 and k2
        do {
            guard let seed = Seed(withKeys: [k1, k2]) else {
                os_log("Could not initialize seed for Oblivious Channel", log: log,
type: .error)
                return CancelledState()
            }
            let cryptoSuiteVersion = 0 // FIXME: Should not be hardcoded
            try
channelDelegate.createObliviousChannelBetweenTheCurrentDeviceOf(ownedIdentity:
ownedIdentity,

andRemoteIdentity: contactIdentity,

withRemoteDeviceUid: contactDeviceUid,
                                                                        with: seed,

cryptoSuiteVersion: cryptoSuiteVersion,
                                                                        within:

obvContext)
        }
        [...]
    }
}
```

Le seed est calculé conformément à **[DOC_CRYPTO]** dans la fonction *Seed*.

```
public final class Seed: NSObject, NSCopying, ObvCodable {
    [...]
    public convenience init?(withKeys keys: [AuthenticatedEncryptionKey]) {
        guard keys.count > 0 else { return nil }
        let zeroSeedRaw = Data(repeating: 0, count: Seed.minLength)
        let zeroSeed = Seed(with: zeroSeedRaw)!
        let prng = PRNGWithHMACWithSHA256(with: zeroSeed)
        let fixedBlockOfSeedLength = Data(repeating: 0x00, count: Seed.minLength)
```

```
        let ciphertexts = keys.map
{ AuthenticatedEncryption.encrypt(fixedBlockOfSeedLength, with: $0, and: prng) }
        let toHash = ciphertexts.reduce(Data()) { return $0 + $1.raw }
        let seed = SHA256.hash(toHash)
        self.init(with: seed)
    }
}
```

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. Il est donc nécessaire de passer en revue le protocole *ChannelCreationWithContactDevice.*

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

```
struct WaitingForK1State: TypeConcreteProtocolState { ... }
struct WaitForFirstAckState: TypeConcreteProtocolState { ... }
struct WaitingForK2State: TypeConcreteProtocolState { ... }
struct WaitForSecondAckState: TypeConcreteProtocolState { ... }
struct PingSentState: TypeConcreteProtocolState { ... }
struct ChannelConfirmedState: TypeConcreteProtocolState { ... }
struct CancelledState: TypeConcreteProtocolState { ... }
```

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

```
final class SendPingStep: ProtocolStep, TypedConcreteProtocolStep { ... }
final class SendPingOrEphemeralKeyStep: ProtocolStep, TypedConcreteProtocolStep { ... }
final class SendEphemeralKeyAndK1Step: ProtocolStep, TypedConcreteProtocolStep { ... }
final class RecoverK1AndSendK2AndCreateChannelStep: ProtocolStep, TypedConcreteProtocolStep
{ ... }
final class RecoverK2CreateChannelAndSendAckStep: ProtocolStep, TypedConcreteProtocolStep {
... }
final class ConfirmChannelAndSendAckStep: ProtocolStep, TypedConcreteProtocolStep { ... }
final class ConfirmChannelStep: ProtocolStep, TypedConcreteProtocolStep { ... }
```

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

```
struct InitialMessage: ConcreteProtocolMessage { ... }
struct PingMessage: ConcreteProtocolMessage { ... }
struct AliceIdentityAndEphemeralKeyMessage: ConcreteProtocolMessage { ... }
struct BobEphemeralKeyAndK1Message: ConcreteProtocolMessage { ... }
struct K2Message: ConcreteProtocolMessage { ... }
struct FirstAckMessage: ConcreteProtocolMessage { ... }
struct SecondAckMessage: ConcreteProtocolMessage { ... }
```

Les différentes étapes définies dans le cas du protocole *ChannelCreationWithContactDevice* ont les constructeurs suivants:

```
init?(startState: ConcreteProtocolInitialState, receivedMessage:
ChannelCreationWithContactDeviceProtocol.InitialMessage, concreteCryptoProtocol:
ConcreteCryptoProtocol) { ... }
init?(startState: ConcreteProtocolInitialState, receivedMessage:
ChannelCreationWithContactDeviceProtocol.PingMessage, concreteCryptoProtocol:
ConcreteCryptoProtocol) { ... }
init?(startState: ConcreteProtocolInitialState, receivedMessage:
ChannelCreationWithContactDeviceProtocol.AliceIdentityAndEphemeralKeyMessage,
concreteCryptoProtocol: ConcreteCryptoProtocol) { ... }
```

```
init?(startState: WaitingForK1State, receivedMessage:
ChannelCreationWithContactDeviceProtocol.BobEphemeralKeyAndK1Message,
concreteCryptoProtocol: ConcreteCryptoProtocol) { ... }
init?(startState: WaitingForK2State, receivedMessage:
ChannelCreationWithContactDeviceProtocol.K2Message, concreteCryptoProtocol:
ConcreteCryptoProtocol) { ... }
init?(startState: WaitForFirstAckState, receivedMessage:
ChannelCreationWithContactDeviceProtocol.FirstAckMessage, concreteCryptoProtocol:
ConcreteCryptoProtocol) { ... }
init?(startState: WaitForSecondAckState, receivedMessage:
ChannelCreationWithContactDeviceProtocol.SecondAckMessage, concreteCryptoProtocol:
ConcreteCryptoProtocol) { ... }
```

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.

```
SendPingStep -> PingSentState
SendPingOrEphemeralKeyStep -> PingSentState, WaitingForK1State
SendEphemeralKeyAndK1Step -> WaitingForK2State
RecoverK1AndSendK2AndCreateChannelStep -> WaitForFirstAckState
RecoverK2CreateChannelAndSendAckStep -> WaitForSecondAckState
ConfirmChannelAndSendAckStep -> ChannelConfirmedState
ConfirmChannelStep -> ChannelConfirmedState
```

Lors des échanges, Alice est choisie comme étant le participant ayant le DeviceUID le plus bas.

Conformément à **[DOC_CRYPTO]** la première étape du protocole *ChannelCreationWithContactDevice* consiste pour Alice à envoyer un *ping* contenant une signature composée des *deviceUid* des deux participants.

```
override func executeStep(within obvContext: ObvContext) throws -> ConcreteProtocolState? {
    [...]
    // Send a signed ping proving you trust the contact and have no channel with him
    let signature: Data
    do {
        let prefix = ChannelCreationWithContactDeviceProtocol.challengePrefix +
contactDeviceUid.raw + currentDeviceUid.raw
        let challenge = contactIdentity.getIdentity() + ownedIdentity.getIdentity()
        guard let res = solveChallengeDelegate.solveChallenge(challenge, prefixedWith:
prefix, for: ownedIdentity, using: prng) else {
            os_log("Could not solve challenge", log: log, type: .fault)
            return CancelledState()
        }
        signature = res
    }
    [...]
}
```

L'étape suivante consiste pour Bob à vérifier la signature envoyée par Alice puis à envoyer un message *ping* à son tour contenant une signature composée des *deviceUid* des deux participants ainsi qu'une clé publique éphémère.

```
override func executeStep(within obvContext: ObvContext) throws -> ConcreteProtocolState? {
    [...]
    // Verify the signature
    do {
        let currentDeviceUid = try
identityDelegate.getCurrentDeviceUidOfOwnedIdentity(ownedIdentity, within: obvContext)
```

```
        let prefix = ChannelCreationWithContactDeviceProtocol.challengePrefix +
currentDeviceUid.raw + contactDeviceUid.raw
        let challenge = ownedIdentity.getIdentity() + contactIdentity.getIdentity()
        guard solveChallengeDelegate.checkResponse(signature, toChallenge: challenge,
prefixedWith: prefix, from: contactIdentity) else {
            os_log("The signature is invalid", log: log, type: .error)
            return CancelledState()
        }
    } catch {
        os_log("Could not check the signature", log: log, type: .fault)
        return CancelledState()
    }
    [...]
    // Compute a signature to prove we trust the contact and don't have any channel/ongoing
protocol with him
    let ownSignature: Data
    do {
        let prefix = ChannelCreationWithContactDeviceProtocol.challengePrefix +
contactDeviceUid.raw + currentDeviceUid.raw
        let challenge = contactIdentity.getIdentity() + ownedIdentity.getIdentity()
        guard let res = solveChallengeDelegate.solveChallenge(challenge, prefixedWith:
prefix, for: ownedIdentity, using: prng) else {
            os_log("Could not solve challenge (1)", log: log, type: .fault)
            return CancelledState()
        }
        ownSignature = res
    }
    [...]
        // Generate an ephemeral pair of encryption keys
        let ephemeralPublicKey: PublicKeyForPublicKeyEncryption
        let ephemeralPrivateKey: PrivateKeyForPublicKeyEncryption
        do {
            let PublicKeyEncryptionImplementation =
ObvCryptoSuite.sharedInstance.getDefaultPublicKeyEncryptionImplementationByteId().algorithm
Implementation
            (ephemeralPublicKey, ephemeralPrivateKey) =
PublicKeyEncryptionImplementation.generateKeyPair(with: prng)
        }
        // Send the public key to Bob, together with our own identity and current device
uid
        do {
            let coreMessage = getCoreMessage(for: .AsymmetricChannel(to: contactIdentity,
remoteDeviceUids: [contactDeviceUid], fromOwnedIdentity: ownedIdentity))
            let concreteProtocolMessage =
AliceIdentityAndEphemeralKeyMessage(coreProtocolMessage: coreMessage,

contactIdentity: ownedIdentity,

contactDeviceUid: currentDeviceUid,
                                                                                signature:
ownSignature,

contactEphemeralPublicKey: ephemeralPublicKey)
            guard let messageToSend =
concreteProtocolMessage.generateObvChannelProtocolMessageToSend(with: prng) else { return
nil }
            _ = try channelDelegate.post(messageToSend, randomizedWith: prng, within:
obvContext)
        }
    [...]
```

```
}
```

L'étape suivante consiste pour Alice à vérifier la signature de Bob et à envoyer une clé secrète chiffrée avec la clé publique envoyée par Bob ainsi qu'une clé publique éphémère.

