Specifications of Olvid

–

Application and Server

Date	May 9, 2020
Number of Pages	122
Written by	Dr. Thomas Baignères Dr. Matthieu Finiasz

Contents

About this document 7

I Notations and Conventions 10

1 Notation, Procedures Default Values, and Conventions 10

2 Names, Lengths, etc. 11

II Cryptographic Primitives 12

3 Preliminaries 12
3.1 Exposed vs. Internal Primitives 12
3.2 Cryptographic Keys 12
3.2.1 Symmetric Keys 13
3.2.2 Public Keys 14
3.2.3 Private Keys 15
3.3 List of Key Byte Identifiers 16

4 Hash Function 16
4.1 SHA-256 17

5 Block Cipher 17
5.1 AES-256 18

6 Symmetric Encryption 19
6.1 AES-256 in CTR Mode 21

7 Message Authentication Code 23
7.1 HMAC with SHA-256 24

8 Key Derivation Function (KDF) 27
8.1 KDF Based on SHA-256 27

9 Commitment 28
9.1 Commitment based on SHA-256 29

10 Pseudorandom Generator 30
10.1 FIPS 800-90A HMAC_DRBG Based on SHA-256 31

11 Authenticated Encryption 33
11.1 Encrypt-then-Mac with AES-256 and HMAC with SHA-256 35

12 Edwards Curves 36
12.1 Curve25519 42
12.2 MDC 43

13 Public and Private Keys over Edwards Curves 43

14 Signature 46
14.1 Signature Key Generation over Curve25519 51
14.2 Signature Key Generation over MDC 51

15 Authentication 52
15.1 Authentication over Curve25519 58
15.2 Authentication over MDC 58

16 Key Encapsulation Mechanism (KEM) 59
16.1 KEM over Curve25519 64
16.2 KEM over MDC 64

17 Cryptographic Identity 65

18 Owned Cryptographic Identity 67

III Encodings 70

19 Encoding Structure 70
19.1 Byte Identifiers 70
19.2 Content Byte-Length 70

20 Byte Representation of Integers and Lengths 71
20.1 32-bit Unsigned Integer 71
20.2 64-bit Unsigned Integer 71
20.3 64-bit Signed Integer 71
20.4 Unsigned Big Integer 72
20.5 Signed Big Integer 72
20.6 Lengths 72

21 Encodings Procedures 72
21.1 Common Encoding/Decoding Rules 73
21.2 Encoding an Array of Bytes 73
21.3 Encoding a String 73
21.4 Encoding a 64-bit Integer 74
21.5 Encoding a Boolean 74
21.6 Encoding an Unsigned Big Integer 74
21.7 Packing Elements and Encoding a List 75
21.8 Encoding a Dictionary 76
21.9 Encoding a Cryptographic Key 77
21.10Encoding a Symmetric Key 77
21.11Encoding a Public Key 78
21.12Encoding a Private Key 78

IV Message Structure and Communication Channels 80

22 General Message Structure 80

22.1 Protocol Message Structure . . . 81
22.2 Application Message Structure . . . 81

23 Encryption **82**
23.1 Message Encryption . . . 82
23.2 Attachment Encryption . . . 82
23.3 Return Receipt Encryption . . . 82

V Cryptographic Protocols **83**

24 Trust Establishment Protocol with SAS **84**
24.1 Purpose and High Level View . . . 84
24.2 Cryptographic Details . . . 84

25 Channel Creation with Contact Device Protocol **87**
25.1 Purpose and High Level View . . . 87
25.2 Cryptographic Details . . . 87

26 Device Discovery Protocol **89**
26.1 Purpose and High Level View . . . 89
26.2 Cryptographic Details . . . 89

27 Contact Mutual Introduction Protocol **89**
27.1 Purpose and High Level View . . . 89
27.2 Cryptographic Details . . . 92

28 Identity Details Publication Protocol and Contact Picture Download Child Protocol **92**
28.1 Purpose and High Level View . . . 92
28.2 Cryptographic Details . . . 92

29 Group Invitation Protocol **94**
29.1 Purpose and High Level View . . . 94
29.2 Cryptographic Details . . . 94

30 Group Management Protocol **94**
30.1 Purpose and High Level View . . . 94
30.2 Cryptographic Details . . . 95

31 Oblivious Channel Management Protocol **95**
31.1 Purpose and High Level View . . . 95
31.2 Cryptographic Details . . . 95

32 Full Ratchet Protocol **95**
32.1 Purpose and High Level View . . . 95
32.2 Cryptographic Details . . . 95

VI Keys and Contacts Backup **97**

33 Backup Seed **97**
33.1 Seed Format 97
33.2 Key Derivation 98

34 Backup Contents **98**
34.1 JSON Structure 98
34.2 Backup Encryption 99
34.3 Backup Decryption 99

35 Backup Restore **100**

VII Olvid Server API **101**

36 Server Authentication API **101**
36.1 Get authentication challenge 102
36.2 Authenticate and get client session token 102

37 Message Upload API **103**
37.1 Get proof of work challenge 103
37.2 Upload message and get UID 104
37.3 Refresh private upload signed urls 105
37.4 Cancel attachment upload 105

38 Message Download API **106**
38.1 Download messages and list attachments 106
38.2 Refresh private download signed urls 107
38.3 Delete message and attachments 107

39 Other REST API Entry Points **108**
39.1 Register push notification 108
39.2 Device discovery 110
39.3 Unregister push notification 110
39.4 Upload return receipt 110
39.5 Put user data 111
39.6 Get user data 112

40 WebSocket API **112**
40.1 Device registration 112
40.2 Message notification 113
40.3 Return receipt download 113
40.4 Return receipt deletion 114

41 Push Notifications Content **114**
41.1 Android - Firebase push notifications 114
41.2 iOS - Apple push notifications 115

Appendices **119**

A Elliptic Curves **119**

A.1 Edwards Curves 119
A.2 Equivalence with Montgomery Curves 119
A.2.1 Montgomery Ladder for Edwards Curve 120
A.2.2 Montgomery Ladder 120
A.2.3 The Ladder on a Montgomery Curve 121
A.2.4 The Ladder on an Edwards Curve 121
A.3 The Curves We consider in These Specifications 121
A.3.1 Million Dollar Curve 122
A.3.2 Curve25519 122

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 receiveing a message, the sever 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 “honnest 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 a 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 26).

Until multi-device is implemented, a single `deviceUid` can be registered on the server for a given `identity`. Changing `deviceUid` (like after a backup restore as described in Section 35) requires to "kick" any other `deviceUid` (see the device registration server entry point in Section 39.1).

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.

API key. Accessing the Olvid server requires a valid API key. Users may use the Olvid application for free using an enbedded "hardcoded" API key which gives access to the general public free features. Paying users will receive a specific API key allowing to unlock additional features on the server (VoIP calls, multi-device, etc.), and in the application (after verification of the key with the server).

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	$\mathbb{Z}$ $\mathbb{Z}$	a $b = 13$
out	$\mathbb{Z}$	c

2 Names, Lengths, etc.

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

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

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

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

When s is a string, we denote by

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

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

When x is an unsigned big integer, we denote by

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

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

Part II
Cryptographic Primitives

In this part, we describe in full 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 specification, 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	$\{0,\dots,255\}$	`algoClassByteId`
	$\{0,\dots,255\}$	`algoImplemByteId`
	Dictionary	`dict`
	$\{0,\dots,255\}$	`encodingByteId`

We denote by

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

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

We denote by

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

the call to the above SymmetricKey initializer.

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

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

We denote by

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

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

We denote by

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

the call to the above PublicKey initializer.

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

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

We denote by

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

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	$\{0, \dots, 255\}$	algoClassByteId
	$\{0, \dots, 255\}$	algoImplemByteId
	Dictionary	dict

We denote by

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

the call to the above PrivateKey initializer.

```
1: procedure PrivateKey(algoClassByteId, algoImplemByteId, dict)
```

2: encodingByteId ← 0x92
3: CryptographicKey(algoClassByteId, algoImplemByteId, dict, encodingByteId)
4: **end procedure**

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

We denote by

$$\text{PrivateKey}(\texttt{b}) \rightarrow \text{privKey}$$

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	$\{0,\dots,255\}^*$	`m`
out	$\{0,\dots,255\}^{\ell_h}$	`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 [14].

5 Block Cipher

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

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

We denote by

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

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

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

We denote by

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

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

5.1 AES-256

We denote by AES256 the concrete subtype of E allowing to encrypt with AES-256.

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

The specifications of the encryption procedure of AES256 are available in [10].

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

The specifications of the decryption procedure of AES256 are available in [10].

6 Symmetric Encryption

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

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

We denote by

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

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	$\{0,\dots,255\}^*$ SymEncKey $\{0,\dots,255\}^{\ell_{iv}+\ell_n}$	`m` symEncKey $\mathtt{iv} \parallel \mathtt{n}$
out	$\{0,\dots,255\}^*$	`c`

We denote by

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

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

We denote by

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

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

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

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

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

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

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

the call to the AES256CTRKey initializer.

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

AES256CTRKey (Initializer)		
parameters		None
in	$\{0, \ldots, 255\}^*$	`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	$\{0,\ldots,255\}^*$ AES256CTRKey $\{0,\ldots,255\}^8$	m symEncKey n
out	$\{0,\ldots,255\}^*$	c

1: **procedure** AES256CTR.encrypt(m, symEncKey, n)
2: encoded_raw ← symEncKey.dict["enckey"]
3: k ← decodeBytes(encoded_raw)
4: ℓ ← length(m)
5: enc ← AES256.encrypt
6: let s $\in \{0,\ldots,255\}^\ell$ be the ℓ 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) $\in \{0,\ldots,255\}^{8+\ell}$
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).

SymEnc.decrypt		
parameters		$\ell_{iv} = 0,\ \ell_n = 8$
in	$\{0,\ldots,255\}^*$ AES256CTRKey	c symEncKey
out	$\{0,\ldots,255\}^*$	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 $\perp$ **end if**
5: let ℓ = len(c) $- 8$ and parse c as (n, c0) $\in \{0,\ldots,255\}^8 \times \{0,\ldots,255\}^\ell$
6: enc ← AES256.encrypt
7: let s $\in \{0,\ldots,255\}^\ell$ be the ℓ first bytes of enc(n$\|\underline{0}$, k) $\|$ enc(n$\|\underline{1}$, k) $\|$ 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(ℓ_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(ℓ_c)
2: **if** $\ell_c < 8$ **then** return $\perp$ **end if**
3: return $\ell_c - 8$
4: **end procedure**

7 Message Authentication Code

MAC is one of 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	$\{0, \ldots, 255\}$ 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	$\{0, \ldots, 255\}^*$ MACKey	m macKey
out	$\{0, \ldots, 255\}^{\ell_t}$	t

The call

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

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

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

the call to the HMACWithSHA256Key initializer.

