TutaCrypt
Protocol Specification

Version 1.0

Valentin Franck
Vitor Sakaguti
Bernd Deterding

tuta.com

March 11, 2024

Abstract

We present TutaCrypt, a protocol for hybrid post-quantum mail encryption. TutaCrypt replaces RSA or ECC-based classical encryption schemes. It combines a post-quantum Key Encapsualation Mechanism (CRYSTALS-Kyber) and an Elliptic-Curve-Diffie-Hellmann key exchange (x25519) to agree on a shared secret.

Contents

1 Introduction

This document specifies TutaCrypt, a protocol designed for hybrid mail encryption in Tuta Mail. The protocol combines a classical Elliptic-Curve-Diffie-Hellman key exchange with a post-quantum KEM. Its goal is to replace the usage of RSA in Tuta Mail. In the remainder of this document we first describe some preliminaries such as the cryptographic primitives used. We define the core algorithms of the protocol and describe the flow of messages between the communicating parties. Finally, we discuss the security properties and some limitations of the protocol.

2 Preliminaries and Notation

In this section we describe the roles, our preliminaries and the cryptographic notation.

2.1 Roles

There are three parties in the TutaCrypt protocol: Alice, Bob and a server.

- Alice wants to send an encrypted message to Bob.
- Bob wants to be able to receive encrypted messages from Alice.
- The server stores messages on behalf of Alice and Bob. It also allows them to publish their public keys.

2.2 Preliminaries

There are several parameters that all parties participating in the TutaCrypt protocol need to agree on:

- $EncodeMsg(V, IK, EK, CT, BK_{Enc})$: A function that encodes the version number V, the elliptic curve public keys IK and EK, the ciphertext CT and the encrypted symmetric key BK_{Enc} into a byte sequence.
- $(V, IK, EK, CT, BK_{Enc}) \leftarrow DecodeMsg(M)$: A function that decodes the byte sequence M into the version number V, the elliptic curve public keys IK and EK, the ciphertext CT and the encrypted symmetric key BK_{Enc}.
- $A \mid\mid B$ represents the concatenation of the byte sequences A and B.

2.3 Cryptographic Notation

TutaCrypt uses the following cryptographic primitives. Key pairs are denoted by public keys only. We assume that the private key can (only) be accessed by the key owner.

- $K \leftarrow AE_{Gen}()$: Generates a 256-bit random key for the Authenticated Encryption.

- $D_{Enc} \leftarrow AE_{Enc}(K, D)$: Authenticated Encryption: AES-256 in CBC mode with HMAC-SHA-256 (Encrypt-then-MAC), where K is the encryption key and D is the data to be encrypted. K is extended into two 256-bit keys, c_K for encryption and m_K for Mac computation, using SHA-512.

- $D \leftarrow AE_{Dec}(K, D_{Enc})$: Authenticated Decryption: AES-256 in CBC mode with HMAC-SHA-256 (Encrypt-then-MAC), where K is the decryption key and D_{Enc} is the data to be decrypted. K is extended into two 256-bit keys, one for encryption and one for Mac computation as explained above.

- $PK_{ECC} \leftarrow ECC_{Gen}()$: Generates an Elliptic Curve key pair (x25519), represented by its public key.

- $DH \leftarrow ECDH(PK_1, PK_2)$: The Elliptic Curve Diffie-Hellman Key Exchange instantiated with x25519 using the private key corresponding to the public key PK_1 as well as the public key PK_2.

- $PK_{PQ} \leftarrow PQ_{Gen}()$: Generates a Post-quantum key pair instantiated by Kyber-1024, represented by its public key.

- $(SS_{PQ}, CT_{PQ}) \leftarrow PQ_{Enc}(PK_{PQ})$: The Post-quantum Key Encapsulation Mechanism instantiated with Kyber-1024. PK_{PQ} is a public Kyber key. SS_{PQ} represents the shared secret output and CT_{PQ} represents the ciphertext output of the KEM.

- $SS_{PQ} \leftarrow PQ_{Dec}(CT_{PQ}, PK_{PQ})$: The Post-quantum Key Decapsulation Mechanism instantiated with Kyber-1024 using the private key corresponding to the public key IK_{PQ}. CT_{PQ} is the ciphertext produced during encapsulation with the public Kyber key PK_{PQ}. The output is the same shared secret SS_{PQ} as during the encapsulation.