```
override func executeStep(within obvContext: ObvContext) throws -> ConcreteProtocolState? {
    [...]
    // Verify the signature
    do {
        let currentDeviceUid = try
identityDelegate.getCurrentDeviceUidOfOwnedIdentity(ownedIdentity, within: obvContext)
        let prefix = ChannelCreationWithContactDeviceProtocol.challengePrefix +
currentDeviceUid.raw + contactDeviceUid.raw
        let challenge = ownedIdentity.getIdentity() + contactIdentity.getIdentity()
        guard solveChallengeDelegate.checkResponse(signature, toChallenge: challenge,
prefixedWith: prefix, from: contactIdentity) else {
            os_log("The signature is invalid", log: log, type: .error)
            return CancelledState()
        }
    } catch {
        os_log("Could not check the signature", log: log, type: .fault)
        return CancelledState()
    }
    [...]
    // Generate an ephemeral pair of encryption keys
    let ephemeralPublicKey: PublicKeyForPublicKeyEncryption
    let ephemeralPrivateKey: PrivateKeyForPublicKeyEncryption
    do {
        let PublicKeyEncryptionImplementation =
ObvCryptoSuite.sharedInstance.getDefaultPublicKeyEncryptionImplementationByteId().algorithm
Implementation
        (ephemeralPublicKey, ephemeralPrivateKey) =
PublicKeyEncryptionImplementation.generateKeyPair(with: prng)
    }
    // Generate k1
    let (c1, k1) = PublicKeyEncryption.kemEncrypt(using: contactEphemeralPublicKey, with:
prng)
    // Send the ephemeral public key and k1 to Alice
    do {
        let coreMessage = getCoreMessage(for: .AsymmetricChannel(to: contactIdentity,
remoteDeviceUids: [contactDeviceUid], fromOwnedIdentity: ownedIdentity))
        let concreteProtocolMessage = BobEphemeralKeyAndK1Message(coreProtocolMessage:
coreMessage,

contactEphemeralPublicKey: ephemeralPublicKey,
                                                                  c1: c1)
        guard let messageToSend =
concreteProtocolMessage.generateObvChannelProtocolMessageToSend(with: prng) else { return
nil }
        _ = try channelDelegate.post(messageToSend, randomizedWith: prng, within:
obvContext)
    }
    [...]
}
```

L'étape suivante consiste pour Bob à envoyer une clé secrète chiffrée avec la clé publique éphémère d'Alice.

```
override func executeStep(within obvContext: ObvContext) throws -> ConcreteProtocolState? {
    [...]
    // Generate k2
```

```
    let (c2, k2) = PublicKeyEncryption.kemEncrypt(using: contactEphemeralPublicKey, with:
prng)
    [...]
    // Send the k2 to Bob
    do {
        let coreMessage = getCoreMessage(for: .AsymmetricChannel(to: contactIdentity,
remoteDeviceUids: [contactDeviceUid], fromOwnedIdentity: ownedIdentity))
        let concreteProtocolMessage = K2Message(coreProtocolMessage: coreMessage, c2: c2)
        guard let messageToSend =
concreteProtocolMessage.generateObvChannelProtocolMessageToSend(with: prng) else { return
nil }
        _ = try channelDelegate.post(messageToSend, randomizedWith: prng, within:
obvContext)
    }
    [...]
}
```

Enfin les deux participants utilisent les deux clés secrètes pour dériver une clé secrète unique.

```
// Create the Oblivious Channel using the seed derived from k1 and k2
do {
    guard let seed = Seed(withKeys: [k1, k2]) else {
        os_log("Could not initialize seed for Oblivious Channel", log: log, type: .error)
        return CancelledState()
    }
    let cryptoSuiteVersion = 0 // FIXME: Should not be hardcoded
    try channelDelegate.createObliviousChannelBetweenTheCurrentDeviceOf(ownedIdentity:
ownedIdentity,
                                                                        andRemoteIdentity:
contactIdentity,

withRemoteDeviceUid: contactDeviceUid,
                                                                        with: seed,
                                                                        cryptoSuiteVersion:
cryptoSuiteVersion,
                                                                        within: obvContext)
}
```

L'analyse du code source réalisant l'authentification des messages ne met en exergue aucun problème d'implémentation.

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,
421c03328ea6b8caee6b42a034c694eff519fd38ecc2ee23e7047a9ed1e9c8da12f8dcb93d531e840ace52ff2c7
cc87d4c6efa0529ce52b3212b22964a70484d2ffd331192e4a677b0e0b32c8733493ca3c1aa2b68f20b6aa8f52d
94a167162c2fb8cb75816cf625ffce02da2c199a4f3152d73a9531b4f988f5f2f4fba3b8cfbbc172d6d933b1c5c
3b46a1e9a2a96892f9a2531f1e1c3af7283df0e13861fdc857906942593894d01a8b6887516215f618a696138a0
85a6ad,
68747470733a2f2f7365727665722e6f6c7669642e696f000023569981ae071758ec2ef28eae66ae9df5223e2fd
a5919fb911ff26daa9026fa010b3b4561f33b73c67c56a5e9e8a6073897c9a675f071dc2073757472bcb13a95]
Message:
862bc7dff04f529fd384f093ad76fad9517e630b35b785f76242c51d2869a4a0ba5fdb1c98bb38ae52e5820c07c
6cb44bc9e838dd36beff5061ea7298679d37380d160c8670a3ad9682a31a12f6b9df528b90cda71e19a3637bf93
186ec45a18a563cba2981ecb8d8095c041bfc6e3cd76e91a36be03b8f4d00bdd170b761cb1f2e4ca481e52081b0
```

```
dc0f82bab1e3d9c91b94dc2efaa9fde5e652b15fd5c0765a3b04faf0fea6dc7718f87eb25cc87e122773951b58a
648fdd6e3347fd9fca5b82626da75d26ce3b187bc58af27129b3b2d5c3a0a06af3aadd0d4fc9382e5b47343bb12
26dae02e44b02389bf1c0255b9d903bc4f2c64bdef654c91c097c0e5591739ca885a374e155c67e0c64b3c64c71
14aa0e6dfee57feaf7f3667a4ee6f3d04232695b07c158cf843d2c7620ece00494c296a08fadf51ad20cfc67602
4afd1500dfedfacbc37f1fb22fd
```

Le message est décomposable en deux parties, un en-tête 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'en-tête contenant le chiffré des clés du message est de 6.88 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'en-tête.

Il est possible de confirmer que les clés sont correctement chiffrées en récupérant les clés privées des participants directement depuis la base de donnée *SQLite* de l'application et en ré-implémentant 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={'enckey':
1427df9dd1dac89eed6aeb03791b54dfcd547ad803cff5e90d0ef0daaadcf9c6, 'mackey':
c167db6fda919978142ac770f60efe33e5ee10716f7754bc774a76bcd046fb99}>
```

Établissement du chiffrement symétrique

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

```
To:
68747470733a2f2f7365727665722e6f6c7669642e696f00002c56c7c4b256d707b3d258c9266942b9eec363651
2f8546bfc7773c5e65ddef50107b054f42552f293df32561630cf960311b268cd925881fbeee073e1ec06a0e2
(f4ee3ae632595d6833b024ae51805fb7403d686560b26246a92a2c8391c37a0d)
Protocol Message
  ChannelCreationWithContactDevice
    Ping
      Contact identity:
 68747470733a2f2f7365727665722e6f6c7669642e696f000023569981ae071758ec2ef28eae66ae9df5223e2f
da5919fb911ff26daa9026fa010b3b4561f33b73c67c56a5e9e8a6073897c9a675f071dc2073757472bcb13a95
      Contact device UIDs: 48e7cb75d852123700690e6eb0552b1dd026e7c2b23ecd57c410788010e22ed7
      Signature:
0eb6d84e40f3ac7c5a0f5cee8b7c46380d60a3600f31283142ed6de38f6d23b22245c8699b907291bea2fea904b
fe76e208ef16e5ac6fcd04ca6b19e62a79850ccb64372feba23a615c41d7115cd83c5

To:
68747470733a2f2f7365727665722e6f6c7669642e696f000023569981ae071758ec2ef28eae66ae9df5223e2fd
a5919fb911ff26daa9026fa010b3b4561f33b73c67c56a5e9e8a6073897c9a675f071dc2073757472bcb13a95
(48e7cb75d852123700690e6eb0552b1dd026e7c2b23ecd57c410788010e22ed7)
Protocol Message
  ChannelCreationWithContactDevice
    Ping
      Contact identity:
 68747470733a2f2f7365727665722e6f6c7669642e696f00002c56c7c4b256d707b3d258c9266942b9eec36365
12f8546bfc7773c5e65ddef50107b054f42552f293df32561630cf960311b268cd925881fbeee073e1ec06a0e2
      Contact device UIDs: f4ee3ae632595d6833b024ae51805fb7403d686560b26246a92a2c8391c37a0d
      Signature:
696fd9a59e17d7d6cd5521e2aa536fc4454e9902de651b41b9f38f7ddd7725d575192d016f28d27cb2aab3e4946
e3ec2181b9d61b3f077b9c0327b2d2e56c1bb5d780944e908adf8711483a76c13a22b
```

```
To:
68747470733a2f2f7365727665722e6f6c7669642e696f00002c56c7c4b256d707b3d258c9266942b9eec363651
2f8546bfc7773c5e65ddef50107b054f42552f293df32561630cf960311b268cd925881fbeee073e1ec06a0e2
(f4ee3ae632595d6833b024ae51805fb7403d686560b26246a92a2c8391c37a0d)
Protocol Message
  ChannelCreationWithContactDevice
    Alice identity and ephemeral key
      Contact identity:
 68747470733a2f2f7365727665722e6f6c7669642e696f000023569981ae071758ec2ef28eae66ae9df5223e2f
da5919fb911ff26daa9026fa010b3b4561f33b73c67c56a5e9e8a6073897c9a675f071dc2073757472bcb13a95
      Contact device UIDs: 48e7cb75d852123700690e6eb0552b1dd026e7c2b23ecd57c410788010e22ed7
      Signature:
6a794c351e23ad22d8792f66c409e048eacd73ac47eaa147db82744ac7b09b7c7d117e2d8dfa7a67baa1398139b
934cb2d8fbea6c6b841bdc952e2eb1afae06bfc53c3a5fa81dfee6875613bb4ab86a9
      Ephemeral key:       <PublicKey algoClassByteId=18 algoImplemByteId=1 dict={'x':
50227081471801935424680307668414108929720222678363096276727708544469853607078, 'y':
19843763805181591759031475882412908708676568161701221348129776557598242440 86}>

To:
68747470733a2f2f7365727665722e6f6c7669642e696f000023569981ae071758ec2ef28eae66ae9df5223e2fd
a5919fb911ff26daa9026fa010b3b4561f33b73c67c56a5e9e8a6073897c9a675f071dc2073757472bcb13a95
(48e7cb75d852123700690e6eb0552b1dd026e7c2b23ecd57c410788010e22ed7)
Protocol Message
  ChannelCreationWithContactDevice
    Bob ephemeral key
      Ephemeral key: <PublicKey algoClassByteId=18 algoImplemByteId=1 dict={'y':
51348980498648686602649848333822776253939416079074064445512741830793889317140, 'x':
39171653284684508147455405502406055395536639675193158083281432029081785170323}>
      c1:            0d0dd9a8e1b90d61a80a1831af747085a53e4a9e23e0c84abc4c221ba9d083fc

