Olvid

Specifications of Olvid

–

Application and Server

Date	October 7, 2024
Number of Pages	157
Written by	Dr. Thomas Baignères Dr. Matthieu Finiasz

Contents

About this document

Olvid is a secure instant messaging application similar in its functionnalities to many other instant messaging applications like WhatsApp, Signal, Citadel, etc. Olvid is architectured 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. The application layer does not implement any form of cryptographic operation and is thus out of the scope of this document. 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, contrary to 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 himself) simply retreives 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 between 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 as 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.

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 x509 certificates, the `identity` itself does not contain any identification element allowing to 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 27).

With a multi-device licence, 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 transfering a profile to another device (see Section 51).

Olvid users without a paid licence are limited to having one `deviceUid` registered on the server: adding a new device requires to decide 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 allowing to 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/#AwAAAIgAAAAAWmh0dHBzOi8vc2VydmVyLm9sdmlkLml\
vAADoTcM7E5duFaKw1mpuyGROJkSM51KOEulxyQEdabcimADZmeYVvTlSy5kkAtfM4o2JJuj\
sZTkrSG-B6VshvRU5gwAAAAAkTWF0dGhpZXUgRmluaWFzeiAoRGV2ZWxvcGVyIEAgT2x2aWQp
```

Such an Invitation Link can also be embedded in a QR-code for direct scanning on a smartphone as seen on 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, Procedures 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 parameters 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 its input/output parameters and its arguments. As an example, here is the table for the procedure Proc:

Proc		
parameters		None
in	$\mathbb{Z}$	a
	$\mathbb{Z}$	$b = 13$
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 $c \in \{0, \dots, 255\}^*$ is an array of bytes, we denote by

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

the length of the array. For example, if $c \in \{0, \dots, 255\}^\ell$ 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 details 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 as 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 allows 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 21.9, 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 allows

to know whether the key is an AES or a DFC (Decorrelated Fast Cipher) key.

- key.dict: A dictionary (in the sense of Section 21.8) which (dictionary) keys depends on the two above bytes.
- key.encodingByteId: A byte that specifies which encoding byte identifier (in the sense of Section 19.1) 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	$\{0, \ldots, 255\}$	algoClassByteId
	$\{0, \ldots, 255\}$	algoImplemByteId
	Dictionary	dict
	$\{0, \ldots, 255\}$	encodingByteId

We denote by

$$\mathsf{CryptographicKey}(\mathtt{algoClassByteId}, \mathtt{algoImplemByteId}, \mathtt{dict}, \mathtt{encodingByteId}) \to \mathsf{key}$$

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 imposes to concrete subtypes to specify the following static value:

- SymmetricKey.length: the raw byte length of the symmetric key. For example, this is 16 for AES256.

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

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

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

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

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

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

the call to the above PublicKey initializer.

```
procedure PublicKey(algoClassByteId, algoImplemByteId, dict)
    encodingByteId ← 0x91
    CryptographicKey(algoClassByteId, algoImplemByteId, dict, encodingByteId)
end procedure
```

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

We denote by

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

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

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

the call to the above PrivateKey initializer.

```
procedure PrivateKey(algoClassByteId, algoImplemByteId, dict)
    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 allowing to compute 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].

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-lenght of a block.

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

We denote by

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

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

E.decrypt		
parameters		$\ell_k, \ell_m \in \mathbb{N}$
in	$\{0,\dots,255\}^{\ell_m}$	c
	$\{0,\dots,255\}^{\ell_k}$	k
out	$\{0,\dots,255\}^{\ell_m}$	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 allowing to encrypt with AES-256.

AES256.encrypt		
parameters		$\ell_k = 32,\ \ell_m = 16$
in	$\{0, \ldots, 255\}^{16}$	`m`
	$\{0, \ldots, 255\}^{32}$	`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	$\{0, \ldots, 255\}^{16}$	`c`
	$\{0, \ldots, 255\}^{32}$	`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	$\{0, \ldots, 255\}$	`algoImplemByteId`
	Dictionary	`dict`

We denote by

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

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	$\{0, \ldots, 255\}^*$	m
	SymEncKey	symEncKey
	$\{0, \ldots, 255\}^{\ell_{iv}+\ell_n}$	$\texttt{iv} \,\|\, \texttt{n}$
out	$\{0, \ldots, 255\}^*$	c

We denote by

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

the symmetric encryption with the symmetric encryption algorithm SymEnc of the message $\texttt{m} \in \{0, \ldots, 255\}^*$ under the key symEncKey $\in$ SymEncKey, initial vector $\texttt{iv} \in \{0, \ldots, 255\}^{\ell_{iv}}$, and nonce $\texttt{n} \in \{0, \ldots, 255\}^{\ell_n}$. An initial 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	$\{0, \ldots, 255\}^*$	m
	SymEncKey	symEncKey
out	$\{0, \ldots, 255\}^*$	m

We denote by

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

the symmetric decryption with the symmetric encryption algorithm SymEnc of the ciphertext $\texttt{c} \in \{0, \ldots, 255\}^*$ under the key symEncKey $\in$ 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

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

the static procedure that returns the final length of the ciphertext corresponding to a plaintext of byte-lenght 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

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

the static procedure that returns the plaintext length corresponding to a ciphertext of byte-lenght 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 allowing to encrypt using AES-256 in CTR mode. We denote by AES256CTRKey the concrete subtype of SymEncKey of AES256CTR keys. The following initializer allows to create a AES256CTRKey.

AES256CTRKey (Initializer)		
parameters		None
in	Dictionary	`dict`

We denote by

$$\mathsf{AES256CTRKey}(\texttt{dict}) \rightarrow \mathsf{symEncKey}$$

the call to the AES256CTRKey initializer.

```
1: procedure AES256CTRKey(dict)
2:     encoded_raw ← dict["enckey"]
```

```
3:     raw ← decodeBytes(encoded_raw)
4:     if len(raw) $\neq$ 32 then return $\perp$ 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:     $\ell$ ← length(m)
5:     enc ← AES256.encrypt
6:     let s $\in \{0,\dots,255\}^\ell$ be the $\ell$ first bytes of enc(n$\|\underline{0}$, k) $\|$ enc(n$\|\underline{1}$, k) $\|$ enc(n$\|\underline{2}$, k) $\| \cdots$
7:     return n$\|$(m $\oplus$ s) $\in \{0,\dots,255\}^{8+\ell}$
8: 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

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

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

```
1: procedure AES256CTR.ciphertextLength($\ell_m$)
2:     return $8 + \ell_m$
3: end procedure
```

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

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

7 Message Authentication Code

MAC is one of symmetric keyed primitive 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 ← 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 allowing to compute a MAC based on HMAC with SHA-256. We denote by HMACWithSHA256Key the concrete subtype of MACKey of HMACWithSHA256 keys. The following initializer allows to create 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	$\{0,\dots,255\}^*$	seed
out	HMACWithSHA256Key	hmacKey

```
procedure HMACWithSHA256.generateKey(seed)
    kdf ← KDFFromPRNGWithHMACWithSHA256
    return kdf.compute(seed, HMACWithSHA256Key)
end procedure
```

HMACWithSHA256.generateKey		
parameters		None
in	PRNG	prng
out	HMACWithSHA256Key	hmacKey

```
procedure HMACWithSHA256.generateKey(prng)
    seed ← prng.bytes(32)
    return HMACWithSHA256.generateKey(seed)
end procedure
```

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

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

```
procedure HMACWithSHA256.compute(m, macKey)
    if macKey is not a HMACWithSHA256Key then return ⊥ end if
    encoded_raw ← macKey.dict["mackey"]
    k ← decodeBytes(encoded_raw)
    opad ← $(92, 92, \dots, 92) \in \{0,\dots,255\}^{64}$
    ipad ← $(54, 54, \dots, 54) \in \{0,\dots,255\}^{64}$
    $k_o = (k \| 0 \cdots 0) \oplus$ opad $\in \{0,\dots,255\}^{64}$
    $k_i = (k \| 0 \cdots 0) \oplus$ ipad $\in \{0,\dots,255\}^{64}$
```

```
 9:     return SHA256($\mathtt{k}_o \parallel$ SHA256($\mathtt{k}_i \parallel \mathtt{m}$))
10: end procedure
```

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

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

```
1: procedure HMACWithSHA256.finalOutputSize()
2:     return 32
3: end procedure
```

8 Key Derivation Function (KDF)

A Key Derivation Function (KDF) allows to create 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

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

computes a symmetric key 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 allowing to compute symmetric keys from a seed using the PRNG defined in Section 10.1.

KDFFromPRNGWithHMACWithSHA256.compute		
parameters		None
in	$\{0, \dots, 255\}^*$	`seed`
	SymmetricKey subtype	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 specification are subtypes of Commitment.

Commitment.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

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

the commitment on a tag `tag` and value `value` using the PRNG instance prng. The result is a `commitment` and a `decommitToken` allowing 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

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

the procedure allowing to open a `commitment` and `tag` using the `decommitToken`, which recovers the `value` that was commited. 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

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

the procedure allowing to compute a commitment on a tag `tag` and value `value` using the PRNG instance prng. The result is a `commitment` and a `decommitToken` allowing 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	$\{0,\dots,255\}^*$	`commitment`
	$\{0,\dots,255\}^*$	`tag`
	$\{0,\dots,255\}^*$	`decommitToken`
out	$\{0,\dots,255\}^*$	`value`

We denote by

$$\mathsf{CommitmentWithSHA256.open}(\mathtt{commitment}, \mathtt{tag}, \mathtt{decommitToken}) \to \mathtt{value}$$

the procedure allowing to open a `commitment` and `tag` using the `decommitToken`, which recovers the `value` that was commited.

```
procedure CommitmentWithSHA256.open(commitment, tag, decommitToken)
    computedCommitment ← SHA256(tag || decommitToken)
    if computedCommitment ≠ commitment then return ⊥ end if
    Parse decommitToken as value || e where len(e) = 32
    return value
end procedure
```

10 Pseudorandom Generator

Most pseudorandom generators (PRNGs) require 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

$$\mathsf{prng}.\mathtt{state}$$

the internal state of this instance.

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

We denote by

$$\mathsf{PRNG}(\mathtt{seed})$$

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

$$\mathsf{prng.bytes}(\ell) \to \mathtt{r}$$

the call to the initialized PRNG instance prng that generates a uniformly distributed pseudorandom byte string `r` of ℓ bytes. This procedures 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

$$\mathsf{prng.bigInt}(n) \to a$$

the call to the initialized PRNG instance prng that generates a uniformly distributed pseudorandom random big integer $a \in [0, n-1]$. This procedures 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. $m \geq \lfloor n/256^{\ell-1} \rfloor$
    while True do
        $s = \mathsf{prng.bytes}(\ell)$
        $s[0] = s[0]$ 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 the hash function is SHA-256. 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:

```
procedure prng.update(data)
    k ← self.state.k
    v ← self.state.v
    hmacKey ← HMACWithSHA256Key(k)
    k = HMACWithSHA256.compute(v || 0x00 || data), hmacKey)
    v = HMACWithSHA256.compute(v, hmacKey)
    if length(data) > 0 then
        k = HMACWithSHA256.compute(v || 0x01 || data), hmacKey)
        v = HMACWithSHA256.compute(v, hmacKey)
    end if
    self.state.k ← k
    self.state.v ← v
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:

```
procedure PRNGWithHMACWithSHA256(seed)
    if len(seed) < 32 then return ⊥ end if
    self.state.k = $(0, 0, \dots, 0) \in \{0, \dots, 255\}^{32}$
    self.state.v = $(1, 1, \dots, 1) \in \{0, \dots, 255\}^{32}$
    self.update(seed)
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]):

```
procedure prng.bytes($\ell$)
    k $\leftarrow$ self.state.k
    v $\leftarrow$ self.state.v
    hmacKey $\leftarrow$ HMACWithSHA256Key(k)
    s $\leftarrow$ [ ]
    while len($s$) $<$ $\ell$ do
        v $\leftarrow$ HMACWithSHA256.compute(v, hmacKey)
        s $\leftarrow$ s $\|$ v
    end while
    self.update([ ])
    self.state.k $\leftarrow$ k
    self.state.v $\leftarrow$ v
    truncate s to its first $\ell$ bytes
    return s
end procedure
```

11 Authenticated Encryption

Authenticated Encryption is one of symmetric keyed primitive 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	$\{0, \ldots, 255\}$	algoImplemByteId
	Dictionary	dict

We denote by

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

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

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

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

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

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

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

the static procedure that returns the final length of the ciphertext corresponding to a plaintext of byte-lenght 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-lenght 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 of AES256CTRHMACSHA256 keys. The following initializer allows to create 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`

```
procedure AES256CTRHMACSHA256Key(b)
    if length(b) $\neq$ 64 then return $\perp$ end if
    let $b_1$ denote the first 32 bytes of b
    let $b_2$ denote the last 32 bytes of b
    dict["mackey"] $\leftarrow$ encodeBytes($b_1$)
    dict["enckey"] $\leftarrow$ encodeBytes($b_2$)
    AES256CTRHMACSHA256Key(dict)
end procedure
```

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

```
procedure AES256CTRHMACSHA256.generateKey(seed)
    kdf $\leftarrow$ KDFFromPRNGWithHMACWithSHA256
    return kdf.compute(seed, AES256CTRHMACSHA256Key)
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. A background on elliptic curves is available in Appendix A. An instance curve of EdwardsCurve provides the following parameters:

- curve.p: the primer 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

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

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:     $(p, d, G, q, \nu, \text{card}) \leftarrow$ 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

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

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, then $1 - dy^2$ is invertible modulo p;
- after step 12 we have $p - 1 = 2^s t$ with t odd;

```
1: procedure curve.xCoordinatesFromY($y$)
2:     $(p, d, G, q, \nu, \text{card}) \leftarrow$ 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 $g^{\frac{p-1}{2}} \bmod p \neq 1$
```

```
        $t \leftarrow p-1$ and $s \leftarrow 0$
        **repeat** $t \leftarrow t/2$ and $s \leftarrow s+1$ **until** $t \bmod 2 \neq 0$
        $e \leftarrow 0$
        **for** $i \leftarrow 2$ to $s$ **do**
            **if** $(x_2 g^{-e})^{\frac{p-1}{2^i}} \bmod p \neq 1$ **then** $e \leftarrow 2^{i-1} + e$ **end if**
        **end for**
        $x \leftarrow g^{-t\frac{e}{2}} x_2^{\frac{t+1}{2}} \bmod p$
    **end if**
    return $(x, -x \bmod p)$
**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 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

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

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 $n = (n_{\ell-1} n_{\ell-2} \ldots n_1 n_0)$ 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):

```
**procedure** curve.scalarMultiplication$(n, y)$
    $(p, d, G, q, \nu, \text{card}) \leftarrow$ curve.parameters
    **if** $n = 0$ or $y = 1$ **then** return 1 **end if**
    **if** $y = -1$ **then** return $1 - 2 \times (n \bmod 2)$ **end if**
    $c \leftarrow (1-d)^{-1} \bmod p$,
    $u_P \leftarrow (1+y) \bmod p$ and $w_P \leftarrow (1-y) \bmod p$
    $u_Q \leftarrow 0$, $w_Q \leftarrow 0$, $u_R \leftarrow u_P$, and $w_R \leftarrow w_P$
    $n \leftarrow n \bmod \text{card}$
    **for** $i \leftarrow \ell$ down to 1 **do**
        $t_1 \leftarrow (u_Q - w_Q)(u_R + w_R) \bmod p$
        $t_2 \leftarrow (u_Q + w_Q)(u_R - w_R) \bmod p$
        $u_{Q+R} \leftarrow w_P\,(t_1 + t_2)^2 \bmod p$
        $w_{Q+R} \leftarrow u_P\,(t_1 - t_2)^2 \bmod p$
        **if** $n_{i-1} = 0$ **then**
```

```
            $t_3 \leftarrow (u_Q + w_Q)^2 \bmod p$
            $t_4 \leftarrow (u_Q - w_Q)^2 \bmod p$
            $t_5 \leftarrow t_3 - t_4 \bmod p$
            $u_{2Q} \leftarrow t_3 t_4 \bmod p$
            $w_{2Q} \leftarrow t_5\,(t_4 + c t_5) \bmod p$
            $(u_Q, w_Q) \leftarrow (u_{2Q}, w_{2Q}) \bmod p$ and $(u_R, w_R) \leftarrow (u_{Q+R}, w_{Q+R}) \bmod p$
        else
            $t_3 \leftarrow (u_R + w_R)^2 \bmod p$
            $t_4 \leftarrow (u_R - w_R)^2 \bmod p$
            $t_5 \leftarrow t_3 - t_4 \bmod p$
            $u_{2R} \leftarrow t_3 t_4 \bmod p$
            $w_{2R} \leftarrow t_5\,(t_4 + c t_5) \bmod p$
            $(u_Q, w_Q) \leftarrow (u_{Q+R}, w_{Q+R}) \bmod p$ and $(u_R, w_R) \leftarrow (u_{2R}, w_{2R}) \bmod p$
        end if
    end for
    return $(u_Q - w_Q)(u_Q + w_Q)^{-1} \bmod p$
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 in argument are on the curve. This check has to be made before this function is called. The procedure works as follows:

```
procedure curve.pointAddition($(x_1, y_1), (x_2, y_2)$)
    $(p, d, G, q, \nu, \text{card}) \leftarrow$ curve.parameters
    $t \leftarrow d x_1 x_2 y_1 y_2 \bmod p$
    $z \leftarrow (1 + t)^{-1} \bmod p$
    $x \leftarrow z(x_1 y_2 + y_1 x_2) \bmod p$
    $z \leftarrow (1 - t)^{-1} \bmod p$
    $y \leftarrow z(y_1 y_2 - x_1 x_2) \bmod p$
    return $(x, y)$
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 $n = (n_{\ell-1}n_{\ell-2} \ldots n_1 n_0)$ the binary representation of n (where ℓ is the bit length of curve.card), the following procedure (based on Montgomery ladder, see Section A.2.2) returns both coordinates of nP on the curve:

```
procedure scalarMultiplicationWithX($n, (x, y)$, curve)
    if not curve.isOnCurve($(x, y)$) then return $\perp$ end if
    if $n = 0$ or $y = 1$ then return $(0, 1)$ end if
    if $y = -1$ then return $(0, 1 - 2 \times (n \bmod 2))$ end if
    $(x_1, y_1) \leftarrow (0, 1)$ and $(x_2, y_2) \leftarrow (x, y)$
    $n \leftarrow n \bmod$ card
    for $i \leftarrow \ell$ down to 1 do
        if $n_i = 0$ then
            $(x_2, y_2) \leftarrow$ curve.pointAddition($(x_1, y_1), (x_2, y_2)$)
            $(x_1, y_1) \leftarrow$ curve.pointAddition($(x_1, y_1), (x_1, y_1)$)
        else
            $(x_1, y_1) \leftarrow$ curve.pointAddition($(x_1, y_1), (x_2, y_2)$)
            $(x_2, y_2) \leftarrow$ curve.pointAddition($(x_2, y_2), (x_2, y_2)$)
        end if
    end for
    return $(x_1, y_1)$
end procedure
```

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

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

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

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 there are 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:

```
procedure curve.mulAdd($a, (x_1, y_1), b, (x_2, y_2)$)
    $(x_3, y_3) \leftarrow$ curve.scalarMultiplicationWithX($a, (x_1, y_1)$)
    if $x_2 \neq \perp$ then
        $(x_4, y_4) \leftarrow$ curve.scalarMultiplicationWithX($b, (x_2, y_2)$)
        $(x, y) \leftarrow$ curve.pointAddition($(x_3, y_3), (x_4, y_4)$)
        $(x', y') \leftarrow (x, y)$
    else
        $y_4 \leftarrow$ curve.scalarMultiplication($b, y_2$)
        $(x_4, x'_4) \leftarrow$ xCoordinatesFromY($y_4$, curve)
        $(x, y) \leftarrow$ curve.pointAddition($(x_3, y_3), (x_4, y_4)$)
        $(x', y') \leftarrow$ curve.pointAddition($(x_3, y_3), (x'_4, y_4)$)
    end if
    return $((x, y), (x', y'))$
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 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

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

the call to the procedure that generates a random scalar $\lambda \in [0, \ldots, q-1]$ and computes the point $Q = \lambda \times \mathsf{curve}.G$. The procedure works as follows:

```
procedure curve.generateRandomScalarAndPoint(prng)
    $\lambda \leftarrow 2 +$ prng.bigInt(curve.$q - 2$)
    $Q \leftarrow$ curve.scalarMultiplicationWithX($\lambda$, curve.$G$)
    return $(\lambda, Q)$
end procedure
```

12.1 Curve25519

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

Curve25519 (Initializer)		
parameters		None
in		

```
procedure Curve25519()
    self.$p$ ← $2^{255} - 19$
    self.$d$ ← 20800338683988658368647408995589388737092878452977063003340006470870624536394
    self.$G.x$ ← 9771384041963202563870679428059935816164187996444183106833894008023910952347
    self.$G.y$ ← 46316835694926478169428394003475163141307993866256225615783033603165251855960
    self.$q$ ← 7237005577332262213973186563042994240857116359379907606001950938285454250989
    self.$\nu$ ← 8
end procedure
```

12.2 MDC

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

MDC (Initializer)		
parameters		None
in		

```
procedure Curve25519()
    self.$p$ ← 109112363276961190442711090369149551676330307646118204517771511330536253156371
    self.$d$ ← 39384817741350628573161184301225915800358770588933756071948264625804612259721
    self.$G.x$ ← 82549803222202399340024462032964942512025856818700414254726364205096731424315
    self.$G.y$ ← 91549545637415734422658288799119041756378259523097147807813396915125932811445
    self.$q$ ← 27278090819240297610677772592287387918930509574048068887630978293185521973243
    self.$\nu$ ← 4
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	$\{0, \ldots, 255\}$	`algoClassByteId`
	$\{0, \ldots, 255\}$	`algoImplemByteId`
	EdwardsCurve	curve
	$[0, \ldots, p-1]^2$	P

We denote by

$$\mathsf{PublicKeyOverEC}(\texttt{algoClassByteId}, \texttt{algoImplemByteId}, \mathsf{curve}, P) \to \mathsf{pk}$$

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

$$\mathsf{PublicKeyOverEC}(\texttt{algoClassByteId},\texttt{algoImplemByteId},\mathsf{curve},y)\to\mathsf{pk}$$

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($y \bmod p$, 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

$$\mathsf{PrivateKeyOverEC}(\texttt{algoClassByteId},\texttt{algoImplemByteId},\mathsf{curve},\lambda)\to\mathsf{sk}$$

the call to the PrivateKeyOverEC initializer.

```
1: procedure PrivateKeyOverEC(algoClassByteId, algoImplemByteId, curve, $\lambda$)
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 allows to obtain a compact byte array representation of the public key. We denote by

$$\mathsf{pk.getCompactKey}() \to \texttt{compactKey}$$

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 allows to recover 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	$\{0, \ldots, 255\}$	`algoImplemByteId`
out	EdwardsCurve	curve

We denote by

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

the call to the SignatureOverEC.curveFromAlgoImplemByteId procedure.

```
procedure SignatureOverEC.curveFromAlgoImplemByteId(algoImplemByteId)
    if algoImplemByteId = 0x00 then return MDC() end if
    if algoImplemByteId = 0x01 then return Curve25519() end if
    return $\perp$
end procedure
```

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

We denote by

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

the call to the SignatureOverEC.algoImplemByteIdFromCurve procedure.

```
procedure SignatureOverEC.algoImplemByteIdFromCurve(curve)
    if curve = MDC() then return 0x00 end if
    if curve = Curve25519() then return 0x01 end if
    return $\perp$
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	curve
	$[0, \ldots, p-1]$	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	curve
	$[0, \ldots, q-1]$	λ

We denote by

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

the call to the SignaturePrivateKeyOverEC initializer.

```
1: procedure SignaturePrivateKeyOverEC(curve, $\lambda$)
2:     algoClassByteId ← 0x11
```

```
3:     algoImplemByteId ← algoImplemByteIdFromCurve(curve)
4:     PrivateKeyOverEC(algoClassByteId, algoImplemByteId, curve, $\lambda$)
5: end procedure
```

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

We denote by

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

the call to the SignatureOverEC.generateKeyPair procedure.

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

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

We denote by

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

the procedure that computes the signature σ of a message m under the private key sk (associated to 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 $\neq$ pk.curve then return $\perp$ end if
```

```
    curve ← sk.curve
    $(p, d, G, q, \nu, \mathsf{card}) \leftarrow$ curve.parameters

    /* The generateKeyPair procedure allows to generate a random scalar and point */
    $(\mathsf{pk}', \mathsf{sk}') \leftarrow$ SignatureOverEC.generateKeyPair(prng, curve)

    /* Construct the data to hash */
    Ay′ ← bytesFromBigUInt(pk′.$y$ mod $p$, len($p$))
    Ay ← bytesFromBigUInt(pk.$y$ mod $p$, len($p$))
    data ← Ay′||Ay||m

    /* Hash and map the digest onto a big unsigned integer */
    h ← SHA256(data)
    $e$ ← bigUIntFromBytes(h)

    /* Sign */
    $r$ ← sk′.scalar
    $a$ ← sk.scalar
    $y \leftarrow (r - a \times e) \bmod q$
    z ← bytesFromBigUInt($y$, len($p$))
    $\sigma$ ← h||z
    return $\sigma$
end procedure
```

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

```
procedure SignatureOverEC.verify(pk, m, $\sigma$)
    curve ← pk.curve
    $(p, d, G, q, \nu, \mathsf{card}) \leftarrow$ curve.parameters

    /* Parse the signature */
    if len($\sigma$) $\neq 32 +$ len($p$) then return False end if
    Parse h||z ← $\sigma$ where len(h) = 32

```

```
9:      /* Extract the big integers */
10:     $e \leftarrow$ bigUIntFromBytes(h)
11:     $y \leftarrow$ bigUIntFromBytes(z)
12:
13:     /* Compute the resulting point(s) */
14:     $P \leftarrow (\mathsf{pk.point} \neq \perp \ ?\ \mathsf{pk.point} : (\perp, \mathsf{pk}.y))$
15:     $(A1, A2) \leftarrow \mathsf{mulAdd}(y, G, b, P)$
16:
17:     /* Compute the corresponding data(s) to hash */
18:     Ay $\leftarrow$ bytesFromBigUInt($\mathsf{pk}.y$, len($p$))
19:     A1y $\leftarrow$ bytesFromBigUInt($A1.y$, len($p$))
20:     A2y $\leftarrow$ bytesFromBigUInt($A2.y$, len($p$))
21:     data1 $\leftarrow$ A1y||Ay||m
22:     data2 $\leftarrow$ A2y||Ay||m
23:
24:     /* Hash the data(s).  The signature is valid if there is a match.  */
25:     h1 $\leftarrow$ SHA256(data1)
26:     h2 $\leftarrow$ SHA256(data2)
27:     return (h = h1 or h = h2)
28: end procedure
```

14.1 Signature Key Generation over Curve25519

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

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

We denote by

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

the call to the SignatureOverCurve25519.generateKeyPair procedure.

```
1: procedure SignatureOverCurve25519.generateKeyPair(prng)
2:     curve $\leftarrow$ 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 that allows to perform and check digital signatures over MDC.

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

We denote by

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

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, authentications 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	$\{0, \ldots, 255\}$	`algoImplemByteId`
out	EdwardsCurve	curve

We denote by

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

the call to the AuthenticationOverEC.curveFromAlgoImplemByteId procedure.

```
procedure AuthenticationOverEC.curveFromAlgoImplemByteId(algoImplemByteId)
    if algoImplemByteId = 0x00 then return MDC() end if
    if algoImplemByteId = 0x01 then return Curve25519() end if
    return $\perp$
end procedure
```

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

We denote by

$$\mathsf{AuthenticationOverEC.algoImplemByteIdFromCurve(curve)} \rightarrow \texttt{algoImplemByteId}$$

the call to the AuthenticationOverEC.algoImplemByteIdFromCurve procedure.

```
procedure AuthenticationOverEC.algoImplemByteIdFromCurve(curve)
    if curve = MDC() then return 0x00 end if
    if curve = Curve25519() then return 0x01 end if
    return $\perp$
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

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

the call to the AuthenticationPublicKeyOverEC initializer.

```
procedure AuthenticationPublicKeyOverEC(curve, $P$)
    algoClassByteId $\leftarrow$ 0x14
    algoImplemByteId $\leftarrow$ 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	curve
	$[0, \dots, p-1]$	y

We denote by

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

the call to the AuthenticationPublicKeyOverEC initializer.

```
1: procedure AuthenticationPublicKeyOverEC(curve, $y$)
2:     algoClassByteId $\leftarrow$ 0x14
3:     algoImplemByteId $\leftarrow$ algoImplemByteIdFromCurve(curve)
4:     PublicKeyOverEC(algoClassByteId, algoImplemByteId, curve, $y$)
5: end procedure
```

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

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

We denote by

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

the call to the AuthenticationPrivateKeyOverEC initializer.

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

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

We denote by

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

the call to the AuthenticationOverEC.generateKeyPair procedure.

```
procedure AuthenticationOverEC.generateKeyPair(prng, curve)
    $(\lambda, P) \leftarrow$ curve.generateRandomScalarAndPoint(prng)
    pk $\leftarrow$ AuthenticationPublicKeyOverEC(curve, $P$)
    sk $\leftarrow$ AuthenticationPrivateKeyOverEC(curve, $\lambda$)
    return (pk, sk)
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

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

the call to the pk.convertToSignatureKey procedure.

```
procedure pk.convertToSignatureKey()
    if pk.point $\neq \perp$ then
        return SignaturePublicKeyOverEC(pk.curve, pk.point)
    else
        return SignaturePublicKeyOverEC(pk.curve, pk.$y$)
    end if
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

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

the call to the sk.convertToSignatureKey procedure.

```
procedure sk.convertToSignatureKey()
    return SignaturePrivateKeyOverEC(sk.curve, sk.$\lambda$)
end procedure
```

AuthenticationOverEC.solve		
parameters		None
in	AuthenticationPrivateKeyOverEC	sk
	$\{0,\dots,255\}^*$	`challenge`
	$\{0,\dots,255\}^*$	`prefix`
	AuthenticationPublicKeyOverEC	pk
	PRNG	prng
out	$\{0,\dots,255\}^*$	`response`

We denote by

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

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

```
procedure AuthenticationOverEC.solve(sk, challenge, prefix, pk, prng)
    if sk.curve $\neq$ pk.curve then return $\perp$ end if
    suffix $\leftarrow$ prng.bytes(16)
    formatted_challenge $\leftarrow$ prefix||challenge||suffix
    $pk_\sigma$ $\leftarrow$ pk.convertToSignatureKey()
    $sk_\sigma$ $\leftarrow$ sk.convertToSignatureKey()
    $\sigma$ $\leftarrow$ SignatureOverEC.sign($sk_\sigma$, formatted_challenge, $pk_\sigma$, prng)
    return suffix||$\sigma$
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 $\perp$
3:     end if
4:     Parse response as suffix$\|\sigma$ where len(suffix) $= 16$
5:     formatted_challenge $\leftarrow$ prefix$\|$challenge$\|$suffix
6:     $\mathsf{pk}_\sigma \leftarrow$ pk.convertToSignatureKey()
7:     return SignatureOverEC.verify($\mathsf{pk}_\sigma$, formatted_challenge, response)
8: end procedure
```

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

We denote by

$$\mathsf{AuthenticationPublicKeyOverEC.expandCompactKey}(\texttt{compactKey}) \to \mathsf{AuthenticationPublicKeyOverEC}$$

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 $\perp$ end if
3:     Parse compactKey as algoImplemByteId$\|$yCoordinate where len(algoImplemByteId) $= 1$
```

```
4:     curve ← AuthenticationPublicKeyOverEC.curveFromAlgoImplemByteId(algoImplemByteId)
5:     if $\mathsf{len}(\texttt{compactKey}) \neq 1 + \mathsf{len}(\mathsf{curve}.p)$ 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 that allows 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

$$\mathsf{AuthenticationOverCurve25519.generateKeyPair}(\mathsf{prng}) \to (\mathsf{pk}, \mathsf{sk})$$

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 that allows 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

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

the call to the AuthenticationOverMDC.generateKeyPair procedure.

```
procedure AuthenticationOverMDC.generateKeyPair(prng)
    curve $\leftarrow$ MDC()
    return AuthenticationOverEC.generateKeyPair(prng, curve)
end procedure
```

16 Key Encapsulation Mechanism (KEM)

Key Encapsulation Mechanism (KEM) are instances of a complex type denoted KEMOverEC. Public keys used for decrypting a encapsulated 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 encapsulating a 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.

```
procedure KEMOverEC.curveFromAlgoImplemByteId(algoImplemByteId)
    if algoImplemByteId = 0x00 then return MDC() end if
    if algoImplemByteId = 0x01 then return Curve25519() end if
    return $\perp$
end procedure
```

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

We denote by

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

the call to the KEMOverEC.algoImplemByteIdFromCurve procedure.

```
procedure KEMOverEC.algoImplemByteIdFromCurve(curve)
    if curve = MDC() then return 0x00 end if
    if curve = Curve25519() then return 0x01 end if
    return $\perp$
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	curve
	$[0, \ldots, p-1]^2$	P

We denote by

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

the call to the KEMPublicKeyOverEC initializer.

```
procedure KEMPublicKeyOverEC(curve, $P$)
    algoClassByteId ← 0x12
    algoImplemByteId ← algoImplemByteIdFromCurve(curve)
    PublicKeyOverEC(algoClassByteId, algoImplemByteId, curve, $P$)
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	curve
	$[0, \ldots, p-1]$	y

We denote by

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

the call to the KEMPublicKeyOverEC initializer.

```
procedure KEMPublicKeyOverEC(curve, $y$)
    algoClassByteId $\leftarrow$ 0x12
    algoImplemByteId $\leftarrow$ algoImplemByteIdFromCurve(curve)
    PublicKeyOverEC(algoClassByteId, algoImplemByteId, curve, $y$)
end procedure
```

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

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

We denote by

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

the call to the KEMPrivateKeyOverEC initializer.