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

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

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

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

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

HMACWithSHA256.generateKey		
parameters		None
in	PRNG	prng
out	HMACWithSHA256Key	hmacKey

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

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

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

1: **procedure** HMACWithSHA256.compute(m, macKey)
2: **if** macKey is not a HMACWithSHA256Key **then** return $\perp$ **end if**
3: encoded_raw ← macKey.dict["mackey"]
4: k ← decodeBytes(encoded_raw)
5: opad ← $(92, 92, \ldots, 92) \in \{0, \ldots, 255\}^{64}$
6: ipad ← $(54, 54, \ldots, 54) \in \{0, \ldots, 255\}^{64}$
7: $k_o = (k \| 0 \cdots 0) \oplus \text{opad} \in \{0, \ldots, 255\}^{64}$
8: $k_i = (k \| 0 \cdots 0) \oplus \text{ipad} \in \{0, \ldots, 255\}^{64}$
9: return $\text{SHA256}(k_o \,\|\, \text{SHA256}(k_i \,\|\, m))$
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

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

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	$\{0, \ldots, 255\}^*$ SymmetricKey subtype	seed T
out	T	k

```
1: procedure KDFFromPRNGWithHMACWithSHA256.compute(seed, T)
```

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

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

9 Commitment

Commitment schemes within these specification are subtypes of Commitment.

Commitment.commit		
parameters		

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

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

We denote by

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

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

Commitment.open		
parameters		

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

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

We denote by

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

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 [15, p.51] using SHA-256.

CommitmentWithSHA256.commit		
parameters		

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

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

We denote by

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

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

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

We denote by

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

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

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

$$\texttt{prng.state}$$

the internal state of this instance.

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

We denote by

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

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

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

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

We denote by

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

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

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

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:

1: **procedure** prng.bigInt(n)
2: Let l be the smallest integer such that $n \leq 2^l$ and $\ell = \lceil l/8 \rceil$
3: Let m be the smallest integer in $\{1, 3, 7, 15, 31, 63, 127, 255\}$ s.t. $m \geq \lfloor n/256^{\ell-1} \rfloor$
4: **while** True **do**
5: s = prng.bytes(ℓ)
6: $s[0] = s[0]$ and m
7: $r = \sum_{i=0}^{\ell-1} s[\ell - 1 - i] \cdot 256^i$
8: If $r < n$, return r
9: **end while**
10: **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 $\in \{0, \ldots, 255\}^{32}$
- prng.state.v $\in \{0, \ldots, 255\}^{32}$

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

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

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

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

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

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

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

```
1: procedure prng.bytes(ℓ)
2:     k ← self.state.k
3:     v ← self.state.v
4:     hmacKey ← HMACWithSHA256Key(k)
5:     s ← [ ]
6:     while len(s) < ℓ do
7:         v ← HMACWithSHA256.compute(v, hmacKey)
8:         s ← s ∥ v
9:     end while
10:    self.update([ ])
11:    self.state.k ← k
12:    self.state.v ← v
13:    truncate s to its first ℓ bytes
14:    return s
15: 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

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

the call to the AuthEncKey initializer.

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

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

The call

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

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	$\{0,\ldots,255\}^*$ AuthEncKey	c authEncKey
out	$\{0,\ldots,255\}^*$	m

The call

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

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

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

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`

```
1: procedure AES256CTRHMACSHA256Key(b)
2:     if length(b) ≠ 64 then return ⊥ end if
3:     let b_1 denote the first 32 bytes of b
4:     let b_2 denote the last 32 bytes of b
5:     dict["mackey"] ← encodeBytes(b_1)
6:     dict["enckey"] ← encodeBytes(b_2)
7:     AES256CTRHMACSHA256Key(dict)
8: end procedure
```

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

```
1: procedure AES256CTRHMACSHA256.generateKey(seed)
2:     kdf ← KDFFromPRNGWithHMACWithSHA256
3:     return kdf.compute(seed, AES256CTRHMACSHA256Key)
4: end procedure
```

12 Edwards Curves

All the elliptic curves we consider within these specifications are Edwards curves [11] 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 the subgroup generated by G,
- curve.ν: the lcm of the cofactor $\#E(\mathbf{F}_p)/q$ of the curve.

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

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

curve.isOnCurve		
parameters		None
in	$\mathbb{N} \times \mathbb{N}$	(x, y)
out	$\{\texttt{True}, \texttt{False}\}$	`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: $(p, d, G, q, \nu) \leftarrow$ 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: $(p, d, G, q, \nu) \leftarrow$ 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 ← 1
10:        repeat g ← g + 1 until g^((p−1)/2) mod p ≠ 1
11:        t ← p − 1 and s ← 0
12:        repeat t ← t/2 and s ← s + 1 until t mod 2 ≠ 0
13:        e ← 0
14:        for i ← 2 to s do
15:            if (x_2 g^(−e))^((p−1)/2^i) mod p ≠ 1 then e ← 2^(i−1) + e end if
16:        end for
17:        x ← g^(−t e/2) x_2^((t+1)/2) mod p
18:    end if
19:    return (x, −x mod p)
20: end procedure
```

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

We know from the discussion of Section A.2.1 that it 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 $n_{\ell-1} = 1$ is the most significant bit), the following procedure returns the y-coordinate y_n of the point $nP \in E$ (note that in practice, one should precompute c and perform the main loop more efficiently to reduce the number of field squarings and multiplications):

```
1: procedure curve.scalarMultiplication(n, y)
2:     (p, d, G, q, ν) ← curve.parameters
3:     if n = 0 or y = 1 then return 1 end if
4:     if y = −1 then return 1 − 2 × (n mod 2) end if
5:     c ← (1 − d)^(−1) mod p,
6:     u_P ← (1 + y) mod p and w_P ← (1 − y) mod p
7:     u_Q ← 0, w_Q ← 0, u_R ← u_P, and w_R ← w_P
8:     for i ← ℓ down to 1 do
9:         t_1 ← (u_Q − w_Q)(u_R + w_R) mod p
10:        t_2 ← (u_Q + w_Q)(u_R − w_R) mod p
11:        u_{Q+R} ← w_P (t_1 + t_2)^2 mod p
```

12: $w_{Q+R} \leftarrow u_P (t_1 - t_2)^2 \bmod p$
13: **if** $n_{i-1} = 0$ **then**
14: $t_3 \leftarrow (u_Q + w_Q)^2 \bmod p$
15: $t_4 \leftarrow (u_Q - w_Q)^2 \bmod p$
16: $t_5 \leftarrow t_3 - t_4 \bmod p$
17: $u_{2Q} \leftarrow t_3 t_4 \bmod p$
18: $w_{2Q} \leftarrow t_5 (t_4 + ct_5) \bmod p$
19: $(u_Q, w_Q) \leftarrow (u_{2Q}, w_{2Q}) \bmod p$ and $(u_R, w_R) \leftarrow (u_{Q+R}, w_{Q+R}) \bmod p$
20: **else**
21: $t_3 \leftarrow (u_R + w_R)^2 \bmod p$
22: $t_4 \leftarrow (u_R - w_R)^2 \bmod p$
23: $t_5 \leftarrow t_3 - t_4 \bmod p$
24: $u_{2R} \leftarrow t_3 t_4 \bmod p$
25: $w_{2R} \leftarrow t_5 (t_4 + ct_5) \bmod p$
26: $(u_Q, w_Q) \leftarrow (u_{Q+R}, w_{Q+R}) \bmod p$ and $(u_R, w_R) \leftarrow (u_{2R}, w_{2R}) \bmod p$
27: **end if**
28: **end for**
29: return $(u_Q - w_Q)(u_Q + w_Q)^{-1} \bmod p$
30: **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:

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

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

We denote by

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

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

Denoting $n = (n_{\ell-1}n_{\ell-2} \ldots n_1n_0)$ the binary representation of n (where $n_{\ell-1} = 1$ is the most significant bit), the following procedure (based on Montgomery ladder, see Section A.2.2) returns both coordinates of nP on the curve:

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

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

```
1:  procedure curve.mulAdd(a, (x1, y1), b, (x2, y2))
2:      (x3, y3) ← curve.scalarMultiplicationWithX(a, (x1, y1))
3:      if x2 ≠ ⊥ then
4:          (x4, y4) ← curve.scalarMultiplicationWithX(b, (x2, y2))
5:          (x, y) ← curve.pointAddition((x3, y3), (x4, y4))
6:          (x', y') ← (x, y)
7:      else
8:          y4 ← curve.scalarMultiplication(b, y2)
9:          (x4, x'4) ← xCoordinatesFromY(y4, curve)
10:         (x, y) ← curve.pointAddition((x3, y3), (x4, y4))
11:         (x', y') ← curve.pointAddition((x3, y3), (x'4, y4))
12:     end if
13:     return ((x, y), (x', y'))
14: end procedure
```

Note that step 2 fails if (x_1, y_1) is not on the curve. Similarly, step 4 fails if (x_2, y_2) is not on the curve. Moreover, step 9 fails in case y_4 cannot be the y-coordinate of point on the curve. For these reasons, we do not need to explicitly check if (x_1, y_1), (x_2, y_2), (x_4, y_4), and (x'_4, y_4) are on the curve.

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

Given an instance prng of PRNG, we denote by

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

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

```
1: procedure curve.generateRandomScalarAndPoint(prng)
2:     λ ← 2 + prng.bigInt(curve.q − 2)
3:     Q ← curve.scalarMultiplicationWithX(λ, curve.G)
4:     return (λ, Q)
5: 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		

```
1: procedure Curve25519()
2:     self.p ← 2^255 − 19
3:     self.d ← 20800338683988658368647408995589388737092878452977063003340006470870624536394
4:     self.G.x ← 9771384041963202563870679428059935816164187996444183106833894008023910952347
5:     self.G.y ← 46316835694926478169428394003475163141307993866256225615783033603165251855960
6:     self.q ← 7237005577332262213973186563042994240857116359379907606001950938285454250989
7:     self.ν ← 8
8: end procedure
```

12.2 MDC

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

MDC (Initializer)		
parameters		None
in		

1: **procedure** Curve25519()
2: self.$p \leftarrow$ 109112363276961190442711090369149551676330307646118204517771511330536253156371
3: self.$d \leftarrow$ 39384817741350628573161184301225915800358770588933756071948264625804612259721
4: self.$G.x \leftarrow$ 82549803222202399340024462032964942512025856818700414254726364205096731424315
5: self.$G.y \leftarrow$ 91549545637415734422658288799119041756378259523097147807813396915125932811445
6: self.$q \leftarrow$ 27278090819240297610677772592287387918930509574048068887630978293185521973243
7: self.$\nu \leftarrow 4$
8: **end procedure**

13 Public and Private Keys over Edwards Curves

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

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

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

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

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

We denote by

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

the call to the PublicKeyOverEC initializer.