To:
68747470733a2f2f7365727665722e6f6c7669642e696f00002c56c7c4b256d707b3d258c9266942b9eec363651
2f8546bfc7773c5e65ddef50107b054f42552f293df32561630cf960311b268cd925881fbeee073e1ec06a0e2
(f4ee3ae632595d6833b024ae51805fb7403d686560b26246a92a2c8391c37a0d)
Protocol Message
  ChannelCreationWithContactDevice
    K2
      c2:            2fc965518192a4504e418ecf184b3c4f84feb1b187c3c7a0aa2aee20c5478b88
```

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 15 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 14, il est nécessaire de vérifier que les autres implémentations sont correctes.

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

- L'utilisation de primitives cryptographiques testées et éprouvées;

- Une ré-implémentation 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 conséquences, 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 au RGS 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 15. Les clés générées spécifiquement pour chaque message sont à leur tour chiffrées via du chiffrement asymétrique puis ajoutées dans l'en-tête du message. Le schéma suivant présente le fonctionnement du chiffrement d'un message protocolaire.

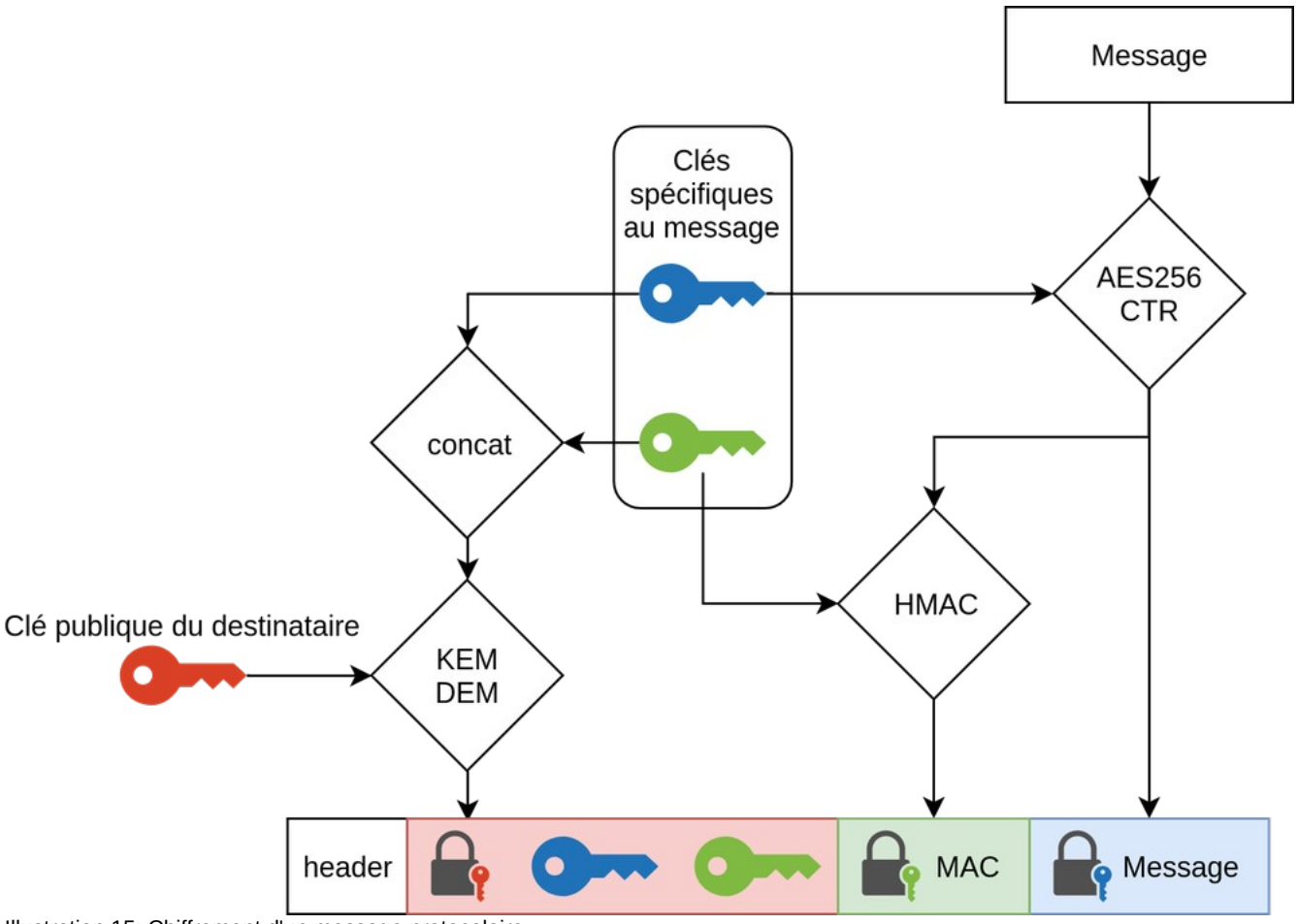

Illustration 15: Chiffrement d'un message protocolaire

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]**.

Implémentation du chiffrement des messages

Les messages échangés sont chiffrés grâce au chiffrement symétrique présenté dans Chiffrement des messages page 17.

Le chiffrement est réalisé lors de l'envoi d'un message via un *ObvNetworkChannel*.

```
extension ObvNetworkChannel {
    [...]
    private static func generateMessageKeyAndHeaders(using acceptableChannels:
[ObvNetworkChannel], randomizedWith prng: PRNGService) -> (AuthenticatedEncryptionKey,
[ObvNetworkMessageToSend.Header])? {
        let cryptoSuiteVersion =
acceptableChannels.reduce(ObvCryptoSuite.sharedInstance.latestVersion) { min($0,
$1.cryptoSuiteVersion) }
        guard let authEnc =
ObvCryptoSuite.sharedInstance.authenticatedEncryption(forSuiteVersion: cryptoSuiteVersion)
else {
            return nil
        }
        let messageKey = authEnc.generateKey(with: prng)
        let headers = acceptableChannels.map { $0.wrapMessageKey(messageKey,
randomizedWith: prng) }
        return (messageKey, headers)
    }
    [...]
    static func post(_ message: ObvChannelMessageToSend, randomizedWith prng: PRNGService,
delegateManager: ObvChannelDelegateManager, within obvContext: ObvContext) throws ->
Set<ObvCryptoIdentity> {
        [...]
        guard let (messageKey, headers) = generateMessageKeyAndHeaders(using:
acceptableChannels, randomizedWith: prng) else { throw NSError() }
        guard let networkMessage = generateObvNetworkMessageToSend(from: message,
messageKey: messageKey, headers: headers, randomizedWith: prng) else { throw NSError() }
        try networkPostDelegate.post(networkMessage, within: obvContext)
        [...]
    }
}
```

Une clé symétrique est alors générée pour chiffrer le message. Le message est ensuite chiffré lors de l'appel à *generateObvNetworkMessageToSend.*

```
private static func generateObvNetworkMessageToSend(from message: ObvChannelMessageToSend,
messageKey: AuthenticatedEncryptionKey, headers: [ObvNetworkMessageToSend.Header],
randomizedWith prng: PRNGService) -> ObvNetworkMessageToSend? {
    let wrapperMessage: ObvChannelMessageToSendWrapper?
    switch message.messageType {
    case .ProtocolMessage:
        wrapperMessage = ObvChannelProtocolMessageToSendWrapper(message: message,
messageKey: messageKey, headers: headers, randomizedWith: prng)
    case .ApplicationMessage:
        wrapperMessage = ObvChannelApplicationMessageToSendWrapper(message: message,
messageKey: messageKey, headers: headers, randomizedWith: prng)
    case .DialogMessage,
         .DialogResponseMessage,
         .ServerQuery,
         .ServerResponse:
        // Dialog/Server Queries messages are not intended to be sent over the network as
protocol or application messages
        wrapperMessage = nil
    }
    guard let msg = wrapperMessage else { return nil }
    return try? msg.generateObvNetworkMessageToSend()
}
```

Le wrapper utilisé pour chiffrer les messages protocolaires fait usage de *CTR_AES_256_THEN_HMAC_SHA_256* comme le confirme l'analyse dynamique.

```
fileprivate extension ObvChannelMessageToSendWrapper {
    [...]
    static func encryptContent(messageKey: AuthenticatedEncryptionKey, type:
ObvChannelMessageType, encodedElements: ObvEncoded, randomizedWith prng: PRNGService) ->
EncryptedData {
        let authEnc = messageKey.algorithmImplementationByteId.algorithmImplementation
        let content = generateContent(type: type, encodedElements: encodedElements)
        return try! authEnc.encrypt(content.rawData, with: messageKey, and: prng)
    }
}
struct ObvChannelProtocolMessageToSendWrapper: ObvChannelMessageToSendWrapper {
    [...]
    func generateObvNetworkMessageToSend() throws -> ObvNetworkMessageToSend {
        let encryptedContent =
ObvChannelProtocolMessageToSendWrapper.encryptContent(messageKey: messageKey,
                                                                                type:
messageType,

 encodedElements: encodedElements,

 randomizedWith: prng)
        [...]
    }
}
```

L'utilisation de *CTR_AES_256_THEN_HMAC_SHA_256* correspond à un appel à *AuthenticatedEncryptionWithAES256CTRThenHMACWithSHA256* dont l'implémentation est conforme à **[DOC_CRYPTO]**.

```
final class AuthenticatedEncryptionWithAES256CTRThenHMACWithSHA256 :
AuthenticatedEncryptionConcrete {
    static var algorithmImplementationByteId: AuthenticatedEncryptionImplementationByteId {
        return .CTR_AES_256_THEN_HMAC_SHA_256
    }
    [...]
    static func encrypt(_ plaintext: Data, with _key: AuthenticatedEncryptionKey, and
_prng: PRNG?) throws -> EncryptedData {
        guard let key = _key as? AuthenticatedEncryptionWithAES256CTRThenHMACWithSHA256Key
else { throw AuthenticatedEncryptionError.incorrectKey }
        let prng = _prng ?? ObvCryptoSuite.sharedInstance.prngService()
        // Encrypt...
        let iv = prng.genBytes(count: SymmetricEncryptionWithAES256CTR.ivLength)
        let ciphertext = try! SymmetricEncryptionWithAES256CTR.encrypt(plaintext, with:
key.aes256CTRKey, andIv: iv)
        // ... then authenticate
        let mac = try! HMACWithSHA256.compute(forData: ciphertext, withKey:
key.hmacWithSHA256Key)
        let authenticatedCiphertext = EncryptedData.byAppending(c1: ciphertext, c2:
EncryptedData(data: mac))
        return authenticatedCiphertext
    }
    [...]
}
```

SymmetricEncryptionWithAES256CTR.encrypt repose ensuite sur le *framework CommonCrypto* d'*Apple*.

