

Specifications of Olvid

–

Application and Server

Date	November 3, 2026
Number of Pages	203
Written by	Dr. Thomas Baignères Dr. Matthieu Finiasz

Contents

About this document 10

I Notations and Conventions 13

1 Notation, Procedure Default Values, and Conventions 13

2 Names, Lengths, etc. 14

II Cryptographic Primitives 15

3 Preliminaries 15
3.1 Exposed vs. Internal Primitives . . . 15
3.2 Cryptographic Keys . . . 15
3.2.1 Symmetric Keys . . . 16
3.2.2 Public Keys . . . 17
3.2.3 Private Keys . . . 18
3.3 List of Key Byte Identifiers . . . 19

4 Hash Function 19
4.1 SHA-256 . . . 20
4.2 SHA-512 . . . 20

5 Block Cipher 20
5.1 AES-256 . . . 21

6 Symmetric Encryption 22
6.1 AES-256 in CTR Mode . . . 23

7 Message Authentication Code 26
7.1 HMAC with SHA-256 . . . 27

8 Key Derivation Function (KDF) 29
8.1 KDF Based on SHA-256 . . . 30

9 Commitment 30
9.1 Commitment based on SHA-256 . . . 31

10 Pseudorandom Generator 32
10.1 FIPS 800-90A HMAC_DRBG Based on SHA-256 . . . 34

11 Authenticated Encryption 35
11.1 Encrypt-then-MAC with AES-256 and HMAC with SHA-256 . . . 37

12 Edwards Curves 39
12.1 Curve25519 . . . 45
12.2 MDC . . . 45

13 Public and Private Keys over Edwards Curves **46**

14 Signature **48**
14.1 Signature Key Generation over Curve25519 53
14.2 Signature Key Generation over MDC 54

15 Authentication **55**
15.1 Authentication over Curve25519 61
15.2 Authentication over MDC 61

16 Key Encapsulation Mechanism (KEM) **62**
16.1 KEM over Curve25519 67
16.2 KEM over MDC 67

17 Public Key Encryption **68**

18 Cryptographic Identity **69**

19 Owned Cryptographic Identity **71**
19.1 Deterministic Seeds 73

III Encodings **75**

20 Encoding Structure **75**
20.1 Byte Identifiers 75
20.2 Content Byte-Length 75

21 Byte Representation of Integers and Lengths **76**
21.1 32-bit Unsigned Integer 76
21.2 64-bit Unsigned Integer 76
21.3 64-bit Signed Integer 76
21.4 Unsigned Big Integer 77
21.5 Signed Big Integer 77
21.6 Lengths 77

22 Encoding Procedures **77**
22.1 Common Encoding/Decoding Rules 78
22.2 Encoding an Array of Bytes 78
22.3 Encoding a String 78
22.4 Encoding a 64-bit Integer 79
22.5 Encoding a Boolean 79
22.6 Encoding an Unsigned Big Integer 79
22.7 Packing Elements and Encoding a List 80
22.8 Encoding a Dictionary 81
22.9 Padded Encodings 82
22.10 Encoding Other Types 82
22.11 Encoding a Cryptographic Key 83
22.12 Encoding a Symmetric Key 83
22.13 Encoding a Public Key 84
22.14 Encoding a Private Key 84

IV Message Structure and Communication Channels 86

23 General Message Structure 87
23.1 Protocol Message Structure . . . 87
23.2 Application Message Structure . . . 87

24 Encryption 90
24.1 Message Encryption . . . 90
24.2 Extended Payload Encryption . . . 93
24.3 Attachment Encryption . . . 93
24.4 Return Receipt Encryption . . . 94

25 Pre-Keys Management 94

V Cryptographic Protocols 96

26 Trust Establishment Protocol with SAS 97
26.1 Purpose and High Level View . . . 97
26.2 Cryptographic Details . . . 100

27 Trust Establishment Protocol with Mutual Scan 100
27.1 Purpose and High Level View . . . 100
27.2 Cryptographic Details . . . 102

28 Contact Mutual Introduction Protocol 102
28.1 Purpose and High Level View . . . 102
28.2 Cryptographic Details . . . 102

29 Keycloak Contact Addition Protocol 105
29.1 Purpose and High Level View . . . 105
29.2 Cryptographic Details . . . 105

30 One-to-One Contact Invitation Protocol 106
30.1 Purpose and High Level View . . . 106
30.2 Cryptographic Details . . . 107

31 Contact Management Protocol 107
31.1 Purpose and High Level View . . . 107
31.2 Cryptographic Details . . . 107

32 Channel Creation with Contact Device Protocol 107
32.1 Purpose and High Level View . . . 107
32.2 Cryptographic Details . . . 109

33 Channel Creation with Owned Device Protocol 109
33.1 Purpose and High Level View . . . 109
33.2 Cryptographic Details . . . 110

34 Full Ratchet Protocol 110
34.1 Purpose and High Level View . . . 110
34.2 Cryptographic Details . . . 112

35 Device Discovery Protocol **112**
35.1 Purpose and High Level View . . . 112
35.2 Cryptographic Details . . . 112

36 Owned Device Discovery Protocol **112**
36.1 Purpose and High Level View . . . 112
36.2 Cryptographic Details . . . 113

37 Owned Device Management Protocol **113**
37.1 Purpose and High Level View . . . 113
37.2 Cryptographic Details . . . 114

38 Device Capabilities Discovery Protocol **114**
38.1 Purpose and High Level View . . . 114
38.2 Cryptographic Details . . . 114

39 Identity Details Publication Protocol and Contact Picture Download Child Protocol **114**
39.1 Purpose and High Level View . . . 114
39.2 Cryptographic Details . . . 115

40 Owned Identity Transfer Protocol **115**
40.1 Purpose and High Level View . . . 115
40.2 Cryptographic Details . . . 116

41 Keycloak Binding and Unbinding Protocol **117**
41.1 Purpose and High Level View . . . 117
41.2 Cryptographic Details . . . 117

42 Owned Identity Deletion Protocol **118**
42.1 Purpose and High Level View . . . 118
42.2 Cryptographic Details . . . 118

43 Synchronization Protocol **119**
43.1 Purpose and High Level View . . . 119
43.2 Cryptographic Details . . . 119

44 Group Invitation Protocol **119**
44.1 Purpose and High Level View . . . 119
44.2 Cryptographic Details . . . 121

45 Group Management Protocol **121**
45.1 Purpose and High Level View . . . 121
45.2 Cryptographic Details . . . 122

VI Groups v2 **123**

46 General Concepts **123**

47 Cryptographic Structures **126**

48 Groups v2 Protocol **127**
48.1 Group Creation . . . 127
48.2 Invitation and Group Joining . . . 129
48.3 Group Update . . . 130
48.4 Member Removal . . . 133
48.5 Leaving a Group or Declining an Invitation . . . 133
48.6 Group Disbanding . . . 134
48.7 Group Keys After Channel Creation . . . 134
48.8 Keycloak Groups . . . 134

49 Groups v2 Picture Download Protocol **135**

VII Backups **136**

50 Backups **136**
50.1 Overview . . . 136
50.2 Backup Seeds . . . 136
50.3 Key Derivation . . . 137
50.4 Storage on the Server . . . 138
50.5 Backup Encryption . . . 138
50.6 Backup Contents . . . 138
50.7 Backup Creation . . . 139
50.8 Backup Restore . . . 139
50.9 Backup Deletion . . . 139

51 Legacy Backups **140**
51.1 Key Derivation . . . 140
51.2 Backup Contents . . . 140
51.3 Backup Encryption . . . 140
51.4 Backup Decryption . . . 141
51.5 Backup Restore . . . 141

VIII Secure Calls **142**

52 Signaling Messages **142**

53 TURN Servers **143**

54 Group Calls **143**

55 Security **144**

IX History Transfer **145**

56 WebRTC History Transfer **145**
56.1 Transfer Establishment . . . 145
56.2 Data Transfer . . . 145
56.3 Security . . . 146

57 Wi-Fi Aware History Transfer **146**

58 Zip File History Transfer **146**

X Keycloak - Olvid User Directory 147

59 General Concepts **147**

60 Configuration and Authentication **148**

61 Signatures **149**

62 Synchronization **150**

63 Transfer Restriction **150**

64 Keycloak REST API **150**
64.1 Get owned details (`me`) . . . 150
64.2 Get groups (`groups`) . . . 152
64.3 Upload `identity` (`putKey`) . . . 152
64.4 Download signed `identity` (`getKey`) . . . 153
64.5 Contact search (`search`) . . . 153
64.6 Revocation test (`revocationTest`) . . . 154
64.7 Contact verification (`verify`) . . . 154
64.8 Group profile picture download (`getData`) . . . 155
64.9 Transfer proof (`transferProof`) . . . 155
64.10 Magic link authentication (`getMagicSession`) . . . 156
64.11 Identity-based authentication (`requestChallenge`, `getSession`) . . . 156
64.12 Olvid discovery document (`.well-known/olvid`) . . . 157

XI Data Storage on Android 158

65 Database Encryption **158**

66 Files **158**

67 Application Lock **159**

XII Olvid Server API 160

68 Server Configuration **161**

69 Server Authentication API **161**
69.1 Get authentication challenge . . . 161
69.2 Authenticate and get client session token . . . 162

70 API Keys and Subscriptions API **163**
70.1 Query API key status . . . 164
70.2 Register API key . . . 164

70.3 Free trial API key query and retrieval . . . 164
70.4 In-App purchase receipt verification . . . 165

71 Message Upload API **166**
71.1 Upload message and get UID . . . 166
71.2 Batch message upload . . . 168
71.3 Refresh private upload signed URLs . . . 169
71.4 Get attachment upload progress . . . 170
71.5 Cancel attachment upload . . . 170

72 Message Download API **171**
72.1 Download messages and list attachments . . . 171
72.2 Download message extended payload . . . 172
72.3 Refresh private download signed URLs . . . 172
72.4 Delete message and attachments . . . 173

73 Device Management API **174**
73.1 Register push notification . . . 174
73.2 Device discovery . . . 176
73.3 Owned device discovery . . . 177
73.4 Device management . . . 177

74 Group V2 Management API **178**
74.1 Group creation . . . 178
74.2 Group download . . . 179
74.3 Group lock . . . 180
74.4 Group update . . . 180
74.5 Group log put . . . 181
74.6 Group deletion . . . 182

75 User Data Management API **182**
75.1 Put user data . . . 182
75.2 Get user data . . . 183
75.3 Refresh user data . . . 183
75.4 Delete user data . . . 184

76 Backup API **184**
76.1 Backup creation . . . 184
76.2 Backup upload . . . 185
76.3 Backup list . . . 186
76.4 Backup deletion . . . 186

77 Other REST API Entry Points **187**
77.1 Upload return receipts . . . 187
77.2 Retrieve TURN credentials to initiate a call . . . 188
77.3 Upload pre-key . . . 188
77.4 Keycloak queries . . . 189

78 WebSocket API **189**
78.1 Device registration . . . 190
78.2 Ping . . . 191

78.3 Message notification . . . 191
78.4 Owned devices notification . . . 191
78.5 Keycloak notification . . . 192
78.6 Push topic notification . . . 192
78.7 Return receipt download . . . 192
78.8 Return receipt deletion . . . 193

79 Transfer WebSocket API **193**
79.1 Source connection . . . 194
79.2 Target connection . . . 194
79.3 Relay request . . . 195
79.4 Relayed message . . . 195

80 Push Notifications Content **195**
80.1 Android - Firebase push notifications . . . 195
80.2 iOS - Apple push notifications . . . 196

Appendices **200**

A Elliptic Curves **200**
A.1 Edwards Curves . . . 200
A.2 Equivalence with Montgomery Curves . . . 200
A.2.1 Montgomery Ladder for Edwards Curves . . . 201
A.2.2 Montgomery Ladder . . . 201
A.2.3 The Ladder on a Montgomery Curve . . . 202
A.2.4 The Ladder on an Edwards Curve . . . 202
A.3 The Curves We Consider in these Specifications . . . 202
A.3.1 Million Dollar Curve . . . 203
A.3.2 Curve25519 . . . 203

About this document

Olvid is a secure instant messaging application similar in functionality to many other instant messaging applications like WhatsApp, Signal, Citadel, etc. Olvid is designed as two independent modules: a *cryptographic engine* in charge of all the cryptographic operations, and an *application* (the instant messaging user interface), working on top of this engine, able to exchange text messages and attachments of any kind. The engine is "generic" in the sense that its API could be used for any kind of communication between persons. This document focuses on the cryptographic engine. The application layer is only described where it plays a role in security: secure calls, message history transfer, Keycloak integration, and local data storage on Android. In particular, the word "message" will always be used with its cryptographic meaning in mind.

This document provides all the technical details about the cryptographic algorithms and protocols used in the cryptographic engine of Olvid. Roughly, Olvid makes it possible for users to create cryptographic keys, to exchange key material (using various methods), to use this material to send messages in a secure way, and to create secure channels that can be used by the application layer.

Architecture & Trust Model

In Olvid, messages exchanged between users transit through servers. Similarly to the SMTP protocol used in email, each user uses a specific Olvid server to receive messages, but several Olvid servers can coexist. However, unlike SMTP, messages for a given user are directly uploaded to their Olvid server, and messages are not relayed through multiple servers.

The Olvid servers do not play any role in the security of exchanges between users. Olvid servers are simple "drop boxes" where anyone can anonymously deposit messages. When receiving a message, the server notifies the recipient that a new message is available, and the recipient (after authenticating themselves) simply retrieves the message and deletes it.

The security of Olvid relies on the cryptographic protocols implemented in the cryptographic engine and on the real-world connection/relationship and trust between users, but does not assume any trusted third party. The main hypotheses of the security model of Olvid are:

- the user devices (smartphone, computer, etc.) on which the Olvid application runs are healthy and can be trusted to execute the cryptographic protocols as they were implemented.
- the users exchanging messages through Olvid have a real-world connection/relationship and they have some level of trust in each other. Depending on the nature of this connection/relationship, different protocols are implemented to initiate a secure communication in Olvid.
- the servers are mostly "honest but curious". What this means is that servers are not trusted, they are considered adversaries, but still behave as expected most of the time. In other words, they will try to learn everything they can about the users and what they exchange, but try to remain undetected and thus operate the service normally most of the time.

For identities managed by a Keycloak server (see Part X), the Keycloak server of the organization is additionally trusted to certify the identities of its users.

Concepts & Terminology

Identity. In Olvid, each user is identified by an `identity`. An `identity` is composed of:

- a server URL (so other users know where to post messages)
- two public keys: one for encryption, one for signature/authentication

As opposed to X.509 certificates, the `identity` itself does not contain any identification element that would tie the public keys it contains to a real-world identity.

Device. Each user can have several devices (phone, computer, etc.), each identified by a `deviceUid` (a random 32-byte sequence). Because of this, there are multiple ways to send a message to a user. It can be sent to:

- an `identity` → broadcast message
- a single device → unicast message
- several devices → multicast message (which has nothing to do with TCP/IP multicast)

Contrary to what is usually understood as "broadcast", broadcast messages are not necessarily received by all devices. Here, the message is put on the server and all recipient devices are notified. The first device to download the message "wins": the message is received by this device and deleted from the server. But there is no guarantee that the message is received by a single device as two devices could download it simultaneously.

Each (`identity`, `deviceUid`) pair is registered on the server so it can receive push notifications when a new message arrives. Anyone can query the server to get the list of registered `deviceUid` for an `identity` (see the device discovery protocol in Section 35).

With a multi-device license, an Olvid user may register as many devices as they want, without limitation. Multi-device is handled in Olvid by having a copy of the long-term keys on each device: there is no master device or master key, all devices play an equivalent role. Adding a device can be done either by restoring a backup (see Part VII) or transferring a profile to another device (see Section 40).

Olvid users without a paid license are limited to having one `deviceUid` registered on the server: adding a new device requires deciding which device should expire after 30 days. This way, when replacing a device, even if the new device should be operational in a few seconds, users have up to 30 days to make sure all messages properly reach the new device.

QR-codes and invitation link. An `identity` does not contain any identification element that would tie the cryptographic keys it contains to a real-world person. When inviting someone to start a discussion on Olvid, it is important that the invited person can identify who this invitation is (supposedly) coming from. Invitations thus contain both the `identity` of a user, and a "display name" packed in an *Invitation Link* similar to this:

```
https://invitation.olvid.io/#AwAAAIgAAAAAWmhOdHBzOi8vc2VydmVyLm9sdmlkLml\
vAADoTcM7E5duFaKw1mpuyGROJkSM51KOEulxyQEdabcimADZmeYVvTlSy5kkAtfM4o2JJuj\
sZTkrSG-B6VshvRU5gwAAAAAkTWFOdGhpZXUgRmluaWFzeiAoRGV2ZWxvcGVyIEAgT2x2aWQp
```

Such an Invitation Link can also be embedded in a QR-code for direct scanning on a smartphone as shown in Figure 1.



Figure 1: Example of a QR-code containing an Invitation Link.

Part I
Notations and Conventions

In what follows, $[0, 1, \ldots, 255] = [\texttt{0x00}, \texttt{0x01}, \ldots, \texttt{0xFF}]$. In other words, we interchangeably use the notation `0x10` (in hexadecimal) or 16 (in base 10).

We denote by `[UTF-8 string]` the set of all arrays of non-zero bytes corresponding to a valid UTF-8 encoding, terminated by a zero byte.

1 Notation, Procedure Default Values, and Conventions

Given a procedure Proc taking, e.g., two integer parameters a and b, we use the following notation to indicate that a is a mandatory parameter and that, when the optional parameter b is not specified, its default value is 13:

$$\mathsf{Proc}(a, b = 13)$$

Assuming that the previous procedure returns the sum of a and b, then $\mathsf{Proc}(4, 2)$ returns 6, and $\mathsf{Proc}(1)$ returns 14. Python and Swift programmers should be familiar with this notation.

Some of the procedures we describe may fail. When a procedure fails, it returns $\perp$. We adopt the convention that, by default, any procedure that calls a procedure that fails also fails.

All the procedures we define in these specifications have strict input parameter domains. We assume that they *cannot* be called outside of their parameter domain. In practice, this is ensured by implementing these procedures using a strongly typed programming language.

To improve readability we will sometimes provide a table specifying the input/output parameters and the arguments of a procedure. As an example, here is the table for the procedure Proc:

Proc		
parameters		None
in	$\mathbb{Z}$ $\mathbb{Z}$	a $b = 13$
out	$\mathbb{Z}$	c

2 Names, Lengths, etc.

In these specifications, `name` corresponds to zero or more modified UTF-8 encoded characters followed by `0x00` (note that there cannot be a `0x00` inside the `name`).

When $c \in \{0, \ldots, 255\}^*$ is an array of bytes, we denote by

$$\mathsf{len}(c)$$

the length of the array. For example, if $c \in \{0, \ldots, 255\}^\ell$ for some $\ell > 0$, we have $\mathsf{len}(c) = \ell$.

When s is a string, we denote by

$$\mathsf{len}(s)$$

the byte-size of its UTF-8 encoded characters, including the last zero byte. For example, $\mathsf{len}(\texttt{"curve"}) = 6$. When restricting to 7-bit ASCII characters, the length simply corresponds to the string length plus one.

When x is an unsigned big integer, we denote by

$$\mathsf{len}(x) = \left\lfloor \frac{\log_2(x)}{8} \right\rfloor + 1,$$

with $\mathsf{len}(0) = 1$.

Part II
Cryptographic Primitives

In this part, we describe in full detail the API of the cryptographic primitives used within the cryptographic protocols. Each primitive (hash function, symmetric encryption, public key encryption, etc.) can have one or several associated functions that we describe in pseudo-code using the conventions presented in Section 1.

3 Preliminaries

3.1 Exposed vs. Internal Primitives

The cryptographic library used within Olvid defines many cryptographic primitives. Certain primitives, such as block cipher (Section 5) or symmetric encryption (Section 6), are not exposed to the rest of the framework, but only used internally, within the cryptographic library. Other cryptographic primitives, such as authenticated encryption (Section 11), are exposed and leverage the internal primitives.

3.2 Cryptographic Keys

Cryptographic keys are usually considered a "simple" type such as an array of bytes for symmetric keys or sometimes integers or points on an elliptic curve for public/private key pairs. Within these specifications, this is *only* the case for internal primitives (see Section 3.1), but not for exposed primitives.

When exposed, keyed cryptographic primitives use a complex type denoted CryptographicKey in order to define the domain of their key space. This makes it possible to add essential information (such as the exact keyed cryptographic algorithm associated with a key) to the raw bytes of the key. As explained in Section 22.11, this also makes it possible to provide robust encoding/decoding procedures for cryptographic keys and to enforce the use of an appropriate key when considering a particular algorithm. In particular, this makes it impossible to use an AES key to compute an HMAC digest.

Within these specifications, a cryptographic key key of type CryptographicKey always specifies the following four values:

- `key.algoClassByteId`: A byte that specifies the algorithm class of this cryptographic key, such as block cipher, symmetric encryption, MAC, authenticated encryption, signature, DH, etc.
- `key.algoImplemByteId`: Given a specific algorithm class, this byte specifies the particular implementation for this cryptographic key. For, e.g., the block cipher class, this byte indicates

whether the key is an AES or a DFC (Decorrelated Fast Cipher) key.
- key.dict: A dictionary (in the sense of Section 22.8) whose (dictionary) keys depend on the two above bytes.
- key.encodingByteId: A byte that specifies which encoding byte identifier (in the sense of Section 20.1) to use when encoding this cryptographic key.

The type CryptographicKey is an *abstract* type. A CryptographicKey instance will always be an instance of a *concrete* subtype of CryptographicKey. The following initializer will systematically be called by the initializer of subtypes of CryptographicKey.

CryptographicKey (Initializer)		
parameters		None
in	$\{0,\dots,255\}$	`algoClassByteId`
	$\{0,\dots,255\}$	`algoImplemByteId`
	Dictionary	`dict`
	$\{0,\dots,255\}$	`encodingByteId`

We denote by

$$\text{CryptographicKey}(\texttt{algoClassByteId}, \texttt{algoImplemByteId}, \texttt{dict}, \texttt{encodingByteId}) \to \text{key}$$

the call to the CryptographicKey initializer.

```
1: procedure CryptographicKey(algoClassByteId, algoImplemByteId, dict, encodingByteId)
2:     self.algoClassByteId ← algoClassByteId
3:     self.algoImplemByteId ← algoImplemByteId
4:     self.dict ← dict
5:     self.encodingByteId ← encodingByteId
6: end procedure
```

3.2.1 Symmetric Keys

A symmetric key is a particular cryptographic key (see Section 3.2), typically used for block ciphers and MACs. When used for an exposed primitive, a symmetric key is an instance of a complex type SymmetricKey, which is a subtype of CryptographicKey (see Section 3.2). The type SymmetricKey is an *abstract* type. A SymmetricKey instance will always be an instance of a *concrete* subtype of SymmetricKey. The SymmetricKey key type requires concrete subtypes to specify the following static value:

- SymmetricKey.length: the raw byte length of the symmetric key. For example, this is 32 for AES-256.

The following initializer will systematically be called by the initializer of subtypes of SymmetricKey.

SymmetricKey (Initializer)		
parameters		None

in	$\{0,\ldots,255\}$
	$\{0,\ldots,255\}$
	Dictionary
		`algoClassByteId`
		`algoImplemByteId`
		`dict`

We denote by

$$\mathsf{SymmetricKey}(\texttt{algoClassByteId}, \texttt{algoImplemByteId}, \texttt{dict}) \rightarrow \mathsf{symKey}$$

the call to the above SymmetricKey initializer.

```
1: procedure SymmetricKey(algoClassByteId, algoImplemByteId, dict)
2:     encodingByteId ← 0x90
3:     CryptographicKey(algoClassByteId, algoImplemByteId, dict, encodingByteId)
4: end procedure
```

SymmetricKey (Initializer)		
parameters		None
in	$\{0,\ldots,255\}^*$	`b`

We denote by

$$\mathsf{SymmetricKey}(\texttt{b}) \rightarrow \mathsf{symKey}$$

the call to the above SymmetricKey initializer. This method is abstract and only implemented by concrete subtypes of SymmetricKey.

3.2.2 Public Keys

A public key is a particular cryptographic key (see Section 3.2), typically used for verifying digital signatures, encrypting data using a public key encryption scheme, and more. When exposed, a public key is an instance of a complex type PublicKey, which is a particular type of CryptographicKey (see Section 3.2). The type PublicKey is an *abstract* type. A PublicKey instance will always be an instance of a *concrete* subtype of PublicKey. The following initializer will systematically be called by the initializer of subtypes of PublicKey.

PublicKey (Initializer)		
parameters		None

in	$\{0,\ldots,255\}$
	$\{0,\ldots,255\}$
	Dictionary
		`algoClassByteId`
		`algoImplemByteId`
		`dict`

We denote by

$$\mathsf{PublicKey}(\texttt{algoClassByteId}, \texttt{algoImplemByteId}, \texttt{dict}) \rightarrow \mathsf{pubKey}$$

the call to the above PublicKey initializer.

```
1: procedure PublicKey(algoClassByteId, algoImplemByteId, dict)
2:     encodingByteId ← 0x91
3:     CryptographicKey(algoClassByteId, algoImplemByteId, dict, encodingByteId)
4: end procedure
```

PublicKey (Initializer)		
parameters		None
in	$\{0,\ldots,255\}^*$	b

We denote by

$$\mathsf{PublicKey}(\texttt{b}) \rightarrow \mathsf{pubKey}$$

the call to the above PublicKey initializer. This method is abstract and only implemented by concrete subtypes of PublicKey.

3.2.3 Private Keys

A private key is a particular cryptographic key (see Section 3.2), typically used for computing digital signatures, decrypting data using a public key encryption scheme, and more. When exposed, a private key is an instance of a complex type PrivateKey, which is a particular type of CryptographicKey (see Section 3.2). The type PrivateKey is an *abstract* type. A PrivateKey instance will always be an instance of a *concrete* subtype of PrivateKey. The following initializer will systematically be called by the initializer of subtypes of PrivateKey.

PrivateKey (Initializer)		
parameters		None
in	$\{0,\ldots,255\}$ $\{0,\ldots,255\}$ Dictionary	algoClassByteId algoImplemByteId dict

We denote by

$$\mathsf{PrivateKey}(\texttt{algoClassByteId}, \texttt{algoImplemByteId}, \texttt{dict}) \rightarrow \mathsf{privKey}$$

the call to the above PrivateKey initializer.

```
1: procedure PrivateKey(algoClassByteId, algoImplemByteId, dict)
2:     encodingByteId ← 0x92
```

3: CryptographicKey(`algoClassByteId`, `algoImplemByteId`, `dict`, `encodingByteId`)
4: **end procedure**

PrivateKey (Initializer)		
parameters		None
in	$\{0,\ldots,255\}^*$	`b`

We denote by

$$\mathsf{PrivateKey}(\mathtt{b}) \to \mathsf{privKey}$$

the call to the above PrivateKey initializer. This method is abstract and only implemented by concrete subtypes of PrivateKey.

3.3 List of Key Byte Identifiers

`algoClassByteId`		`algoImplemByteId`	
byte	algorithm class	byte	algorithm implementation
0x00	SymEncKey	0x00	AES256CTRKey
0x01	MACKey	0x00	HMACWithSHA256Key
0x02	AuthEncKey	0x00	AES256CTRHMACSHA256Key
0x11	SignaturePublicKeyOverEC	0x00	SignatureOverMDC
		0x01	SignatureOverCurve25519
0x12	KEMPublicKeyOverEC	0x00	KEMOverMDC
		0x01	KEMOverCurve25519
0x14	AuthenticationPublicKeyOverEC	0x00	AuthenticationOverMDC
		0x01	AuthenticationOverCurve25519

4 Hash Function

We denote by H the abstract type common to all hash functions. We denote by ℓ_h the byte-length of a digest.

H		
parameters		$\ell_h \in \mathbb{N}$
in	$\{0,\ldots,255\}^*$	`m`
out	$\{0,\ldots,255\}^{\ell_h}$	`h`

We denote by

$$\mathsf{H}(\mathtt{m}) \to \mathtt{h}$$

the procedure that computes the hash of a message `m`. This method is abstract and only implemented by concrete subtypes of H.

4.1 SHA-256

We denote by SHA256 the concrete subtype of H that computes a hash with SHA-256.

SHA256		
parameters		$\ell_h = 32$
in	$\{0,\dots,255\}^*$	`m`
out	$\{0,\dots,255\}^{32}$	`h`

The specifications of SHA256 are available in [15].

4.2 SHA-512

We denote by SHA512 the concrete subtype of H that computes a hash with SHA-512.

SHA512		
parameters		$\ell_h = 64$
in	$\{0,\dots,255\}^*$	`m`
out	$\{0,\dots,255\}^{64}$	`h`

The specifications of SHA512 are available in [15]. It is currently only used for the verification of signatures (see Section 14).

5 Block Cipher

We denote by E the abstract type common to all block ciphers. We denote by ℓ_k the byte-length of a secret key and by ℓ_m the byte-length of a block.

E.encrypt		
parameters		$\ell_k, \ell_m \in \mathbb{N}$
in	$\{0,\dots,255\}^{\ell_m}$ $\{0,\dots,255\}^{\ell_k}$	`m` `k`
out	$\{0,\dots,255\}^{\ell_m}$	`c`

We denote by

$$\mathsf{E.encrypt}(\mathtt{m}, \mathtt{k}) \to \mathtt{c}$$

the symmetric encryption with the block cipher E of the message $\mathtt{m} \in \{0, \ldots, 255\}^{\ell_m}$ under the key $\mathtt{k} \in \{0, \ldots, 255\}^{\ell_k}$. This method is abstract and only implemented by concrete subtypes of E.

E.decrypt		
parameters		$\ell_k, \ell_m \in \mathbb{N}$
in	$\{0, \ldots, 255\}^{\ell_m}$ $\{0, \ldots, 255\}^{\ell_k}$	c k
out	$\{0, \ldots, 255\}^{\ell_m}$	m

We denote by

$$\mathsf{E.decrypt}(\mathtt{c}, \mathtt{k}) \to \mathtt{m}$$

the symmetric decryption with the block cipher E of the ciphertext $\mathtt{c} \in \{0, \ldots, 255\}^{\ell_m}$ under the key $\mathtt{k} \in \{0, \ldots, 255\}^{\ell_k}$. This method is abstract and only implemented by concrete subtypes of E.

5.1 AES-256

We denote by AES256 the concrete subtype of E that encrypts with AES-256.

AES256.encrypt		
parameters		$\ell_k = 32$, $\ell_m = 16$
in	$\{0, \ldots, 255\}^{16}$ $\{0, \ldots, 255\}^{32}$	m k
out	$\{0, \ldots, 255\}^{16}$	c

The specifications of the encryption procedure of AES256 are available in [11].

AES256.decrypt		
parameters		$\ell_k = 32$, $\ell_m = 16$
in	$\{0, \ldots, 255\}^{16}$ $\{0, \ldots, 255\}^{32}$	c k
out	$\{0, \ldots, 255\}^{16}$	m

The specifications of the decryption procedure of AES256 are available in [11].

6 Symmetric Encryption

Symmetric keys used for symmetric encryption are instances of a complex type, denoted SymEncKey, which is a subtype of SymmetricKey (see Section 3.2.1). The type SymEncKey is an *abstract* type. A SymEncKey instance will always be an instance of a *concrete* subtype of SymEncKey. The following initializer will systematically be called by the initializer of subtypes of SymEncKey.

SymEncKey (Initializer)		
parameters		None
in	$\{0,\dots,255\}$	`algoImplemByteId`
	Dictionary	`dict`

We denote by

$$\mathsf{SymEncKey}(\texttt{algoImplemByteId}, \texttt{dict}) \rightarrow \mathsf{symEncKey}$$

the call to the SymEncKey initializer.

1: **procedure** SymEncKey(`algoImplemByteId`, `dict`)
2: `algoClassByteId` ← `0x00`
3: SymmetricKey(`algoClassByteId`, `algoImplemByteId`, `dict`)
4: **end procedure**

SymEnc.encrypt		
parameters		$\ell_{iv}, \ell_n \in \mathbb{N}$
in	$\{0,\dots,255\}^*$	`m`
	SymEncKey	symEncKey
	$\{0,\dots,255\}^{\ell_{iv}+\ell_n}$	`iv` ‖ `n`
out	$\{0,\dots,255\}^*$	`c`

We denote by

$$\mathsf{SymEnc.encrypt}(\texttt{m}, \mathsf{symEncKey}, \texttt{iv} \,\|\, \texttt{n}) \rightarrow \mathsf{c}$$

the symmetric encryption with the symmetric encryption algorithm SymEnc of the message `m` $\in \{0,\dots,255\}^*$ under the key symEncKey $\in$ SymEncKey, initialization vector `iv` $\in \{0,\dots,255\}^{\ell_{iv}}$, and nonce `n` $\in \{0,\dots,255\}^{\ell_n}$. An initialization vector `iv` should be unpredictable. A nonce `n` does not have to be unpredictable, but should never be used twice with the same key. If the nonce size is small (e.g., 64 bits), taking a nonce at random is not enough. In that case, the nonce can be kept in memory and incremented each time a new message is sent. Both `iv` and `n` are transmitted in clear. This method is abstract and only implemented by concrete subtypes of SymEnc.

SymEnc.decrypt		
parameters		$\ell_{iv}, \ell_n \in \mathbb{N}$
in	$\{0,\dots,255\}^*$ SymEncKey	c symEncKey
out	$\{0,\dots,255\}^*$	m

We denote by

$$\mathsf{SymEnc.decrypt}(\mathsf{c}, \mathsf{symEncKey}) \to \mathtt{m}$$

the symmetric decryption with the symmetric encryption algorithm SymEnc of the ciphertext $\mathsf{c} \in \{0,\dots,255\}^*$ under the key $\mathsf{symEncKey} \in \mathsf{SymEncKey}$. This method is abstract and only implemented by concrete subtypes of SymEnc.

SymEnc.ciphertextLength		
parameters		None
in	$\mathbb{N}$	ℓ_m
out	$\mathbb{N}$	ℓ_c

We denote by

$$\mathsf{SymEnc.ciphertextLength}(\ell_m) \to \ell_c$$

the static procedure that returns the final length of the ciphertext corresponding to a plaintext of byte-length ℓ_m if encrypted with the symmetric encryption algorithm SymEnc. This method is abstract and only implemented by concrete subtypes of SymEnc.

SymEnc.plaintextLength		
parameters		None
in	$\mathbb{N}$	ℓ_c
out	$\mathbb{N}$	ℓ_m

We denote by

$$\mathsf{SymEnc.plaintextLength}(\ell_c) \to \ell_m$$

the static procedure that returns the plaintext length corresponding to a ciphertext of byte-length ℓ_c if decrypted with the symmetric encryption algorithm SymEnc. This method is abstract and only implemented by concrete subtypes of SymEnc.

6.1 AES-256 in CTR Mode

We denote by AES256CTR the concrete subtype of SymEnc that encrypts using AES-256 in CTR mode. We denote by AES256CTRKey the concrete subtype of SymEncKey for AES256CTR keys. The following initializer creates a AES256CTRKey.

AES256CTRKey (Initializer)		
parameters		None
in	Dictionary	`dict`

We denote by

$$\text{AES256CTRKey}(\texttt{dict}) \to \text{symEncKey}$$

the call to the AES256CTRKey initializer.

```
1: procedure AES256CTRKey(dict)
2:     encoded_raw ← dict["enckey"]
3:     raw ← decodeBytes(encoded_raw)
4:     if len(raw) ≠ 32 then return ⊥ end if
5:     algoImplemByteId ← 0x00
6:     SymEncKey(algoImplemByteId, dict)
7: end procedure
```

AES256CTRKey (Initializer)		
parameters		None
in	$\{0,\dots,255\}^*$	`b`

```
1: procedure AES256CTRKey(b)
2:     encoded_raw ← encodeBytes(b)
3:     dict["enckey"] ← encoded_raw
4:     AES256CTRKey(dict)
5: end procedure
```

AES256CTR.encrypt		
parameters		$\ell_{iv} = 0$, $\ell_n = 8$

in	$\{0,\dots,255\}^*$
	AES256CTRKey
	$\{0,\dots,255\}^8$
		`m`
		symEncKey
		`n`

out	$\{0,\dots,255\}^*$	`c`

```
1: procedure AES256CTR.encrypt(m, symEncKey, n)
2:     encoded_raw ← symEncKey.dict["enckey"]
3:     k ← decodeBytes(encoded_raw)
4:     ℓ ← length(m)
```

enc ← AES256.encrypt
let $\mathtt{s} \in \{0,\dots,255\}^{\ell}$ be the first ℓ bytes of $\mathsf{enc}(\mathtt{n}\|\underline{0},\mathtt{k}) \,\|\, \mathsf{enc}(\mathtt{n}\|\underline{1},\mathtt{k}) \,\|\, \mathsf{enc}(\mathtt{n}\|\underline{2},\mathtt{k}) \,\|\, \cdots$
return $\mathtt{n}\|(\mathtt{m} \oplus \mathtt{s}) \in \{0,\dots,255\}^{8+\ell}$
end procedure

In the previous algorithm, $\underline{x}$ denotes the big-endian representation of the integer x as an array of bytes. It can be obtained with bytesFromBigUInt$(x, 8)$.

AES256CTRKey.decrypt		
parameters		$\ell_{iv} = 0$, $\ell_n = 8$
in	$\{0,\dots,255\}^*$ AES256CTRKey	c symEncKey
out	$\{0,\dots,255\}^*$	m

procedure AES256CTR.decrypt(c, symEncKey)
encoded_raw ← symEncKey.dict["enckey"]
k ← decodeBytes(encoded_raw)
if len(c) < 8 **then** return $\perp$ **end if**
let $\ell = \mathsf{len}(\mathtt{c}) - 8$ and parse c as $(\mathtt{n}, \mathtt{c0}) \in \{0,\dots,255\}^8 \times \{0,\dots,255\}^{\ell}$
enc ← AES256.encrypt
let $\mathtt{s} \in \{0,\dots,255\}^{\ell}$ be the first ℓ bytes of $\mathsf{enc}(\mathtt{n}\|\underline{0},\mathtt{k}) \,\|\, \mathsf{enc}(\mathtt{n}\|\underline{1},\mathtt{k}) \,\|\, \mathsf{enc}(\mathtt{n}\|\underline{2},\mathtt{k}) \,\|\, \cdots$
return $\mathtt{c0} \oplus \mathtt{s}$
end procedure

AES256CTR.ciphertextLength		
parameters		None
in	$\mathbb{N}$	ℓ_m
out	$\mathbb{N}$	ℓ_c

procedure AES256CTR.ciphertextLength(ℓ_m)
return $8 + \ell_m$
end procedure

AES256CTR.plaintextLength		
parameters		None
in	$\mathbb{N}$	ℓ_c
out	$\mathbb{N}$	ℓ_m

1: **procedure** AES256CTR.plaintextLength(ℓ_c)
2: **if** $\ell_c < 8$ **then** return $\perp$ **end if**
3: return $\ell_c - 8$
4: **end procedure**

7 Message Authentication Code

MAC is one of the symmetric keyed primitives exposed by the cryptographic library. As such, symmetric keys are instances of a complex type, denoted MACKey, which is a subtype of SymmetricKey (see Section 3.2.1). The type MACKey is an *abstract* type. A MACKey instance will always be an instance of a *concrete* subtype of MACKey. The following initializer will systematically be called by the initializer of subtypes of MACKey.

MACKey (Initializer)		
parameters		None
in	$\{0,\dots,255\}$ Dictionary	`algoImplemByteId` `dict`

We denote by

$$\mathsf{MACKey}(\texttt{algoImplemByteId}, \texttt{dict}) \to \mathsf{macKey}$$

the call to the MACKey initializer.

1: **procedure** MACKey(`algoImplemByteId`, `dict`)
2: `algoClassByteId` $\leftarrow$ `0x01`
3: SymmetricKey(`algoClassByteId`, `algoImplemByteId`, `dict`)
4: **end procedure**

MAC.compute		
parameters		$\ell_t \in \mathbb{N}$
in	$\{0,\dots,255\}^*$ MACKey	`m` macKey
out	$\{0,\dots,255\}^{\ell_t}$	`t`

The call

$$\mathsf{MAC.compute}(\texttt{m}, \mathsf{macKey}) \to \texttt{t}$$

computes the MAC of the plaintext `m` under the key macKey. This method is abstract and only implemented by concrete subtypes of MAC.

MAC.finalOutputSize		
parameters		None
in	None	None
out	$\mathbb{N}$	ℓ_t

We denote by

$$\mathsf{MAC.finalOutputSize}() \to \ell_t$$

the static procedure that returns the final length of a MAC digest. This method is abstract and only implemented by concrete subtypes of MAC.

7.1 HMAC with SHA-256

We denote by HMACWithSHA256 the concrete subtype of MAC that computes a MAC based on HMAC with SHA-256. We denote by HMACWithSHA256Key the concrete subtype of MACKey for HMACWithSHA256 keys. The following initializer creates a HMACWithSHA256Key key.

HMACWithSHA256Key (Initializer)		
parameters		None
in	Dictionary	`dict`

We denote by

$$\mathsf{HMACWithSHA256Key}(\texttt{dict}) \to \mathsf{hmacKey}$$

the call to the HMACWithSHA256Key initializer.

```
1: procedure HMACWithSHA256Key(dict)
2:     encoded_raw ← dict["mackey"]
3:     raw ← decodeBytes(encoded_raw)
4:     if len(raw) < 32 then return ⊥ end if
5:     algoImplemByteId ← 0x00
6:     MACKey(algoImplemByteId, dict)
7: end procedure
```

HMACWithSHA256Key (Initializer)		
parameters		None
in	$\{0,\ldots,255\}^*$	`b`

```
1: procedure HMACWithSHA256Key(b)
2:     encoded_raw ← encodeBytes(b)
```

```
3:     dict["mackey"] ← encoded_raw
4:     HMACWithSHA256Key(dict)
5: end procedure
```

HMACWithSHA256.generateKey		
parameters		None
in	$\{0,\ldots,255\}^*$	seed
out	HMACWithSHA256Key	hmacKey

```
1: procedure HMACWithSHA256.generateKey(seed)
2:     kdf ← KDFFromPRNGWithHMACWithSHA256
3:     return kdf.compute(seed, HMACWithSHA256Key)
4: end procedure
```

HMACWithSHA256.generateKey		
parameters		None
in	PRNG	prng
out	HMACWithSHA256Key	hmacKey

```
1: procedure HMACWithSHA256.generateKey(prng)
2:     seed ← prng.bytes(32)
3:     return HMACWithSHA256.generateKey(seed)
4: end procedure
```

HMACWithSHA256.compute		
parameters		None
in	$\{0,\ldots,255\}^*$ MACKey	m macKey
out	$\{0,\ldots,255\}^{\ell_t}$	t

The implementation of the compute procedure for HMACWithSHA256 is the following:

```
1: procedure HMACWithSHA256.compute(m, macKey)
2:     if macKey is not a HMACWithSHA256Key then return ⊥ end if
3:     encoded_raw ← macKey.dict["mackey"]
```

```
    k ← decodeBytes(encoded_raw)
    opad ← (92, 92, ..., 92) ∈ {0, ..., 255}^64
    ipad ← (54, 54, ..., 54) ∈ {0, ..., 255}^64
    k_o = (k||0 ··· 0) ⊕ opad ∈ {0, ..., 255}^64
    k_i = (k||0 ··· 0) ⊕ ipad ∈ {0, ..., 255}^64
    return SHA256(k_o || SHA256(k_i || m))
end procedure
```

HMACWithSHA256.finalOutputSize		
parameters		None
in	None	None
out	$\mathbb{N}$	32

The implementation of the finalOutputSize procedure for HMACWithSHA256 is the following:

```
procedure HMACWithSHA256.finalOutputSize()
    return 32
end procedure
```

8 Key Derivation Function (KDF)

A Key Derivation Function (KDF) creates a SymmetricKey instance in a deterministic way from an input seed.

KDF.compute		
parameters		None
in	$\{0, \ldots, 255\}^*$ SymmetricKey subtype	seed T
out	T	k

In the previous definition, init is a deterministic procedure that takes an array of bytes as an input and outputs an instance of T (which is a concrete subtype of SymmetricKey).

The call

$$\mathsf{KDF}(\mathtt{seed}, \mathsf{T}) \to \mathsf{k}$$

computes a symmetric key k of type T from a seed seed. This method is abstract and only implemented by concrete subtypes of KDF.

8.1 KDF Based on SHA-256

We denote by KDFFromPRNGWithHMACWithSHA256 the concrete subtype of KDF that computes symmetric keys from a seed using the PRNG defined in Section 10.1.

KDFFromPRNGWithHMACWithSHA256.compute		
parameters		None
in	$\{0,\dots,255\}^*$ SymmetricKey subtype	`seed` T
out	T	k

```
1: procedure KDFFromPRNGWithHMACWithSHA256.compute(seed, T)
2:     prng = PRNGWithHMACWithSHA256(seed)
3:     b ← prng.bytes(T.length)
4:     return T(b)
5: end procedure
```

Note that this procedure fails in case the initialization of the PRNG instance fails, which happens when the seed `seed` is not long enough. See Section 10.

9 Commitment

Commitment schemes within these specifications are subtypes of Commitment.

Commitment.commit		
parameters		

in	$\{0,\dots,255\}^*$
	$\{0,\dots,255\}^*$
	PRNG
		`tag`
		`value`
		prng

out	$\{0,\dots,255\}^*$ $\{0,\dots,255\}^*$	`commitment` `decommitToken`

We denote by

$$\text{Commitment.commit}(\texttt{tag}, \texttt{value}, \text{prng}) \to (\texttt{commitment}, \texttt{decommitToken})$$

the commitment on a tag `tag` and value `value` using the PRNG instance prng. The result is a `commitment` and a `decommitToken` that can be used to open the commitment. This method is abstract and only implemented by concrete subtypes of Commitment.

Commitment.open		
parameters		
in	$\{0,\dots,255\}^*$	`commitment`
	$\{0,\dots,255\}^*$	`tag`
	$\{0,\dots,255\}^*$	`decommitToken`
out	$\{0,\dots,255\}^*$	`value`

We denote by

$$\text{Commitment.open}(\texttt{commitment}, \texttt{tag}, \texttt{decommitToken}) \to \texttt{value}$$

the procedure that opens a `commitment` and `tag` using the `decommitToken`, which recovers the `value` that was committed. This method is abstract and only implemented by concrete subtypes of Commitment.

9.1 Commitment based on SHA-256

We denote by CommitmentWithSHA256 the subtype of Commitment that implements the extractable random oracle commitment described in [16, p.51] using SHA-256.

CommitmentWithSHA256.commit		
parameters		
in	$\{0,\dots,255\}^*$	`tag`
	$\{0,\dots,255\}^*$	`value`
	PRNG	prng
out	$\{0,\dots,255\}^*$	`commitment`
	$\{0,\dots,255\}^*$	`decommitToken`

We denote by

$$\text{CommitmentWithSHA256.commit}(\texttt{tag}, \texttt{value}, \text{prng}) \to (\texttt{commitment}, \texttt{decommitToken})$$

the procedure that computes a commitment on a tag `tag` and value `value` using the PRNG instance prng. The result is a `commitment` and a `decommitToken` that can be used to open the commitment.

```
1: procedure CommitmentWithSHA256.commit(tag, value, prng)
2:     e ← prng.bytes(32)
3:     d ← value ‖ e
4:     commitment ← SHA256(tag ‖ d)
5:     return (commitment, d)
6: end procedure
```

CommitmentWithSHA256.open		
parameters		

in	$\{0,\dots,255\}^*$
	$\{0,\dots,255\}^*$
	$\{0,\dots,255\}^*$
		`commitment`
		`tag`
		`decommitToken`

out	$\{0,\dots,255\}^*$	`value`

We denote by

$$\mathsf{CommitmentWithSHA256.open}(\texttt{commitment}, \texttt{tag}, \texttt{decommitToken}) \rightarrow \texttt{value}$$

the procedure that opens a `commitment` and `tag` using the `decommitToken`, which recovers the `value` that was committed.

1: **procedure** CommitmentWithSHA256.open(`commitment`, `tag`, `decommitToken`)
2: `computedCommitment` $\leftarrow$ SHA256(`tag` $\|$ `decommitToken`)
3: **if** `computedCommitment` $\neq$ `commitment` **then** return $\perp$ **end if**
4: Parse `decommitToken` as `value` $\|$ `e` where len(`e`) $= 32$
5: return `value`
6: **end procedure**

10 Pseudorandom Generator

Most pseudorandom generators (PRNGs) need to keep track of an internal state between successive calls. This makes PRNGs quite different from the other primitives defined in this document. For this reason, we use slightly different notations for PRNGs than for, e.g., hash functions or block ciphers.

We denote by PRNG the abstract type common to all PRNGs. Letting prng be an instance of a PRNG, we denote by

`prng.state`

the internal state of this instance.

PRNG (Initializer)		
parameters		None
in	$\{0,\dots,255\}^*$	`seed`

We denote by

PRNG(`seed`)

the procedure that initializes a fresh instance prng of type PRNG. Note that, in this document, we always denote a PRNG instance by prng. This method is abstract and only implemented by concrete subtypes of PRNG.