- $K \leftarrow KDF(CTX, IKM, INFO, L)$: The Key Derivation Function instantiated with HKDF-SHA-256, that derives a key of length L bits. CTX is a context string, IKM is the secret input key material and $INFO$ is a string for domain separation.

2.4 Cryptographic Keys

TutaCrypt uses the following cryptographic keys:

- BK: a symmetric bucket key used for message encryption. Usually, this will be a key encryption key used to encrypt the actual session keys (that encrypt the payload message). The same BK will be reused when sending the same message to more than one recipient.

- KEK: a symmetric key encryption key both parties derive in the protocol run and use to encrypt or decrypt the bucket key BK.

- IKA_{ECC}: Alice's long term elliptic curve identity key pair for $ECDH$ computations, denoted by the public key.

- IKA_{PQ}: Alice's long term post-quantum identity key pair for PQ_{Enc} and PQ_{Dec} computations, denoted by the public key. This is only needed in case Alice receives messages herself.
- EKA_{ECC}: An ephemeral elliptic curve key pair for $ECDH$ computations, denoted by the public key. This key pair is generated by Alice during the encapsulation.
- IKB_{ECC}: Bob's long term elliptic curve identity key pair for $ECDH$ computations, denoted by the public key.
- IKB_{PQ}: Bob's long term post-quantum identity key pair for PQ_{Enc} and PQ_{Dec} computations, denoted by the public key.

3 Protocol Phases

3.1 Setup

Bob generates the TutaCrypt identity keys: an X25519 key pair IKB_{ECC} and a Kyber key pair IKB_{PQ}. Bob publishes both public keys (IKB_{ECC}, IKB_{PQ}). Alice generates and publishes her own TutaCrypt identity keys in the same way: (IKA_{ECC}, IKA_{PQ}). The key generation is described in Algorithm 1.

Algorithm 1 Setup

1: **procedure** GENERATEKEYS
2: $IK_{ECC} \leftarrow ECC_{Gen}()$
3: $IK_{PQ} \leftarrow PQ_{Gen}()$
4: $IK \leftarrow (IK_{ECC}, IK_{PQ})$
5: **return** IK
6: **end procedure**

3.2 Encapsulation

Alice generates BK with $BK \leftarrow AE_{Gen}()$ and runs Algorithm 2 in order to derive a key encryption key KEK and encrypt BK with it.[1]

[1] Please note that we are aware that formally this protocol would only be a KEM if the bucket key were to be generated inside the encapsulation algorithm. We choose otherwise to be able to reuse BK for multiple recipients. Anyway, it seems justified to call the algorithms we use *encapsulate* and *decapsulate*.

Algorithm 2 Encapsulation

procedure ENCAPSULATE(V, IKA_{ECC}, IKB_{ECC}, IKB_{PQ}, BK)
  if $V \neq 0$ **then** ▷ V is included to be able to make breaking changes in future versions
    return $\perp$
  end if
  $EKA_{ECC} \leftarrow ECC_{Gen}()$
  $DH_I \leftarrow ECDH(IKA_{ECC}, IKB_{ECC})$
  $DH_E \leftarrow ECDH(EKA_{ECC}, IKB_{ECC})$
  $(SS_{PQ}, CT_{PQ}) \leftarrow PQ_{Enc}(IKB_{PQ})$
  $CTX \leftarrow IKA_{ECC} \,||\, EKA_{ECC} \,||\, IKB_{ECC} \,||\, IKB_{PQ} \,||\, CT_{PQ} \,||\, V$
  $IKM \leftarrow DH_E \,||\, DH_I \,||\, SS_{PQ}$
  $KEK \leftarrow KDF(CTX, IKM, \text{“}kek\text{”}, 256)$
  $BK_{Enc} \leftarrow AE_{Enc}(KEK, BK)$
  $M_{TC} \leftarrow EncodeMsg(V, IKA_{ECC}, EKA_{ECC}, CT_{PQ}, BK_{Enc})$
  return M_{TC}
end procedure

Figure 1 depicts how Alice derives the shared secrets as described in Algorithm 2.