```
1: procedure PublicKeyOverEC(algoClassByteId, algoImplemByteId, curve, P)
2:     if not curve.isOnCurve(P) then return ⊥ end if
3:     self.point ← P
4:     self.y ← P.y
5:     self.curve ← curve
6:     dict["x"] ← encodeBigUInt(P.x mod p, len(curve.p))
7:     dict["y"] ← encodeBigUInt(P.y mod p, len(curve.p))
8:     PublicKey(algoClassByteId, algoImplemByteId, dict)
9: end procedure
```

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

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

We denote by

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

the call to the PublicKeyOverEC initializer.

```
1: procedure PublicKeyOverEC(algoClassByteId, algoImplemByteId, curve, y)
2:     self.point ← ⊥
3:     self.y ← y
```

```
4:     self.curve ← curve
5:     dict["y"] ← encodeBigUInt(y mod p, len(curve.p))
6:     PublicKey(algoClassByteId, algoImplemByteId, dict)
7: end procedure
```

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

PrivateKeyOverEC (Initializer)		
parameters		None
in	$\{0, \ldots, 255\}$	algoClassByteId
	$\{0, \ldots, 255\}$	algoImplemByteId
	EdwardsCurve	curve
	$[0, \ldots, q-1]$	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, λ)
2:     self.scalar ← λ
3:     self.curve ← curve
4:     dict["n"] ← encodeBigUInt(λ  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	$\{0, \dots, 255\}^*$	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	$\{0, \dots, 255\}$	algoImplemByteId
out	EdwardsCurve	curve

We denote by

SignatureOverEC.curveFromAlgoImplemByteId(algoImplemByteId) $\to$ curve

the call to the SignatureOverEC.curveFromAlgoImplemByteId procedure.

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

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

We denote by

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

the call to the SignatureOverEC.algoImplemByteIdFromCurve procedure.

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

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

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

We denote by

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

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.

<table>
<tr><th colspan="3">SignaturePublicKeyOverEC (Initializer)</th></tr>
<tr><td colspan="2">parameters</td><td>None</td></tr>
<tr><td rowspan="2">in</td><td>EdwardsCurve</td><td>curve</td></tr>
<tr><td>$[0, \ldots, p-1]$</td><td>y</td></tr>
</table>

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.

<table>
<tr><th colspan="3">SignaturePrivateKeyOverEC (Initializer)</th></tr>
<tr><td colspan="2">parameters</td><td>None</td></tr>
<tr><td rowspan="2">in</td><td>EdwardsCurve</td><td>curve</td></tr>
<tr><td>$[0, \ldots, q-1]$</td><td>λ</td></tr>
</table>

We denote by

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

the call to the SignaturePrivateKeyOverEC initializer.

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

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

We denote by

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

the call to the SignatureOverEC.generateKeyPair procedure.

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

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

We denote by

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

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 [16, p.166]):

```
1: procedure SignatureOverEC.sign(sk, m, pk, prng)
2:     if sk.curve ≠ pk.curve then return ⊥ end if
3:     curve ← sk.curve
4:     (p, d, G, q, ν) ← curve.parameters
5:
6:     /* The generateKeyPair procedure allows to generate a random scalar and point */
```

```
7:      (pk′, sk′) ← SignatureOverEC.generateKeyPair(prng, curve)
8:
9:      /* Construct the data to hash */
10:     Ay′ ← bytesFromBigUInt(pk′.y mod p, len(p))
11:     Ay ← bytesFromBigUInt(pk.y mod p, len(p))
12:     data ← Ay′||Ay||m
13:
14:     /* Hash and map the digest onto a big unsigned integer */
15:     h ← SHA256(data)
16:     e ← bigUIntFromBytes(h)
17:
18:     /* Sign */
19:     r ← sk′.scalar
20:     a ← sk.scalar
21:     y ← (r − a × e) mod q
22:     z ← bytesFromBigUInt(y, len(p))
23:     σ ← h||z
24:     return σ
25: end procedure
```

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

```
1:  procedure SignatureOverEC.verify(pk, m, σ)
2:      curve ← pk.curve
3:      (p, d, G, q, ν) ← curve.parameters
4:
5:      /* Parse the signature */
6:      if len(σ) ≠ 32 + len(p) then return False end if
7:      Parse h||z ← σ where len(h) = 32
8:
9:      /* Extract the big integers */
10:     e ← bigUIntFromBytes(h)
11:     y ← bigUIntFromBytes(z)
```

```
12:
13:     /* Compute the resulting point(s) */
14:     P ← (pk.point ≠ ⊥ ? pk.point : (⊥, pk.y))
15:     (A1, A2) ← mulAdd(y, G, b, P)
16:
17:     /* Compute the corresponding data(s) to hash */
18:     Ay ← bytesFromBigUInt(pk.y, len(p))
19:     A1y ← bytesFromBigUInt(A1.y, len(p))
20:     A2y ← bytesFromBigUInt(A2.y, len(p))
21:     data1 ← A1y||Ay||m
22:     data2 ← A2y||Ay||m
23:
24:     /* Hash the data(s).  The signature is valid if there is a match.  */
25:     h1 ← SHA256(data1)
26:     h2 ← 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

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

the call to the SignatureOverCurve25519.generateKeyPair procedure.

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

14.2 Signature Key Generation over MDC

We denote by SignatureOverMDC the concrete subtype of SignatureOverEC 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

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

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 specification, authentication 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

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

the call to the AuthenticationOverEC.curveFromAlgoImplemByteId procedure.

```
1: procedure AuthenticationOverEC.curveFromAlgoImplemByteId(algoImplemByteId)
```

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

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

We denote by

$$\mathsf{AuthenticationOverEC.algoImplemByteIdFromCurve}(\mathsf{curve}) \to \mathtt{algoImplemByteId}$$

the call to the AuthenticationOverEC.algoImplemByteIdFromCurve procedure.

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

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

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

We denote by

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

the call to the AuthenticationPublicKeyOverEC initializer.

```
1: procedure AuthenticationPublicKeyOverEC(curve, P)
2:     algoClassByteId ← 0x14
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 AuthenticationPublicKeyOverEC when the full base point is *not* available.

AuthenticationPublicKeyOverEC (Initializer)		
parameters		None
in	EdwardsCurve	curve
	$[0, \ldots, p-1]$	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 ← 0x14
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 AuthenticationPrivateKeyOverEC

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

We denote by

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

the call to the AuthenticationPrivateKeyOverEC initializer.

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

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

We denote by

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

the call to the AuthenticationOverEC.generateKeyPair procedure.

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

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

pk.convertToSignatureKey		
parameters		None
in	None	None
out	SignaturePublicKeyOverEC	pk_σ

We denote by

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

the call to the pk.convertToSignatureKey procedure.

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

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

sk.convertToSignatureKey		
parameters		None
in	None	None
out	SignaturePrivateKeyOverEC	sk_σ

We denote by

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

the call to the sk.convertToSignatureKey procedure.

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

AuthenticationOverEC.solve		
parameters		None

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

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

We denote by

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

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:

```
1: procedure AuthenticationOverEC.solve(sk, challenge, prefix, pk, prng)
2:     if sk.curve ≠ pk.curve then return ⊥ end if
3:     suffix ← prng.bytes(16)
4:     formatted_challenge ← prefix||challenge||suffix
```

5: $\mathsf{pk}_\sigma \leftarrow \mathsf{pk.convertToSignatureKey}()$
6: $\mathsf{sk}_\sigma \leftarrow \mathsf{sk.convertToSignatureKey}()$
7: $\sigma \leftarrow \mathsf{SignatureOverEC.sign}(\mathsf{sk}_\sigma, \texttt{formatted_challenge}, \mathsf{pk}_\sigma, \mathsf{prng})$
8: return $\texttt{suffix} \| \sigma$
9: **end procedure**

AuthenticationOverEC.check		
parameters		None
in	$\{0, \ldots, 255\}^*$	`response`
	$\{0, \ldots, 255\}^*$	`challenge`
	$\{0, \ldots, 255\}^*$	`prefix`
	AuthenticationPublicKeyOverEC	pk
out	$\{\texttt{True}, \texttt{False}\}$	`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** $\mathsf{AuthenticationOverEC.check}(\texttt{response}, \texttt{challenge}, \texttt{prefix}, \mathsf{pk})$
2: **if** $\mathsf{len}(\texttt{response}) < 16$ **then** return $\perp$
3: **end if**
4: Parse `response` as $\texttt{suffix} \| \sigma$ where $\mathsf{len}(\texttt{suffix}) = 16$
5: $\texttt{formatted_challenge} \leftarrow \texttt{prefix} \| \texttt{challenge} \| \texttt{suffix}$
6: $\mathsf{pk}_\sigma \leftarrow \mathsf{pk.convertToSignatureKey}()$
7: return $\mathsf{SignatureOverEC.verify}(\mathsf{pk}_\sigma, \texttt{formatted_challenge}, \texttt{response})$
8: **end procedure**

AuthenticationPublicKeyOverEC.expandCompactKey		
parameters		None
in	$\{0, \ldots, 255\}^*$	`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 ⊥ end if
3:     Parse compactKey as algoImplemByteId||yCoordinate where len(algoImplemByteId) = 1
4:     curve ← AuthenticationPublicKeyOverEC.curveFromAlgoImplemByteId(algoImplemByteId)
5:     if len(compactKey) ≠ 1 + len(curve.p) then return ⊥ end if
6:     y ← bigUIntFromBytes(yCoordinate)
7:     return AuthenticationPublicKeyOverEC(curve, y)
8: end procedure
```

15.1 Authentication over Curve25519

We denote by AuthenticationOverCurve25519 the concrete subtype of AuthenticationOverEC 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

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

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

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

the call to the AuthenticationOverMDC.generateKeyPair procedure.

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

16 Key Encapsulation Mechanism (KEM)

Key Encapsulation 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.

```
1: procedure KEMOverEC.curveFromAlgoImplemByteId(algoImplemByteId)
2:     if algoImplemByteId = 0x00 then return MDC() end if
```

```
3:     if algoImplemByteId = 0x01 then return Curve25519() end if
4:     return ⊥
5: end procedure
```

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

We denote by

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

the call to the KEMOverEC.algoImplemByteIdFromCurve procedure.