The minimum seed size is determined by the concrete subtype of PRNG (which assumes that a `seed` of ℓ bytes contains 8ℓ bits of entropy). If the `seed` is not long enough, the procedure shall fail and the PRNG instance is not initialized.

prng.bytes		
parameters		None
in	$\mathbb{N}$	$\ell = 32$
out	$\{0,\ldots,255\}^*$	`r`

We denote by

$$\text{prng.bytes}(\ell) \rightarrow \texttt{r}$$

the call to the initialized PRNG instance prng that generates a uniformly distributed pseudorandom byte string `r` of ℓ bytes. This procedure updates the internal state of the PRNG instance prng. This method is abstract and only implemented by concrete subtypes of PRNG.

prng.bigInt		
parameters		None
in	$\mathbb{N}$	n
out	$\mathbb{N}$	a

We denote by

$$\text{prng.bigInt}(n) \rightarrow a$$

the call to the initialized PRNG instance prng that generates a uniformly distributed pseudorandom big integer $a \in [0, n-1]$. This procedure updates the internal state of the PRNG instance prng. We define the bigInt procedure using the procedure bytes:

procedure prng.bigInt(n)
  Let l be the smallest integer such that $n \leq 2^l$ and $\ell = \lceil l/8 \rceil$
  Let m be the smallest integer in $\{1, 3, 7, 15, 31, 63, 127, 255\}$ s.t. $m \geq \lfloor (n-1)/256^{\ell-1} \rfloor$
  while `True` **do**
    $s =$ prng.bytes(ℓ)
    $s[0] = s[0]$ and m
    $r = \sum_{i=0}^{\ell-1} s[\ell - 1 - i] \cdot 256^i$
    If $r < n$, return r
  end while
end procedure

10.1 FIPS 800-90A HMAC_DRBG Based on SHA-256

We denote by PRNGWithHMACWithSHA256 the concrete subtype of PRNG that implements the FIPS 800-90A HMAC_DRBG algorithm described in [3], using SHA-256 as the hash function. In this section, $\ell_h = 32$ denotes the byte output length of SHA-256. The internal state prng.state of an initialized instance prng of type PRNGWithHMACWithSHA256 gives access to two variables

- prng.state.k $\in \{0,\dots,255\}^{32}$
- prng.state.v $\in \{0,\dots,255\}^{32}$

We first define an additional procedure, called update, which takes some auxiliary data $\in \{0,\dots,255\}^*$ in order to update the internal state. This procedure works as follows:

```
1: procedure prng.update(data)
2:     k ← self.state.k
3:     v ← self.state.v
4:     k = HMACWithSHA256.compute(v || 0x00 || data, HMACWithSHA256Key(k))
5:     v = HMACWithSHA256.compute(v, HMACWithSHA256Key(k))
6:     if length(data) > 0 then
7:         k = HMACWithSHA256.compute(v || 0x01 || data, HMACWithSHA256Key(k))
8:         v = HMACWithSHA256.compute(v, HMACWithSHA256Key(k))
9:     end if
10:    self.state.k ← k
11:    self.state.v ← v
12: end procedure
```

PRNGWithHMACWithSHA256 (Initializer)		
parameters		None
in	$\{0,\dots,255\}^*$	seed

The initialization procedure expects a seed of at least $\ell_h = 32$ bytes. Note that although the FIPS 800-90A standard does not enforce the seed size, we choose to do so. Note also that the input seed that we use here corresponds to the concatenation of the parameters *entropy_input*, *nonce*, and *personalization_string* in [3, Sec. 10.1.2.3], and that we omit the *reseed_counter*. The procedure works as follows:

```
1: procedure PRNGWithHMACWithSHA256(seed)
2:     if len(seed) < 32 then return ⊥ end if
3:     self.state.k = (0, 0, ..., 0) ∈ {0,...,255}^32
4:     self.state.v = (1, 1, ..., 1) ∈ {0,...,255}^32
5:     self.update(seed)
6: end procedure
```

prng.bytes		
parameters		None
in	$\mathbb{N}$	$\ell = 32$
out	$\{0,\ldots,255\}^*$	`r`

When prng is an instance of PRNGWithHMACWithSHA256, the bytes procedure works as follows (note that we omit the *additional_input* of [3, Sec. 10.1.2.5]):

```
1: procedure prng.bytes(ℓ)
2:     k ← self.state.k
3:     v ← self.state.v
4:     hmacKey ← HMACWithSHA256Key(k)
5:     s ← [ ]
6:     while len(s) < ℓ do
7:         v ← HMACWithSHA256.compute(v, hmacKey)
8:         s ← s ‖ v
9:     end while
10:    self.state.v ← v
11:    self.update([ ])
12:    truncate s to its first ℓ bytes
13:    return s
14: end procedure
```

Periodic refresh. Besides the instances of PRNGWithHMACWithSHA256 that are explicitly seeded (for deterministic derivations), the engine uses a single shared instance of PRNGWithHMACWithSHA256 as its source of randomness. This shared instance is regularly refreshed with fresh entropy: every 100 calls to bytes, 32 fresh random bytes e are obtained from the operating system's random number generator and injected into the state by calling self.update(e) (which corresponds to the *reseed* procedure of [3, Sec. 10.1.2.4]).

11 Authenticated Encryption

Authenticated Encryption is one of the symmetric keyed primitives exposed by the cryptographic library. As such, symmetric keys are instances of a complex type, denoted AuthEncKey, which is a subtype of SymmetricKey (see Section 3.2.1). The type AuthEncKey is an *abstract* type. A AuthEncKey instance will always be an instance of a *concrete* subtype of AuthEncKey. The following initializer will systematically be called by the initializer of subtypes of AuthEncKey.

AuthEncKey (Initializer)		
parameters		None
in	$\{0,\ldots,255\}$ Dictionary	`algoImplemByteId` `dict`

We denote by

$$\mathsf{AuthEncKey}(\texttt{algoImplemByteId}, \texttt{dict}) \rightarrow \mathsf{authEncKey}$$

the call to the AuthEncKey initializer.

```
1: procedure AuthEncKey(algoImplemByteId, dict)
2:     algoClassByteId ← 0x02
3:     SymmetricKey(algoClassByteId, algoImplemByteId, dict)
4: end procedure
```

AuthEnc.encrypt		
parameters		None
in	$\{0,\dots,255\}^*$ AuthEncKey PRNG	m authEncKey prng
out	$\{0,\dots,255\}^*$	c

The call

$$\mathsf{AuthEnc.encrypt}(\texttt{m}, \mathsf{authEncKey}, \mathsf{prng}) \rightarrow \mathsf{c}$$

encrypts the plaintext m under the key authEncKey using the PRNG instance prng. This method is abstract and only implemented by concrete subtypes of AuthEnc.

AuthEnc.decrypt		
parameters		None
in	$\{0,\dots,255\}^*$ AuthEncKey	c authEncKey
out	$\{0,\dots,255\}^*$	m

The call

$$\mathsf{AuthEnc.decrypt}(\mathsf{c}, \mathsf{authEncKey}) \rightarrow \texttt{m}$$

decrypts the ciphertext c under the key authEncKey. This method is abstract and only implemented by concrete subtypes of AuthEnc.

AuthEnc.ciphertextLength		
parameters		None
in	$\mathbb{N}$	ℓ_m
out	$\mathbb{N}$	ℓ_c

We denote by

$$\mathsf{AuthEnc.ciphertextLength}(\ell_m) \to \ell_c$$

the static procedure that returns the final length of the ciphertext corresponding to a plaintext of byte-length ℓ_m. This method is abstract and only implemented by concrete subtypes of AuthEnc.

AuthEnc.plaintextLength		
parameters		None
in	$\mathbb{N}$	ℓ_c
out	$\mathbb{N}$	ℓ_m

We denote by

$$\mathsf{AuthEnc.plaintextLength}(\ell_c) \to \ell_m$$

the static procedure that returns the final length of the plaintext corresponding to a ciphertext of byte-length ℓ_c. This method is abstract and only implemented by concrete subtypes of AuthEnc.

11.1 Encrypt-then-MAC with AES-256 and HMAC with SHA-256

We denote by AES256CTRHMACSHA256 the concrete subtype of AuthEnc that implements Encrypt-then-MAC with AES-256 and HMAC with SHA-256. We denote by AES256CTRHMACSHA256Key the concrete subtype of AuthEncKey for AES256CTRHMACSHA256 keys. The following initializer creates a AES256CTRHMACSHA256Key key.

AES256CTRHMACSHA256Key (Initializer)		
parameters		None
in	Dictionary	`dict`

We denote by

$$\mathsf{AES256CTRHMACSHA256Key}(\texttt{dict}) \to \mathsf{authEncKey}$$

the call to the AES256CTRHMACSHA256Key initializer.

```
1: procedure AES256CTRHMACSHA256Key(dict)
2:     self.ke ← AES256CTRKey(dict)
3:     self.ka ← HMACWithSHA256Key(dict)
4:     algoImplemByteId ← 0x00
5:     AuthEncKey(algoImplemByteId, dict)
6: end procedure
```

AES256CTRHMACSHA256Key (Initializer)		
parameters		None
in	$\{0,\ldots,255\}^*$	`b`

1: **procedure** AES256CTRHMACSHA256Key(b)
2: **if** length(b) $\neq 64$ **then** return $\perp$ **end if**
3: let b_1 denote the first 32 bytes of b
4: let b_2 denote the last 32 bytes of b
5: dict["mackey"] $\leftarrow$ encodeBytes(b_1)
6: dict["enckey"] $\leftarrow$ encodeBytes(b_2)
7: AES256CTRHMACSHA256Key(dict)
8: **end procedure**

AES256CTRHMACSHA256.generateKey		
parameters		None
in	$\{0,\ldots,255\}^*$	seed
out	AES256CTRHMACSHA256Key	authEncKey

1: **procedure** AES256CTRHMACSHA256.generateKey(seed)
2: kdf $\leftarrow$ KDFFromPRNGWithHMACWithSHA256
3: return kdf.compute(seed, AES256CTRHMACSHA256Key)
4: **end procedure**

The encryption procedure first encrypts the plaintext with AES-256 in CTR mode using a random 8-byte nonce, then computes an HMAC with SHA-256 of the resulting ciphertext (including the nonce):

1: **procedure** AES256CTRHMACSHA256.encrypt(m, authEncKey, prng)
2: n $\leftarrow$ prng.bytes(8)
3: c_0 $\leftarrow$ AES256CTR.encrypt(m, authEncKey.ke, n) // $8 +$ len(m) bytes
4: t $\leftarrow$ HMACWithSHA256.compute(c_0, authEncKey.ka) // 32 bytes
5: return $c_0 \,\|\, t$
6: **end procedure**

The decryption procedure checks the MAC before performing any decryption. The comparison of the MAC values is performed in constant time.

1: **procedure** AES256CTRHMACSHA256.decrypt(c, authEncKey)
2: **if** len(c) < 40 **then** return $\perp$ **end if**
3: Parse c as $c_0 \,\|\, t$ where len(t) $= 32$
4: **if** HMACWithSHA256.compute(c_0, authEncKey.ka) $\neq$ t **then** return $\perp$ **end if**

5: return AES256CTR.decrypt(c_0, authEncKey.ke)
6: **end procedure**

The ciphertext is thus always 40 bytes longer than the plaintext (8 bytes of nonce and 32 bytes of MAC):

1: **procedure** AES256CTRHMACSHA256.ciphertextLength(ℓ_m)
2: return $\ell_m + 40$
3: **end procedure**
4: **procedure** AES256CTRHMACSHA256.plaintextLength(ℓ_c)
5: **if** $\ell_c < 40$ **then** return $\perp$ **end if**
6: return $\ell_c - 40$
7: **end procedure**

12 Edwards Curves

All the elliptic curves we consider within these specifications are Edwards curves [12] and formalized as an instance of a complex type denoted EdwardsCurve. The type EdwardsCurve is an *abstract* type. An EdwardsCurve instance will always be an instance of a *concrete* subtype of EdwardsCurve. Background on elliptic curves is available in Appendix A. An instance curve of EdwardsCurve provides the following parameters:

- curve.p: the prime order of the underlying finite field $\mathbf{F}_p$,
- curve.d: the parameter defining the Edwards curve over $\mathbf{F}_p$,
- curve.$G = (G_x, G_y)$: the base point explicitly defined by the curve,
- curve.q: the prime order of the subgroup generated by G,
- curve.ν: the lcm of the cofactor $\#E(\mathbf{F}_p)/q$ of the curve,
- curve.card: the cardinality of the curve $q \times \nu$.

For simplicity, a curve can also return all parameters at once:

$$\text{curve.parameters} \rightarrow (p, d, G, q, \nu, \text{card})$$

curve.isOnCurve		
parameters		None
in	$\mathbb{N} \times \mathbb{N}$	(x, y)
out	{`True`, `False`}	`bool`

We let

$$\text{curve.isOnCurve}((x, y)) \rightarrow \texttt{bool}$$

be the procedure that checks whether a point (x, y) is on the curve instance curve. The procedure works as follows:

1: **procedure** curve.isOnCurve((x, y))

2: $(p, d, G, q, \nu, \mathsf{card}) \leftarrow \mathsf{curve.parameters}$
3: $x2 \leftarrow x^2 \bmod p$
4: $y2 \leftarrow y^2 \bmod p$
5: return $x2 + y2 \bmod p = 1 + dx2y2 \bmod p$
6: **end procedure**

curve.xCoordinatesFromY		
parameters		None
in	$\mathbb{N}$	y
out	$\mathbb{N} \times \mathbb{N}$	(x_1, x_2)

We let

$$\mathsf{curve.xCoordinatesFromY}(y)$$

be the procedure that returns the two possible x coordinates of a point on the curve instance curve, when they exist, where the point is specified using its y coordinate only. In the following procedure, note that:

- since d is assumed not to be a square, $1 - dy^2$ is invertible modulo p;
- after step 12 we have $p - 1 = 2^s t$ with t odd.

1: **procedure** $\mathsf{curve.xCoordinatesFromY}(y)$
2: $(p, d, G, q, \nu, \mathsf{card}) \leftarrow \mathsf{curve.parameters}$
3: $y_2 \leftarrow y^2 \bmod p$
4: $x_2 \leftarrow (1 - y_2)(1 - dy_2)^{-1} \bmod p$
5: **if** $x_2^{\frac{p-1}{2}} \bmod p \neq 1$ **then** return $\perp$ **end if**
6: **if** $p \bmod 4 = 3$ **then**
7: $x \leftarrow x_2^{\frac{p+1}{4}} \bmod p$
8: **else**
9: $g \leftarrow 1$
10: **repeat** $g \leftarrow g + 1$ **until** $g^{\frac{p-1}{2}} \bmod p \neq 1$
11: $t \leftarrow p - 1$ and $s \leftarrow 0$
12: **repeat** $t \leftarrow t/2$ and $s \leftarrow s + 1$ **until** $t \bmod 2 \neq 0$
13: $e \leftarrow 0$
14: **for** $i \leftarrow 2$ to s **do**
15: **if** $(x_2 g^{-e})^{\frac{p-1}{2^i}} \bmod p \neq 1$ **then** $e \leftarrow 2^{i-1} + e$ **end if**
16: **end for**
17: $x \leftarrow g^{-t\frac{e}{2}} x_2^{\frac{t+1}{2}} \bmod p$
18: **end if**
19: return $(x, -x \bmod p)$
20: **end procedure**

curve.scalarMultiplication		
parameters		$q, p \in \mathbb{N}$ fixed by curve.
in	$[0, 1, \ldots, q-1]$	n
	$[0, 1, \ldots, p-1]$	y
out	$[0, 1, \ldots, p-1]$	y_n

We know from the discussion of Section A.2.1 that it is possible to perform the scalar multiplication of a point $P = (x, y)$ by n, on an Edwards curve defined by the parameter $d \in \mathbf{F}_p$ (which must be a non-square in $\mathbf{F}_p$), using y-coordinate only computations. We denote by

$$\mathsf{curve.scalarMultiplication}(n, y) \to y_n$$

the call to the procedure that, given a scalar n, a point P of y-coordinate y, and the curve instance curve, returns the y-coordinate y_n of the point $nP \in E$.

Denoting by $n = (n_{\ell-1} n_{\ell-2} \ldots n_1 n_0)$ the binary representation of n (where ℓ is the bit length of curve.card), the following procedure returns the y-coordinate y_n of the point $nP \in E$ (note that in practice, one should precompute c and perform the main loop more efficiently to reduce the number of field squarings and multiplications):

1: **procedure** curve.scalarMultiplication(n, y)
2: $(p, d, G, q, \nu, \mathsf{card}) \leftarrow$ curve.parameters
3: **if** $n = 0$ or $y = 1$ **then** return 1 **end if**
4: **if** $y = -1$ **then** return $1 - 2 \times (n \bmod 2)$ **end if**
5: $c \leftarrow (1-d)^{-1} \bmod p$
6: $u_P \leftarrow (1+y) \bmod p$ and $w_P \leftarrow (1-y) \bmod p$
7: $u_Q \leftarrow 1$, $w_Q \leftarrow 0$, $u_R \leftarrow u_P$, and $w_R \leftarrow w_P$
8: $n \leftarrow n \bmod \mathsf{card}$
9: **for** $i \leftarrow \ell$ down to 1 **do**
10: $t_1 \leftarrow (u_Q - w_Q)(u_R + w_R) \bmod p$
11: $t_2 \leftarrow (u_Q + w_Q)(u_R - w_R) \bmod p$
12: $u_{Q+R} \leftarrow w_P (t_1 + t_2)^2 \bmod p$
13: $w_{Q+R} \leftarrow u_P (t_1 - t_2)^2 \bmod p$
14: **if** $n_{i-1} = 0$ **then**
15: $t_3 \leftarrow (u_Q + w_Q)^2 \bmod p$
16: $t_4 \leftarrow (u_Q - w_Q)^2 \bmod p$
17: $t_5 \leftarrow t_3 - t_4 \bmod p$
18: $u_{2Q} \leftarrow t_3 t_4 \bmod p$
19: $w_{2Q} \leftarrow t_5 (t_4 + c t_5) \bmod p$
20: $(u_Q, w_Q) \leftarrow (u_{2Q}, w_{2Q}) \bmod p$ and $(u_R, w_R) \leftarrow (u_{Q+R}, w_{Q+R}) \bmod p$
21: **else**
22: $t_3 \leftarrow (u_R + w_R)^2 \bmod p$
23: $t_4 \leftarrow (u_R - w_R)^2 \bmod p$
24: $t_5 \leftarrow t_3 - t_4 \bmod p$
25: $u_{2R} \leftarrow t_3 t_4 \bmod p$
26: $w_{2R} \leftarrow t_5 (t_4 + c t_5) \bmod p$
27: $(u_Q, w_Q) \leftarrow (u_{Q+R}, w_{Q+R}) \bmod p$ and $(u_R, w_R) \leftarrow (u_{2R}, w_{2R}) \bmod p$

28: **end if**
29: **end for**
30: return $(u_Q - w_Q)(u_Q + w_Q)^{-1} \bmod p$
31: **end procedure**

curve.pointAddition		
parameters		$q, p \in \mathbb{N}$ fixed by curve.
in	$[0, 1, \ldots, p-1]^2$	(x_1, y_1)
	$[0, 1, \ldots, p-1]^2$	(x_2, y_2)
out	$[0, 1, \ldots, p-1]^2$	(x, y)

We denote by

$$\text{curve.pointAddition}((x_1, y_1), (x_2, y_2))$$

the call to the procedure that, given two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, and the curve instance curve, returns the coordinates of the point $P = P_1 + P_2$ on the curve. Note that this procedure does *not* check whether the points given as arguments are on the curve. This check has to be made before this function is called. The procedure works as follows:

1: **procedure** curve.pointAddition$((x_1, y_1), (x_2, y_2))$
2: $(p, d, G, q, \nu, \text{card}) \leftarrow$ curve.parameters
3: $t \leftarrow dx_1x_2y_1y_2 \bmod p$
4: $z \leftarrow (1 + t)^{-1} \bmod p$
5: $x \leftarrow z(x_1y_2 + y_1x_2) \bmod p$
6: $z \leftarrow (1 - t)^{-1} \bmod p$
7: $y \leftarrow z(y_1y_2 - x_1x_2) \bmod p$
8: return (x, y)
9: **end procedure**

curve.scalarMultiplicationWithX		
parameters		$q, p \in \mathbb{N}$ fixed by curve.
in	$[0, 1, \ldots, q-1]$	n
	$[0, 1, \ldots, p-1]^2$	(x, y)
out	$[0, 1, \ldots, p-1]^2$	(x_n, y_n)

We denote by

$$\text{curve.scalarMultiplicationWithX}(n, (x, y))$$

the call to the procedure that, given a scalar n, a point $P = (x, y)$, and the curve instance curve, returns both coordinates of the point $nP \in E$. This procedure fails if P is not on the curve defined by curve.

Denoting by $n = (n_{\ell-1}n_{\ell-2}\dots n_1 n_0)$ the binary representation of n (where ℓ is the bit length of curve.card), the following procedure (based on the Montgomery ladder, see Section A.2.2) returns both coordinates of nP on the curve:

```
1: procedure scalarMultiplicationWithX(n, (x, y), curve)
2:     if not curve.isOnCurve((x, y)) then return ⊥ end if
3:     if n = 0 or y = 1 then return (0, 1) end if
4:     if y = −1 then return (0, 1 − 2 × (n mod 2)) end if
5:     (x1, y1) ← (0, 1) and (x2, y2) ← (x, y)
6:     n ← n mod card
7:     for i ← ℓ down to 1 do
8:         if n_{i−1} = 0 then
9:             (x2, y2) ← curve.pointAddition((x1, y1), (x2, y2))
10:            (x1, y1) ← curve.pointAddition((x1, y1), (x1, y1))
11:        else
12:            (x1, y1) ← curve.pointAddition((x1, y1), (x2, y2))
13:            (x2, y2) ← curve.pointAddition((x2, y2), (x2, y2))
14:        end if
15:    end for
16:    return (x1, y1)
17: end procedure
```

curve.mulAdd		
parameters		
in	$[0, 1, \dots, q-1]$	a
	$[0, 1, \dots, p-1]^2$	(x_1, y_1)
	$[0, 1, \dots, q-1]$	b
	$\big([0, 1, \dots, p-1] \cup \perp\big) \times [0, 1, \dots, p-1]$	(x_2, y_2)
out	$[0, 1, \dots, p-1]^2 \times [0, 1, \dots, p-1]^2$	(Q, Q')

Letting $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$, we denote by

$$\text{curve.mulAdd}(a, (x_1, y_1), b, (x_2, y_2))$$

the call to the procedure that computes $Q = aP_1 + bP_2$ on the curve identified by curve. In case $x_2 = \perp$, the procedure either fails (when there is no x_2 such that (x_2, y_2) is on the curve) or finds two candidates for Q. Those two candidates are returned by the function. When $x_2 \neq \perp$, the procedure either fails (when (x_2, y_2) is not on the curve) or outputs (Q, Q') such that $Q = Q'$. The procedure also fails if (x_1, y_1) is not on the curve. The procedure works as follows:

```
1: procedure curve.mulAdd(a, (x1, y1), b, (x2, y2))
2:     (x3, y3) ← curve.scalarMultiplicationWithX(a, (x1, y1))
3:     if x2 ≠ ⊥ then
4:         (x4, y4) ← curve.scalarMultiplicationWithX(b, (x2, y2))
5:         (x, y) ← curve.pointAddition((x3, y3), (x4, y4))
```

6: $(x', y') \leftarrow (x, y)$
7: **else**
8: $y_4 \leftarrow$ curve.scalarMultiplication(b, y_2)
9: $(x_4, x_4') \leftarrow$ xCoordinatesFromY$(y_4,$ curve$)$
10: $(x, y) \leftarrow$ curve.pointAddition$((x_3, y_3), (x_4, y_4))$
11: $(x', y') \leftarrow$ curve.pointAddition$((x_3, y_3), (x_4', y_4))$
12: **end if**
13: return $((x, y), (x', y'))$
14: **end procedure**

Note that step 2 fails if (x_1, y_1) is not on the curve. Similarly, step 4 fails if (x_2, y_2) is not on the curve. Moreover, step 9 fails in case y_4 cannot be the y-coordinate of a point on the curve. For these reasons, we do not need to explicitly check if (x_1, y_1), (x_2, y_2), (x_4, y_4), and (x_4', y_4) are on the curve.

curve.generateRandomScalarAndPoint		
parameters		
in	PRNG	prng
out	$[0, 1, \ldots, q-1]$ $[0, 1, \ldots, p-1]^2$	λ Q

Given an instance prng of PRNG, we denote by

$$\text{curve.generateRandomScalarAndPoint}(\text{prng}) \rightarrow (\lambda, Q)$$

the call to the procedure that generates a random scalar $\lambda \in [2, \ldots, q-1]$ and computes the point $Q = \lambda \times$ curve.G. The procedure works as follows:

1: **procedure** curve.generateRandomScalarAndPoint(prng)
2: **repeat** $\lambda \leftarrow$ prng.bigInt(curve.q) **until** $\lambda \notin \{0, 1\}$
3: $Q \leftarrow$ curve.scalarMultiplicationWithX$(\lambda,$ curve.$G)$
4: return (λ, Q)
5: **end procedure**

curve.isLowOrderPoint		
parameters		None
in	$[0, 1, \ldots, p-1]$	y
out	{`True`, `False`}	`bool`

Given the y-coordinate of a point P of the curve, we denote by

$$\text{curve.isLowOrderPoint}(y) \rightarrow \texttt{bool}$$

the procedure that returns `True` if P belongs to the small subgroup of order dividing the cofactor ν, that is, if $\nu \times P$ is the neutral element (whose y-coordinate is 1). This procedure is used to reject low order public keys. It works as follows:

```
1: procedure curve.isLowOrderPoint(y)
2:     return (curve.scalarMultiplication(curve.ν, y) = 1)
3: end procedure
```

12.1 Curve25519

We denote by Curve25519 the concrete subtype of EdwardsCurve that corresponds to Curve25519 [5]. The following initializer creates a Curve25519 instance.

Curve25519 (Initializer)		
parameters		None
in		

```
1: procedure Curve25519()
2:     self.p ← 2^255 − 19
3:     self.d ← 20800338683988658368647408995589388737092878452977063003340006470870624536394
4:     self.G.x ← 9771384041963202563870679428059935816164187996444183106833894008023910952347
5:     self.G.y ← 4631683569492647816942839400347516314130799386625622561578303360316525185596O
6:     self.q ← 7237005577332262213973186563042994240857116359379907606001950938285454250989
7:     self.ν ← 8
8: end procedure
```

12.2 MDC

We denote by MDC the concrete subtype of EdwardsCurve that corresponds to MDC [2]. The following initializer creates a MDC instance.

MDC (Initializer)		
parameters		None
in		

```
1: procedure MDC()
2:     self.p ← 109112363276961190442711090369149551676330307646118204517771511330536253156371
3:     self.d ← 39384817741350628573161184301225915800358770588933756071948264625804612259721
4:     self.G.x ← 8254980322220239934002446203296494251202585681870041425472636420509673142431S
5:     self.G.y ← 9154954563741573442265828879911904175637825952309714780781339691512593281144S
6:     self.q ← 272780908192402976106777725922873879189305095740480688876309782931855219732O3
```

7: self.$\nu \leftarrow 4$
8: **end procedure**

13 Public and Private Keys over Edwards Curves

Public keys used for digital signatures are instances of a complex type, denoted PublicKeyOverEC, which is a subtype of PublicKey (see Section 3.2.2). The type PublicKeyOverEC is an *abstract* type. A PublicKeyOverEC instance will always be an instance of a *concrete* subtype of PublicKeyOverEC. Private keys used for digital signatures are instances of a complex type, denoted PrivateKeyOverEC, which is a subtype of PrivateKey (see Section 3.2.3). The type PrivateKeyOverEC is an *abstract* type. A PrivateKeyOverEC instance will always be an instance of a *concrete* subtype of PrivateKeyOverEC.

An instance pk of PublicKeyOverEC gives access to the underlying elliptic curve instance by calling pk.curve, which is one of the concrete subtypes of EdwardsCurve. A public key on an elliptic curve is also defined by a base point. The special form of elliptic curves we use makes it possible to only keep the y-coordinate of this base point. As a consequence, an instance pk of PublicKeyOverEC always gives access to the y-coordinate of the base point by calling pk.y. When available, it also gives access to the full base point, by calling pk.point.

An instance sk of PrivateKeyOverEC gives access to the underlying elliptic curve instance by calling sk.curve, which is one of the concrete subtypes of EdwardsCurve. Moreover, a call to sk.scalar gives access to the underlying scalar of the private key.

The following initializer will systematically be called by the initializer of subtypes of PublicKeyOverEC when the full base point is available.

PublicKeyOverEC (Initializer)		
parameters		None
in	$\{0,\dots,255\}$	`algoClassByteId`
	$\{0,\dots,255\}$	`algoImplemByteId`
	EdwardsCurve	curve
	$[0,\dots,p-1]^2$	P

We denote by

$$\text{PublicKeyOverEC}(\texttt{algoClassByteId}, \texttt{algoImplemByteId}, \text{curve}, P) \rightarrow \text{pk}$$

the call to the PublicKeyOverEC initializer.

1: **procedure** PublicKeyOverEC(`algoClassByteId`, `algoImplemByteId`, curve, P)
2: **if** not curve.isOnCurve(P) **then** return $\perp$ **end if**
3: self.point $\leftarrow P$
4: self.$y \leftarrow P.y$
5: self.curve $\leftarrow$ curve
6: `dict["x"]` $\leftarrow$ encodeBigUInt($P.x \mod p$, len(curve.p))

7: `dict["y"]` $\leftarrow$ encodeBigUInt($P.y \mod p$, len(curve.p))
8: PublicKey(`algoClassByteId`, `algoImplemByteId`, `dict`)
9: **end procedure**

The following initializer will systematically be called by the initializer of subtypes of PublicKeyOverEC when the full base point is *not* available.

PublicKeyOverEC (Initializer)		
parameters		None
in	$\{0,\dots,255\}$	`algoClassByteId`
	$\{0,\dots,255\}$	`algoImplemByteId`
	EdwardsCurve	curve
	$[0,\dots,p-1]$	y

We denote by

$$\text{PublicKeyOverEC}(\texttt{algoClassByteId}, \texttt{algoImplemByteId}, \text{curve}, y) \rightarrow \text{pk}$$

the call to the PublicKeyOverEC initializer.

1: **procedure** PublicKeyOverEC(`algoClassByteId`, `algoImplemByteId`, curve, y)
2: self.point $\leftarrow \perp$
3: self.$y \leftarrow y$
4: self.curve $\leftarrow$ curve
5: `dict["y"]` $\leftarrow$ encodeBigUInt($y \mod p$, len(curve.p))
6: PublicKey(`algoClassByteId`, `algoImplemByteId`, `dict`)
7: **end procedure**

The following initializer will systematically be called by the initializer of subtypes of PrivateKeyOverEC.

PrivateKeyOverEC (Initializer)		
parameters		None
in	$\{0,\dots,255\}$	`algoClassByteId`
	$\{0,\dots,255\}$	`algoImplemByteId`
	EdwardsCurve	curve
	$[0,\dots,q-1]$	λ

We denote by

$$\text{PrivateKeyOverEC}(\texttt{algoClassByteId}, \texttt{algoImplemByteId}, \text{curve}, \lambda) \rightarrow \text{sk}$$

the call to the PrivateKeyOverEC initializer.

1: **procedure** PrivateKeyOverEC(`algoClassByteId`, `algoImplemByteId`, curve, λ)
2: self.scalar $\leftarrow \lambda$
3: self.curve $\leftarrow$ curve
4: `dict["n"]` $\leftarrow$ encodeBigUInt($\lambda \mod q$, len(curve.q))
5: PrivateKey(`algoClassByteId`, `algoImplemByteId`, `dict`)
6: **end procedure**

pk.getCompactKey		
parameters		None
in		
out	$\{0,\ldots,255\}^*$	`compactKey`

Given a public key instance pk of PublicKeyOverEC, the above procedure returns a compact byte array representation of the public key. We denote by

pk.getCompactKey() $\rightarrow$ `compactKey`

the call to the pk.getCompactKey procedure.

1: **procedure** pk.getCompactKey()
2: return self.`algoImplemByteId` $\|$ bytesFromBigUInt(pk.y)
3: **end procedure**

PublicKeyOverEC.expandCompactKey		
parameters		None
in	$\{0,\ldots,255\}^*$	`compactKey`
out	PublicKeyOverEC	pk

Given a compact key `compactKey`, the previous procedure recovers a PublicKeyOverEC instance. This method is abstract and only implemented by concrete subtypes of PublicKeyOverEC.

14 Signature

All signature schemes we consider within these specifications are defined over an Edwards curve and are subtypes of SignatureOverEC. Public keys used for digital signatures are instances of a complex type, denoted SignaturePublicKeyOverEC, which is a subtype of PublicKeyOverEC (see Section 13). The type SignaturePublicKeyOverEC is an *abstract* type. A SignaturePublicKeyOverEC instance will always be an instance of a *concrete* subtype of SignaturePublicKeyOverEC. Private keys used for digital signatures are instances of a complex type, denoted SignaturePrivateKeyOverEC, which is a subtype of PrivateKeyOverEC (see Section 13). The type SignaturePrivateKeyOverEC is an *abstract*

type. A SignaturePrivateKeyOverEC instance will always be an instance of a *concrete* subtype of SignaturePrivateKeyOverEC.

SignatureOverEC.curveFromAlgoImplemByteId		
parameters		None
in	$\{0,\dots,255\}$	`algoImplemByteId`
out	EdwardsCurve	curve

We denote by

$$\text{SignatureOverEC.curveFromAlgoImplemByteId}(\texttt{algoImplemByteId}) \to \text{curve}$$

the call to the SignatureOverEC.curveFromAlgoImplemByteId procedure.

```
1: procedure SignatureOverEC.curveFromAlgoImplemByteId(algoImplemByteId)
2:     if algoImplemByteId = 0x00 then return MDC() end if
3:     if algoImplemByteId = 0x01 then return Curve25519() end if
4:     return ⊥
5: end procedure
```

SignatureOverEC.algoImplemByteIdFromCurve		
parameters		None
in	EdwardsCurve	curve
out	$\{0,\dots,255\}$	`algoImplemByteId`

We denote by

$$\text{SignatureOverEC.algoImplemByteIdFromCurve}(\text{curve}) \to \texttt{algoImplemByteId}$$

the call to the SignatureOverEC.algoImplemByteIdFromCurve procedure.

```
1: procedure SignatureOverEC.algoImplemByteIdFromCurve(curve)
2:     if curve = MDC() then return 0x00 end if
3:     if curve = Curve25519() then return 0x01 end if
4:     return ⊥
5: end procedure
```

The following initializer will systematically be called by the initializer of subtypes of SignaturePublicKeyOverEC when the full base point is available.

SignaturePublicKeyOverEC (Initializer)		
parameters		None
in	EdwardsCurve $[0, \ldots, p-1]^2$	curve P

We denote by

$$\mathsf{SignaturePublicKeyOverEC}(\mathsf{curve}, P) \to \mathsf{pk}$$

the call to the SignaturePublicKeyOverEC initializer.

```
1: procedure SignaturePublicKeyOverEC(curve, P)
2:     algoClassByteId ← 0x11
3:     algoImplemByteId ← algoImplemByteIdFromCurve(curve)
4:     PublicKeyOverEC(algoClassByteId, algoImplemByteId, curve, P)
5: end procedure
```

The following initializer will systematically be called by the initializer of subtypes of SignaturePublicKeyOverEC when the full base point is *not* available.

SignaturePublicKeyOverEC (Initializer)		
parameters		None
in	EdwardsCurve $[0, \ldots, p-1]$	curve y

We denote by

$$\mathsf{SignaturePublicKeyOverEC}(\mathsf{curve}, y) \to \mathsf{pk}$$

the call to the SignaturePublicKeyOverEC initializer.

```
1: procedure SignaturePublicKeyOverEC(curve, y)
2:     algoClassByteId ← 0x11
3:     algoImplemByteId ← algoImplemByteIdFromCurve(curve)
4:     PublicKeyOverEC(algoClassByteId, algoImplemByteId, curve, y)
5: end procedure
```

The following initializer will systematically be called by the initializer of subtypes of SignaturePrivateKeyOverEC.

SignaturePrivateKeyOverEC (Initializer)		
parameters		None
in	EdwardsCurve $[0, \ldots, q-1]$	curve λ

We denote by

$$\mathsf{SignaturePrivateKeyOverEC}(\mathsf{curve}, \lambda) \rightarrow \mathsf{sk}$$

the call to the SignaturePrivateKeyOverEC initializer.

```
1: procedure SignaturePrivateKeyOverEC(curve, λ)
2:     algoClassByteId ← 0x11
3:     algoImplemByteId ← algoImplemByteIdFromCurve(curve)
4:     PrivateKeyOverEC(algoClassByteId, algoImplemByteId, curve, λ)
5: end procedure
```

SignatureOverEC.generateKeyPair		
parameters		None
in	PRNG	prng
	EdwardsCurve	curve
out	SignaturePublicKeyOverEC	pk
	SignaturePrivateKeyOverEC	sk

We denote by

$$\mathsf{SignatureOverEC.generateKeyPair}(\mathsf{prng}, \mathsf{curve}) \rightarrow (\mathsf{pk}, \mathsf{sk})$$

the call to the SignatureOverEC.generateKeyPair procedure.

```
1: procedure SignatureOverEC.generateKeyPair(prng, curve)
2:     (λ, P) ← curve.generateRandomScalarAndPoint(prng)
3:     pk ← SignaturePublicKeyOverEC(curve, P)
4:     sk ← SignaturePrivateKeyOverEC(curve, λ)
5:     return (pk, sk)
6: end procedure
```

SignatureOverEC.sign		
parameters		None
in	SignaturePrivateKeyOverEC	sk
	$\{0,\ldots,255\}^*$	m
	SignaturePublicKeyOverEC	pk
	PRNG	prng
out	$\{0,\ldots,255\}^*$	σ

We denote by

$$\mathsf{SignatureOverEC.sign}(\mathsf{sk}, \mathtt{m}, \mathsf{pk}, \mathsf{prng}) \rightarrow \sigma$$

the procedure that computes the signature σ of a message $\mathtt{m}$ under the private key sk (associated with the public key pk), using the initialized PRNG instance prng. The procedure works as follows (see [17, p.166]):

```
1: procedure SignatureOverEC.sign(sk, m, pk, prng)
2:     if sk.curve ≠ pk.curve then return ⊥ end if
3:     curve ← sk.curve
4:     (p, d, G, q, ν, card) ← curve.parameters
5:
6:     /* The generateKeyPair procedure is used to generate a random scalar and
   point */
7:     (pk′, sk′) ← SignatureOverEC.generateKeyPair(prng, curve)
8:
9:     /* Construct the data to hash */
10:    Ay′ ← bytesFromBigUInt(pk′.y mod p, len(p))
11:    Ay ← bytesFromBigUInt(pk.y mod p, len(p))
12:    data ← Ay′‖Ay‖m
13:
14:    /* Hash and map the digest onto a big unsigned integer */
15:    h ← SHA256(data)
16:    e ← bigUIntFromBytes(h)
17:
18:    /* Sign */
19:    r ← sk′.scalar
20:    a ← sk.scalar
21:    y ← (r − a × e) mod q
22:    z ← bytesFromBigUInt(y, len(p))
23:    σ ← h‖z
24:    return σ
25: end procedure
```

Note that the signing procedure currently uses SHA256 as hash function. The verification procedure below already accepts signatures computed with SHA512 instead of SHA256 (in which case $\mathtt{h}$ is 64 bytes long), as signing with SHA512 will soon become the default.

SignatureOverEC.verify		
parameters		None
in	SignaturePublicKeyOverEC	pk
	$\{0,\ldots,255\}^*$	m
	$\{0,\ldots,255\}^*$	σ
out	$\{\mathtt{True},\mathtt{False}\}$	b

We denote by

$$\mathsf{SignatureOverEC.verify}(\mathsf{pk},\mathtt{m},\sigma) \to b$$

the procedure that verifies the signature σ of a message `m` under the public key `pk`. It returns `True` if the signature is valid, `False` otherwise. The procedure works as follows:

```
1:  procedure SignatureOverEC.verify(pk, m, σ)
2:      curve ← pk.curve
3:      (p, d, G, q, ν, card) ← curve.parameters
4:
5:      /* Parse the signature and determine the hash function */
6:      if len(σ) = 32 + len(p) then
7:          hash ← SHA256 and ℓ_h ← 32
8:      else if len(σ) = 64 + len(p) then
9:          hash ← SHA512 and ℓ_h ← 64
10:     else
11:         return False
12:     end if
13:     Parse h||z ← σ where len(h) = ℓ_h
14:
15:     /* Reject low order public keys */
16:     if curve.isLowOrderPoint(pk.y) then return False end if
17:
18:     /* Extract the big integers */
19:     e ← bigUIntFromBytes(h)
20:     y ← bigUIntFromBytes(z)
21:     if y ≥ q then return False end if
22:
23:     /* Compute the resulting point(s) */
24:     P ← (pk.point ≠ ⊥ ? pk.point : (⊥, pk.y))
25:     (A1, A2) ← mulAdd(y, G, e, P)
26:
27:     /* Compute the corresponding data to hash */
28:     Ay ← bytesFromBigUInt(pk.y, len(p))
29:     A1y ← bytesFromBigUInt(A1.y, len(p))
30:     A2y ← bytesFromBigUInt(A2.y, len(p))
31:     data1 ← A1y||Ay||m
32:     data2 ← A2y||Ay||m
33:
34:     /* Hash the data.  The signature is valid if there is a match.  */
35:     h1 ← hash(data1)
36:     h2 ← hash(data2)
37:     return (h = h1 or h = h2)
38: end procedure
```

14.1 Signature Key Generation over Curve25519

We denote by SignatureOverCurve25519 the concrete subtype of SignatureOverEC used to perform and check digital signatures over Curve25519.

SignatureOverCurve25519.generateKeyPair		
parameters		None
in	PRNG	prng
out	SignaturePublicKeyOverEC SignaturePrivateKeyOverEC	pk sk

We denote by

$$\text{SignatureOverCurve25519.generateKeyPair}(\text{prng}) \rightarrow (\text{pk}, \text{sk})$$

the call to the SignatureOverCurve25519.generateKeyPair procedure.

```
1: procedure SignatureOverCurve25519.generateKeyPair(prng)
2:     curve ← Curve25519()
3:     return SignatureOverEC.generateKeyPair(prng, curve)
4: end procedure
```

14.2 Signature Key Generation over MDC

We denote by SignatureOverMDC the concrete subtype of SignatureOverEC used to perform and check digital signatures over MDC.

SignatureOverMDC.generateKeyPair		
parameters		None
in	PRNG	prng
out	SignaturePublicKeyOverEC SignaturePrivateKeyOverEC	pk sk

We denote by

$$\text{SignatureOverMDC.generateKeyPair}(\text{prng}) \rightarrow (\text{pk}, \text{sk})$$

the call to the SignatureOverMDC.generateKeyPair procedure.

```
1: procedure SignatureOverMDC.generateKeyPair(prng)
2:     curve ← MDC()
3:     return SignatureOverEC.generateKeyPair(prng, curve)
4: end procedure
```

15 Authentication

Within these specifications, authentication schemes leverage digital signatures and are subtypes of AuthenticationOverEC. Public keys used for checking a solution to an authentication challenge are instances of a complex type, denoted AuthenticationPublicKeyOverEC, which is a subtype of PublicKeyOverEC (see Section 13). The type AuthenticationPublicKeyOverEC is an *abstract* type. A AuthenticationPublicKeyOverEC instance will always be an instance of a *concrete* subtype of AuthenticationPublicKeyOverEC. Private keys used for solving an authentication challenge are instances of a complex type, denoted AuthenticationPrivateKeyOverEC, which is a subtype of PrivateKeyOverEC (see Section 13). The type AuthenticationPrivateKeyOverEC is an *abstract* type. A AuthenticationPrivateKeyOverEC instance will always be an instance of a *concrete* subtype of AuthenticationPrivateKeyOverEC.

AuthenticationOverEC.curveFromAlgoImplemByteId		
parameters		None
in	$\{0,\dots,255\}$	`algoImplemByteId`
out	EdwardsCurve	curve

We denote by

AuthenticationOverEC.curveFromAlgoImplemByteId(`algoImplemByteId`) → curve

the call to the AuthenticationOverEC.curveFromAlgoImplemByteId procedure.

```
1: procedure AuthenticationOverEC.curveFromAlgoImplemByteId(algoImplemByteId)
2:     if algoImplemByteId = 0x00 then return MDC() end if
3:     if algoImplemByteId = 0x01 then return Curve25519() end if
4:     return ⊥
5: end procedure
```

AuthenticationOverEC.algoImplemByteIdFromCurve		
parameters		None
in	EdwardsCurve	curve
out	$\{0,\dots,255\}$	`algoImplemByteId`

We denote by

AuthenticationOverEC.algoImplemByteIdFromCurve(curve) → `algoImplemByteId`

the call to the AuthenticationOverEC.algoImplemByteIdFromCurve procedure.

```
1: procedure AuthenticationOverEC.algoImplemByteIdFromCurve(curve)
```

```
    if curve = MDC() then return 0x00 end if
    if curve = Curve25519() then return 0x01 end if
    return ⊥
end procedure
```

The following initializer will systematically be called by the initializer of subtypes of AuthenticationPublicKeyOverEC when the full base point is available.

AuthenticationPublicKeyOverEC (Initializer)		
parameters		None
in	EdwardsCurve $[0, \ldots, p-1]^2$	curve P

We denote by

$$\mathsf{AuthenticationPublicKeyOverEC}(\mathsf{curve}, P) \to \mathsf{pk}$$

the call to the AuthenticationPublicKeyOverEC initializer.

```
procedure AuthenticationPublicKeyOverEC(curve, P)
    algoClassByteId ← 0x14
    algoImplemByteId ← algoImplemByteIdFromCurve(curve)
    PublicKeyOverEC(algoClassByteId, algoImplemByteId, curve, P)
end procedure
```

The following initializer will systematically be called by the initializer of subtypes of AuthenticationPublicKeyOverEC when the full base point is *not* available.

AuthenticationPublicKeyOverEC (Initializer)		
parameters		None
in	EdwardsCurve $[0, \ldots, p-1]$	curve y

We denote by

$$\mathsf{AuthenticationPublicKeyOverEC}(\mathsf{curve}, y) \to \mathsf{pk}$$

the call to the AuthenticationPublicKeyOverEC initializer.

```
procedure AuthenticationPublicKeyOverEC(curve, y)
    algoClassByteId ← 0x14
    algoImplemByteId ← algoImplemByteIdFromCurve(curve)
    PublicKeyOverEC(algoClassByteId, algoImplemByteId, curve, y)
end procedure
```

The following initializer will systematically be called by the initializer of subtypes of AuthenticationPrivateKeyOverEC.

AuthenticationPrivateKeyOverEC (Initializer)		
parameters		None
in	EdwardsCurve $[0, \ldots, q-1]$	curve λ

We denote by

$$\mathsf{AuthenticationPrivateKeyOverEC}(\mathsf{curve}, \lambda) \rightarrow \mathsf{sk}$$

the call to the AuthenticationPrivateKeyOverEC initializer.

1: **procedure** AuthenticationPrivateKeyOverEC(curve, λ)
2: `algoClassByteId` $\leftarrow$ `0x14`
3: `algoImplemByteId` $\leftarrow$ algoImplemByteIdFromCurve(curve)
4: PrivateKeyOverEC(`algoClassByteId`, `algoImplemByteId`, curve, λ)
5: **end procedure**

AuthenticationOverEC.generateKeyPair		
parameters		None
in	PRNG EdwardsCurve	prng curve
out	AuthenticationPublicKeyOverEC AuthenticationPrivateKeyOverEC	pk sk

We denote by

$$\mathsf{AuthenticationOverEC.generateKeyPair}(\mathsf{prng}, \mathsf{curve}) \rightarrow (\mathsf{pk}, \mathsf{sk})$$

the call to the AuthenticationOverEC.generateKeyPair procedure.

1: **procedure** AuthenticationOverEC.generateKeyPair(prng, curve)
2: **repeat** $(\lambda, P) \leftarrow$ curve.generateRandomScalarAndPoint(prng) **until** not curve.isLowOrderPoint($P.y$)
3: pk $\leftarrow$ AuthenticationPublicKeyOverEC(curve, P)
4: sk $\leftarrow$ AuthenticationPrivateKeyOverEC(curve, λ)
5: return (pk, sk)
6: **end procedure**

Given an instance pk of AuthenticationPublicKeyOverEC, the following procedure returns an instance pk_σ of SignaturePublicKeyOverEC over the same curve, with the same base point.

pk.convertToSignatureKey		
parameters		None
in	None	None
out	SignaturePublicKeyOverEC	pk_σ

We denote by

$$\mathsf{pk.convertToSignatureKey}() \rightarrow \mathsf{pk}_\sigma$$

the call to the pk.convertToSignatureKey procedure.

```
1: procedure pk.convertToSignatureKey()
2:     if pk.point ≠ ⊥ then
3:         return SignaturePublicKeyOverEC(pk.curve, pk.point)
4:     else
5:         return SignaturePublicKeyOverEC(pk.curve, pk.y)
6:     end if
7: end procedure
```

Given an instance sk of AuthenticationPrivateKeyOverEC, the following procedure returns an instance sk_σ of SignaturePrivateKeyOverEC over the same curve, with the same scalar.

sk.convertToSignatureKey		
parameters		None
in	None	None
out	SignaturePrivateKeyOverEC	sk_σ

We denote by

$$\mathsf{sk.convertToSignatureKey}() \rightarrow \mathsf{sk}_\sigma$$

the call to the sk.convertToSignatureKey procedure.

```
1: procedure sk.convertToSignatureKey()
2:     return SignaturePrivateKeyOverEC(sk.curve, sk.λ)
3: end procedure
```

AuthenticationOverEC.solve		
parameters		None

in	AuthenticationPrivateKeyOverEC
	$\{0,\dots,255\}^*$
	$\{0,\dots,255\}^*$
	AuthenticationPublicKeyOverEC
	PRNG
		sk
		`challenge`
		`prefix`
		pk
		prng

out	$\{0,\dots,255\}^*$	`response`

We denote by

$$\mathsf{AuthenticationOverEC.solve}(\mathsf{sk}, \texttt{challenge}, \texttt{prefix}, \mathsf{pk}, \mathsf{prng}) \to \texttt{response}$$

the procedure that produces a solution `response` to an authentication challenge `challenge` prefixed with `prefix` under the private key sk (associated with the encoded public key pk), using the initialized PRNG instance prng. The procedure works as follows:

1: **procedure** AuthenticationOverEC.solve(sk, `challenge`, `prefix`, pk, prng)
2: **if** sk.curve $\neq$ pk.curve **then** return $\perp$ **end if**
3: `suffix` $\leftarrow$ prng.bytes(16)
4: `formatted_challenge` $\leftarrow$ `prefix`$\|$`challenge`$\|$`suffix`
5: $\mathsf{pk}_\sigma \leftarrow$ pk.convertToSignatureKey()
6: $\mathsf{sk}_\sigma \leftarrow$ sk.convertToSignatureKey()
7: $\sigma \leftarrow$ SignatureOverEC.sign(sk_σ, `formatted_challenge`, pk_σ, prng)
8: return `suffix`$\|\sigma$
9: **end procedure**

AuthenticationOverEC.check		
parameters		None

in	$\{0,\dots,255\}^*$
	$\{0,\dots,255\}^*$
	$\{0,\dots,255\}^*$
	AuthenticationPublicKeyOverEC
		`response`
		`challenge`
		`prefix`
		pk

out	{`True`, `False`}	`bool`

We denote by

$$\mathsf{AuthenticationOverEC.check}(\texttt{response}, \texttt{challenge}, \texttt{prefix}, \mathsf{pk}) \to \texttt{bool}$$

the procedure that checks the challenge response `response` to an authentication challenge `challenge` prefixed with `prefix` using the public key pk. The procedure works as follows:

1: **procedure** AuthenticationOverEC.check(`response`, `challenge`, `prefix`, pk)
2: **if** len(`response`) < 16 **then** return `False` **end if**
3: Parse `response` as `suffix`$\|\sigma$ where len(`suffix`) $= 16$
4: `formatted_challenge` $\leftarrow$ `prefix`$\|$`challenge`$\|$`suffix`
5: $\text{pk}_\sigma \leftarrow$ pk.convertToSignatureKey()
6: return SignatureOverEC.verify(pk_σ, `formatted_challenge`, σ)
7: **end procedure**

Signatures with context. Beyond the authentication to the server, the solve and check procedures are used everywhere a signature is required in the engine: the data to sign is passed as the `challenge`, and the `prefix` identifies the context in which the signature is produced. This ensures that a signature produced in one context can never be accepted in another context. Table 1 lists all the prefixes used in Olvid (none of these prefixes is a prefix of another one). They are encoded in UTF-8, without any terminating character. Unless stated otherwise, the key used is the authentication key of the owned identity of the signer.