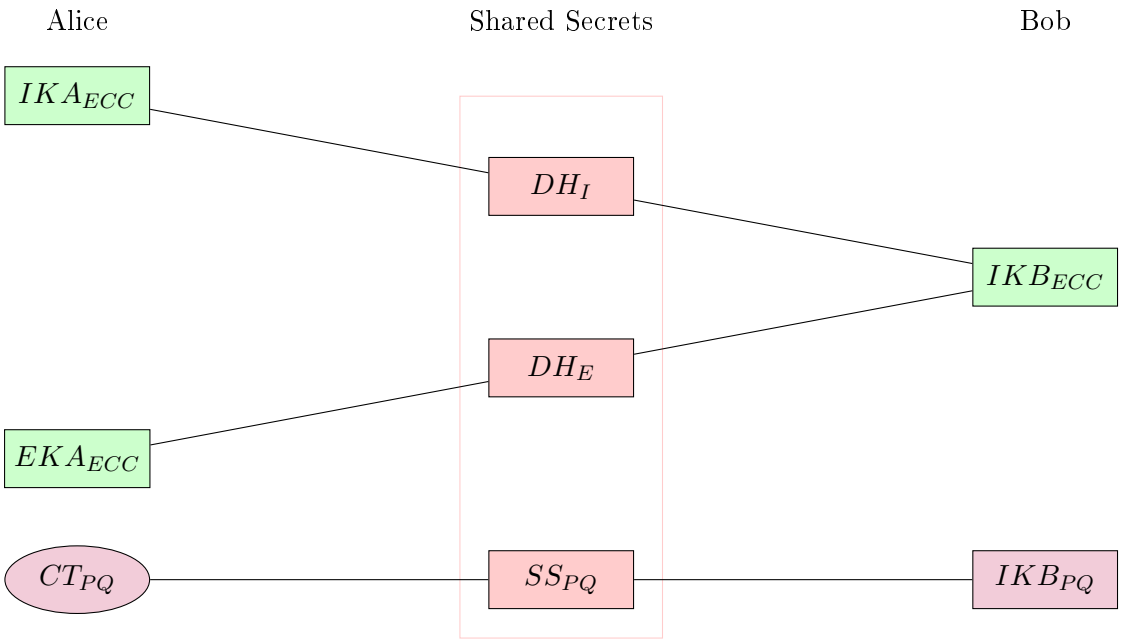

Figure 1: The diagram shows how the key pairs involved in TutaCrypt interact to derive shared secrets. The shared secrets are concatenated and used to derive an encryption key for the bucket key at a later step.

3.3 Decapsulation

In order to decrypt Alice's message, which is encrypted with the bucket key BK, Bob needs to derive the same key encryption key KEK as Alice and decrypt the bucket key BK_{Enc} with it. Algorithm 3 describes Bob's computations to obtain BK. Again the core part of the algorithm

is displayed in Figure 1. Note that the authentication of the sender is achieved via the identity key used in the encapsulation.[2] However, in order to base the authentication not on trust in the server Bob would need to verify Alice's identity key. As explained in 5.3.1 this is not yet possible.

Algorithm 3 Decapsulation

procedure DECAPSULATE(M_{TC}, IKB_{ECC}, IKB_{PQ}, IKA_{ECC})
  $(V, IKA'_{ECC}, EKA_{ECC}, CT_{PQ}, BK_{Enc}) \leftarrow DecodeMsg(M_{TC})$
  if $V \neq 0$ **then**
    return $\perp$
  end if
  if $IKA'_{ECC} \neq IKA_{ECC}$ **then** ▷ Key verification
    return $\perp$
  end if
  $DH_I \leftarrow ECDH(IKB_{ECC}, IKA_{ECC})$
  $DH_E \leftarrow ECDH(IKB_{ECC}, EKA_{ECC})$
  $SS_{PQ} \leftarrow PQ_{Dec}(CT_{PQ}, IKB_{PQ})$
  $CTX \leftarrow IKA_{ECC} \,||\, EKA_{ECC} \,||\, IKB_{ECC} \,||\, IKB_{PQ} \,||\, CT_{PQ} \,||\, V$
  $IKM \leftarrow DH_E \,||\, DH_I \,||\, SS_{PQ}$
  $KEK \leftarrow KDF(CTX, IKM, \text{“}kek\text{”}, 256)$
  $BK \leftarrow AE_{Dec}(KEK, BK_{Enc})$
  return BK
end procedure

4 Protocol Messages

In this section we describe how messages are sent between the communicating parties. Figure 2 provides an overview over the three phases: setup, encapsulation and decapsulation.