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

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

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

We denote by

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

the call to the KEMPublicKeyOverEC initializer.

```
1: procedure KEMPublicKeyOverEC(curve, P)
2:     algoClassByteId ← 0x12
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 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.

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

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

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

We denote by

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

the call to the KEMPrivateKeyOverEC initializer.

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

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

We denote by

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

the call to the KEMOverEC.generateKeyPair procedure.

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

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

We denote by

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

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 [17, p.26]):

```
1: procedure KEMOverEC.encrypt(pk, prng, T)
2:     curve ← pk.curve
3:     (p, d, G, q, ν) ← curve.parameters
4:     repeat r ← prng.bigInt(q) until r ≠ 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	$\{0, \dots, 255\}^*$	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 [17, p.26]):

```
1: procedure KEMOverEC.decrypt(encrypted_key, sk, T)
2:     curve ← sk.curve
3:     (p, d, G, q, ν) ← curve.parameters
4:     if len(encrypted_key) ≠ len(p) then return ⊥ end if
5:     y ← bigUIntFromBytes(encrypted_key)
6:     B_y ← curve.scalarMultiplication(ν, y)
7:     if B_y = 1 then return ⊥ end if
8:     a ← sk.scalar × (ν^-1 mod q) mod q
9:     D_y ← curve.scalarMultiplication(a, B_y)
10:    seed ← c||bytesFromBigUInt(D_y, len(p))
11:    return KDFFromPRNGWithHMACWithSHA256.compute(seed, T)
12: end procedure
```

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

We denote by

$$\mathsf{KEMPublicKeyOverEC.expandCompactKey}(\texttt{compactKey}) \rightarrow \mathsf{KEMPublicKeyOverEC}$$

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

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

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

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 instance 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	$\{0, \ldots, 255\}^*$ AuthenticationPublicKeyOverEC KEMPublicKeyOverEC	serverURL pk_a pk_e

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

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

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

```
1: procedure CryptoIdentity(serverURL, pk_a, pk_e)
2:     self.serverURL ← serverURL
3:     self.publicKeyForAuthentication ← pk_a
4:     self.publicKeyForKEM ← pk_e
5: end procedure
```

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

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

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

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

the call to the cryptoIdentity.getIdentity procedure.

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

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

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

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

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

```
1: procedure CryptoIdentity(identity)
```

```
2:      Parse identity as serverURL || 0x00 || keys. Return ⊥ if this fails.
3:      Check that serverURL is a valid URL
4:      if len(keys) = 0 then return ⊥ end if
5:      authImplemByteId ← keys[0]  // one byte
6:      curve_a ← AuthenticationOverEC.curveFromAlgoImplemByteId(authImplemByteId)
7:      ℓ_a ← len(curve_a.p)
8:      if len(keys) < 2 + ℓ_a then return ⊥ end if
9:      compactAuthKey ← keys[1 ... ℓ_a]  // ℓ_a bytes
10:     pk_a ← AuthenticationPublicKeyOverEC.expandCompactKey(compactAuthKey)
11:     kemImplemByteId ← keys[ℓ_a + 1]  // One byte
12:     curve_e ← KEMPublicKeyOverEC.curveFromAlgoImplemByteId(kemImplemByteId)
13:     ℓ_e ← len(curve_e.p)
14:     if len(keys) ≠ 2 + ℓ_a + ℓ_e then return ⊥ end if
15:     compactKEMKey ← keys[ℓ_a + 2 ... ℓ_a + 2 + ℓ_e − 1]  // ℓ_e bytes
16:     pk_e ← KEMPublicKeyOverEC.expandCompactKey(compactKEMKey)
17:     self.serverURL ← serverURL
18:     self.publicKeyForAuthentication ← pk_a
19:     self.publicKeyForKEM ← pk_e
20: 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	$\{0,\ldots,255\}^*$ AuthenticationPublicKeyOverEC AuthenticationPrivateKeyOverEC KEMPublicKeyOverEC KEMPrivateKeyOverEC MACKey	serverURL pk_a sk_a pk_e sk_e 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

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

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

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

OwnedCryptoIdentity.generateOwnedCryptoIdentity		
parameters		None
in	$\{0,\ldots,255\}^*$ 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

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

the call to the previous procedure.

```
1: procedure OwnedCryptoIdentity.generateOwnedCryptoIdentity(serverURL, prng)
```

```
2:     (pk_a, sk_a) ← AuthenticationOverMDC.generateKeyPair(prng)
3:     (pk_e, sk_e) ← KEMOverCurve25519.generateKeyPair(prng)
4:     key ← HMACWithSHA256.generateKey(prng)
5:     return OwnedCryptoIdentity(serverURL, pk_a, sk_a, pk_e, sk_e, key)
6: end procedure
```

ownedCryptoIdentity.getCryptoIdentity		
parameters		None
in		
out	CryptoIdentity	cryptoIdentity

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

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

the call to the previous procedure.

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

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,

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

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

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

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,

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

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

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

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,

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

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

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

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

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

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 $[\mathtt{b}_0, \mathtt{b}_1, \dots, \mathtt{b}_n]$ before trying to decode any further:

1: **procedure** $\mathsf{Parse}([\mathtt{b}_0, \mathtt{b}_1, \dots, \mathtt{b}_{n-1}])$
2: **if** $n < 5$ **then** return $\perp$ **end if**
3: **if** $\mathtt{b}_0$ is not a known byte identifier **then** return $\perp$ **end if**
4: $\ell \leftarrow \mathsf{uint32FromBytes}([\mathtt{b}_1, \mathtt{b}_2, \mathtt{b}_3, \mathtt{b}_4])$
5: **if** $5 + \ell \neq n$ **then** return $\perp$ **end if**
6: return $(\mathtt{b}_0, [\mathtt{b}_5, \dots, \mathtt{b}_{5+(\ell-1)}])$
7: **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 $\mathtt{b} \in \{0, \dots, 255\}^*$:

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

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

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

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

1: **procedure** $\mathsf{decodeBytes}([\mathtt{b}_0, \mathtt{b}_1, \dots, \mathtt{b}_{n-1}])$
2: $(\mathtt{byteId}, [\mathtt{c}_0, \mathtt{c}_1, \dots, \mathtt{c}_{\ell-1}]) \leftarrow \mathsf{Parse}([\mathtt{b}_0, \mathtt{b}_1, \dots, \mathtt{b}_{n-1}])$
3: **if** $\mathtt{byteId} \neq \mathtt{0x00}$ **then** return $\perp$ **end if**
4: return $[\mathtt{c}_0, \mathtt{c}_1, \dots, \mathtt{c}_{\ell-1}]$
5: **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** $\mathsf{decodeInt}([\mathtt{b}_0, \mathtt{b}_1, \ldots, \mathtt{b}_{n-1}])$
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** $\mathsf{decodeBool}([\mathtt{b}_0, \mathtt{b}_1, \ldots, \mathtt{b}_{n-1}])$
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 $[\mathtt{b}_0, \mathtt{b}_1, \ldots, \mathtt{b}_{n-1}] \in \{0, \ldots, 255\}^n$ assumed to be the encoding of an unsigned big integer, we apply the following procedure:

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

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, $\ldots \in \{0, \ldots, 255\}^*$, the pack procedure is defined as follows:

$$\begin{aligned}&\mathsf{pack}(\mathtt{byteId}, \mathtt{encoded_val1}, \mathtt{encoded_val2}, \ldots) \\ &\qquad \to [\mathtt{byteId}, \mathsf{uint32FromBytes}(\ell), \mathtt{encoded_val1}, \mathtt{encoded_val2}, \ldots] \end{aligned} \tag{2}$$

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 $[\mathtt{b}_0, \mathtt{b}_1, \ldots, \mathtt{b}_{n-1}] \in \{0, \ldots, 255\}^n$ assumed to be the result of the above pack procedure, we apply the following procedure:

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

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

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

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

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

$$\begin{aligned}\mathsf{encodeList}(\texttt{encoded_val1}, \texttt{encoded_val2}, \ldots) \\ \to \mathsf{pack}(\texttt{0x03}, \texttt{encoded_val1}, \texttt{encoded_val2}, \ldots)\end{aligned} \quad (3)$$

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

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

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

The following procedure returns a proper encoding of a dictionary dict respectively associating the keys $\texttt{key1}, \texttt{key2}, \ldots$ with the encoded values $\texttt{encoded_val1}, \texttt{encoded_val2}, \ldots$:

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

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

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

```
4:     if len(encoded_vals) is not even then return ⊥ end if
5:     dict ← empty dictionary
6:     while len(encoded_vals) > 0 do
7:         (encoded_key, val) ← pop the first two elements of encoded_vals
8:         key ← decodeBytes(encoded_key)
9:         dict[key] ← val
10:    end while
11:    return dict
12: 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:

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

In order to decode an array of $n \geq 0$ bytes $[b_0, b_1, \ldots, b_{n-1}] \in \{0, \ldots, 255\}^n$ assumed to be the encoding of a cryptographic key for a specific KeyedAlgo, the following procedure will be systematically used within the decoding of a cryptographic key:

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

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:

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

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

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

In order to decode an array of $n \geq 0$ bytes $[b_0, b_1, \ldots, b_{n-1}] \in \{0, \ldots, 255\}^n$ assumed to be the encoding of a symmetric key for a specific symmetric key algorithm SymKeyedAlgo, we apply the following procedure:

```
1: procedure decodeSymKey([b_0, b_1, ..., b_{n-1}])
2:     (algoClassByteId, algoImplemByteId, dict, byteId) ← preDecodeKey([b_0, b_1, ..., b_{n-1}])
3:     if byteId ≠ 0x90 then return ⊥ end if
4:     return SymKeyedAlgo.SymKeyInit(algoClassByteId, algoImplemByteId, dict)
5: 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 `pubKey.encodingByteId` = `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 $[b_0, b_1, \ldots, b_{n-1}] \in \{0, \ldots, 255\}^n$ assumed to be the encoding of a public key for a specific public key cryptographic algorithm PubKeyedAlgo, we apply the following procedure:

```
1: procedure decodeSymKey([b_0, b_1, ..., b_{n-1}])
2:     (algoClassByteId, algoImplemByteId, dict, byteId) ← preDecodeKey([b_0, b_1, ..., b_{n-1}])
3:     if byteId ≠ 0x91 then return ⊥ end if
4:     return PubKeyedAlgo.PubKeyInit(algoClassByteId, algoImplemByteId, dict)
5: 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 `privKey.encodingByteId` = `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:

```
1: procedure decodeSymKey([b_0, b_1, ..., b_{n-1}])
2:     (algoClassByteId, algoImplemByteId, dict, byteId) ← preDecodeKey([b_0, b_1, ..., b_{n-1}])
3:     if byteId ≠ 0x92 then return ⊥ end if
4:     return PubKeyedAlgo.PrivKeyInit(algoClassByteId, algoImplemByteId, dict)
5: end procedure
```

Part IV
Message Structure and Communication Channels

There are two main categories of messages in Olvid:

- Application messages, containing the text messages and attachments the 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 always sent through what we call Oblivious Channels. 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 25), using symmetric encryption with 1-time keys (see Section 23.1 for encryption details). 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 all the devices of an `identity`.