Prefix	Key	Usage
`authentChallenge`	identity	authentication to the server
`mutualScan`	identity	trust establishment with mutual QR-code scan
`mutualIntroduction`	identity	contact mutual introduction
`channelCreation`	identity	channel creation (with a contact or owned device)
`devicePreKey`	identity	publication of a device pre-key
`encryptionWithPreKey`	identity	messages encrypted with a pre-key
`ownedIdentityDeletion`	identity	notification of the deletion of an owned identity
`groupAdministratorsChain`	identity	groups v2: administrators chain blocks
`groupBlob`	identity	groups v2: group blob
`groupLeave`	identity	groups v2: leaving a group
`joinGroup`	identity	groups v2: group joining ping
`groupKick`	identity	groups v2: removal of a member
`lockNonce`	group admin key	groups v2: lock of the group blob on the server
`updateGroup`	group admin key	groups v2: update of the group blob on the server
`deleteGroup`	group admin key	groups v2: deletion of the group blob on the server
`backupUpload`	backup key	upload of a backup to the server
`backupDelete`	backup key	deletion of a backup from the server
`keycloakChallenge`	identity	identity-based authentication to a Keycloak server

Table 1: Prefixes used for signatures in Olvid.

AuthenticationPublicKeyOverEC.expandCompactKey		
parameters		None
in	$\{0,\dots,255\}^*$	`compactKey`
out	AuthenticationPublicKeyOverEC	pk

We denote by

AuthenticationPublicKeyOverEC.expandCompactKey(`compactKey`) $\rightarrow$ AuthenticationPublicKeyOverEC

the procedure that recovers an instance pk of AuthenticationPublicKeyOverEC given a compact key `compactKey`. The procedure works as follows:

```
1: procedure AuthenticationPublicKeyOverEC.expandCompactKey(compactKey)
2:     if len(compactKey) = 0 then return ⊥ end if
3:     Parse compactKey as algoImplemByteId‖yCoordinate where len(algoImplemByteId) = 1
4:     curve ← AuthenticationPublicKeyOverEC.curveFromAlgoImplemByteId(algoImplemByteId)
5:     if len(compactKey) ≠ 1 + len(curve.p) then return ⊥ end if
6:     y ← bigUIntFromBytes(yCoordinate)
7:     return AuthenticationPublicKeyOverEC(curve, y)
8: end procedure
```

15.1 Authentication over Curve25519

We denote by AuthenticationOverCurve25519 the concrete subtype of AuthenticationOverEC used to solve authentication challenges and to check those solutions over Curve25519.

AuthenticationOverCurve25519.generateKeyPair		
parameters		None
in	PRNG	prng
out	AuthenticationPublicKeyOverEC AuthenticationPrivateKeyOverEC	pk sk

We denote by

$$\text{AuthenticationOverCurve25519.generateKeyPair}(\text{prng}) \rightarrow (\text{pk}, \text{sk})$$

the call to the AuthenticationOverCurve25519.generateKeyPair procedure.

```
1: procedure AuthenticationOverCurve25519.generateKeyPair(prng)
2:     curve ← Curve25519()
3:     return AuthenticationOverEC.generateKeyPair(prng, curve)
4: end procedure
```

15.2 Authentication over MDC

We denote by AuthenticationOverMDC the concrete subtype of AuthenticationOverEC used to solve authentication challenges and to check those solutions over MDC.

AuthenticationOverMDC.generateKeyPair		
parameters		None
in	PRNG	prng
out	AuthenticationPublicKeyOverEC AuthenticationPrivateKeyOverEC	pk sk

We denote by

$$\mathsf{AuthenticationOverMDC.generateKeyPair}(\mathsf{prng}) \rightarrow (\mathsf{pk}, \mathsf{sk})$$

the call to the AuthenticationOverMDC.generateKeyPair procedure.

```
1: procedure AuthenticationOverMDC.generateKeyPair(prng)
2:     curve ← MDC()
3:     return AuthenticationOverEC.generateKeyPair(prng, curve)
4: end procedure
```

16 Key Encapsulation Mechanism (KEM)

Key Encapsulation Mechanisms (KEMs) are instances of a complex type denoted KEMOverEC. Public keys used for encapsulating a key are instances of a complex type, denoted KEMPublicKeyOverEC, which is a subtype of PublicKeyOverEC (see Section 13). The type KEMPublicKeyOverEC is an *abstract* type. A KEMPublicKeyOverEC instance will always be an instance of a *concrete* subtype of KEMPublicKeyOverEC. Private keys used for decrypting an encapsulated key are instances of a complex type, denoted KEMPrivateKeyOverEC, which is a subtype of PrivateKeyOverEC (see Section 3.2.3). The type KEMPrivateKeyOverEC is an *abstract* type. A KEMPrivateKeyOverEC instance will always be an instance of a *concrete* subtype of KEMPrivateKeyOverEC.

KEMOverEC.curveFromAlgoImplemByteId		
parameters		None
in	$\{0, \ldots, 255\}$	`algoImplemByteId`
out	EdwardsCurve	curve

We denote by

$$\mathsf{KEMOverEC.curveFromAlgoImplemByteId}(\texttt{algoImplemByteId}) \rightarrow \mathsf{curve}$$

the call to the KEMOverEC.curveFromAlgoImplemByteId procedure.

```
1: procedure KEMOverEC.curveFromAlgoImplemByteId(algoImplemByteId)
2:     if algoImplemByteId = 0x00 then return MDC() end if
3:     if algoImplemByteId = 0x01 then return Curve25519() end if
4:     return ⊥
```

5: **end procedure**

KEMOverEC.algoImplemByteIdFromCurve		
parameters		None
in	EdwardsCurve	curve
out	$\{0,\dots,255\}$	`algoImplemByteId`

We denote by

$$\mathsf{KEMOverEC.algoImplemByteIdFromCurve}(\mathsf{curve}) \to \texttt{algoImplemByteId}$$

the call to the KEMOverEC.algoImplemByteIdFromCurve procedure.

1: **procedure** KEMOverEC.algoImplemByteIdFromCurve(curve)
2: **if** curve = MDC() **then** return `0x00` **end if**
3: **if** curve = Curve25519() **then** return `0x01` **end if**
4: return $\perp$
5: **end procedure**

The following initializer will systematically be called by the initializer of subtypes of KEMPublicKeyOverEC when the full base point is available.

KEMPublicKeyOverEC (Initializer)		
parameters		None
in	EdwardsCurve $[0,\dots,p-1]^2$	curve P

We denote by

$$\mathsf{KEMPublicKeyOverEC}(\mathsf{curve}, P) \to \mathsf{pk}$$

the call to the KEMPublicKeyOverEC initializer.

1: **procedure** KEMPublicKeyOverEC(curve, P)
2: `algoClassByteId` $\leftarrow$ `0x12`
3: `algoImplemByteId` $\leftarrow$ algoImplemByteIdFromCurve(curve)
4: PublicKeyOverEC(`algoClassByteId`, `algoImplemByteId`, curve, P)
5: **end procedure**

The following initializer will systematically be called by the initializer of subtypes of KEMPublicKeyOverEC when the full base point is *not* available.

KEMPublicKeyOverEC (Initializer)		
parameters		None
in	EdwardsCurve $[0, \ldots, p-1]$	curve y

We denote by

$$\mathsf{KEMPublicKeyOverEC}(\mathsf{curve}, y) \to \mathsf{pk}$$

the call to the KEMPublicKeyOverEC initializer.

```
1: procedure KEMPublicKeyOverEC(curve, y)
2:     algoClassByteId ← 0x12
3:     algoImplemByteId ← algoImplemByteIdFromCurve(curve)
4:     PublicKeyOverEC(algoClassByteId, algoImplemByteId, curve, y)
5: end procedure
```

The following initializer will systematically be called by the initializer of subtypes of KEMPrivateKeyOverEC.

KEMPrivateKeyOverEC (Initializer)		
parameters		None
in	EdwardsCurve $[0, \ldots, q-1]$	curve λ

We denote by

$$\mathsf{KEMPrivateKeyOverEC}(\mathsf{curve}, \lambda) \to \mathsf{sk}$$

the call to the KEMPrivateKeyOverEC initializer.

```
1: procedure KEMPrivateKeyOverEC(curve, λ)
2:     algoClassByteId ← 0x12
3:     algoImplemByteId ← algoImplemByteIdFromCurve(curve)
4:     PrivateKeyOverEC(algoClassByteId, algoImplemByteId, curve, λ)
5: end procedure
```

KEMOverEC.generateKeyPair		
parameters		None
in	PRNG EdwardsCurve	prng curve
out	KEMPublicKeyOverEC KEMPrivateKeyOverEC	pk sk

We denote by

$$\mathsf{KEMOverEC.generateKeyPair}(\mathsf{prng}, \mathsf{curve}) \rightarrow (\mathsf{pk}, \mathsf{sk})$$

the call to the KEMOverEC.generateKeyPair procedure.

procedure KEMOverEC.generateKeyPair(prng, curve)
    repeat $(\lambda, P) \leftarrow$ curve.generateRandomScalarAndPoint(prng) **until** not curve.isLowOrderPoint($P.y$)
    pk $\leftarrow$ KEMPublicKeyOverEC(curve, P)
    sk $\leftarrow$ KEMPrivateKeyOverEC(curve, λ)
    return (pk, sk)
end procedure

KEMOverEC.encrypt		
parameters		None
in	KEMPublicKeyOverEC PRNG SymmetricKey subtype	pk prng T
out	$\{0, \ldots, 255\}^*$ T	`encrypted_key` `key`

We denote by

$$\mathsf{KEMOverEC.encrypt}(\mathsf{pk}, \mathsf{prng}, \mathsf{T}) \rightarrow (\texttt{encrypted_key}, \texttt{key})$$

the procedure that produces a ciphertext `encrypted_key` containing a symmetric key `key` of type T, encrypted under the public key pk, using the initialized PRNG instance prng. The procedure works as follows (see [18, p.26]):

procedure KEMOverEC.encrypt(pk, prng, T)
    curve $\leftarrow$ pk.curve
    $(p, d, G, q, \nu,$ card$) \leftarrow$ curve.parameters
    if curve.isLowOrderPoint(pk.y) **then** return $\perp$ **end if**
    repeat $r \leftarrow$ prng.bigInt(q) **until** $r \neq 0$
    $B_y \leftarrow$ curve.scalarMultiplication($r, G.y$)
    $D_y \leftarrow$ curve.scalarMultiplication(r, pk.y)
    c $\leftarrow$ bytesFromBigUInt(B_y, len(p))
    seed $\leftarrow$ c$\|$bytesFromBigUInt(D_y, len(p))
    key $\leftarrow$ KDFFromPRNGWithHMACWithSHA256.compute(seed, T)
    return (c, key)
end procedure

KEMOverEC.decrypt		
parameters		None
in	$\{0,\dots,255\}^*$ KEMPrivateKeyOverEC SymmetricKey subtype	`encrypted_key` sk T
out	T	`key`

We denote by

$$\text{KEMOverEC.decrypt}(\texttt{encrypted_key}, \text{sk}, \text{T}) \to \texttt{key}$$

the procedure that decrypts the ciphertext `encrypted_key` containing a symmetric key `key` of type T, using the private key sk. The procedure works as follows (see [18, p.26]):

procedure KEMOverEC.decrypt(`encrypted_key`, sk, T)
  curve ← sk.curve
  $(p, d, G, q, \nu, \text{card})$ ← curve.parameters
  if len(`encrypted_key`) $\neq$ len(p) **then** return $\perp$ **end if**
  y ← bigUIntFromBytes(`encrypted_key`)
  B_y ← curve.scalarMultiplication(ν, y)
  if $B_y = 1$ **then** return $\perp$ **end if**
  a ← sk.scalar $\times$ ($\nu^{-1} \bmod q$) $\bmod q$
  D_y ← curve.scalarMultiplication(a, B_y)
  `seed` ← `encrypted_key`‖bytesFromBigUInt(D_y, len(p))
  return KDFFromPRNGWithHMACWithSHA256.compute(`seed`, T)
end procedure

KEMPublicKeyOverEC.expandCompactKey		
parameters		None
in	$\{0,\dots,255\}^*$	`compactKey`
out	KEMPublicKeyOverEC	pk

We denote by

$$\text{KEMPublicKeyOverEC.expandCompactKey}(\texttt{compactKey}) \to \text{KEMPublicKeyOverEC}$$

the procedure that recovers an instance pk of KEMPublicKeyOverEC given a compact key `compactKey`. The procedure works as follows:

procedure KEMPublicKeyOverEC.expandCompactKey(`compactKey`)
  if len(`compactKey`) $= 0$ **then** return $\perp$ **end if**
  Parse `compactKey` as `algoImplemByteId`‖`yCoordinate` where len(`algoImplemByteId`) $= 1$
  curve ← KEMPublicKeyOverEC.curveFromAlgoImplemByteId(`algoImplemByteId`)

5: **if** len(`compactKey`) $\neq 1 +$ len(curve.p) **then** return $\perp$ **end if**
6: $y \leftarrow$ bigUIntFromBytes(`yCoordinate`)
7: return KEMPublicKeyOverEC(curve, y)
8: **end procedure**

16.1 KEM over Curve25519

We denote by KEMOverCurve25519 the concrete subtype of KEMOverEC used to perform KEM operations over Curve25519.

KEMOverCurve25519.generateKeyPair		
parameters		None
in	PRNG	prng
out	KEMPublicKeyOverEC KEMPrivateKeyOverEC	pk sk

We denote by

$$\text{KEMOverCurve25519.generateKeyPair}(\text{prng}) \rightarrow (\text{pk}, \text{sk})$$

the call to the KEMOverCurve25519.generateKeyPair procedure.

1: **procedure** KEMOverCurve25519.generateKeyPair(prng)
2: curve $\leftarrow$ Curve25519()
3: return KEMOverEC.generateKeyPair(prng, curve)
4: **end procedure**

16.2 KEM over MDC

We denote by KEMOverMDC the concrete subtype of KEMOverEC used to perform KEM operations over MDC.

KEMOverMDC.generateKeyPair		
parameters		None
in	PRNG	prng
out	KEMPublicKeyOverEC KEMPrivateKeyOverEC	pk sk

We denote by

$$\text{KEMOverMDC.generateKeyPair}(\text{prng}) \rightarrow (\text{pk}, \text{sk})$$

the call to the KEMOverMDC.generateKeyPair procedure.

```
1: procedure KEMOverMDC.generateKeyPair(prng)
2:     curve ← MDC()
3:     return KEMOverEC.generateKeyPair(prng, curve)
4: end procedure
```

17 Public Key Encryption

Encrypting a message of arbitrary length under a public key is done by combining the KEM of the previous section with the authenticated encryption scheme AES256CTRHMACSHA256 of Section 11 (used as a DEM). We denote by PublicKeyEncryption this construction.

PublicKeyEncryption.encrypt		
parameters		None
in	KEMPublicKeyOverEC $\{0,\dots,255\}^*$ PRNG	pk m prng
out	$\{0,\dots,255\}^*$	c

We denote by

$$\text{PublicKeyEncryption.encrypt}(\text{pk}, \texttt{m}, \text{prng}) \to \text{c}$$

the procedure that encrypts the message m under the public key pk, using the initialized PRNG instance prng. The procedure works as follows:

```
1: procedure PublicKeyEncryption.encrypt(pk, m, prng)
2:     (encrypted_key, key) ← KEMOverEC.encrypt(pk, prng, AES256CTRHMACSHA256Key)
3:     if encrypted_key = ⊥ then return ⊥ end if
4:     c_0 ← AES256CTRHMACSHA256.encrypt(m, key, prng)
5:     return encrypted_key ‖ c_0
6: end procedure
```

The resulting ciphertext is thus $\text{len}(p) + 40$ bytes longer than the plaintext, that is, 72 bytes longer for both curves considered in these specifications.

PublicKeyEncryption.decrypt		
parameters		None
in	KEMPrivateKeyOverEC $\{0,\dots,255\}^*$	sk c
out	$\{0,\dots,255\}^*$	m

We denote by

$$\mathsf{PublicKeyEncryption.decrypt}(\mathsf{sk}, \mathsf{c}) \rightarrow \mathtt{m}$$

the procedure that decrypts the ciphertext c using the private key sk. The procedure works as follows:

```
1: procedure PublicKeyEncryption.decrypt(sk, c)
2:     (p, d, G, q, ν, card) ← sk.curve.parameters
3:     if len(c) < len(p) then return ⊥ end if
4:     Parse c as encrypted_key ‖ c_0 where len(encrypted_key) = len(p)
5:     key ← KEMOverEC.decrypt(encrypted_key, sk, AES256CTRHMACSHA256Key)
6:     if key = ⊥ then return ⊥ end if
7:     return AES256CTRHMACSHA256.decrypt(c_0, key)
8: end procedure
```

18 Cryptographic Identity

In Olvid, all communications occur between cryptographic identities. These identities are instances of a complex type denoted CryptoIdentity. An instance cryptoIdentity of CryptoIdentity gives access to three values:

- cryptoIdentity.serverURL: All messages sent to an identity pass through a server, identified by this URL.
- cryptoIdentity.publicKeyForAuthentication: This public key is used to check challenge responses computed by the owner of the associated private key (i.e., by the owner of the corresponding owned cryptographic identity, see Section 19). It is an instance of AuthenticationPublicKeyOverEC.
- cryptoIdentity.publicKeyForKEM: This public key is an instance of KEMPublicKeyOverEC and is used to perform a KEM encryption that can only be decrypted by the owner of the associated private key (i.e., the owner of the corresponding owned cryptographic identity, see Section 19).

CryptoIdentity (Initializer)		
parameters		None
in	$\{0,\dots,255\}^*$	serverURL
	AuthenticationPublicKeyOverEC	pk_a
	KEMPublicKeyOverEC	pk_e

Given a server URL serverURL, a public key for authentication pk_a, and a public key for KEM pk_e, the previous initializer returns an instance cryptoIdentity of CryptoIdentity. We denote by

$$\mathsf{CryptoIdentity}(\mathtt{serverURL}, \mathsf{pk}_a, \mathsf{pk}_e) \rightarrow \mathsf{cryptoIdentity}$$

the call to the previous initializer. This initializer works as follows:

```
1: procedure CryptoIdentity(serverURL, pk_a, pk_e)
```

```
2:     self.serverURL ← serverURL
3:     self.publicKeyForAuthentication ← pk_a
4:     self.publicKeyForKEM ← pk_e
5: end procedure
```

Within the rest of this section, we denote by cryptoIdentity an instance of CryptoIdentity.

cryptoIdentity.getIdentity		
parameters		None
in		
out	$\{0,\ldots,255\}^*$	`identity`

Given a cryptographic identity cryptoIdentity, the procedure getIdentity returns a byte-array representation of this cryptographic identity. In this document, this byte-array is what we call an *identity*. We denote by

$$\text{cryptoIdentity.getIdentity}() \rightarrow \texttt{identity}$$

the call to the cryptoIdentity.getIdentity procedure.

```
1: procedure cryptoIdentity.getIdentity()
2:     identity ← serverURL
3:     identity ← identity || 0x00
4:     identity ← identity || self.publicKeyForAuthentication.getCompactKey()
5:     identity ← identity || self.publicKeyForKEM.getCompactKey()
6:     return identity
7: end procedure
```

CryptoIdentity (Initializer)		
parameters		None
in	$\{0,\ldots,255\}^*$	`identity`

Given an identity `identity`, the previous initializer returns an instance cryptoIdentity of CryptoIdentity. We denote by

$$\text{CryptoIdentity}(\texttt{identity}) \rightarrow \text{cryptoIdentity}$$

the call to the previous initializer. This initializer works as follows:

```
1: procedure CryptoIdentity(identity)
2:     Parse identity as serverURL || 0x00 || keys. Return ⊥ if this fails.
3:     Check that serverURL is a valid URL
4:     if len(keys) = 0 then return ⊥ end if
5:     authImplemByteId ← keys[0]  // one byte
```

```
    curve_a ← AuthenticationOverEC.curveFromAlgoImplemByteId(authImplemByteId)
    ℓ_a ← len(curve_a.p)
    if len(keys) < 2 + ℓ_a then return ⊥ end if
    compactAuthKey ← keys[0 ... ℓ_a]  // 1 + ℓ_a bytes
    pk_a ← AuthenticationPublicKeyOverEC.expandCompactKey(compactAuthKey)
    kemImplemByteId ← keys[ℓ_a + 1]  // one byte
    curve_e ← KEMPublicKeyOverEC.curveFromAlgoImplemByteId(kemImplemByteId)
    ℓ_e ← len(curve_e.p)
    if len(keys) ≠ 2 + ℓ_a + ℓ_e then return ⊥ end if
    compactKEMKey ← keys[ℓ_a + 1 ... ℓ_a + ℓ_e + 1]  // 1 + ℓ_e bytes
    pk_e ← KEMPublicKeyOverEC.expandCompactKey(compactKEMKey)
    self.serverURL ← serverURL
    self.publicKeyForAuthentication ← pk_a
    self.publicKeyForKEM ← pk_e
end procedure
```

19 Owned Cryptographic Identity

A cryptographic identity is necessarily *owned* by a user. This user knows the private keys associated with the public keys presented in Section 18. The type OwnedCryptoIdentity is used to represent these owned identities. An instance ownedCryptoIdentity of OwnedCryptoIdentity gives access to six values:

- ownedCryptoIdentity.serverURL: All messages sent to an identity pass through a server, identified by this URL.
- ownedCryptoIdentity.publicKeyForAuthentication: This public key is used to check challenge responses computed by the owner of the associated private key (i.e., by the owner of the corresponding owned cryptographic identity, see Section 19). It is an instance of AuthenticationPublicKeyOverEC.
- ownedCryptoIdentity.privateKeyForAuthentication: Instance of AuthenticationPrivateKeyOverEC, this is the private key associated with the above public key.
- ownedCryptoIdentity.publicKeyForKEM: This public key is an instance of KEMPublicKeyOverEC and is used to perform a KEM encryption that can only be decrypted by the owner of the associated private key (i.e., the owner of the corresponding owned cryptographic identity, see Section 19).
- ownedCryptoIdentity.privateKeyForKEM: Instance of KEMPrivateKeyOverEC, this is the private key associated with the above public key.
- ownedCryptoIdentity.secretMACKey: Instance of MACKey.

OwnedCryptoIdentity (Initializer)		
parameters		None
in	$\{0,\dots,255\}^*$	`serverURL`
	AuthenticationPublicKeyOverEC	pk_a
	AuthenticationPrivateKeyOverEC	sk_a
	KEMPublicKeyOverEC	pk_e
	KEMPrivateKeyOverEC	sk_e
	MACKey	`key`

Given a server URL `serverURL`, a key pair for authentication $\mathsf{pk}_a/\mathsf{sk}_a$, a key pair for KEM $\mathsf{pk}_e/\mathsf{sk}_e$, and a secret MAC key `key`, the previous initializer returns an instance ownedCryptoIdentity of OwnedCryptoIdentity. We denote by

$$\mathsf{OwnedCryptoIdentity}(\texttt{serverURL}, \mathsf{pk}_a, \mathsf{sk}_a, \mathsf{pk}_e, \mathsf{sk}_e, \texttt{key}) \to \mathsf{ownedCryptoIdentity}$$

the call to the previous initializer. This initializer works as follows:

```
1: procedure OwnedCryptoIdentity(serverURL, pk_a, sk_a, pk_e, sk_e, key)
2:     self.serverURL ← serverURL
3:     self.publicKeyForAuthentication ← pk_a
4:     self.privateKeyForAuthentication ← sk_a
5:     self.publicKeyForKEM ← pk_e
6:     self.privateKeyForKEM ← sk_e
7:     self.secretMACKey ← key
8: end procedure
```

OwnedCryptoIdentity.generateOwnedCryptoIdentity		
parameters		None
in	$\{0,\dots,255\}^*$	`serverURL`
	PRNG	prng
out	OwnedCryptoIdentity	ownedCryptoIdentity

The previous procedure generates a fresh random owned identity, given a `serverURL` and an instance prng of PRNG. We denote by

$$\mathsf{OwnedCryptoIdentity.generateOwnedCryptoIdentity}(\texttt{serverURL}, \mathsf{prng}) \to \mathsf{ownedCryptoIdentity}$$

the call to the previous procedure.

```
1: procedure OwnedCryptoIdentity.generateOwnedCryptoIdentity(serverURL, prng)
2:     (pk_a, sk_a) ← AuthenticationOverMDC.generateKeyPair(prng)
3:     (pk_e, sk_e) ← KEMOverCurve25519.generateKeyPair(prng)
```

```
4:     key ← HMACWithSHA256.generateKey(prng)
5:     return OwnedCryptoIdentity(serverURL, pk_a, sk_a, pk_e, sk_e, key)
6: end procedure
```

ownedCryptoIdentity.getCryptoIdentity		
parameters		None
in		
out	CryptoIdentity	cryptoIdentity

Given an owned identity instance ownedCryptoIdentity, the previous procedure extracts the public information by returning an instance cryptoIdentity of CryptoIdentity. We denote by

$$\text{ownedCryptoIdentity.getCryptoIdentity}() \rightarrow \text{cryptoIdentity}$$

the call to the previous procedure.

```
1: procedure ownedCryptoIdentity.getCryptoIdentity()
2:     return CryptoIdentity(self.serverURL, self.pk_a, self.pk_e)
3: end procedure
```

Profile backup seed. In addition to its cryptographic keys, each owned identity is associated with a 20-byte *profile backup seed*, used to encrypt and authenticate the backups of this owned identity (see Part VII). This seed is generated when the owned identity is created, using prng.bytes(20). For owned identities created before this seed was introduced, the profile backup seed is instead derived deterministically from the secret MAC key: it consists of the first 20 bytes of

$$\text{SHA256}\big(\text{HMACWithSHA256.compute}(\texttt{0xCC}, \text{ownedCryptoIdentity.secretMACKey}) \,\|\, \texttt{"backupKey"}\big),$$

where "backupKey" is encoded in UTF-8.

19.1 Deterministic Seeds

Some protocols require the owner of an owned identity to generate random-looking values that can be recomputed identically later on. These values are generated with a PRNGWithHMACWithSHA256 instance seeded with a *deterministic seed*, computed from the secret MAC key of the owned identity, a one-byte tag identifying the context, and a diversification value div:

```
1: procedure ownedCryptoIdentity.deterministicSeed(tag, div)
2:     d ← HMACWithSHA256.compute(tag, self.secretMACKey)
3:     return SHA256(d ‖ div)
4: end procedure
```

The contexts in which such a seed is used are the following:

Tag	Context	Diversification value
0x55	SAS in the trust establishment protocol	received commitment
0x56	SAS in the owned identity transfer protocol	received commitment
0x57	encryption of return receipts	nonce ‖ encoded payload

Part III
Encodings

Several features of the Olvid engine require the serialization of data and objects as bytes, for example for storing strongly typed cryptographic keys in a database, or exchanging data with the server. As most of the data that requires encoding is in binary format, we developed our own binary-friendly encoding format which is described in this section.

20 Encoding Structure

In this part, we describe the rules for encoding data. The encoding structure is a basic type-length-value (TLV) encoding similar to ASN.1 or BSON.

Identifier (1 byte)	Content byte-length (4 bytes)	Content

20.1 Byte Identifiers

The Identifier octet specifies the type of the encoded object. The exhaustive list of all identifiers considered in these specifications is available in Table 2.

Table 2: List of all identifiers and corresponding number of bytes for determining the content length.

Identifier	Type	Section
`0x00`	Array of bytes	22.2
`0x01`	64-bit Integer	22.4
`0x02`	Boolean	22.5
`0x80`	Unsigned Big Integer	22.6
`0x03`	List	22.7
`0x04`	Dictionary	22.8
`0x90`	Symmetric key	22.12
`0x91`	Public key	22.13
`0x92`	Private key	22.14

20.2 Content Byte-Length

The content byte-length specifies the exact number of bytes of the content. This length is always coded on 4 bytes. We use a big-endian representation, and consider lengths as unsigned 32-bit

integers. Section 21.1 describes the algorithm used to compute this representation.

21 Byte Representation of Integers and Lengths

We first define a few helper procedures, used to represent unsigned 32-bit integers, signed 32-bit integers, unsigned 64-bit integers, signed 64-bit integers, unsigned big integers and signed big integers as an array of bytes.

21.1 32-bit Unsigned Integer

Given an unsigned 32-bit integer s, we denote by $\mathsf{bytesFromUInt32}(s) \in \{0,\dots,255\}^4$ the big-endian representation of s on 4 bytes. More precisely,

$$\mathsf{bytesFromUInt32}(s) = \begin{cases} [\lfloor s/256^3 \rfloor \bmod 256, \dots, \lfloor s/256 \rfloor \bmod 256, s \bmod 256] & \text{if } s \in [0, 2^{32}-1], \\ \bot & \text{otherwise.} \end{cases}$$

We denote by $\mathsf{uint32FromBytes}$ the inverse procedure of $\mathsf{bytesFromUInt32}$. Given a 4-byte array $\mathsf{b} = [\mathsf{b}_0, \mathsf{b}_1, \mathsf{b}_2, \mathsf{b}_3] \in \{0,\dots,255\}^4$, we have

$$\mathsf{uint32FromBytes}(\mathsf{b}) = \mathsf{b}_3 + \mathsf{b}_2 \times 256 + \mathsf{b}_1 \times 256^2 + \mathsf{b}_0 \times 256^3.$$

21.2 64-bit Unsigned Integer

Given an unsigned 64-bit integer s, we denote by $\mathsf{bytesFromUInt64}(s) \in \{0,\dots,255\}^8$ the big-endian representation of s on 8 bytes. More precisely,

$$\mathsf{bytesFromUInt64}(s) = \begin{cases} [\lfloor s/256^7 \rfloor \bmod 256, \dots, \lfloor s/256 \rfloor \bmod 256, s \bmod 256] & \text{if } s \in [0, 2^{64}-1], \\ \bot & \text{otherwise.} \end{cases}$$

We denote by $\mathsf{uint64FromBytes}$ the inverse procedure of $\mathsf{bytesFromUInt64}$. Given an 8-byte array $\mathsf{b} = [\mathsf{b}_0, \mathsf{b}_1, \dots, \mathsf{b}_7] \in \{0,\dots,255\}^8$, we have

$$\mathsf{uint64FromBytes}(\mathsf{b}) = \mathsf{b}_7 + \mathsf{b}_6 \times 256 + \mathsf{b}_5 \times 256^2 + \dots + \mathsf{b}_0 \times 256^7.$$

21.3 64-bit Signed Integer

Given a signed 64-bit integer s, we denote by $\mathsf{bytesFromInt64}(s) \in \{0,\dots,255\}^8$ the big-endian representation of s on 8 bytes. More precisely,

$$\mathsf{bytesFromInt64}(s) = \begin{cases} \mathsf{bytesFromUInt64}(s) & \text{if } s \in [0, 2^{63}-1], \\ \mathsf{bytesFromUInt64}(2^{64}+s) & \text{if } s \in [-2^{63}, -1], \\ \bot & \text{otherwise.} \end{cases}$$

We denote by $\mathsf{int64FromBytes}$ the inverse procedure of $\mathsf{bytesFromInt64}$. Given an 8-byte array $\mathsf{b} = [\mathsf{b}_0, \mathsf{b}_1, \dots, \mathsf{b}_7] \in \{0,\dots,255\}^8$, we have

$$\mathsf{int64FromBytes}(\mathsf{b}) = \mathsf{uint64FromBytes}(\mathsf{b}) - (\mathsf{b}_0 \texttt{ and } \texttt{0x80}) \ll 57.$$

21.4 Unsigned Big Integer

Given an unsigned integer s, we denote by $\mathsf{bytesFromBigUInt}(s, \ell) \in \{0, \ldots, 255\}^\ell$ the big-endian representation of s on ℓ bytes. More precisely, if $s \in [0, 2^{8\ell} - 1]$,

$$\mathsf{bytesFromBigUInt}(s, \ell) = \begin{cases} [\lfloor s/256^{\ell-1} \rfloor \bmod 256, \ldots, \lfloor s/256 \rfloor \bmod 256, s \bmod 256] & \text{if } 0 \leq s \leq 2^{8\ell} - 1 \\ \perp & \text{otherwise.} \end{cases}$$

We denote by $\mathsf{bigUIntFromBytes}$ the inverse procedure of $\mathsf{bytesFromBigUInt}$. Given an ℓ-byte array $\mathtt{b} = [\mathtt{b}_0, \mathtt{b}_1, \ldots, \mathtt{b}_{\ell-1}] \in \{0, \ldots, 255\}^\ell$, we have

$$\mathsf{bigUIntFromBytes}(\mathtt{b}) = \mathtt{b}_{\ell-1} + \mathtt{b}_{\ell-2} \times 256 + \cdots + \mathtt{b}_0 \times 256^{\ell-1}$$

when $\ell > 0$, and 0 otherwise.

21.5 Signed Big Integer

Given a signed integer s, we denote by $\mathsf{bytesFromBigInt}(s, \ell) \in \{0, \ldots, 255\}^\ell$ the big-endian representation of s on ℓ bytes. More precisely, if $s \in [-2^{8\ell-1}, 2^{8\ell-1} - 1]$,

$$\mathsf{bytesFromBigInt}(s, \ell) = \begin{cases} \mathsf{bytesFromBigUInt}(s, \ell) & \text{if } s \in [0, 2^{8\ell-1} - 1], \\ \mathsf{bytesFromBigUInt}(2^{8\ell} + s, \ell) & \text{if } s \in [-2^{8\ell-1}, -1], \\ \perp & \text{otherwise.} \end{cases}$$

We denote by $\mathsf{bigIntFromBytes}$ the inverse procedure of $\mathsf{bytesFromBigInt}$. Given an ℓ-byte array $\mathtt{b} = [\mathtt{b}_0, \mathtt{b}_1, \ldots, \mathtt{b}_{\ell-1}] \in \{0, \ldots, 255\}^\ell$, we have

$$\mathsf{bigIntFromBytes}(\mathtt{b}) = \mathsf{bigUIntFromBytes}(\mathtt{b}) - (\mathtt{b}_0 \texttt{ and } \texttt{0x80}) \ll (8\ell - 7)$$

when $\ell > 0$, and 0 otherwise.

21.6 Lengths

As stated in Section 20.2, all encodings include a 4-byte Content byte-length where lengths are considered as 32-bit unsigned integers. Given a content length $\ell \in [0, 2^{32} - 1]$, we populate the Content byte-length region using $\mathsf{bytesFromUInt32}(\ell)$. When decoding, this 4-byte Content byte-length $\mathtt{b} = [\mathtt{b}_0, \mathtt{b}_1, \mathtt{b}_2, \mathtt{b}_3] \in \{0, \ldots, 255\}^4$ is mapped to a 32-bit unsigned integer length using $\mathsf{uint32FromBytes}(\mathtt{b})$.

22 Encoding Procedures

In this section, we define several procedures used to encode, decode, and manipulate all the mathematical and abstract objects we consider in these specifications.

22.1 Common Encoding/Decoding Rules

As opposed to encoding procedures which cannot fail, the decoding procedures may all fail. All the decoding procedures defined in sections 22.2 to 22.14 execute the following parsing procedure on an input $[\mathsf{b}_0, \mathsf{b}_1, \ldots, \mathsf{b}_{n-1}]$ before trying to decode any further:

1: **procedure** Parse($[\mathsf{b}_0, \mathsf{b}_1, \ldots, \mathsf{b}_{n-1}]$)
2: **if** $n < 5$ **then** return $\perp$ **end if**
3: **if** b_0 is not a known byte identifier **then** return $\perp$ **end if**
4: $\ell \leftarrow$ uint32FromBytes($[\mathsf{b}_1, \mathsf{b}_2, \mathsf{b}_3, \mathsf{b}_4]$)
5: **if** $5 + \ell \neq n$ **then** return $\perp$ **end if**
6: return $(\mathsf{b}_0, [\mathsf{b}_5, \ldots, \mathsf{b}_{5+(\ell-1)}])$
7: **end procedure**

Note that all the known byte identifiers are listed in Table 2.

22.2 Encoding an Array of Bytes

The following procedure returns the encoding of an array of bytes $\mathtt{b} \in \{0, \ldots, 255\}^*$:

$$\mathsf{encodeBytes}(\mathtt{b}) \rightarrow [\mathtt{0x00}, \mathsf{bytesFromUInt32}(\mathsf{len}(\mathtt{b})), \mathtt{b}]$$

We use the byte identifier `0x00` for arrays of bytes. The total length of the encoding of $\mathtt{b} \in \{0, \ldots, 255\}^*$ is

$$\ell = 1 + 4 + \mathsf{len}(\mathtt{b}) = 5 + \mathsf{len}(\mathtt{b}).$$

In order to decode an array of $n \geq 0$ bytes $[\mathsf{b}_0, \mathsf{b}_1, \ldots, \mathsf{b}_{n-1}] \in \{0, \ldots, 255\}^n$ assumed to be the encoding of an array of bytes, we apply the following procedure:

1: **procedure** decodeBytes($[\mathsf{b}_0, \mathsf{b}_1, \ldots, \mathsf{b}_{n-1}]$)
2: $(\mathtt{byteId}, [\mathsf{c}_0, \mathsf{c}_1, \ldots, \mathsf{c}_{\ell-1}]) \leftarrow$ Parse($[\mathsf{b}_0, \mathsf{b}_1, \ldots, \mathsf{b}_{n-1}]$)
3: **if** $\mathtt{byteId} \neq \mathtt{0x00}$ **then** return $\perp$ **end if**
4: return $[\mathsf{c}_0, \mathsf{c}_1, \ldots, \mathsf{c}_{\ell-1}]$
5: **end procedure**

22.3 Encoding a String

Strings can be directly encoded and decoded by transforming them into byte arrays using UTF-8 encoding and using the array of bytes encoding from the previous section.

$$\mathsf{encodeString}(\mathtt{s}) \rightarrow \mathsf{encodeBytes}(\mathsf{stringToBytes}(\mathtt{s}, \texttt{"utf-8"}))$$
$$\mathsf{decodeString}(\mathtt{b}) \rightarrow \mathsf{bytesToString}(\mathsf{decodeBytes}(\mathtt{b}), \texttt{"utf-8"})$$

22.4 Encoding a 64-bit Integer

The following procedure returns the encoding of a 64-bit integer s:

$$\mathsf{encodeInt}(s) \to [\texttt{0x01}, \mathsf{bytesFromUInt32}(8), \mathsf{bytesFromInt64}(s)]$$

We use the identifier `0x01` for 64-bit integers. The encoding of a 64-bit integer using the above procedure has a total length of

$$1 + 4 + 8 = 13.$$

In order to decode an array of $n \geq 0$ bytes $[\mathsf{b}_0, \mathsf{b}_1, \ldots, \mathsf{b}_{n-1}] \in \{0, \ldots, 255\}^n$ assumed to be the encoding of a 64-bit integer, we apply the following procedure:

```
1: procedure decodeInt([b_0, b_1, ..., b_{n-1}])
2:     (byteId, [c_0, c_1, ..., c_{ℓ-1}]) ← Parse([b_0, b_1, ..., b_{n-1}])
3:     if byteId ≠ 0x01 then return ⊥ end if
4:     if ℓ ≠ 8 then return ⊥ end if
5:     return int64FromBytes([c_0, c_1, ..., c_7])
6: end procedure
```

22.5 Encoding a Boolean

The following procedure returns the encoding of a Boolean value $s \in \{\texttt{true}, \texttt{false}\}$:

$$\mathsf{encodeBool}(s) \to [\texttt{0x02}, \mathsf{bytesFromUInt32}(1), s \ ?\ \texttt{0x01} : \texttt{0x00}]$$

We use the identifier `0x02` for Booleans. The encoding of a Boolean using the above procedure has a total length of

$$1 + 4 + 1 = 6.$$

In order to decode an array of $n \geq 0$ bytes $[\mathsf{b}_0, \mathsf{b}_1, \ldots, \mathsf{b}_{n-1}] \in \{0, \ldots, 255\}^n$ assumed to be the encoding of a Boolean, we apply the following procedure:

```
1: procedure decodeBool([b_0, b_1, ..., b_{n-1}])
2:     (byteId, [c_0, c_1, ..., c_{ℓ-1}]) ← Parse([b_0, b_1, ..., b_{n-1}])
3:     if byteId ≠ 0x02 then return ⊥ end if
4:     if ℓ ≠ 1 then return ⊥ end if
5:     if c_0 ∉ {0x00, 0x01} then return ⊥ end if
6:     return c_0 = 0x01
7: end procedure
```

22.6 Encoding an Unsigned Big Integer

The following procedure returns the encoding of an unsigned big integer $s \in \mathbb{N}$ on ℓ bytes:

$$\mathsf{encodeBigUInt}(s, \ell) \to [\texttt{0x80}, \mathsf{bytesFromUInt32}(\ell), \mathsf{bytesFromBigUInt}(s, \ell)] \tag{1}$$

Note that encodeBigUInt returns $\perp$ if any internal call returns $\perp$. We use the identifier `0x80` for unsigned big integers. The encoding of an unsigned big integer using the above procedure has a total length of

$$1 + 4 + \ell = 5 + \ell.$$

In order to decode an array of $n \geq 0$ bytes $[\mathsf{b}_0, \mathsf{b}_1, \ldots, \mathsf{b}_{n-1}] \in \{0, \ldots, 255\}^n$ assumed to be the encoding of an unsigned big integer, we apply the following procedure:

```
1: procedure decodeBigUInt([b_0, b_1, ..., b_{n-1}])
2:     (byteId, [c_0, ..., c_{ℓ-1}]) ← Parse([b_0, b_1, ..., b_{n-1}])
3:     if byteId ≠ 0x80 then return ⊥ end if
4:     return bigUIntFromBytes([c_0, ..., c_{ℓ-1}])
5: end procedure
```

22.7 Packing Elements and Encoding a List

The following pack procedure is used to pack several encoded elements and to specify a byte type `byteId` for the resulting container, which is itself an encoded element. Given an arbitrary number of encoded values `encoded_val1`, `encoded_val2`, $\ldots \in \{0, \ldots, 255\}^*$, the pack procedure is defined as follows:

$$\begin{aligned} &\mathsf{pack}(\texttt{byteId}, \texttt{encoded_val1}, \texttt{encoded_val2}, \ldots) \\ &\qquad \rightarrow [\texttt{byteId}, \mathsf{bytesFromUInt32}(\ell), \texttt{encoded_val1}, \texttt{encoded_val2}, \ldots] \end{aligned} \tag{2}$$

where ℓ is the total length of the list content, i.e.,

$$\ell = \mathsf{len}(\texttt{encoded_val1}) + \mathsf{len}(\texttt{encoded_val2}) + \cdots$$

In order to unpack an array of $n > 0$ bytes $[\mathsf{b}_0, \mathsf{b}_1, \ldots, \mathsf{b}_{n-1}] \in \{0, \ldots, 255\}^n$ assumed to be the result of the above pack procedure, we apply the following procedure:

```
1: procedure unpack([b_0, b_1, ..., b_{n-1}])
2:     (byteId, innerData) ← Parse([b_0, b_1, ..., b_{n-1}])
3:     encoded_vals = []
4:     while len(innerData) > 0 do
5:         (encoded_val, innerData) ← extractFirstEncodedValue(innerData)
6:         Append encoded_val to encoded_vals
7:     end while
8:     return (byteId, encoded_vals)
9: end procedure
```

where the extractFirstEncodedValue procedure removes the first encoded value it finds in its argument and returns this element and the remaining bytes, as follows:

```
1: procedure extractFirstEncodedValue([b_0, b_1, ..., b_{n-1}])
2:     if n < 5 then return ⊥ end if
3:     ℓ ← uint32FromBytes([b_1, b_2, b_3, b_4])
4:     if n < 5 + ℓ then return ⊥ end if
5:     return ([b_0, ..., b_{5+(ℓ-1)}], [b_{5+ℓ}, ..., b_{n-1}])
6: end procedure
```

We provide a specific procedure to encode a list of elements, based on the above pack procedure. The following procedure returns the encoded list of an arbitrary number of encoded values

encoded_val1, encoded_val2, ... $\in \{0, \ldots, 255\}^*$:

$$\begin{aligned} &\text{encodeList}(\texttt{encoded_val1}, \texttt{encoded_val2}, \ldots) \\ &\qquad \rightarrow \text{pack}(\texttt{0x03}, \texttt{encoded_val1}, \texttt{encoded_val2}, \ldots) \end{aligned} \tag{3}$$

We use the identifier 0x03 for lists.

In order to decode an array of $n \geq 0$ bytes $[\texttt{b}_0, \texttt{b}_1, \ldots, \texttt{b}_{n-1}] \in \{0, \ldots, 255\}^n$ assumed to be the encoding of a list, we apply the following procedure:

```
1: procedure decodeList([b_0, b_1, ..., b_{n-1}])
2:     (byteId, encoded_vals) ← unpack([b_0, b_1, ..., b_{n-1}])
3:     if byteId ≠ 0x03 then return ⊥ end if
4:     return encoded_vals
5: end procedure
```

22.8 Encoding a Dictionary

In most programming languages, a dictionary is an associative array where each element is associated with a string key. We adopt a more restrictive approach in these specifications where we consider that a dictionary is an associative array where a key is a byte array representing the UTF-8 encoding of a string, and where a value is an array of bytes representing a proper encoding of some value.

Given a dictionary dict, the action of associating the value encoded_val $\in \{0, \ldots, 255\}^*$ with the key key $\in \{0, \ldots, 255\}^*$ is denoted

$$\text{dict}[\texttt{key}] \leftarrow \texttt{encoded_val}.$$

Recovering the encoded value encoded_val is denoted

$$\text{dict}[\texttt{key}] \rightarrow \texttt{encoded_val} \quad \text{or} \quad \texttt{encoded_val} \leftarrow \text{dict}[\texttt{key}].$$

The following procedure returns a proper encoding of a dictionary dict respectively associating the keys key1, key2, ... with the encoded values encoded_val1, encoded_val2, ... :

$$\begin{aligned} &\text{encodeDictionary}(\text{dict}) \\ &\quad \rightarrow \text{pack}(\texttt{0x04}, \text{encodeBytes}(\texttt{key1}), \texttt{encoded_val1}, \text{encodeBytes}(\texttt{key2}), \texttt{encoded_val2}, \ldots) \end{aligned} \tag{4}$$

In most programming languages, dictionaries use hash tables and the order of the keys is not deterministic. As a consequence this encoding is not deterministic. Checking the equality of two encoded dictionaries requires decoding them and comparing the keys and their values (recursively).

In order to decode an array of $n \geq 0$ bytes $[\texttt{b}_0, \texttt{b}_1, \ldots, \texttt{b}_{n-1}] \in \{0, \ldots, 255\}^n$ assumed to be the encoding of a dictionary, we apply the following procedure:

```
1: procedure decodeDictionary([b_0, b_1, ..., b_{n-1}])
2:     (byteId, encoded_vals) ← unpack([b_0, b_1, ..., b_{n-1}])
3:     if byteId ≠ 0x04 then return ⊥ end if
```

```
    if len(encoded_vals) is not even then return ⊥ end if
    dict ← empty dictionary
    while len(encoded_vals) > 0 do
        (encoded_key, val) ← pop the first two elements of encoded_vals
        key ← decodeBytes(encoded_key)
        dict[key] ← val
    end while
    return dict
end procedure
```

22.9 Padded Encodings

Sometimes, in order to hide the length of an encoded content, it is padded to a fixed length (or to a multiple of a fixed length) before being encrypted. To pad an encoded value, simply encode it using the standard encoding procedure and append some bytes (typically zeros) to the output. The resulting byte array is no longer a valid encoding, as its length does not match the content byte-length. To decode it, we use the following decodeWithPadding procedure which discards the padding bytes before calling the standard decoding procedure decode corresponding to the expected type of the encoded value (e.g., decodeList or decodeDictionary):

```
procedure decodeWithPadding([b_0, b_1, ..., b_{n-1}], decode)
    if n < 5 then return ⊥ end if
    ℓ ← uint32FromBytes([b_1, b_2, b_3, b_4])
    if 5 + ℓ > n then return ⊥ end if
    return decode([b_0, b_1, ..., b_{5+(ℓ-1)}])
end procedure
```

Note that only the outermost encoded value may be padded: the encoded values it contains must be well-formed.

22.10 Encoding Other Types

Several other types of values are exchanged between the components of Olvid (in protocol messages or with the server). They are encoded using the encodings above, according to the following conventions:

- **Integers.** Whenever this document mentions an `int` (in particular in the description of the server API in Part XII), it is a 64-bit signed integer encoded with encodeInt.
- **Timestamps.** Timestamps are integers representing a number of milliseconds since the Unix epoch (1970-01-01 00:00:00 UTC), encoded with encodeInt.
- **UIDs.** UIDs (unique identifiers, such as device UIDs, message UIDs, protocol instance UIDs, etc.) are 32-byte arrays, encoded with encodeBytes.
- **UUIDs.** UUIDs (for example API keys) are encoded with encodeString applied to their canonical textual representation: 32 hexadecimal digits, in groups of 8-4-4-4-12 separated by hyphens. Hexadecimal digits may be upper or lower case.
- **Identities.** An identity (see Section 18) is encoded with encodeBytes applied to its byte-array representation.
- **Ciphertexts and seeds.** Ciphertexts and seeds are encoded with encodeBytes.