During setup, both Alice and Bob generate keys as described in the previous section and publish their public keys to the server. During encapsulation Alice first requests Bob's public keys from the server. She then runs the encapsulation algorithm as described before to encrypt a bucket key BK. It is out of the scope of this document how she can use that bucket key to encrypt her message and send it to Bob. During decapsulation Bob fetches Alice's TutaCrypt message M_{TC} and her public key from the server. He then decapsulates the message as described in the previous section and derives the same bucket key BK as Alice.

[2] For this purpose it would not have been necessary to include the used identity key in the message (IKA_{ECC}) and compare it to the one from the server (IKA'_{ECC}) as decryption would just fail if the wrong key were used. The necessity of the key comparison rather results from the Tuta Mail application. We have simplified this in the specified algorithm. In the rare case that Alice uses an email alias to send the mail and then moves the alias to a different user (of her own account) before Bob decrypts the message, IKA_{ECC} would indeed be diffrent from IKA'_{ECC} even though the mail is not forged. In addition, IKA_{ECC} must be used to decrypt the message if Alice deleted her account before Bob received the message. Bob could not get IKA'_{ECC} from the server anymore. Whenever IKA_{ECC} is used to decrypt a message instead of IKA'_{ECC}, we display a warning banner that the sender could not be authenticated.