```
static func encrypt(_ plaintext: Data, with key: SymmetricEncryptionKey, andIv iv: Data)
throws -> EncryptedData {
```

```
  // ...
  CCCryptorCreateWithMode(CCOperation(kCCEncrypt), CCMode(kCCModeCTR),
CCAlgorithm(kCCAlgorithmAES), CCPadding(ccNoPadding), ivPtr, keyPtr, keyLength, nil, 0, 0,
CCModeOptions(kCCModeOptionCTR_BE), &cryptoRef)
                        var dataOutMoved = 0
                        CCCryptorUpdate(cryptoRef, plaintextPtr, plaintext.count,
ciphertextPtr.advanced(by: ivLength), ciphertextLength-ivLength, &dataOutMoved)
                        let status = CCCryptorFinal(cryptoRef, ciphertextPtr.advanced(by:
ivLength + dataOutMoved), ciphertextLength-ivLength-dataOutMoved, &dataOutMoved)

  // ..
}
```

On constate qu'un *nonce* de 8 octets est bien généré à partir du *prngService* et que ce nonce est passé en argument de la fonction CCCryptorCreateWithMode.

Forward Secrecy

Pour assurer une Forward Secrecy si une clé venait à être compromise, les clés utilisées sont rafraîchies régulièrement par le mécanisme de cliquet.

Ce mécanisme est similaire à ce qui est déjà fait lors de la procédure d'échange de clés.

[DOC_CRYPTO] spécifie que les clés ont une durée de vie de 100 messages ou une semaine.

Cette propriété peut être observée dans le code de l'application:

```
private var requiresFullRatchet: Bool {
    // [...]
    if aFullRatchetOfTheSendSeedIsInProgress {
        // [...]
    } else {
        // 1. If the number of encrypted messages since the last successfull full ratchet
is too high, we must start a new full ratchet
        guard numberOfEncryptedMessagesSinceLastFullRatchet <
ObvConstants.thresholdNumberOfEncryptedMessagesPerFullRatchet else {
            os_log("Full ratchet required for the send seed: %d >= %d", log: log,
type: .info, numberOfEncryptedMessagesSinceLastFullRatchet,
ObvConstants.thresholdNumberOfEncryptedMessagesPerFullRatchet)
            return true
        }
        os_log("[1/2] No need to perform a full ratchet of the send seed: %d < %d", log:
log, type: .info, numberOfEncryptedMessagesSinceLastFullRatchet,
ObvConstants.thresholdNumberOfEncryptedMessagesPerFullRatchet)

        // 2. If the elapsed time since the last successfull full ratchet is too high, we
must start a new full ratchet
        guard Date().(timestampOfLastFullRatchet) <
ObvConstants.fullRatchetTimeIntervalValidity else {
            os_log("Full ratchet required because of too much time passed since the last
full ratchet", log: log, type: .info)
            return true
        }
    }
```

On constate donc que les conditions de déclenchement sont conformes à la spécification.

Le cas où un utilisateur changerait l'heure et la date de son mobile après un rafraîchissement pourrait retarder la génération d'une nouvelle clé. Mais ceci est jugé hors de propos à la vue des autres garanties.

Quand *requiresFullRatchet* est vraie, la fonction *startFullRatchetProtocolForObliviousChannelBetween* est appelée. Celle ci démarre un échange protocolaire via le premier message du protocole *FullRatchetProtocol*.

```
public func startFullRatchetProtocolForObliviousChannelBetween(currentDeviceUid: UID,
andRemoteDeviceUid remoteDeviceUid: UID, ofRemoteIdentity remoteIdentity:
ObvCryptoIdentity) throws {
    // [...]
    let coreMessage = CoreProtocolMessage(channelType: .Local(ownedIdentity:
ownedIdentity),
            cryptoProtocolId: .FullRatchet,
            protocolInstanceUid: protocolInstanceUid)
    let initialMessage = FullRatchetProtocol.InitialMessage(coreProtocolMessage:
coreMessage,
            contactIdentity: remoteIdentity,
            contactDeviceUid: remoteDeviceUid)
    _ = try channelDelegate.post(initialMessageToSend, randomizedWith: prng, within:
obvContext)
    // [...]
}
```

À ce stade, le reste de la logique est implémentée par la machine à état. Les transitions possibles sont les suivantes :

- ConcreteProtocolInitialState -> AliceSendEphemeralKeyStep -> AliceWaitingForK1State
- ConcreteProtocolInitialState -> BobSendEphemeralKeyAndK1FromInitialStateStep -> BobWaitingForK2State
- AliceWaitingForK1State -> AliceRecoverK1AndSendK2Step -> AliceWaitingForAckState
- AliceWaitingForK1State -> AliceResendEphemeralKeyFromAliceWaitingForK1StateStep -> AliceWaitingForK1State
- AliceWaitingForAckState -> AliceResendEphemeralKeyFromAliceWaitingForAckStateStep -> AliceWaitingForK1State
- AliceWaitingForAckState -> AliceUpdateSendSeedStep -> FullRatchetDoneState
- BobWaitingForK2State -> BobSendEphemeralKeyAndK1BobWaitingForK2StateStep -> BobWaitingForK2State
- BobWaitingForK2State -> BobRecoverK2ToUpdateReceiveSeedAndSendAckStep -> BobWaitingForK2State, FullRatchetDoneState

Ce graphe correspond à la spécification de **[DOC_CRYPTO]**. On constate que l'implémentation des steps correspond à ceux détaillés dans FS2 : Authentification des échanges page 40, à la différence que les correspondants disposent déjà d'un canal authentifié. En particulier les mêmes primitives sont utilisées pour la génération et l'échange des clés éphémères.

Destruction des clés de provisions

Les clés provisionnées par le mécanisme de cliquet simple sont successivement utilisées pour chiffrer puis déchiffrer les messages. Quand une clé a été utilisée par un destinataire celle-ci est détruite pour ne pas rester en mémoire.

```
static func unwrapMessageKey(wrappedKey: EncryptedData, toOwnedIdentity: ObvCryptoIdentity,
delegateManager: ObvChannelDelegateManager, within obvContext: ObvContext) throws ->
(AuthenticatedEncryptionKey, ObvProtocolReceptionChannelInfo)? {
    // [...]
    guard let (encryptedMessageKey, keyId) = ObvObliviousChannel.parse(wrappedKey) else
{ return nil }
        let provisionedKeys = try KeyMaterial.getAll(cryptoKeyId: keyId, currentDeviceUid:
deviceUid, within: obvContext)
```

```
        // Given the keyId of the received message, we might have several candidate for the
decryption key (i.e., several provisioned received keys). We try them one by one until one
successfully decrypts the message

        os_log("Number of potential provisioned keys for this key id: %d", log: log,
type: .debug, provisionedKeys.count)

        for provisionedKey in provisionedKeys {

            let provision = provisionedKey.provision
            let obliviousChannel = provision.obliviousChannel
            let authEnc =
provisionedKey.key.algorithmImplementationByteId.algorithmImplementation

            if let rawEncodedMessageKey = try? authEnc.decrypt(encryptedMessageKey, with:
provisionedKey.key) {

                guard let encodedMessageKey = ObvEncoded(withRawData: rawEncodedMessageKey)
else { return nil }
                guard let messageKey = try?
AuthenticatedEncryptionKeyDecoder.decode(encodedMessageKey) else { return nil }

                os_log("Received a message on ratchet generation %d - %d", log: log,
type: .info, provision.fullRatchetingCount, provisionedKey.selfRatchetingCount)

                // We set the expiration timestamp of older keys. We self-ratchet (if
required) all the provisions that contained one or more keys for which we set the
expiration timestamp.
                let provisionsWithLessKeys = try
provisionedKey.setExpirationTimestampOfOlderButNotYetExpiringProvisionedReceiveKeys()
                for provision in provisionsWithLessKeys {
                    try provision.selfRatchetIfRequired()
                }

                // If a full ratcheting is currently in place for refreshing the send key
and send key id, we increment the number of decrypted messages since the last full ratchet
sent message counter
                if obliviousChannel.aFullRatchetOfTheSendSeedIsInProgress {

obliviousChannel.numberOfDecryptedMessagesSinceLastFullRatchetSentMessage += 1
                }
                // We self-ratchet the provision which is about to "lose" a key
                try provisionedKey.provision.selfRatchetIfRequired()
                // The provisioned key we just used to decrypt the message will never be
used again, so we delete it
                os_log("Since we used it to decrypt, we delete the provisioned key with
selft ratcheting count %d", log: log, type: .debug, provisionedKey.selfRatchetingCount)
                obvContext.delete(provisionedKey)

                // If successfully decrypted, so we can mark the channel as 'confirmed'
                obliviousChannel.confirm()

                return (messageKey, obliviousChannel.type)

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

Chiffrement des pièces jointes

Lors de l'envoi d'un message applicatif avec des pièces jointes, ces dernières sont envoyées séparément.

La création d'un message applicatif pour envoi est réalisée lors de l'instanciation de *ObvChannelApplicationMessageToSendWrapper*. Pour chaque pièce jointe, une clé symétrique est générée du même type que le chiffrement du corps du message.

```
struct ObvChannelApplicationMessageToSendWrapper: ObvChannelMessageToSendWrapper {
    [...]
    init?(message: ObvChannelMessageToSend, messageKey: AuthenticatedEncryptionKey,
headers: [ObvNetworkMessageToSend.Header], randomizedWith prng: PRNGService) {
        [...]
        let authEnc = messageKey.algorithmImplementationByteId.algorithmImplementation
        let attachmentsAndKeys = applicationMessage.attachments.map { ($0,
authEnc.generateKey(with: prng)) }
        self.encodedElements =
ObvChannelApplicationMessageToSendWrapper.generateEncodedElements(fromMessagePayload:
self.applicationMessage.messagePayload, and: attachmentsAndKeys)
        self.attachments =
ObvChannelApplicationMessageToSendWrapper.generateObvNetworkMessageToSendAttachments(from:
attachmentsAndKeys)
    }
```

Le corps du message contient les *metadata* et clés de chaque pièce jointe.

```
fileprivate extension ObvChannelApplicationMessageToSend.Attachment {
    func generateEncodedElement(including key: AuthenticatedEncryptionKey) -> ObvEncoded {
        return [key, metadata].encode()
    }
    [...]
}
```

Pour envoyer les pièces jointes l'application doit attendre d'avoir envoyé le message et reçu en échange une liste d'URL utilisables pour l'envoi des pièces jointes. Le code de chiffrement des pièces jointes est implémenté dans une tache de fond. Cette tache est activée après réception des URLs.

```
final class UploadAttachmentChunksOperation: ObvOperation {
    [...]
    /// The purpose of this operation is to upload all the unaknowledged chunks of the
attachment passed in the initializer.
    /// We create one background URLSession, valid for all the chunks upload tasks.
    override func execute() {
        [...]
        contextCreator.performBackgroundTaskAndWait(flowId: flowId) { (obvContext) in
            [...]
            for chunkRangeToUpload in chunkRangesToUpload {
                for chunkNumberToUpload in chunkRangeToUpload {
                    [...]
                    let encryptedChunk: EncryptedData
                    do {
                        encryptedChunk = try getEncryptedChunk(number: chunkNumberToUpload,
of: attachment, logTo: log)
                    } catch let error {
                        os_log("Could not open FileHandle for reading from URL %@
corresponding to attachment %{public}@: %@", log: log, type: .fault,
attachment.fileURL.debugDescription, attachment.attachmentId.debugDescription,
error.localizedDescription)
                        return cancelAndFinishWithReason(.couldNotReadAttachment)
                    }
```

```
                    let signedURL = chunkUploadPrivateUrls[chunkNumberToUpload]
                    let method: ObvS3UploadAttachmentChunkMethod
                    do {
                        method = try ObvS3UploadAttachmentChunkMethod(attachmentId:
attachmentId,
                                                                      encryptedChunk:
encryptedChunk,
                                                                      chunkNumber:
chunkNumberToUpload,
                                                                      signedURL: signedURL,
                                                                      flowId: flowId)
                    } catch let error {
                        [...]
                    }
                    [...]
                }
            }
            [...]
        }
    }
}
```