22.11 Encoding a Cryptographic Key

As explained in Section 3.2, cryptographic keys are not always considered as a "simple" byte array within these specifications. Instead they are instances of a Key class that provides convenience values that make it possible to provide robust encoding/decoding procedures. See Section 3.2 for more details.

All cryptographic keys are encoded in the exact same way. The following procedure returns a proper encoding of a cryptographic key key of type Key:

```
1: procedure encodeKey(key)
2:     encoded_byteIds ← encodeBytes([key.algoClassByteId, key.algoImplemByteId])
3:     encoded_dict ← encodeDictionary(key.dict)
4:     return pack(key.encodingByteId, encoded_byteIds, encoded_dict)
5: end procedure
```

In order to decode an array of $n \geq 0$ bytes $[\mathtt{b}_0, \mathtt{b}_1, \ldots, \mathtt{b}_{n-1}] \in \{0, \ldots, 255\}^n$ assumed to be the encoding of a cryptographic key for a specific KeyedAlgo, the following procedure will be systematically used within the decoding of a cryptographic key:

```
1: procedure preDecodeKey([b_0, b_1, ..., b_{n-1}])
2:     (byteId, encoded_vals) ← unpack([b_0, b_1, ..., b_{n-1}])
3:     if len(encoded_vals) ≠ 2 then return ⊥ end if
4:     byteIds ← decodeBytes(encoded_vals[0])
5:     if len(byteIds) ≠ 2 then return ⊥ end if
6:     algoClassByteId ← byteIds[0]
7:     algoImplemByteId ← byteIds[1]
8:     dict ← decodeDictionary(encoded_vals[1])
9:     return (algoClassByteId, algoImplemByteId, dict, byteId)
10: end procedure
```

22.12 Encoding a Symmetric Key

A symmetric key is a particular cryptographic key and is an instance of the type SymKey (see Section 3.2.1). The following procedure returns a proper encoding of a symmetric key symKey, instance of SymKey:

$$\mathsf{encodeSymKey}(\mathsf{symKey}) \rightarrow \mathsf{encodeKey}(\mathsf{symKey})$$

Note that symKey.encodingByteId = 0x90 for symmetric keys.

Within these specifications, a symmetric keyed cryptographic algorithm SymKeyedAlgo (such as a block cipher, a MAC, etc.) always provides an initializer SymKeyInit that, given an algorithm class byte identifier, an algorithm implementation byte identifier and a dictionary, returns a symmetric key. This initializer fails, e.g., when the dictionary is not appropriate.

In order to decode an array of $n \geq 0$ bytes $[\mathtt{b}_0, \mathtt{b}_1, \ldots, \mathtt{b}_{n-1}] \in \{0, \ldots, 255\}^n$ assumed to be the encoding of a symmetric key for a specific symmetric key algorithm SymKeyedAlgo, we apply the following procedure:

```
1: procedure decodeSymKey([b_0, b_1, ..., b_{n-1}])
```

2: $(\texttt{algoClassByteId}, \texttt{algoImplemByteId}, \texttt{dict}, \texttt{byteId}) \leftarrow \mathsf{preDecodeKey}([\texttt{b}_0, \texttt{b}_1, \ldots, \texttt{b}_{n-1}])$
3: **if** $\texttt{byteId} \neq \texttt{0x90}$ **then** return $\perp$ **end if**
4: return $\mathsf{SymKeyedAlgo.SymKeyInit}(\texttt{algoClassByteId}, \texttt{algoImplemByteId}, \texttt{dict})$
5: **end procedure**

22.13 Encoding a Public Key

A public key is a particular cryptographic key and is an instance of PubKey (see Section 3.2.2). The following procedure returns a proper encoding of a public key pubKey:

$$\mathsf{encodePubKey}(\mathsf{pubKey}) \rightarrow \mathsf{encodeKey}(\mathsf{pubKey})$$

Note that $\texttt{pubKey.encodingByteId} = \texttt{0x91}$ for public keys.

Within these specifications, a public key cryptographic algorithm PubKeyedAlgo (such as a digital signature scheme, a public key encryption scheme, etc.) always provides an initializer PubKeyInit that, given an algorithm class byte identifier, an algorithm implementation byte identifier and a dictionary, returns a public key. This initializer fails, e.g., when the dictionary is not appropriate.

In order to decode an array of $n \geq 0$ bytes $[\texttt{b}_0, \texttt{b}_1, \ldots, \texttt{b}_{n-1}] \in \{0, \ldots, 255\}^n$ assumed to be the encoding of a public key for a specific public key cryptographic algorithm PubKeyedAlgo, we apply the following procedure:

1: **procedure** $\mathsf{decodePubKey}([\texttt{b}_0, \texttt{b}_1, \ldots, \texttt{b}_{n-1}])$
2: $(\texttt{algoClassByteId}, \texttt{algoImplemByteId}, \texttt{dict}, \texttt{byteId}) \leftarrow \mathsf{preDecodeKey}([\texttt{b}_0, \texttt{b}_1, \ldots, \texttt{b}_{n-1}])$
3: **if** $\texttt{byteId} \neq \texttt{0x91}$ **then** return $\perp$ **end if**
4: return $\mathsf{PubKeyedAlgo.PubKeyInit}(\texttt{algoClassByteId}, \texttt{algoImplemByteId}, \texttt{dict})$
5: **end procedure**

22.14 Encoding a Private Key

A private key is a particular cryptographic key and is an instance of PrivKey (see Section 3.2.3). The following procedure returns a proper encoding of a private key privKey:

$$\mathsf{encodePrivKey}(\mathsf{privKey}) \rightarrow \mathsf{encodeKey}(\mathsf{privKey})$$

Note that $\texttt{privKey.encodingByteId} = \texttt{0x92}$ for private keys.

Within these specifications, a public key cryptographic algorithm PubKeyedAlgo (such as a digital signature scheme, a public key encryption scheme, etc.) always provides an initializer PrivKeyInit that, given an algorithm class byte identifier, an algorithm implementation byte identifier and a dictionary, returns a private key. This initializer fails, e.g., when the dictionary is not appropriate.

In order to decode an array of $n \geq 0$ bytes $[\texttt{b}_0, \texttt{b}_1, \ldots, \texttt{b}_{n-1}] \in \{0, \ldots, 255\}^n$ assumed to be the encoding of a private key for a specific public key cryptographic algorithm PubKeyedAlgo (see Section 22.13), we apply the following procedure:

1: **procedure** $\mathsf{decodePrivKey}([\texttt{b}_0, \texttt{b}_1, \ldots, \texttt{b}_{n-1}])$

2: $(\texttt{algoClassByteId}, \texttt{algoImplemByteId}, \texttt{dict}, \texttt{byteId}) \leftarrow \mathsf{preDecodeKey}([\mathtt{b}_0, \mathtt{b}_1, \ldots, \mathtt{b}_{n-1}])$
3: **if** $\texttt{byteId} \neq \texttt{0x92}$ **then** return $\perp$ **end if**
4: return $\mathsf{PubKeyedAlgo.PrivKeyInit}(\texttt{algoClassByteId}, \texttt{algoImplemByteId}, \texttt{dict})$
5: **end procedure**

Part IV
Message Structure and Communication Channels

There are two main categories of messages in Olvid:

- Application messages, containing the text messages and attachments the users exchange using the application
- Protocol messages, which are internal messages, usually not displayed to the user, sent directly by the protocol engine (see Part V)

Application messages. They are the only messages to have attachments and are normally sent through what we call Oblivious Channels, but can be sent through Pre-Key Channels if an Oblivious Channel is not yet available. In Olvid, an Oblivious Channel is a secure channel between two devices (the creation of such a channel is described in the channel creation protocol of Section 32), using symmetric encryption with 1-time keys (see Section 24.1 for encryption details). A Pre-Key Channel uses a pre-key, uploaded to the server by the recipient's device, for encryption (see Section 25). An application message can then be sent to:

- either all the devices of a single user (in the case of one-to-one discussions),
- or all the devices of multiple users (in the case of group discussions),
- or a single specific device of a user (for example for WebRTC call signaling messages). If this device is not known, the message is sent to all the devices of the user.

Protocol messages. They can be sent through a variety of channels: network channels (of course), but also a variety of "local" channels where the message never leaves the device. Typical examples are dialog messages which prompt the user with an accept / reject dialog (like when receiving an invitation) or where the user can enter an input (like during the SAS exchange). We will not discuss these local messages here and only focus on messages which are sent through network channels and thus need to be encrypted.

Network protocol messages can be sent through:

- an asymmetric channel, either in broadcast to an `identity`, or in unicast or multicast to one or several devices of the same user,
- an Oblivious Channel, either in unicast to one specific device, or in multicast to multiple devices of an `identity`.
- a Pre-Key Channel, either in unicast to one specific device, or in multicast to multiple devices of an `identity`.

23 General Message Structure

A message is composed of up to four different parts:

- a message header for each device this message is sent to,
- the message itself, containing the protocol message payload, or the application message text and attachment keys and metadata,
- optionally for application messages, an extended message payload (for small previews of image attachments)
- attachments, which are sent/received after the message is uploaded/downloaded.

Message headers only contain a wrapped key. The way this key is wrapped depends on the type of channel the message is sent on (oblivious channel, pre-key channel, or asymmetric channel), and the key itself is used to encrypt the message. This way, the message (and attachments) can be uploaded once while still being delivered to multiple users: one header will be uploaded for each destination device. As headers are small (about 185 bytes, or about 450 bytes for pre-key channels), sending a message to a large number of devices is possible, but this structure is not well suited for very large groups.

When sending a message to multiple users (for example, a group message), different headers will be associated with different `identity`, but the message and attachment can still be uploaded only once. One exception to this is when the users are on different servers: in this case the message will be uploaded once to each server.

The extended message payload (if any) is encrypted using a key derived from the message key, so no additional key is transmitted when such an extended payload is available.

Attachments are split into chunks before being sent. See Section 24.3 for more details.

A (decrypted) message always has the same structure: it is an encoded list of:

- an integer identifying the message type (protocol or application),
- an encoded list of elements (the elements depend on the type).

The following sections detail the structure of this encoded list of elements.

23.1 Protocol Message Structure

For a protocol message, the encoded list of elements contains 4 elements:

- an integer identifying the protocol ID (see Table 3),
- a 32-byte unique protocol instance identifier corresponding to the `protocolUid`,
- an integer identifying the protocol message ID (each protocol has its own set of message IDs),
- an encoded list of message inputs.

The first 3 elements are used by the protocol engine to identify which protocol step to run, and which internal protocol state to recover. The encoded list of message inputs is the effective serialized payload of the protocol message, which is given as input to the protocol step being run.

23.2 Application Message Structure

For an application message, the encoded list of elements contains 1 more element than the number of attachments (so only 1 element for messages without attachments). So for a message with n

attachments, this is:

- n encoded lists, each containing:
  - an encoded authenticated encryption key, used to encrypt/decrypt the attachment
  - the encoded `byte[]` of the UTF-8 serialized attachment metadata JSON (see below).
- the encoded `byte[]` of the UTF-8 serialized message payload JSON (see below).

Attachment metadata JSON:

```
{
  "type": String,   // attachment MIME type
  "file_name": String,
  "sha256": byte[]  // attachment SHA256 hash
}
```

Message payload JSON:

```
{
  "message": <Message content>,
  "rr": <Return receipt>,
  "rtc": <WebRTC message>,
  "settings": <Ephemeral message settings>,
  "qss": <Query shared settings>,
  "upm": <Message update request>,
  "delm": <Messages delete request>,
  "deld": <Discussion delete request>,
  "reacm": <Reaction>,
  "scd": <Screen capture detection>,
  "lvo": <Limited visibility message opened>,
  "dr": <Discussion read>,
  "pvm": <Poll vote>,
  "ht": <History transfer message>,
  "htc": <History transfer control>
}
```

`rr` and `message` are present for all user messages (text, attachments, location, poll). `rtc` is set for WebRTC signaling messages. `settings` messages allow setting the shared default ephemeral message settings for a discussion. `qss` allows querying another user for the shared settings of a discussion. `upm` is used when modifying a message after it was sent. `delm` and `deld` are used to "delete everywhere" a message or discussion. `reacm` is used when reacting to a message. `scd` is sent (from iOS and macOS devices only) when a screen capture of a limited visibility message is detected. `lvo` and `dr` are multi-device synchronization messages used to keep opened/read messages in sync on all devices. `pvm` is used to vote in a poll. `ht` and `htc` are used to establish a message history transfer between two devices of the same owned identity (see Part IX).

Message content:

```
{
  "body": String,   // message text
  "ssn": int,       // sender sequence number
  "sti": UUID,      // sender thread identifier
  "guid": byte[],   // group identifier
  "go": byte[],     // group owner identity
  "gid2": byte[],   // group v2 identifier
  "fw": boolean,    // forwarded message flag
  "ost": int,       // original server timestamp
  "o2oi": <One-to-one discussion identifier>,
  "re": <Message reference>, // for reply
  "exp": <Message expiration>,
  "loc": <Location information>,
  "um": [<User mention>],
  "p": <Poll>
}
```

`guid` and `go` allow determining which group discussion this message is part of. `gid2` does the same for groups v2. `ost` is used when resending a message in a group v2 after a member joins the group: the original server timestamp of the message (in milliseconds since Epoch) is included to allow this new member to properly sort the messages. Message reply is present if the message is a reply to another message. Location information is only included in location messages. `um` is an array of user mention objects which refer to a range of the message body corresponding to a user mention. Poll is only included in poll messages.

One-to-one discussion identifier:

```
[
  byte[],
  byte[]
]
```

For one-to-one discussions, this is an array containing the `identity` of both sender and receiver, making it possible to determine which discussion this is when receiving a message sent from another owned device in a multi-device context.

Message reference:

```
{
  "ssn": int,       // sender sequence number
  "sti": UUID,      // sender thread identifier
  "si": byte[]      // sender identifier
}
```

These elements allow referencing a message. This structure is used for replies, message updates, remote deletion, reactions, poll votes, and limited visibility message opening.

Message expiration:

```
{
  "ex": int,        // existence duration (seconds)
  "vis": int,       // visibility duration (seconds)
  "ro": boolean     // read-once message
}
```

Read-once and limited visibility duration messages are wiped after being seen once, or a certain time after being seen. Limited existence duration is relative to the timestamp at which a message was posted on the server.

Location information:

```
{
  "t": int,          // location message type
  "ts": int,         // location timestamp (ms)
  "lat": Float,      // latitude
  "long": Float,     // longitude
  "alt": Float,      // altitude (in meters)
  "prec": Float,     // precision (in meters)
  "add": String,     // address
  "c": int,          // location update counter
  "q": int,          // location sharing quality
  "se": int          // sharing expiration (ms)
}
```

The location message type is one of SEND (1), SHARING (2), or END_SHARING (3). Location timestamp is either the GPS timestamp (for locations received from the GPS) or the local timestamp (when sending a pin position). Altitude and precision are optional and only available for location information from the GPS, not when sending a pin position. Address is the address corresponding to a position sent (only available for SEND type messages). Counter, quality and expiration are related to SHARING type messages: the counter is incremented with each location update being sent, the quality is one of PRECISE (1), BALANCED (2), or POWER_SAVE (3), and the expiration is the timestamp (in milliseconds since Epoch) at which the sharing ends.

User mention:

```
{
  "uid": byte[],     // identity of the contact
  "rs": int,         // range start
  "re": int          // range end
}
```

Range start and end are measured in characters for a UTF-16 encoded String. Range start is inclusive, range end is exclusive.

Poll:

```
{
  "t": String,       // answer type
  "q": String,       // question
  "c": [<Poll candidate>],
  "m": boolean,      // multiple choice
  "e": int           // expiration (ms)
}
```

The answer type is currently always `"string"`. The expiration is optional: it is the timestamp (in milliseconds since Epoch) after which votes are no longer possible. The body of a poll message contains a text version of the poll, for applications that do not support polls.

Poll candidate:

```
{
  "uuid": UUID,      // candidate identifier
  "t": String        // candidate text
}
```

Return receipt:

```
{
  "nonce": byte[],
  "key": byte[]
}
```

WebRTC message:

```
{
  "ci": UUID,        // call identifier
  "mt": int,         // message type
  "smp": String      // serialized message payload
}
```

The message payload is parsed by the WebRTC service and may contain all sorts of information: SDPs, ICE candidates, relay messages for call participants, etc.

Ephemeral message settings:

```
{
  "version": int,    // settings version
  "guid": byte[],    // group identifier
  "go": byte[],      // group owner identity
  "gid2": byte[],    // group v2 identifier
  "o2oi": <One-to-one discussion identifier>,
  "exp": <Message expiration>
}
```

As for messages, `guid` and `go`, `gid2`, and `o2oi` allow determining which discussion these settings should be applied to. Message expiration defines the default expiration setting for following messages.

Query shared settings:

```
{
  "guid": byte[],    // group identifier
  "go": byte[],      // group owner identity
  "gid2": byte[],    // group v2 identifier
  "o2oi": <One-to-one discussion identifier>,
  "ksv": int,        // known settings version
  "exp": <Message expiration> // known settings
}
```

As for messages, `guid` and `go`, `gid2`, and `o2oi` allow determining which discussion these settings should be applied to. When receiving such a message, a user checks the discussion shared settings and makes sure the version and known settings match. If not, a shared settings update message may be sent in return.

Message update request:

```
{
  "body": String,    // updated message body
  "guid": byte[],    // group identifier
  "go": byte[],      // group owner identity
  "gid2": byte[],    // group v2 identifier
  "o2oi": <One-to-one discussion identifier>,
  "ref": <Message reference>,
  "loc": <Location information>,
  "um": [<User mention>]
}
```

This message is used to update the content of a message after it was sent. There is a reference to a discussion and to a message inside that discussion. The message body is replaced by the new body, mentions are updated, and for location messages, this is used to update a SHARING type message or to notify the end of the sharing with an END_SHARING update.

Messages delete request:

```
{
  "guid": byte[],    // group identifier
  "go": byte[],      // group owner identity
  "gid2": byte[],    // group v2 identifier
  "o2oi": <One-to-one discussion identifier>,
  "refs": [<Message reference>]
}
```

This message is used to delete messages after they were sent. There is a reference to a discussion and an array of message references inside that discussion. Only messages inside a single discussion can be deleted this way.

Discussion delete request:

```
{
  "guid": byte[],    // group identifier
  "go": byte[],      // group owner identity
  "gid2": byte[],    // group v2 identifier
  "o2oi": <One-to-one discussion identifier>
}
```

This message is used to delete a whole discussion on all participants' devices.

Reaction:
```
{
  "reac": String,   // the reaction
  "guid": byte[],   // group identifier
  "go": byte[],     // group owner identity
  "gid2": byte[],   // group v2 identifier
  "o2oi": <One-to-one discussion identifier>,
  "ref": <Message reference>,
  "ost": int        // original server timestamp
}
```
This message adds a reaction (typically an emoji) to a message referenced by a discussion and a message reference, or removes the previous reaction of the sender if `reac` is absent. The message structure itself does not impose any constraint on the `reac` string nature or length. As for messages, `ost` is used when resending reactions in a group v2 after a member joins the group.

Screen capture detection:
```
{
  "guid": byte[],   // group identifier
  "go": byte[],     // group owner identity
  "gid2": byte[],   // group v2 identifier
  "o2oi": <One-to-one discussion identifier>
}
```
This message references a discussion in which the screen capture was made.

Limited visibility message opened:
```
{
  "guid": byte[],   // group identifier
  "go": byte[],     // group owner identity
  "gid2": byte[],   // group v2 identifier
  "o2oi": <One-to-one discussion identifier>,
  "m": <Message reference>
}
```
This message is sent to other owned devices when a limited visibility message (either read-once, or with a visibility duration) is opened on one device. It prevents a read-once message from being viewed once on each device!

Discussion read:
```
{
  "tim": int,       // last message timestamp (ms)
  "guid": byte[],   // group identifier
  "go": byte[],     // group owner identity
  "gid2": byte[],   // group v2 identifier
  "o2oi": <One-to-one discussion identifier>
}
```
This message is sent to other owned devices when messages are read in a discussion. It allows synchronizing the unread messages in multi-device. The timestamp is the server timestamp of the most recent message read in the discussion.

Poll vote:
```
{
  "pcuid": UUID,    // poll candidate identifier
  "v": boolean,     // voted
  "ver": int,       // vote version
  "guid": byte[],   // group identifier
  "go": byte[],     // group owner identity
  "gid2": byte[],   // group v2 identifier
  "o2oi": <One-to-one discussion identifier>,
  "ref": <Message reference>,
  "ost": int        // original server timestamp
}
```
This message adds (`v` is `True`) or removes (`v` is `False`) the vote of the sender for a candidate of the poll message referenced by a discussion and a message reference. The vote version is incremented with each change, and only the vote with the highest version is kept. As for messages, `ost` is used when resending votes in a group v2 after a member joins the group.

History transfer message:
```
{
  "id": String,     // transfer identifier
  "sdp": {
    "t": String,    // "offer" or "answer"
    "sdp": String   // session description
  },
  "ice": [<ICE candidate>]
}
```
This message contains either a session description, or a list of ICE candidates (each containing the candidate `"sdp"` string, its `"mli"` media line index, and its `"mid"` media identifier), exchanged to establish a history transfer (see Part IX).

History transfer control:
```
{
  "id": String,     // transfer identifier
  "t": int          // control message type
}
```
The control message type is one of REQUEST (1), sent by the source device, ACCEPT (2), sent by the destination device, or REJECT (3), sent by either device to reject or abort the transfer.

24 Encryption

24.1 Message Encryption

The message itself is always encrypted with an authenticated encryption symmetric primitive. The authenticated encryption message key is generated with the gkmv2 method described below. Once generated, this message key is wrapped in the headers, using a method that depends on the channel the message is being sent on.

Before being encrypted, the encoded message payload is padded to a multiple of 512 bytes. This is especially useful for short text-only application messages to avoid disclosing the length of the text, or for protocol messages to avoid disclosing which protocol message this could be. After

decryption, the decodeWithPadding procedure (see Section 22.9) is used.

GkmV2. In order to avoid message manipulation by a group member having full control of the Olvid message distribution server, the content of each header must be tied to the message content itself. We denote by gkmv2 the technique we implemented to achieve this.

In order to remain backward compatible when implementing gkmv2, we opted to leave the message headers untouched, and change the way the message key itself is generated to tie it to the message content. Using the padded message plaintext and a prng as input, the message key is generated using the following procedure:

```
1: procedure GenerateMessageKey(prng, paddedPlaintext)
2:     seed_enc ← prng.bytes(32)
3:     key_enc ← KDFFromPRNGWithHMACWithSHA256.compute(seed_enc, AES256CTRKey)
4:     seed_mac ← key_enc.bytes || paddedPlaintext
5:     key_mac ← KDFFromPRNGWithHMACWithSHA256.compute(seed_mac, HMACWithSHA256Key)
6:     return AES256CTRHMACSHA256Key(key_enc, key_mac)
7: end procedure
```

When receiving a message, the message key is verified by recomputing $\texttt{key}_{mac}$ from $\texttt{key}_{enc}$ and the decrypted padded plaintext, and checking that it matches the $\texttt{key}_{mac}$ of the message key. This verification is mandatory for all channel types (asymmetric, pre-key, and oblivious channels): any message whose message key does not pass this verification is discarded.

Asymmetric channel. This uses the recipient's `identity` encryption public key to wrap the message key, using the public key encryption of Section 17. The header contains

PublicKeyEncryption.encrypt(cryptoIdentity.publicKeyForKEM, encodeSymKey(msgKey), prng),

where cryptoIdentity is the cryptographic identity of the recipient. It is thus the concatenation of a KEM ciphertext (32 bytes) and of the authenticated encryption of the encoded message key.

Pre-key channel. As detailed in Section 25, pre-keys are short-lived encryption public keys that can be used to exchange messages while the oblivious channel is being created. Unlike asymmetric encryption, which does not include any sender authentication (this is taken care of in the protocols using that channel), pre-key message encryption authenticates the sender so that the application is certain about who the message sender is. The "sign-then-encrypt" technique is used with the following procedure, taking as input the recipient's pre-key, the message key to wrap, the sender's ownedCryptoIdentity (including private keys), the sender's `identity` and `deviceUid`, the recipient's `identity` and `deviceUid`, and a prng:

```
1: procedure WrapWithPreKey(preKey, msgKey, ownedCryptoIdentity, ownId, ownDevUid,
2:                          remoteId, remoteDevUid, prng)
3:     encodedPayload ← encodeList(encodeSymKey(msgKey),
4:                                 encodeBytes(ownDevUid), encodeBytes(ownId))
5:     signaturePayload ← encodeList(encodedPayload, encodeBytes(remoteId),
6:                                   encodeBytes(remoteDevUid), encodeBytes(preKey.id))
7:     signature ← AuthenticationPublicKeyOverEC.solve(
8:                                 ownedCryptoIdentity.privateKeyForAuthentication,
9:                                 signaturePayload,
```

```
10:                                    "encryptionWithPreKey",
11:                                    ownedCryptoIdentity.publicKeyForAuthentication,
12:                                    prng)
13:     encodedPlaintext ← encodeList(encodedPayload, encodeBytes(signature))
14:     return preKey.id || PublicKeyEncryption.encrypt(preKey.publicKey, encodedPlaintext, prng)
15: end procedure
```

Upon reception of a header wrapped with a pre-key, the recipient device, whose owned identity is ownId and current device UID is ownDevUid, proceeds as follows:

- it parses the header as preKeyId || c where preKeyId is 32 bytes long, and retrieves the private key of its own pre-key with this ID (if it does not have such a pre-key, the header is ignored);
- it decrypts c with PublicKeyEncryption.decrypt and this private key, decodes the resulting encoded list (encodedPayload, encodeBytes(signature)), and decodes encodedPayload to recover msgKey, the sender's device UID remoteDevUid, and the sender's identity remoteId;
- it checks the signature using AuthenticationPublicKeyOverEC.check, with the prefix `"encryptionWithPreKey"`, the public key for authentication of remoteId, and the challenge

  encodeList(encodedPayload, encodeBytes(ownId), encodeBytes(ownDevUid), encodeBytes(preKeyId)).

  If the signature is invalid, the header is discarded;
- it checks that this signature was not already received with this pre-key, and discards the header otherwise.

To prevent replays, the recipient records the signature of each pre-key message it processes, along with the pre-key used. The signature being randomized, a sender never produces the same signature twice, so that only a replayed header can contain a signature that was already recorded. Signatures are recorded when the message is actually processed (not when it is kept for later processing, see below), and are deleted along with their pre-key, after which messages wrapped with this pre-key cannot be decrypted anyway. The message key msgKey is then used to decrypt the message, which is considered as coming from the device remoteDevUid of remoteId. If remoteId is not (yet) a contact of the recipient, the message is not processed immediately: it is kept for about two weeks, so that it can be processed if the contact is added in the meantime (for example, when the pre-key message is received before the protocol message adding the contact), and deleted otherwise.

Oblivious channel. Oblivious channels use a self ratchet system (in combination with the full ratchet described in Section 34) which generates a series of authenticated encryption keys and key IDs (32-byte random identifier). The key ID allows the recipient to efficiently identify which key to use for decryption.

Each direction of an oblivious channel is associated with a seed: when a channel is created (see Section 32) or fully ratcheted, the device uses a send seed to generate the keys it uses to wrap message keys, while the remote device uses the same seed (its receive seed) to generate the corresponding keys for unwrapping. For each message to send, the self ratchet procedure below is applied to the current send seed: it returns the seed to use for the next message, a key ID, and an authenticated encryption key.

```
1: procedure SelfRatchet(seed)
2:     prng ← PRNGWithHMACWithSHA256(seed)
3:     seed_next ← prng.bytes(32)
4:     keyId ← prng.bytes(32)
```

```
5:     key ← AES256CTRHMACSHA256.generateKey(prng.bytes(32))
6:     return (seed_next, keyId, key)
7: end procedure
```

The send seed is then replaced by $\texttt{seed}_{next}$, and the header is

$$\mathsf{keyId} \parallel \mathsf{AES256CTRHMACSHA256.encrypt}(\mathsf{encodeSymKey}(\mathsf{msgKey}), \texttt{key}, \mathsf{prng}),$$

that is, the concatenation of the key ID (32 bytes) and of the authenticated encryption of the encoded message key.

On the receiving side, the device applies the same procedure to its receive seed in advance, to maintain a set of upcoming (key ID, key) pairs. When a message is received, the key ID of the header is looked up in this set to find the key used to decrypt the message key. The self ratchet being a one-way process, the keys used for past messages cannot be recovered from the current seeds.

More precisely, the (key ID, key) pairs generated from a receive seed form a *provision*. When a provision is created (at channel creation or after a full ratchet), 100 pairs are generated, and 50 new pairs are generated each time fewer than 50 pairs remain unused. Each key is deleted as soon as it has been used to decrypt a message key. As messages may be received out of order, when a key is used, the keys that were generated before it (either earlier in the same provision, or in a previous provision) are not deleted immediately: they are given an expiration date 60 days in the future (the maximum time a message can remain on the server), after which they are deleted.

24.2 Extended Payload Encryption

When a message has an extended payload, the key used to encrypt/decrypt this payload is directly derived from the message key using the following procedure:

```
1: procedure DeriveExtendedPayloadKey(msgKey)
2:     prng ← PRNGWithHMACWithSHA256(SeedFromKeys(msgKey))
3:     seed ← prng.bytes(32)
4:     return AES256CTRHMACSHA256.generateKey(seed)
5: end procedure
```

where SeedFromKeys is the seed computation from authenticated encryption keys described in Section 32 (encryption of a 32-byte all 0 plaintext with each key using a prng initialized with an all 0 seed, then hash of the concatenated ciphertexts), applied here to the single key msgKey.

24.3 Attachment Encryption

Each attachment is encrypted using a random authenticated encryption key (sent with the message). Attachments are split into chunks of size determined by the sender (typically a few MB, larger chunks being used for very large attachments in order to limit the number of chunks). The chunk size is actually the encrypted chunk size, so the plaintext size is a little smaller. Each chunk of the attachment is independently encrypted with the authenticated encryption key. The plaintext of the i-th chunk (starting at $i = 0$) is

$$\mathsf{encodeList}(\mathsf{encodeInt}(i), \mathsf{encodeBytes}(\mathsf{chunkData}_i)),$$

and, after decryption, the recipient checks that the chunk number it contains matches the number of the chunk it downloaded, so that chunks cannot be reordered.

24.4 Return Receipt Encryption

When sending an application message, the sender generates a random 16-byte nonce and a random authenticated encryption key $\texttt{key}_{rr}$ (using AES256CTRHMACSHA256.generateKey), and includes both in the message payload (see the `rr` element of the message payload JSON). Upon reception of the message (and, if read receipts are enabled, when the message is read), the recipient sends a return receipt, containing the encrypted payload

$$\mathsf{AES256CTRHMACSHA256.encrypt}(\mathsf{payload}, \texttt{key}_{rr}, \mathsf{prng}_{rr})$$

where payload is either $\mathsf{encodeList}(\mathsf{encodeBytes}(\texttt{identity}), \mathsf{encodeInt}(\mathsf{status}))$ for a return receipt concerning the message, or $\mathsf{encodeList}(\mathsf{encodeBytes}(\texttt{identity}), \mathsf{encodeInt}(\mathsf{status}), \mathsf{encodeInt}(\mathsf{attachmentNumber}))$ for a return receipt concerning one of its attachments. Here, `identity` is the identity of the recipient of the message (sender of the return receipt), and status is 1 for "delivered" and 2 for "read". The key $\texttt{key}_{rr}$ makes it possible to mask the `identity` of the sender of the return receipt as well as its nature, and the nonce allows the message sender (return receipt recipient) to identify which key to use for decryption.

The prng_{rr} used for this encryption is a PRNGWithHMACWithSHA256 initialized with the deterministic seed of the owned identity sending the return receipt, computed with the tag `0x57` and the diversification value $\mathsf{nonce} \,\|\, \mathsf{payload}$ (see Section 19.1). Including the payload in the diversification value ensures that two different return receipts for the same message (for instance "delivered" and "read") never reuse the same AES-CTR IV under the key $\texttt{key}_{rr}$. The encryption is deterministic: all the devices of the user sending the return receipt generate the exact same encrypted return receipt. This allows server-side deduplication: the return receipts sent by the multiple devices of a user for the same message are identical, so they can be stored and delivered only once.

25 Pre-Keys Management

Pre-keys are short-lived encryption public keys that a device stores on the server for other devices to use when sending it a message. Pre-keys are intended to be used only when an oblivious channel between the devices is not available. If an oblivious channel is available, it is more efficient and makes sure messages are encrypted with single-use keys.

A pre-key is composed of the following elements:

- the `deviceUid` of the device it belongs to
- a 32-byte pre-key ID (picked at random)
- an encryption public key (generated as any other key pair)
- an expiration timestamp (based on the current server timestamp and a default pre-key lifespan)

In addition to this, as pre-keys can be used to exchange any kind of application message, it is important that device capabilities are known to the sender (typically, whether the device supports groups v2, continuous ICE for calls, etc.). These capabilities used to be negotiated *after* the oblivious channel creation, which is no longer sufficient, so they are also included with the

pre-key on the server. An encoded signed pre-key, as uploaded on the server, is an encoded list containing:

- an encoded dictionary containing:
  - for key `"prk"`, the encoded pre-key which is an encoded list of:
    - the encoded 32-byte pre-key ID
    - the encryption public key, encoded with encodeBytes applied to its compact representation
    - the encoded `deviceUid`
    - the encoded expiration timestamp `int`
  - for key `"cap"`, an encoded list of encoded `String`, each one corresponding to a device capability
- a signature of the bytes of the previous encoded dictionary, using the pre-key owner's server authentication key and the prefix `"devicePreKey"`

Pre-keys verification. When a device retrieves the signed pre-key of a remote device (during a device discovery, see Section 35), it only accepts it if:

- the signature is valid, using AuthenticationPublicKeyOverEC.check with the prefix `"devicePreKey"` and the public key for authentication of the identity owning the device,
- the `deviceUid` contained in the pre-key is that of the device for which the pre-key was retrieved,
- the pre-key has not expired yet, i.e., its expiration timestamp is larger than the current server timestamp returned along with it.

Pre-keys lifecycle. Currently, pre-keys have a lifespan of 60 days and are renewed every 7 days. Therefore, when running an owned device discovery, if the application receives a pre-key for the current device that expires in less than 53 days, a new pre-key with a fresh 60-day lifespan is generated and uploaded on the server. Such an owned device discovery is run at every app startup (and once a day for Android applications running for more than one day).

When an owned pre-key expires, it is kept in the database for another 60 days in case a message sent just before its expiration was to remain 60 days on the server before being listed by the device.

On the contact side, a contact device discovery is run every 7 days to make sure we always have a fresh pre-key at hand for our contacts.

Part V
Cryptographic Protocols

The Olvid engine runs a protocol manager able to execute cryptographic protocols step by step, similarly to finite state automatons. Each protocol is thus defined by a number of possible states (including an initial state and a set of final states), transitions between these states called protocol steps, and messages triggering the execution of such steps.

Protocol messages are sent and received over the network and must be serialized. In order for the recipient to identify the cryptographic protocol a message corresponds to and the nature of the message itself, each cryptographic protocol implemented in Olvid is assigned a unique protocol ID, and each possible message in the protocol is assigned a message ID. The list of all protocol IDs is included in Table 3. The protocol ID and message ID of a message are serialized alongside the message payload.

Protocol IDs 1 and 5 were used by older protocols which are no longer used.

In addition, each execution of a protocol is identified by a unique identifier, the `protocolUid`, which is stored alongside the state of the protocol and is included in any protocol message. This allows a user running multiple instances of the same protocol to uniquely identify which instance a message is related to. Some protocols (the Full Ratchet Protocol, the Group Management Protocol and the Groups v2 Protocol) use a deterministic `protocolUid`, computed from information known to all participants, so that all participants use the same one. Otherwise, the `protocolUid` is generated at random and must be kept secret, only shared between legitimate parties to the protocol, as the security of some protocols relies on it.

When a protocol message is received, the protocol manager:

- decodes the protocol ID, message ID and `protocolUid`
- looks up the `protocolUid` in its database of protocol states
  - if a state is found it is restored
  - if no state is found, a new initial state for the corresponding protocol is created
- finds a protocol step to execute, matching the state and the message
  - if a step matches, it is executed
  - if no step matches, the protocol message is stored "for later use"; such messages are deleted if no step matches them within 15 days
- at the end of the step execution
  - if the protocol reached a final state, everything related to this `protocolUid` is erased
  - if it reached a non-final state, the current protocol state is updated in the database

Each protocol step also specifies through which type of channel (local channel, asymmetric channel, oblivious or pre-key channel, channel with another device of the same owned identity, etc.) the message triggering it must have been received. A message received through another type of channel is rejected. This is what allows protocol steps to rely on the authenticity guarantees of

ID	Protocol name
0	Device Discovery Protocol (see Section 35)
2	Channel Creation with Contact Device Protocol (see Section 32)
3	Device Discovery Child Protocol (see Section 35)
4	Contact Mutual Introduction Protocol (see Section 28)
6	Identity Details Publication Protocol (see Section 39)
7	Contact Picture Download Child Protocol (see Section 39)
8	Group Invitation Protocol (groups v1, see Section 44)
9	Group Management Protocol (groups v1, see Section 45)
10	Contact Management Protocol (see Section 31)
11	Trust Establishment Protocol with SAS (see Section 26)
12	Trust Establishment Protocol with Mutual Scan (see Section 27)
13	Full Ratchet Protocol (see Section 34)
14	Group Picture Download Child Protocol (groups v1, see Section 45)
15	Keycloak Contact Addition Protocol (see Section 29)
16	Device Capabilities Discovery Protocol (see Section 38)
17	One-to-One Contact Invitation Protocol (see Section 30)
18	Groups v2 Protocol (see Section 48)
19	Groups v2 Picture Download Protocol (see Section 49)
20	Owned Identity Deletion Protocol (see Section 42)
21	Owned Device Discovery Protocol (see Section 36)
22	Channel Creation with Owned Device Protocol (see Section 33)
23	Keycloak Binding and Unbinding Protocol (see Section 41)
24	Owned Device Management Protocol (see Section 37)
25	Synchronization Protocol (see Section 43)
26	Owned Identity Transfer Protocol (see Section 40)

Table 3: List of protocols implemented in Olvid and their corresponding protocol ID.

the channel they receive messages from.

Note that during a protocol step execution many different actions are performed, but these actions can only be database modification operations: no direct network operations or user interface interactions. This way, after a successful step execution, all the modifications can be committed atomically to the database, and if the execution is interrupted it can be replayed at the next protocol manager start.

26 Trust Establishment Protocol with SAS

26.1 Purpose and High Level View

The Trust Establishment Protocol with SAS represented in Figure 2 allows two users to mutually authenticate each other's cryptographic identity. The protocol typically starts when Alice obtains Bob's identity by means of some untrusted channel (such as email, SMS, WhatsApp, etc.). Using this identity (assumed to be that of Bob), Alice starts the protocol by executing the SendCommitment step. When receiving this protocol message, Bob gets a chance to accept or reject the invitation, i.e., to continue or abort the protocol. Bob can do so on any of his devices.

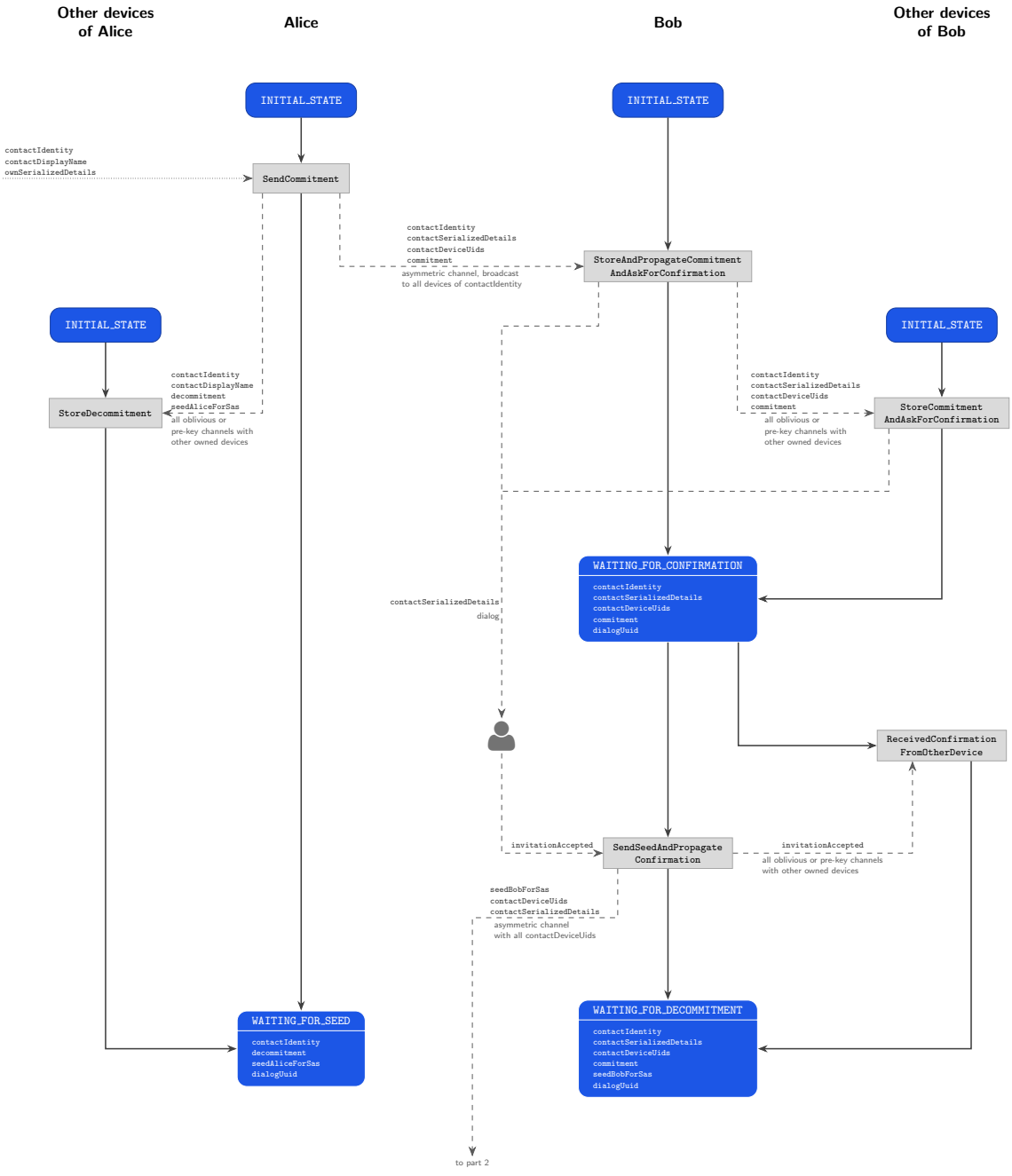


Figure 2: Trust Establishment Protocol with SAS (part 1)

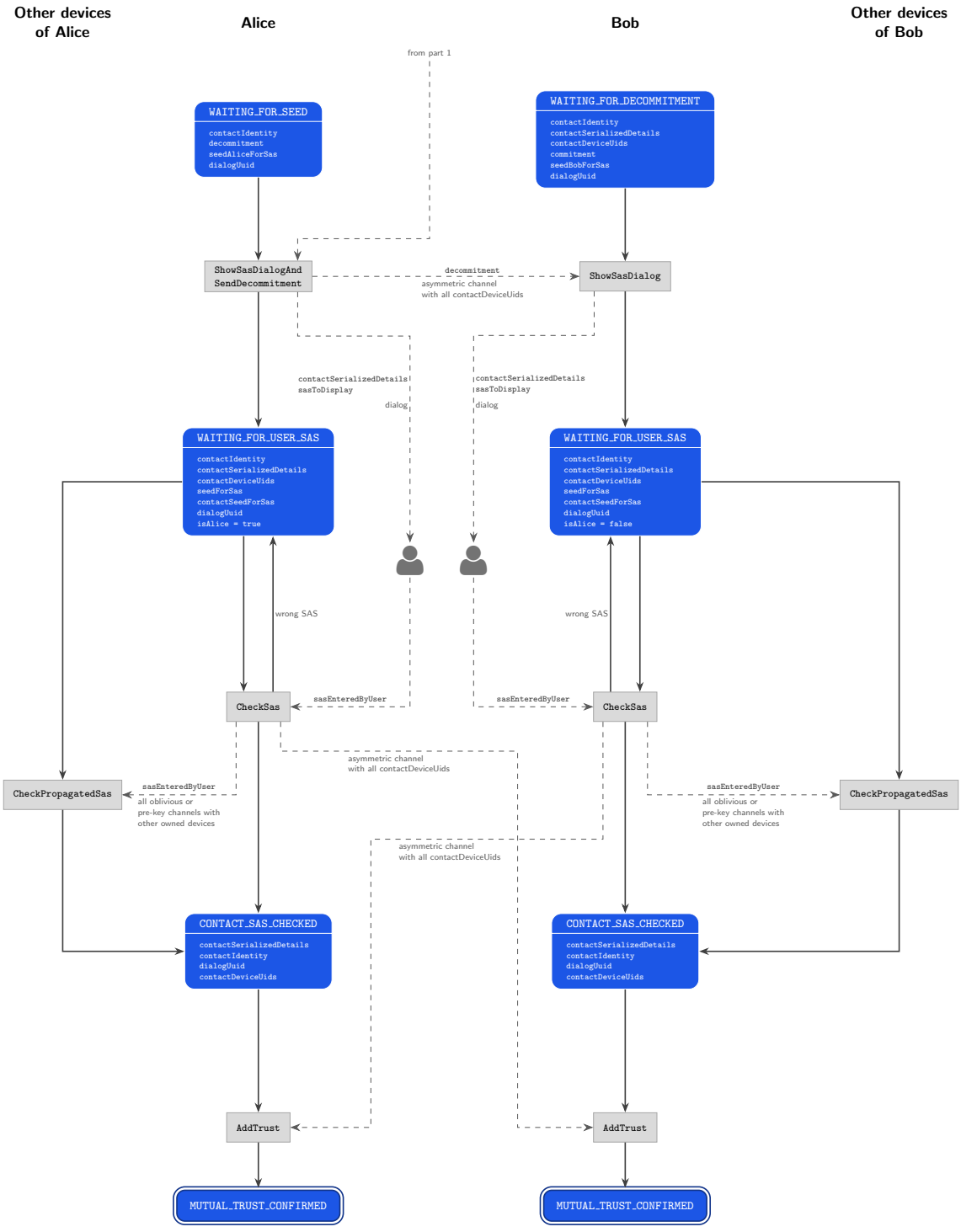


Figure 3: Trust Establishment Protocol with SAS (part 2)

If Bob accepts, the protocol continues until an 8-digit SAS is generated on the basis of the transcript of the protocol. Assuming no man-in-the-middle attack occurred, the SAS will be identical on Alice and Bob's sides (on all of their devices). Four of these digits are displayed to Alice, the other four being displayed to Bob. At this point, Alice and Bob should exchange their digits on an *authenticated* channel (e.g., face-to-face or phone call, the channel is not required to be confidential). If the digits match, both Alice and Bob are assured of each other's true identities, i.e., that their public keys are authentic.

An adversary modifying the messages and trying to perform a man-in-the-middle attack has a success probability of 10^{-8}.

26.2 Cryptographic Details

Commitment. In the SendCommitment step, Alice computes a commitment using the scheme of Section 9 with inputs: her `identity` as the `tag` and a random `seedForSas` she just generated as the `value`.

Bob stores all the commitments he ever received in a database to avoid commitment replay. Indeed, to make this protocol work in a multi-device setting, the $\texttt{seedForSas}_{Bob}$ is computed deterministically from the commitment sent by Alice: it is the deterministic seed of Bob's owned identity computed with the tag `0x55` and the commitment as diversification value (see Section 19.1), so that all of Bob's devices compute the same value. This means that with commitment replay, it would be possible to guess Bob's response and improve the probability of success of a man-in-the-middle attack. This database prevents this.

SAS computation. Once a user has both $\texttt{seedForSas}_{Alice}$ and $\texttt{seedForSas}_{Bob}$, they can compute the 8-digit SAS as follows:

procedure ComputeSAS($\texttt{seedForSas}_{Alice}, \texttt{seedForSas}_{Bob}, \texttt{identity}_{Bob}$)
    $\texttt{hash} \leftarrow \text{SHA256}(\texttt{identity}_{Bob} \| \texttt{seedForSas}_{Alice})$
    $\texttt{seed} \leftarrow \texttt{hash} \oplus \texttt{seedForSas}_{Bob}$
    Initialize prng, a PRNGWithHMACWithSHA256 using `seed`
    $SAS \leftarrow \text{prng.bigInt}(10^8)$
end procedure

The SAS is the decimal representation of this integer, padded with leading zeros to 8 digits. Alice is shown the first 4 digits and Bob the last 4.

27 Trust Establishment Protocol with Mutual Scan

27.1 Purpose and High Level View

The Trust Establishment Protocol with Mutual Scan represented in Figure 4 allows two users who are physically together to mutually authenticate each other's cryptographic identity by scanning each other's QR-code, without having to exchange a SAS. It works as follows:

- Bob scans the QR-code of Alice, containing her Invitation Link (see Figure 1) and thus her `identity`.