```
procedure KEMPrivateKeyOverEC(curve, $\lambda$)
    algoClassByteId $\leftarrow$ 0x12
    algoImplemByteId $\leftarrow$ algoImplemByteIdFromCurve(curve)
    PrivateKeyOverEC(algoClassByteId, algoImplemByteId, curve, $\lambda$)
end procedure
```

KEMOverEC.generateKeyPair		
parameters		None
in	PRNG	prng
	EdwardsCurve	curve
out	KEMPublicKeyOverEC	pk
	KEMPrivateKeyOverEC	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)
    $(\lambda, P) \leftarrow$ curve.generateRandomScalarAndPoint(prng)
    pk $\leftarrow$ KEMPublicKeyOverEC(curve, $P$)
    sk $\leftarrow$ KEMPrivateKeyOverEC(curve, $\lambda$)
    return (pk, sk)
end procedure
```

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

```
1:  procedure KEMOverEC.encrypt(pk, prng, T)
2:      curve ← pk.curve
3:      $(p, d, G, q, \nu, \mathsf{card})$ ← curve.parameters
4:      repeat $r$ ← prng.bigInt($q$) until $r \neq 0$
5:      $B_y$ ← curve.scalarMultiplication($r$, $G.y$)
6:      $D_y$ ← curve.scalarMultiplication($r$, pk.$y$)
7:      c ← bytesFromBigUInt($B_y$, len($p$))
8:      seed ← c||bytesFromBigUInt($D_y$, len($p$))
9:      key ← KDFFromPRNGWithHMACWithSHA256.compute(seed, T)
10:     return (c, k)
11: end procedure
```

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

We denote by

$$\mathsf{KEMOverEC.decrypt}(\texttt{encrypted\_key}, \mathsf{sk}, \mathsf{T}) \rightarrow \texttt{key}$$

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

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

We denote by

KEMPublicKeyOverEC.expandCompactKey(compactKey) $\rightarrow$ 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)
    if len(compactKey) $\neq 1 +$ len(curve.$p$) then return $\perp$ end if
    $y \leftarrow$ bigUIntFromBytes(yCoordinate)
    return KEMPublicKeyOverEC(curve, $y$)
end procedure
```

16.1 KEM over Curve25519

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

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

We denote by

$$\mathsf{KEMOverCurve25519.generateKeyPair}(\mathsf{prng}) \to (\mathsf{pk}, \mathsf{sk})$$

the call to the KEMOverCurve25519.generateKeyPair procedure.

```
1: procedure KEMOverCurve25519.generateKeyPair(prng)
2:     curve ← Curve25519()
3:     return KEMOverEC.generateKeyPair(prng, curve)
4: end procedure
```

16.2 KEM over MDC

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

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

We denote by

$$\mathsf{KEMOverMDC.generateKeyPair}(\mathsf{prng}) \to (\mathsf{pk}, \mathsf{sk})$$

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 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 an instance of AuthenticationPublicKeyOverEC and allows to check challenge responses computed by the owner of the associated private key (i.e., of the owner of the corresponding owned cryptographic identity, see Section 18).
- cryptoIdentity.publicKeyForKEM: This public key is an instance of KEMPublicKeyOverEC and allows to perform a KEM encrypt that shall only be decrypted by the owner of the associated private key (i.e., of the owner of the corresponding owned cryptographic identity, see Section 18).

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

$$\mathsf{CryptoIdentity}(\texttt{serverURL}, \mathsf{pk}_a, \mathsf{pk}_e) \to \mathsf{cryptoIdentity}$$

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

```
procedure CryptoIdentity(serverURL, $\mathsf{pk}_a$, $\mathsf{pk}_e$)
    self.serverURL ← serverURL
    self.publicKeyForAuthentication ← $\mathsf{pk}_a$
    self.publicKeyForKEM ← $\mathsf{pk}_e$
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

$$\mathsf{cryptoIdentity.getIdentity}() \to \texttt{identity}$$

the call to the cryptoIdentity.getIdentity procedure.

```
procedure cryptoIdentity.getIdentity()
    identity ← serverURL
    identity ← identity || 0x00
    identity ← identity || self.publicKeyForAuthentication.getCompactKey()
    identity ← identity || self.publicKeyForKEM.getCompactKey()
    return identity
end procedure
```

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

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

$$\mathsf{CryptoIdentity}(\texttt{identity}) \to \mathsf{cryptoIdentity}$$

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

```
procedure CryptoIdentity(identity)
    Parse identity as serverURL || 0x00 || keys. Return ⊥ if this fails.
    Check that serverURL is a valid URL
    if len(keys) = 0 then return ⊥ end if
    authImplemByteId ← keys[0]  // one byte
    $\mathsf{curve}_a$ ← AuthenticationOverEC.curveFromAlgoImplemByteId(authImplemByteId)
    $\ell_a \leftarrow \mathsf{len}(\mathsf{curve}_a.p)$
    if len(keys) $< 2 + \ell_a$ then return ⊥ end if
    compactAuthKey ← keys$[1 \dots \ell_a]$  // $\ell_a$ bytes
    $\mathsf{pk}_a$ ← AuthenticationPublicKeyOverEC.expandCompactKey(compactAuthKey)
    kemImplemByteId ← keys$[\ell_a + 1]$  // One byte
    $\mathsf{curve}_e$ ← KEMPublicKeyOverEC.curveFromAlgoImplemByteId(kemImplemByteId)
    $\ell_e \leftarrow \mathsf{len}(\mathsf{curve}_e.p)$
    if len(keys) $\neq 2 + \ell_a + \ell_e$ then return ⊥ end if
    compactKEMKey ← keys$[\ell_a + 2 \dots \ell_a + 2 + \ell_e - 1]$  // $\ell_e$ bytes
    $\mathsf{pk}_e$ ← KEMPublicKeyOverEC.expandCompactKey(compactKEMKey)
    self.serverURL ← serverURL
    self.publicKeyForAuthentication ← $\mathsf{pk}_a$
    self.publicKeyForKEM ← $\mathsf{pk}_e$
end procedure
```

18 Owned Cryptographic Identity

A cryptographic identity is necessarily *owned* by a user. In that case, this user knows about the private keys associated to the public keys presented in Section 17. The type OwnedCryptoIdentity allows to represent these owned identities. An instance ownedCryptoIdentity of OwnedCryptoIdentity gives access to three values:

- ownedCryptoIdentity.`serverURL`: All messages sent to an identity pass through a server, identified by this URL.
- ownedCryptoIdentity.publicKeyForAuthentication: This public key is an instance of AuthenticationPublicKeyOverEC and allows to check challenge responses computed by the owner of the associated private key (i.e., of the owner of the corresponding owned cryptographic identity, see Section 18).
- 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 allows to perform a KEM encrypt that shall only be decrypted by the owner of the associated private key (i.e., of the owner of the corresponding owned cryptographic identity, see Section 18).
- 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	$\{0, \ldots, 255\}^*$	`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

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

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

```
procedure OwnedCryptoIdentity(serverURL, $\mathsf{pk}_a$, $\mathsf{sk}_a$, $\mathsf{pk}_e$, $\mathsf{sk}_e$, key)
    self.serverURL $\leftarrow$ serverURL
    self.publicKeyForAuthentication $\leftarrow \mathsf{pk}_a$
    self.privateKeyForAuthentication $\leftarrow \mathsf{sk}_a$
    self.publicKeyForKEM $\leftarrow \mathsf{pk}_e$
    self.privateKeyForKEM $\leftarrow \mathsf{sk}_e$
    self.secretMACKey $\leftarrow$ key
end procedure
```

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

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

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

the call to the previous procedure.

```
1: procedure OwnedCryptoIdentity.generateOwnedCryptoIdentity(serverURL, prng)
2:     $(\mathsf{pk}_a, \mathsf{sk}_a) \leftarrow$ AuthenticationOverMDC.generateKeyPair(prng)
3:     $(\mathsf{pk}_e, \mathsf{sk}_e) \leftarrow$ KEMOverCurve25519.generateKeyPair(prng)
4:     key $\leftarrow$ HMACWithSHA256.generateKey(prng)
5:     return OwnedCryptoIdentity(serverURL, $\mathsf{pk}_a, \mathsf{sk}_a, \mathsf{pk}_e, \mathsf{sk}_e$, key)
6: end procedure
```

ownedCryptoIdentity.getCryptoIdentity		
parameters		None
in		
out	CryptoIdentity	cryptoIdentity

Given an owned identity instance ownedCryptoIdentity, the previous procedure allows to extract the public informations by returning an instance cryptoIdentity of CryptoIdentity. We denote by

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

the call to the previous procedure.

```
1: procedure ownedCryptoIdentity.getCryptoIdentity()
2:     return CryptoIdentity(self.serverURL, self.$\mathsf{pk}_a$, self.$\mathsf{pk}_e$)
3: end procedure
```

Part III
Encodings

Several features of the Olvid engine require the serialization of data and objects as bytes. For example, 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 developped our own binary-friendly encoding format which is described in this section.

19 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 ASN1 or BSON.

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

19.1 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 1.

Table 1: List of all identifiers and corresponding number of bytes for determining the content length.

Identifier	Type	Section
`0x00`	Array of bytes	21.2
`0x01`	64-bit Integer	21.4
`0x02`	Boolean	21.5
`0x80`	Unsigned Big Integer	21.6
`0x03`	List	21.7
`0x04`	Dictionary	21.8
`0x90`	Symmetric key	21.10
`0x91`	Public key	21.11
`0x92`	Private key	21.12

19.2 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 20.1 describes the algorithm used to compute this representation.

20 Byte Representation of Integers and Lengths

We first define a few helpers procedures, allowing 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.

20.1 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,

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

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

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

20.2 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,

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

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

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

20.3 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,

$$\mathsf{bytesFromInt64}(s) = \begin{cases} \mathsf{bytesFromUInt64}(s) & \text{if } s \in [0, 2^{63}-1], \\ \mathsf{bytesFromUInt64}(2^{32}+s) & \text{if } s \in [-2^{63}, -1], \\ \perp & \text{otherwise.} \end{cases}$$

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

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

20.4 Unsigned Big Integer

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

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

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

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

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

20.5 Signed Big Integer

Given a signed integer s, we denote by $\mathsf{bytesFromBigInt}(s,\ell) \in \{0,\dots,255\}^\ell$ 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 a ℓ-byte array $\mathtt{b} = [\mathtt{b}_0, \mathtt{b}_1, \dots, \mathtt{b}_{\ell-1}] \in \{0,\dots,255\}^\ell$, we have

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

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

20.6 Lengths

As stated in Section 19.2, 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 $\mathtt{b} = [\mathtt{b}_0, \mathtt{b}_1, \mathtt{b}_2, \mathtt{b}_3] \in \{0,\dots,255\}^4$ is mapped to a 32-bit unsigned integer length using $\mathsf{uint32FromBytes}(\mathtt{b})$.

21 Encodings Procedures

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

21.1 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 21.2 to 21.12 execute the following parsing procedure on an input $[\mathtt{b}_0, \mathtt{b}_1, \ldots, \mathtt{b}_n]$ before trying to decode any further:

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

Note that all the known byte identifiers are listed in Table 1.

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

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

21.3 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.

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

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

21.4 Encoding a 64-bit Integer

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

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

We use the identifier `0x01` for 64-bit integers. The length of 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 $[\mathtt{b}_0, \mathtt{b}_1, \ldots, \mathtt{b}_{n-1}] \in \{0, \ldots, 255\}^n$ assumed to be the encoding of a 64-bit integer, we apply the following procedure:

```
1: procedure decodeInt($[\mathtt{b}_0, \mathtt{b}_1, \ldots, \mathtt{b}_{n-1}]$)
2:     $(\mathtt{byteId}, [\mathtt{c}_0, \mathtt{c}_1, \ldots, \mathtt{c}_{\ell-1}]) \leftarrow \mathsf{Parse}([\mathtt{b}_0, \mathtt{b}_1, \ldots, \mathtt{b}_{n-1}])$
3:     if $\mathtt{byteId} \neq \mathtt{0x01}$ then return $\perp$ end if
4:     if $\ell \neq 8$ then return $\perp$ end if
5:     return $[\mathtt{c}_0, \mathtt{c}_1, \ldots, \mathtt{c}_7]$
6: end procedure
```

21.5 Encoding a Boolean

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

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

We use the identifier `0x02` for Booleans. The length of 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 $[\mathtt{b}_0, \mathtt{b}_1, \ldots, \mathtt{b}_{n-1}] \in \{0, \ldots, 255\}^n$ assumed to be the encoding of a Boolean, we apply the following procedure:

```
1: procedure decodeBool($[\mathtt{b}_0, \mathtt{b}_1, \ldots, \mathtt{b}_{n-1}]$)
2:     $(\mathtt{byteId}, [\mathtt{c}_0, \mathtt{c}_1, \ldots, \mathtt{c}_{\ell-1}]) \leftarrow \mathsf{Parse}([\mathtt{b}_0, \mathtt{b}_1, \ldots, \mathtt{b}_{n-1}])$
3:     if $\mathtt{byteId} \neq \mathtt{0x02}$ then return $\perp$ end if
4:     if $\ell \neq 1$ then return $\perp$ end if
5:     return $\mathtt{c}_0 == 1$
6: end procedure
```

21.6 Encoding an Unsigned Big Integer

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

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

Note that encodeBigUInt returns $\perp$ if any internal call returns $\perp$. We use the identifier `0x80` for unsigned big integers. The length of 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 $[\mathtt{b}_0, \mathtt{b}_1, \dots, \mathtt{b}_{n-1}] \in \{0, \dots, 255\}^n$ assumed to be the encoding of an unsigned big integer, we apply the following procedure:

```
1: procedure decodeBigUInt($[\mathtt{b}_0, \mathtt{b}_1, \dots, \mathtt{b}_{n-1}]$)
2:     $(\mathtt{byteId}, [\mathtt{c}_0, \dots, \mathtt{c}_{\ell-1}]) \leftarrow \mathsf{Parse}([\mathtt{b}_0, \mathtt{b}_1, \dots, \mathtt{b}_{n-1}])$
3:     if $\mathtt{byteId} \neq \mathtt{0x80}$ then return $\perp$ end if
4:     return $\mathsf{bigUIntFromBytes}([\mathtt{c}_0, \dots, \mathtt{c}_{\ell-1}])$
5: end procedure
```

21.7 Packing Elements and Encoding a List

The following pack procedure allows to packs 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, $\dots \in \{0, \dots, 255\}^*$, the pack procedure is defined as follows:

$$\begin{aligned} &\mathsf{pack}(\mathtt{byteId}, \mathtt{encoded\_val1}, \mathtt{encoded\_val2}, \dots) \\ &\qquad \rightarrow [\mathtt{byteId}, \mathsf{uint32FromBytes}(\ell), \mathtt{encoded\_val1}, \mathtt{encoded\_val2}, \dots] \end{aligned} \tag{2}$$

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

$$\ell = \mathsf{len}(\mathtt{encoded\_val1}) + \mathsf{len}(\mathtt{encoded\_val2}) + \cdots$$

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

```
1: procedure unpack($[\mathtt{b}_0, \mathtt{b}_1, \dots, \mathtt{b}_{n-1}]$)
2:     $(\mathtt{byteId}, \mathtt{innerData}) \leftarrow \mathsf{Parse}([\mathtt{b}_0, \mathtt{b}_1, \dots, \mathtt{b}_{n-1}])$
3:     $\mathtt{encoded\_vals} = []$
4:     while $\mathsf{len}(\mathtt{innerData}) > 0$ do
5:         $(\mathtt{encoded\_val}, \mathtt{innerData}) \leftarrow \mathsf{extractFirstEncodedValue}(\mathtt{innerData})$
6:         Append encoded_val to encoded_vals
7:     end while
8:     return $(\mathtt{byteId}, \mathtt{encoded\_vals})$
9: end procedure
```

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

```
1: procedure extractFirstEncodedValue($[\mathtt{b}_0, \mathtt{b}_1, \dots, \mathtt{b}_{n-1}]$)
2:     if $n < 5$ then return $\perp$ end if
3:     $\ell \leftarrow \mathsf{uint32FromBytes}([\mathtt{b}_1, \mathtt{b}_2, \mathtt{b}_3, \mathtt{b}_4])$
4:     if $n < 5 + \ell$ then return $\perp$ end if
5:     return $([\mathtt{b}_0, \dots, \mathtt{b}_{5+(\ell-1)}], [\mathtt{b}_{5+\ell}, \dots, \mathtt{b}_{n-1}])$
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

$\texttt{encoded\_val1}, \texttt{encoded\_val2}, \ldots \in \{0, \ldots, 255\}^*$:

$$\mathsf{encodeList}(\texttt{encoded\_val1}, \texttt{encoded\_val2}, \ldots) \rightarrow \mathsf{pack}(\texttt{0x03}, \texttt{encoded\_val1}, \texttt{encoded\_val2}, \ldots) \quad (3)$$

We use the identifier 0x03 for lists.

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 list, we apply the following procedure:

```
1: procedure decodeList($[\mathtt{b}_0, \mathtt{b}_1, \ldots, \mathtt{b}_{n-1}]$)
2:     $(\texttt{byteId}, \texttt{encoded\_vals}) \leftarrow \mathsf{unpack}([\mathtt{b}_0, \mathtt{b}_1, \ldots, \mathtt{b}_{n-1}])$
3:     if $\texttt{byteId} \neq \texttt{0x03}$ then return $\perp$ end if
4:     return encoded_vals
5: end procedure
```

Padded encoded lists. Sometimes, in order to hide the length of an encoded content, we might want to pad it to a fixed length. In that case, the decoding procedure should be able to handle an input with additional bytes at the end that can be safely ignored. This is possible with encoded lists using a special decoding procedure.

To encode a padded list, simply encode the list using the standard encodeList procedure and append some bytes to the output. However, to decode it we must use a special decodeListWithPadding procedure that accepts a length mismatch between the input and the innerData length.