Le chiffrement des *chunks* de la pièce jointe est réalisé par *getEncryptedChunk*.

```
private func getEncryptedChunk(number chunkNumber: Int, of attachment: OutboxAttachment,
logTo log: OSLog) throws -> EncryptedData  {
    // Fast forward the plaintext attachment to the first byte of the first chunk that has
not been sent already
    let fh = try FileHandle(forReadingFrom: attachment.fileURL)
    fh.seek(toFileOffset: UInt64(chunkNumber) * UInt64(attachment.cleartextChunkLength))
    let chunkToSend = Chunk(index: chunkNumber, data: fh.readData(ofLength:
attachment.cleartextChunkLength))
    fh.closeFile()
    // Encrypt the chunk
    let prngService = ObvCryptoSuite.sharedInstance.prngService()
    let authEnc = attachment.key.algorithmImplementationByteId.algorithmImplementation
    let encodedChunk = chunkToSend.encode()
    let encryptedChunk = try! authEnc.encrypt(encodedChunk.rawData, with: attachment.key,
and: prngService) // cannot throw in this case
    return encryptedChunk
}
```

L'analyse du code source réalisant le chiffrement des messages et des pièces jointes ne met en exergue aucun problème d'implémentation.

Analyse dynamique du chiffrement des messages protocolaires

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

```
header: [ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff,
421c03328ea6b8caee6b42a034c694eff519fd38ecc2ee23e7047a9ed1e9c8da12f8dcb93d531e840ace52ff2c7
cc87d4c6efa0529ce52b3212b22964a70484d2ffd331192e4a677b0e0b32c8733493ca3c1aa2b68f20b6aa8f52d
94a167162c2fb8cb75816cf625ffce02da2c199a4f3152d73a9531b4f988f5f2f4fba3b8cfbbc172d6d933b1c5c
3b46a1e9a2a96892f9a2531f1e1c3af7283df0e13861fdc857906942593894d01a8b6887516215f618a696138a0
85a6ad,
68747470733a2f2f7365727665722e6f6c7669642e696f000023569981ae071758ec2ef28eae66ae9df5223e2fd
a5919fb911ff26daa9026fa010b3b4561f33b73c67c56a5e9e8a6073897c9a675f071dc2073757472bcb13a95]
Message:
862bc7dff04f529fd384f093ad76fad9517e630b35b785f76242c51d2869a4a0ba5fdb1c98bb38ae52e5820c07c
6cb44bc9e838dd36beff5061ea7298679d37380d160c8670a3ad9682a31a12f6b9df528b90cda71e19a3637bf93
```

```
186ec45a18a563cba2981ecb8d8095c041bfc6e3cd76e91a36be03b8f4d00bdd170b761cb1f2e4ca481e52081b0
dc0f82bab1e3d9c91b94dc2efaa9fde5e652b15fd5c0765a3b04faf0fea6dc7718f87eb25cc87e122773951b58a
648fdd6e3347fd9fca5b82626da75d26ce3b187bc58af27129b3b2d5c3a0a06af3aadd0d4fc9382e5b47343bb12
26dae02e44b02389bf1c0255b9d903bc4f2c64bdef654c91c097c0e5591739ca885a374e155c67e0c64b3c64c71
14aa0e6dfee57feaf7f3667a4ee6f3d04232695b07c158cf843d2c7620ece00494c296a08fadf51ad20cfc67602
4afd1500dfedfacbc37f1fb22fd
```

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

À titre d'information l'entropie du corps du message est de 7.43 bit/octet. L'entropie est calculée comme dans la section Analyse dynamique p.56, 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 donnée *SQLite* de l'application et en ré-implémentant le déchiffrement. Le code réalisant cette opération est disponible en annexe.

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

```
Symetric key: <SymmetricKey algoClassByteId=2 algoImplemByteId=0 dict={'enckey':
1427df9dd1dac89eed6aeb03791b54dfcd547ad803cff5e90d0ef0daaadcf9c6, 'mackey':
c167db6fda919978142ac770f60efe33e5ee10716f7754bc774a76bcd046fb99}>
```

Il est ensuite possible de déchiffrer le corps du message.

```
Protocol Message
  Trust Establishment with SAS
    Protocol UID: 1f47df688d146a241bdd27547ba5cf4833c800985a94ade9b8a15e8a809adfaa
    Alice sends commitment
      Contact identity:
68747470733a2f2f7365727665722e6f6c7669642e696f00002c56c7c4b256d707b3d258c9266942b9eec363651
2f8546bfc7773c5e65ddef50107b054f42552f293df32561630cf960311b268cd925881fbeee073e1ec06a0e2
      Encoded Contact identity: 7b2266697273745f6e616d65223a22416c696365227d
      Contact device UIDs:
 [f4ee3ae632595d6833b024ae51805fb7403d686560b26246a92a2c8391c37a0d]
      Commitment:
18453f7d65aa8fa83baf92dc0c5dbb5fbf932dfffd28c65a8d4368cd4b9cc2a4
```

Analyse dynamique du chiffrement des messages applicatifs

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

```
header: [f4ee3ae632595d6833b024ae51805fb7403d686560b26246a92a2c8391c37a0d,
36742df2ffa1b66f1b9595f4901a29a9eaebfc112494c0dd191231ea261a982f5a4604dda7e066aacd6fdaba810
744048c7352957b6cc8be8726a0ef4fe47fa24b24ec6a269a064f53eead740eff2c31c3c912622e5e2ea683e8cb
67b36b5d8edd7ef1df22646fa2dbedaa496fd3eba5f74ee23cc1a49089b8efcd127c517028a87f2a9175cf704bb
08dd8290824effbeb262edcde98c5bcced9318e487ffe1ee769dfa69082638923de9b28a7cde66f866b4c10843b
04a3fa,
68747470733a2f2f7365727665722e6f6c7669642e696f00002c56c7c4b256d707b3d258c9266942b9eec363651
2f8546bfc7773c5e65ddef50107b054f42552f293df32561630cf960311b268cd925881fbeee073e1ec06a0e2]
Message:
bb6a0328e5a98e611ef57250f5fe9aa2ca1fe89aacaeda7422da522ca8d36c7d191beeafa032a4edfd2f6de42d2
e3e998cc6e4f7f23f5e5ed75f179af895bf7abf701a89739ce02927132764ab184afb913b2e99f6964d72d47bea
d08e9b6e60ea202fa4578d33f35c53ef8d44967337f4f669db70ba792018b1cfc31d893870ce15e8786d4d91303
533a2ff73472d6c9183b9a05a895c48e6ef314d20901b42f810526589c9180f505ae70a243547a1b53666655169
4531fb0beadf1f96021c78d0f0841b1c182629bd7c6092a22f4167bd6f60c44cb17b8cf9f3c231fe5cc797b9a4c
6734656047eb1bfc49e32c4666430463d52e2e3c127bae13a24b04b4084322d73a5a1720469d39aa2fd028941dc
4082fd913e554227aa99178ec4962da1582a214818be21bbe61dad8adef71787793e0686b40d0439b12bde4e698
e6d78e5794f14dd316cc7256702355899d457ca8857817dfa489a9c24ca0fb1e978f1041148f1fba4c6b8a0d8
```

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

À titre d'information l'entropie du corps du message est de 7.40 bit/octet. L'entropie est calculée comme dans la section Analyse dynamique p.56, 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 donnée *SQLite* de l'application et en ré-implémentant le déchiffrement. Le code réalisant cette opération est disponible en annexe.

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

```
Symetric key: <SymmetricKey algoClassByteId=2 algoImplemByteId=0 dict={'enckey':
cfcecc685fcdafc7e83d3a1f0e36ea8e442caac2813b1a67772830186e34ef6f, 'mackey':
7395fe15a564febab2d0891596abdda3041cdbdbfe334fb13704354b6372373a}>
```

Il est ensuite possible de déchiffrer le corps du message.

```
{
  "message": {
    "ssn": 2,
    "sti": "69C66972-3070-4861-893F-97D3A8F669D3",
    "body": "Bonjour Alice"
  },
  "rr": {
    "nonce": "VKfFaAvZMMB6DcYo3RzNRQ==",
    "key":
"kAAAAGwAAAAAgIABAAAAGAAAAAABmVuY2tleQAAAAAgm/hN7wCQSiUnZ+s/GvcC9+opFIVQ620uuxZ8wQ0FCE0AAA
AABm1hY2tleQAAAAAgsT4I33Jpzlp65tgnzRtEqMv5+VvP02oBq/8sEjdd4qc="
  }
}
```

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 14, il était nécessaire de vérifier que les implémentations sont correctes.

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

- L'utilisation de primitives cryptographiques testées et éprouvées;
- Une ré-implémentation 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 conséquences, 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 17 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 14 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 ré-implémentation du déchiffrement *hors-ligne* d'un fichier de sauvegarde.

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

Génération, encodage et décodage de la passphrase en graine

Lors de l'initialisation d'un fichier de sauvegarde, une graine est créée et constitue le secret partagé à l'utilisateur.

La génération est effectuée dans *ObvBackupManagerImplementation.swift* à l'aide de l'instance du PRNG et de la fonction *genBackupSeed*.

```
    public func genBackupSeed() -> BackupSeed {
        let rawSeed = genBytes(count: BackupSeed.byteLength)
        return BackupSeed(with: rawSeed)!
    }
```

Cette fonction renvoie alors un nombre de 20 octets. Le PRNG étant déjà validé dans Analyse de la spécification cryptographique page 14, on a donc bien 160 bits d'entropie.

La *passphrase* présentée à l'utilisateur pour retrouver son fichier de sauvegarde est cependant une chaîne de caractères présentée en 8 fois 4 lettres.

