Stream:	Internet Research Task Force (IRTF)			
RFC:	9807			
Category:	Informational			
Published:	July 2025			
ISSN:	2070-1721			
Authors:	D. Bourdrez	H. Krawczyk	K. Lewi	C. A. Wood
		AWS	*Meta*	*Cloudflare, Inc.*

RFC 9807
The OPAQUE Augmented Password-Authenticated Key Exchange (aPAKE) Protocol

Abstract

This document describes the OPAQUE protocol, an Augmented (or Asymmetric) Password-Authenticated Key Exchange (aPAKE) protocol that supports mutual authentication in a client-server setting without reliance on PKI and with security against pre-computation attacks upon server compromise. In addition, the protocol provides forward secrecy and the ability to hide the password from the server, even during password registration. This document specifies the core OPAQUE protocol and one instantiation based on 3DH. This document is a product of the Crypto Forum Research Group (CFRG) in the IRTF.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This document is a product of the Internet Research Task Force (IRTF). The IRTF publishes the results of Internet-related research and development activities. These results might not be suitable for deployment. This RFC represents the consensus of the Crypto Forum Research Group of the Internet Research Task Force (IRTF). Documents approved for publication by the IRSG are not candidates for any level of Internet Standard; see Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9807.

Copyright Notice

Table of Contents

1. Introduction

Password authentication is ubiquitous in many applications. In a common implementation, a client authenticates to a server by sending its client ID and password to the server over a secure connection. This makes the password vulnerable to server mishandling, including accidentally logging the password or storing it in plaintext in a database. Server compromise resulting in access to these plaintext passwords is not an uncommon security incident, even among security-conscious organizations. Moreover, plaintext password authentication over secure channels such as TLS is also vulnerable in cases where TLS may fail, including PKI attacks, certificate mishandling, termination outside the security perimeter, visibility to TLS-terminating intermediaries, and more.

Augmented (or Asymmetric) Password Authenticated Key Exchange (aPAKE) protocols are designed to provide password authentication and mutually authenticated key exchange in a client-server setting without relying on PKI (except during client registration) and without disclosing passwords to servers or other entities other than the client machine. A secure aPAKE should provide the best possible security for a password protocol. Indeed, some attacks are inevitable, such as online impersonation attempts with guessed client passwords and offline dictionary attacks upon the compromise of a server and leakage of its credential file. In the latter case, the attacker learns a mapping of a client's password under a one-way function and uses such a mapping to validate potential guesses for the password. It is crucially important for the password protocol to use an unpredictable one-way mapping. Otherwise, the attacker can pre-compute a deterministic list of mapped passwords leading to almost instantaneous leakage of passwords upon server compromise.

This document describes OPAQUE, an aPAKE protocol that is secure against pre-computation attacks (as defined in [JKX18Full]). OPAQUE provides forward secrecy with respect to password leakage while also hiding the password from the server, even during password registration. OPAQUE allows applications to increase the difficulty of offline dictionary attacks via iterated hashing or other key-stretching schemes. OPAQUE is also extensible, allowing clients to safely store and retrieve arbitrary application data on servers using only their password.

OPAQUE is defined and proven as the composition of three functionalities: an Oblivious Pseudorandom Function (OPRF), a key recovery mechanism, and an authenticated key exchange (AKE) protocol. It can be seen as a "compiler" for transforming any suitable AKE protocol into a secure aPAKE protocol. (See Section 10 for requirements of the OPRF and AKE protocols.) This document specifies one OPAQUE instantiation based on [TripleDH]. Other instantiations are possible, as discussed in Appendix B, but their details are out of scope for this document. In general, the modularity of OPAQUE's design makes it easy to integrate with additional AKE protocols, e.g., TLS or HMQV (Hashed Menezes-Qu-Vanstone), and with future AKE protocols such as those based on post-quantum techniques.