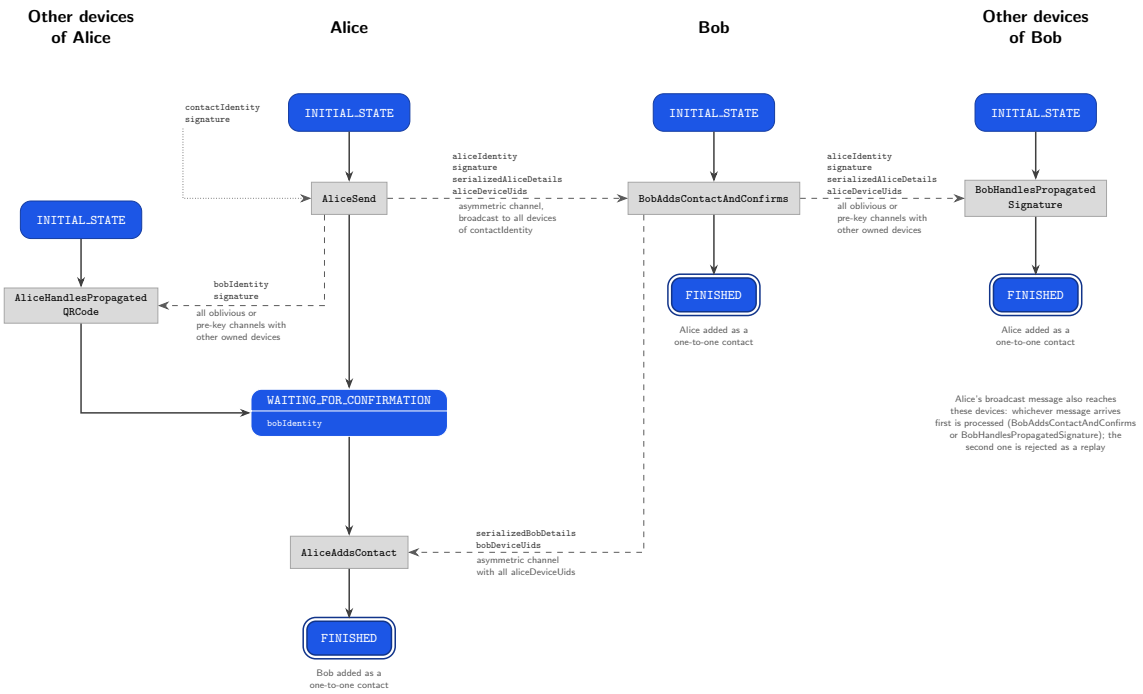


Figure 4: Trust Establishment Protocol with Mutual Scan

- Bob's device then displays a *mutual scan QR-code*, containing Bob's `identity`, his display name, and a signature computed by Bob over Alice's and Bob's identities.
- Alice scans this mutual scan QR-code. Her device checks the signature, and starts the protocol by executing the AliceSend step: it sends the signature, along with her own details and the list of her `deviceUid`, to Bob through an asymmetric channel (in broadcast to all his devices), and propagates the QR-code content to her other devices.
- Upon reception, Bob executes the BobAddsContactAndConfirms step: he checks that the signature is a valid signature he computed himself over Alice's and his own identities, and that he never received this signature before. If so, he adds Alice as a one-to-one contact (with a direct trust origin), propagates the signature to his other devices, and sends his own details and the list of his `deviceUid` to Alice's devices through an asymmetric channel.
- Upon reception of this confirmation, Alice executes the AliceAddsContact step and adds Bob as a one-to-one contact (with a direct trust origin).

Both users then run a device discovery (if needed) and the usual channel creation protocols with each other's devices.

Alice trusts Bob's `identity` because she obtained it by scanning Bob's screen. Bob trusts Alice's `identity` because the message he receives contains a fresh signature he computed himself over Alice's `identity`, which he only displayed on his screen after scanning Alice's QR-code. The confirmation Bob sends to Alice is bound to the protocol instance by the `protocolUid`, which is random and was only sent to Bob, encrypted, in Alice's first message.

27.2 Cryptographic Details

Mutual scan QR-code. The mutual scan QR-code contains a link of the form

$$\texttt{https://invitation.olvid.io/2\#}\langle data \rangle$$

where $\langle data \rangle$ is the base64url encoding (without padding) of

$$\mathsf{encodeList}(\mathsf{encodeBytes}(\texttt{identity}_{Bob}), \mathsf{encodeString}(\mathsf{displayName}_{Bob}), \mathsf{encodeBytes}(\sigma)).$$

Signature. The signature σ is computed by Bob with AuthenticationOverEC.solve, using the authentication private key of his owned identity, the prefix `"mutualScan"` (see Table 1), and the challenge $\texttt{identity}_{Alice} \,\|\, \texttt{identity}_{Bob}$ (the concatenation of the byte representations of both identities). Alice checks it with AuthenticationOverEC.check and Bob's public key for authentication before starting the protocol. Upon reception, Bob (and each of his other devices when it is propagated to them) checks it using his own public key for authentication.

Replay protection. Each of Bob's devices stores all the mutual scan signatures it ever received and rejects any signature it already received. A given mutual scan QR-code can thus only be used once to add Alice as a contact: for example, if Bob later deletes Alice from his contacts, Alice cannot use the same signature again to be re-added.

Messages. All messages sent to the contact are sent through asymmetric channels. Messages propagated to the other devices of a user are sent through oblivious channels or pre-key channels.

28 Contact Mutual Introduction Protocol

28.1 Purpose and High Level View

This protocol allows a user to introduce two users they are in contact with to each other.

Suppose Alice is in contact with Bob and Dave. This protocol allows her to push Bob's `identity` to Dave and Dave's `identity` to Bob. Bob and Dave can then choose to trust Alice and add the `identity` she sent them to their contact database, without ever having to exchange a SAS, and without the need for an authentic channel between them. Here, Alice plays the role of a trusted third party distributing cryptographic keys. If she decides to manipulate the identities she sends, Bob and Dave have no way to directly detect it. Still, at any time, Bob and Dave can verify the keys by running an instance of the Trust Establishment with SAS protocol (see Section 26).

Alice can only introduce contacts which are one-to-one contacts. When receiving an introduction, Bob (resp. Dave) rejects it if Alice is not one of his one-to-one contacts. If the introduced contact is already a one-to-one contact, the introduction is automatically accepted. Otherwise, the user is asked whether to accept it or not.

28.2 Cryptographic Details

Signature. The signature uses the signature/authentication key inside the `identity` of the party sending the notification that they accepted the contact introduction. As the notification is sent through an asymmetric channel, this signature is necessary to authenticate Alice/Bob and guarantee that it is indeed Alice/Bob accepting the contact introduction. When Alice sends the notification, this signature is computed over the concatenation of:

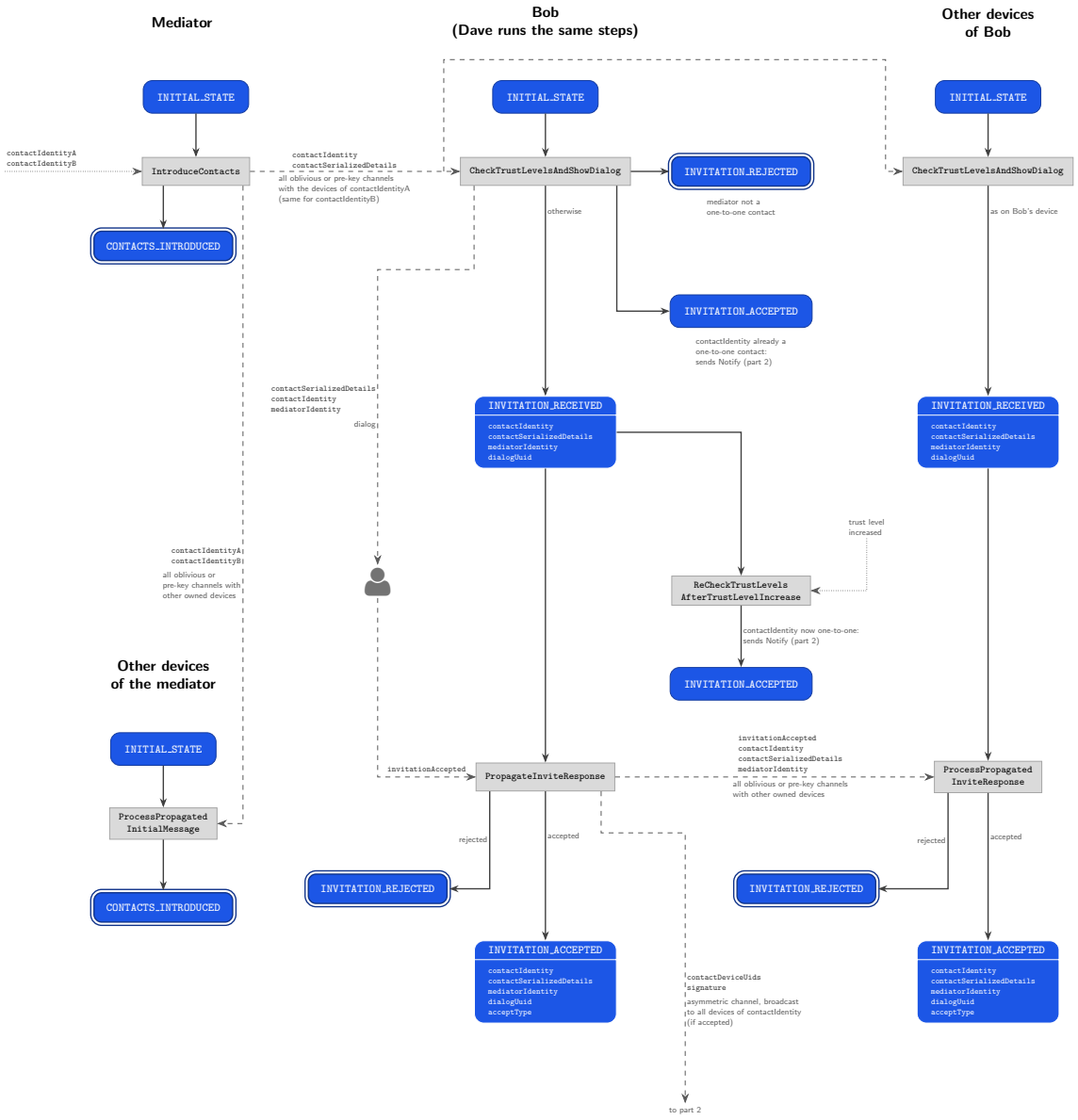


Figure 5: Contact Mutual Introduction Protocol (part 1)

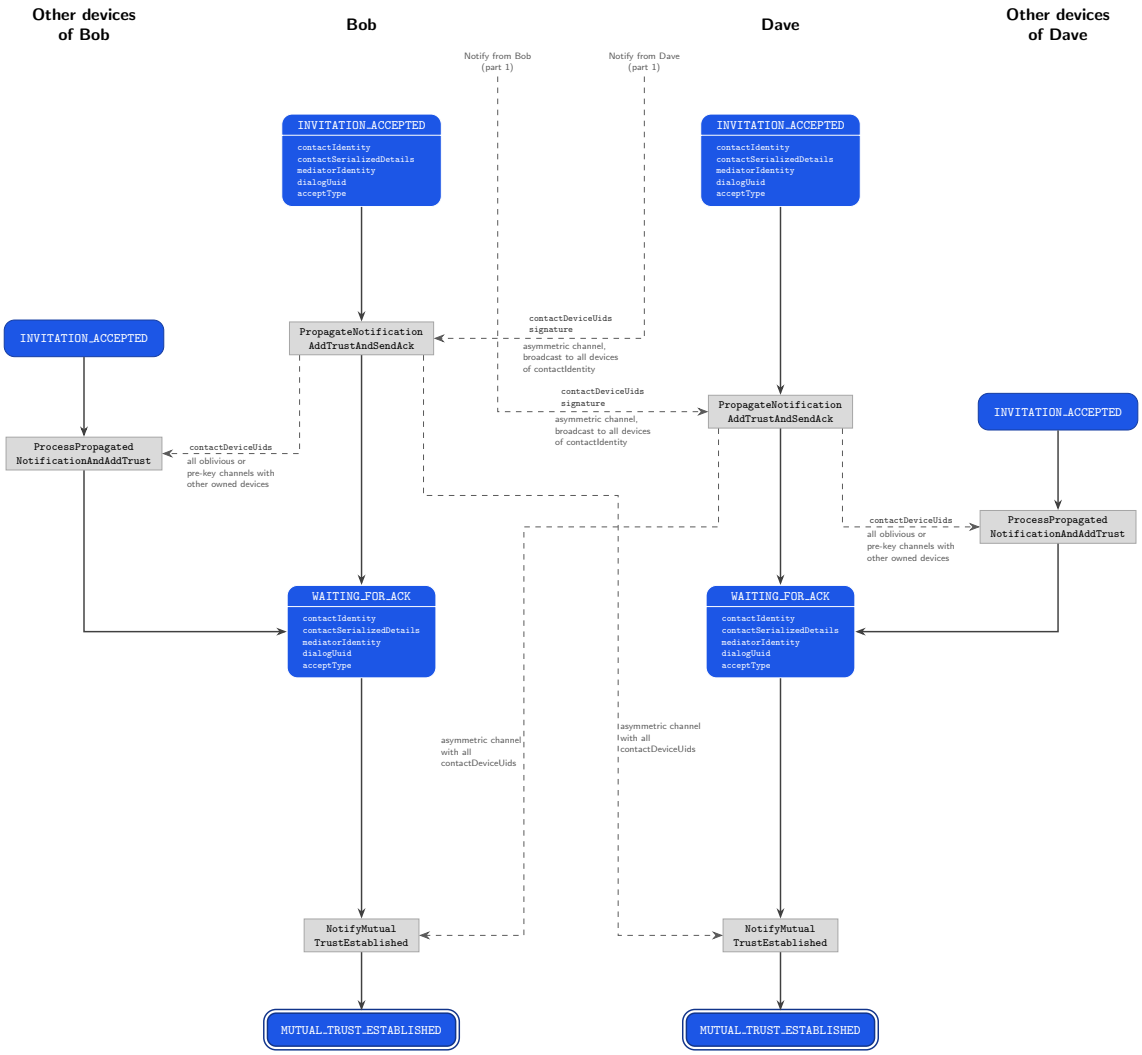


Figure 6: Contact Mutual Introduction Protocol (part 2)

- a constant prefix "mutualIntroduction"
- the `identity` of the mediator (the party running the `IntroduceContacts` step)
- the `identity` of Bob
- the `identity` of Alice
- a random 16-byte padding (also included in the signature)

29 Keycloak Contact Addition Protocol

29.1 Purpose and High Level View

When an owned identity is managed by a Keycloak server (see Part X), the Keycloak server acts as a directory: it provides, for each of its users, *signed user details*, a JWS signed by the Keycloak server containing the `identity` of the user and their details (name, position, company, etc.). The Keycloak Contact Addition Protocol allows Alice, a user of a Keycloak server, to add Bob, another user of the same Keycloak server, as a one-to-one contact, without any other trust establishment protocol: the trust in Bob's `identity` comes from the Keycloak server.

The protocol works as follows:

- Alice retrieves Bob's signed user details from the Keycloak server (see Part X) and starts the protocol with Bob's `identity` and these signed details.
- In the VerifyContactAndStartDeviceDiscovery step, Alice's device checks the signed details of Bob, as well as her own signed details (see below), and runs a device discovery for Bob's `identity` using the Device Discovery Child Protocol (see Section 35).
- In the AddContactAndSendRequest step, Alice's device adds Bob as a one-to-one contact with a Keycloak trust origin, along with his devices and their (verified) pre-keys. It propagates this contact addition to Alice's other devices, and sends to all of Bob's devices, through an asymmetric channel, an invitation containing: Alice's `identity`, Alice's signed user details, the list of Alice's `deviceUid`, and the URL of Alice's Keycloak server.
- Upon reception, in the ProcessReceivedKeycloakInvite step, Bob's device checks Alice's signed details (see below), and asks its Keycloak server whether Alice's identity was revoked. If all these checks succeed, in the AddContactAndSendConfirmation step, Bob's device adds Alice as a one-to-one contact with a Keycloak trust origin, along with her devices, and sends a positive confirmation to Alice. If any of these checks fails, Bob's device sends a negative confirmation to Alice.
- Upon reception of a negative confirmation, if Bob is only trusted by Alice through this Keycloak server, Alice's device deletes Bob from her contacts.

29.2 Cryptographic Details

Verification of signed user details. Signed user details are only accepted if all the following checks succeed:

- the owned identity verifying them is managed by a Keycloak server, and the signature of the JWS is valid for the signature key of this Keycloak server (see Part X);
- the signed details contain the `identity` expected by the verifying device (for example, the `identity` of the sender of the invitation);
- this `identity` was not revoked by the Keycloak server, according to the list of revocations the device obtained from the Keycloak server: an identity revoked because it was compromised

is always rejected, and an identity revoked because the user left the company is rejected if the details were signed before the revocation;
- the signed details are not too old, i.e., they were signed less than 60 days ago.

The same verification is used whenever signed user details are received from a contact, for example in the published details of a contact (see Section 39), in order to determine whether the contact is certified by the Keycloak server of the owned identity.

Revocation check. Bob's device sends Alice's signed user details to its Keycloak server, which answers whether the corresponding identity was revoked (see Part X).

Messages. The invitation and the confirmation are sent through asymmetric channels. The propagation of the contact addition to the other devices of Alice is sent through oblivious channels or pre-key channels.

30 One-to-One Contact Invitation Protocol

30.1 Purpose and High Level View

In Olvid, a user may have two kinds of contacts:

- *one-to-one* contacts, with whom the user explicitly agreed to be in contact (for example, after a trust establishment protocol or an accepted contact introduction), and with whom the user can have a one-to-one discussion,
- other contacts, which the user only knows indirectly (typically, other members of a group), and with whom they cannot have a one-to-one discussion.

The One-to-One Contact Invitation Protocol allows Alice to invite Bob, who is already one of her contacts but not a one-to-one contact, to become a one-to-one contact. When receiving such an invitation:

- if Alice is already a one-to-one contact of Bob, or if Bob has himself already invited Alice to become a one-to-one contact, the invitation is automatically accepted,
- otherwise, Bob is asked whether to accept it or not.

If Bob accepts, both Alice and Bob mark each other as one-to-one contacts. Alice can also abort her invitation before Bob responds. The invitation, the response and the abort are propagated to the other devices of Alice and Bob.

As the one-to-one status of a contact might become inconsistent between Alice and Bob, the protocol also includes a *one-to-one status sync* mechanism: Alice sends Bob a message indicating whether she considers Bob as a one-to-one contact. If Bob does not agree:

- if Bob considers Alice as a one-to-one contact but Alice does not, Bob downgrades Alice to a non one-to-one contact,
- if Alice considers Bob as a one-to-one contact but Bob does not, Bob marks Alice as one-to-one only if he had himself invited her; otherwise he asks Alice to downgrade him.

This mechanism never upgrades a contact to one-to-one without the consent of both users. It is run with a contact after a channel is created with one of their devices, and with all contacts when the capabilities of the device change (see Section 38).

30.2 Cryptographic Details

There is no cryptography involved in this protocol. All messages are exchanged through oblivious channels or pre-key channels, both of which guarantee the authenticity of the sender.

31 Contact Management Protocol

31.1 Purpose and High Level View

This protocol serves the same purpose as the group management protocol (see Section 45), but for one-to-one relations. It is a collection of "1-step" protocols:

- Contact deletion: when Alice deletes Bob from her contacts, Bob is notified so that his oblivious channels with Alice are also destroyed and Alice is removed from his contacts. The deletion is also propagated to Alice's other devices.
- Contact downgrade: when Alice decides that Bob should no longer be one of her one-to-one contacts (while keeping him as a contact, for example because they share a group), Bob is notified so that Alice is also downgraded on his side. The downgrade is also propagated to Alice's other devices.
- Contact device discovery request: Alice can ask her contacts to perform a new device discovery for her identity (see Section 35). This is used for example when one of Alice's devices is deactivated, so that her contacts quickly stop using it.

31.2 Cryptographic Details

There is no cryptography involved in this protocol. All messages are exchanged through oblivious channels or pre-key channels, both of which guarantee the authenticity of the sender (for pre-key channels, thanks to the signature of the sender, see Section 24.1).

32 Channel Creation with Contact Device Protocol

32.1 Purpose and High Level View

The Channel Creation with Contact Device Protocol represented in Figure 7 allows two mutually authenticated users, say Alice and Bob, to create a secure channel from Alice to Bob, and another secure channel from Bob to Alice.

This protocol typically starts whenever a new contact is inserted in the database of trusted contacts, or when a new device is added to the list of the contacts' owned devices. This means that both parties will start the protocol, yet only a single instance should finish. Also, both parties might not trust each other at the exact same time (typically, in the SAS protocol of Section 26, one party will enter their SAS before the other). For this reason the protocol is designed in the following way:

- the protocol starts with a ping stating something along the lines of: "I'm sending a ping because I trust your `identity`, but don't have a channel with your device". This ping is sent as soon as the contact `deviceUid` is created, and sent again if there is still no channel 30 days after the last ping.

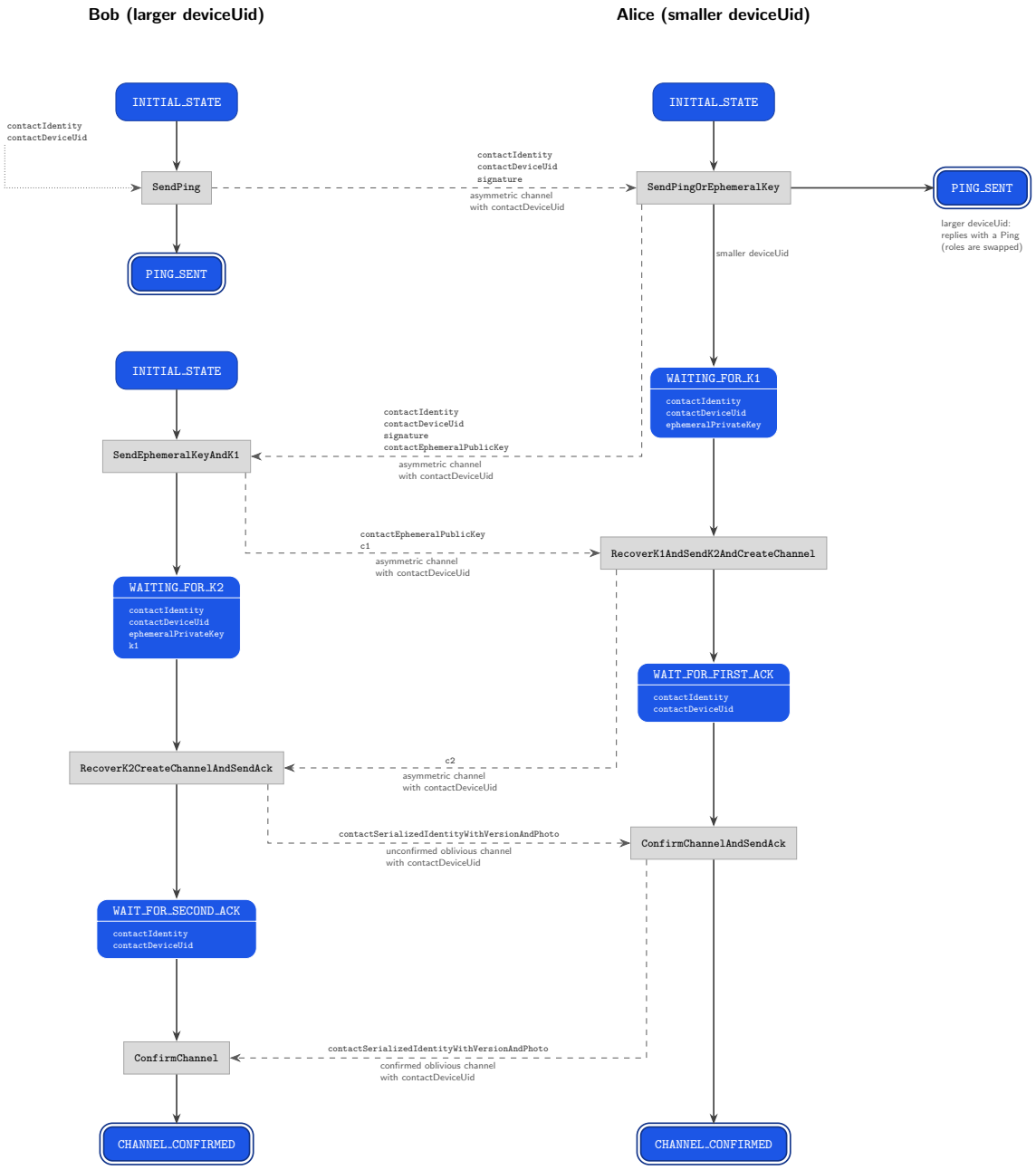


Figure 7: Channel Creation with Contact Device Protocol

- this ping is sent through an asymmetric channel, and must be signed to guarantee its origin
- depending on the actual `deviceUid` of Alice and Bob, the smallest `deviceUid` (with respect to lexicographical ordering of the bytes of the uid) assumes the role of Alice (on the right in Figure 7)
- if Alice or Bob receives a ping from an `identity` they do not trust yet, they discard it
- if Alice receives a ping from Bob and trusts his `identity`, she actually starts the protocol by sending an ephemeral key
- if Bob receives a ping from Alice and trusts her `identity`, he replies with a ping
- as soon as Alice or Bob receives the ack (final steps of the protocol), they confirm the channel and can start using it to send messages
- in practice, receiving any message on the channel is also enough to confirm it

32.2 Cryptographic Details

Signature. The signature uses the signature/authentication key inside the `identity` of the party sending the ping. When Alice sends a ping, this signature is computed over the concatenation of:

- a constant prefix "channelCreation"
- the `deviceUid` of Bob
- the `deviceUid` of Alice
- the `identity` of Bob
- the `identity` of Alice
- a random 16-byte padding (also included in the signature)

Each participant stores all the signatures they ever received from other users to prevent ping replay. Without this, an adversary could simply replay a ping message to force Alice and Bob to recreate a channel, thus effectively performing a denial of service on them.

KEM and seed computation. The `ephemeralPublicKey` exchanged during the protocol are KEM keys, allowing the parties to send ciphertexts c_1 and c_2 and to recover authenticated encryption keys k_1 and k_2.

These keys k_1 and k_2 are used to compute a `seed` in the following way:

- initialize a `prng` with a 32-byte all 0 seed
- encrypt the 32-byte all 0 plaintext with k_1 and the previous `prng`
- encrypt the 32-byte all 0 plaintext with k_2 and the previous `prng`
- concatenate the ciphertexts obtained in the 2 previous steps and hash them
- the output of the hash is the `seed`

This `seed` is then diversified (using Alice's and Bob's `deviceUid`) into a send seed and a receive seed, which are then used by Alice and Bob to initialize each direction of the channel.

33 Channel Creation with Owned Device Protocol

33.1 Purpose and High Level View

The Channel Creation with Owned Device Protocol allows two devices of the same owned identity to create an oblivious channel between them. These channels are used to propagate to the

other devices of a user the actions performed on one device (see for example the Synchronization Protocol), and to send them the messages the user sends to their contacts.

This protocol is identical to the Channel Creation with Contact Device Protocol (see Section 32), with the same messages, the same roles (the device with the smallest `deviceUid` sends the ephemeral key), and the same cryptographic operations, except that the `identity` of both parties is the owned identity. In particular, the ping is sent through an asymmetric channel to the owned identity itself, in unicast to the other device.

The protocol starts when a device learns about a new device of its owned identity (typically, during an owned device discovery), or when it receives a valid ping from such a device. If the remote device was not yet known, it is added to the list of devices of the owned identity during the protocol. Once the channel is confirmed, the two devices exchange their device capabilities (see the Device Capabilities Discovery Protocol), and the details of the owned identity: the acknowledgment messages contain the latest owned identity details (including the photo label and key) known to the sending device, so that the other device can update its own details and download the profile picture if needed.

33.2 Cryptographic Details

Signature. The ping signature is computed with the prefix `"channelCreation"` as in Section 32, over the `deviceUid` of the remote device, the `deviceUid` of the sending device, and twice the `identity` of the owned identity. It can only be computed by a device holding the private key for authentication of the owned identity, which is what allows the receiving device to consider the remote device as one of its own devices. As for the channel creation with a contact device, each device stores all the ping signatures it received to prevent ping replay.

KEM and seed computation. They are identical to those of Section 32.

34 Full Ratchet Protocol

34.1 Purpose and High Level View

The full ratchet protocol makes it possible to completely refresh the encryption keys of an oblivious channel in a way similar to what is done during the Channel Creation Protocol (see Section 32). The main differences are that this protocol only refreshes one direction of the channel at a time and that it can use the oblivious channels already established, making it much simpler than the channel creation.

This protocol is triggered automatically after Alice sends a message to Bob if more than 500 messages were sent since the last full ratchet or if the last full ratchet was more than 30 days ago. A full ratchet which did not complete within 30 days is restarted. This protocol is designed in a way that allows it to be restarted in the middle of a full ratchet. This guarantees that even if Bob receives Alice's messages out of order, he will be able to decrypt all of them properly, without the full ratchet interfering.

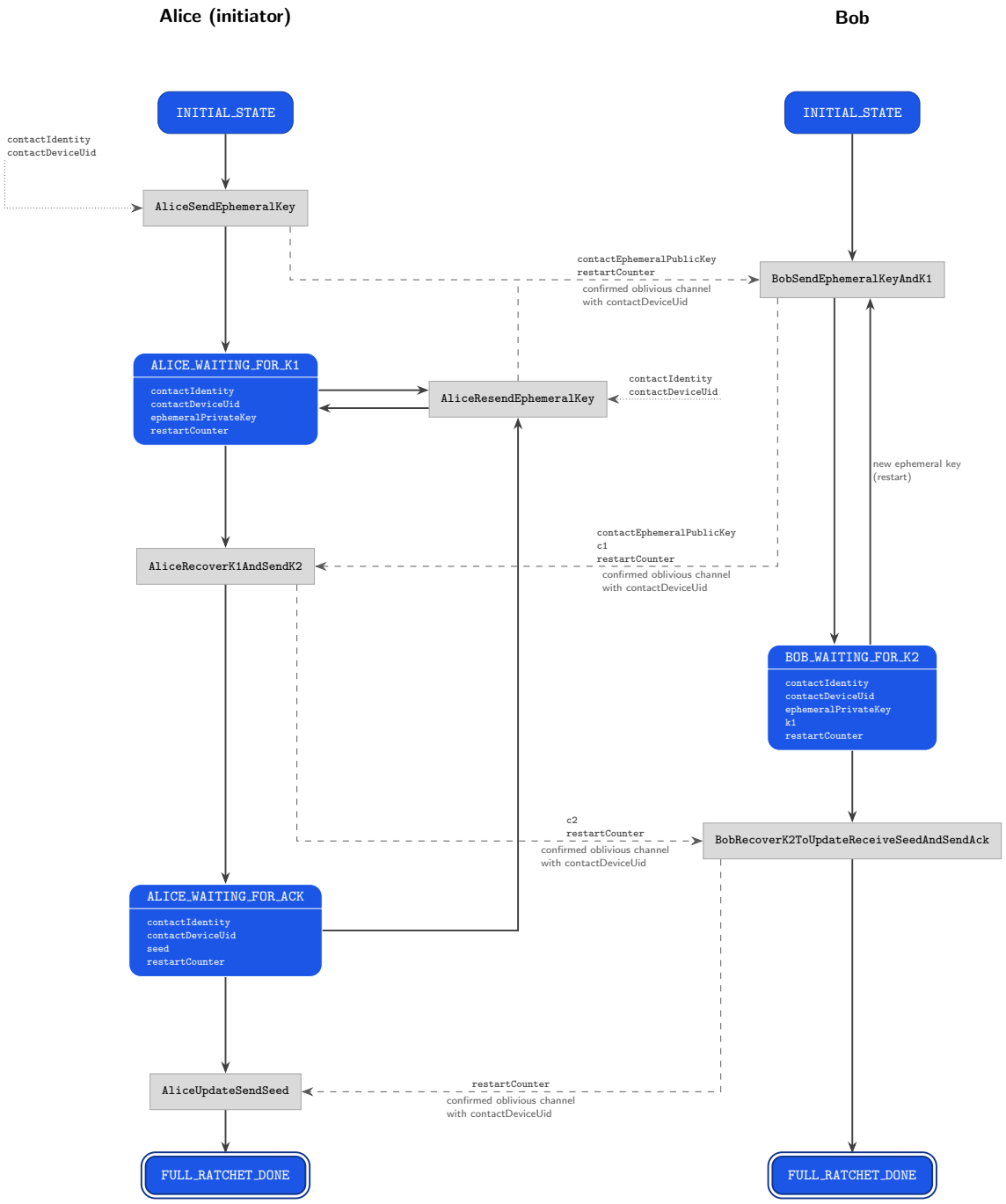


Figure 8: Full Ratchet Protocol

34.2 Cryptographic Details

The cryptography of the full ratchet is exactly the same as in the Channel Creation with Contact Device Protocol (see Section 32), except that no signature is required as users already have an oblivious channel guaranteeing the authenticity of the messages.

35 Device Discovery Protocol

35.1 Purpose and High Level View

The device discovery protocol is a simple protocol used to discover the set of all `deviceUid` of a given `identity`. Before being able to retrieve messages for a specific `deviceUid`, a device must register itself with the server. This way, the server always knows which `deviceUid` exist for a given `identity`.

Thanks to this, the device discovery protocol simply queries the server associated with the `identity`, which responds with the list of `deviceUid`. This query is anonymous (no need to authenticate to the server).

In practice, the protocol is split into two parts:

- a child protocol in charge of querying the server and getting the set of `deviceUid`
- the parent protocol which takes the set of `deviceUid` returned by the child protocol, and updates the contact database.

The purpose of this architecture is to allow other protocols to run the device discovery child protocol without necessarily adding the received set of `deviceUid` to the database. This is for example the case of the Keycloak Contact Addition Protocol.

35.2 Cryptographic Details

For each `deviceUid`, the server also returns the signed pre-key of the device (see Section 25), which includes the capabilities of the device. Each signed pre-key is verified as described in Section 25 ("Pre-keys verification") before being stored, and devices which are no longer listed by the server are removed from the contact database.

36 Owned Device Discovery Protocol

36.1 Purpose and High Level View

The Owned Device Discovery Protocol allows a device to retrieve from the server the list of all the devices of its owned identity, and to keep its local list of owned devices in sync with it. It is run at every application startup (and once a day for Android applications running for more than one day), and whenever an event suggests that the list of devices may have changed (for example when the server notifies a change of the owned devices list).

The server returns, for each device registered for the owned identity: its `deviceUid`, its expiration timestamp (for devices which will expire), the timestamp of its latest registration, its encrypted name (see below), and its signed pre-key (see Section 25). The device then:

- adds the devices it did not know yet to its list of owned devices (which triggers a Channel Creation with Owned Device Protocol, see Section 33),
- removes the devices which are no longer listed by the server, along with the oblivious channels with these devices,
- updates the information of the other devices (name, expiration, pre-key),
- if its own `deviceUid` is not listed by the server, registers again on the server (see Section 73.1), the server deciding whether the device should be deactivated or not,
- checks whether the pre-key of the current device on the server needs to be renewed, and if so, uploads a new pre-key (see "Pre-keys lifecycle" in Section 25).

36.2 Cryptographic Details

Encrypted device list. The query to the server is anonymous. The server encrypts its response with PublicKeyEncryption.encrypt (see Section 17) using the encryption public key of the owned identity, so that only the devices of this owned identity can read the list of its devices. The decrypted response is an encoded dictionary containing:

- for key `"st"`, the current server timestamp,
- for key `"multi"`, a Boolean indicating whether the owned identity is allowed to have multiple devices,
- for key `"dev"`, an encoded list of encoded dictionaries, one for each device, containing the keys `"uid"` (the `deviceUid`), and optionally `"exp"` (expiration timestamp), `"reg"` (latest registration timestamp), `"name"` (encrypted device name), and `"prk"` (signed pre-key).

Device names. Device names are set by the devices themselves (when registering on the server, or with the Owned Device Management Protocol) and are encrypted, so that the server never learns them. A device name is encrypted as

$$\mathsf{PublicKeyEncryption.encrypt}(\mathsf{pk}, \mathsf{pad}_{128}(\mathsf{encodeList}(\mathsf{encodeString}(\mathsf{deviceName}))), \mathsf{prng}),$$

where pk is the encryption public key of the owned identity, and pad_{128} pads its input with zeros to the next multiple of 128 bytes. After decryption, it is decoded using decodeWithPadding (see Section 22.9).

Pre-keys. The signed pre-keys of the other owned devices are verified as described in Section 25 ("Pre-keys verification") before being stored.

37 Owned Device Management Protocol

37.1 Purpose and High Level View

The Owned Device Management Protocol allows a user to manage the devices of one of their owned identities from any of these devices. It supports three actions, each one performed through an authenticated query to the server (see the `/deviceManagement` entry point in Part XII):

- set the name of a device,
- deactivate a device: the device is removed from the list of devices of the owned identity on the server and stops receiving messages,
- set a device as *unexpiring*: for users without a multi-device license, this is how the user chooses which of their devices remains active, the other devices expiring.

Once the server has processed the query, an Owned Device Discovery Protocol (see Section 36) is run to update the local list of owned devices. When a device was deactivated, a contact device discovery request (see Section 31) is also sent to all the contacts of the owned identity, so that they quickly stop using the deactivated device.

37.2 Cryptographic Details

Device names are encrypted as described in Section 36 before being sent to the server. The queries to the server are authenticated with the authentication token of the owned identity (see Section 69.2).

38 Device Capabilities Discovery Protocol

38.1 Purpose and High Level View

Not all devices support the same features, depending on the version of the application they run. Each device thus has a set of *capabilities*, represented as strings. The current capabilities are:

- `webrtc_continuous_ice`: support for continuous ICE candidates gathering in secure calls,
- `one_to_one_contacts`: support for the distinction between one-to-one contacts and other contacts (see the One-to-One Contact Invitation Protocol),
- `groups_v2`: support for groups v2 (see Part VI).

The Device Capabilities Discovery Protocol allows devices to inform the devices of their contacts and their other owned devices of their capabilities:

- when the capabilities of a device change (typically after an application update), it sends its new capabilities to all the devices of its contacts and to its other owned devices,
- when a channel is created with a contact device or another owned device, the device sends its capabilities to this device,
- when a device receives capabilities from a device for the first time, it responds with its own capabilities.

The capabilities of a device are also included in its signed pre-key (see Section 25), so that they are known as soon as a device discovery is performed.

38.2 Cryptographic Details

There is no cryptography involved in this protocol. All messages are exchanged through oblivious channels or pre-key channels, both of which guarantee the authenticity of the sender.

39 Identity Details Publication Protocol and Contact Picture Download Child Protocol

39.1 Purpose and High Level View

All the trust establishment protocols implemented in Olvid (SAS, contact introduction or group creation) take care of exchanging an up-to-date version of the contact details (name, position,

company, etc.). Still, when a user updates their own details and publishes them, all their contacts must be informed. This is the purpose of this protocol.

When a profile picture is set for these details, it is encrypted and uploaded to the server before sending the details to all contacts. On the contact side, the details are received and this triggers the contact picture download child protocol if a picture is present.

39.2 Cryptographic Details

Details distribution. The details (including, if any, the label and key of the profile picture, see below) are sent to all contacts and to the other devices of the user through oblivious channels or pre-key channels.

Profile picture encryption. When a new picture needs to be uploaded, a random 32-byte label is generated along with an AES256CTRHMACSHA256Key (see Section 11.1). Knowledge of the label and the `identity` of the uploader is sufficient to download the encrypted picture (using the user data API of the server, see Section 75.2), and the authenticated encryption key makes it possible to decrypt the picture and verify that it was not manipulated by the server.

40 Owned Identity Transfer Protocol

40.1 Purpose and High Level View

The Owned Identity Transfer Protocol allows a user to transfer one of their owned identities (their profile) from a device where it already exists, the *source* device, to a new device, the *target* device, which then becomes a new device of this owned identity. As the two devices do not share any secret at the beginning of the protocol, all messages are relayed by the transfer server through the Transfer WebSocket API (see Section 79). This server is not trusted: the protocol uses a SAS, verified by the user, to make sure no man-in-the-middle can obtain the owned identity. The protocol works as follows:

1. On the target device, the engine generates an *ephemeral identity*: a fresh owned cryptographic identity (key pairs for authentication and KEM, and a MAC key, see Section 19), with a placeholder server URL. This identity is only used during the protocol.
2. The source device connects to the transfer server and obtains an 8-digit session number, which is displayed to the user.
3. The user enters this session number on the target device. The target device connects to the transfer server and sends its ephemeral identity to the source device (in the clear).
4. The source device picks a random 32-byte $\texttt{seedForSas}_{source}$ and commits to it (see below). It sends to the target device, encrypted for the ephemeral identity, its connection identifier, the owned identity to transfer, and the commitment.
5. The target device checks that the connection identifier matches that of the connection relaying the message and that it does not already have this owned identity. It computes $\texttt{seedForSas}_{target}$ and sends it, along with its device name, to the source device, encrypted for the transferred identity.
6. The source device decrypts this message and sends the decommitment to the target device,

encrypted for the ephemeral identity. It then computes the 8-digit SAS.

7. The target device opens the commitment, computes the same SAS, and displays all 8 digits.
8. The user types the SAS displayed on the target device into the source device. If it does not match the SAS computed by the source device, the user is asked to type it again. At this point, users without a multi-device license also choose which of their current devices should remain active.
9. If the owned identity is managed by a Keycloak server enforcing transfer restrictions, the target device must first prove that the user successfully authenticated on this Keycloak server (see below).
10. The source device sends a snapshot of the owned identity to the target device, encrypted for the ephemeral identity, and closes its connection. This snapshot contains everything needed to restore the owned identity: the private keys, the contacts, the groups, etc. It optionally also contains the `deviceUid` of the device to keep active.
11. The target device restores the owned identity from the snapshot and registers itself as a new device of this identity. If a device to keep active was specified, the target device requests that this device be set as unexpiring, as soon as its own registration is done (see Section 73.1).

40.2 Cryptographic Details

Encryption. Except for the first message from the target device (containing its ephemeral identity), all payloads relayed by the transfer server are encrypted with PublicKeyEncryption.encrypt (see Section 17): payloads sent by the source device are encrypted for the ephemeral identity, and payloads sent by the target device are encrypted for the transferred identity. More precisely:

- step 4: encodeList(encodeString(sourceConnectionId), encodeBytes(`identity`), encodeBytes(commitment)), where `identity` is the transferred identity;
- step 5: encodeList(encodeString(deviceName), encodeBytes($\texttt{seedForSas}_{target}$));
- step 6: the decommitment;
- step 10: encodeList(encodeDictionary(snapshot)), or encodeList(encodeDictionary(snapshot), encodeBytes(deviceUidToKeepActive)).

Commitment and seeds. The commitment is computed with the scheme of Section 9, using the transferred `identity` as the `tag` and $\texttt{seedForSas}_{source}$ as the `value`. The $\texttt{seedForSas}_{target}$ is computed deterministically from the commitment: it is the deterministic seed of the ephemeral identity, computed with the tag `0x56` and the commitment as diversification value (see Section 19.1).

SAS computation. Both devices compute the SAS with the ComputeSAS procedure of Section 26, using $\texttt{seedForSas}_{source}$ as $\texttt{seedForSas}_{Alice}$, $\texttt{seedForSas}_{target}$ as $\texttt{seedForSas}_{Bob}$, and the ephemeral identity as $\texttt{identity}_{Bob}$. The SAS thus depends on the ephemeral identity received by the source device: if the transfer server replaced the ephemeral identity of the target device with its own, the SAS displayed on the target device would not match the SAS computed by the source device. As the source device commits to its seed before receiving $\texttt{seedForSas}_{target}$, and $\texttt{seedForSas}_{target}$ is determined by the commitment, an adversary cannot choose the seeds so that the two SAS match, except with probability 10^{-8}. The snapshot is only sent once the user has typed the correct SAS on the source device.

Keycloak transfer restriction. When the owned identity is managed by a Keycloak server enforcing transfer restrictions (see Part X), after the SAS is verified:

- the source device sends to the target device, encrypted for the ephemeral identity, the configuration needed to authenticate on the Keycloak server: a JSON containing the server URL (`server`), the client ID (`cid`) and the client secret (`secret`);
- the user authenticates on the Keycloak server from the target device, which then requests a *transfer proof* from the Keycloak server (see Part X): a JWS signed by the Keycloak server, containing the session number (`session_id`), the SAS (`sas`), the transferred identity (`identity`), and the Keycloak user ID (`keycloak_id`);
- the target device sends this transfer proof to the source device, encrypted for the transferred identity;
- the source device checks the signature of the transfer proof with the signature key of the Keycloak server, and checks that the session number, the SAS, the identity and the Keycloak user ID it contains match those of the current transfer. Only then is the snapshot sent.

41 Keycloak Binding and Unbinding Protocol

41.1 Purpose and High Level View

An owned identity can be bound to a Keycloak server (see Part X), for example when the user configures their profile with a Keycloak configuration provided by their company, and unbound from it later. The binding itself (authentication on the Keycloak server, upload of the identity, etc.) is performed by the application on one device. The Keycloak Binding and Unbinding Protocol then takes care of updating the engine state on this device and of propagating the binding or unbinding to the other devices of the owned identity:

- on the device where the binding is performed, the owned identity is marked as managed by the Keycloak server, and the Keycloak parameters are stored: the Keycloak user ID, the server URL, the client ID and client secret, the Keycloak server public keys (JWKS), and the signature key used by the server to sign user details. All the contacts are then checked again to determine which ones are certified by this Keycloak server (see Section 29). Finally, the binding, with all the above parameters, is propagated to the other devices of the owned identity;
- upon reception of a propagated binding, a device performs the same operations (binding and check of the contacts) and notifies the application, so that it can synchronize with the Keycloak server;
- an unbinding is processed and propagated in the same way: the owned identity is no longer marked as managed by a Keycloak server, the Keycloak parameters are deleted, the contacts are no longer considered as certified by this Keycloak server, the groups managed by this Keycloak server are deleted, and the details of the owned identity are updated (removing the signed user details, the position and the company) and published to all contacts (see Section 39).

41.2 Cryptographic Details

There is no cryptography involved in this protocol. Propagated bindings and unbindings are sent through oblivious channels or pre-key channels with the other devices of the owned identity, and are only accepted when received from another device of the same owned identity.

42 Owned Identity Deletion Protocol

42.1 Purpose and High Level View

This protocol deletes an owned identity. The user can choose to delete it only from the current device (local deletion), or from all their devices (global deletion). In the latter case, the owned identity is meant to disappear for good: its contacts and groups are notified so that they also remove it.

The protocol starts on the device where the user requests the deletion:

- the pending messages and protocol instances of the owned identity are deleted, and the owned identity is marked for deletion;
- for a global deletion, the deletion is propagated to the other devices of the owned identity. Each of them runs the same protocol, but only deletes the owned identity locally: contacts and groups are only notified by the device where the deletion was requested;
- if the owned identity is active, the current device is deactivated on the server (see the `/deviceManagement` entry point in Part XII).

Once the device is deactivated, the device where the deletion was requested notifies the other users:

- for a local deletion, the other devices of the owned identity are asked to run an Owned Device Discovery Protocol (see Section 36), and the contacts receive a contact device discovery request (see Section 31), so that everyone quickly stops using the deleted device;
- for a global deletion:
  - in groups v1 owned by the owned identity, all the members and pending members are kicked. For other groups v1, the group owner is notified that the owned identity left the group;
  - groups v2 (except those managed by a Keycloak server) where the owned identity is the only administrator are disbanded: the group blob is deleted from the server and all the members are kicked. The owned identity leaves all the other groups v2 by putting a leave entry in the group log on the server (see Part VI);
  - each contact receives a notification that the owned identity was deleted.

Finally, the server session, all the channels and the owned identity itself (including its private keys) are deleted from the device. On devices that received a propagated global deletion, the application is notified that the owned identity was deleted from another device.

When a contact receives the deletion notification, it propagates it to its other devices and removes the deleted identity: all the channels with it are deleted, it is removed from the groups v1 (and the group is left if the deleted identity was its owner) and from the groups v2 (if the contact is an administrator of a group v2, it also starts a group update to remove the deleted identity from the group), and it is finally deleted from the contacts.

42.2 Cryptographic Details

The propagation of a global deletion to the other devices of the owned identity is sent through oblivious channels or pre-key channels, and is only accepted when received from another device of the same owned identity.

Contacts are notified through an asymmetric channel, in broadcast to all their devices, so that

the notification reaches all of them even if no channel was established with some of these devices. The notification contains a signature of the identity of the contact, made with the private key of the deleted identity and the `ownedIdentityDeletion` signature prefix (see Table 1). Upon reception, the contact checks this signature with the public key of the deleted identity, which authenticates the notification. To prevent replays, each device keeps a list of the signatures it already received and ignores any notification whose signature is in this list.

The signatures used to disband or leave groups v2 are described in Part VI.

43 Synchronization Protocol

43.1 Purpose and High Level View

The aim of the Synchronization Protocol is to keep all the devices of an owned identity in sync. In its current version, it does not detect discrepancies between devices: it only propagates some changes made on one device to the other devices. A more complete version, able to detect and resolve discrepancies between devices, is planned for the future.

When the user changes a setting or a local property on one of their devices (a nickname, a personal note, a pinned discussion, etc.), the change is described by a *sync atom* and sent to all the other devices of the owned identity, which apply it. Each sync atom is sent in its own protocol instance, with a single message.

A sync atom is encoded as a list whose first element is its type (an `int`), followed by the parameters of the change. Table 4 lists the existing types. Depending on its type, a received sync atom is either applied by the engine (for the atoms that make the device trust the published details of a contact or a group), or passed to the application, which applies it. An atom type unknown to a client is ignored.

43.2 Cryptographic Details

There is no cryptography involved in this protocol. Sync atoms are sent through oblivious channels or pre-key channels with the other devices of the owned identity, and are only accepted when received from another device of the same owned identity.

44 Group Invitation Protocol

44.1 Purpose and High Level View

Note that groups v1 have been superseded by groups v2 (see Part VI). New groups are always created as groups v2, but groups v1 are still supported.

This protocol allows a group owner to invite a user to join the group. When a group is created (see Section 45), an instance of this protocol is run independently with each pending group member. Every time a user is added to the group, this protocol is run with them.