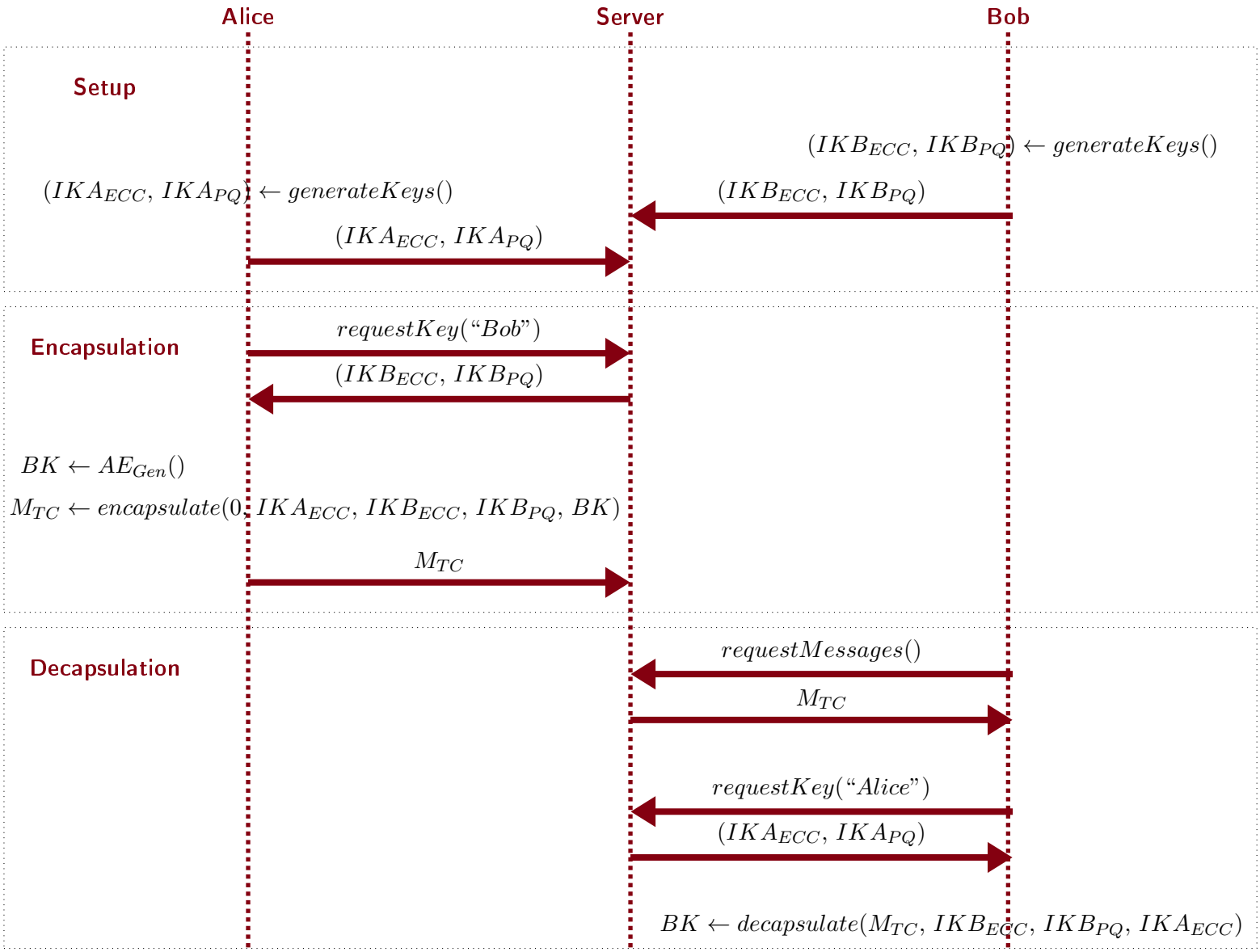

Figure 2: The diagram shows the flow of computations and messages in TutaCrypt.

5 Evaluation

In this section informally discuss the security properties and limitations of the protocol.

5.1 Threat Model

We consider an attacker that records messages during a run of the protocol and tries decrypting them once they have access to a quantum computer (Harvest Now Decrypt Later). We do not consider an active quantum adversary that is able to break the security of ECDH during the protocol execution. We do, however, consider an active adversary that might tamper with the exchanged data in the classical attacker model.

5.2 Security Properties

TutaCrypt provides confidentiality both against active classical and quantum adversaries. It provides integrity against classical adversaries. It provides authentication in the classical scenario as discussed in section 3.3. TutaCrypt neither provides perfect forward secrecy nor post-compromise security.

5.3 Limitations

In this section we discuss some known limitations of the presented protocol design. We acknowledge that message authentication is not as robust as confidentiality, in particular in scenarios where the identity keys of either party are compromised or messages are sent to untrusted recipients.

5.3.1 Missing key verification

As explained in section 3.3 the communicating parties authenticate themselves by using their identity keys in the encapsulation algorithm. However, the identity keys are delivered by the server and there are currently no means to verify them. A malicious server, for example a compromised one or a Machine-in-the-Middle (MITM), could deliver a different key claiming it to be Alice's. We will address this issue in the future by implementing proper key verification mechanisms.

5.3.2 Insider non-repudiation

All symmetric encryption is Authenticated Encryption. However, anyone with access to the bucket key BK could change the message and recompute the MACs on the encrypted instances as they can decrypt the session keys with the bucket key. This is relevant in a scenario where Alice sends her message to multiple recipients, say to both Bob and Charlie while using the same bucket key BK. Currently, while the protocol provides plausible deniability it does not achieve insider non-repudiation as anyone with access to the bucket key could have created or modified the message.

This could be mitigated by including a tag over the ciphertext of the entire message in the context string when deriving the key encryption key. That way no other recipient could have forged the message, because the key derivation and thus decryption would just fail if it were modified after sending. However, this is not trivial in the Tuta Mail application.

5.3.3 Identity Key Compromise

If Alice's long term identity keys are compromised by Charlie, Charlie can send messages to Alice impersonating any other party to Alice. Also, Charlie can impersonate Alice to any other party. Both attacks are possible because authentication in the protocol is only based on the ECDH between static identity keys. The protocol is not able to recover from this state. There are two possible changes that could mitigate this. The first is allowing Alice to rotate her identity keys (periodically and/or on demand) and the second is switching to a double ratchet based protocol that provides perfect forward secrecy and future secrecy.

5.3.4 Identity Binding

It is currently possible that Charlie presents Bob's public keys as his own to Alice. This is known as an unknown key share attack. If Charlie does that, Alice will think that she sends a message to Charlie and Charlie can forward the message to Bob. Charlie can even change the recipient field on the mail instance as it is not encrypted and more importantly not authenticated.

It might be possible to reduce the probability of such an attack by including more data such as mail addresses in the context of the KDF.

5.3.5 Replayable Authentication

If an attacker, called Charlie, ever obtains a valid bucket key BK and the corresponding ciphertext M_{TC} from a message that Alice sent to Bob, Charlie can use the bucket key to encrypt his own payload message M_C and send M_C and M_{TC} to Bob pretending it came from Alice. Bob would not be able to recognize that M_{TC} is being replayed to him and would assume that M_C is an authenticated message from Alice.