OPAQUE consists of two stages: registration and authenticated key exchange. In the first stage, a client registers its password with the server and stores information used to recover authentication credentials on the server. Recovering these credentials can only be done with

knowledge of the client password. In the second stage, a client uses its password to recover those credentials and subsequently uses them as input to an AKE protocol. This stage has additional mechanisms to prevent an active attacker from interacting with the server to guess or confirm clients registered via the first phase. Servers can use this mechanism to safeguard registered clients against this type of enumeration attack; see Section 10.9 for more discussion.

The name "OPAQUE" is a homonym of O-PAKE, where O is for Oblivious. The name "OPAKE" was taken.

This document complies with the requirements for PAKE protocols set forth in [RFC8125]. This document represents the consensus of the Crypto Forum Research Group (CFRG). It is not an IETF product and is not a standard.

1.1. Requirements Notation

The key words "**MUST**", "**MUST NOT**", "**REQUIRED**", "**SHALL**", "**SHALL NOT**", "**SHOULD**", "**SHOULD NOT**", "**RECOMMENDED**", "**NOT RECOMMENDED**", "**MAY**", and "**OPTIONAL**" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.2. Notation

The following functions are used throughout this document:

I2OSP and OS2IP: Convert a byte string to and from a non-negative integer as described in Section 4 of [RFC8017]. Note that these functions operate on byte strings in big-endian byte order.

concat(x0, ..., xN): Concatenate byte strings. For example, `concat(0x01, 0x0203, 0x040506) = 0x010203040506`.

random(n): Generates a cryptographically secure pseudorandom byte string of length `n` bytes.

zeroes(n): Generate a string of `n` bytes all equal to 0 (zero).

xor(a,b): Apply XOR to byte strings. For example, `xor(0xF0F0, 0x1234) = 0xE2C4`. It is an error to call this function with arguments of unequal length.

ct_equal(a, b): Return `true` if `a` is equal to `b`, and false otherwise. The implementation of this function must be constant-time in the length of `a` and `b`, which are assumed to be of equal length, irrespective of the values `a` or `b`.

Except if said otherwise, random choices in this specification refer to drawing with uniform distribution from a given set (i.e., "random" is short for "uniformly random"). Random choices can be replaced with fresh outputs from a cryptographically strong pseudorandom generator, according to the requirements in [RFC4086], or a pseudorandom function. For convenience, we define `nil` as a lack of value.

All protocol messages and structures defined in this document use the syntax from Section 3 of [RFC8446].

2. Cryptographic Dependencies

OPAQUE depends on the following cryptographic protocols and primitives:

- Oblivious Pseudorandom Function (OPRF); Section 2.1
- Key Derivation Function (KDF); Section 2.2
- Message Authentication Code (MAC); Section 2.2
- Cryptographic Hash Function; Section 2.3
- Key Stretching Function (KSF); Section 2.3

This section describes these protocols and primitives in more detail. Unless said otherwise, all random nonces and `seeds` used in these dependencies and the rest of the OPAQUE protocol are of length `Nn` and `Nseed` bytes, respectively, where `Nn` = `Nseed` = 32.

2.1. Oblivious Pseudorandom Function

An Oblivious Pseudorandom Function (OPRF) is a two-party protocol between client and server for computing a Pseudorandom Function (PRF), where the PRF key is held by the server and the input to the function is provided by the client. The client does not learn anything about the PRF other than the obtained output, and the server learns nothing about the client's input or the function output. This specification depends on the prime-order OPRF construction specified as `modeOPRF` (`0x00`) from Section 3.1 of [RFC9497].

The following OPRF client APIs are used:

Blind(element): Create and output (`blind`, `blinded_element`), consisting of a blinded representation of input `element`, denoted `blinded_element`, along with a value to revert the blinding process, denoted `blind`. This is equivalent to the Blind function described in Section 3.3.1 of [RFC9497].

Finalize(element, blind, evaluated_element): Finalize the OPRF evaluation using input `element`, random inverter `blind`, and evaluation output `evaluated_element`, yielding output `oprf_output`. This is equivalent to the Finalize function described in Section 3.3.1 of [RFC9497].