```
1: procedure decodeListWithPadding($[\mathtt{b}_0, \mathtt{b}_1, \ldots, \mathtt{b}_{n-1}]$)
2:     $\ell \leftarrow \mathsf{uint32FromBytes}([\mathtt{b}_1, \mathtt{b}_2, \mathtt{b}_3, \mathtt{b}_4])$
3:     if $\ell > n - 5$ then return $\perp$ end if
4:     $(\texttt{byteId}, \texttt{encoded\_vals}) \leftarrow \mathsf{unpack}([\mathtt{b}_0, \mathtt{b}_1, \ldots, \mathtt{b}_{5+(\ell-1)}])$
5:     if $\texttt{byteId} \neq \texttt{0x03}$ then return $\perp$ end if
6:     return encoded_vals
7: end procedure
```

21.8 Encoding a Dictionary

In most programming languages, a dictionary is an associative array where each element is associated to 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 $\texttt{encoded\_val} \in \{0, \ldots, 255\}^*$ to the key $\texttt{key} \in \{0, \ldots, 255\}^*$ is denoted

$$\mathsf{dict}[\texttt{key}] \leftarrow \texttt{encoded\_val}.$$

Recovering the encoded value encoded_val is denoted

$$\mathsf{dict}[\texttt{key}] \rightarrow \texttt{encoded\_val} \quad \text{or} \quad \texttt{encoded\_val} \leftarrow \mathsf{dict}[\texttt{key}].$$

The following procedure returns a proper encoding of a dictionary dict respectively associating the keys $\texttt{key1}, \texttt{key2}, \ldots$ with the encoded values $\texttt{encoded\_val1}, \texttt{encoded\_val2}, \ldots$:

$$\mathsf{encodeDictionary}(\mathsf{dict}) \rightarrow \mathsf{pack}(\texttt{0x04}, \mathsf{encodeBytes}(\texttt{key1}), \texttt{encoded\_val1}, \mathsf{encodeBytes}(\texttt{key2}), \texttt{encoded\_val2}, \ldots) \quad (4)$$

In most programming languages, dictionaries use hash tables and the order of the key 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:

```
procedure decodeDictionary($[b_0, b_1, \ldots, b_{n-1}]$)
    (byteId, encoded_vals) $\leftarrow$ unpack($[b_0, b_1, \ldots, b_{n-1}]$)
    if byteId $\neq$ 0x04 then return $\perp$ end if
    if len(encoded_vals) is not even then return $\perp$ end if
    dict $\leftarrow$ empty dictionary
    while len(encoded_vals) $> 0$ do
        (encoded_key, val) $\leftarrow$ pop the first two elements of encoded_vals
        key $\leftarrow$ decodeBytes(encoded_key)
        dict[key] $\leftarrow$ val
    end while
    return dict
end procedure
```

21.9 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 allowing 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:

```
procedure encodeKey(key)
    encoded_byteIds $\leftarrow$ encodeList([key.algoClassByteId, key.algoImplemByteId])
    encoded_dict $\leftarrow$ encodeDictionary(key.dict)
    return pack(key.encodingByteId, encoded_byteIds, encoded_dict)
end procedure
```

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 cryptographic key for a specific KeyedAlgo, the following procedure will be systematically used within the decoding of a cryptographic key:

```
procedure preDecodeKey($[b_0, b_1, \ldots, b_{n-1}]$)
    (byteId, encoded_vals) $\leftarrow$ unpack($[b_0, b_1, \ldots, b_{n-1}]$)
    if $\mathsf{len}(\texttt{encoded\_vals}) \neq 2$ then return $\perp$ end if
    $\texttt{byteIds} \leftarrow \mathsf{decodeList}(\texttt{encoded\_vals}[0])$
    if $\mathsf{len}(\texttt{byteIds}) \neq 2$ then return $\perp$ end if
    $\texttt{algoClassByteId} \leftarrow \texttt{byteIds}[0]$
    $\texttt{algoImplemByteId} \leftarrow \texttt{byteIds}[1]$
    $\texttt{dict} \leftarrow \mathsf{decodeDictionary}(\texttt{encoded\_vals}[1])$
    return $(\texttt{algoClassByteId}, \texttt{algoImplemByteId}, \texttt{dict}, \texttt{byteId})$
end procedure
```

21.10 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 $\mathsf{symKey.encodingByteId} = \texttt{0x90}$ 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 $[\texttt{b}_0, \texttt{b}_1, \ldots, \texttt{b}_{n-1}] \in \{0, \ldots, 255\}^n$ assumed to be the encoding of a symmetric key for a specific symmetric key algorithm SymKeyedAlgo, we apply the following procedure:

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

21.11 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:

```
procedure decodeSymKey($[\mathtt{b}_0, \mathtt{b}_1, \ldots, \mathtt{b}_{n-1}]$)
    $(\mathtt{algoClassByteId}, \mathtt{algoImplemByteId}, \mathtt{dict}, \mathtt{byteId}) \leftarrow \mathsf{preDecodeKey}([\mathtt{b}_0, \mathtt{b}_1, \ldots, \mathtt{b}_{n-1}])$
    if $\mathtt{byteId} \neq \mathtt{0x91}$ then return $\perp$ end if
    return PubKeyedAlgo.PubKeyInit(algoClassByteId, algoImplemByteId, dict)
end procedure
```

21.12 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 $\mathtt{privKey.encodingByteId} = \mathtt{0x92}$ 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 $[\mathtt{b}_0, \mathtt{b}_1, \ldots, \mathtt{b}_{n-1}] \in \{0, \ldots, 255\}^n$ assumed to be the encoding of a private key for a specific public key cryptographic algorithm PubKeyedAlgo (see Section 21.11), we apply the following procedure:

```
procedure decodeSymKey($[\mathtt{b}_0, \mathtt{b}_1, \ldots, \mathtt{b}_{n-1}]$)
    $(\mathtt{algoClassByteId}, \mathtt{algoImplemByteId}, \mathtt{dict}, \mathtt{byteId}) \leftarrow \mathsf{preDecodeKey}([\mathtt{b}_0, \mathtt{b}_1, \ldots, \mathtt{b}_{n-1}])$
    if $\mathtt{byteId} \neq \mathtt{0x92}$ then return $\perp$ end if
    return PubKeyedAlgo.PrivKeyInit(algoClassByteId, algoImplemByteId, dict)
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 user 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 channel is described in the channel creation protocol of Section 26), using symmetric encryption with 1-time keys (see Section 23.1 for encryption details). A Pre-Key Channel uses a pre-key, uploaded to the server by the recipient's device, for encryption (see Section 24. 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).

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 exemples 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 message 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`.
- an Pre-Key Channel, either in unicast to one specific device, or in multicast to multiple devices of an `identity`.

22 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 100 bytes), 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 to 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 in chunks before being sent. See Section 23.3 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.

22.1 Protocol Message Structure

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

- an integer identifying the protocol Id (see Table 2),
- 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 Id),
- 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.

22.2 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
  - an encoded String corresponding to the serialized attachment metadata JSON (see below).
- an encoded serialized message payload JSON (see below).

Attachment metadata JSON:
```
{
  "type": String,   // attachment MIME type
  "fileName": 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>
}
```
`rr` and `message` are present for text messages. `rtc` is set for WebRTC signaling messages. `settings` message allow to set 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 catpture of a limited visibility message is detected. `lvo` and `dr` are multi-device synchronisation messages used to keep opened/read messages in sync on all devices.

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>]
}
```
`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 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 sharing messages. `um` is an array of user mention object which refer to a range of the message body corresponding to a user mention.

One-to-one discussion identifier:
```
[
  byte[],
  byte[]
]
```
For one-to-one discussions, this is an array containing the `identity` of both sender and receiver, allowing 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, or remote deletion.

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
  "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         // location sharing expiration
}
```
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.

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.

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 make sure the version and known settings match. If not, a shared settings update message may be sent in return.

Update message 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 message, this is used to update a SHARING type message or to notify the end of the sharing with a END_SHARING update.

Messages defete 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 defete 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.

Update message request:
```
{
  "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>
}
```
This message adds a reaction (typically an emoji) to a message referenced by a discussion and a message reference. The message structure itself does not impose any contraint on the **reac** string nature or length.

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>,
  "ref": <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 being able to view read once messages once on each device!

Discussion read:
```
{
  "tim": int,       // timestamp of last message
  "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 sent it the server timestamp of the most recent message read in the discussion.

23 Encryption

23.1 Message Encryption

The message itself is always encrypted with an authenticated encryption symmetric primitive. The authenticated encryption message key used to be generated at random but now uses the gkmv2 method described below. Once generated, ths 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 decription, the decoding method for padded encoded lists 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:     $\texttt{seed}_{enc} \leftarrow \mathsf{prng.bytes}(32)$
3:     $\texttt{key}_{enc} \leftarrow \mathsf{KDFFromPRNGWithHMACWithSHA256.compute}(\texttt{seed}_{enc}, \mathsf{AES256CTRKey})$
4:     $\texttt{seed}_{mac} \leftarrow \texttt{key}_{enc}.\mathsf{bytes} \,\|\, \mathsf{paddedPlaintext}$
5:     $\texttt{key}_{mac} \leftarrow \mathsf{KDFFromPRNGWithHMACWithSHA256.compute}(\texttt{seed}_{mac}, \mathsf{HMACWithSHA256Key})$
6:     return $\mathsf{AES256CTRHMACSHA256Key}(\texttt{key}_{enc}, \texttt{key}_{mac})$
7: end procedure
```

When receiving a message, the message key is verified by checking the $\texttt{key}_{mac}$ matches what is expected.

Asymmetric channel. This uses the recipient's `identity` encryption public key to wrap the message key. Wrapping here is a simple KEM/DEM. The header contains the concatenation of:

- a KEM ciphertext (32 bytes),
- the DEM of the encoded message key.

Pre-key channel. As details in Section 24, pre-keys are short-lived encryption public keys that can be used to exchange messages while the oblivious channel is being created. Contrary to asymetric encryption that 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, remoteId, remoteDevUid, prng
    encodedPayload ← encodeList(encodeSymKey(msgKey),
                                 encodeBytes(ownDevUid), encodeBytes(ownId))
    signaturePayload ← encodeList(encodedPayload, encodeBytes(remoteDevUid),
                                 encodeBytes(remoteId), encodeBytes(preKey.id))
    signature ← AuthenticationPublicKeyOverEC.solve(
                                ownedCryptoIdentity.privateKeyForAuthentication,
                                signaturePayload,
                                "encryptionWithPreKey",
                                ownedCryptoIdentity.publicKeyForAuthentication,
                                prng)
    encodedPlaintext ← encodeList(encodedPayload, encodeBytes(signature)))
    return preKey.id || KemDem.encrypt(preKey.publicKey, encodedPlaintext, prng)
end procedure
```

Oblivious channel. Oblivious channels use a self ratchet system (in combination with the full ratchet described in Section 33) which allows to generate 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. The header contains the concatenation of:

- a key Id (32 bytes),
- the authenticated encryption of the encoded message key.

For backward compatibility, each oblivious channel is created as a normal channel able to receive messages that either use a random message key or a gkmv2 message key. However, as soon as a message with a valid gkmv2 key is received, the channel is tagged as "gkmv2-compatible" and any subsequent message must have a valid gkmv2 key or is discarded.

23.2 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:

```
procedure DeriveExtendedPayloadKey(msgKey)
    prng ← PRNG(msgKey)
    seed ← prng.bytes(32)
    return AES256CTRHMACSHA256.generateKey(seed)
end procedure
```

23.3 Attachment Encryption

Each attachment is encrypted using a random authenticated encryption key (sent with the message). Attachments are split in chunks of size determined by the sender (current implementations always use chunks of 8MB). 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.

23.4 Return Receipt Encryption

Return receipt encryption is very similar to message encryption on an Oblivious channel. Each received application message contains a nonce and authenticated encryption key. The authenticated encryption key allows to mask the `identity` of the sender return receipt as well as its nature (received or read), and the nonce allows the message sender (return receipt recipient) to identify which key to use for decryption.

24 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-bytes 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 an 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 negociated 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, has is an encoded list containing:

- an encoded dictionnary containing:
  - for key `"prk"`, the encoded pre-key which is an encoded list of:
    * the encoded 32-bytes pre-key id
    * the encoded encryption public key
    * 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 prvious encoded, using the pre-key owner's server authentication key and the prefix `"devicePreKey"`

Pre-keys lifecycle. Currently, pre-keys have a lifespan of 60 days and are renewed every 7 days. Therefor, 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 fesh 60 days 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 database for another 60 days in case a message sent juste 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 week or so 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 2. The protocol ID and message ID of a message are serialized alongside the message payload.

ID	Protocol name
0	Device Discovery Protocol (see Section 27)
2	Channel Creation with Contact Device Protocol (see Section 26)
3	Device Discovery Child Protocol (see Section 27)
4	Contact Mutual Introduction Protocol (see Section 28)
6	Identity Details Publication Protocol (see Section 29)
7	Contact Picture Download Child Protocol (see Section 29)
8	Group Invitation Protocol (see Section 30)
9	Group Management Protocol (see Section 31)
10	Oblivious Channel Management Protocol (see Section 32)
11	Trust Establishment Protocol with SAS (see Section 25)
12	Trust Establishment Protocol with Mutual Scan (coming soon)
13	Full Ratchet Protocol (see Section 33)
14	Group Picture Download Child Protocol (see Section 31)

Table 2: List of protocols implemented in Olvid and their corresponding protocol ID.

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. The `protocolUid` should be kept secret and only shared between legitimate parties to the protocol.

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"
- 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 update in the database

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.

25 Trust Establishment Protocol with SAS

25.1 Purpose and High Level View

The Trust Establishment Protocol with SAS represented on 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.

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 ensured to know about their 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}.

25.2 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 randomness sent by Alice. This meens 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.

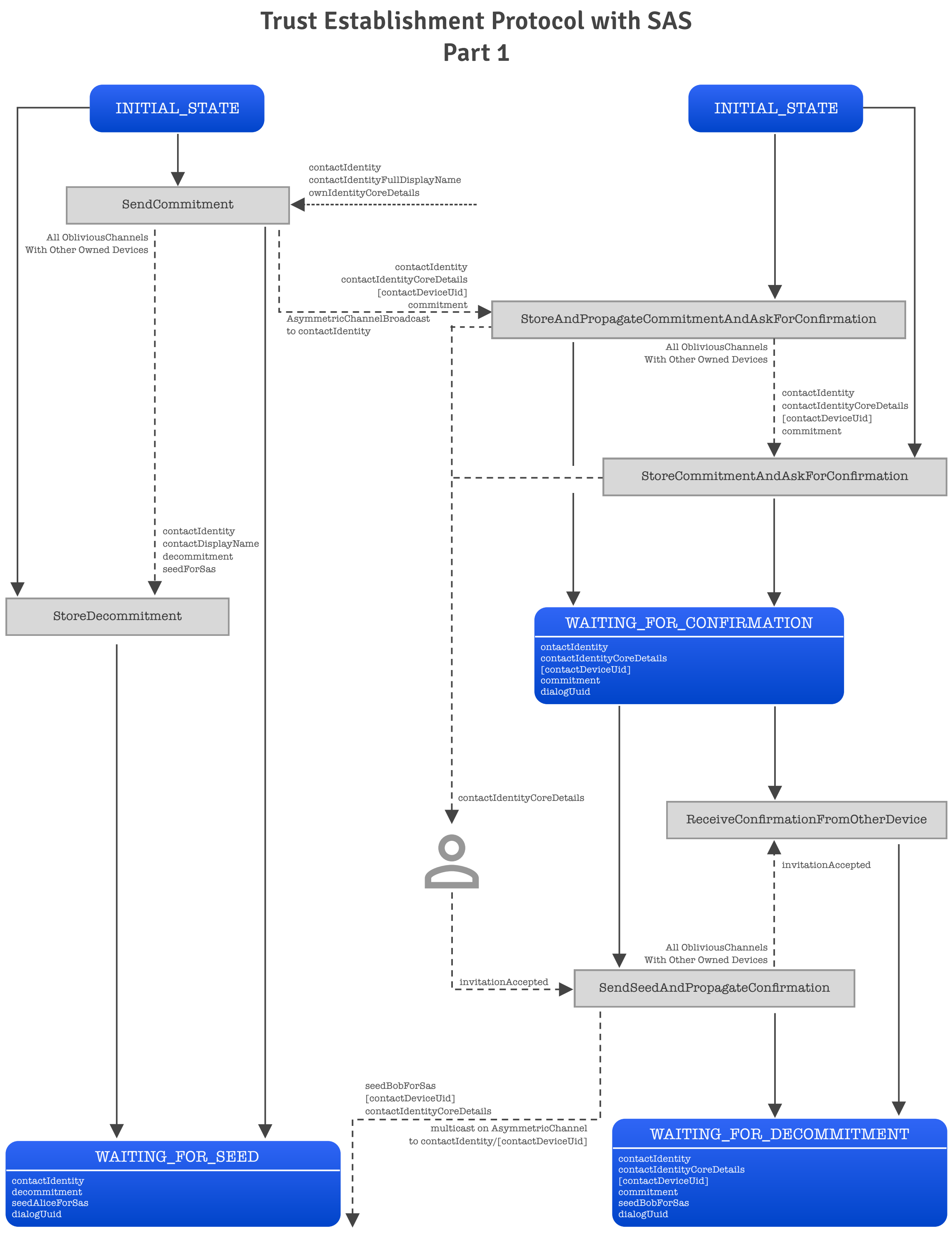


Figure 2: Trust Establishment Protocol with SAS (part 1)

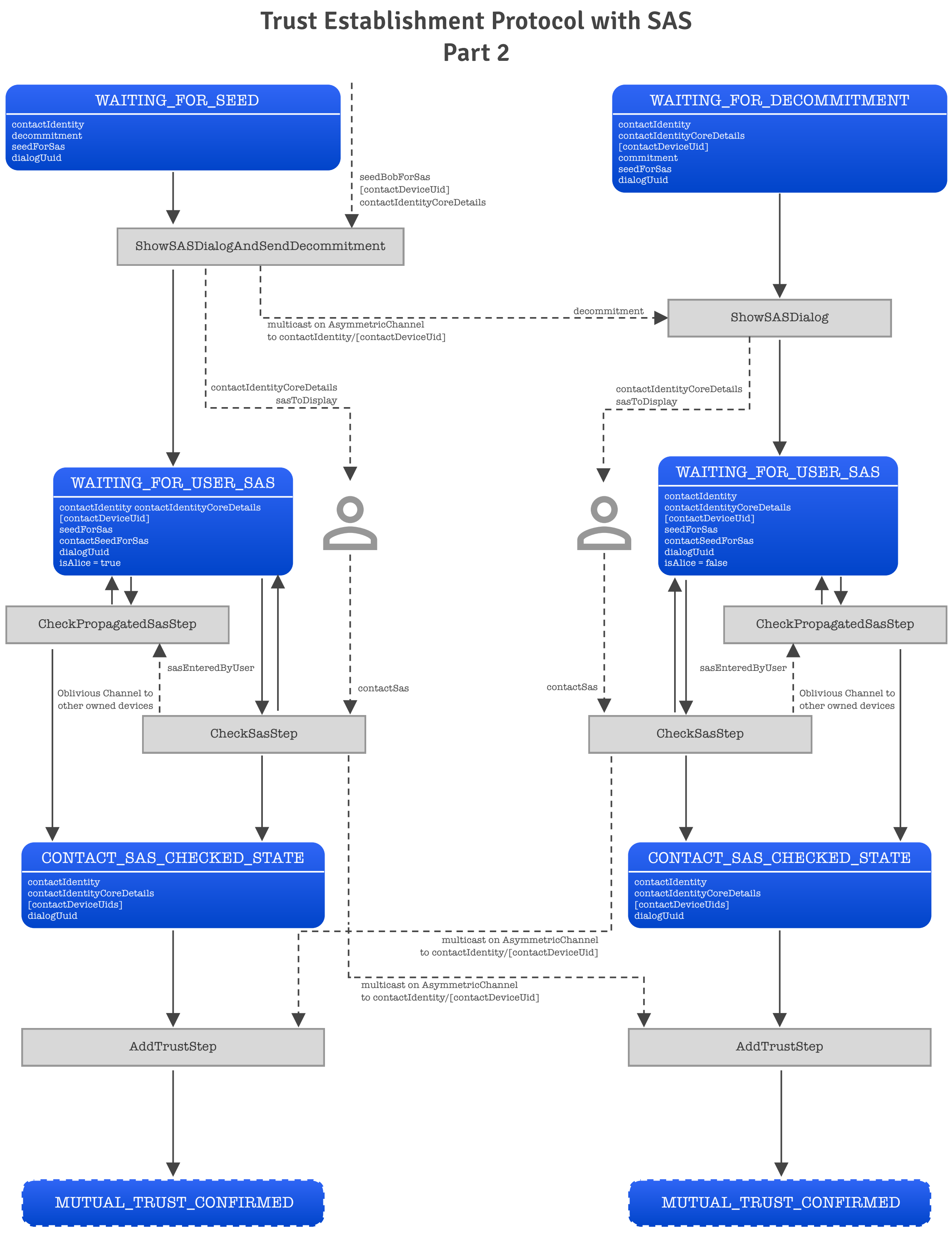


Figure 3: Trust Establishment Protocol with SAS (part 2)

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