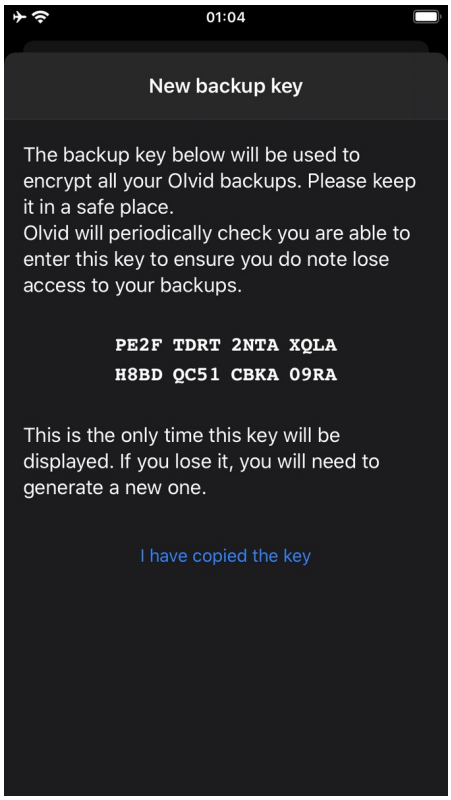

Illustration 16: Une passphrase de déchiffrement de fichier de sauvegarde

Cette *passphrase* est en réalité la graine précédente encodée par 32 caractères représentant 5 bits d'entropie chacun. L'alphabet utilisé contient donc 32 caractères allant de '0' à '9' puis de 'A' à 'Y' en excluant 'I', 'O' et 'S' qui pourraient mal être interprétés.

La passphrase du fichier de sauvegarde ne présente donc pas de biais dans le sens où toutes les lettres de cet alphabet sont équiprobables. Les fonctions d'encodage et décodage sont implémentées dans *BackupSeed.swift* et ont été ré-implémentées dans *BackupSeed.py*.

Des vecteurs de tests sont joués sur chacune des implémentations. On constate que les résultats sont similaires et bien ceux attendus.

Un attaquant qui aurait récupéré un fichier de sauvegarde sans la passphrase, n'aurait donc pas de meilleure attaque que le *bruteforce* sur 160 bits pour retrouver le secret (graine). L'implémentation de Olvid est donc conforme.

Dérivation du secret partagé en clés

La graine générée sert à dériver plusieurs clés, ceci est effectué dans la fonction *deriveKeysForBackup* de *BackupSeed.swift*.

```
    public func deriveKeysForBackup() -> DerivedKeysForBackup {
        // We padd the backup seed with 0x00's in order to generate a seed
        let rawSeed = self.raw + Data(repeating: 0, count: max(0, Seed.minLength -
self.raw.count))
        let seed = Seed(with: rawSeed)!
        let prng = PRNGWithHMACWithSHA256(with: seed)
        return DerivedKeysForBackup.gen(with: prng)
    }
```

Dans un premier temps on constate que 12 octets nuls sont ajoutés à la graine qui sert à initialiser un nouveau PRNG. Ensuite, la fonction *DerivedKeysForBackup* est utilisée avec le PRNG instancié pour générer les éléments suivants:

- Un UID sur 32 octets ;
- une clé privée et clé publique pour l'ECIES sur la courbe 25519 ;
- une clé pour le HMAC avec SHA256.

```
    static func gen(with prng: PRNG) -> Self {
        let backupKeyUid = UID.gen(with: prng)
        let (publicKeyForEncryption, privateKeyForEncryption) =
ECIESwithCurve25519andDEMwithCTRAES256thenHMACSHA256.generateKeyPairForBackupKey(with:
prng)
        let macKey = HMACWithSHA256.generateKeyForBackup(with: prng)
        return DerivedKeysForBackup(backupKeyUid: backupKeyUid, publicKeyForEncryption:
publicKeyForEncryption, privateKeyForEncryption: privateKeyForEncryption, macKey: macKey)
    }
```

Cette génération de clés est ré-implémentée dans *DerivedKeysForBackup.py*.

Des vecteurs de tests sont joués sur chacune des implémentations afin de vérifier qu'une *passphrase* donne bien les mêmes clés. On constate que les résultats sont similaires et bien ceux attendus.

Les propriétés du PRNG utilisé garantissent qu'un attaquant qui ne dispose pas de la graine n'aurait pas de meilleurs attaque que le *bruteforce* de cette dernière pour retrouver les mêmes clés de chiffrement. L'implémentation de Olvid est donc conforme.

Vérification du HMAC

La clé HMAC dérivée à l'étape précédente sert à vérifier l'intégrité de l'archive. Cette vérification est présente dans *ObvBackupManagerImplementation.swift*, avant de déchiffrer la sauvegarde.

```
let receivedMac = backupData[backupData.endIndex-macLength..<backupData.endIndex]
let encryptedBackup = backupData[backupData.startIndex..<backupData.endIndex-macLength]
let computedMac = try MAC.compute(forData: encryptedBackup, withKey:
derivedKeysForBackup.macKey)
guard computedMac == receivedMac else {
   os_log("The mac comparison failed during the recover of the backup within flow %
{public}@", log: log, type: .error)
   return
}
```

Une tentative d'importation d'un fichier de sauvegarde modifié avec la *passphrase* d'origine n'aboutit pas.

Les propriétés du HMAC assurent qu'un attaquant n'ayant pas connaissance de la clé utilisée ne pourra pas compromettre ou forger un fichier de sauvegarde qui resterait valide pour une *passphrase* donnée.

Déchiffrement de la sauvegarde

Le déchiffrement de la sauvegarde se fait en deux étapes.

1. Le déchiffrement de deux clés de 32 bytes chacune à partir des premiers octets du chiffré via le *KEM* (asymétrique) basé sur la courbe 25519.
2. Le déchiffrement symétrique (AES256) du reste du chiffré à partir de ces clés.

```
    static func decrypt(_ ciphertext: EncryptedData, using privateKey:
PrivateKeyForPublicKeyEncryption) -> Data? {
        guard ciphertext.count >= KEM.length else { return nil }
        let c0 = ciphertext[ciphertext.startIndex..<ciphertext.startIndex+KEM.length]
        let c1 = ciphertext[ciphertext.startIndex+KEM.length..<ciphertext.endIndex]
        guard let key = KEM.decrypt(c0, using: privateKey,
{ AuthenticatedEncryptionWithAES256CTRThenHMACWithSHA256Key(data: $0)! }) else { return nil
}
        let plaintext = try?
AuthenticatedEncryptionWithAES256CTRThenHMACWithSHA256.decrypt(c1, with: key)
        return plaintext
    }
```

Déchiffrement des clés de sessions via le KEM

La routine de déchiffrement via le KEM est donnée dans le prochain extrait. Cette implémentation est propre à Olvid et se base sur Curve25519 implémentée dans l'application.

```
    static func decrypt<T: SymmetricKey>(_ ciphertext: EncryptedData, using _privateKey:
PrivateKeyForPublicKeyEncryption, _ convertBytesToKey: (Data) -> T) -> T? {
        guard let privateKey = _privateKey as?
PrivateKeyForPublicKeyEncryptionOnEdwardsCurve else { return nil }
        guard ciphertext.count == privateKey.curve.parameters.p.byteSize() else { return
nil }
        let nu = curve.parameters.nu
        let q = curve.parameters.q
        let ciphertextAsData = Data(encryptedData: ciphertext)
        let yCoordinate = BigInt(ciphertextAsData)
        guard yCoordinate != BigInt(1) else { return nil }
        guard let By = curve.scalarMultiplication(scalar: nu, yCoordinate: yCoordinate)
else { return nil }
        let a = BigInt(privateKey.scalar).mul(try! BigInt(nu).invert(modulo: q), modulo: q)
        let Dy = curve.scalarMultiplication(scalar: a, yCoordinate: By)!
        var rawSeed = ciphertextAsData
        let pLength = curve.parameters.p.byteSize()
        rawSeed.append(try! Data(Dy, count: pLength))
        guard let seed = Seed(with: rawSeed) else { return nil }
        guard let key = KDFFromPRNGWithHMACWithSHA256.generate(from: seed,
convertBytesToKey) else { return nil }
        return key
    }
```

L'analyse de cette fonction montre qu'elle est conforme au schéma ECIES.

L'implémentation de Olvid fait l'objet de tests unitaires à l'aide de vecteurs. Une ré-implémentation complète par le CESTI est disponible dans *KEM.py* et *EdwardCurve.py*. Elle donne des résultats identiques à celle de Olvid. L'implémentation de Olvid est donc conforme.

Déchiffrement des données

La première clé sert pour effectuer un HMAC SHA256 pour vérifier à nouveau le reste du chiffré. La seconde sert de clé de session pour du chiffrement symétrique AES256 en mode CTR pour déchiffrer le reste du fichier.

```
    static func decrypt(_ ciphertext: EncryptedData, with _key: AuthenticatedEncryptionKey)
throws -> Data {
        guard let key = _key as? AuthenticatedEncryptionWithAES256CTRThenHMACWithSHA256Key
else { throw AuthenticatedEncryptionError.incorrectKey }
```

```
        guard ciphertext.count >=
AuthenticatedEncryptionWithAES256CTRThenHMACWithSHA256.minimumCiphertextLength else { throw
AuthenticatedEncryptionError.ciphertextIsNotLongEnough }
        let ciphertextRange = ciphertext.startIndex..<ciphertext.endIndex -
HMACWithSHA256.outputLength
        let macRange = ciphertext.endIndex -
HMACWithSHA256.outputLength..<ciphertext.endIndex
        let innerCiphertext = ciphertext[ciphertextRange]
        let mac = Data(encryptedData: ciphertext[macRange])
        guard try! HMACWithSHA256.verify(mac: mac, forData: innerCiphertext, withKey:
key.hmacWithSHA256Key) else { throw AuthenticatedEncryptionError.integrityCheckFailed }
        let plaintext = try! SymmetricEncryptionWithAES256CTR.decrypt(innerCiphertext,
with: key.aes256CTRKey)
        return plaintext
    }
```

L'extrait de code suivant montre l'implémentation (simplifiée) de *decrypt* de *SymmetricEncryptionWithAES256CTR* :

```
static func decrypt(_ ciphertext: EncryptedData, with key: SymmetricEncryptionKey) throws -
> Data {

    // [...]
    let status = ciphertext.withUnsafeBytes { (ciphertextBufferPtr) -> Int in
        let ciphertextPtr = ciphertextBufferPtr.baseAddress!
        let status = plaintext.withUnsafeMutableBytes { (plaintextBufferPtr) -> Int in
            let plaintextPtr = plaintextBufferPtr.baseAddress!
            let status = key.data.withUnsafeBytes { (keyBufferPtr) -> Int in
                let keyPtr = keyBufferPtr.baseAddress!
                let status = ccIv.withUnsafeBytes { (ivBufferPtr) -> Int in
                    let ivPtr = ivBufferPtr.baseAddress!
                    var cryptoRef: CCCryptorRef?
                    CCCryptorCreateWithMode(CCOperation(kCCDecrypt), CCMode(kCCModeCTR),
CCAlgorithm(kCCAlgorithmAES), CCPadding(ccNoPadding), ivPtr, keyPtr, keyLength, nil, 0, 0,
CCModeOptions(kCCModeOptionCTR_BE), &cryptoRef)
                    var dataOutMoved = 0
                    CCCryptorUpdate(cryptoRef, ciphertextPtr.advanced(by: ivLength),
plaintextLength, plaintextPtr, plaintextLength, &dataOutMoved)
                    let status = CCCryptorFinal(cryptoRef, plaintextPtr.advanced(by:
dataOutMoved), plaintextLength-dataOutMoved, &dataOutMoved)
                    return Int(status)
                }
                return status
            }
            return status
        }
        return status
    }
    // [...]
    return plaintext
}
```

Olvid n'implémente donc pas directement l'algorithme AES mais utilise la bibliothèque d'Apple *CommonCrypto* (*https://developer.apple.com/library/archive/documentation/System/Conceptual/ManPages_iPhoneOS/man3/Common%20Crypto.3cc.html*).