This protocol takes as input the `identity` of the contact to invite, the group information (group identifier and group details) as well as the set of all group members and pending members

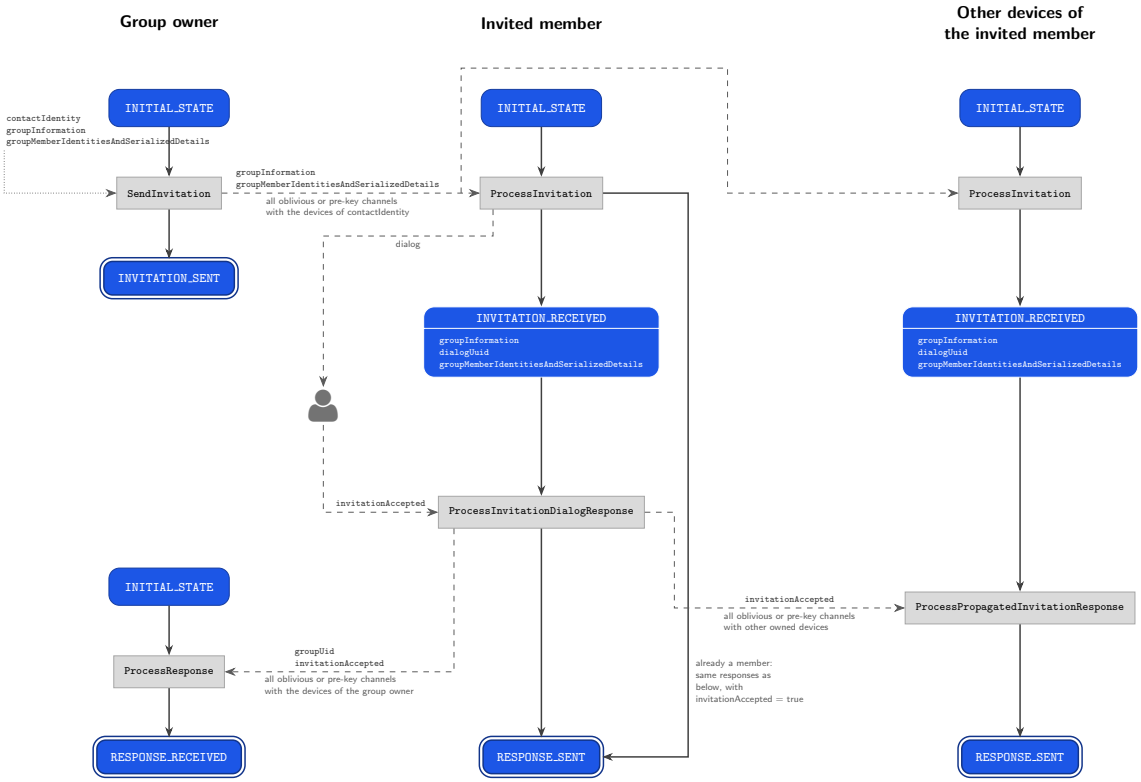


Figure 9: Group Invitation Protocol (groups v1, superseded by groups v2)

Type	Change	Parameters
0	contact nickname	contact identity, nickname (optional)
1	group v1 nickname	group owner, group UID, nickname (optional)
2	group v2 nickname	group identifier, nickname (optional)
3	contact personal note	contact identity, note (optional)
4	group v1 personal note	group owner, group UID, note (optional)
5	group v2 personal note	group identifier, note (optional)
6	own profile nickname	nickname (optional)
7	contact custom color	contact identity, hue (optional)
8	contact read receipts	contact identity, boolean (optional)
9	group v1 read receipts	group owner, group UID, boolean (optional)
10	group v2 read receipts	group identifier, boolean (optional)
11	pinned discussions	list of discussion identifiers, boolean (ordered or not)
12	trust contact details	contact identity, serialized details to trust
13	trust group v1 details	group owner, group UID, serialized details to trust
14	trust group v2 details	group identifier, version to trust
15	default read receipts setting	boolean
16	auto-join groups setting	category (string)
17	bookmarked message	message identifier, boolean
18	archived discussions	list of discussion identifiers, boolean
19	discussions mute	list of discussion identifiers, mute parameters
20	unarchive on notification setting	boolean
21	last app rating	rating, timestamp
22	stop suggesting a contact	contact identity
23	preferred reaction	discussion identifier (optional), emoji (optional)
24	unread discussions	list of discussion identifiers, boolean

Table 4: Types of sync atoms.

(including their identity details). This way, the invited member can identify who is in the group even if they are not in contact with them.

When receiving an invitation to join a group, if the user is already a member of the group, the invitation is automatically accepted. Otherwise, the user is asked whether to accept it or not. Depending on a user setting (automatically accept invitations from nobody, from one-to-one contacts only, or from everyone), the application may automatically accept the invitation on behalf of the user. The same setting applies to groups v2 invitations.

44.2 Cryptographic Details

There is no cryptography involved in this protocol. All messages are exchanged through oblivious channels or pre-key channels, both of which guarantee the authenticity of the sender (for pre-key channels, thanks to the signature of the sender, see Section 24.1).

45 Group Management Protocol

45.1 Purpose and High Level View

Note that groups v1 have been superseded by groups v2 (see Part VI), but are still supported.

This protocol is in fact a collection of "1-step" protocols related to group management which all

start from an empty initial state and end in a final state. It uses a deterministic `protocolUid` (see above). Actions performed on one device of a user are also propagated to their other devices.

The micro-protocols are:

- Initiate the creation of the group: creates the group in the database and launches all the group invitation protocols (see Section 44)
- Notify group members that the members of the group have changed (with the corresponding step to process group members change on the member side). On the group owner side, this micro-protocol also takes care of uploading the group "profile picture" if required, and of downloading it on the members side.
- Add members to a group (and launch the group invitation protocols)
- Remove members from a group (and the corresponding member-side step to "get kicked" from a group)
- Reinvite someone to a group after they declined an invitation
- Disband a group when the owner wants to remove everyone (all users receive a "kick" message)
- Leave a group you do not own
- Query the group owner for the latest group members (and the corresponding owner-side step to send group members)
- Two steps to reinvite an actual group member and forcibly push an updated group members list to a member. These steps are used after an oblivious channel is reconstructed (typically after a backup restore) to make sure all group members are in sync with the group owner.

45.2 Cryptographic Details

There is no cryptography involved in this protocol. All messages are exchanged through oblivious channels or pre-key channels, both of which guarantee the authenticity of the sender (for pre-key channels, thanks to the signature of the sender, see Section 24.1). Each group has an owner attached to its definition, and members can check that messages are indeed received from the group owner.

Part VI
Groups v2

46 General Concepts

Legacy Olvid groups have the advantage of being completely decentralized but were limited to one administrator: the creator of the group. The initial group creator was linked to this group forever, with no possibility of transferring the group ownership. In order to change this, we decided to design groups v2 that would allow multiple administrators, each of them having the capacity to invite new members and modify the group. The presence of multiple administrators yielded a new complexity: we had to be able to handle concurrent modifications of the group. The best solution we found to handle this was to rely on a single "source of truth", an encrypted blob describing the group structure uploaded to the server.

Of course, as the server is considered malicious in our security model, the structure of the group had to be secured with cryptographic techniques to avoid any form of server manipulation and prevent the server from learning the content of these blobs.

Blob encryption. The blob containing the group structure is encrypted before being uploaded to the server. The encryption key is derived from two seeds: a **main seed** that stays constant for the whole group's lifespan and that should never be transmitted through an asymmetric channel (to ensure forward secrecy of the group structure), and a **version seed** that is renewed whenever members are removed from the group or have left it, to ensure former members can no longer access the blob's contents.

Administration public key. As the server does not know who is an administrator of the group, deciding whether a user has the right to overwrite the group's blob on the server requires a proof that they are an administrator. For this, each blob is uploaded with an authentication public key attached to it: modifying the blob requires signing the new blob with the private key associated with this public key.

This authentication private key is shared among all group administrators and is renewed every time an administrator is removed from the group or loses their administrator privileges. This way, a former administrator is not able to overwrite the blob once their administrator privileges are lost.

Group identifier. In order to uniquely identify a group discussion, each Olvid group must have a unique identifier. In legacy groups, the `identity` of the owner of the group was included in this unique group identifier making it impossible for another user to clone a group identifier.

In groups v2, there is no owner `identity` to include and the uniqueness of a group identifier

must be guaranteed differently. This group identifier is composed of:

- a group category: CATEGORY_SERVER (0) or CATEGORY_KEYCLOAK (1)
- the server or Keycloak server URL
- the `groupUid`: a 32-byte identifier computed as the SHA-256 hash of the first block of the administrators chain (see below)

When uploading a blob to the server, the `groupUid` is used as a unique identifier for this server, and the server checks a blob does not already exist for this `groupUid`. When downloading a blob following an invitation to a group, the group members can then check that the `groupUid` matches the hash of the first administrators chain block.

Administrators chain. In order to validate that a group blob was not tampered with by a group member colluding with the server, group members need to be able to check that the blob was signed by a legitimate group administrator. However, as group administrators may change over time, we need to keep track of how an administrator became an administrator of the group. For this, we build some sort of block chain to which a new block is added every time administrators of the group change.

- block n contains the SHA-256 hash of block $n-1$ and the encoded updated list of administrators (their `identity`)
- block n is then signed by the administrator updating the group, who must be a valid administrator of block $n-1$
- block 0 contains 32 random bytes instead of a block hash and the list of initial administrators
- block 0 must be signed by the group creator

When downloading a blob, each group member validates the administrators chain, starting by checking that the hash of block 0 matches the `groupUid`, and verifying each block's signature. Then, the whole blob signature is verified against the list of administrators of the last block.

Group v2 versioning. In order to easily make sure each user has the latest version of the group, the blob contains a version number that must be increased every time the group is updated. This protects users from a malicious server trying to "replay" an old group blob. In addition to checking that the version number is greater than the last known version number, when updating a group, the Olvid application also checks that the last known administrators chain is a prefix of the new administrators chain.

Group member permissions. For each member of the group the blob includes the following information:

- an `identity`
- some identity details (first name, last name, etc.)
- a 16-byte group invitation nonce
- a list of permission strings:
  - `"ga"`: group admin permission, allowing the member to update the group blob
  - `"rd"`: remote delete anything permission, allowing the member to delete messages sent by other users for all group members
  - `"eo"`: edit or remote delete own messages permission
  - `"cs"`: change settings permission, allowing the member to change the group's shared settings (ephemeral settings for example)
  - `"sm"`: send message permission, allowing the member to post messages in this group

The group invitation nonce is used when accepting an invitation to join a group or when leaving a group, so that it is impossible to replay these protocol messages and so that, if you join a group and leave it, even if you are re-invited to the group and messages are delivered out of order, other group members know exactly who agreed to join or not.

Group types. To make editing a group more convenient, when a group v2 is created it is assigned a "group type" that sets default permissions for members and admins of the group. These are some preset permissions that simplify the user interfaces by not having to select fine-grained permissions for each member. There are currently four group types:

- **Standard**: all members are admins and can invite other members. Anyone can send messages, change settings, edit or delete own messages. No one can remote delete other members' messages.
- **Controlled**: admins have the same permissions as in standard groups, but some members do not have admin permission and can only send messages and edit or delete own messages.
- **Read only**: admins have the same permissions as in standard groups, but members only have the permission to edit or delete own messages (they do not have the permission to send messages).
- **Advanced**: admins can be chosen (and have admin, change settings, send messages and edit or remote delete own messages), and one can choose:
  - who can remote delete anything: no one, admins only, or everyone
  - who can send messages: admins only, or everyone

Group v2 log. In legacy groups, leaving a group required notifying the group owner, who would broadcast the information to group members. In groups v2, leaving a group must rely on the server. As simple members do not have the permission to modify the blob, leaving the group consists in uploading to the server a "proof" that you left the group. This proof takes the form of a log item, and the server keeps a log of all such items along with the group blob. Every time the group blob is updated, the log is consolidated inside the blob.

One important point is that such a log item is uploaded without encryption to the server: it must not contain any element allowing the server to know who was a member of the group. To achieve this, log items only contain a signature of the group identifier and of the group invitation nonce of the user, with the authentication key included in this user's `identity`. **Only the signature is included**, not what was signed. As invitation nonces are not public, the server cannot guess who issued the signature. Group members can however check for all group members if a log item corresponds to them leaving the group or not by performing n signature verifications for a group with n members.

Posting messages to a group v2. When posting a message to a group v2 we want to make sure messages are delivered to all group members, even if they have not yet agreed to join the group or if we are not yet aware they agreed to join the group. When a message is posted, it is "prepared" to be sent to every member (pending or not) of the group, but is actually only sent to confirmed members of the group. Every time a pending member confirms they accept to join the group, the prepared (but unsent) messages are resent to them. If a pending member declines to join the group, all prepared messages are discarded.

When resending a message, the original server timestamp when first sending the message is included so that these new members can sort old messages from different group members in the proper order. The recipient only uses this original timestamp if it is earlier than the actual server timestamp of the resent message. In terms of user experience, the benefit of this is that when a

group is created, new members will be able to see the full discussion, from the start, even if their device was offline at the group creation time.

Keycloak groups. In addition to normal groups where the blob is uploaded by an administrator on the server, users of the Olvid Enterprise offer also have access to Keycloak groups, managed directly by the user directory that is in place in the company. These groups can only be modified by users of the Olvid Administration Console (implemented as a Keycloak plugin) and there are no user administrators.

These groups also use a signed blob, but with a much simpler structure as they are downloaded directly from the organization's Keycloak server and can systematically be trusted. Their identifier uses the Keycloak server URL and a random `groupUid` chosen by Keycloak. The signature uses the same Keycloak signature key as for a user's signed details, again in the form of a JWT (see Section 48.8).

47 Cryptographic Structures

Group keys. The keys of a group v2 are the following:

- the **main seed** and the **version seed**, two random 32-byte seeds known to all the members of the group. The blob encryption key is an AuthEncKey computed as KDFFromPRNGWithHMACWithSHA256.compute(`mainSeed`‖`versionSeed`, AuthEncKey);
- the **group admin key pair**, an authentication key pair of the same type as the one of an `identity`. The public key is stored by the server along with the blob, and the private key is only shared with the administrators of the group. It is used to sign the requests modifying or deleting the blob on the server (see the `lockNonce`, `updateGroup` and `deleteGroup` prefixes in Table 1).

When these keys are sent to a member, they are encoded as a dictionary with keys `"ms"` (main seed), `"vs"` (version seed) and, for administrators only, `"ga"` (group admin private key).

Administrators chain. The administrators chain is encoded as a list of blocks. Each block is a list $[\texttt{innerData}, \sigma]$ where:

- `innerData` is the encoding of the list $[\texttt{previousBlockHash}, [\texttt{identity}_1, \ldots, \texttt{identity}_n]]$. `previousBlockHash` is the SHA-256 hash of the encoding of the previous block (32 random bytes for block 0) and $\texttt{identity}_1, \ldots, \texttt{identity}_n$ are the administrators of the group after the block was added, the administrator who created the block always being $\texttt{identity}_1$;
- σ is the signature of `innerData` with the `groupAdministratorsChain` prefix by the administrator who created the block.

The `groupUid` is the SHA-256 hash of the encoding of block 0. To check the integrity of a chain, a member checks that each block contains the hash of the previous block and that each block is signed by one of the administrators listed in the previous block (block 0 being signed by one of its own administrators). When the member already knows a previous version of the chain, this previous version must be a prefix of the new chain, and only the new blocks need to be checked. Otherwise, the member checks that the hash of block 0 matches the `groupUid` of the group identifier.

Group blob. The group blob is an encoded dictionary with the following keys:

- `"ac"`: the administrators chain;

- `"mem"`: the list of the group members (including pending members), each member being encoded as the list [`identity`, [`permissions`], `details`, `nonce`], where `permissions` are the permission strings of the member, `details` are their serialized identity details, and `nonce` is their 16-byte group invitation nonce;
- `"v"`: the version of the blob (an `int`);
- `"det"`: the serialized group details (name, description, etc.);
- `"ph"` (optional): the information needed to download the group photo, encoded as the list [`identity`, `label`, `key`] where `identity` is the member who uploaded the photo on the server, `label` is a random UID identifying the photo on the server and `key` is the AuthEncKey used to encrypt it (see Section 49);
- `"t"` (optional): the serialized group type.

Before being uploaded, the blob is signed and encrypted by the administrator who created or modified it:

- the encoded blob is signed with the `groupBlob` prefix, giving a signature σ;
- the list [`blob`, `identity`, σ], where `identity` is the identity of the administrator, is encoded and padded with zeros to the smallest multiple of 4096 bytes larger than its length;
- the result is encrypted with AuthEnc under the blob encryption key.

When downloading a blob, a member decrypts it, removes the padding (see Section 22.9) and checks that:

- the integrity of the administrators chain is valid;
- `identity` is one of the administrators of the last block of the chain, and σ is a valid signature of the blob by `identity`;
- all the members of the blob with the `"ga"` permission are administrators of the last block of the chain;
- the version of the blob is not smaller than the last version known to the member.

A blob that does not pass these checks is rejected.

48 Groups v2 Protocol

All the operations on a group v2 (creation, invitation, update, leaving, etc.) are performed through the Groups v2 Protocol. Its `protocolUid` is computed deterministically from the group identifier, so that all the devices of all the group members use the same `protocolUid` for a given group. Unless stated otherwise, the protocol messages sent to other group members or to other owned devices are sent through oblivious channels or pre-key channels. The main steps of the group creation, invitation and joining are represented in Figure 10, and those of a group update on Figure 11.

48.1 Group Creation

The user creating a group chooses its members among their contacts, the permissions of each member, the group details (name, description, etc.), its type and, optionally, a photo. The device of the creator then:

- creates block 0 of the administrators chain, containing the creator and the other members chosen as administrators. The group identifier is built from this block: the category is CATEGORY_SERVER, the server URL is the one of the creator's `identity`, and the `groupUid`

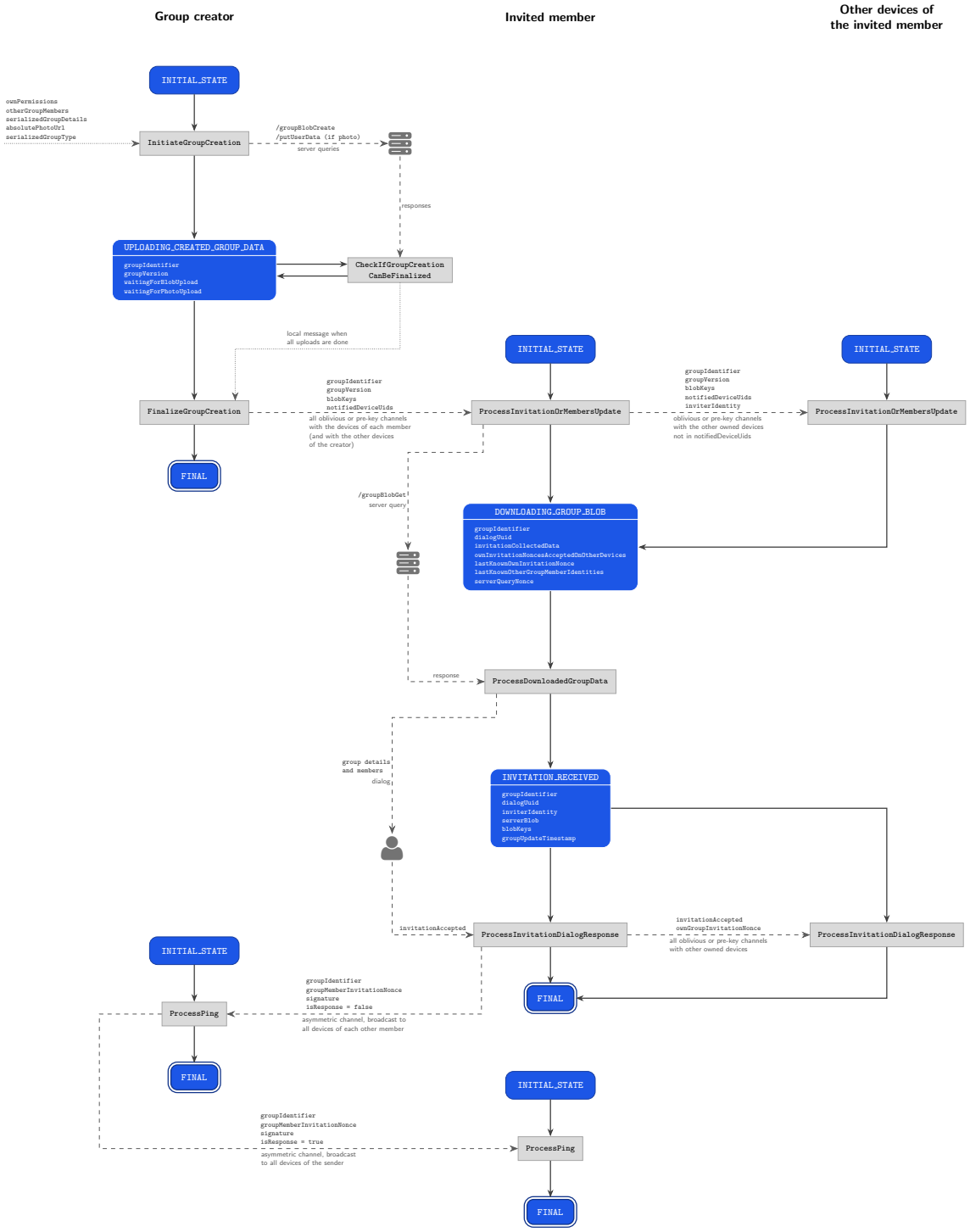


Figure 10: Groups v2 Protocol: group creation, invitation and joining

is the SHA-256 hash of block 0;
- generates a random main seed, a random version seed, a group admin key pair, and a random group invitation nonce for each member (including the creator);
- builds version 0 of the group blob, using the published details of each contact as their member details;
- if the group has a photo, generates a random label and a random AuthEncKey, encrypts the photo with this key and uploads it to the server under this label (see the `/putUserData` entry point in Part XII);
- signs, pads and encrypts the blob (see Section 47), and uploads it to the server along with the public key of the group admin key pair (see the `/groupBlobCreate` entry point in Part XII). If this upload fails, the group is deleted.

Once the blob is uploaded, the creator sends an invitation to each member, containing the group identifier, the blob version and the group keys: the main seed, the version seed and, for administrators only, the group admin private key. The same message, with all the keys, is sent to the other devices of the creator. If one of the members cannot be reached (because the creator has neither an oblivious channel nor a pre-key with any of their devices), the creation is cancelled: the blob is deleted from the server and the group is deleted locally. The processing of the invitation by the members is described in Section 48.2.

48.2 Invitation and Group Joining

Invitation message. Group members learn about a new group, or about a new version of a group they are part of, through an invitation message containing the group identifier, the blob version, the group keys, and the list of the device UIDs of the recipient to which the message was sent. This message is sent by the creator of the group or by the administrator who updated it (see Section 48.3). When the recipient receives it from a contact and some of their devices are not in this list, the message is propagated to these devices, along with the identity of the sender.

The main seed is only ever sent through oblivious channels or pre-key channels. Members with whom the sender has no such channel receive the message through an asymmetric channel (in broadcast to all their devices) without the main seed, and a main seed received through an asymmetric channel is always ignored. This way, the compromise of the long-term private key of an `identity` never gives access to the main seed of a group.

Blob download. Upon reception of an invitation message, the device first checks that the `protocolUid` matches the group identifier. If the group is already known locally with a version greater than or equal to the received one, the message is ignored. Otherwise, the received keys are added to a set of candidate keys: pairs (sender, main seed), version seeds and group admin private keys. For a group already known locally, the keys stored locally are also added to the candidates, as well as the pre-shot version seeds received for this group (see Section 48.3).

The device then downloads the encrypted blob from the server (see the `/groupBlobGet` entry point in Part XII), along with the group log entries, the group admin public key and the timestamp of the last update of the blob. It tries all the candidate main seed and version seed pairs until one decrypts the blob, and checks the decrypted blob as described in Section 47. The log entries are then consolidated: each log entry is checked against the group invitation nonce and the `identity` of each member (see the `groupLeave` prefix in Table 1) and the members who left the group are removed from the blob. The user must be one of the remaining members, otherwise the blob is rejected. If the user is an administrator in the blob, one of the candidate group admin private keys must match the group admin public key returned by the server. If no candidate makes it possible

to decrypt and validate the blob, the device waits for more keys.

Joining the group. For a group the user has not joined yet, the invitation is presented to the user, with the group details, the members, and the identity of the contact who sent the main seed as inviter. The invitation is accepted automatically if it was already accepted on another device of the user, or if the main seed was received from another device of the user (the group was created or joined from this other device). When the user accepts the invitation:

- the response is propagated to the other devices of the user, along with the group invitation nonce of the user;
- the group is created locally, all the other members being considered as pending members;
- a ping is sent to each other member.

Declining the invitation is equivalent to leaving the group (see Section 48.5).

Pings. The blob does not tell which members have joined the group and which members are still pending: each device keeps track of this locally, thanks to pings. A ping is sent through an asymmetric channel, in broadcast to all the devices of the recipient, and contains the group identifier, the group invitation nonce of the sender, a flag indicating whether it is a response to a ping, and a signature of the encoded group identifier, the nonce and the `identity` of the recipient with the `joinGroup` prefix. Upon reception of a ping, a device:

- checks that the `protocolUid` matches the group identifier, and that the signature is not in its list of already received signatures, to prevent replays;
- propagates the ping to its other owned devices;
- if it has joined the group, looks for the members and pending members of the group whose group invitation nonce is the one in the ping, and checks the signature with their `identity`. If the signature is valid, it is added to the list of received signatures, and the sender is considered as a member who joined the group. If the sender is not yet a contact, they are added as a (not one-to-one) contact, with this group as trust origin;
- if the ping is not a response, sends back its own ping (with the response flag), so that the sender also learns that the recipient joined the group.

Pings are also sent to new members when a member processes an update of the group, and to the sender of an invitation message when this sender is still a pending member.

48.3 Group Update

Only group administrators can update a group, as updating the blob on the server requires the group admin private key. An update can add or remove members, change the permissions of members, the group details, the group type or the group photo. When updating a group, the administrator also refreshes the details of each member in the blob with the latest published details they know.

Pre-shot version seed. If the update removes some members, a new random version seed is generated at the very beginning of the update, and immediately sent to all the other members and pending members of the group (except those being removed) and to the other devices of the administrator, through oblivious channels or pre-key channels. This is called a *pre-shot* version seed: it ensures that the remaining members can decrypt the new blob, even if the update is interrupted before the new keys are distributed at the end of the update. Upon reception, a pre-shot version seed is stored for 60 days and used as a version seed candidate when decrypting the blob of this

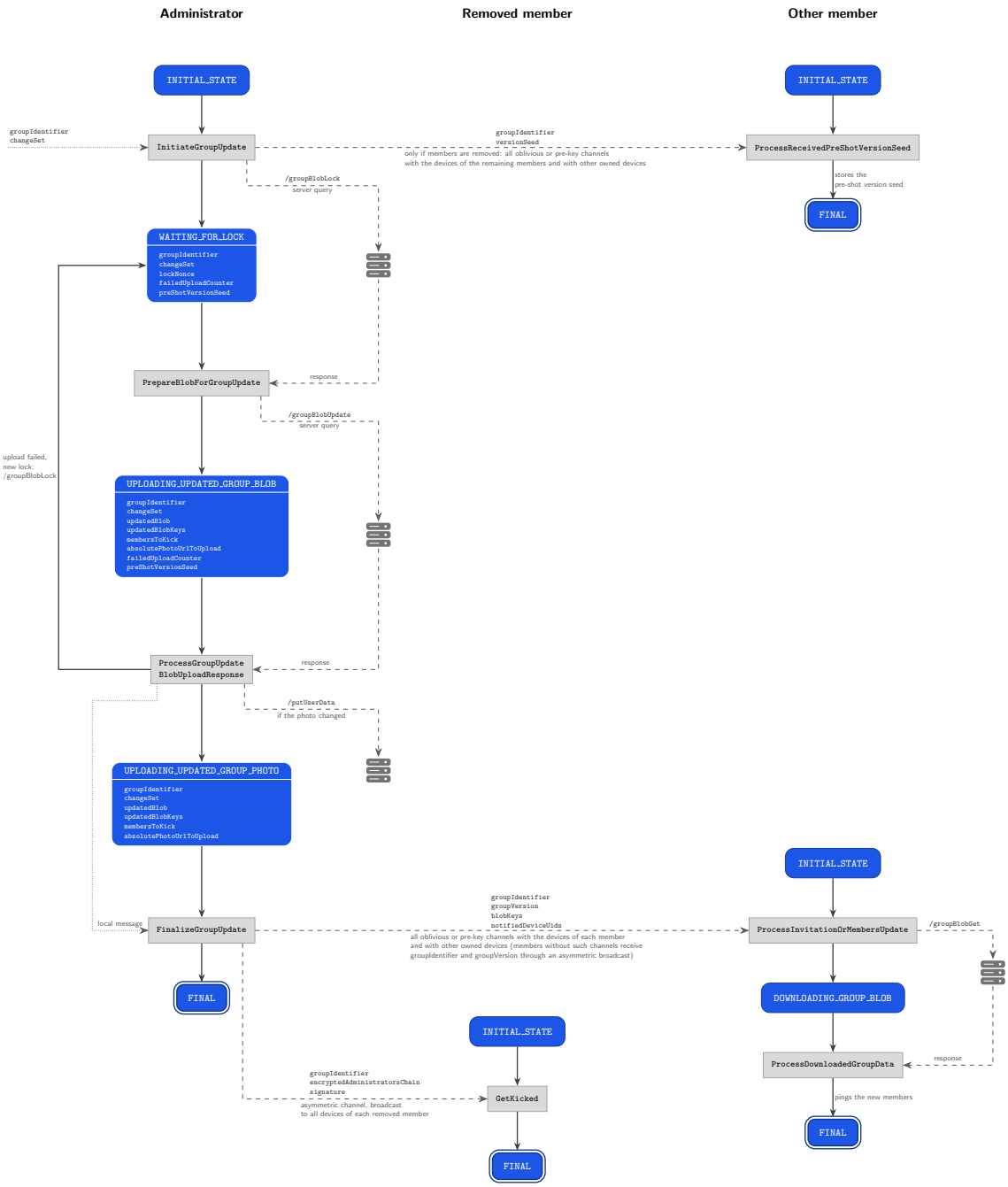


Figure 11: Groups v2 Protocol: group update

group. The sender is not checked, as it might be an administrator the recipient does not know yet.

Locking the blob. The administrator then locks the blob on the server: they generate a random 32-byte lock nonce and sign it with the `lockNonce` prefix and the group admin private key. The server checks this signature with the group admin public key it stores, locks the blob, and returns the current encrypted blob, along with the group log entries and the group admin public key (see the `/groupBlobLock` entry point in Part XII). The lock is released if the update is not performed in time.

Building the new blob. The administrator decrypts the blob with the main seed and either the current version seed or one of the pre-shot version seeds received, checks it as described in Section 47, consolidates the log entries, and applies the changes:

- each new member gets a random group invitation nonce. Each new member must be reachable through an oblivious channel or a pre-key channel, otherwise the update is cancelled;
- the version is incremented by one;
- if the set of administrators changes, a new block is appended to the administrators chain. It lists the new administrators, the administrator performing the update first, and is signed by them;
- if at least one administrator is removed from the group or loses their administrator permission, a new group admin key pair is generated.

The version seed used to encrypt the new blob is the pre-shot version seed if one was generated, and the current version seed otherwise. However, if the consolidation of the log entries removed some members who could know the version seed that would be used, a new version seed is generated and pre-shot to the remaining members. This way, the version seed changes whenever members are removed or leave the group, and only then.

Uploading the new blob. The new blob is signed, padded and encrypted (see Section 47). The administrator then signs, with the `updateGroup` prefix and the current group admin private key, the concatenation of the lock nonce, the encrypted blob and the encoded group admin public key to use from now on (the new one if a new key pair was generated, the current one otherwise). The server checks this signature with the group admin public key it stores, then replaces the blob and the group admin public key, and purges the group log entries, which were consolidated in the new blob (see the `/groupBlobUpdate` entry point in Part XII). If the photo changed, or if the current photo was uploaded by another administrator, the photo is uploaded again with a new random label and a new random AuthEncKey.

Distributing the keys. Once the blob is uploaded, the administrator updates the group locally and sends an invitation message (see Section 48.2) to all the members and pending members of the group, including the new ones, and to their own other devices. The message contains the new version and the group keys: the main seed (only through oblivious channels or pre-key channels), the version seed, and the group admin private key for administrators only. Members with whom the administrator has no oblivious channel or pre-key channel receive the message through an asymmetric channel, without the main seed. Finally, the removed members are kicked (see Section 48.4).

Processing an update. Members process the invitation message as described in Section 48.2: they download the new blob, check it (in particular, the administrators chain they know must be a

prefix of the new chain, and the version must not be smaller than the version they know), and update their local copy of the group. New members are added as pending members and receive a ping. When downloading a blob, the server also returns the timestamp of its last update, which the application uses to date the group update.

48.4 Member Removal

As the version seed is renewed when members are removed, a removed member can no longer decrypt the new blob, and thus cannot learn from the blob that they were removed. Instead, the administrator who removed them sends them a kick message. The administrator:

- encrypts the encoded administrators chain of the new blob with AuthEnc, under the key computed as KDFFromPRNGWithHMACWithSHA256.compute(`mainSeed`, AuthEncKey);
- for each removed member (or pending member), signs the concatenation of this encrypted chain and the group invitation nonce of the removed member with the `groupKick` prefix;
- sends to each removed member, through an asymmetric channel in broadcast to all their devices, a kick message containing the group identifier, the encrypted chain and the signature.

Upon reception of a kick message, the device of the removed member:

- checks that the `protocolUid` matches the group identifier, and propagates the message to its other owned devices;
- decrypts the administrators chain with the main seed of the group, and checks its integrity: the chain it already knows must be a prefix of the received chain;
- checks that the signature was produced by one of the administrators of the last block of the received chain, using its own group invitation nonce;
- if all these checks pass, deletes the group (or the pending invitation) and the pre-shot version seeds it received for this group.

48.5 Leaving a Group or Declining an Invitation

A member who is not an administrator cannot modify the blob. To leave a group, or to decline an invitation, a member instead puts a log entry on the server, which administrators will later consolidate in the blob (see the group log paragraph in the general concepts above). An administrator cannot leave a group if they are its only administrator.

When the user leaves a group, their device:

- propagates the request to their other devices, along with their group invitation nonce. These devices simply delete the group locally;
- signs the encoded group identifier and their group invitation nonce with the `groupLeave` prefix, and puts this signature as a log entry on the server (see the `/groupLogPut` entry point in Part XII);
- deletes the group locally, along with the pre-shot version seeds received for this group;
- once the log entry is on the server, notifies all the other members through an asymmetric channel (in broadcast to all their devices). This notification only contains the group identifier.

Declining an invitation works the same way: the response is propagated to the other devices of the user, the log entry is put on the server, and the other members are notified.

The notification is not authenticated: it only triggers a new download of the blob. Upon reception, a member propagates it to their other devices and downloads the blob and the log entries from the server, as described in Section 48.2. The consolidation of the log entries reveals

who left the group, and these members are removed from the local copy of the group. The next update of the group by an administrator consolidates the log entries in the blob and renews the version seed (see Section 48.3).

48.6 Group Disbanding

Any administrator of a group can disband it. Their device:

- deletes the blob from the server: the request contains a signature of an empty challenge with the `deleteGroup` prefix and the group admin private key, which the server checks with the group admin public key it stores (see the `/groupBlobDelete` entry point in Part XII);
- once the blob is deleted, propagates the disbanding to the other devices of the user, which delete the group locally;
- kicks all the other members and pending members of the group, as described in Section 48.4, using the current administrators chain;
- deletes the group locally, along with the pre-shot version seeds received for this group.

48.7 Group Keys After Channel Creation

The group keys are only sent to devices with which an oblivious channel or a pre-key channel exists. When a new oblivious channel is confirmed with a device of a contact (for instance a new device of this contact), the device sends it, through this channel, a single message containing, for each group v2 shared with this contact (where the contact is a member or a pending member), the group identifier, the group version and the group keys. The group admin private key is only included if the contact is an administrator of the group and the device has this key. When the new channel is with another device of the user, all the groups of the user and all their keys are sent.

Any member of a group can send these keys, not only administrators. Upon reception, each group in the message is processed as an invitation message received from the sender of the keys (see Section 48.2): the keys are added to the candidate keys and the blob is downloaded, so that the new device can join the group or update it.

48.8 Keycloak Groups

Keycloak groups (category CATEGORY_KEYCLOAK) are entirely managed by the Keycloak server: they have no administrators, no main seed, no version seed and no group admin key, and their members can neither update, leave nor disband them. Their identifier is made of the URL of the Keycloak server and of a `groupUid` chosen by the Keycloak server.

The devices of a user bound to a Keycloak server regularly retrieve from this server three lists of signed elements (see the `groups` entry point in Part X): signed group blobs, signed group deletions and signed group kicks. Each of them is a JWT signed with the Keycloak signature key (the same key as for signed user details), and is ignored if its signature is invalid. Each element contains a timestamp, which is compared with the timestamp of the last modification of the local copy of the group:

- a signed group blob contains the `groupUid`, the group details, the label and the AuthEncKey of the group photo, the shared settings of the group, and the list of its members, with, for each of them, their `identity`, their signed details, their permissions and their group invitation nonce. A blob older than the validity period of Keycloak signatures (60 days) is ignored. If

the group is unknown, it is created, the user becoming a member without any invitation, and a ping is sent to all the other members (see Section 48.2). If the group is known and the blob is more recent, the group is updated, and a ping is sent to the members whose group invitation nonce changed. The group photo is downloaded as described in Section 49;
- a signed group deletion contains a `groupUid`: the group is deleted locally, unless it was modified after the deletion;
- a signed group kick contains a `groupUid` and an `identity`: if this `identity` is the one of the user, the group is deleted locally, unless it was modified after the kick.

Keycloak groups are also deleted when the user unbinds their identity from the Keycloak server (see Section 41).

49 Groups v2 Picture Download Protocol

The group photo is not part of the blob: it is encrypted with a random AuthEncKey and stored on the server by the administrator who set it (or, for Keycloak groups, by the Keycloak server), and the blob only contains the information needed to download and decrypt it, in its `"ph"` entry (see Section 47).

When a device processes a blob containing photo information and does not have this photo yet, it starts an instance of the Groups v2 Picture Download Protocol, with a random `protocolUid`. This protocol:

- downloads the encrypted photo from the server, using the `identity` of the uploader and the label found in the photo information (see the `/getUserData` entry point in Section 75.2). For Keycloak groups, the encrypted photo is downloaded from the Keycloak server using the label (see the `getData` entry point in Part X);
- decrypts the photo with the AuthEncKey found in the photo information, and stores it as the photo of the group.

Part VII
Backups

The Olvid engine implements mechanisms allowing a user to back up their profiles (owned identities), including their long-term private keys, their contacts and the groups they belong to, as well as some application settings. Backups never contain any messages.

Backups should of course never be done in clear, and encrypting them with a password would be far too weak for most users. Backups are thus encrypted with keys derived from strong random seeds, called backup seeds.

The current backup mechanism is described in Section 50. It replaced a previous mechanism, described in Section 51, which can no longer be used to create new backups, but whose backups can still be restored.

50 Backups

50.1 Overview

Backups rely on two kinds of backup seeds:

- each owned identity has a **profile backup seed** (see Section 19), used to encrypt the *profile backups* of this owned identity. A profile backup contains everything needed to restore the owned identity on a new device. The profile backup seed is the same on all the devices of the owned identity, and is never shown to the user;
- each device on which the user activates backups has a **device backup seed**, used to encrypt the *device backup* of this device. The device backup contains the list of the owned identities of the device, along with their profile backup seeds. The device backup seed is the backup key shown to the user, which they need to restore their backups.

To restore a profile, the user enters their device backup seed. It is used to decrypt the device backup, which contains the profile backup seeds of all the profiles of the device, which in turn are used to decrypt the profile backups.

50.2 Backup Seeds

A backup seed is a 160-bit (20-byte) seed. Device backup seeds are generated at random when the user activates backups on a device, and a device has at most one active device backup seed. Profile backup seeds are generated at random when an owned identity is created, or derived deterministically from the secret MAC key for owned identities created before backups were introduced (see Section 19). They are transferred to the other devices of the owned identity along with the rest of

the owned identity (see Section 40).

A device backup seed is presented to the user as 8 strings of 4 characters (see Figure 12). Each of these 32 characters contains 5 bits of entropy with the correspondence of Table 5. When displayed to the user, the first of the corresponding characters is used (number or capital letter), but when the user enters the key for a restore, all equivalent characters are accepted.

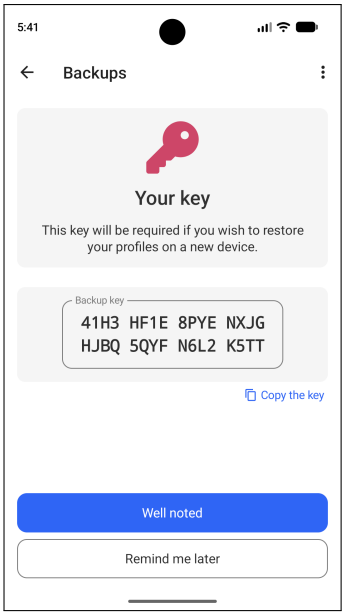


Figure 12: Example of an Olvid backup key.

dec.	hex.	chars	dec.	hex.	chars	dec.	hex.	chars	dec.	hex.	chars
0	0x00	0, O, o	8	0x08	8	16	0x10	G, g	24	0x18	Q, q
1	0x01	1, I, i	9	0x09	9	17	0x11	H, h	25	0x19	R, r
2	0x02	2, Z, z	10	0x0a	A, a	18	0x12	J, j	26	0x1a	T, t
3	0x03	3	11	0x0b	B, b	19	0x13	K, k	27	0x1b	U, u
4	0x04	4	12	0x0c	C, c	20	0x14	L, l	28	0x1c	V, v
5	0x05	5, S, s	13	0x0d	D, d	21	0x15	M, m	29	0x1d	W, w
6	0x06	6	14	0x0e	E, e	22	0x16	N, n	30	0x1e	X, x
7	0x07	7	15	0x0f	F, f	23	0x17	P, p	31	0x1f	Y, y

Table 5: Backup seed character to 5-bit value correspondence.

The application may also store the device backup seed in the password manager of the platform, so that the user does not have to type it during a restore.

50.3 Key Derivation

The keys used for a backup are derived from its backup seed as follows:

1: **procedure** DeriveKeysV2(backupSeed)
2: `/* The backupSeed is 0-padded to 32-byte length */`

```
    seed ← backupSeed||0...0
    Initialize prng, a PRNGWithHMACWithSHA256 using seed
    backupKeyUid ← prng.bytes(32)
    encryptionKey ← AuthEnc.generateKey(prng)
    authenticationKeyPair ← AuthenticationOverEC.generateKeyPair(prng)
end procedure
```

The backupKeyUid identifies the backups encrypted with this seed on the server, the encryptionKey is used to encrypt them, and the authenticationKeyPair is used to authenticate to the server when uploading or deleting them.

50.4 Storage on the Server

Backups are stored on the Olvid server (see Part XII). Device backups are stored on the server of the application, and profile backups on the server of the owned identity. All the backups encrypted with a given seed are grouped under its backupKeyUid, in *threads*, each thread containing the latest *version* of a backup:

- the device backup is stored in a thread whose identifier is 32 `0xFE` bytes;
- as all the devices of an owned identity share the same profile backup seed, each device uses its own thread for the profile backups of each owned identity: its identifier is a random 32-byte UID, chosen by the device and stored locally;
- the version is an integer that increases with each upload in a thread.

Before the first upload with a given seed, the device registers the backupKeyUid on the server, along with the public key of the authenticationKeyPair. Uploading or deleting a backup then requires a signature with the private key of this key pair, with the `backupUpload` or `backupDelete` prefix (see Table 1), of the encoded list of the backupKeyUid, the thread identifier, the version and, for an upload, the encrypted backup. Listing the backups of a backupKeyUid (thread identifiers, versions and download URLs) and downloading them requires no authentication: the confidentiality and integrity of backups only rely on their encryption.

50.5 Backup Encryption

A backup is an encoded dictionary (see Part III) containing a part generated by the engine and a part generated by the application, each in JSON format. Before being uploaded, this encoded dictionary is padded with zeros to the next multiple of 512 bytes, to hide its exact size, then encrypted with AuthEnc using the encryptionKey. When downloading a backup, the device decrypts it (which checks its integrity) and removes the padding (see Section 22.9).

50.6 Backup Contents

Device backup. For each owned identity of the device (including hidden ones), the device backup contains:

- the `identity`, its published details and whether it is managed by a Keycloak server;
- its profile backup seed;
- the name given to the profile by the user.

Profile backup. A profile backup contains a snapshot of the owned identity. This snapshot is the same as the one sent during an owned identity transfer (see Section 40). It contains:

- the `identity`, its private keys and its published details;
- the profile backup seed;
- if the owned identity is bound to a Keycloak server, the information related to this server (URL, client identifier and secret, signature key, user identifier, transfer restriction, etc.);
- the contacts, with their details, trust level and trust origins, and whether they are one-to-one, revoked or forcefully trusted;
- the groups v1, with their details and members;
- the groups v2, with their identifier, version, keys (main seed, version seed and, for administrators, group admin private key), administrators chain, own group invitation nonce, and members;
- application data: some global settings, and for each contact and group, the name and note given by the user, and the discussion settings.

The profile backup also contains the name of the device that made the backup, its platform, and the time at which the backup was made. Profile photos are not part of backups: only their label and key are, so that they can be downloaded from the server after a restore.

50.7 Backup Creation

When the user activates backups on a device, a device backup and a profile backup of each owned identity are immediately made. Then, new backups are made when the corresponding data changes, as well as periodically. Only the latest version of each backup is kept on the server.

50.8 Backup Restore

To restore a profile, the user enters their device backup seed (or the application retrieves it from the password manager of the platform). The device then:

- derives the keys of the device backup seed, lists the backups of its backupKeyUid, and downloads and decrypts the device backup. The user chooses the profile to restore among the profiles it contains;
- derives the keys of the profile backup seed of this profile, downloads and decrypts all the profile backups of this owned identity (one per device that made backups of it), and presents them to the user, the most recent first;
- if the profile is bound to a Keycloak server that restricts transfers, requires the user to authenticate on this Keycloak server before restoring the profile;
- restores the owned identity from the chosen profile backup, with a new random `deviceUid`, as well as its contacts, groups and application data.

After the restore, the restored device downloads the blobs of all the groups v2, runs an Owned Device Discovery Protocol (see Section 36) and a device discovery for each contact (see Section 35), which then triggers the creation of channels with the devices of the contacts, and downloads the profile photos.

50.9 Backup Deletion

The user can delete a specific profile backup from the server: this requires a signature with the authenticationKeyPair derived from the profile backup seed. When the user deletes or resets the

device backup seed of a device, the device backup is deleted from the server, along with the profile backups made by this device when backups are deactivated.

51 Legacy Backups

Legacy backups were used before the backup mechanism of Section 50. A legacy backup is protected by a legacy backup key, which is a backup seed with the same format as described in Section 50.2. New legacy backup keys can no longer be generated, but a device that already had one keeps making legacy backups until a device backup seed is generated, at which point the legacy backup key and legacy backups are deleted from the device. Legacy backups are exported to a file, or uploaded automatically to the cloud storage of the user (iCloud for the iOS client, Google Drive for the Android client). They can still be restored on a device with no owned identity.

51.1 Key Derivation

The backup seed itself is never stored in the application. It is displayed to the user, then a set of keys is derived from this seed, and some of these keys are stored. This way, it is impossible to recover the backup decryption key without the seed itself. The derived keys are computed from the backup seed as follows:

```
1: procedure DeriveKeys(backupSeed)
2:     /* The backupSeed is 0-padded to 32-byte length */
3:     seed ← backupSeed||0...0
4:     Initialize prng, a PRNGWithHMACWithSHA256 using seed
5:     backupKeyUid ← prng.bytes(32)
6:     encryptionKeyPair ← KEMOverCurve25519.generateKeyPair(prng)
7:     macKey ← HMACWithSHA256.generateKey(prng)
8: end procedure
```

Only the backupKeyUid, macKey and public part of encryptionKeyPair are stored in the application. The backupSeed and private part of encryptionKeyPair are discarded.

The backupKeyUid is not used by legacy backups.

51.2 Backup Contents

A legacy backup is a JSON object containing:

- a part generated by the engine, containing, for each owned identity: its `identity`, private keys and details, the information related to its Keycloak server if it is bound to one, its contacts (with their details, trust level and trust origins), and its groups v1 and groups v2 (including their keys);
- a part generated by the application;
- the version of the JSON format and the time at which the backup was made.

51.3 Backup Encryption

After a backup JSON is generated, it must be encrypted before being exported. This uses the keys derived when the backup key was generated (see Section 51.1) in the following manner:

- use the public part of encryptionKeyPair to perform a PublicKeyEncryption.encrypt (see Section 17) of the JSON string, encoded in UTF-8
- compute a HMACWithSHA256.compute of the ciphertext using `macKey` and append it to the ciphertext

51.4 Backup Decryption

Backup decryption only happens during a restore (see Section 51.5). The user must first enter the backup key from which the keys (including the private part of encryptionKeyPair) can be derived. Decryption is the exact reverse of the encryption, with a check at each step aborting the decryption if it fails:

- compute a HMACWithSHA256.compute of the ciphertext using `macKey` and verify it matches
- then use the private part of encryptionKeyPair to perform a PublicKeyEncryption.decrypt of the ciphertext
- the plaintext is the JSON string (older legacy backups were compressed with raw deflate, and are decompressed)

51.5 Backup Restore

The first part of the restoration of a backup is rather straightforward: simply restore each owned identity in the backup, generate a random `deviceUid` for the device on which they are restored, and restore all the contacts and groups associated with this owned identity.

The tricky part is then to make sure that this restored device is in sync with all its contacts regarding their details, but most importantly that all group members agree on who is a member of a group or not. This is of particular importance when restoring an "outdated" backup. Here are the different steps run after a restore:

- After a contact is created/restored, a device discovery protocol (see Section 35) is run
- Device discovery adds some `deviceUid` for each contact, which triggers channel creation protocols (see Section 32)
- When a channel is created with a contact:
  - Each user sends their published owned identity details to the other (part of the ack messages of the protocol)
    ⇾ during this protocol, if the received details version is lower than what is already in the database, a "downgrade" is authorized.
  - For every group owned by the other user, query the latest group members (see Section 45)
  - For every owned group to which the other user belongs, reinvite them and forcibly push the updated group members (see Section 45)
    ⇾ again, in this protocol, "downgrade" of the group members and details is possible

Part VIII
Secure Calls

Olvid allows users to make secure audio and video calls with their contacts, either with a single contact or with several contacts at once (group calls). Olvid relies on WebRTC for all the "heavy lifting" of media transport, and uses its secure channels to exchange the signaling messages needed to establish the WebRTC connections. Starting a call requires the call permission of the user's account on the server, which is checked when the caller requests TURN credentials (see Section 53).