```
1: procedure ComputeSAS($\texttt{seedForSas}_{Alice}, \texttt{seedForSas}_{Bob}, \texttt{identity}_{Bob}$)
2:     $\texttt{hash} \leftarrow \mathsf{SHA256}(\texttt{identity}_{Bob} \| \texttt{seedForSas}_{Alice})$
3:     $\texttt{seed} \leftarrow \texttt{hash} \oplus \texttt{seedForSas}_{Bob}$
4:     Initialize prng, a PRNGWithHMACWithSHA256 using seed
5:     $SAS \leftarrow \mathsf{prng.bigInt}(10^8)$
6: end procedure
```

26 Channel Creation with Contact Device Protocol

26.1 Purpose and High Level View

The Channel Creation with Contact Device Protocol represented on Figure 4 allows two mutually authenticated users, say Alice and Bob, to create an 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, still 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 25, one party will enter their SAS before the other). For this reason the protocol is architectured in the following way:

- the protocol starts with a ping stating something along the line: "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.
- 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 4)
- 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 replies with a ping
- if Bob receives a ping from Alice and trusts her `identity`, he actually starts the protocol by sending an ephemeral key
- 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

26.2 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

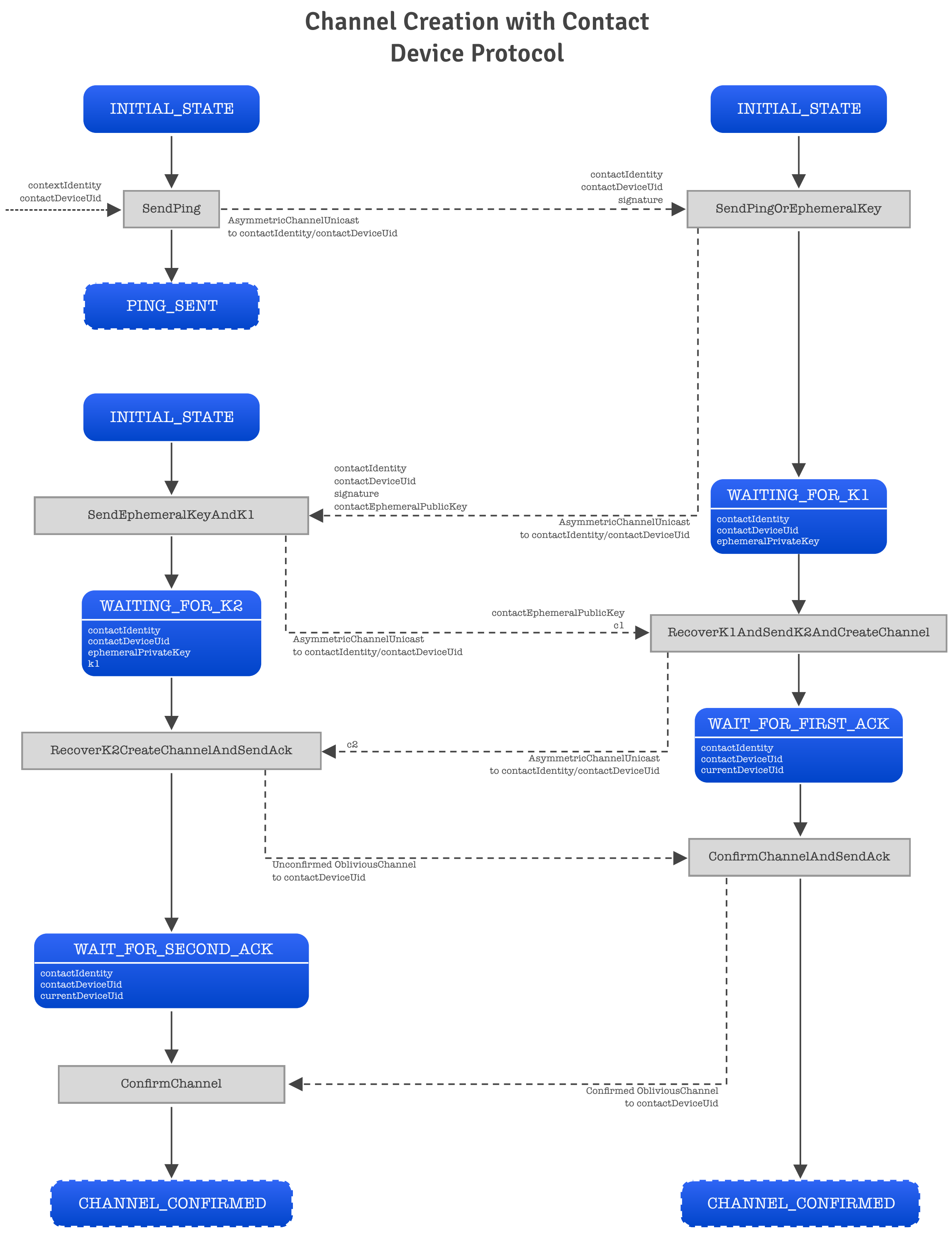


Figure 4: Channel Creation with Contact Device Protocol

- 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 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 a 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 then used by Alice and Bob to initialize each direction of the channel.

27 Device Discovery Protocol

27.1 Purpose and High Level View

The device discovery protocol is a simple protocol allowing 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 to 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 to 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 in 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 for other protocols to run the device discovery child protocol without necessarily adding the received set of `deviceUid` to the database. At the moment, no other protocol does this...

27.2 Cryptographic Details

There is no cryptography involved in this protocol.

28 Contact Mutual Introduction Protocol

28.1 Purpose and High Level View

This protocol allows a user to introduce two users he is 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 chose 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 shes 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 25).

Note that depending on the trust level between Bob and Alice (resp. Dave and Alice), the contact introduction may be automatically accepted by Bob (resp. Dave). This is easily configured in the app, but not through a user setting. Future versions of the app will probably never accept an introduction automatically.

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:

- a constant prefix "mutualIntroduction"
- the `identity` of the mediator (the party running the `IntroducContacts` step)
- the `identity` of Bob
- the `identity` of Alice
- a random 16-byte padding (also included in the signature)

29 Identity Details Publication Protocol and Contact Picture Download Child Protocol

29.1 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 his own details and publishes them, all his 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.

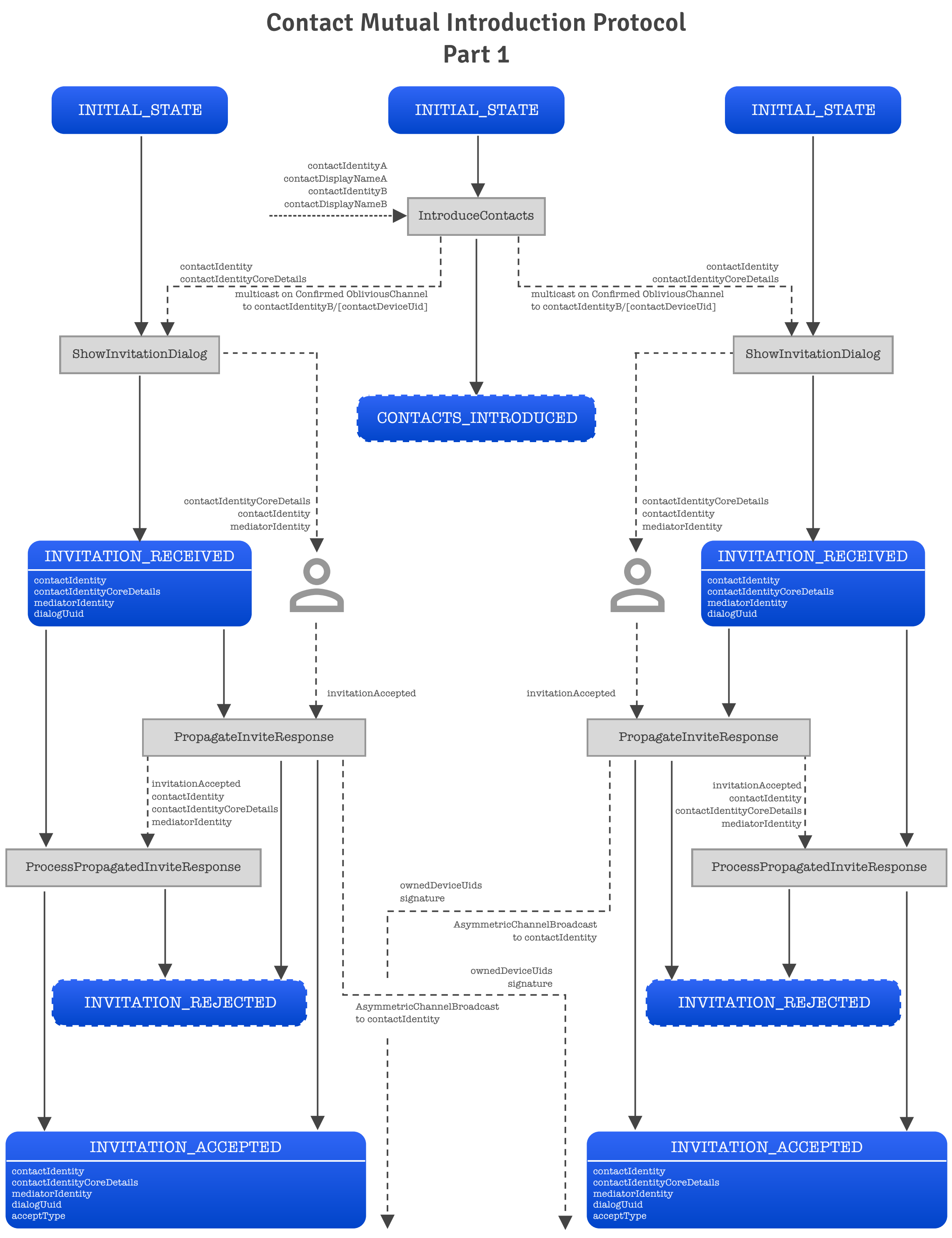


Figure 5: Contact Mutual Introduction Protocol (part 1)

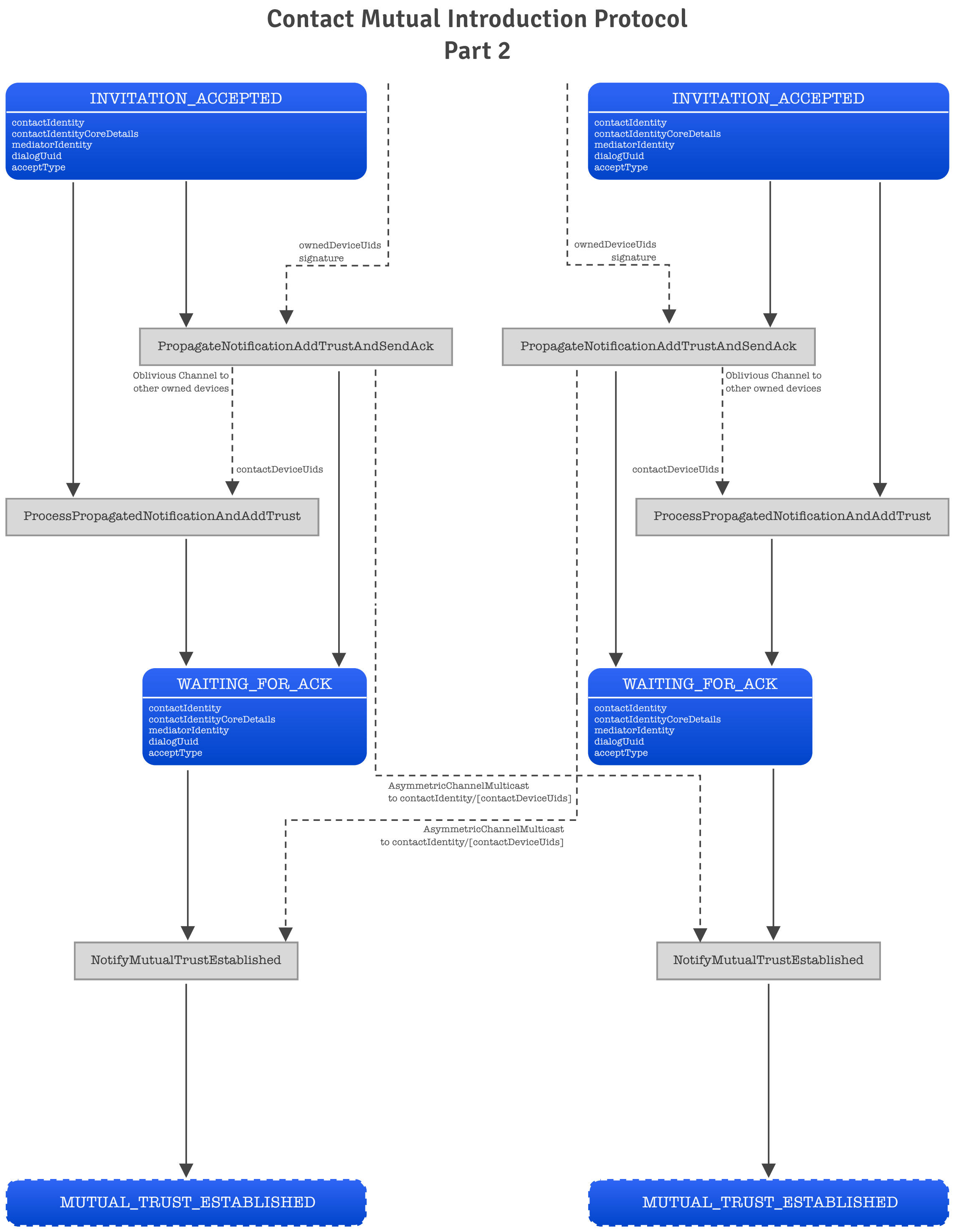

Contact Mutual Introduction Protocol
Part 2
INVITATION_ACCEPTED
contactIdentity
contactIdentityCoreDetails
mediatorIdentity
dialogUuid
acceptType
ownedDeviceUids
signature
PropagateNotificationAddTrustAndSendAck
Oblivious Channel to other owned devices
contactDeviceUids
ProcessPropagatedNotificationAndAddTrust
WAITING_FOR_ACK
AsymmetricChannelMulticast to contactIdentity/[contactDeviceUids]
NotifyMutualTrustEstablished
MUTUAL_TRUST_ESTABLISHED

Figure 6: Contact Mutual Introduction Protocol (part 2)

29.2 Cryptographic Details

Profile picture encryption. When a new picture needs to be uploaded, a random label (byte array) 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, the authenticated encryption key allows to decrypt the picture and verify that it was not manipulated by the server.

30 Group Invitation Protocol

30.1 Purpose and High Level View

This protocol allows a group owner to invite a user to join the group. When a group is created (see Section 31), 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 him.

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 (including their identity details). This way, the invited member can identify who is in the group even if they are not in contact with him.

When receiving an invitation to join a group, similarly to the Contact Introduction Protocol (see Section 28), the behavior depends on the level of trust with the group owner. The user may auto-accept the invitation or be asked for a confirmation. This is easily configured in the app, but not through a user setting. Future versions of the app will probably let users choose whether they want to auto-accept group invitations or not.

The `ReCheckTrustLevel` step is here to handle cases where the trust level of the group owner increases, switching from a trust level requiring confirmation to an auto-accept trust level.

30.2 Cryptographic Details

There is no cryptography involved in this protocol. All messages are exchanged through oblivious channels, which is enough to ensure their authenticity.

31 Group Management Protocol

31.1 Purpose and High Level View

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` for when group ownership transfer is implemented. As of today, having a deterministic `protocolUid` is not useful.

The micro-protocols are:

- Initiate the creation of the group: creates the group in database and launches all the group invitation protocols (see Section 30)

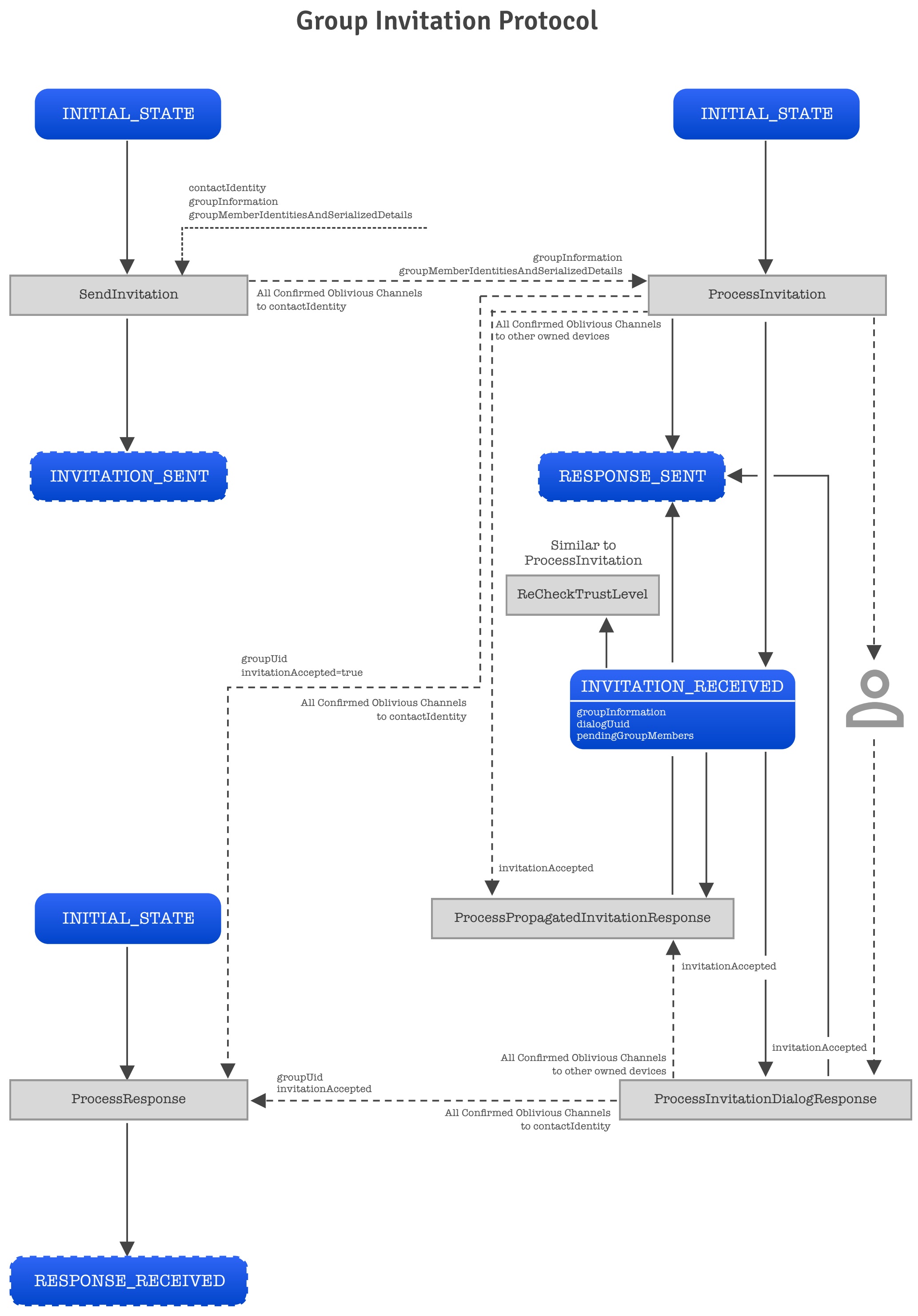

Group Invitation Protocol
INITIAL_STATE
INITIAL_STATE
contactIdentity
groupInformation
groupMemberIdentitiesAndSerializedDetails
groupInformation
groupMemberIdentitiesAndSerializedDetails
SendInvitation
All Confirmed Oblivious Channels to contactIdentity
ProcessInvitation
All Confirmed Oblivious Channels to other owned devices
INVITATION_SENT
RESPONSE_SENT
Similar to ProcessInvitation
ReCheckTrustLevel
groupUid
invitationAccepted=true
All Confirmed Oblivious Channels to contactIdentity
INVITATION_RECEIVED
groupInformation
dialogUuid
pendingGroupMembers
invitationAccepted
INITIAL_STATE
ProcessPropagatedInvitationResponse
invitationAccepted
All Confirmed Oblivious Channels to other owned devices
invitationAccepted
groupUid
invitationAccepted
ProcessResponse
ProcessInvitationDialogResponse
All Confirmed Oblivious Channels to contactIdentity
RESPONSE_RECEIVED

Figure 7: Group Invitation Protocol

- Notify group members that the members of the groups 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 he 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.

31.2 Cryptographic Details

There is no cryptography involved in this protocol. All messages are exchanged through oblivious channels, which is enough to ensure their authenticity. Each group has an owner attached to its definition, and members can check that messages are indeed received from the group owner through an oblivious channel.

32 Oblivious Channel Management Protocol

32.1 Purpose and High Level View

This protocol serves the same purpose as the group management protocol (see Section 31), but for a one-to-one relation. It currently contains a single 1-step protocol which is run when a contact is deleted. This protocol makes sure that when Alice removes Bob from her contact list, Bob's oblivious channel with Alice is also destroyed and Alice his removed from Bob's contacts.

32.2 Cryptographic Details

There is no cryptography involved in this protocol. All messages are exchanged through oblivious channels, which is enough to ensure their authenticity.

33 Full Ratchet Protocol

33.1 Purpose and High Level View

The full ratchet protocol allows 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 26). The main differences are that this protocol only refreshes one direction of the channel at a time and that is can use the oblivious channels already established, making it much simpler than the channel creation.

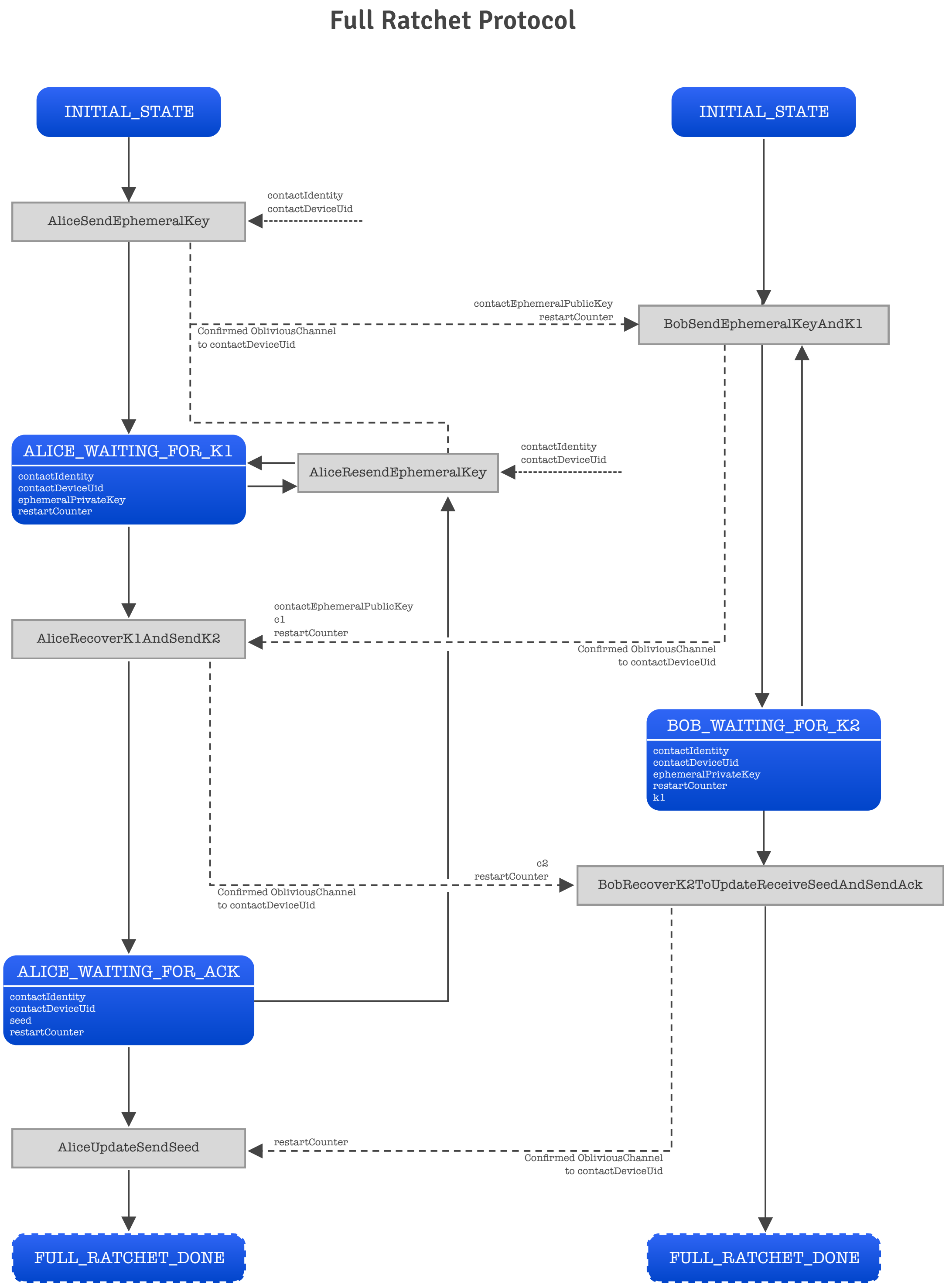


Figure 8: Full Ratchet Protocol

This protocol is triggered automatically after Alice sends a message to Bob if more than 100 messages were sent since the last full ratchet or if the last full ratchet was more than a week ago. This protocol is designed in a way allowing it to be restarted in the middle of a full ratchet. This guarantees that even if Bob receive Alice's message in disorder he will be able to decrypt all of them properly, without the full ratchet interfering.

33.2 Cryptographic Details

The cryptography of the full ratchet is exacly the same as in the Channel Creation with Contact Device Protocol (see Section 26), except that no signature is required as users already have an oblivious channel guaranteeing the authenticity of the messages.

Part VI
Groups v2

34 General Concepts

Legacy Olvid groups has the advantage of being completely decentralized but were limited to one administator: the creator of the group. The initial group creator was linked to this group forever, with no possibility to transfer 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 mulitple administator 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 on the server.

Of course, as the server is considered as 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 asymetric channel (to ensure forward secrecy of the group structure), and a **version seed** that is updated with each group modification to ensure members who were kicked from the group can no longer access the blob's contents.

Administation 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 administator. For this, each blob is uploaded with an authentication public key attached to it: modifying the blob requires to sign the new blob with the private key associated to this public key.

This authentication private key is shared among all groups administrator and is modified everytime the group administrators are modified. 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 SHA256 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 indeed check the `groupUid` matches the hash of the first administator 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 administator becam administrator of the group. For this, we build some sort of block chain to which a new block is added everytime administrators of the group change.

- block n contains the SHA256 hash of block $n-1$ and the encoded updated list of administrator `identity`
- block n is then signed by the administrator updating the group, which 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 administator 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 administators 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 everytime 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-bytes group invitation nonce
- a list of premission strings:
  - `"ga"`: group admin permission, allowing to update the group blob
  - `"rd"`: remote delete anything permission, allowing to delete messages sent by other users for all group members
  - `"eo"`: edit or remote delete own messages permission
  - `"cs"`: change settings permission, allowing to change the groups shared settings (ephemeral settings for example)
  - `"sm"`: send message permission, allowing 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 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 accepted 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 grain permissions for each member. There are currently four group types:

- **Standard**: all members are admin and can invite other members. Anyone can send messages, changes settings, edit or deleete owned 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**: where admins can be chosen (and have admin, change settings and edit or remote delete own messages) and 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 to notify the group owner that 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. Everytime the group blob is updated, the log is consolidated inside the blob.

One important point is that such 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 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 accepted to join the group or if we are not yet aware they accepted to join the group. When posting a message 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 htey 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. In terms of user experience, the benefit of this is that when a group is created, they will be able to have 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 uplodaded 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 organisation'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 siganture key as for a user's signed details, again in the form of a JWT.

Part VII
Keys and Contacts Backup

The Olvid engine implements mechanisms to allow a user to backup the long term key pairs of his `identity` as well as the `identity` of his trusted contacts and the groups he belongs to.

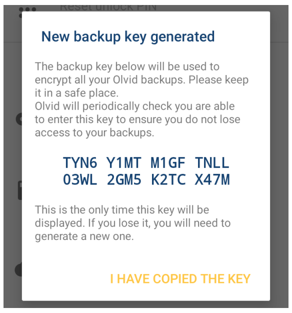
Of course, backups should never be done in clear and encrypting them with a password would be way too weak for most users. So before proceeding to a backup, a strong backup key must be generated. In practice this backup key is a seed used with a PRNG as described in Section 35.

Then, a backup is a JSON string (formatted as described in Section 36.1) containing dumps from the identity databases. This JSON string is first compressed, then encrypted using the backup key (see Section 36.2) and can be either exported to a file, or uploaded automatically to the cloud (iCloud for the iOS client, Google Drive for the Android client).

35 Backup Seed

35.1 Seed Format

An Olvid backup key is a 160-bit seed which is presented to the user as 8 strings of 4 characters (see Figure 9). Each of these 32 characters contains 5 bits of entropy with the correspondance of Table 3. When displayed to the user, the first of the corresponding character is used (number or capital letter), but when the user enters the key for a restore, all equivalent characters are accepted.



New backup key generated

The backup key below will be used to encrypt all your Olvid backups. Please keep it in a safe place.
Olvid will periodically check you are able to enter this key to ensure you do not lose access to your backups.

TYN6 Y1MT M1GF TNLL
03WL 2GM5 K2TC X47M

This is the only time this key will be displayed. If you lose it, you will need to generate a new one.

Figure 9: 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 3: Backup seed character to 5-bit value correspondance.

35.2 Key Derivation

The backup seed itself is never stored in the application. It is displayed to the user, then a set of keys are 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.

Note that, as of today, the backupKeyUid is not used. It will be used when storing encrypted backups directly on the Olvid server becomes an option.

36 Backup Contents

36.1 JSON Structure

Each backup contains a JSON object having the following structure. Types in between <> refer to other intermediate objects defined here. This object naturally has a tree structure, and the complete path from the object root to any leaf is important to determine the meaning of a leaf. For example, a Contact groups that is a direct descendants of an Owned identity is a group for which you are the owner, whereas a direct descendant of a Contact identity is a group owned by this contact.

Top level object:

```
{
  "engine": {
    "identity_manager": [<Owned identity>]
  },
  "backup_json_version": int,
  "backup_timestamp": int
}
```

Owned identity:
```
{
  "owned_identity": byte[],
  "private_identity": <Private identity>,
  "published_details": <Owned identity details>,
  "latest_details": <Owned identity details>,
  "api_key": String,
  "contact_identities": [<Contact identity>],
  "owned_groups": [<Contact group>]
}
```

Private identity:
```
{
  "server_authentication_private_key": byte[],
  "encryption_private_key": byte[],
  "mac_key": byte[]
}
```

Owned identity details:
```
{
  "version": int,
  "serialized_details": String,
  "photo_server_label": byte[],
  "photo_server_key": byte[]
}
```

Contact identity:
```
{
  "contact_identity": byte[],
  "trusted_details": <Contact identity details>,
  "published_details": <Contact identity details>,
  "trust_level": String,
  "trust_origins": [<Contact trust origin>],
  "contact_groups": [<Contact group>]
}
```

Contact identity details:
```
{
  "version": int,
  "serialized_details": String,
  "photo_server_label": byte[],
  "photo_server_key": byte[]
}
```

Contact trust origin:
```
{
  "trust_type": int,
  "mediator_or_group_owner_identity": byte[],
  "mediator_or_group_owner_trust_level_major": int
}
```

Trust type is one of 0 (SAS exchange), 1 (group invitation), 2 (contact introduction).

Contact group:
```
{
  "group_uid": byte[],
  "published_details": <Contact group details>,
  "latest_details": <Contact group details>,
  "trusted_details": <Contact group details>,
  "group_members_version": int,
  "members": [<Group member>],
  "pending_members": [<Pending group member>]
}
```

Latest details are only set for groups you own, trusted only for groups you joined.

Contact group details:
```
{
  "version": int,
  "serialized_details": String,
  "photo_server_label": byte[],
  "photo_server_key": byte[]
}
```

Group member:
```
{
  "contact_identity": byte[]
}
```

Pending group member:
```
{
  "contact_identity": byte[],
  "serialized_details": String,
  "declined": boolean
}
```

36.2 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 35.2) in the following manner:

- first compress the JSON string (using raw deflate/zlib compression) into a byte array
- then use the KEMPublicKeyOverEC to perform a KEMOverEC.encrypt of the byte array
- compute a HMACWithSHA256.compute of the ciphertext using `macKey` and append it to the ciphertext

36.3 Backup Decryption

Backup decryption only happens during a restore (see Section 37). The user must first enter the backup key from which the keys (including the KEMPrivateKeyOverEC) can be derived. Decryption is the exact reverse of the encryption, with checks 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 KEMPrivateKeyOverEC to perform a KEMOverEC.decrypt of the ciphertext
- finally, decompress the plaintext into a JSON string

37 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 to 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 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 27) is run
- Device discovery adds some `deviceUid` for each contact, which triggers channel creation protocols (see Section 26)
- 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)
    $\multimap$ during this protocol, if the received details version is lower than what is already in database, a "downgrade" is authorized.
  - For every group owned by the other user, query the latest group members (see Section 31)
  - For every owned group to which the other user belongs, reinvite him and forcibly push the updated group members (see Section 31)
    $\multimap$ again, in this protocol, "downgrade" of the group members and details is possible

Part VIII
Secure Olvid Calls

Olvid allows payed users to initiate secure phone calls with their contacts. It relies on WebRTC for all the "heavy-lifting" and uses the established oblivious channels to securely exchange signaling messages. The way call initiation works is very similar to the model used in cryptographic protocols:

- each call is assigned a random call identifier (an UUID)
- each signaling message has integer "message type"
- each call participant maintains a "state"

38 Olvid Call Signaling Messages

The following message types are exchanged during Olvid secure VoIP calls. Some contain Session Description Protocol (SDP) data, other simply contain their "message type" to let the correspondant advance to the next protocol step. All messages are exchanged as serialized JSON strings, encapsulated in a Message payload (see Section 22.2).

ID	Call signaling message
0	Start call message
1	Answer call message
2	Reject call message
3	Hanged up message
4	Ringing message
5	Busy message
6	Reconnect call message

Table 4: List of call signaling messages in Olvid.

In addition to signaling messages used to establish the WebRTC connection, a data channel between participants allows to exchange in-call messages. A single in-call message type exists

ID	In-call message
0	Muted data message (indicating whether a participant is muted or not)

Table 5: List of in-call messages in Olvid.

The JSON structure of these messages is as follows:

Start call message:
```
{
  "sdt": String,    // session description type
  "sd": byte[],     // gzipped session description
  "tu": String,     // TURN server username
  "tp": String      // TURN server password
}
```
Answer call message:
```
{
  "sdt": String,    // session description type
  "sd": byte[],     // gzipped session description
}
```
Reject call message:
```
{
}
```
Hanged up message:
```
{
}
```
Ringing message:
```
{
}
```
Busy message:
```
{
}
```
Reconnect call message:
```
{
  "sdt": String,    // session description type
  "sd": byte[],     // gzipped session description
}
```
Muted data message:
```
{
  "muted": boolean  // true if participant is muted
}
```

39 Technical Details

39.1 TURN Credentials

All calls in Olvid are relayed through a TURN server. This TURN server runs the coTURN [10] software and is hosted at `turn.olvid.io`. Before being able to participate in a call, Olvid has to authenticate with the server using the TURN REST API protocol [19]. This authentication mechanism uses a timestamp as the username and a HMAC of this timestamp as the password, allowing coTURN to run in a completely "anonymous" way. The call iniator queries the Olvid server for valid credentials (see `getTurnCredentials` detailed in Section 49.2) and received two pairs of username/password: one for him, one for the call recipient. Both are valid for 24 hours but are in fact discarded at the end of the call.

The recipient's credentials are sent in the start call message, allowing him to participate to the call without needing him to access the `getTurnCredentials`.

39.2 Session Description Protocol (SDP)

An SDP (see Figure 10) contains all the required information for the negotiation of a WebRTC connection. In particular it contains:

- the different channels to create (lines starting with `m=`). This is one audio channel and one data channel in Olvid.
- ICE candidates for each channel. These are filtered to only offer TURN relay candidates (this allows to hide each participant's IP address to his peer)
- the SHA256 fingerprint of the certificate used for DTLS
- for the audio channel, all the codec options and parameters supported. These are filtered to only offer constant bit-rate (CBR) Opus, PCMA and PCMU.

The SDP exchanged during the signaling are also accompanied by a session description type which is always the string "offer" for the start call and reconnect call messages and "pranswer" for the answer call message.