22 General Message Structure

A message is composed of three 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,

- 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 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 device is possible, but this structure is not well suited for very large groups.

When sending an application message to multiple users (for 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.

Attachments are split in chunks before being sent. See Section 23.2 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 attachment metadata JSON (see below).
- an encoded 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>
}
```

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
  "re": <Message reply>
}
```
`guid` and `go` allow determining which group discussion this message is part of. Message reply is present if the message is a reply to another message.

Message reply:
```
{
  "ssn": int,       // sender sequence number
  "sti": UUID,      // sender thread identifier
  "si": byte[]      // sender identifier
}
```
These elements allow identifying the original message, if not deleted yet.

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

23 Encryption

23.1 Message Encryption

The message itself is always encrypted with an authenticated encryption symmetric primitive. The authenticated encryption key is generated at random when sending the message and wrapped in the headers. The key wrap method of the headers depend on the channel the message is being sent on, but the size of the header is the same for both methods.

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.

Oblivious channel. Oblivious channels use a self ratchet system (in combination with the full ratchet described in Section 32) 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.

23.2 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 implementation always uses chunks of 2MB). 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.

23.3 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.

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 26)
2	Channel Creation with Contact Device Protocol (see Section 25)
3	Device Discovery Child Protocol (see Section 26)
4	Contact Mutual Introduction Protocol (see Section 27)
6	Identity Details Publication Protocol (see Section 28)
7	Contact Picture Download Child Protocol (see Section 28)
8	Group Invitation Protocol (see Section 29)
9	Group Management Protocol (see Section 30)
10	Oblivious Channel Management Protocol (see Section 31)
11	Trust Establishment Protocol with SAS (see Section 24)
12	Trust Establishment Protocol with Mutual Scan (coming soon)
13	Full Ratchet Protocol (see Section 32)

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.

24 Trust Establishment Protocol with SAS

24.1 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}.

24.2 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`.

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

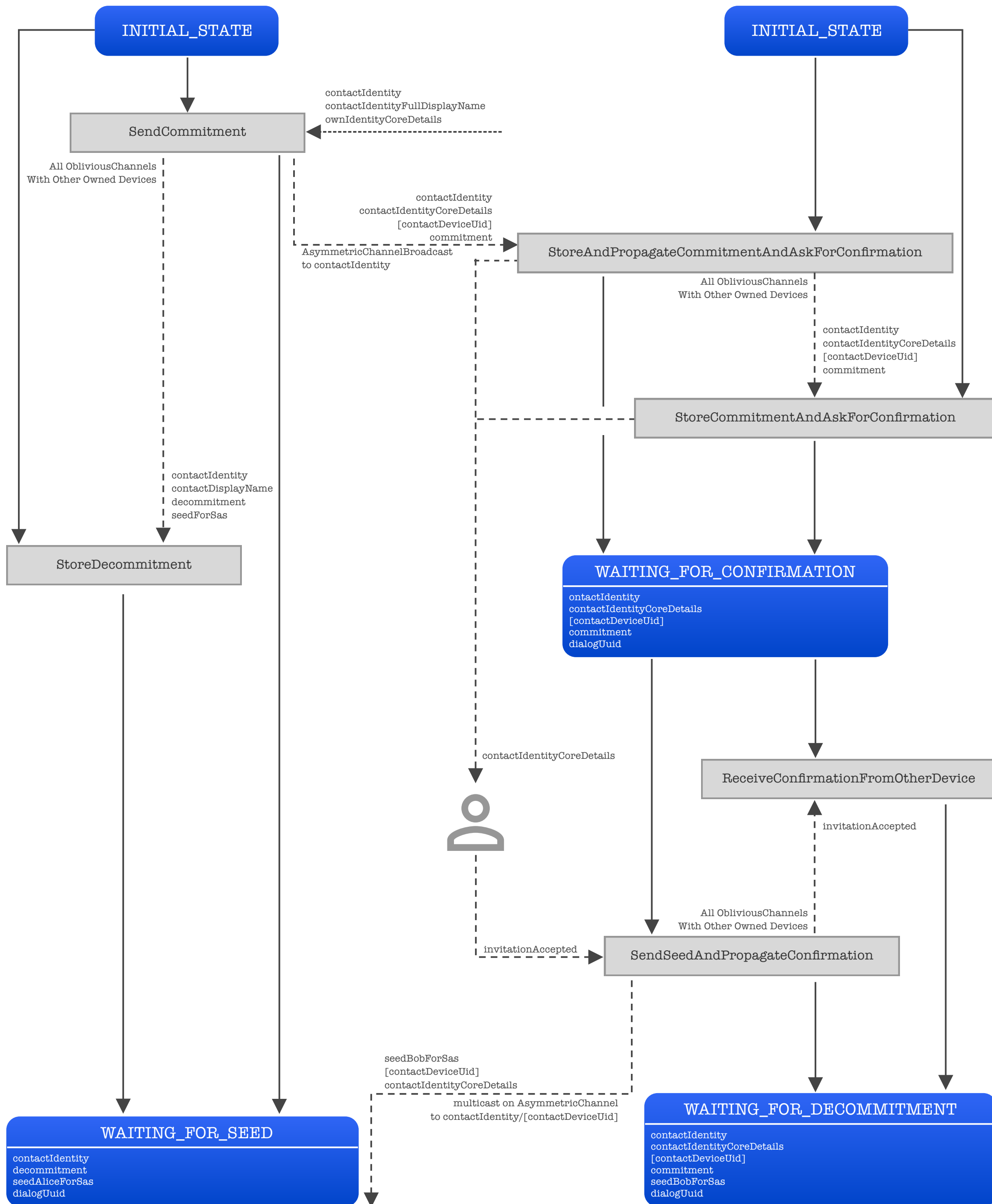

Figure 2: Trust Establishment Protocol with SAS (part 1)

Trust Establishment Protocol with SAS
Part 2

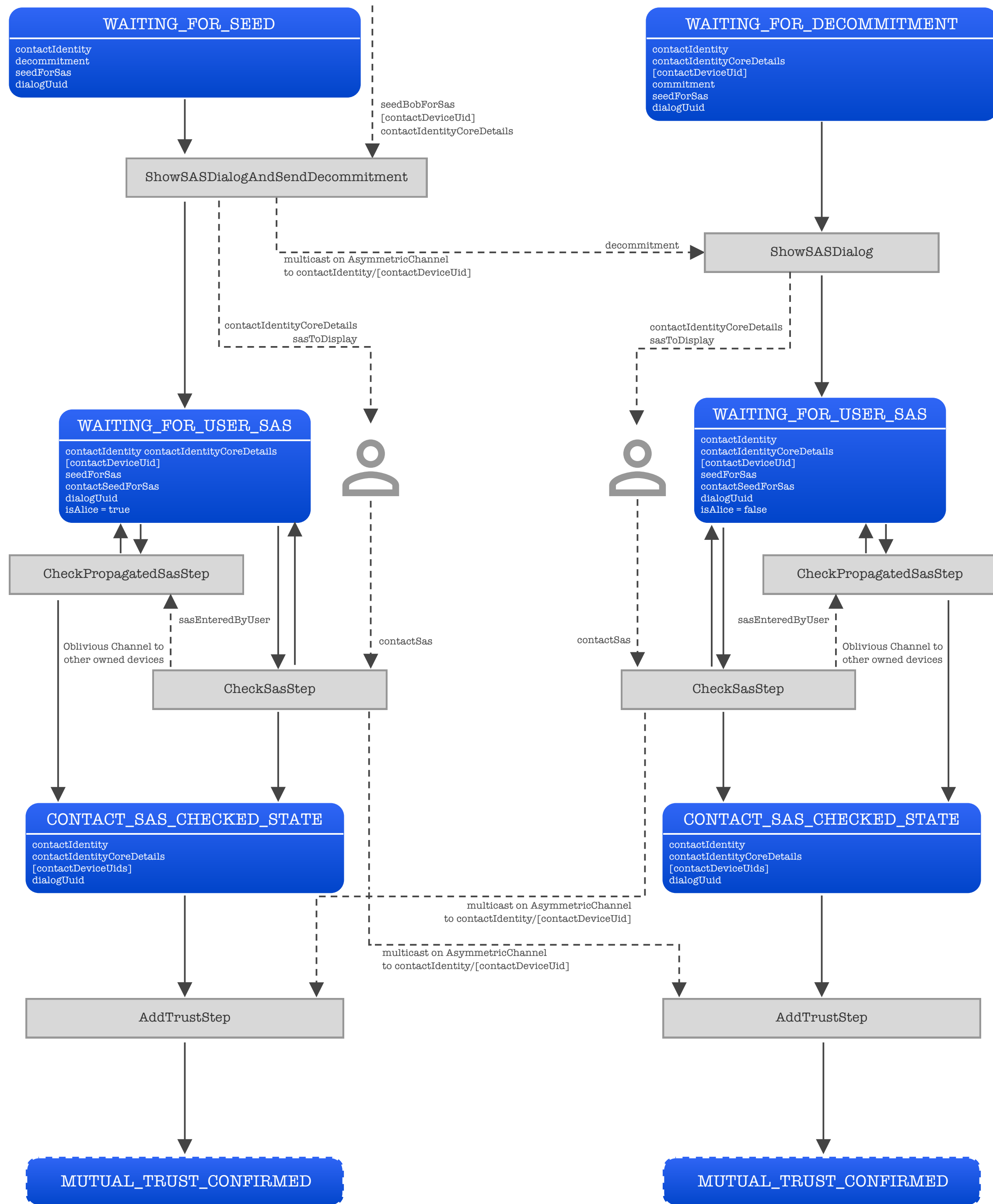

Figure 3: Trust Establishment Protocol with SAS (part 2)

6: **end procedure**

25 Channel Creation with Contact Device Protocol

25.1 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 24, 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 pratice, receiving any message on the channel is also enough to confirm it

25.2 Cryptographic Details

Signature. The signature uses the signature/authentication key inside the `identity` of the party sending the ping. When Alice sends a ping, this signature is computed over the concatenation of:

- a constant prefix "channelCreation"
- the `deviceUid` of Bob
- the `deviceUid` of Alice
- the `identity` of Bob
- the `identity` of Alice
- a random 16-byte padding (also included in the signature)

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.