52 Signaling Messages

Each call is identified by a random UUID chosen by the caller. Signaling messages are JSON objects sent as application messages (see Section 23.2) under the `"rtc"` key. They contain the call identifier (`ci`), the type of the message (`mt`), and the serialized content of the message (`smp`). Table 6 lists the existing message types.

ID	Call signaling message
0	Start call (offer SDP, TURN credentials and servers for the recipient)
1	Answer call (answer SDP)
2	Reject call
3	Hung up
4	Ringing
5	Busy
6	Reconnect call (new offer or answer SDP, to renegotiate the connection)
7	New participant offer (group calls, offer SDP)
8	New participant answer (group calls, answer SDP)
9	Kick (group calls)
10	New ICE candidate
11	Remove ICE candidates
12	Answered or rejected on another device
13	Batch of ICE candidates

Table 6: List of call signaling messages in Olvid.

Session descriptions (SDP) are compressed with raw DEFLATE before being included in these messages. Depending on the capabilities of the contact, the SDP is either sent once all the ICE candidates have been gathered, or sent immediately, the ICE candidates being then sent separately as they are gathered ("trickle ICE").

Signaling messages are sent through confirmed oblivious channels or pre-key channels (see Part IV):

- the start call message is sent to all the devices of the recipient. Once one of these devices answers, all the subsequent messages of the call are only sent to this device. This device also informs the other devices of its owned identity that the call was answered or rejected;
- upon reception, the sender of a message is the one authenticated by the channel through which it was received. A start call message is only accepted from a contact, a message indicating that the call was answered or rejected on another device is only accepted from another device of the same owned identity, and the other messages are only accepted from the participants of the corresponding call.

Once a WebRTC connection is established between two participants, they also exchange in-call messages through a WebRTC data channel: whether a participant is muted, whether they share their camera or screen, and, for group calls, the list of participants and relayed signaling messages (see Section 54).

53 TURN Servers

All the media of Olvid calls is relayed through TURN servers: WebRTC connections are configured to only use relay candidates, so that participants never connect directly to each other and do not learn each other's IP address. The list of TURN servers is part of the configuration published by the server (an alternate list may be chosen by the user).

Authentication to these TURN servers uses the TURN REST API mechanism [20]: the username contains an expiration timestamp, and the password is an HMAC of this username under a key shared by the Olvid server and the TURN servers, allowing the TURN servers to run without any user database (they run the coTURN [10] software). The caller requests credentials from the Olvid server (see Section 77.2) and receives two pairs of credentials, valid for 24 hours: one for themselves, and one for the recipients. The recipients' credentials and the list of TURN servers are sent in the start call message, allowing the recipients to participate in the call without needing to request credentials themselves.

54 Group Calls

In a group call, the caller establishes a WebRTC connection with each participant, as for a call with a single contact. Once connected, the caller sends to each participant, through the data channel, the list of the other participants (their `identity` and display name). Each pair of participants then establishes its own WebRTC connection: the participant with the largest `identity` sends a new participant offer, and the other one answers.

When two participants are contacts, they exchange their signaling messages through their own secure channels. When they are not, they cannot communicate directly: their signaling messages are sent to the caller through the data channel, and the caller relays them to their recipient, indicating the `identity` of their sender.

55 Security

WebRTC handles most of the security by itself: media and data channels are protected by DTLS-SRTP and DTLS, using keys negotiated between the two endpoints of each connection. The only requirement is that the SDP be exchanged over an authentic channel: each SDP contains the fingerprint of the certificate used by its sender during the DTLS handshake. In Olvid, SDP are exchanged through oblivious channels or pre-key channels, so that each participant receives the correct fingerprint of the other participant. TURN servers are thus only untrusted relays: an adversary controlling them (or the network) cannot run a man-in-the-middle attack, and can neither eavesdrop on nor modify the conversations.

In group calls, the SDP of participants who are not contacts of each other are relayed by the caller, through the data channels of the caller's connections. The authenticity of these SDP, and of the `identity` of these participants, thus relies on the caller.

Part IX
History Transfer

The history of the messages of an owned identity is not part of backups (see Part VII), and is not sent during an owned identity transfer (see Section 40). Instead, users can transfer the message history of a profile, with or without attachments, from one of their devices to another of their devices. The user can choose to transfer the whole history of the profile, or only some discussions. Three transfer methods are available: a direct WebRTC connection between the two devices (see Section 56), a Wi-Fi Aware connection between two iOS or iPadOS devices (see Section 57), or a zip file exported on one device and imported on the other (see Section 58).

56 WebRTC History Transfer

56.1 Transfer Establishment

The transfer is initiated on the source device, for a destination device chosen among the other devices of the owned identity. Two kinds of messages are exchanged between these devices, as application messages (see Section 23.2) sent through the oblivious channel or the pre-key channel between the two devices, and only accepted when received from another device of the same owned identity:

- control messages (under the `"htc"` key), containing a random transfer identifier and a type: request, accept, or reject/abort. The source device sends a transfer request, which is presented to the user on the destination device. The transfer only starts if the user accepts it on this device;
- signaling messages (under the `"ht"` key), containing the transfer identifier, and the SDP offer (sent by the source device), the SDP answer (sent by the destination device), or ICE candidates.

Each device uses TURN credentials obtained from its server (see Section 53). Unlike calls, direct connections between the two devices are allowed, so that the transfer can use the local network when both devices are on the same network. As both devices belong to the same user, revealing the IP address of one device to the other is not an issue.

56.2 Data Transfer

Once the WebRTC connection is established, the source device creates a data channel and sends the content to transfer through it: the list of discussions, then their messages, and finally the attachments if the user chose to transfer them. All the messages sent through the data channel are split into chunks and acknowledged by the destination device. Attachments are sent in blocks, along

with their SHA-256 hash: once an attachment is fully received, the destination device recomputes its hash and discards it if it does not match.

56.3 Security

As for calls, the security of the transfer relies on the authenticity of the SDP: they are exchanged through the channels between the two devices, so that each device receives the correct DTLS certificate fingerprint of the other device. The data channel is then protected by DTLS, with keys negotiated between the two devices, and the TURN servers, the network and the Olvid server can neither read nor modify the transferred history.

57 Wi-Fi Aware History Transfer

Between two iOS or iPadOS devices, the history can also be transferred through a direct Wi-Fi Aware connection between the two devices, after they have been paired using the pairing interface of the operating system. The content is then sent over this connection as through the WebRTC data channel (see Section 56.2).

58 Zip File History Transfer

Instead of a WebRTC connection, the history can be exported to a zip file, either for a whole profile, or for a single discussion. When exporting, the user chooses where the file is saved and can optionally set a password. The zip file contains:

- a `discussions_and_messages.json` file, containing the owned identity, the list of exported discussions with their titles, the exported messages (grouped by discussion, sender and thread, in the same format as in a WebRTC transfer), the display names of the senders of these messages, and the SHA-256 and size of each exported attachment;
- the attachments, if the user chose to export them, in a `files/` directory, each one named after the hexadecimal SHA-256 of its content;
- static HTML, CSS and JavaScript files making it possible to browse the exported history in a web browser.

When a password is set, every entry of the zip file is encrypted with the AES encryption of the zip format (with a 256-bit key on Android), using a key derived from the password. Without a password, the zip file is not encrypted.

A zip file can then be imported on a device on which the profile exists: the import is refused if the owned identity contained in the JSON file is not the one of the profile into which it is imported. The import then proceeds as a WebRTC transfer in which the zip file plays the role of the source device: only the messages and attachments that the destination device does not already have are imported, and the SHA-256 of each imported attachment is recomputed and compared with the name of its file in the zip.

A zip file export leaves the protection of Olvid: its confidentiality only relies on the password chosen by the user (if any), and anyone able to modify the file (and knowing its password, if any) can modify the history that will be imported from it.

Part X
Keycloak - Olvid User Directory

59 General Concepts

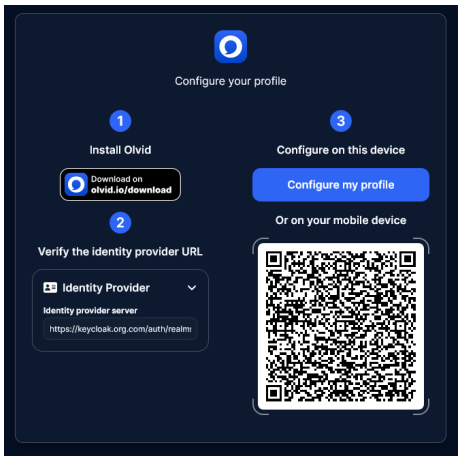
Olvid is designed to allow the creation of secure communication channels between users without having to trust any server. This is great from a security standpoint, but also requires users to interact in order to get in contact. To simplify the deployment of Olvid in large organizations, the Olvid Enterprise subscription allows an organization to deploy an Olvid user directory that can be used to get in contact with anyone in the company in one click. In that case, the user directory is a trusted third party on which users inside the organization agree to rely to get in contact with one another.

This user directory is built upon the Keycloak software (see `https://keycloak.org`) and takes the form of a plugin implementing:

- a REST API, used by the Olvid application
- a management console, used by administrators to manage Olvid users

Keycloak Managed Olvid `identity`. Users may enroll in the Keycloak of their organization by opening/scanning a "Keycloak configuration link" of the form:

`https://configuration.olvid.io/#eyJzZXJ2ZXIiOiJodHR...SyZCI6Im9sdmlkX2NsIn19`



This link contains the URL of the Keycloak server and some OpenID Connect parameters required for the user to authenticate with Keycloak. Once authenticated, the application receives an authentication token that allows the application to interact with the REST API implemented

by the Keycloak Olvid plugin.

The application then uploads the user's `identity` to Keycloak, which in return sends the "signed details" of the user. These signed details are a JWT, signed by Keycloak, containing claims that include:

- the user's first and last name, position and company,
- the user's `identity`,
- the user's Keycloak id,
- a timestamp at which these details were signed.

Once received, these signed details are included in the user's details that are shared with all their contacts, allowing a Keycloak-managed user to detect which of their contacts are managed on the same Keycloak as them. This detection consists in verifying the signed details as described in Section 61. Such users are identified in the app with a small shield next to their name.

In the Olvid clients, the communication with Keycloak (discovery, authentication, and calls to the REST API) is handled by the application, while the engine stores the Keycloak configuration of each owned identity, verifies the signatures of the elements signed by Keycloak, and runs the protocols involving Keycloak (see Section 29 and Section 41).

60 Configuration and Authentication

Configuration. A Keycloak configuration link contains a base64url-encoded JSON object with the URL of the Olvid server (`"server"`), and a `"keycloak"` object containing the URL of the Keycloak server (`"server"`), the OpenID Connect client identifier (`"cid"`) and, optionally, the client secret (`"secret"`). It may also contain a magic link (see below). The same link may be provided to the application through a mobile device management (MDM) configuration. From the Keycloak server URL, the application retrieves:

- the OpenID Connect discovery document (`.well-known/openid-configuration`), and the set of public keys of the server (JWKS) it references;
- the Olvid discovery document (`.well-known/olvid`, see Section 64.12), which indicates whether the server supports identity-based authentication, and the minimum versions of the Olvid clients compatible with the server.

OpenID Connect authentication. The main authentication method is the OpenID Connect authorization code flow with PKCE, in which the user authenticates with their credentials in a web browser. The application then obtains an access token, used as a bearer token for the authenticated entry points of the REST API, and a refresh token. This authentication state is stored by the engine of the device, but is never part of backups, nor transferred to other devices.

Identity-based authentication. When their authentication expires, users already bound to Keycloak can authenticate again using their `identity` instead of their credentials, if the Keycloak server allows it:

- the device sends its Keycloak user identifier and a random 32-byte nonce to the `requestChallenge` entry point, and receives a random 32-byte challenge;
- it signs this challenge with the authentication private key of its owned identity, using the `keycloakChallenge` prefix (see Table 1), and sends the signature and the nonce to the `getSession` entry point;

- Keycloak checks the signature with the `identity` uploaded for this user, and returns an access token and a refresh token.

A challenge can only be used once, and only for one minute. To avoid revealing which user identifiers exist, Keycloak answers the same way whether or not the user exists. If identity-based authentication fails, the user is asked to authenticate with OpenID Connect.

Magic links. A configuration link may also contain a magic link, made of a username and a secret token. When the Keycloak server allows it, sending these to the `getMagicSession` entry point returns an access token for this user, without any other authentication. Magic links are only used when binding an owned identity to Keycloak.

61 Signatures

Signature key. All the elements signed by Keycloak (signed details, revocations, group blobs, group deletions and kicks, transfer proofs) are JWS signed with the signature key of the Keycloak server. When an owned identity is bound to Keycloak, the key of the JWKS that verifies the signed details returned by the `me` entry point is stored, and all the subsequent signatures are verified with this stored key. If the server later signs with another key of its JWKS, the user is asked whether to accept this new key; when they accept, the certification of all their contacts is re-evaluated with the new key. The JWKS itself is only downloaded again when the user authenticates again.

Signed details. The signed details of a user contain their Keycloak user identifier (`id`), their `identity`, their first name, last name, position and company, and the time at which they were signed (`timestamp`). Keycloak signs new details when the user's information changes, and regularly refreshes the signature. When verifying the signed details of a contact, the device checks:

- the signature, with the stored signature key;
- that the `identity` in the signed details is the one of the contact;
- that the signed details are not too old: their timestamp must not be more than 60 days older than the latest timestamp received from the Keycloak server (in the `current-timestamp` field of the `me` output), so that this check does not depend on the clock of the device;
- that the contact was not revoked (see below).

When these checks pass, the contact is marked as certified by the Keycloak server of the owned identity: the details signed by Keycloak are automatically trusted, and a Keycloak trust origin is added to the contact. Otherwise, the contact loses this certification.

Revocations. A revocation is a JWS containing a revoked `identity`, a timestamp, and a revocation type:

- type 0 (identity compromised): Keycloak revoked the `identity` of the user. Unless the user chose to forcefully trust this contact, all the channels with its devices are deleted, the devices are removed, and the contact becomes inactive;
- type 1 (user left the company): the user was deleted on Keycloak. The contact keeps its channels, but loses its certification if its signed details are older than the revocation;
- type 2 (user disabled): the user was disabled on Keycloak. This type is only used on the Keycloak server: it is handled as a user who left the company, and the revocation sent to the devices carries type 1.

Devices keep the revocations of the last 60 days, the same duration as the validity of signed details.

62 Synchronization

Each device of a user bound to Keycloak regularly synchronizes with the Keycloak server, as well as when it receives a push notification for one of the Keycloak push topics of the user (see Section 73.1). During a synchronization, the device:

- calls the `me` entry point, uploads its `identity` with `putKey` if needed, and republishes its details if the signed details changed;
- registers the API key received from Keycloak on the Olvid server (see Section 70.2);
- updates its push topics, its self revocation test `nonce`, and its list of revocations;
- calls the `groups` entry point and processes the Keycloak groups (see Section 48.8).

Before asking the user to authenticate again, the device first runs a self revocation test (see the `revocationTest` entry point): if the user was removed from Keycloak, the owned identity is unbound from Keycloak instead.

63 Transfer Restriction

A Keycloak server may restrict the transfer of the owned identities of its users. For such owned identities:

- the user cannot unbind their owned identity from Keycloak;
- an owned identity transfer requires the user to authenticate on Keycloak from the target device, and the source device only sends the owned identity once it has received a transfer proof signed by Keycloak (see Section 40 and the `transferProof` entry point);
- restoring the owned identity from a backup requires the user to authenticate on Keycloak before the restore.

64 Keycloak REST API

The Keycloak API uses JSON for requests and responses. Each entry point is detailed below. Each entry point may return an error code in the form of a JSON object containing an `int "error"` entry.

64.1 Get owned details (`me`)

This is the main entry point to retrieve information about you on Keycloak.

olvid-rest/me		
in	`"timestamp"`	`int` Keycloak timestamp (`current-timestamp`) received in the last `me` response
out	`"signature"`	user's signed details
	`"server"`	Olvid distribution server URL
	`"revocation-allowed"`	owned `identity` overwrite allowed
	`"transfer-restricted"`	`boolean` owned identity transfer restricted
	`"api-key"`	Olvid API key
	`"nonce"`	self revocation test `nonce` `String`
	`"push-topics"`	list of `String` push topics to register to
	`"signed-revocations"`	list of `String` signed revocations
	`"current-timestamp"`	current Keycloak timestamp
	`"min-build-versions"`	`String` to `int` dictionary of minimum supported builds
error codes		1: unknown error 2: authentication required

The (optional) input timestamp is used to let Keycloak know what portion of the revocation history the client needs. For the first query, this timestamp is not set in the request.

- the distribution server URL is used only when first registering with Keycloak: when creating an `identity`, this indicates on which server to create it; when binding an existing `identity` to Keycloak, this is used to check that the servers match, otherwise Keycloak will not be able to deliver a valid API key/license on a different server.
- the owned `identity` overwrite allowed boolean should always be `False` and is used mostly for testing. If true, this indicates that an authenticated user has the permission to overwrite an `identity` that was already associated with their Keycloak user. Having this setting set to `True` would allow an attacker having stolen a user's Keycloak credentials to "replace" them on Keycloak.
- the transfer restricted boolean indicates whether the owned identity of the user is subject to a transfer restriction (see Section 63).
- the Olvid API key is the license a user gets to unlock paid Olvid features. This API key must be presented to the server using the `registerApiKey` entry point (see Section 70.2).
- the self revocation test `nonce` is used to test whether the user's account still exists on Keycloak. It is used in the `/revocationTest` entry point.
- the list of push topics should be transmitted to the server (in the `/registerPushNotification` entry point, see Section 73.1). One such push topic is associated with each Keycloak group the user is a member of, and one is global to all Keycloak users. It is used by Keycloak to efficiently notify a group of users simultaneously, typically when a group is modified or a user is revoked.
- Keycloak can revoke the `identity` of a user in case it was compromised. The list of signed revocations contains one item for each such revocation. If a `timestamp` was provided in input, only revocations more recent than the timestamp are returned, otherwise the full 2-month revocation history is transmitted. Note that we only send 2 months of revocation history in that case as signed details more than 2 months old are considered invalid.

- the current Keycloak timestamp is sent to be used the next time `/me` is queried
- minimum supported builds is a map associating a minimum build number for `"android"`, `"ios"`, `"desktop"` or `"daemon"`. In case of a breaking change in the Keycloak API, this map can be used to notify the user that their version of Olvid is outdated and is no longer compatible with the Keycloak they are connected to.

64.2 Get groups (groups)

This entry point allows retrieving Keycloak group blobs for any group that was modified since our last request. As for the `me` entry point, the (optional) input timestamp is the timestamp of our last `groups` query and avoids downloading all group blobs at every request.

olvid-rest/groups		
in	`"timestamp"`	`int` timestamp of last `groups` request

out	`"signed_group_blobs"`
	`"signed_group_deletions"`
	`"signed_group_kicks"`
	`"current_timestamp"`
		`String` array of signed group blobs
		`String` array of signed group deletions
		`String` array of signed group kicks
		current Keycloak server timestamp

error codes		1: unknown error 2: authentication required

All signed elements in the output are timestamped JWT, and only those with a large enough timestamp are returned:

- group blobs contain the group details (name, description, and profile picture information), group members and their permissions, as well as ephemeral settings for this group,
- group deletions are generated every time a group is disbanded,
- group "kicks" are generated when a user is removed from the group.

Deletions and kicks are used as proof that the user can safely remove the Keycloak group from their device. They are timestamped and thus cannot be replayed if the group blob timestamp is greater. For group kicks, we do not want to send the updated blob as a proof to the user as there is no reason they should know who is part of the group once they have been kicked.

64.3 Upload `identity` (putKey)

This entry point is used to upload your Olvid `identity` to Keycloak.

olvid-rest/putKey		
in	`"identity"`	`identity` to upload
out	-	*no output if successful*
error codes		1: unknown error 2: authentication required 3: invalid request 4: another identity is uploaded 6: identity was revoked

The entry point does not have any output if the identity is successfully uploaded, but may return an error:

- invalid request: if the server was not able to parse the received `identity`,
- another identity is uploaded: if an identity is already associated with the authenticated Keycloak user and `identity` overwrite is not allowed,
- identity was revoked: if the input `identity` is on Keycloak's revocation list, as a revoked identity can never be reassociated with a user.

64.4 Download signed `identity` (`getKey`)

This entry point is used when adding a contact from Keycloak: it is used to download the signed details of the user and start the Keycloak contact addition protocol from there (see Section 29).

olvid-rest/getKey		
in	`"user-id"`	Keycloak user id to query
out	`"signature"`	`String` signed details of the user's `identity`
error codes		1: unknown error 2: authentication required 3: invalid request

The invalid request error code corresponds either to a malformed query, or a `"user-id"` not found on Keycloak. The output is signed details formatted exactly as in the `"me"` output.

64.5 Contact search (`search`)

This entry point allows searching for users on Keycloak, as you would expect from a user directory!

olvid-rest/search		
in	`"filter"`	`String` array search query
out	`"results"` `"count"`	a list of search results the total number of results
error codes		`1`: unknown error `2`: authentication required `3`: invalid request

The input filter `String` array is treated as an AND query on each of the elements. The output contains users that match all the input filters, either in the first name, last name, position, or company fields. The matching ignores case and accents, and only users that have uploaded their `identity` to Keycloak are returned. The maximum number of results can be configured in Keycloak, but the total number of matches is always sent. Each result takes the exact same form as the claims inside the signed details JWT.

64.6 Revocation test (`revocationTest`)

This entry point is used to test whether your account is still active on Keycloak or not. A call to this entry point should be made before requesting the user to authenticate: if their account was removed from Keycloak, there is no point in asking them to authenticate and the application can simply unbind them from Keycloak. Unlike other entry points, it does not require being authenticated with Keycloak.

olvid-rest/revocationTest		
in	`"nonce"`	`String` `nonce` received after calling `me`
out	`byte[1]`	a single `byte`
error codes		`1`: unknown error `3`: invalid request

This output is not JSON: it is a single `byte` equal to `0x01` if the user was indeed revoked and their `nonce` can no longer be found in the database. Any other response is considered `False`, meaning the user still exists and can be requested to authenticate.

64.7 Contact verification (`verify`)

Like the revocation test, this entry point is not authenticated. It is used in the Keycloak contact addition protocol to verify directly with the server that the contact is indeed still registered on Keycloak. Here, the input is still JSON, but the output is an encoded list like for the server REST API, starting with a status code `byte`.

olvid-rest/verify		
in	"signature"	String signed details of the user to verify
out	byte[1] boolean	return status verification response
return statuses		0x00: OK 0xff: unknown error

The output verification response is True if the contact is still a Keycloak user, False otherwise.

64.8 Group profile picture download (getData)

This entry point is also not authenticated and replicates the getUserData server entry point (see Section 75.2). It is used to download the encrypted profile picture of a group; the decryption key is part of the Keycloak group blob.

olvid-rest/getData		
in	byte[32]	label
out	byte[1] byte[]	return status data
return statuses		0x00: OK 0x09: data not available (bad label) 0xff: unknown error

64.9 Transfer proof (transferProof)

This entry point is used by the target device of an owned identity transfer, when the owned identity is subject to a transfer restriction (see Section 63), to obtain a transfer proof after the user authenticated on Keycloak.

olvid-rest/transferProof		
in	"session_id" "sas"	String transfer session number String full SAS of the transfer
out	"signature"	String signed transfer proof
error codes		1: unknown error 2: authentication required 3: invalid request 8: identity not uploaded yet

The transfer proof is a JWS containing the input session number (`session_id`) and SAS (`sas`), along with the `identity` uploaded by the authenticated user (`identity`) and their Keycloak user identifier (`keycloak_id`). The source device of the transfer checks that all these values match the transfer and the transferred owned identity.

64.10 Magic link authentication (`getMagicSession`)

This entry point is not authenticated. It allows obtaining an access token from a magic link (see Section 60). It is only available if magic links are enabled on the Keycloak server.

`olvid-rest/getMagicSession`		
in	`"username"` `"token"`	`String` Keycloak username `String` magic token

out	`"access_token"`
	`"refresh_token"`
	`"client_id"`
	`"client_secret"`
		access token
		refresh token
		OpenID Connect client identifier
		OpenID Connect client secret

error codes		`1`: unknown error `3`: invalid request

The same invalid request error is returned whether the username does not exist, the token is invalid, or magic links are disabled, so as not to reveal which usernames exist.

64.11 Identity-based authentication (`requestChallenge`, `getSession`)

These two entry points are not authenticated and implement identity-based authentication (see Section 60). Like the `verify` and `getData` entry points, their input and output are encoded lists, as for the Olvid server API.

`olvid-rest/requestChallenge`		
in	`String` `byte[32]`	Keycloak user id `nonce`

out	`byte[1]`
	`byte[32]`
	`byte[32]`
		return status
		challenge
		`nonce` (same as input)

return statuses		`0x00`: OK `0xfe`: parsing error

A challenge is always returned, even if the user does not exist or has not uploaded their `identity`. The challenge is only stored by Keycloak (for one minute, associated with the `nonce`)

if the user exists and has an `identity`.

olvid-rest/getSession		
in	`byte[]` `byte[32]`	challenge response `nonce`
out	`byte[1]` `byte[]`	return status session (JSON)
return statuses		`0x00`: OK `0x0e`: permission denied `0xfe`: parsing error `0xff`: unknown error

The challenge response is computed with AuthenticationOverEC.solve (see Section 14) using the authentication private key of the owned identity, with the `"keycloakChallenge"` prefix and the challenge. Keycloak retrieves (and deletes) the challenge associated with the `nonce`, and checks the response with the `identity` uploaded by the user. The session is a JSON object with the same fields as the output of `getMagicSession`. The same permission denied status is returned when the challenge is unknown, when the user does not exist, when the response is invalid, or when identity-based authentication is disabled for the user.

64.12 Olvid discovery document (`.well-known/olvid`)

This document is retrieved with a `GET` request, without authentication. Older Keycloak servers do not provide it.

.well-known/olvid		

out	`"supportIdentityAuthentication"`
	`"apiVersion"`
	`"minBuildVersions"`
		`boolean` identity-based authentication support
		`int` version of the Keycloak plugin API
		`String` to `int` dictionary of minimum supported builds

The dictionary of minimum supported builds has the same format as in the output of the `me` entry point.

Part XI
Data Storage on Android

This part describes how the Android application protects the data it stores on the device. These protections come in addition to those of the operating system: the private storage of the application is only accessible to the application itself, and is encrypted by the file-based encryption of the device. All the data of the application (databases and files) is stored in this private storage, in the directory excluded from backups, and the automatic backup of Android is disabled for the application, so that no application data leaves the device through the backups of the operating system.

65 Database Encryption

The application uses two databases: the engine database, containing in particular the private keys of the owned identities and the keys of the channels (see Part IV), and the application database, containing the messages, discussions and settings. Both databases are encrypted with SQLCipher [19], each with its own key:

- each key is a 32-byte random value, passed to SQLCipher as a raw key (no key derivation from a passphrase is involved);
- these keys are generated at the first launch of the application and stored in an `EncryptedSharedPreferences` file of the Android Jetpack Security library: preference names are encrypted with AES-256-SIV and values with AES-256-GCM, under a master key (AES-256-GCM) stored in the Android Keystore, which never leaves the Keystore.

The `secure_delete` option of SQLite is enabled on both databases, so that deleted content is overwritten with zeros in the database file.

Migration of unencrypted databases. Older versions of the application used unencrypted databases. At startup, if a database cannot be opened with its key, the application assumes it is an unencrypted database: it exports it to a new database encrypted with the key (using `sqlcipher_export`), and replaces the unencrypted database with the encrypted one. If no key can be obtained from the `EncryptedSharedPreferences`, or if this migration fails, the database is used without encryption.

66 Files

Attachments (sent or received), discussion backgrounds, contact and owned identity photos, as well as attachments being downloaded by the engine, are stored as files in the private storage of

the application. These files are not encrypted by the application: they are only protected by the operating system.

67 Application Lock

The user may protect access to the application with a PIN or a password. The PIN is stored in the preferences of the application as an 8-byte random salt, followed by the 160-bit output of PBKDF2 with HMAC-SHA1 and 1000 iterations, computed from the PIN and this salt. After 3 failed attempts, PIN input is disabled for 60 seconds.

The user may also unlock the application with biometric authentication (using the `BiometricPrompt` API of Android, without any cryptographic object). In this case, the application creates an AES key in the Android Keystore, configured to be invalidated when a new biometric is enrolled on the device. This key is not used to protect any data: it is only used to detect a change in the enrolled biometrics, after which the application requires the PIN again before allowing biometric unlock.

The application lock only controls access to the user interface of the application: it does not encrypt any data, and the databases and files described above remain accessible to the application while it is locked (for instance to receive messages in the background).

Part XII
Olvid Server API

This section describes the different API entry points of the Olvid server. This API is composed of the following components:

- a REST API:
  - each entry point corresponds to a specific `path` on the server
  - all entry points are accessed with a POST request
  - the POST request must indicate a `"Content-type: application/bytes"` header.
  - the POST request should specify a server API version in an `"Olvid-API-Version"` header. The current server API version is 21, but older versions of the Olvid application may still use the same server entry points with a different API version. Not specifying an API version is equivalent to using API version 0. This section only describes the latest versions of the entry points, please refer directly to the source code for older versions. The latest versions introduced a listing start timestamp in `/downloadMessagesAndListAttachments` (version 19), the group update timestamp in the outputs of `/groupBlobGet` and `/groupBlobLock` (version 20), and the status of each attachment in the output of `/downloadMessagesAndListAttachments` (version 21).
  - each entry point expects a POST body containing an encoded list of elements (see Part III), or nothing for the few entry points without an input. In the following section, we will detail the list of items expected by each of these entry points.
  - each entry point then outputs an encoded list of elements. The first element of this list is always a return status in the form of a byte array. The rest of the list contains the various outputs of the entry point, if any.
  - the HTTP return status is 200, whether or not an error occurred. Only the byte return status is of importance. The only exception is a request too large for the server, which gets an HTTP 413 status: clients treat it as the "payload too large" return status `0x18`.
- a WebSocket API:
  - once connected to the WebSocket server, the client may send or receive messages in the form of JSON encoded messages
  - each message must contain an `action` key, defining the purpose of this message, which is similar to the `path` of the REST API
  - the rest of the JSON message may contain additional keys depending on the `action`
- a Transfer WebSocket API:
  - this second WebSocket API is used only when transferring a profile from one device to another
  - this WebSocket server behaves similarly to the main WebSocket API, with JSON messages containing an `action`
  - each connection is given an 8-digit connection ID used to initiate the connection between the two devices

In addition to this, the server may also send some Pre-Signed S3 URLs (valid for 30 minutes) allowing the client to directly upload a file to AWS S3, or a direct S3 URL to download it. This is used when uploading or downloading an attachment, but we will not detail the S3 REST API here, as it is not really a part of the Olvid Server API. Please refer to [1]. Some attachments are stored outside of AWS S3: their upload URLs point to an Olvid upload endpoint (`upload.attachments.olvid.io`), the upload parameters being authenticated by a MAC computed by the server, and their download URLs point to an Olvid download endpoint (`download.attachments.olvid.io`).

68 Server Configuration

Each Olvid server publishes a configuration file, which the applications download (without authentication) with a `GET` request to the `/.well-known/server-config.json` path of the server, and refresh regularly. This JSON file contains two dictionaries:

- under the `"server"` key, the addresses of the servers used by the applications:
  - `"ws_server"`: the URL of the WebSocket server (see Section 78),
  - `"turn_servers"` and `"alt_turn_servers"`: two lists of TURN server URLs used for calls (see Section 53), the second one being used only if the user chose it,
  - `"address_server"` and `"osm_styles"`: the address search server and the map styles used when sharing locations;
- under the `"app"` key, the latest and minimum supported build numbers of the applications for each platform (`"latest_android"`, `"min_android"`, `"latest_ios"`, `"min_ios"`, etc.), used to invite users to update their application.

69 Server Authentication API

The following two entry points are used when a user authenticates with the server. This registers a client session with the server, in the form of a (`identity`, `token`) pair, which is part of the input of entry points requiring authentication.

69.1 Get authentication challenge

The `/requestChallenge` entry point allows a user to request an authentication `challenge` from the server. The `nonce` sent by the user allows multiple authentications from the same `identity` simultaneously. It is sent again (see Section 69.2) with the `response` to retrieve the corresponding `challenge`. The `nonce` in the output is the same as in the input.

/requestChallenge		
in	`identity` `byte[32]`	identity to authenticate `nonce`

out	`byte[1]`
	`byte[32]`
	`byte[32]`
		return status
		`challenge`
		`nonce`

return statuses		`0x00`: OK `0xff`: unknown error

69.2 Authenticate and get client session token

After receiving a `challenge`, the user must compute a `response` given by the AuthenticationOverEC.solve primitive described in Section 15. The received `token` is then stored for later use. The server also accepts a 112-byte `response`, computed with a signature using SHA-512 instead of SHA-256.

/getToken		

in	`identity`
	`byte[80]`
	`byte[32]`
		identity to authenticate
		`response` to the challenge
		`nonce`

out	`byte[1]`
	`byte[32]`
	`byte[32]`
	`int`
	`long`
	`long`
		return status
		`token`
		`nonce`
		API key status
		API key permissions
		API key expiration

return statuses		`0x00`: OK `0x04`: invalid session, unable to retrieve the challenge for this (`identity`, `nonce`) pair `0xff`: unknown error (also returned when `response` validation fails)

API key status. The API key status returned is one of:

`int`	API key status
0	Valid API key
1	Unknown API key
2	Number of licenses for this API key is exhausted
3	API key expired
4	Beta feature tryout API key
5	Free trial API key
6	API key in grace period (awaiting payment)
7	API key on hold (awaiting payment)
8	Expired free trial API key

If the API key expiration is non-zero, it means the API key has an expiration timestamp. In this case, the `long` represents the number of milliseconds since Epoch for this timestamp. For expired keys, this timestamp is in the past.

API key permissions. The permissions `long` is a bit-encoded set of permissions. A 1-bit indicates the permission is granted, a 0-bit indicates it is denied.

Bit	Permission
`1L` $\ll$ `0`	Call permission
`1L` $\ll$ `2`	Multi-device permission

Structure of Free Trial API keys. Free trial API keys are designed to be "anonymous" in the sense that they are never stored in the database of API keys: they are only recorded as the current API key of the `identity` that retrieved them. They are composed of:

- the expiration timestamp of the key (31 days after its creation) in milliseconds since Epoch on 8 bytes
- a HMACWithSHA256 of the 8 timestamp bytes, truncated to 8 bytes

This 16-byte API key is then written as a UUID like other API keys. The HMACWithSHA256Key used for this computation is stored only on the server, which uses it to check free trial API keys in the `/getToken`, `/queryApiKeyStatus`, `/registerApiKey` and `/freeTrial` server entry points.

70 API Keys and Subscriptions API

As described in the `/getToken` entry point, all permissions and access to premium features in Olvid are managed through the API key. To associate an API key to an `identity`, this API key must be registered with `/registerApiKey`. The following entry points make it possible to check the status of a (not yet activated) API key and to start a 1-month free trial for a new user. For in-App purchases, the server generates a new API key in exchange for a valid purchase receipt.

70.1 Query API key status

The `/queryApiKeyStatus` entry point allows a user to query whether the license activation link they clicked is indeed valid, and what permissions are associated with it. The output uses the same values as the `/getToken` entry point. The `identity` is used only for already activated licenses: if the API key is already used by someone else, the current user will not be able to activate it.

`/queryApiKeyStatus`		
in	`identity` `UUID`	identity API key

out	`byte[1]`
	`int`
	`long`
	`long`
		return status
		API key status
		API key permissions
		API key expiration

return statuses		`0x00`: OK `0xff`: unknown error

70.2 Register API key

The `/registerApiKey` entry point allows changing the API key associated with the `identity` calling it. This entry point requires being authenticated. Each `identity` can only be associated with one API key at a time. By default, it is not associated with any API key. The registration may fail if the API key is unknown or already associated with another `identity`. On success, all the sessions of the `identity` are invalidated, so that a new `token`, carrying the permissions of the new API key, must be obtained with `/getToken`.

`/registerApiKey`		

in	`identity`
	`byte[32]`
	`UUID`
		identity
		Authentication `token`
		API key

out	`byte[1]`	return status
return statuses		`0x00`: OK `0x04`: invalid `token` (re-authentication required) `0x16`: API key association failed `0xff`: unknown error

70.3 Free trial API key query and retrieval

This entry point allows users to check whether they are still eligible for a free trial or not, and if yes, allows them to retrieve a free trial API key valid for one month. This entry point requires

being authenticated. The free trial API keys have the structure described in Section 69.2. In retrieval mode, the free trial API key is directly associated with the `identity` (there is no need to call `/registerApiKey`) and, as for `/registerApiKey`, all the sessions of the `identity` are invalidated.

`/freeTrial`		

in	`identity`
	`byte[32]`
	`boolean`
		`identity`
		Authentication `token`
		`true` to retrieve the API key, `false` for an eligibility query

out	`byte[1]` `UUID`	return status Free trial API key (only in retrieval mode)
return statuses		`0x00`: OK `0x04`: invalid `token` (re-authentication required) `0x0f`: free trial already used `0xff`: unknown error

70.4 In-App purchase receipt verification

This entry point verifies a purchase receipt from either a Google Play or an App Store purchase and returns an API key with the appropriate expiration date if the receipt is valid. For iOS purchases, this entry point is also used every time a subscription is renewed as a new receipt is sent to the app. For Google Play purchases and legacy App Store receipts, the server does not verify the receipt itself, but rather queries the Google and Apple servers which take care of the validation and can provide the most up-to-date expiration time for the purchase. For App Store purchases made with StoreKit 2, the receipt is a transaction signed by Apple, which the server verifies locally using Apple's root certificates. When the API key of a subscription expires, the `/getToken` entry point queries the Google or Apple servers when a user authenticates to see if their subscription was renewed. As with `/registerApiKey`, the returned API key is directly associated with the `identity` and all the sessions of the `identity` are invalidated.

/verifyReceipt		

in	`identity`
	`byte[32]`
	`byte[1]`
	`String`
		`identity`
		Authentication `token`
		Store identifier (see below)
		Store purchase receipt

out	`byte[1]` `UUID`	return status a new API key (or the same as before for iOS renewals)
return statuses		`0x00`: OK `0x04`: invalid `token` (re-authentication required) `0x10`: receipt is expired (legacy App Store receipts only) `0xff`: unknown error

Store identifiers. This entry point uses different receipt verification queries depending on the store the receipt was generated on.

Byte	Store
`0x00`	legacy iOS store (using StoreKit 1)
`0x01`	Android store
`0x02`	iOS store (using StoreKit 2)

71 Message Upload API

Uploading a message to the Olvid server happens in two steps:

- the user uploads the message payload
- the user uploads the attachments, chunk by chunk

At any time, the sender may also cancel an attachment upload, allowing recipients to know the attachment may never be fully uploaded on the server. Messages without attachments can also be uploaded in batches.

71.1 Upload message and get UID

The `/uploadMessageAndGetUids` entry point allows uploading a single message to the server and can be used for any message, with or without attachments. A "payload too large" return status means the message payload (or extended payload) is too large for the server database: this is not a retriable error.

/uploadMessageAndGetUids		
in	`[encoded_vals]`	list of encoded headers (see details 1 below)
	`byte[]`	encrypted message payload
	(`byte[]`)	(optional) message extended payload (see details 2 below)
	`boolean`	message contains an application payload (see details 3 below)
	`boolean`	message is a start call VoIP message (see details 3 below)
	`[encoded_vals]`	list of encoded attachment lengths (see details 4 below)
	`[encoded_vals]`	list of encoded attachment chunk lengths (see details 4 below)
out	`byte[1]`	return status
	`byte[32]`	message unique identifier on the server
	`byte[32]`	nonce (see details 5 below)
	`long`	server timestamp
	`[encoded_vals]`	list of private signed URLs (see details 6 below)
return statuses		`0x00`: OK `0x18`: payload too large `0xff`: unknown error

Detailed description of the different inputs and outputs:

1. **Encoded headers**: this list contains $3n$ encoded elements, 3 for each recipient device. For the i-th device this list contains:
   - at position $3i$, the encoded `deviceUid` of the recipient
   - at position $3i + 1$, the encoded byte array containing the header payload
   - at position $3i + 2$, the encoded `identity` of the recipient
2. **Extended payload**: for messages containing image attachments, this extended payload contains the JPEG data of a "mosaic" of small 40×40 pixel thumbnails of (at most 25 of) these images, along with the numbers of the corresponding attachments. It is encrypted as described in Section 24.2 and is typically a few kB long. This input is optional and is only present for messages with such an extended payload.
3. The two booleans `isApplicationMessage` and `isVoipMessage` are used when notifying the message recipient:
   - application messages showing a "heads up" notification must be sent encrypted in the notification to be received in the notification extension of the application.
   - VoIP call initiation messages use the special `voip` push type which allows instant delivery to CallKit enabled applications.
   - on Android, both these message types use a high priority notification, instead of a normal priority notification for other messages.
4. **Encoded attachment lengths**: the lists of encoded attachment lengths and attachment chunk lengths must both contain one length per message attachment. The first list contains encoded 64-bit integers corresponding to the total byte length of each encrypted attachment. The second list contains encoded 64-bit integers corresponding to the length of the encrypted

chunks into which the corresponding attachment is split.

5. **Nonce**: a random 32-byte nonce is received along with the unique message identifier. This nonce will be required to authenticate the uploader when refreshing attachment URLs or cancelling an attachment upload. Indeed, the message identifier is sent to the recipient of the message and, for group messages, one group member should not be able to cancel the upload of an attachment, effectively preventing other group members from receiving the attachment.
6. **Private signed urls**: after uploading the message, the user must also upload all the attachments. For each attachment chunk, the server computes a private signed URL allowing the client to upload it directly to AWS S3, without going through one of the server entry points. The list of private signed URLs actually contains:
   - for each attachment, an encoded list of private signed URLs
   - these encoded lists contain as many private signed URLs as this attachment has chunks. Each signed URL is a string, encoded as described in Section 22.3.

   Computing signed URLs is quite time consuming: for messages with multiple attachments (or attachments with multiple chunks), once the server has spent more than 5 seconds processing the request, it returns "dummy" signed URLs for the remaining chunks instead. Such URLs are already expired and need to be refreshed using the `/uploadAttachment` entry point below.

71.2 Batch message upload

To optimize server bandwidth and decrease the number of API requests, it is possible to upload attachment-less messages in batches. Contrary to other entry points that expect a fixed length input list, this entry point takes an n-item list as input when uploading a batch of n messages. Each of these n items is itself an encoded list of 4 elements.

The response is an encoded list of $(n + 1)$ items, starting with the status byte, followed by n encoded lists of 3 elements. When receiving a "payload too large" response, the client should retry uploading each message one at a time using the `/uploadMessageAndGetUids` entry point.

/batchUploadMessages		

in	`[`
	`[encoded_vals]`
	`byte[]`
	`boolean`
	`boolean`
	`]`
		a repetition of n of the following list-encoded 4 elements
		list of encoded headers (see details 1 above)
		encrypted message payload
		message contains an application payload (see details 3 above)
		message is a start call VoIP message (see details 3 above)

out	`byte[1]`
	`[`
	`byte[32]`
	`byte[32]`
	`long`
	`]`
		return status
		a repetition of n of the following list-encoded 3 elements
		message unique identifier on the server
		nonce (see details 5 above)
		server timestamp

return statuses		`0x00`: OK `0x18`: payload too large `0xff`: unknown error

71.3 Refresh private upload signed URLs

The private upload signed URLs obtained from the `/uploadMessageAndGetUids` entry point have an expiration date. Past this date, the URL becomes invalid and must be refreshed to upload the corresponding chunk to AWS S3. For historical reasons and backward compatibility the path to this entry point is `/uploadAttachment`.

/uploadAttachment		

in	`byte[32]`
	`int`
	`byte[32]`
		message unique identifier on the server
		attachment number
		the nonce received when uploading the message

out	`byte[1]` `[encoded_vals]`	return status list of private upload signed URLs
return statuses		`0x00`: OK `0x09`: attachment was already deleted from the server `0x0c`: invalid nonce `0xff`: unknown error

The message unique identifier on the server and the nonce are taken from the output of the `/uploadMessageAndGetUids` entry point. The attachment number corresponds to the position of this attachment in the attachment lengths lists sent in the upload message entry point. As in the upload message entry point, the list of private signed URLs contains as many private signed URLs

as the attachment with the specified number has chunks. Each signed URL is a string, encoded as described in Section 22.3.

71.4 Get attachment upload progress

The `/getAttachmentUploadProgress` entry point allows asking the server which attachment chunks have been successfully uploaded. The output is a list of indexes of chunks already on the server. This is used on iOS/macOS (as the chunks are uploaded in parallel) to restart an interrupted transfer. On other platforms, chunks are uploaded in order and this entry point is not used.

`/getAttachmentUploadProgress`		
in	`byte[32]` `int`	message unique identifier on the server attachment number
out	`byte[1]` `[encoded_vals]`	return status list of attachment chunk indexes as encoded `int`
return statuses		`0x00`: OK `0x09`: attachment was already deleted from the server `0x0d`: attachment upload was cancelled `0xff`: unknown error

71.5 Cancel attachment upload

This tags an attachment as cancelled on the server. The consequence is that recipients no longer receive download URLs for this attachment, but a status indicating that its upload was cancelled: for incomplete attachments, the recipients know that they will never be completed and can cancel their download.

`/cancelAttachmentUpload`		

in	`byte[32]`
	`int`
	`byte[32]`
		message unique identifier on the server
		attachment number
		the nonce received when uploading the message

out	`byte[1]`	return status
return statuses		`0x00`: OK `0x0c`: invalid nonce `0xff`: unknown error

72 Message Download API

72.1 Download messages and list attachments

This entry point retrieves the messages for a specific device of an `identity` that are available on the server and were not marked as listed. Some of these messages may already have been listed and not deleted yet. In addition, for each message, it retrieves a list of URLs allowing to download its attachment chunks directly from AWS S3. Only messages whose server timestamp is strictly greater than the listing start timestamp passed as input are returned (the client passes 0 to list all messages). This entry point is "time limited" on the server side meaning that if listing all pending messages takes too long it will return a truncated output. In that case, the return status is different, and the client immediately calls this entry point again, with a listing start timestamp equal to the timestamp of the last message it received, to list the following messages.

Note that this method retrieves messages for a specific (`identity`, `deviceUid`) pair, so it returns messages that have either been sent to this specific `deviceUid` (through multicast or unicast), or are broadcast messages for this `identity`.

The method also returns a server timestamp to allow devices to measure precisely the age of downloaded messages, even if the device clock is skewed.

`/downloadMessagesAndListAttachments`		

in	`identity`
	`byte[32]`
	`byte[32]`
	`long`
		`identity`
		Authentication `token`
		`deviceUid`
		listing start timestamp (milliseconds since Epoch)

out	`byte[1]`
	`long`
	`encoded_val`
	`...`
		return status
		current server timestamp (milliseconds since Epoch)
		one encoded value for each message (see description below)

return statuses		`0x00`: OK `0x17`: OK, but message list was truncated `0x04`: invalid `token` (re-authentication required) `0x0b`: `deviceUid` is not registered, or is inactive or expired (device registration required) `0xff`: unknown error

Each message `encoded_val` is an encoded list containing:

- the encoded `byte[32]` of the message unique identifier on the server
- the encoded `long` of the message timestamp
- the encoded `byte[]` of the header payload
- the encoded `byte[]` of the encrypted message payload
- the encoded `boolean` indicating whether this message has an extended payload available
- for each attachment, an encoded list containing:

- the encoded `int` of the attachment number
- the encoded `byte[1]` of the attachment status: `0x00` if the attachment is available, or `0x0d` if its upload was cancelled by the sender (see Section 71.5)
- the encoded `long` of the encrypted attachment length
- the encoded `long` of the encrypted attachment chunk length
- only if the attachment status is `0x00`, the encoded list of encoded `String` of the URLs for attachment chunks download. This list contains as many elements as the number of attachment chunks.

Note that the messages are always sorted before being returned, from oldest to newest, based on the server timestamp at upload time (messages sent to this `deviceUid` and broadcast messages being merged in this order). If the listing is truncated, only the oldest messages are returned. A message with attachments is only listed once all its attachments have been uploaded or cancelled by the sender.

72.2 Download message extended payload

The entry point `/downloadMessageExtendedContent` allows downloading the extended payload of messages that have one (indicated by a boolean when listing messages). Please refer to Section 24.2 for information on how this extended payload is encrypted.

`/downloadMessageExtendedContent`		

in	`identity`
	`byte[32]`
	`byte[32]`
		`identity`
		Authentication `token`
		message unique identifier on the server

out	`byte[1]` `byte[]`	return status the encrypted extended content
return statuses		`0x00`: OK `0x04`: invalid `token` (re-authentication required) `0x11`: extended content not found `0xff`: unknown error

72.3 Refresh private download signed URLs

This entry point only exists for legacy reasons. Attachment download URLs are no longer signed and do not expire.

The private download signed URLs obtained from old versions of the download message and list attachments entry point had an expiration date. Past this date, the URL became invalid and had to be refreshed to download the corresponding chunk from AWS S3. For historical reasons and backward compatibility the path to this entry point is `/downloadAttachmentChunk`.

/downloadAttachmentChunk		
in	`byte[32]` `int`	message unique identifier on the server attachment number
out	`byte[1]` `[encoded_vals]`	return status list of download URLs
return statuses		`0x00`: OK `0x09`: attachment was already deleted from the server `0x0d`: attachment upload was cancelled `0xff`: unknown error

If the attachment was marked as cancelled (see Section 71.5), this method returns the `0x0d` status. Otherwise, it returns a download URL for each attachment chunk, or the `0x09` status if some chunk is no longer available on the server.

72.4 Delete message and attachments

Once a message and all its attachments have been downloaded by a recipient (`identity`, `deviceUid`) pair, this recipient can notify the server to delete the message and its attachments. The server will indeed delete the message once all recipients have requested to delete it. In practice, the server simply deletes the message header corresponding to this (`identity`, `deviceUid`) pair, and when no message header remains for a message, the message is deleted.

If a message has some attachments that are not yet downloaded, the client may also instruct the server to "mark the message as listed" so that `/downloadMessagesAndListAttachments` no longer lists it. Also, this entry point now processes a batch of requests to delete messages or mark them as listed.