```
v=0
o=- 9073337791557194l4 2 IN IP4 127.0.0.1
s=-
t=0 0
a=group:BUNDLE 0 1
a=msid-semantic: WMS
m=audio 57544 UDP/TLS/RTP/SAVPF 111 0 8 110 112 113 126
c=IN IP4 15.237.42.185
a=rtcp:9 IN IP4 0.0.0.0
a=candidate:3072461082 1 udp 8331007 15.237.42.185 57544 typ relay raddr 0.0.0.0 rport 0 generation 0 network-id 4 network-cost 10
a=candidate:3072461082 1 udp 8331263 15.237.42.185 52261 typ relay raddr 0.0.0.0 rport 0 generation 0 network-id 4 network-cost 10
a=ice-ufrag:95hC
a=ice-pwd:Zg7QjjhtqaH24W8RDU4UDPkl
a=ice-options:trickle renomination
a=fingerprint:sha-256 20:B9:CB:D5:FA:20:DE:D5:4E:8B:82:B6:52:73:8A:0B:2E:D4:A5:64:EA:FB:06:E5:F1:60:F3:94:E2:D9:E9:7B
a=setup:actpass
a=mid:0
a=extmap:1 urn:ietf:params:rtp-hdrext:ssrc-audio-level
a=extmap:2 http://www.webrtc.org/experiments/rtp-hdrext/abs-send-time
a=extmap:3 http://www.ietf.org/id/draft-holmer-rmcat-transport-wide-cc-extensions-01
a=extmap:4 urn:ietf:params:rtp-hdrext:sdes:mid
a=extmap:5 urn:ietf:params:rtp-hdrext:sdes:rtp-stream-id
a=extmap:6 urn:ietf:params:rtp-hdrext:sdes:repaired-rtp-stream-id
a=sendrecv
a=msid:- audio0
a=rtcp-mux
a=rtpmap:111 opus/48000/2
a=rtcp-fb:111 transport-cc
a=fmtp:111 cbr=1;minptime=10;useinbandfec=1
a=rtpmap:0 PCMU/8000
a=rtpmap:8 PCMA/8000
a=rtpmap:110 telephone-event/48000
a=rtpmap:112 telephone-event/32000
a=rtpmap:113 telephone-event/16000
a=rtpmap:126 telephone-event/8000
a=ssrc:2169359256 cname:B2s7qDzQUyg2Z8aX
a=ssrc:2169359256 msid:- audio0
a=ssrc:2169359256 mslabel:-
a=ssrc:2169359256 label:audio0
m=application 53302 UDP/DTLS/SCTP webrtc-datachannel
c=IN IP4 15.237.42.185
a=candidate:3072461082 1 udp 8331263 15.237.42.185 53302 typ relay raddr 0.0.0.0 rport 0 generation 0 network-id 4 network-cost 10
a=candidate:3072461082 1 udp 8331007 15.237.42.185 58800 typ relay raddr 0.0.0.0 rport 0 generation 0 network-id 4 network-cost 10
a=ice-ufrag:95hC
a=ice-pwd:Zg7QjjhtqaH24W8RDU4UDPkl
a=ice-options:trickle renomination
a=fingerprint:sha-256 20:B9:CB:D5:FA:20:DE:D5:4E:8B:82:B6:52:73:8A:0B:2E:D4:A5:64:EA:FB:06:E5:F1:60:F3:94:E2:D9:E9:7B
a=setup:actpass
a=mid:1
a=sctp-port:5000
a=max-message-size:262144
```

Figure 10: Example of Session Description Protocol used in Olvid calls

39.3 Security

WebRTC handles most of the security by itself, the only requirement is to be able to exchange the SDP over an authentic channel. This is the case in Olvid as the oblivious channels used to send application messages are used. The SDP contain a fingerprint of the certificate that will be used in the DTLS negotiation. If the SDP are exchanged over an authentic channels, both peers are guaranteed to receive an unmodified SDP and to receive the correct fingerprint. From there, it is impossible for an adversary (even an adversary controlling the TURN server) to run a man-in-the-middle attack and eavesdrop on the conversations.

Part IX
Keycloak - Olvid User Directory

40 General Concepts

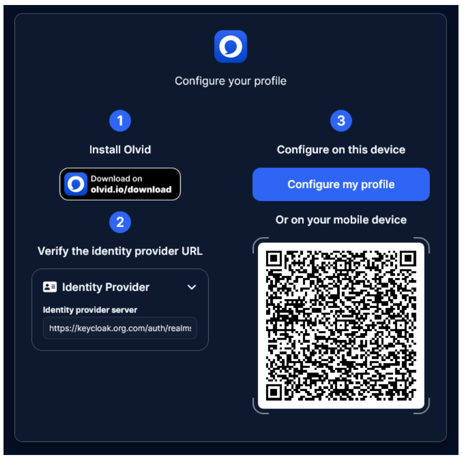
Olvid is designed to allow the creation of secure communication channels between users without having to trust any server. This is great from a security standpoint, but also requires user to interact in order to get in contact. To simplify the deployment of Olvid in large organisations, the Olvid Enterprise subscription allows an organisation 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 organisation 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 organisation 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 users. 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 him. This detection consists in verifying the signature of the JWT (using the same signature public key as for the user's owned `identity` signature) and making sure the signature is less than **2 months old**. Such users are identified in app with a small shield next to their name.

41 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.

41.1 Get owned details (`me`)

This is the main entry point to retrieve information about you on Keycloak.

`olvid-rest/me`		
in	`"timestamp"`	`int` timestamp of last `me` request
out	`"signature"`	user's signed details
	`"server"`	Olvid distribution server url
	`"revocation-allowed"`	owned `identity` overwrite allowed
	`"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 they need. 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 allows to know on which server to create it, when binding an existing `identity` to Keycloak, this is used to check the servers match, otherwise Keycloak will not be able to deliver a valid api key/licence 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 to their Keycloak user. Having this setting to `True` would allow an attacker having stolen a user's Keycloak credentials to "replace" them on Keycloak.
- the Olvid api key is the licence a user gets to unlock paid Olvid features. This api key must be presented to the server using the `registerApiKey` entry point (see Section 43.2).
- 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 46.1). One such push topic is associated to each Keycloak group the user is 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 has the possibility to 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 revocation 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 that 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"` or `"dekstop"`. In case of 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.

41.2 Get groups (groups)

This entry point allows retrieving Keycloak group blobs for any group that was modified since our last request. Like 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"`	`String` array of signed group blobs
	`"signed_group_deletions"`	`String` array of signed group deletions
	`"signed_group_kicks"`	`String` array of signed group kicks
	`"current_timestamp"`	current Keycloak server timestamp
error codes		1: unknown error 2: authentication required

All signed elements in the output are timnestamped 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 everytime a group is disbanded,
- group "kicks" are generated when a user is removed from the group.

Deletion and kicks are used as a 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.

41.3 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 ouput 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 to the authenticated keycloak user and `identity` overwrite is not allowed,
- idenitty was revoked: if the input `identity` is on Keycloak's revocation list, a revoked identity can never be reassociated to a user.

41.4 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.

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"` nos found on Keycloak. The output are signed details formated exactly as in the `"me"` output.

41.5 Contact search (`search`)

This entry points 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 onb each of the element. 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.

41.6 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. Contrary to 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 a JSON, it is a single `byte` equal to `0x01` if the user was indeed revoked and their `nonce` can no longer be found in database. Any other response is canisdered as `False`, meaning the user still exists and can be requested to authenticate.

41.7 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 ouptput verification response is `True` if the contact is still a Keycloak user, `False` otherwise.

41.8 Group profile picture download (`getData`)

This entry point is also not authenticated and replicates the `getUserData` server entry point (see Section 48.2). 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

Part X
Olvid Server API

This section describes the different API entry point 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 18, 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.
  - 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 always 200, whether or not an error occured. Only the byte return status is of importance.
- 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 additionnal keys depending on the `action`
- a Transfer WebSocket API:
  - this second WebSocket API is used only when transfering a profile from a 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 URL allowing 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].

42 Server Authentication API

The folowing 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.

42.1 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 mulitple authentications from the same `identity` simultaneously. It is sent again (see Section 42.2) with the `response` to retrieve the corresponding `challenge`. The `nonce` in the output in the same as in the input.

`/requestChallenge`		
in	`identity`	identity to authenticate
	`byte[32]`	`nonce`
out	`byte[1]`	return status
	`byte[32]`	`challenge`
	`byte[32]`	`nonce`
return statuses		`0x00`: OK
		`0xff`: unknown error

42.2 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.

/getToken		
in	identity	identity to authenticate
	byte[80]	response to the challenge
	byte[32]	nonce
out	byte[1]	return status
	byte[32]	token
	byte[32]	nonce
	int	API key status
	long	API key permissions
	long	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 licences 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 if non-zero, it means the API key has an expiration timestamp. In this case, the `long` reprensents the number of milliseconds since Epoch for this timestamp. For expired key, this timestamp is in the past.

API key permissions. The permissions `long` is a bit-encoded set of permissions. A 1-bit indicate the permission is granted, a 0-bit indicate 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 not associated to an identity and are never stored in a database. They are composed of:

- a timestamp 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 an UUID like other API keys. The HMACWithSHA256Key used for this computation is stored only on the server and is also used in the `/queryApiKeyStatus` and `/freeTrial` server entry points.

43 API Keys and Subscriptions API

As descriped in the `/getToken` entry point, all permissions and access to premium features in Olvid is 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 allow 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 new API key in exchange for a valid purchase receipt.

43.1 Query API key status

The `/queryApiKeyStatus` entry point allows a user to query whether the licence activation link he 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 licences: if the API key is already used by someone else, the current user will not be able to activate it.

`/queryApiKeyStatus`		
in	`identity`	identity
	`UUID`	API key
out	`byte[1]`	return status
	`int`	API key status
	`long`	API key permissions
	`long`	API key expiration
return statuses		`0x00`: OK
		`0xff`: unknown error

43.2 Register API key

The `registerApiKey` entry point allows to change the API key associated to the `identity` calling it. This entry point requires being authenticated. Each `identity` can only be associated to one API key at a time. By default, it is not associated to any API key. The registration may fail if the API key is unknown, already expired or already associated to another `identity`.

`/registerApiKey`		
in	`identity`	identity
	`byte[32]`	Authentication `token`
	`UUID`	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

43.3 Free trial API key query and retrieval

This entry point allows user whether they are still eligible to a free trial or not, and if yes, allows them to retreive a free trial API key valid one month. This entry point requires being authenticated. The free trial API keys have the structure described in Section 42.2.

`/freeTrial`		
in	`identity`	`identity`
	`byte[32]`	Authentication `token`
	`boolean`	`true` to retrieve the API key, `false` for an eligibility query
out	`byte[1]`	return status
	`UUID`	Free trial API key (only in retrieve mode)
return statuses		`0x00`: OK
		`0x04`: invalid `token` (re-authentication required)
		`0x0f`: free trial already used
		`0xff`: unknown error

43.4 In-App purchase receipt verification

This entry point verifies a purchase receipt from either a Goole Play or an App Store purchase and returns an API key with the appropriate expiry 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 Android, the `/getToken` entry point takes care of querying the Google Play server when a user authenticates to see if his subscription was renewed. Eventhough purchase receipts are signed, receipt verification is not done on the server, but the server rather queries the Google and Apple servers which take care of the validation and can provide the most up-to-date expire time for the purchase.

/verifyReceipt		
in	`identity`	`identity`
	`byte[32]`	Authentication `token`
	`byte[1]`	Store identifier (see below)
	`String`	Store purchase receipt
out	`byte[1]`	return status
	`UUID`	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
		`0xff`: unknown error

Store identitfiers. This entry points uses different receipt verification queries depending on the store it was generated on.

Byte	Store
`0x00`	legacy iOS store (using StoreKit 1)
`0x01`	Android store
`0x02`	iOS store (using StoreKit 2)

44 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.

44.1 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 message containing image attachments, this extended payload is the JPEG data of a "mosaic" of small 40×40 pixels thumbnails of these images. This extended payload is typically a few kB long. This input is optionnal and is only present for messages with such a extended payload.
3. The two booleans `isApplicationMessage` and `isVoipMessage` are used when notifying the message recipient:
   - application messages showing an "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 compared to a normal notification for others.
4. **Encoded attachment lengths**: the lists of encoded attachment lengths and attachments chunk length 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 canceling 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 to upload it directly to AWS S3, without going through one of the server entry points. The list of private signed url contains 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 21.3.

   Computing signed urls is quite time consuming and for messages with multiple attachments (or attachments with mutliple chunks), the server may decide to return "dummy" signed urls instead. Such urls are already expired and need to be refreshed using the `/uploadAttachment` entry point below.

44.2 Batch message upload

To optimize server bandwidth and decrease the number of API requests, it is possible to upload attachment-less messages by 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	[	a repetition of n of the following list-encoded 4 elements
	[encoded_vals]	list of encoded headers (see details 1 above)
	byte[]	encrypted message payload
	boolean	message contains an application payload (see details 3 above)
	boolean	message is a start call VoIP message (see details 3 above)
	]	
out	byte[1]	return status
	[	a repetition of n of the following list-encoded 3 elements
	byte[32]	message unique identifier on the server
	byte[32]	nonce (see details 5 above)
	long	server timestamp
	]	
return statuses		0x00: OK
		0x18: payload too large
		0xff: unknown error

44.3 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 reason and backward compatibility the path to this entry point is `/uploadAttachment`.

/uploadAttachment		
in	byte[32]	message unique identifier on the server
	int	attachment number
	byte[32]	the nonce received when uploading the message
out	byte[1]	return status
	[encoded_vals]	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 poiny, the list of private signed urls contains many private signed urls as the attachment with the specified number has chunks. Each signed url is a string, encoded as described in Section 21.3.

44.4 Get attachment upload progress

The `/getAttachmentUploadProgress` entry point allows querying the server which attachment chunks have been succefully upoaded. 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

44.5 Cancel attachment upload

This tags an attachment as canceled on the server. The consequence is that recipients will only receive download signed urls for chunks that have actually been uploaded to the server. For uncomplete attachments, the recipients can know that it will never be completed and cancel its donwload.

`/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 `0xff`: unknown error

45 Message Download API

45.1 Download messages and list attachments

This entry point retrieves all messages for a specific device of an `identity` that are available on the server. 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. 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, allowing the client to know a new listing is required (once all downloaded messages have been deleted/marked as listed).

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`	`identity`
	`byte[32]`	Authentication `token`
	`byte[32]`	`deviceUid`
out	`byte[1]`	return status
	`long`	current server timestamp (milliseconds since Epoch)
	`encoded_val`	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 (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 `long` of the encrypted attachment length
  - the encoded `int` of the encrypted attachment chunk length
  - the encoded list of encoded `String` of the urls for attachment chunks download. This

list contains as many element as the number of attachment chunks.

Note that (apart from broadcast messages that arrive last) the messages are always sorted before being returned, from oldest to newest, based on the server timestamp at upload time. If the listing is truncated, only the oldest messages are returned.

45.2 Download message extended payload

Then entry point `/downloadMessageExtendedContent` allows to download the extended payload of messages that have one (indicated by a boolean when listing messages). Please refer to Section 23.2 for information on how this extended paylod 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

45.3 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 reason 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 `0xff`: unknown error

If the attachment was not marked as canceled (see Section 44.5), this method returns a valid download url for each attachment chunk. On the contrary, if the attachment was marked as canceled, a valid download url is returned only for chunks that are actually available on AWS S3. For other chunks, an encoded empty `String` is returned instead.

45.4 Delete message and attachments

Once a message and all its attachments have been downloaded by a recipient (`identity`, `deviceUid`) pair, it 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 remain for a message, it 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 or mark as listed messages.

`/deleteMessageAndAttachments`		
in	`identity`	`identity`
	`byte[32]`	Authentication `token`
	`byte[32]`	`deviceUid`
	`[encoded_vals]`	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 do 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 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.

46 Device Management API

The device management API regroups 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.

46.1 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 46.2).

/registerPushNotification		
in	`identity`	`identity`
	`byte[32]`	Authentication `token`
	`byte[32]`	`deviceUid`
	`byte[1]`	push notification type identifier (see below)
	`extra_info`	push notification extra information (see below)
	`boolean`	reactivate current device (see below)
	`[String]`	keycloak push topics (see below)
	`byte[]`	encrypted device name (first registration only)
	`(byte[32])`	(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 notifications 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 allows to tell the device which of his `identity` (in case several identities are installed on the same device) has received a message. The `identity` itslef 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 protuction 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 avaible 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 disbled. Notifications are sent only through the WebSocket connection. There are no `extra_info` for this type.
- `0x11` **- Windows**: this notification type indicates a Windows device. Notifications are sent only through the WebSocket connection. There are no `extra_info` for this type.
- `0x12` **- Linux**: this notification type indicates a Linux device. Notifications are sent only through the WebSocket connection. There are no `extra_info` for this type.
- `0x13` **- Daemon**: this notification type indicates the device is a deamon as used by Olvid Bots. Notifications are sent only through the WebSocket connection. There are no `extra_info` 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 notification. If a notification must be sent to a user without an active WebSocket connection, an 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:

- if, without the multi-device option activated, the user added another device, this device may expire after 30 days and become inactive,
- 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 licence).

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 simply updates the list of push topics to which the `identity` should belong.

Encrypted device name. To help users identify their devices, when a device is first registered with the server, an encrypted device name is passed. This device name is encrypted using the `identity` public key and is thus only accessible to the user themselves. This encrypted device name is not updated when calling this entrypoint again if the device is already registered. Modifying it is only possible through the `/deviceManagement` entry point.

46.2 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 24) for each device and be informed of inactive contacts.

`/deviceDiscovery`		
in	`identity`	`identity`
out	`byte[1]`	return status
	`encoded_dict`	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`: the pre key for this device
- 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 or called `/registerPushNotification`) less than 40 days ago (`True`) or not (`False`)

46.3 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 dicover 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 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`: the pre key for 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 licence active

46.4 Device management

The `/deviceManagement` entry point allows to manage devices associated with an `identity`: change a device name, remove a device, determine which device should not expire (without a multi-device licence). 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) 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 encryted 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 transfered to the current non-expiring device.

47 Group V2 Manamgenent API

Groups V2 rely on an encrypted blob uploaded on the server to guarantee that all group members have the same view off 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.

47.1 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 administation public key that will be used to update the blob later on is uploaded.

`/groupBlobCreate`		
in	`identity`	`identity`
	`byte[32]`	Authentication `token`
	`byte[32]`	group unique identifier
	`public_key`	encoded administration public key
	`ciphertext`	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

47.2 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`
		return status
		encrypted group blob
		list of encoded group log items
		encoded administration public key

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.

47.3 Group lock

Before updating a group blob, an administator must first lock the group, effectively preventing concurrent modifications of the blob. An administrator must prove to the server it knows the private key associated to the encoded administation 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`
		return status
		encrypted group blob
		list of encoded group log items
		encoded administration public key

return statuses		`0x00`: OK `0x09`: group identifier not found (or already deleted) `0x13`: group is already locked `0x14`: invalid signature `0xff`: unknown error

The output of this entry point is the same as the `/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.

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 to the blob.

47.4 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]`	group unique identifier
	`byte[32]`	lock `nonce`
	`ciphertext`	encrypted group blob
	`public_key`	encoded administration public key
	`byte[]`	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, 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 to the blob.

47.5 Group log put

When leaving a group, a member must upload a "log entry" to the server. This log entry is a signature of the group invitation nonce of the user. This entry point is not authenticated and the server only checks the length of the log entry. 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

47.6 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 to the blob. Note that the signature does not depend on the blob or the group identifier, but only on the signature key.

48 User Data Management API

48.1 Put user data

This entry point allows users to upload some data associated to 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.

`/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

48.2 Get user data

This entry point allows any user to download user data associated to 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

48.3 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 a 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

48.4 Delete user data

This entry point allows a user to delete obsolete user data he uploaded on the server. Typically, after changing 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

49 Other REST API Entry Points

49.1 Upload return receipts

The application has the possibility to notify a message sender that his message was indeed delivered to the its recipient (or read by the user). This entry point allows to upload a batch of return receipts on the server. The delivery of the return receipt to the application message sender is then done through the WebSocket.

/uploadReturnReceipt		
in	[	a repetition of n of the following list-encoded 4 elements
	identity	identity
	[encoded_vals]	list of encoded deviceUid
	byte[16]	nonce
	byte[]	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 him 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.

Note that return receipts also contain a timestamp which is set by the server during the execution of this entry point.

49.2 Retrieve TURN credentials to initiate a call

This entry point requires a licence with the call permission. When checking the validity of the `token`, the server also checks the associated permission. If user is allowed to initiate call, 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 39.1).

/getTurnCredentials		
in	identity	identity
	byte[32]	authentication token
	String	username 1
	String	username 2
out	byte[1]	return status
	String	timestamped username 1
	String	password 1
	String	timestamped username 2
	String	password 2
return statuses		0x00: OK
		0x04: invalid token (re-authentication required)
		0x0e: permission denied (call initiation not allowed for user)
		0xff: unknown error

49.3 Upload pre key

The /uploadPreKey entry point allows a user to create or update the pre key stored by the server (see Section 24). 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 during using the /deviceDiscovery entry point.

/uploadPreKey		
in	identity	identity
	byte[32]	authentication token
	byte[32]	deviceUid
	byte[]	encoded signed pre key
out	byte[1]	return status
return statuses		0x00: OK
		0x04: invalid token (re-authentication required)
		0x0b: device is not registered
		0x14: invalid signature
		0xff: unknown error

The invalid signature return status indicates that: the pre key signature is invalid, or the pre key is for another deviceUid, or the server already has a more recent pre key. See Section 24 for details on how pre-keys are signed and how the signature can be verified.

49.4 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 licences,
- 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.

50 WebSocket API

When the application is running in the foreground, it continuously stays connected to the Olvid server through a WebSocket. This WebSocket is currently used to receive new message push notifications in a more efficient/responsive way than through Apple and Google's push notification services and also to receive return receipts. Note that push notifications will be sent though both the WebSocket and Apple and Google's services, but return receipts are only sent through the WebSocket.

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.

50.1 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"`	base64-encoded `identity`
	`"token"`	base64-encoded authentication `token`
	`"deviceUid"`	base64-encoded `deviceUid`

In response to this message, the server will send a error message or an acknowledgement message of the following form.

"action": "register"		
direction		server → device
JSON	"identity"	base64-encoded identity
	("err")	(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

50.2 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, the full message is directly sent, for message with attachments, no payload is sent, instructing the application to trigger a /downloadMessagesAndListAttachments.

"action": "message"		
direction		server → device
JSON	"identity"	base64-encoded identity
	("message")	(optional) base64-encoded message payload

The server does not expect any response to this message.

50.3 Owned devices notification

Certain actions on the server may require all your devices to call the /ownedDeviceDiscovery entry point. Typically, one of your devices expiring, the addition of a new device or a modification to your licence. In that case, the server sends the following message to all your devices.

"action": "ownedDevices"		
direction		server → device
JSON	"identity"	base64-encoded identity

The server does not expect any response to this message.

50.4 Keycloak notification

When a Keycloak server needs an `identity` to come 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.

50.5 Push topic notification

When a Keycloak server needs a group of users to come 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.

50.6 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

The timestamp is set by the server during the upload return receipt entry point execution (see Section 49.1).

50.7 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.

51 Transfer WebSocket API

The transfer WebSocket API is used only when transfering an `identity` from a device to another in a multi-device setting. This API is used to connect the two devices to one another and relay messages between them. As for the main WebSocket API, all messages transmitted through this API are formatted in JSON and must contain an `"action"` key determining the prupose of the message. 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

51.1 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"`	a unique `String` identifier for this connection
	`"sessionNumber"`	a unique 8-digit `int` the user enters on the target device

51.2 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 51.4).

`"action": "target"`		
direction		device → server
JSON	`"sessionNumber"`	the 8-digit `int` that was displayed on the source device
	`"payload"`	a first `byte[]` payload to relay to the source device

The device does not expect any response to this message.

51.3 Relay request

Comunications 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. 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.

51.4 Relayed message

After a `"transfer"` message or a `"relay"` request, the server send the following message to the peer, if it is still connected.

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

52 Push Notifications Content

52.1 Android - Firebase push notifications

On Android, when the sever receives a message for a registered `identity` (i.e. an `identity` for which a device succesfully called the register push notification entry point describerd in Section 46.1), it sends a "background" notification (i.e. a notification without a title and body) to each registered device. The structure of the push notification is the following.