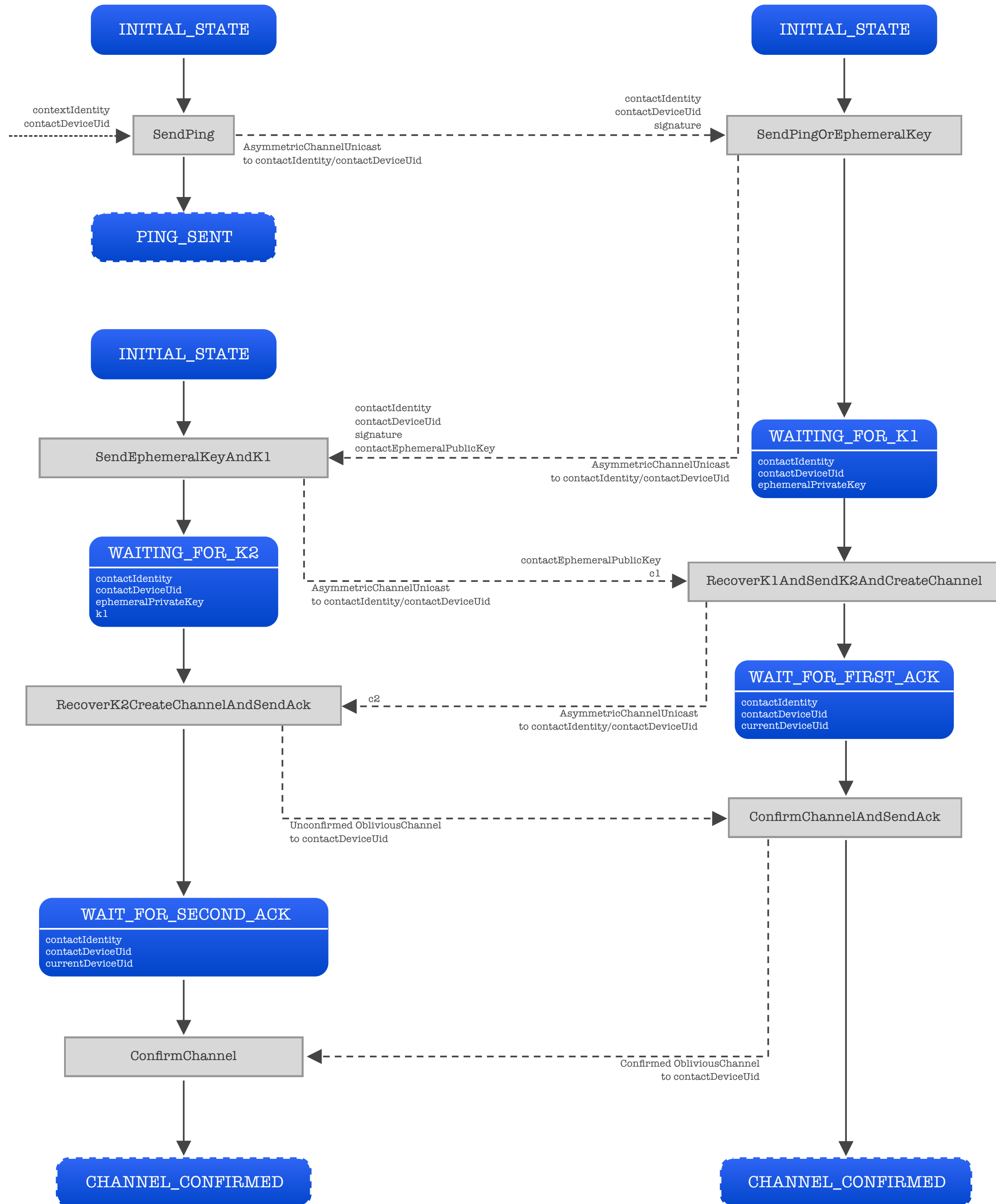

Figure 4: Channel Creation with Contact Device Protocol

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.

26 Device Discovery Protocol

26.1 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. . .

26.2 Cryptographic Details

There is no cryptography involved in this protocol.

27 Contact Mutual Introduction Protocol

27.1 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

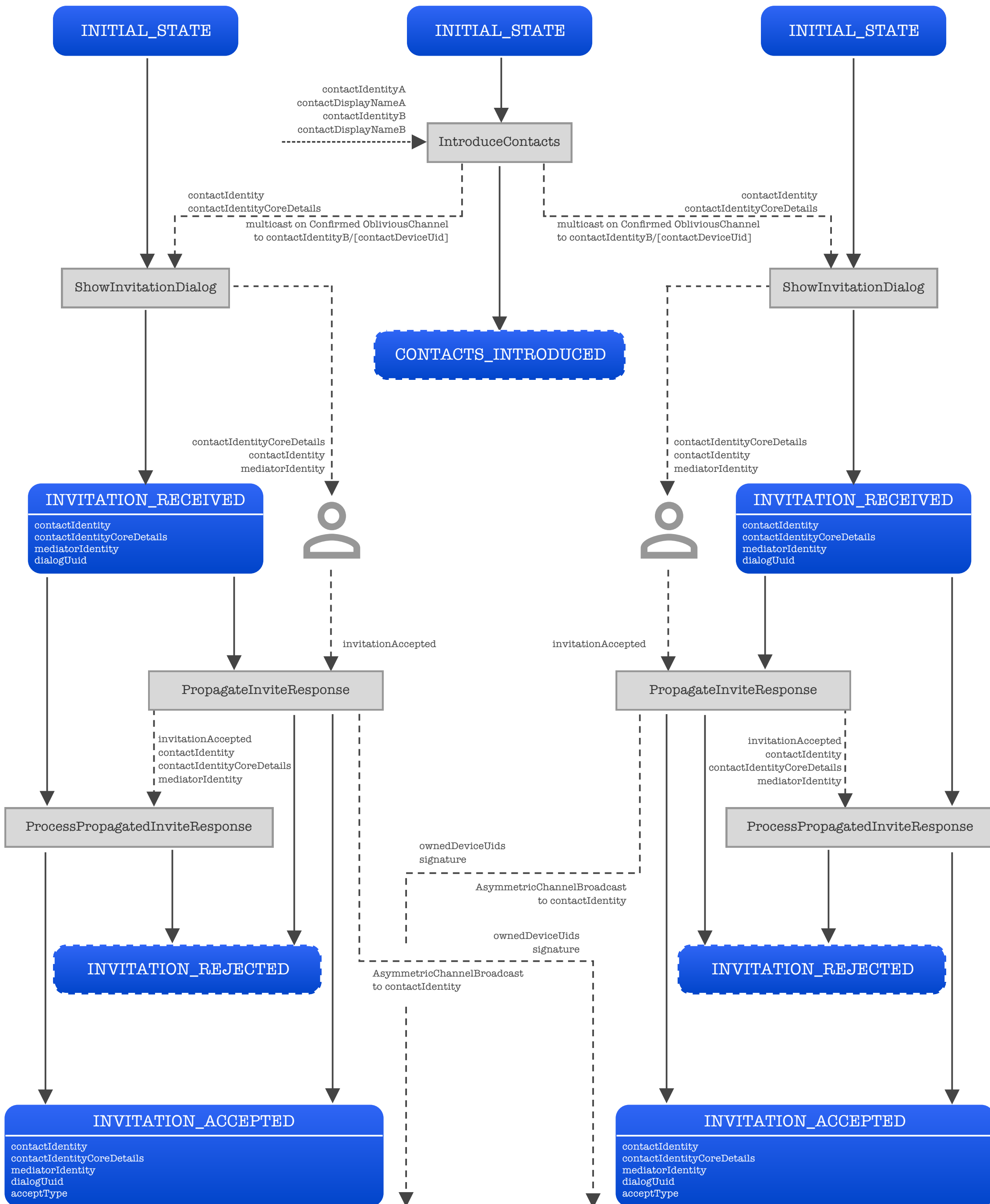

Figure 5: Contact Mutual Introduction Protocol (part 1)

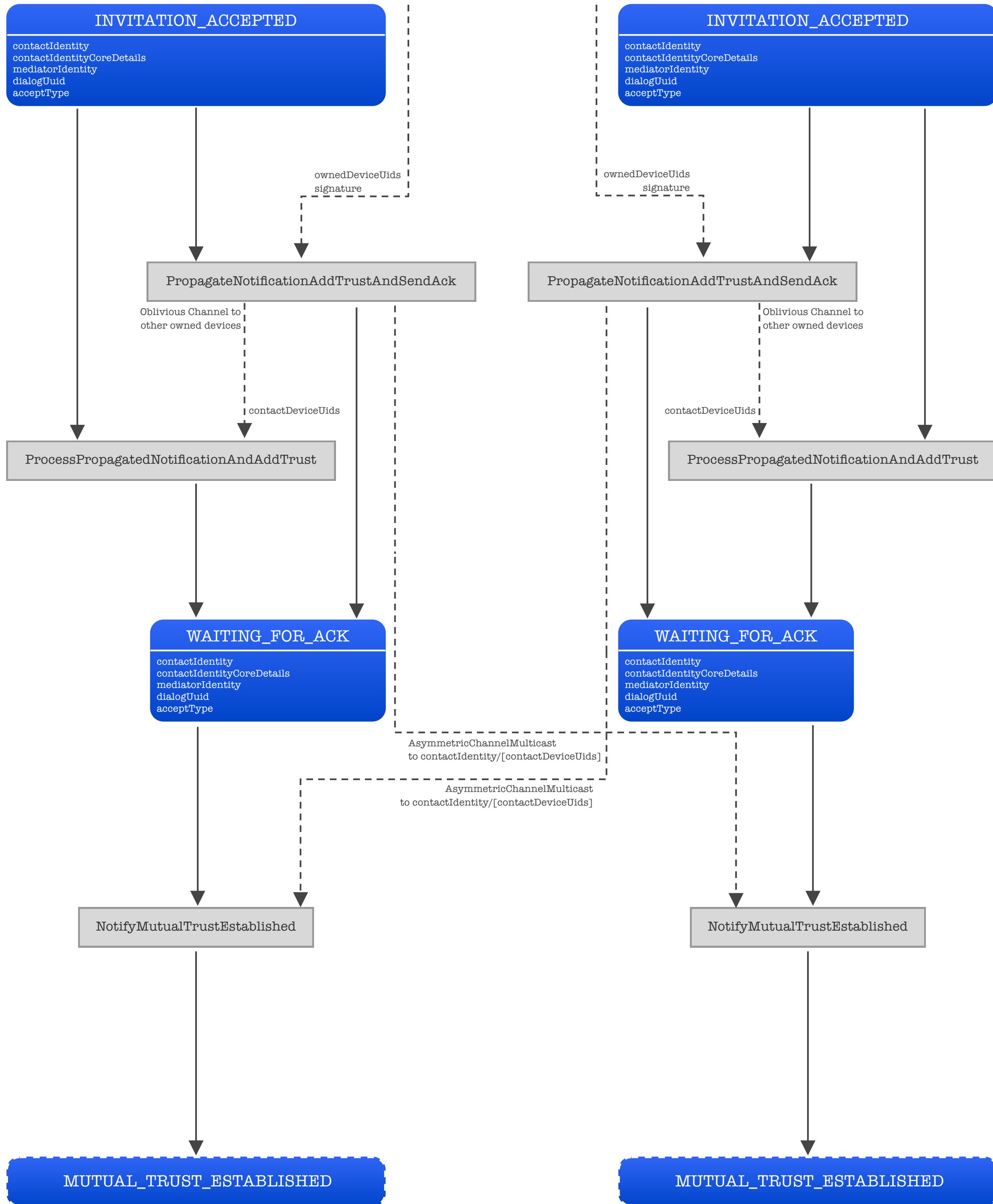

Figure 6: Contact Mutual Introduction Protocol (part 2)

keys by running an instance of the Trust Establishment with SAS protocol (see Section 24).