/deleteMessageAndAttachments		

in	`identity`
	`byte[32]`
	`byte[32]`
	`[encoded_vals]`
		`identity`
		Authentication `token`
		`deviceUid`
		list of encoded requests (see details below)

out	`byte[1]`	return status
return statuses		`0x00`: OK `0x04`: invalid `token` (re-authentication required) `0xff`: unknown error

The **encoded requests** input list contains $2n$ encoded elements, 2 for each message to delete or mark as listed. For the i-th message this list contains:

- at position $2i$, the encoded `byte[32]` message unique identifier

- at position $2i+1$, the encoded `boolean` indicating if this message should be marked as listed (`True`) or deleted (`False`)

If the server cannot find a header with the corresponding unique identifier for this `deviceUid`, it will search for a broadcast message instead (only for delete operations). If the message was already deleted on the server, this entry point returns OK.

73 Device Management API

The device management API groups together all server entry points used to register a device on the server, discover the devices of a contact, or manage your own devices. The legacy `/unregisterPushNotification` entry point no longer exists and has been replaced by the `/deviceManagement` entry point.

73.1 Register push notification

In order to be instantly notified of new incoming messages, each user device must be registered on the server. Depending on the kind of notifications the device can receive, the server will notify it through a different service. Note that devices that cannot receive push notifications (typically, an Android phone without the Google services installed) still need to register with the server as this registration is also used for device discovery (see Section 73.2).

`/registerPushNotification`		

in	`identity`
	`byte[32]`
	`byte[32]`
	`byte[1]`
	`extra_info`
	`boolean`
	`[String]`
	`byte[]`
	`(byte[32])`
		`identity`
		Authentication `token`
		`deviceUid`
		push notification type identifier (see below)
		push notification extra information (see below)
		reactivate current device (see below)
		keycloak push topics (see below)
		encrypted device name (see below)
		(optional) `deviceUid` to replace

out	`byte[1]`	return status
return statuses		`0x00`: OK `0x04`: invalid `token` (re-authentication required) `0x0a`: device is inactive and reactivation was not requested `0x0b`: `deviceUid` to replace is not registered (or is inactive) `0xff`: unknown error

Push notification types. The following push notification type identifiers currently exist on the server. For each of them, the extra information `extra_info` has a specific format.

- `0x01` - **Android**: this notification type sends push notifications through Google's Firebase Cloud Messaging service. The `extra_info` is a list of encoded elements containing:
  - a `String` containing the device token provided by Firebase, required to send the push notification to the correct device.
  - a `byte[32]` containing a random masking unique identifier, allowing the device to know which `identity` received a message.

  The masking unique identifier makes it possible to tell the device which of its identities (in case several identities are installed on the same device) has received a message. The `identity` itself cannot be sent in the push notification, otherwise it would allow the push notification server operated by Google to associate an Olvid `identity` with the real identity of the device owner (already known by the Firebase Cloud Messaging service).
- `0x05` - **iOS with extension**: this notification type sends push notifications through the Apple Push Notification service production server and includes the encrypted payload of application messages in the notification. This allows the notification extension of the iOS application to notify the user with the decrypted message content while the application is in the background. The `extra_info` is a list of encoded elements containing:
  - a `byte[]` containing the device token provided by Apple, required to send the push notification to the correct device.
  - a `byte[32]` containing a random masking unique identifier, allowing the device to know which `identity` received a message.
  - an optional `byte[]` containing the device VoIP token provided by Apple. This is required to receive VoIP notifications for incoming Olvid calls, but is not available for users who deactivated CallKit in the App.

  The role of the masking unique identifier is the same as in Android notifications.
- `0x04` - **iOS sandbox with extension**: this notification type is exactly the same as the iOS with extension, but uses the development server provided by Apple for tests. It is not used in the production version of the application.
- `0x06` - **macOS**: this notification type is exactly the same as the iOS with extension.
- `0x10` - **Android WebSocket only**: this notification type indicates an Android device with Firebase notifications disabled. Notifications are sent only through the WebSocket connection. The `extra_info` is empty for this type.
- `0x11` - **Windows**: this notification type indicates a Windows device. Notifications are sent only through the WebSocket connection. The `extra_info` is empty for this type.
- `0x12` - **Linux**: this notification type indicates a Linux device. Notifications are sent only through the WebSocket connection. The `extra_info` is empty for this type.
- `0x13` - **Daemon**: this notification type indicates the device is a daemon as used by Olvid Bots. Notifications are sent only through the WebSocket connection. The `extra_info` is empty for this type.
- `0xff` - **No notifications**: this notification type is for devices which cannot receive push notifications in the background. It simply allows the device to register on the server for device discovery. The `extra_info` does not need to contain anything here.

Note that for WebSocket only notifications, the server takes care of "storing" pending notifications. If a notification must be sent to a user without an active WebSocket connection, some information is kept on the server to notify them the next time a WebSocket connection is established.

Reactivate current device. Once registered with the server, a device can be deactivated for two reasons:

- without a multi-device license, when the user registers a new device, the server sets a 30-day expiration on their other devices, and deactivates them once this expiration has passed,
- if the user deliberately removes a device from their device list.

Once a device is deactivated, the `/registerPushNotification` entry point must be called with "reactivate current device" set to `True`to reactivate it. In that case, the user may specify which device to replace (if they do not own a multi-device license).

Keycloak push topics. For an `identity` that is managed by a Keycloak server (part of the Olvid Enterprise offer), some "push topics" may be passed to the server. These push topics are registered on the server by the client's Keycloak server and allow Keycloak to efficiently notify a large number of users, typically when an `identity` is revoked on Keycloak. When passing push topics to the `/registerPushNotification` entry point, the server registers the device for these push topics. For devices receiving notifications only through the WebSocket, the server also removes the registrations to topics that are no longer in the list.

Encrypted device name. To help users identify their devices, an encrypted device name is passed when registering a device. The device name is encoded in a list (containing the encoded UTF-8 `String`), padded with zeros to a multiple of 128 bytes, and encrypted with PublicKeyEncryption (see Section 17) using the `identity` public key: it is thus only accessible to the user themselves. This encrypted device name is not updated when calling this entry point again if the device is already registered. Modifying it is only possible through the `/deviceManagement` entry point.

73.2 Device discovery

The device discovery entry point allows anyone to query the server for the list of `deviceUid` of a given `identity`. When adding a new contact in Olvid, this is used to list all the devices with which a secure channel must be created. It is also used to retrieve a pre-key (see Section 25) for each device and be informed of inactive contacts.

`/deviceDiscovery`		
in	`identity`	`identity`
out	`byte[1]` `encoded_dict`	return status device discovery output (see below)
return statuses		`0x00`: OK `0xff`: unknown error

The output `encoded_dict` is an encoded dictionary containing:

- for key `dev`: an encoded list of encoded dictionaries (one such per active device) containing:
  - for key `uid`: the `deviceUid` of the device
  - for key `prk` (only if the device uploaded a pre-key): the encoded signed pre-key of this device, as uploaded with `/uploadPreKey` (see Section 25)
- for key `st`: the current server timestamp (in milliseconds since Epoch)

- for key `ro`: a "recently online" `boolean` indicating whether the `identity` was online (listed messages on the server, connected to the WebSocket, or called `/registerPushNotification`) less than 40 days ago (`True`) or not (`False`)

73.3 Owned device discovery

In a multi-device context, it is important for an `identity` to know exactly which devices are registered. The `/deviceDiscovery` entry point gives only limited information and is intended for contact device discovery. A dedicated entry point is available to discover one's owned devices. For technical reasons, this entry point cannot use the `token` authentication mechanism used by other entry points, especially as the application may have to query this entry point during a backup restoration, before even having an `identity` in the engine. Instead, the output of this entry point is encrypted using the public key of the queried `identity`, so that only the owner of the `identity` has access to the response.

`/ownedDeviceDiscovery`		
in	`identity`	`identity`
out	`byte[1]` `ciphertext`	return status the encrypted output (see below)
return statuses		`0x00`: OK `0xff`: unknown error

The output `ciphertext` is an encrypted encoded dictionary containing:

- for key `dev`: an encoded list of encoded dictionaries (one such per active device) containing:
  - for key `uid`: the `deviceUid` of the device
  - for key `reg`: the timestamp (milliseconds since Epoch) of the last time the device was online (listed messages on the server, connected to the WebSocket, or called `/registerPushNotification`)
  - for key `name`: the encrypted device name
  - for key `exp`: the expiration timestamp (milliseconds since Epoch) of this device if this device is set to expire.
  - for key `prk` (only if the device uploaded a pre-key): the encoded signed pre-key of this device
- for key `st`: the current server timestamp (in milliseconds since Epoch)
- for key `multi`: a `boolean` indicating whether the user has a multi-device license active

73.4 Device management

The `/deviceManagement` entry point allows managing devices associated with an `identity`: change a device name, remove a device, determine which device should not expire (without a multi-device license). The number of elements in the input list depends on the type of request being made.

/deviceManagement		

in	`identity`
	`byte[32]`
	`byte[1]`
	`...`
		`identity`
		Authentication `token`
		request type
		additional request parameters

out	`byte[1]`	return status
return statuses		`0x00`: OK `0x04`: invalid `token` (re-authentication required) `0x0b`: `deviceUid` is not registered (or is inactive, for request type `0x02`) `0xff`: unknown error

The following request types are currently implemented:

- `0x00` **- rename a device:** additional request parameters are:
  - `byte[32]`: the `deviceUid` to rename
  - `ciphertext`: the encrypted device name to set for the device

  This simply overwrites the encrypted device name of the device, immediately changing what is output by the `/ownedDeviceDiscovery` entry point.
- `0x01` **- deactivate a device:** additional request parameters are:
  - `byte[32]`: the `deviceUid` to deactivate

  This request immediately deactivates a device which will no longer be returned by the `/deviceDiscovery` and `/ownedDeviceDiscovery` entry points.
- `0x02` **- set the non-expiring device:** additional request parameters are:
  - `byte[32]`: the `deviceUid` to set as non-expiring

  If the selected device has an expiration and there is a non-expiring device, the expiration of the selected device will be transferred to the current non-expiring device.

74 Group V2 Management API

Groups V2 rely on an encrypted blob uploaded on the server to guarantee that all group members have the same view of who is a member and what permissions each member has. The following entry points are here to allow uploading such a blob, downloading it and updating it. A lock system is implemented to prevent concurrent modifications of the blob.

74.1 Group creation

When a group is created, the first version of the blob is uploaded. The `/groupBlobCreate` entry point can only be called for new groups with a group identifier that is not yet known to the server. At the same time, the administration public key that will be used to update the blob later on is uploaded.

/groupBlobCreate		

in	`identity`
	`byte[32]`
	`byte[32]`
	`public_key`
	`ciphertext`
		`identity`
		Authentication `token`
		group unique identifier
		encoded administration public key
		encrypted group blob

out	`byte[1]`	return status
return statuses		`0x00`: OK `0x04`: invalid `token` (re-authentication required) `0x12`: group identifier already exists `0xff`: unknown error

74.2 Group download

Downloading a group blob does not require any authentication. Knowing its identifier is enough to retrieve the encrypted blob. The actual access control is handled by group administrators who only share the blob decryption keys with actual group members. If the group is currently locked because it is being updated, the download will fail and the application should retry downloading later.

/groupBlobGet		
in	`byte[32]`	group unique identifier

out	`byte[1]`
	`byte[]`
	`[encoded_vals]`
	`public_key`
	`long`
		return status
		encrypted group blob
		list of encoded group log items
		encoded administration public key
		group update timestamp

return statuses		`0x00`: OK `0x09`: group identifier not found (or already deleted) `0x13`: group is currently locked for an update `0xff`: unknown error

The `encoded_vals` correspond to a list of "log items" which are signed proofs that a member left the group. These log items are only consolidated inside the blob during an update. Each member consolidates them inside the blob after the download. The group update timestamp is the server timestamp (in milliseconds since Epoch) of the last creation or update of the blob, or 0 if it is unknown.

74.3 Group lock

Before updating a group blob, an administrator must first lock the group, effectively preventing concurrent modifications of the blob. An administrator must prove to the server that it knows the private key associated with the encoded administration public key to lock a group. When locking a group, the user provides a random `nonce` to the server that allows them to be sure they have the lock (and not another admin) when updating the blob.

`/groupBlobLock`		

in	`byte[32]`
	`byte[32]`
	`byte[]`
		group unique identifier
		lock `nonce`
		signature

out	`byte[1]`
	`byte[]`
	`[encoded_vals]`
	`public_key`
	`long`
		return status
		encrypted group blob
		list of encoded group log items
		encoded administration public key
		group update timestamp

return statuses		`0x00`: OK `0x09`: group identifier not found (or already deleted) `0x13`: group is already locked (with a different `nonce`) `0x14`: invalid signature `0xff`: unknown error

The output of this entry point is the same as that of `/groupBlobGet`. This allows making sure you have the latest version of the group blob before updating it. On success, the group will be locked for **30 seconds**: if the `/groupBlobUpdate` entry point is not called in time, the lock is released. Locking a group that is already locked with the same `nonce` is accepted and renews the lock.

The signature should be computed using AuthenticationOverEC.solve using the group administration private key on an input made of:

- `prefix`: the group lock signature prefix `"lockNonce"` encoded to `byte[]` in UTF-8,
- `challenge`: the `nonce`.

The server verifies this signature using the AuthenticationOverEC.check procedure from Section 15 with the group administration public key associated with the blob.

74.4 Group update

Once a group is locked, the administrator who locked it may update the blob and the administration public key. This administrator must provide the lock `nonce` to make sure it was them who locked the group. When the group blob is updated, the server also purges all log items for this group: it is the role of the updater to consolidate them in the blob.

/groupBlobUpdate		

in	`byte[32]`
	`byte[32]`
	`ciphertext`
	`public_key`
	`byte[]`
		group unique identifier
		lock `nonce`
		encrypted group blob
		encoded administration public key
		signature

out	`byte[1]`	return status
return statuses		`0x00`: OK `0x09`: group identifier not found (or already deleted) `0x13`: group is locked with a different `nonce` `0x14`: invalid signature `0x15`: group is not locked (or the lock expired) `0xff`: unknown error

The signature should be computed using AuthenticationOverEC.solve using the group administration private key on an input made of:

- `prefix`: the group update signature prefix `"updateGroup"` encoded to `byte[]` in UTF-8,
- `challenge`: the concatenation of the `nonce`, the blob, and the encoded new administration public key.

The server verifies this signature using the AuthenticationOverEC.check procedure from Section 15 with the group administration public key associated with the blob.

74.5 Group log put

When leaving a group, a member must upload a "log entry" to the server. This log entry is a signature, with the `groupLeave` prefix, of the group identifier and of the group invitation nonce of the user (see Section 48.5). This entry point is not authenticated and the server only checks the length of the log entry (80 bytes, or 112 bytes for a signature using SHA-512). If the group is locked, the server does not accept the log entry and the user should retry later.

/groupLogPut		
in	`byte[32]` `byte[80]`	group unique identifier log item
out	`byte[1]`	return status
return statuses		`0x00`: OK `0x09`: group identifier not found (or already deleted) `0x13`: group is locked `0xff`: unknown error

74.6 Group deletion

Contrary to group updates, group deletion does not require locking the group: any administrator with access to the group administration private key may delete the group and remove any trace of it from the server. If the group is already deleted, the server responds with OK.

`/groupBlobDelete`		
in	`byte[32]` `byte[]`	group unique identifier signature
out	`byte[1]`	return status
return statuses		`0x00`: OK `0x13`: group is locked `0x14`: invalid signature `0xff`: unknown error

The signature should be computed using AuthenticationOverEC.solve using the group administration private key on an input made of:

- `prefix`: the group deletion signature prefix `"deleteGroup"` encoded to `byte[]` in UTF-8,
- `challenge`: none, a zero length `byte[]`

The server verifies this signature using the AuthenticationOverEC.check procedure from Section 15 with the group administration public key associated with the blob. Note that the signature does not depend on the blob or the group identifier, but only on the signature key.

75 User Data Management API

75.1 Put user data

This entry point allows users to upload some data associated with a label to the server. Anyone knowing the `identity` of the user and the label will then be able to download this data (using the get user data entry point hereafter). This is used, for example, to upload an encrypted user profile picture and push this picture to your contacts along with your name. The label is a random 32-byte identifier chosen by the user, and the data is encrypted by the user with authenticated encryption, under a random key sent to the contacts along with the label. Uploading data with a label already in use replaces the previous data.

/putUserData		

in	`identity`
	`byte[32]`
	`byte[]`
	`byte[]`
		`identity`
		authentication `token`
		label
		data

out	`byte[1]`	return status
return statuses		`0x00`: OK `0x04`: invalid `token` (re-authentication required) `0xff`: unknown error

75.2 Get user data

This entry point allows any user to download user data associated with an `identity` and a label from the server. Note that downloading user data can be done anonymously, without requiring authentication.

/getUserData		
in	`identity` `byte[]`	`identity` label
out	`byte[1]` `byte[]`	return status data
return statuses		`0x00`: OK `0x09`: data not available (not uploaded yet, or deleted) `0xff`: unknown error

75.3 Refresh user data

User data uploaded to the server naturally expires after 90 days. This entry point is used to extend the expiration date of user data that is still useful. It is called every week by the application to refresh its user data.

/refreshUserData		

in	`identity`
	`byte[32]`
	`byte[]`
		`identity`
		authentication `token`
		label

out	`byte[1]`	return status
return statuses		`0x00`: OK `0x04`: invalid `token` (re-authentication required) `0x09`: the user data was deleted (re-upload required) `0xff`: unknown error

75.4 Delete user data

This entry point allows a user to delete obsolete user data they uploaded on the server. Typically, after changing their profile picture, the old picture can be deleted.

/deleteUserData		

in	`identity`
	`byte[32]`
	`byte[]`
		`identity`
		authentication `token`
		label

out	`byte[1]`	return status
return statuses		`0x00`: OK `0x04`: invalid `token` (re-authentication required) `0xff`: unknown error

76 Backup API

The following entry points are used to store backups on the server (see Section 50.4). They do not use the `token` authentication: backups are identified by their 32-byte backupKeyUid, and uploads and deletions are authenticated by a signature with the private key of the authentication key pair derived from the backup seed, whose public key is registered on the server along with the backupKeyUid. All these entry points return the `0xfe` status (parsing error) when their input is malformed, for instance when a UID is not 32 bytes long.

76.1 Backup creation

The `/backupCreate` entry point registers a new backupKeyUid on the server, along with the public key used to verify the signatures of the uploads and deletions of backups under this backupKeyUid.

/backupCreate		
in	`byte[32]` `public_key`	backupKeyUid encoded authentication public key
out	`byte[1]`	return status
return statuses		`0x00`: OK `0x19`: backupKeyUid already used `0xfe`: parsing error `0xff`: unknown error

76.2 Backup upload

The `/backupUpload` entry point uploads a new version of a backup in a thread. The version must be strictly greater than the version currently stored in this thread (if any): the new version replaces the previous one, which is deleted from the server.

/backupUpload		

in	`byte[32]`
	`byte[32]`
	`long`
	`ciphertext`
	`byte[]`
		backupKeyUid
		thread identifier
		version
		encrypted backup
		signature

out	`byte[1]`	return status
return statuses		`0x00`: OK `0x14`: invalid signature `0x1a`: version too small `0x1b`: unknown backupKeyUid `0xfe`: parsing error `0xff`: unknown error

The signature is computed with AuthenticationOverEC.solve, using the private key of the authentication key pair, on an input made of:

- `prefix`: the backup upload signature prefix `"backupUpload"` encoded to `byte[]` in UTF-8,
- `challenge`: the encoded list of the four previous elements of the input.

The server verifies this signature using the AuthenticationOverEC.check procedure from Section 15 with the public key registered for the backupKeyUid.

76.3 Backup list

The `/backupList` entry point lists the backups stored under a backupKeyUid. It does not require any authentication: knowing the backupKeyUid is enough.

`/backupList`		
in	`byte[32]`	backupKeyUid
out	`byte[1]` `[encoded_vals]`	return status list of backups (see below)
return statuses		`0x00`: OK `0x1b`: unknown backupKeyUid `0xfe`: parsing error `0xff`: unknown error

Each element of the list of backups is an encoded list containing:

- the encoded `byte[32]` thread identifier,
- the encoded `long` version of the backup stored in this thread,
- the encoded `String` of a URL allowing the client to download the encrypted backup directly from AWS S3.

The encrypted backup is downloaded with a `GET` request to this URL, without authentication.

76.4 Backup deletion

The `/backupDelete` entry point deletes the backup stored in a thread. The version passed as input must be the version currently stored in this thread. When the last thread of a backupKeyUid is deleted, the backupKeyUid itself is deleted from the server, along with its public key.

/backupDelete		

in	`byte[32]`
	`byte[32]`
	`long`
	`byte[]`
		backupKeyUid
		thread identifier
		version
		signature

out	`byte[1]`	return status
return statuses		`0x00`: OK `0x14`: invalid signature `0x1b`: unknown backupKeyUid `0x1c`: unknown thread identifier `0x1d`: unknown version `0xfe`: parsing error `0xff`: unknown error

The signature is computed as for `/backupUpload`, with the backup deletion signature prefix `"backupDelete"` and, as `challenge`, the encoded list of the three previous elements of the input.

77 Other REST API Entry Points

77.1 Upload return receipts

The application can notify a message sender that their message was indeed delivered to its recipient (or read by the user). This entry point allows uploading a batch of return receipts to the server. The delivery of the return receipt to the application message sender is then done through the WebSocket.

/uploadReturnReceipt		

in	`[`
	`identity`
	`[encoded_vals]`
	`byte[16]`
	`byte[]`
	`]`
		a repetition of n of the following list-encoded 4 elements
		`identity`
		list of encoded `deviceUid`
		`nonce`
		encrypted return receipt payload

out	`byte[1]`	return status
return statuses		`0x00`: OK `0xff`: unknown error

The list of encoded `deviceUid` corresponds to the list of devices of the application message sender. It lets the server know which devices should be notified.

The 16-byte `nonce` is part of the application message the recipient receives. It is sent back to the message sender to allow them to identify which key to use to decrypt the return receipt payload. The content of the payload itself is sent encrypted so as not to disclose any information to the server about when a message is read. The payload is an encoded list containing:

- the `identity` of the application message recipient (the sender of the return receipt)
- an `int` representing the status of the return receipt: 1 for message delivered, 2 for message read,
- optionally, an `int` containing the number of an attachment, for return receipts concerning a specific attachment of the message.

Note that return receipts also contain a timestamp which is set by the server during the execution of this entry point. Clients upload at most 50 return receipts in each call to this entry point.

77.2 Retrieve TURN credentials to initiate a call

This entry point requires a license with the call permission. When checking the validity of the `token`, the server also checks the associated permission. If the user is allowed to initiate calls, the server returns two timestamped usernames and the associated passwords. The caller and recipient can use these to authenticate with the TURN server (see Section 53).

`/getTurnCredentials`		

in	`identity`
	`byte[32]`
	`String`
	`String`
		`identity`
		authentication `token`
		username 1
		username 2

out	`byte[1]`
	`String`
	`String`
	`String`
	`String`
		return status
		timestamped username 1
		password 1
		timestamped username 2
		password 2

return statuses		`0x00`: OK `0x04`: invalid `token` (re-authentication required) `0x0e`: permission denied (call initiation not allowed for user) `0xff`: unknown error

77.3 Upload pre-key

The `/uploadPreKey` entry point allows a user to create or update the pre-key stored by the server (see Section 25). Each pre-key is signed using the `identity` authentication key and the server checks this signature before accepting a pre-key. Contacts may retrieve such a pre-key using the `/deviceDiscovery` entry point. The encoded signed pre-key is the encoded list described in

Section 25, directly included in the input list (it is not wrapped in an encoded `byte[]`).

`/uploadPreKey`		

in	`identity`
	`byte[32]`
	`byte[32]`
	`[encoded_vals]`
		`identity`
		authentication `token`
		`deviceUid`
		encoded signed pre-key

out	`byte[1]`	return status
return statuses		`0x00`: OK `0x04`: invalid `token` (re-authentication required) `0x0b`: device is not registered, or is inactive or expired `0x14`: invalid signature `0xff`: unknown error

The invalid signature return status indicates that either the pre-key signature is invalid, or the pre-key is for another `deviceUid`, or the server already has a pre-key with a later expiration timestamp. See Section 25 for details on how pre-keys are signed and how the signature can be verified.

77.4 Keycloak queries

The final entry point of the REST API is the `/keycloakQuery` entry point, used by Keycloak servers of clients of the Olvid Enterprise offer to:

- request new licenses,
- manage Keycloak push topics,
- notify users of an update on the Keycloak server (typically a key revocation or a group update).

This entry point is out of the scope of these specifications and is not detailed here.

78 WebSocket API

When the application is running, it continuously stays connected to the Olvid server through a WebSocket, whose URL is given by the server configuration (see Section 68). This WebSocket is used to receive new message push notifications in a more efficient/responsive way than through Apple and Google's push notification services, to receive return receipts, and to receive the other notifications described below. Note that push notifications will be sent through both the WebSocket and Apple and Google's services, but return receipts are only sent through the WebSocket. For devices which cannot receive push notifications (see Section 73.1), the WebSocket is the only notification channel.

The WebSocket is a fully asynchronous 2-way communication channel between the device and the server. All messages transmitted through the WebSocket are formatted in JSON and must contain an `"action"` key determining the scope of this message. As opposed to a REST API,

the channel being asynchronous, the WebSocket API does not expect an immediate response after sending a message.

78.1 Device registration

As soon as the connection to the WebSocket is established, the device sends a registration message. This message binds an `identity` and a `deviceUid` to the WebSocket, allowing the server to know through which WebSocket to send information. Note that if multiple identities are configured on the same device, multiple registration messages can be sent.

`"action": "register"`		
direction		device → server

JSON	`"identity"`
	`"token"`
	`"deviceUid"`
	(`"os"`,
	`"osVersion"`,
	`"appBuild"`,
	`"appVersion"`)
		base64-encoded `identity`
		base64-encoded authentication `token`
		base64-encoded `deviceUid`
		(optional) `String` operating system,
		`String` operating system version,
		`int` application build number,
		`String` application version

The optional keys are only used by the server for statistics. In response to this message, the server will send an error message or an acknowledgement message of the following form.

`"action": "register"`		
direction		server → device
JSON	`"identity"` (`"err"`)	base64-encoded `identity` (optional) `int` containing the error code

If registration was successful, the `"err"` key is omitted and the message simply contains the `identity` to let the device know which `identity` was successfully registered. If registration failed, the `"err"` key is present and contains a `byte` (in integer representation) representing the type of error:

- `0x04`: invalid `token` (re-authentication required)
- `0xff`: unknown error (in this case, the `"identity"` key is absent if the registration message could not be parsed)

After a successful registration, the server sends a message notification (without payload) if messages are waiting for this device on the server, as well as the notifications described below that were not delivered to the device.

78.2 Ping

The device may regularly send a ping message to the server, to check that the connection is still alive and measure its latency. Devices may also rely on the ping frames of the WebSocket protocol.

`"action": "ping"`		
direction		device → server
JSON	`"cnt"`	`long` ping counter
	`"timestamp"`	`long` device timestamp (milliseconds since Epoch)

The server responds with the same JSON message, in which the `"action"` is replaced by `"pong"`.

78.3 Message notification

When sending a push notification to the device, if it is connected through a WebSocket and registered to the `identity`, the server sends a message. For messages without attachments, and whose size is at most 10 000 bytes, the message is directly sent: the payload is the base64 encoding of an encoded list containing the message unique identifier on the server, the `long` server timestamp of the message, the header for this device, and the encrypted message payload (but not the extended payload). For other messages, no payload is sent, instructing the application to trigger a `/downloadMessagesAndListAttachments`. A message without payload is also sent when all the attachments of a message have been uploaded or cancelled by its sender, as the message is only listed from then on.

`"action": "message"`		
direction		server → device
JSON	`"identity"`	base64-encoded `identity`
	`("message")`	(optional) base64-encoded message (see above)

The server does not expect any response to this message.

78.4 Owned devices notification

Certain actions on the server may require all your devices to call the `/ownedDeviceDiscovery` entry point. Typical cases include one of your devices expiring, the registration of a new device, the deactivation or renaming of a device, or a modification to your license. In that case, the server sends the following message to all your devices. This notification, as well as the Keycloak notification below, is also sent through push notifications and, for devices that only receive notifications through the WebSocket, kept on the server for 7 days until the device registers.

"action": "ownedDevices"		
direction		server → device
JSON	"identity"	base64-encoded identity

The server does not expect any response to this message.

78.5 Keycloak notification

When a Keycloak server needs an identity to refresh some information, it instructs the server to send the following message:

"action": "keycloak"		
direction		server → device
JSON	"identity"	base64-encoded identity

The server does not expect any response to this message.

78.6 Push topic notification

When a Keycloak server needs a group of users to refresh some information, it instructs the server to send the following push topic message. This message is only sent to devices that are not able to receive push notifications. The topic name is included in the message.

"action": "push_topic"		
direction		server → device
JSON	"topic"	topic name String

The server does not expect any response to this message.

78.7 Return receipt download

Once an identity is registered, it will also receive return receipts through the WebSocket. After a successful registration, the device receives one return receipt message for each pending return receipt on the server. A message is also sent directly after a return receipt is uploaded if a WebSocket is currently connected.

"action": "return_receipt"		
direction		server → device
JSON	"identity"	base64-encoded identity
	"serverUid"	base64-encoded byte[32] identifier for this return receipt
	"nonce"	base64-encoded byte[16] nonce
	"encryptedPayload"	base64-encoded byte[] encrypted return receipt payload
	"timestamp"	long server timestamp of the return receipt upload

The `serverUid` is a unique identifier used only to delete the return receipt on the server (see next section). The `nonce` allows the application to identify which key was used to encrypt the return receipt payload. Once decrypted, the return receipt payload is an encoded list containing:

- the `identity` of the application message recipient (the sender of the return receipt)
- an `int` representing the status of the return receipt: 1 for message delivered, 2 for message read,
- optionally, an `int` containing the number of an attachment, for return receipts concerning a specific attachment of the message.

The timestamp is set by the server during the upload return receipt entry point execution (see Section 77.1).

78.8 Return receipt deletion

Once a return receipt is received on the device, it can be deleted from the server. The device simply sends the following message.

"action": "delete_return_receipt"		
direction		device → server
JSON	"serverUid"	base64-encoded byte[32] identifier for this return receipt

The device does not expect any response to this message.

79 Transfer WebSocket API

The transfer WebSocket API is used only when transferring an `identity` from one device to another in a multi-device setting (see Section 40). This API is used to connect the two devices to one another and relay messages between them. Its URL is `wss://transfer.olvid.io`. As for the main WebSocket API, all messages transmitted through this API are formatted in JSON and must contain an `"action"` key determining the purpose of the message, and `byte[]` payloads are base64-encoded. Again, this API does not systematically expect an immediate response after sending a message, but all entry points may return an error message in the form of a JSON message containing a single `int` element `"errorCode"`. These error codes may be:

- **-1**: unknown error
- **1**: unknown `"sessionNumber"`
- **2**: peer has disconnected (`"connectionId"` no longer exists)
- **3**: payload too large

79.1 Source connection

The source device (the device where the `identity` to transfer is active) initiates a transfer with a `source` request.

`"action": "source"`		
direction		device → server
JSON	-	*no payload for this message*

When receiving this message, the server will respond with the following message (without any action).

no action		
direction		server → device
JSON	`"awsConnectionId"` `"sessionNumber"`	a unique `String` identifier for this connection a unique 8-digit `int` the user enters on the target device (see below)

The session number is chosen at random by the server (it may have leading zeros, so it is displayed with 8 digits) and is valid for 5 minutes.

79.2 Target connection

Once the target device knows the `"sessionNumber"` of the source, it can send the following `"target"` message. This message contains a `"payload"` that is relayed to the source device in the form of a relayed message (see Section 79.4).

`"action": "target"`		
direction		device → server
JSON	`"sessionNumber"` `"payload"`	the 8-digit `int` that was displayed on the source device a first `byte[]` payload to relay to the source device

The device does not expect any response to this message, unless an error occurs (see the error codes above).

79.3 Relay request

Communications between the source and target devices take the form of relay requests that trigger a relay message sent to the other device. As WebSocket messages are limited in size, a fragmentation mechanism is implemented: payloads larger than 10 000 bytes are split into fragments of 10 000 bytes (before base64 encoding), numbered from 0 to `totalFragments` − 1. For small enough messages `"fragmentNumber"` and `"totalFragments"` are not set, but if the message is fragmented they allow the recipient to rebuild the complete message payload.

`"action": "relay"`		
direction		device → server
JSON	`"relayConnectionId"`	the `String` AWS connectionId of the peer
	`"payload"`	the `byte[]` payload to relay to the peer
	`"fragmentNumber"`	if present, the `int` indicating the fragment number
	`"totalFragments"`	if present, the `int` indicating the total fragment count

The device does not expect any response to this message, but the peer will receive the relayed message detailed in the next section.

79.4 Relayed message

After a `"target"` message or a `"relay"` request, the server sends the following message to the peer, if it is still connected (the fragment fields are only present after a fragmented `"relay"` request).

no action		
direction		server → device
JSON	`"otherConnectionId"`	the `String` AWS connectionId of the sender
	`"payload"`	the `byte[]` payload relayed from the sender
	`"fragmentNumber"`	if present, the `int` indicating the fragment number
	`"totalFragments"`	if present, the `int` indicating the total fragment count

80 Push Notifications Content

80.1 Android - Firebase push notifications

On Android, when the server receives a message for a registered `identity` (i.e. an `identity` for which a device successfully called the register push notification entry point described in Section 73.1), it sends a "background" notification (i.e. a notification without a title and body) to each registered device, using the device token received from the Firebase service on the device. Notifications are never sent to inactive or expired devices. The data of the push notification is the following.

```
{
  "identity": "[identity_mask]"
}
```

Here `identity_mask` is a random identifier chosen by the device to mask the real `identity` of the user (see Section 73.1), sent as a hexadecimal string. The notification has a high priority for application messages and VoIP messages, and a normal priority otherwise.

Keycloak and owned devices notifications (see Section 78) have the same structure, with an additional `"keycloak": "true"` or `"ownedDevices": "true"` key. Keycloak push topic notifications only contain a `"topic"` key, whose value is the name of the topic.

If the device is connected to the WebSocket, another notification is also sent through it (see Section 78.3).

80.2 iOS - Apple push notifications

On iOS, "background" notifications may be arbitrarily delayed. In order to have notifications delivered as fast as possible, we have the option to send an "alert" notification containing the title and body to display in the notification. The content of the notification is encrypted and sent to a notification extension of the application which takes care of decrypting the title and body. In addition to this, "voip" notifications are also delivered instantly and are received directly by the App, but require it to initiate an incoming call flow with CallKit.

Because of this, we distinguish three types of messages on the server, depending on whether there is something to display to the user and the VoIP flag is set (see Section 71.1):

- protocol messages without an application payload, triggering only the background notification
- application messages with something to display to the user, triggering the background notification and, if the encrypted message payload is smaller than 2048 bytes, the alert notification
- VoIP messages, triggering only a VoIP notification (which starts an incoming call dialog in CallKit) if the device registered a VoIP token, and handled as application messages otherwise

Background notification. The background push notification has the following structure.

```
{
  "aps": {
    "content-available": 1
  },
  "maskinguid": "[masking_uid]"
}
```

Here `masking_uid` is a random identifier chosen by the device to mask the real `identity` of the user (see Section 73.1), sent as a hexadecimal string.

Alert notification. When sent, the alert notification has the following structure.

```
{
  "aps": {
    "alert": {
      "title": "Olvid",
      "body": "Olvid requires your attention.",
      "title-loc-key": "Olvid",
      "loc-key": "Olvid requires your attention."
    },
    "mutable-content": 1
  },
```

```
  "timestamp": [timestamp],
  "maskinguid": "[masking_uid]",
  "messageuid": "[message_uid]",
  "encryptedHeader": "[header]",
  "encryptedMessage": "[content]",
  "extendedContent": "[extended_content]"
}
```

Here:

- `timestamp` is a `long` corresponding to the message timestamp on the server (milliseconds since Epoch)
- `masking_uid` is a random identifier chosen by the device to mask the real `identity` of the user (see Section 73.1), sent as a hexadecimal string
- `message_uid` is the unique message identifier on the server, sent as a hexadecimal string
- `header` is the encrypted header received by the server for this device, sent as a base64 string (see Section 71.1)
- `content` is the encrypted message payload received by the server, sent as a base64 string (see Section 71.1)
- `extended_content` is the extended message content and is only included if there is room (the notification is limited to 4096 bytes). It is sent as a base64 string.

When received, the `mutable-content` flag triggers the call to the Olvid notification extension which decrypts the `header` and `content` and shows a notification to the user. If the notification extension fails, the default localized title and body are displayed.

VoIP notifications. When sent, VoIP notifications have the following structure. In addition, specific APNS flags are set to give them the `voip` push type.

```
{
  "aps": {
    "alert": "alert"
  },
  "timestamp": [timestamp],
  "maskinguid": "[masking_uid]",
  "messageuid": "[message_uid]",
  "encryptedHeader": "[header]",
  "encryptedMessage": "[content]"
}
```

The encrypted content is the same as for alert notifications, with the difference that the `content` does not contain a text message, but the SDP of the start call message. Apple limits VoIP notifications to 5kB, which is one of the reasons why SDPs are compressed (with raw DEFLATE, see Section 52) before being sent.

Other notifications. Keycloak and owned devices notifications (see Section 78) are background notifications containing, in addition to the `"maskinguid"`, a `"keycloak": "true"` or `"ownedDevices": "true"` key. Keycloak push topic notifications are background notifications containing only a `"topic"` key, whose value is the name of the topic.

References

[1] AWS Documentation. Serving Private Content with Signed URLs and Signed Cookies. https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/PrivateContent.html.

[2] Thomas Baignères, Cécile Delerablée, Matthieu Finiasz, Louis Goubin, Tancrède Lepoint, and Matthieu Rivain. Trap Me If You Can – Million Dollar Curve, February 2016. https://eprint.iacr.org/2015/1249.pdf.

[3] Elaine Barker and John Kelsey. NIST Special Publication 800-90A Revision 1 – Recommendation for Random Number Generation Using Deterministic Random Bit Generators, June 2015. http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-90Ar1.pdf.

[4] Daniel J. Bernstein. Edwards coordinates for elliptic curves (Slides of the NSA talk). http://cr.yp.to/talks/2007.06.07/slides.pdf.

[5] Daniel J. Bernstein. Curve25519: New Diffie-Hellman Speed Records. In *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings*, Lecture Notes in Computer Science, 2006.

[6] Daniel J. Bernstein, Peter Birkner, Marc Joye, Tanja Lange, and Christiane Peters. Twisted edwards curves. In Serge Vaudenay, editor, *Progress in Cryptology - AFRICACRYPT 2008, First International Conference on Cryptology in Africa, Casablanca, Morocco, June 11-14, 2008. Proceedings*, volume 5023 of *Lecture Notes in Computer Science*, pages 389–405. Springer, 2008.

[7] Daniel J. Bernstein and Tanja Lange. Edwards coordinates for elliptic curves. http://cr.yp.to/newelliptic/newelliptic.html.

[8] Daniel J. Bernstein and Tanja Lange. Faster addition and doubling on elliptic curves. In Kaoru Kurosawa, editor, *Advances in Cryptology - ASIACRYPT 2007, 13th International Conference on the Theory and Application of Cryptology and Information Security, Kuching, Malaysia, December 2-6, 2007, Proceedings*, volume 4833 of *Lecture Notes in Computer Science*, pages 29–50. Springer, 2007.

[9] Henri Cohen and Gerhard Frey, editors. *Handbook of Elliptic and Hyperelliptic Curve Cryptography*. Discrete mathematics and its applications. Chapman & Hall/CRC, 2006.

[10] coturn TURN server project. Available on GitHub. https://github.com/coturn/coturn.

[11] Morris J. Dworkin, Elaine B. Barker, James R. Nechvatal, James Foti, Lawrence E. Bassham, E. Roback, and James F. Dray Jr. Federal Inf. Process. Stds. (NIST FIPS) - 197 – Advanced Encryption Standard (AES), November 2001. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf.

[12] Harold M. Edwards. A normal form for elliptic curves. *Bulletin (New Series) of the American Mathematical Society*, 44(3):393–422, July 2007. http://www.ams.org/journals/bull/2007-44-03/S0273-0979-07-01153-6/S0273-0979-07-01153-6.pdf.

[13] Tanja Lange. Side-channel attacks and countermeasures for curve based cryptography, May 28 2007. http://www.hyperelliptic.org/tanja/vortraege/Lange_SCA.ps.

[14] Peter L. Montgomery. Speeding the Pollard and Elliptic Curve Methods of Factorization.

Mathematics of Computation, 48(177):243–264, January 1987.

[15] National Institute of Standards and Technology (NIST). FIPS PUB 180-4 – Secure Hash Standard (SHS), August 2015. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf.

[16] Sylvain Pasini. *Secure Communication Using Authenticated Channels*. PhD thesis, EPFL, 2009. https://infoscience.epfl.ch/record/138488/files/EPFL_TH4452.pdf.

[17] Claus P. Schnorr. Efficient Signature Generation by Smart Cards. *Journal of Cryptology*, 4(3):161–174, 1991.

[18] Victor Shoup. A Proposal for an ISO Standard for Public Key Encryption (version 2.1), December 20 2001. https://www.shoup.net/papers/iso-2_1.pdf.

[19] SQLCipher. https://www.zetetic.net/sqlcipher/.

[20] J. Uberti. A rest api for access to turn services. https://tools.ietf.org/html/draft-uberti-behave-turn-rest-00.

Appendices

A Elliptic Curves

A.1 Edwards Curves

The elliptic curves considered in these specifications are Edwards curves [7, 8, 12] or are birationally equivalent to an Edwards curve. Given a finite field $\mathbf{F}$ of odd characteristic, an Edwards curve over $\mathbf{F}$ is

$$E(\mathbf{F}) : x^2 + y^2 = 1 + dx^2y^2$$

with $d \notin \{0, 1\}$.

When d is not a square in $\mathbf{F}$, Edwards curves have a *complete* addition law [8, Theorem 3.3]. Given $(x_1, y_1), (x_2, y_2) \in E(\mathbf{F})$, the Edwards addition law defined by

$$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1y_2 + y_1x_2}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 - x_1x_2}{1 - dx_1x_2y_1y_2} \right)$$

corresponds to the standard elliptic curve addition law and works for all points (i.e., even if one of the points is the point at infinity, and even if the points are actually the same point). All curves used within these specifications are such that d is not a square in $\mathbf{F}$, so the addition law above can be used.

In what follows, we denote by

- p the prime order of the underlying finite field $\mathbf{F}_p$,
- d the parameter defining the Edwards curve over $\mathbf{F}_p$, such that d is not a square in $\mathbf{F}_p$,
- $G = (G_x, G_y)$ the base point explicitly defined by the curve,
- q the prime order of the subgroup generated by G,
- q' the order of the largest prime subgroup of the twist of $E(\mathbf{F}_p)$,
- ν the lcm of the cofactor $\#E(\mathbf{F}_p)/q$ of the curve and of the cofactor $\#E'(\mathbf{F}_p)/q'$ of the twist E' of E.

The neutral element of the group of points of an Edwards curve E over $\mathbf{F}_p$ is $(0, 1)$. The point $(0, -1)$ is the unique point of order 2 [4]. The points $(1, 0)$ and $(-1, 0)$ are the unique points of order 4. The opposite of a point (x, y) is $-(x, y) = (-x, y)$.

A.2 Equivalence with Montgomery Curves

Any Edwards curve over a finite field $\mathbf{F}$ of odd characteristic is birationally equivalent to a Montgomery curve (consequence of Theorem 3.2 in [6]). More precisely the Edwards curve

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

where d is not a square in $\mathbf{F}_p$ is birationally equivalent to the Montgomery curve

$$E_{A,B} : Bv^2 = u^3 + Au^2 + u,$$

where $A = 2(1+d)/(1-d)$ and $B = 4/(1-d)$ are such that $A \in \mathbf{F} \setminus \{-2, 2\}$ and $B \in \mathbf{F} \setminus \{0\}$. The map

$$(x, y) \mapsto (u, v) = \left(\frac{1+y}{1-y}, \frac{1+y}{1-y}\frac{1}{x}\right)$$

is a birational equivalence from E_d to $E_{A,B}$, the inverse mapping being

$$(u, v) \mapsto (x, y) = \left(\frac{u}{v}, \frac{u-1}{u+1}\right).$$

The only points of E_d for which the mapping is not defined are those for which $y = 1$ and those for which $x = 0$. Since $d \neq 1$, the only possibilities are

- $(0, 1)$, which is the point at infinity, and
- $(0, -1)$, which is the only point of order 2 of E_d.

Besides the point at infinity, the only points of $E_{A,B}$ for which the inverse mapping is not defined are those for which $v = 0$ and those for which $u = -1$. Considering a Montgomery curve for which there exists some non-square d such that $A = 2(1+d)/(1-d)$ and $B = 4/(1-d)$, we have the following results:

- Since d is not a square in $\mathbf{F}_p$, $A^2 - 4 = \frac{16d}{(1-d)^2}$ is not a square either. Thus, the only point for which $v = 0$ is $(0, 0)$.
- For $u = -1$, we must solve $v^2 = (A-2)/B = d$. Since d is not a square in $\mathbf{F}_p$, there is no solution.

The bottom line is that, for the Edwards curves we consider in this document, the only problematic points for the mapping are the point at infinity $(0, 1)$ and the only point of order 2, which is $(0, -1)$. We emphasize that $(0, 1)$ is the only point of E_d with $y = 1$, and $(0, -1)$ is the only point of E_d with $y = -1$. So even when working with y-only coordinates (see below), those problematic points can be dealt with.

A.2.1 Montgomery Ladder for Edwards Curves

Given a point P on a Montgomery curve, it is possible to restrict computations to the x-coordinate only to compute nP. The technique first appeared in [14] and is well documented in [9, 13]. This section describes the Montgomery ladder and a straightforward way to benefit from this technique when working with an Edwards curve.

A.2.2 Montgomery Ladder

Whatever the form of the curve, the Montgomery ladder computes nP from P by performing, at each step, exactly one point addition and one doubling (which makes it less subject to side-channel attacks than a simple square-and-multiply technique). Denoting $n = (n_{\ell-1}n_{\ell-2}\ldots n_1 n_0)$ the binary representation of n (where $n_{\ell-1}$ is the most significant bit), the Montgomery ladder computes nP as follows:

Algorithm 1 Montgomery ladder: compute nP from P

1: $Q_\ell = 0$ and $R_\ell = P$
2: **for** $i = \ell$ down to 1 **do**
3: **if** $n_{i-1} = 0$ **then**
4: $Q_{i-1} = 2Q_i$ and $R_{i-1} = Q_i + R_i$
5: **else**
6: $Q_{i-1} = Q_i + R_i$ and $R_{i-1} = 2R_i$
7: **end if**
8: **end for**
9: return Q_0

It is easy to see that we always have $R_i - Q_i = R_{i+1} - Q_{i+1} = \cdots = P$.

A.2.3 The Ladder on a Montgomery Curve

Montgomery shows [14] how to compute the u-coordinate of $Q_i + R_i$, $2Q_i$, and $2R_i$, when both Q_i and R_i are multiples of P and $R_i = Q_i + P$, using only the u-coordinates of Q_i, R_i, and P. These formulas use projective coordinates. Starting with $P = (u_P, v_P)$ on a Montgomery curve $E_{A,B}$, we write P in projective coordinates $P = (U_P : V_P : W_P)$ where $U_P = u_P$, $V_P = v_P$, and $W_P = 1$. With $Q = (U_Q : \cdot : W_Q)$, $R = (U_R : \cdot : W_R)$, $Q + R = (U_{Q+R} : \cdot : W_{Q+R})$, and $2Q = (U_{2Q} : \cdot : W_{2Q})$, we have

$$\begin{aligned} U_{Q+R} &= W_P \left((U_Q - W_Q)(U_R + W_R) + (U_Q + W_Q)(U_R - W_R)\right)^2 \\ W_{Q+R} &= U_P \left((U_Q - W_Q)(U_R + W_R) - (U_Q + W_Q)(U_R - W_R)\right)^2 \end{aligned}$$

and

$$\begin{aligned} U_{2Q} &= (U_Q + W_Q)^2 (U_Q - W_Q)^2 \\ W_{2Q} &= 4U_Q W_Q \left((U_Q - W_Q)^2 + \frac{A+2}{4}(4U_Q W_Q)\right) \end{aligned}$$

Note that $4U_Q W_Q = (U_Q + W_Q)^2 - (U_Q - W_Q)^2$ and that $\frac{A+2}{4}$ can be precomputed. Thus the addition costs 4 field multiplications and 2 field squarings, while the doubling costs 3 field multiplications and 2 field squarings.

A.2.4 The Ladder on an Edwards Curve

As we will see in the next section, the curves we consider in these specifications are birationally equivalent to a Montgomery curve, so there is a straightforward way to benefit from the algorithm of the previous section. We provide a precise description of how this can be achieved in Section 12.

A.3 The Curves We Consider in these Specifications

In this version of the specifications, we consider two specific elliptic curves, namely, the Million Dollar Curve and Curve25519.

A.3.1 Million Dollar Curve

Million Dollar Curve is an Edwards curve over the prime field $\mathbf{F}_p$ with

$$p = 1091123632769611904427110903691495516763303076461182045177715113305362531563 71,$$

defined by

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

where

$$d = 393848177413506285731611843012259158003587705889337560719482646258046122597 21.$$

A.3.2 Curve25519

Curve25519 is a Montgomery curve over the prime field $\mathbf{F}_p$ with $p = 2^{255} - 19$ defined by

$$y^2 = x^3 + 486662x^2 + x.$$

Curve25519 allows simple point compression when used for ECDH since it allows restricting to x-coordinate-only scalar multiplication. The base point suggested by Bernstein thus only specifies the x-coordinate:

$$G = (9, \cdot)$$

The order of the subgroup generated by G is a prime larger than 2^{252}. Bernstein and Lange show in [8, Sec. 2] that this curve is birationally equivalent over $\mathbf{F}_p$ to the Edwards curve

$$x^2 + y^2 = 1 + \frac{121665}{121666}x^2y^2.$$