```
{
  "token": "[token]",
  "data": {
    "identity": "[identity_mask]"
  }
}
```

Here `token` is the push notification token received from the Firebase service on the device, and `identity_mask` is a random identifier chosen by the device to mask the real `identity` of the user (see Section 46.1), sent as an hexadecimal string.

If the device is currently active, with the application in foreground, another notification is sent through the WebSocket (see Section 50.2).

52.2 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 to 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 44.1):

- protocol messages without an application payload, triggering only the background notification
- application messages with something to display to the user, triggering both notifications
- voip notifications, triggering an incoming call dialog in CallKit

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 46.1), sent as an 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 46.1), sent as an hexadecimal string
- message_uid is the unique message identifier on the server, sent as an hexadecimal string
- header is the encrypted header received by the server for this device, sent as a base64 string (see Section 44.1)
- content is the encrypted message payload received by the server, sent as a base64 string (see Section 44.1)
- extended_content is the extended message content and is only included if there is room. 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 set its type to voip.

```
{
  "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. VoIP notifications are limited to 5kB, which is why SDP are gzipped before being sent.

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] 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 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 $\#E'(\mathbf{F})/q'$ 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 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 $\mathbf{F}_p$, then $A^2 - 4 = \frac{16d}{(1-d)^2}$ 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 $\mathbf{F}_q$, 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 bellow), those problematic points can be dealt with.

A.2.1 Montgomery Ladder for Edwards Curve

Given a point P on a Montgomery curve, it is possible to restrict to x-coordinate only computations 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 allows to compute 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 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

$$\begin{aligned}U_{2Q} &= (U_Q + W_Q)^2(U_Q - W_Q)^2\\ W_{2Q} &= 4U_QW_Q\left((U_Q - W_Q)^2 + \frac{A+2}{4}(4U_QW_Q)\right)\end{aligned}$$

Note that $4U_QW_Q = (U_Q + W_Q)^2 - (U_Q - W_Q)^2$ and that $\frac{A+2}{4}$ can be precomputed. Thus the addition costs 4 field multiplications and 2 field squaring, while the doubling costs 3 field multiplications and 2 field squaring.

A.2.4 The Ladder on an Edwards Curve

As we will see in the next section, the curve we consider in these specifications are birationally equivalent to a Montgomery curve, 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 F_p with

$$p = 1091123632769611904427110903691495516763303076461182045177715113305362531563 71,$$

defined by

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

where

$$d = 39384817741350628573161184301225915800358770588933756071948264625804612259721.$$

A.3.2 Curve25519

Curve25519 is a Montgomery curve over the prime field 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 to restrict to x-coordinate 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

$$x^2 + y^2 = 1 + \frac{121665}{121666}x^2y^2.$$