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.

27.2 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)

28 Identity Details Publication Protocol and Contact Picture Download Child Protocol

28.1 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.

28.2 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.

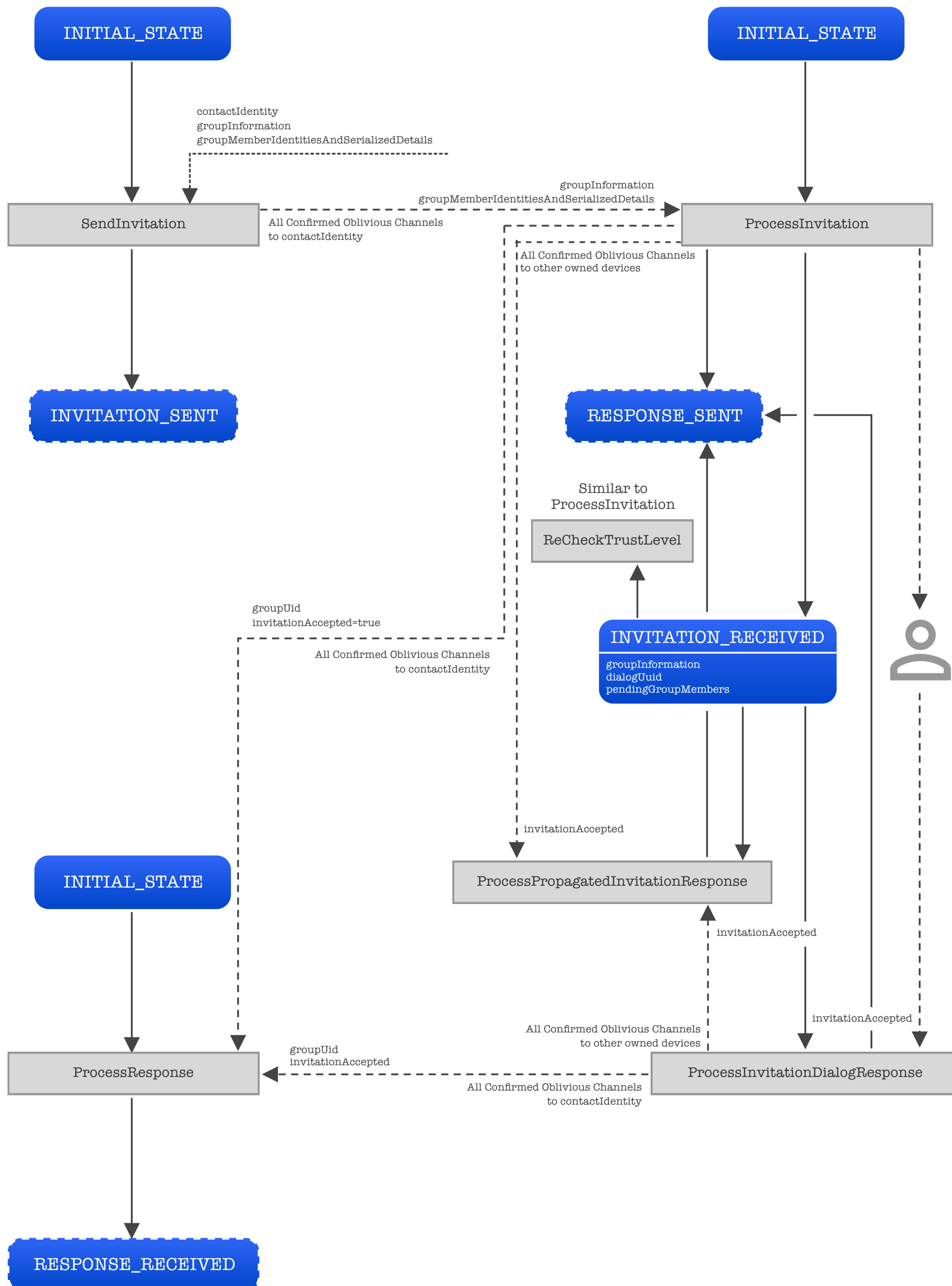

Figure 7: Group Invitation Protocol

29 Group Invitation Protocol

29.1 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 30), 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 27), 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.

29.2 Cryptographic Details

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

30 Group Management Protocol

30.1 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 29)
- Notify group members that the members of the groups have changed (with the corresponding step to process group members change on the member 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.

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. 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.

31 Oblivious Channel Management Protocol

31.1 Purpose and High Level View

This protocol serves the same purpose as the group management protocol (see Section 30), 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.

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.

32 Full Ratchet Protocol

32.1 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 25). 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.

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.

32.2 Cryptographic Details

The cryptography of the full ratchet is exacly the same as in the Channel Creation with Contact Device Protocol (see Section 25), except that no signature is required as users already have an oblivious channel guaranteeing the authenticity of the messages.

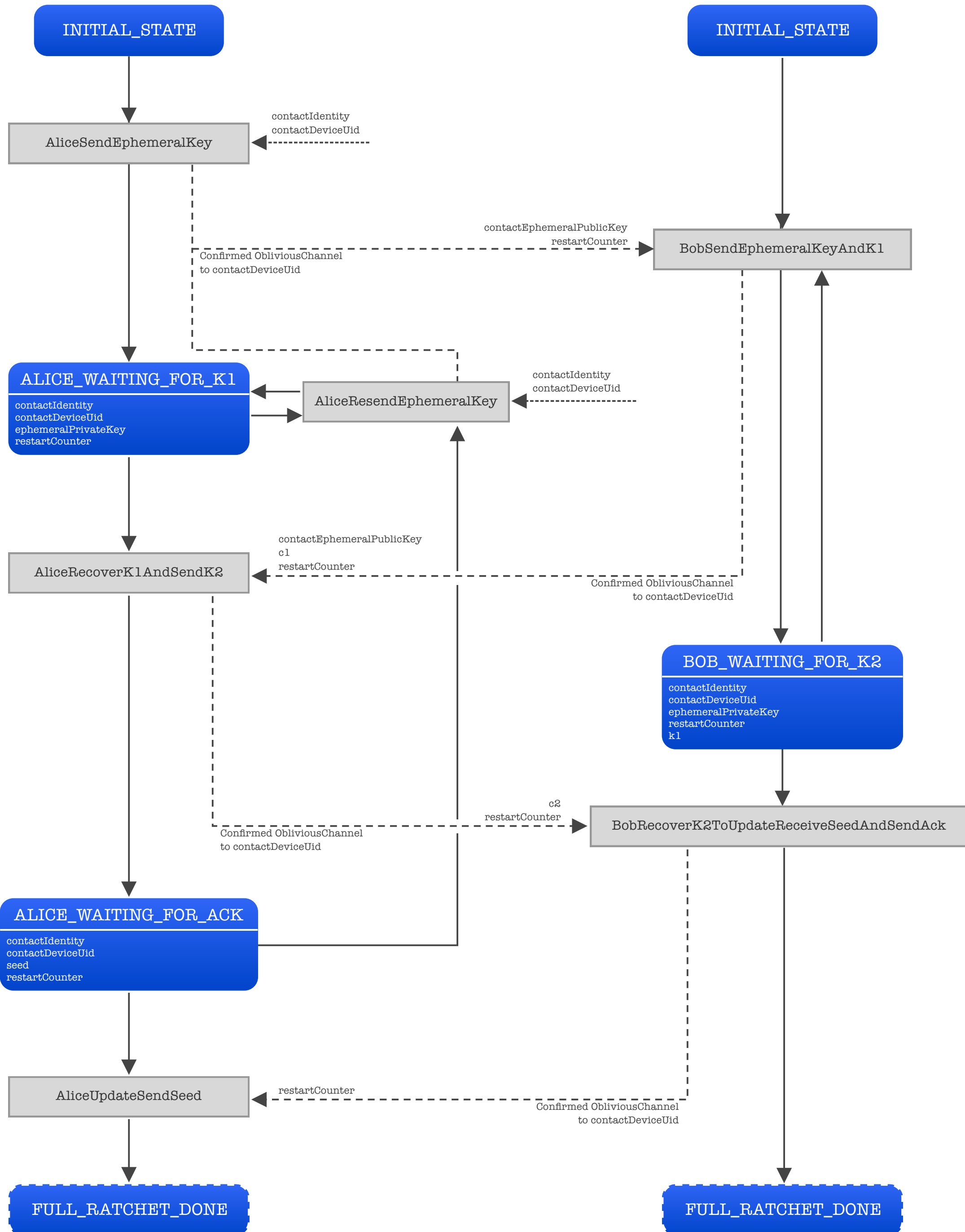

Figure 8: Full Ratchet Protocol

Part VI
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 33.

Then, a backup is a JSON string (formatted as described in Section 34.1) containing dumps from the identity databases. This JSON string is first compressed, then encrypted using the backup key (see Section 34.2) 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).

33 Backup Seed

33.1 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.

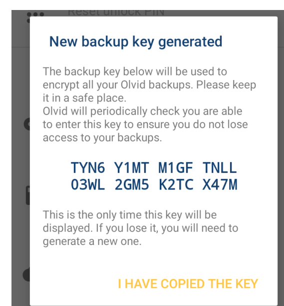

Figure 9: Example of an Olvid backup key.