Une ré-implémentation du déchiffrement symétrique est présenté dans *decrypt_backup.py*. On constate que l'implémentation d'Olvid et celle du CESTI, bien que basées sur des bibliothèques pour hmac et AES différentes, produisent des déchiffrés identiques. L'implémentation de Olvid est donc conforme.

Un attaquant n'ayant pas connaissance de la clé privée dérivée de la *passphrase* n'aura donc pas de meilleurs options que de trouver une faiblesse dans AES en mode CTR pour déchiffrer le contenu.

Décompression des données

La dernière étape lors de l'importation d'un fichier de sauvegarde est la décompression des données.

Celle-ci est effectuée à l'aide de la fonction d'Apple *compression_decode_buffer* (*https://developer.apple.com/documentation/compression/1481000-compression_decode_buffer*) utilisée avec l'algorithme ZLIB.

Sans faiblesse dans AES en mode CTR, cette étape de transformation d'un clair ne change pas le niveau de sécurité du chiffrement du carnet de contact.

Déchiffrement hors-ligne d'un fichier de sauvegarde

En enchaînant les étapes précédentes, la ré-implémentation permet de déchiffrer les données d'un fichier de sauvegarde. Le déchiffrement d'un cas où Alice connaît Bob donne les informations suivantes:


```
{
    "backup_json_version": 0,
    "engine": {
        "identity_manager": "[
          {
              \"private_identity\":{
                  \"encryption_private_key\":\"kgAAAD [...] KB\",
                  \"mac_key\":\"kAAAAD [...] yOE=\",
                  \"server_authentication_private_key\":\"kgAA [...] lv\"
              },
              \"owned_identity\":\"aHR [...] Z6f\",
              \"contact_identities\":[
                  {
                      \"contact_groups\":[],
                      \"trusted_details\":{
                          \"serialized_details\":\"{
                              \\\"first_name\\\":\\\"Bob\\\"
                          }\",
                          \"version\":0
                      },
                      \"contact_identity\":\"aHR0 [...] 8k07\",
                      \"trust_origins\":[
                          {
                              \"trust_type\":0,
                              \"timestamp\":1588601037292
                          }
                      ],
                      \"published_details\":{
                          \"serialized_details\":\"{
                              \\\"first_name\\\":\\\"Bob\\\"
                          }\",
                          \"version\":0
                      },
                      \"trust_level\":\"4.0\"}],
              \"api_key\":\"5288AFB8-BFE0-2AB9-CB24-7B93A54BE5D5\",
              \"published_details\":{
                  \"serialized_details\":\"{
                      \\\"first_name\\\":\\\"Alice\\\"
                  }\",
```

```
                \"version\":0
            }
        }]"
    },
    "backup_timestamp": 1588602841530
}
```

Conclusion

La fonctionnalité de chiffrement du fichier de sauvegarde repose d'un côté sur des API éprouvées et fournies par *Apple* (AES, HMAC, zlib) et de l'autre sur des implémentations spécifiques d'algorithmes connus (courbes 25519, KEM, PRNG).

Le design des fonctionnalités cryptographiques ayant déjà été analysé dans Analyse de la spécification cryptographique page 14, il est nécessaire de vérifier que les implémentations sont correctes.

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 ré-implémentation 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 conséquences, 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 au RGS et à l'état de l'art. Aucune vulnérabilité de conception ou d'implémentation n'a été identifiée.

2.1.4.3. Avis d’expert et vulnérabilités potentielles identifiées

Une seule faiblesse a été identifiée lors de l'étude de **[FS1]**: V-03 REJEU-COMMITMENT page 80.

Elle concerne une augmentation du taux de réussite d'une attaque de type *Man-in-the-Middle* sur le protocole *Trust Establishment Protocol with SAS* quand des attaques précédentes ont échouées.

Comme discuté dans V-03 REJEU-COMMITMENT page 80, cette attaque est difficile à mettre en œuvre pour un attaquant et elle ne lui apporte que peu d'avantages. De plus il est facile pour des participants de se rendre compte de la tentative d’attaque.

L'avis du CESTI est que la vulnérabilité mériterait d'être corrigée mais ne constitue pas une réelle réduction du niveau de sécurité de **[FS1]**.

Le reste des 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
V03-REJEU-COMMITMENT	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
l'attaquant	Information publique	0
l'attaquant	Difficile	10
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, tel qu'illustré dans la figure suivante où deux contacts *Bob* ont été introduits à *Dave*:

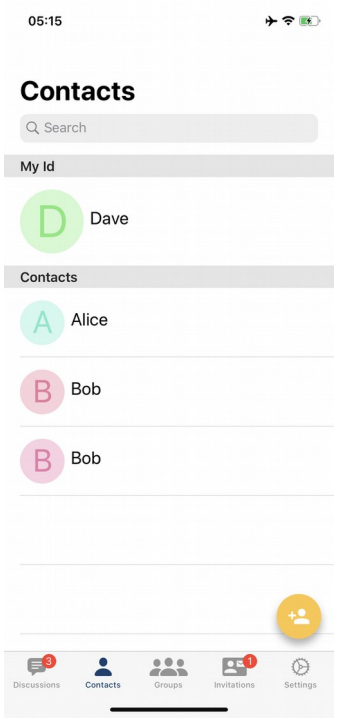

Illustration 17: Deux contacts de même nom dans le carnet d'adresse de Dave

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:

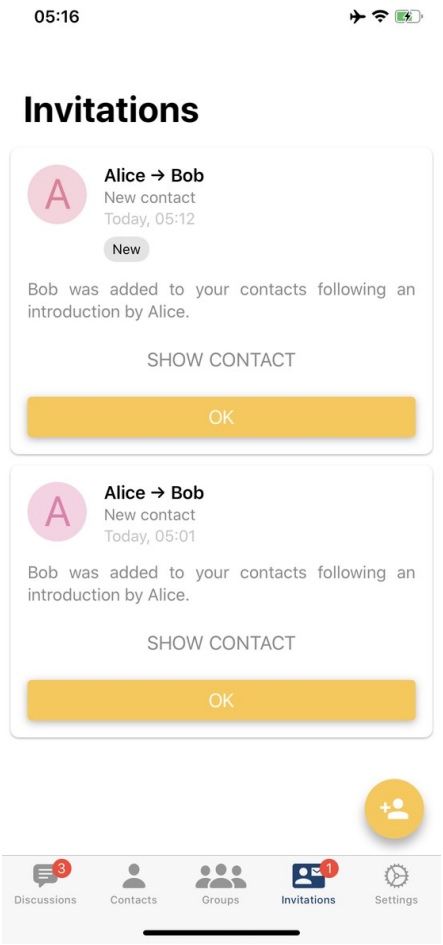

Illustration 18: L'onglet Invitation montre les deux introduction de contacts homonymes

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 la CSPN. Il ne fera donc pas l'objet d'une cotation.

2.2.3. V-03 REJEU-COMMITMENT

Le protocole *Trust Establishment Protocol with SAS* permet la mise en relation de deux tiers en s'accordant sur un secret commun de 8 chiffres, le SAS. La sécurité de ce protocole contre le *Man-in-the-Middle* imperceptible est donné pour un taux de succès de l'attaque à 10^{-8}. La suite détaille une faiblesse avec laquelle cette probabilité peut être réduite suite à des échecs de mises en relation de manière perceptible.

Procédure d'échange de graines

Si Eve veut mettre en place un Man-in-the-Middle sans que les participants Alice et Bob ne s'en rendent compte elle doit négocier indépendamment le protocole avec chacune des parties et trouver un moyen pour qu'Alice et Bob calculent le même SAS.

Le calcul du SAS fait intervenir (entre autres) deux graines choisies par chacun des correspondants. Grossièrement l'échange de graine suit le schéma suivant.

- Alice fourni un haché de sa graine (avec en plus son identité et un nonce) à Bob, appelé le *Commitment*.
- Bob envoie sa graine à Alice.
- Alice envoie le *Décommitment* c'est à dire sa graine, identité et nonce correspondant au haché précédent.
- Bob vérifie que le haché des données reçues correspond avec le haché initial.

Cette manière de procéder assure l'indépendance des graines, c'est à dire qu'Alice au moment de choisir sa graine ne connaît pas celle de Bob et inversement. Cette propriété est cruciale pour la sécurité du protocole, car si l'un des deux participants (ou un attaquant) connaît à l'avance la graine du correspondant sans avoir déjà fixé la sienne alors il serait en mesure de choisir ou du moins d'influencer le calcul du SAS.

Calcul déterministe de la graine de Bob

Lors de l'échange la graine choisie par Alice est choisie aléatoirement. Cependant dans la conception du protocole par Olvid, la graine choisie par Bob ne l'est pas. Elle est en effet déterministe et est dérivée d'un secret connu de Bob et du *commitment* envoyé par Alice.

L'extrait suivant montre la procédure de génération de la graine de Bob. La fonction *getDeterministicSeedForOwnedIdentity* est appelée avec *diversifiedUsing* data qui représente les octets du *commitment.*

```
public func getDeterministicSeedForOwnedIdentity(_ identity: ObvCryptoIdentity,
diversifiedUsing data: Data, within obvContext: ObvContext) throws -> Seed {
    guard let ownedIdentityObj = OwnedIdentity.get(identity, delegateManager:
delegateManager, within: obvContext)  else {
        throw ObvIdentityManagerError.ownedIdentityNotFound.error(withDomain:
ObvIdentityManagerImplementation.errorDomain)
    }
    guard !data.isEmpty else {
        throw ObvIdentityManagerError.diversificationDataCannotBeEmpty.error(withDomain:
ObvIdentityManagerImplementation.errorDomain)
    }
    let sha256 = ObvCryptoSuite.sharedInstance.hashFunctionSha256()
    let fixedByte = Data([0x55])
    var hashInput = try MAC.compute(forData: fixedByte, withKey:
ownedIdentityObj.ownedCryptoIdentity.secretMACKey)
    hashInput.append(data)
    let r = sha256.hash(hashInput)
    guard let seed = Seed(with: r) else {
        throw ObvIdentityManagerError.failedToTurnRandomIntoSeed.error(withDomain:
ObvIdentityManagerImplementation.errorDomain)
    }
    return seed
}
```

La conséquence est que pour un *commitment* donné, qui fixe la graine utilisée par Alice, Bob générera toujours la même graine puis le même SAS. Cela signifie que si Alice (ou un attaquant) ré-utilise un *commitment* utilisé lors d'une précédente mise en place du protocole, il est en mesure de prévoir à l'avance le SAS que Bob calculera.

Ce design est en fait prévu pour la gestion future du multi-appareils, hors cadre de l'étude.

Ce comportement ne remet pas en cause l'indépendance des graines car Alice (ou un attaquant) en fournissant à nouveau un *commitment* utilise bien une graine qui a été choisie sans connaissance de celle de Bob. Cependant cela donne une information intéressante à Alice (ou un potentiel attaquant) qui apprend un couple constant (*commitment,* SAS calculé par Bob).

Attaque via rejeu de commitment

À partir des résultats précédents, on peut alors imaginer une attaque dans laquelle Eve, en mesure de contrôler les communications (serveur d'Olvid) ainsi que le vecteur de mise en relation (QR code échangé / invitation par e-mail ou SMS), essaye de mettre en place un Man-in-the-Middle ou simplement d'usurper l'identité d'Alice auprès de Bob.

Pour arriver à ses fins, Eve va construire une base de donnée contenant des couples (*commitment*, SAS calculés par Bob) pour des *commitments* qu'elle maîtrise, c'est à dire avec des identités envoyées à Bob choisies.

L'idée de l'attaque est de récupérer assez d'information lors de mises en relation ratées entre Alice et Bob pour devenir capable de "choisir" la graine de Bob parmi un ensemble grandissant, en attendant la tentative où Alice générerait un SAS que Eve pourrait faire calculer à Bob.

Eve suit donc la procédure suivante:

1. Alice, qui pense échanger avec Bob, commence le protocole de mise en relation avec Eve en envoyant un *commitement.*
2. Eve répond à Alice avec une graine quelconque
3. Alice fourni à Eve le *décommitment,* calcule le SAS et son mobile est en attente de la confirmation du demi-SAS qui doit être fourni par Bob
4. Eve, retrouve le SAS calculé par Alice et regarde dans sa base si elle connaît un *commitment* qui ferait générer à Bob le même SAS
5. Si aucun *commitment* n'est dans la base
    1. Eve commence le protocole de mise en relation avec Bob en envoyant un *commitement* dans lequel elle choisit l'identité cryptographique présentée ainsi qu'une graine et un nonce. Eve stock ce *commitement* utilisé.
    2. Bob répond à Eve avec sa graine, calcule le SAS et son mobile est en attente de la confirmation du demi-SAS qui doit être fourni par Alice
    3. Eve reçoit la graine calculée par Bob et retrouve le SAS qu'il a calculé. Elle stocke cette information dans sa base.
    4. Alice et Bob échangent les demi-SAS respectivement calculés. Ces derniers sont différents avec une probabilité de 1 - 10^-8 et le *MitM* échoue
    5. Eve recommence l'attaque à partir de l'étape 1.
6. Si Eve trouve un *commitment* valable, alors elle envoie ce dernier à Bob
    1. Bob retrouve le seed attendue et calcul le même SAS qu'Alice
    2. Alice et Bob échangent les demi-SAS sur le canal authentique, ces derniers correspondent des deux cotés et le protocole se termine alors que Alice et Bob échangent en réalité avec Eve.

À chaque fois que la mise en relation échoue à l'étape 5.5 alors Eve apprend un nouveau couple (*commitment*, SAS calculé par Bob) et augmente ses chances à l'étape 4. pour réussir son *MitM.*

Discussion

L'attaque précédente permet à Eve, en faisant échouer des mises en relation entre Alice et Bob de successivement augmenter ses chances de réussite d'un *MitM.* L'augmentation de la probabilité de réussite du *MitM* est complexe à calculer de manière exacte. On peut cependant considérer le pire cas pour essayer de mesurer l'impact de la faiblesse présentée. Le pire cas correspond à celui où chaque mise en relation ratée apporte à Eve un nouveau couple (*commitment*, SAS calculé par Bob), pour un SAS dont elle ne disposait pas encore de *commitement* associé. Ce cas est le moins favorable pour Alice et Bob car, pour plusieurs c*ommitment* donnés, Bob se retrouve normalement assez vite à calculer des SAS identiques (paradoxe des anniversaires).

Dans ce cas, au bout de N tentatives échouées de *MitM*, Eve n'a non plus une probabilité de succès de 1 / 10^8 mais une probabilité de 1 / (10^8 - N) pour sa prochaine tentative. On constate alors que le gain, même s'il existe, n'est pas significatif sans qu'un grand nombre de tentatives aient échouées. Ceci, alors que l'on considère déjà le cas le plus favorable pour un attaquant. Par exemple, faire échouer 5 tentatives n'aurait que peu d'impact sur l'augmentation du taux de réussite d'Eve. En contre-partie pour Alice et Bob il est probable que l'échec de 5 tentatives de mise en relation les rendent méfiants.

Il est important aussi de noter que dans l'attaque précédente, quand Eve fait une tentative de *MitM* et quelle ne trouve pas dans sa base de *commitment* correspondant au SAS d'Alice, Eve ne peut pas "abandonner" son attaque et laisser Alice et Bob s'entendre sur un SAS. En effet, à ce stade de l'échange Eve a déjà fourni à Alice une fausse identité de Bob qui est utilisée dans le calcul du SAS. De plus Eve ne peut pas simplement glaner de l'information en faisant une écoute passive, car les messages échangés sont chiffrés de manière asymétrique avant l'établissement du canal sécurisé.

Conclusion

La faiblesse identifiée constitue une vulnérabilité qui mériterait d'être corrigée mais qui ne remet pas en cause la sécurité de Olvid. L'exploitation de cette vulnérabilité nécessite des moyens importants pour un attaquant avec la nécessité de maîtriser les canaux de communications (ceux de l'application et celui utilisé par Bob pour fournir son identité: QRCode via e-mail, SMS ou autre). Essayer d'exploiter la faiblesse a de grandes chances d'être perçu par les utilisateurs et de créer de la suspicion. Enfin le biais apporté ne semble pas suffisamment intéressant pour réduire de manière considérable la sécurité

Cette vulnérabilité est jugée comme exploitable, mais n'as pas été exploitée en vu de la complexité de mise en œuvre.

Lors d'un entretien avec le CESTI, les développeurs d'Olvid ont confirmé la découverte et ont manifesté leur intention de corriger la vulnérabilité. La solution technique évoquée ne nécessiterait pas de refonte globale de l'application.

Une cotation est donnée suivant **[CEM]**

Facteur	Valeur	Score
Temps mis pour l'exploitation	> 6 mois	19
Expertise de l'attaquant	Expert	6
l'attaquant	Information publique	0
l'attaquant	Difficile	10
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

3. Synthèse de l'évaluation

3.1. Synthèse de la sécurité du produit

L'application Olvid propose un modèle de messagerie sécurisée pour iOS, 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 et que l'utilisation du langage Swift, en particulier de son système de typage, rend les éventuels problèmes moins probables.

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	10
Exploitation des résultats	2
Synthèse et rédaction du rapport	5
Total	40

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 obtention de cette dernière.

4. Références

Sigle	Référence
[CEM]	Common Methodology for Information Technology Security Evaluation : Evaluation Methodology, version en vigueur.
[CIBLE]	Cible de Sécurité CSPN – Olvid v2.2 du 05/04/2020
[H1]	Hypothèse de [CIBLE] sur les conditions d'installation de Olvid
[H2]	Hypothèse de [CIBLE] sur les utilisateurs de Olvid
[H3]	Hypothèse de [CIBLE] sur le serveur de l'application
[M1]	Menace de [CIBLE] concernant la modification d'information sur le réseau
[M2]	Menace de [CIBLE] concernant l'interception d'information sur le réseau
[M3]	Menace de [CIBLE] concernant l'usurpation d'identité
[M4]	Menace de [CIBLE] concernant la récupération d'un fichier de sauvegarde
[FS1]	Fonction de sécurité de [CIBLE] : Authentification des utilisateurs
[FS2]	Fonction de sécurité de [CIBLE] : Authentification des échanges
[FS3]	Fonction de sécurité de [CIBLE] : Chiffrement des messages et des pièces jointes
[FS4]	Fonction de sécurité de [CIBLE] : Chiffrement des sauvegardes du carnet de contact
[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.
[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.
[NIST.SP.800-90Ar1]	https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90Ar1.pdf
[DRBG_HMAC_PROOF]	Security Analysis of DRBG Using HMAC in NIST SP 800-90 - https://core.ac.uk/download/pdf/61342884.pdf
[SecRandomCopyBytes]	https://developer.apple.com/documentation/security/1399291-secrandomcopybytes
[DRBG_TESTVECTORS]	https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Algorithm-Validation-Program/documents/drbg/drbgtestvectors.zip
[idevicebackup2]	https://github.com/libimobiledevice/libimobiledevice/blob/master/tools/idevicebackup2.c
[checkra1n]	https://checkra.in/
[l'entropie de Shanon]	https://fr.wikipedia.org/wiki/Entropie_de_Shannon

5. Annexes

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

SHA256 : be3a70e992718a4977ab3e927681e7259b669fd8d6290aeca704febe55bf4722

Les fichiers présents sont les suivants :

```
annexes
├── BackupSeed.py
├── decrypt_backup.py
├── DerivedKeysForBackup.py
├── dummy_server
│   ├── ca.crt
│   ├── ca.key
│   ├── ca.mk
│   ├── ca.srl
│   ├── dump-oblivious-channel.pickle
│   ├── dump.pickle
│   ├── dump-trust-establishment.pickle
│   ├── keys-asymetric.json
│   ├── keys-symetric.json
│   ├── ObvEngine-alice.sqlite
│   ├── ObvEngine.sqlite
│   ├── olvid.conf
│   ├── olvid.crt
│   ├── OlvidCrypto
│   │   ├── EdwardsCurve.py
│   │   ├── hmac_drbg.py
│   │   ├── __init__.py
│   │   ├── KEM.py
│   │   ├── PRNG.py
│   │   └── Symmetric.py
│   ├── OlvidDummyClient
│   │   └── __init__.py
│   ├── OlvidDummyServer
│   │   ├── Encoding.py
│   │   └── __init__.py
│   ├── olvid.key
│   ├── olvid.pem
│   ├── PoW
│   ├── scripts
│   │   ├── olvid_decode_oblivious_channel.py
│   │   ├── olvid_decode.py
│   │   ├── olvid_decode_trust_establishment.py
│   │   ├── olvid_dummy_client.py
│   │   ├── olvid_dummy_server.py
│   │   └── olvid_trust_establishment_mitm.py
│   ├── setup.py
│   ├── storage-invit.pickle
│   └── storage.pickle
├── hmac_drbg.py
├── KEM.py
├── PRNG.py
├── Tests.py
├── TestVectorsBackupKeysFromBackupSeedString.json
├── TestVectorsBackupSeedFromString.json
├── TestVectorsPRNGGenBigInt.json
└── TestVectorsPRNGWithHMACWithSHA256.json
```