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.

33.2 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$\|0\dots0$
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.

34 Backup Contents

34.1 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
}
```

34.2 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 33.2) 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

34.3 Backup Decryption

Backup decryption only happens during a restore (see Section 35). 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

35 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 26) is run
- Device discovery adds some `deviceUid` for each contact, which triggers channel creation protocols (see Section 25)
- When a channel is created with a contact:
  - Each user sends their published owned identity details to the other (part of the ack messages of the protocol)
    → during this protocol, if the received details version is lower than what is already in database, a "downgrade" is authorized.
  - For every group owned by the other user, query the latest group members (see Section 30)
  - For every owned group to which the other user belongs, reinvite him and forcibly push the updated group members (see Section 30)
    → again, in this protocol, "downgrade" of the group members and details is possible

Part VII
Olvid Server API

This section describes the different API entry point of the Olvid server. This API is composed of two 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 8, 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`

In addition to this, the server may also send some Pre-Signed S3 URL allowing to directly upload or download a file to or from AWS S3. 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].

36 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.

36.1 Get authentication challenge

The `/requestChallenge` entry point allows a user to request an authentication `challenge` from the server. At the same time, the server validates that the API key presented by the user is valid (and that a license for this specific `identity` is available). The `nonce` sent by the user allows mulitple authentications from the same `identity` simultaneously. It is sent again (see Section 36.2) with the `response` to retrieve the corresponding `challenge`.

/requestChallenge		

in	`identity`
	`byte[32]`
	`UUID`
		identity to authenticate
		`nonce`
		API key

out	`byte[1]`
	`byte[32]`
	`byte[32]`
		return status
		`challenge`
		`nonce`

return statuses		`0x00`: OK `0x07`: unknown API key `0x08`: number of API key licenses exhausted `0xff`: unknown error

36.2 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`
	`byte[80]`
	`byte[32]`
		identity to authenticate
		`response` to the challenge
		`nonce`

out	`byte[1]`
	`byte[32]`
	`byte[32]`
		return status
		`token`
		`nonce`

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)

37 Message Upload API

Uploading a message to the Olvid server happens in three step:

- the user requests a proof of work challenge and solves it
- 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.

37.1 Get proof of work challenge

This entry point allows to retrieve a proof of work `challenge` and its identifier. Once solved, the proof of work `solution` can be used to upload one message.

/getPoWChallenge		
in	-	Nothing

out	`byte[1]`
	`byte[32]`
	`byte[]`
		return status
		proof of work unique identifier
		proof of work `challenge`

return statuses		`0x00`: OK `0xff`: unknown error

37.2 Upload message and get UID

/uploadMessageAndGetUids		
in	`byte[32]`	proof of work unique identifier
	`byte[57]`	proof of work `solution`
	`[encoded_vals]`	list of encoded headers (see details 1 below)
	`byte[]`	encrypted message payload
	`boolean`	whether the message contains an application payload
	`[encoded_vals]`	list of encoded attachment lengths (see details 2 below)
	`[encoded_vals]`	list of encoded attachment chunk lengths (see details 2 below)
out	`byte[1]`	return status
	`byte[32]`	message unique identifier on the server
	`byte[32]`	nonce (see details 3 below)
	`long`	server timestamp
	`[encoded_vals]`	list of private signed urls (see details 4 below)
return statuses		`0x00`: OK `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. **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.
3. **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.
4. **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.

37.3 Refresh private upload signed urls

The private upload signed urls obtained from the upload message 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]`
	`int`
	`byte[32]`
		message unique identifier on the server
		attachment number
		the nonce received when uploading the message

out	`byte[1]` `[encoded_vals]`	return status list of private upload signed urls
return statuses		`0x00`: OK `0x09`: attachment was already deleted from the server `0x0c`: invalid nonce `0xff`: unknown error

The message unique identifier on the server and the nonce are taken from the output of the upload message 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.

37.4 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

38 Message Download API

38.1 Download messages and list attachments

This entry point retrieves all messages for 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 private signed urls allowing to download its attachment chunks directly from AWS S3. This list may be truncated to avoid slowing down this entry point too much.

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`.

/downloadMessagesAndListAttachments		

in	identity
	byte[32]
	byte[32]
		`identity`
		Authentication `token`
		`deviceUid`

out	byte[1] encoded_val ...	return status one encoded value for each message (see description below)
return statuses		0x00: OK 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

- 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 private signed urls for attachment chunks download. This list contains as many element as the number of attachment chunks. If the entry point is taking too long to execute, these urls may be replaced by dummy expired urls requiring to be refreshed immediately.

Note that the messages are always sorted before being returned, from oldest to newest, based on the server timestamp at upload time.

38.2 Refresh private download signed urls

The private download signed urls obtained from the download message and list attachments entry point have an expiration date. Past this date, the url becomes invalid and must 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 private download signed urls
return statuses		0x00: OK 0x09: attachment was already deleted from the server 0xff: unknown error

If the attachment was not marked as deleted (see Section 37.4), this method returns a valid private download signed url for each attachment chunk. On the contrary, if the attachment was marked as canceled, a valid download signed url is returned only for chunks that are actually available on AWS S3. For other chunks, an encoded empty String is returned instead.

38.3 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.

/deleteMessageAndAttachments		
in	identity	identity
	byte[32]	Authentication token
	byte[32]	message unique identifier on the server
	byte[32]	deviceUid
out	byte[1]	return status
return statuses		0x00: OK 0x04: invalid token (re-authentication required) 0xff: unknown error

Note that if the message was already deleted on the server, this entry point returns OK.

39 Other REST API Entry Points

39.1 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 push 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 39.2).

Note that unless the multi-device option is activated (not available yet), a single device can be registered on the server for an identity. When restoring Olvid on a new device, the "kick other devices" boolean should be true to replace the previous deviceUid with the new one on the server.

/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	kick other devices
	boolean	use multi-device
out	byte[1]	return status
return statuses		0x00: OK
		0x04: invalid token (re-authentication required)
		0x0a: another deviceUid is already registered (and "kick" is false)
		0xff: unknown error

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).
- 0x04 - **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.

  The role of the masking unique identifier is the same as in Android notifications.
- 0x05 - **iOS sandbox with extension**: this notification type is exactly the same as the iOS with extention, but uses the development server provided by Apple for tests. It is not used in the production version of the application.
- 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.

39.2 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.

`/deviceDiscovery`		
in	`identity`	`identity`
out	`byte[1]` `[encoded_vals]`	return status list of encoded `deviceUid`
return statuses		`0x00`: OK `0xff`: unknown error

The output `encoded_vals` contains a list of encoded `byte[32]`, each representing a `deviceUid` of the user owning the input `identity`.

39.3 Unregister push notification

When a `deviceUid` needs to be de-assossiated from an `identity`, a user can use the unregister push notification entry point. This entry point is only of interest in a multi-device scenario.

`/unregisterPushNotification`		

in	`identity`
	`byte[32]`
	`byte[32]`
		`identity`
		Authentication `token`
		`deviceUid`

out	`byte[1]`	return status
return statuses		`0x00`: OK `0x04`: invalid `token` (re-authentication required) `0xff`: unknown error

39.4 Upload return receipt

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 return receipt on the server. The delivery of the return receipt to the application message sender is then done through the WebSocket.

<table>
<tr><th colspan="3">/uploadReturnReceipt</th></tr>
<tr><td>in</td><td>identity
[encoded_vals]
byte[16]
byte[]</td><td>identity
list of encoded deviceUid
nonce
encrypted return receipt payload</td></tr>
<tr><td>out</td><td>byte[1]</td><td>return status</td></tr>
<tr><td colspan="2">return statuses</td><td>0x00: OK
0xff: unknown error</td></tr>
</table>

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.

39.5 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.

<table>
<tr><th colspan="3">/putUserData</th></tr>
<tr><td>in</td><td>identity
byte[32]
byte[]
byte[]</td><td>identity
authentication token
label
data</td></tr>
<tr><td>out</td><td>byte[1]</td><td>return status</td></tr>
<tr><td colspan="2">return statuses</td><td>0x00: OK
0x04: invalid token (re-authentication required)
0xff: unknown error</td></tr>
</table>

39.6 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 `0x05`: data not available (not uploaded yet, or deleted) `0xff`: unknown error

40 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.

40.1 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

40.2 Message notification

When sending a push notification to the device, if it is connected through a WebSocket and registered to the identity, the following message is sent.

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

The server does not expect any response to this message.

40.3 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 39.4).

40.4 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.

41 Push Notifications Content

41.1 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 39.1), 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 39.1), 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 40.2).

41.2 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.

Because of this, we distinguish two types of messages on the server, depending on whether there is something to display to the user (see Section 37.2):

- protocol messages without an application payload, triggering only the background notification
- application messages with something to display to the user, triggering both notifications

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 39.1), sent as an hexadecimal string.

Alert notification. When sent, the alert notification has the following structure.

```
{
  "aps": {
    "alert": {
      "title": "New message",
      "body": "Tap this notification to download the message",
      "title-loc-key": "New message",
      "loc-key": "Tap this notification to download the message"
    },
    "mutable-content": 1
  },
  "timestamp": [timestamp],
  "maskinguid": "[masking_uid]"
  "messageuid": "[message_uid]",
  "encryptedHeader": "[header]",
  "encryptedMessage": "[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 39.1), 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 37.2)
- `content` is the encrypted message payload received by the server, sent as a base64 string (see Section 37.2)

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.

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] 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.

[11] 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.

[12] Tanja Lange. Side-channel attacks and countermeasures for curve based cryptography, May 28 2007. http://www.hyperelliptic.org/tanja/vortraege/Lange_SCA.ps.

[13] Peter L. Montgomery. Speeding the Pollard and Elliptic Curve Methods of Factorization. *Mathematics of Computation*, 48(177):243–264, January 1987.

[14] 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`.

[15] Sylvain Pasini. *Secure Communication Using Authenticated Channels*. PhD thesis, EPFL, 2009. `https://infoscience.epfl.ch/record/138488/files/EPFL_TH4452.pdf`.

[16] Claus P. Schnorr. Efficient Signature Generation by Smart Cards. *Journal of Cryptology*, 4(3):161–174, 1991.

[17] 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`.

Appendices

A Elliptic Curves

A.1 Edwards Curves

The elliptic curves considered in these specifications are Edwards curves [7, 8, 11] 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 [13] and is well documented in [9, 12]. 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 [13] 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

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

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

$$p = 109112363276961190442711090369149551676330307646118204517771511330536253156371,$$

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

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