Moreover, the following OPRF server APIs are used:

BlindEvaluate(k, blinded_element): Evaluate blinded input `blinded_element` using input key `k`, yielding output element `evaluated_element`. This is equivalent to the BlindEvaluate function described in Section 3.3.1 of [RFC9497], where `k` is the private key parameter.

DeriveKeyPair(seed, info): Create and output (`sk`, `pk`), consisting of a private and public key derived deterministically from an input `seed` and input `info` parameter, as described in Section 3.2 of [RFC9497].

Finally, this specification makes use of the following shared APIs and parameters:

SerializeElement(element): Map input `element` to a fixed-length byte array.

DeserializeElement(buf): Attempt to map input byte array `buf` to an OPRF group element. This function can raise a DeserializeError upon failure; see Section 2.1 of [RFC9497] for more details.

Noe: The size of a serialized OPRF group element output from SerializeElement.

Nok: The size of an OPRF private key as output from DeriveKeyPair.

2.2. Key Derivation Function and Message Authentication Code

A Key Derivation Function (KDF) is a function that takes some source of initial keying material and uses it to derive one or more cryptographically strong keys. This specification uses a KDF with the following API and parameters:

Extract(salt, ikm): Extract a pseudorandom key of fixed length `Nx` bytes from input keying material `ikm` and an optional byte string `salt`.

Expand(prk, info, L): Expand a pseudorandom key `prk`, using the string `info`, into `L` bytes of output keying material.

`Nx`: The output size of the `Extract()` function in bytes.

This specification also makes use of a random-key robust Message Authentication Code (MAC). See Section 10.6 for more details on this property. The API and parameters for the random-key robust MAC are as follows:

MAC(key, msg): Compute a message authentication code over input `msg` with key `key`, producing a fixed-length output of `Nm` bytes.

`Nm`: The output size of the `MAC()` function in bytes.

2.3. Hash Functions

This specification makes use of a collision-resistant hash function with the following API and parameters:

Hash(msg): Apply a cryptographic hash function to input `msg`, producing a fixed-length digest of size `Nh` bytes.

`Nh`: The output size of the `Hash()` function in bytes.

This specification makes use of a Key Stretching Function (KSF), which is a slow and expensive cryptographic hash function with the following API:

Stretch(msg): Apply a key stretching function to stretch the input `msg` and harden it against offline dictionary attacks. This function also needs to satisfy collision resistance.

3. Protocol Overview

OPAQUE consists of two stages: registration and authenticated key exchange (AKE). In the first stage, a client registers its password with the server and stores its credential file on the server. In the second stage (also called the "login" or "online" stage), the client recovers its authentication material and uses it to perform a mutually authenticated key exchange.

3.1. Setup

Prior to both stages, the client and server agree on a configuration that fully specifies the cryptographic algorithm dependencies necessary to run the protocol; see Section 7 for details. The server chooses a pair of keys (`server_private_key` and `server_public_key`) for the AKE protocol and chooses a `seed` (`oprf_seed`) of `Nh` bytes for the OPRF. The server can use `server_private_key` and `server_public_key` with multiple clients. The server can also opt to use a different `seed` for each client (i.e., each client can be assigned a single `seed`), so long as they are maintained across the registration and online AKE stages and kept consistent for each client (since an inconsistent mapping of clients to `seeds` could leak information as described in Section 10.9).

3.2. Registration

Registration is the only stage in OPAQUE that requires a server-authenticated channel with confidentiality and integrity: either physical, out-of-band, PKI-based, etc.

The client inputs its credentials, which include its password and user identifier, and the server inputs its parameters, which include its private key and other information.

The client output of this stage is a single value `export_key` that the client may use for application-specific purposes, e.g., as a symmetric key used to encrypt additional information for storage on the server. The server does not have access to this `export_key`.

The server output of this stage is a `record` corresponding to the client's registration that it stores in a credential file alongside other clients registrations as needed.

The registration flow is shown in Figure 1, and the process is described in more detail in Section 5:

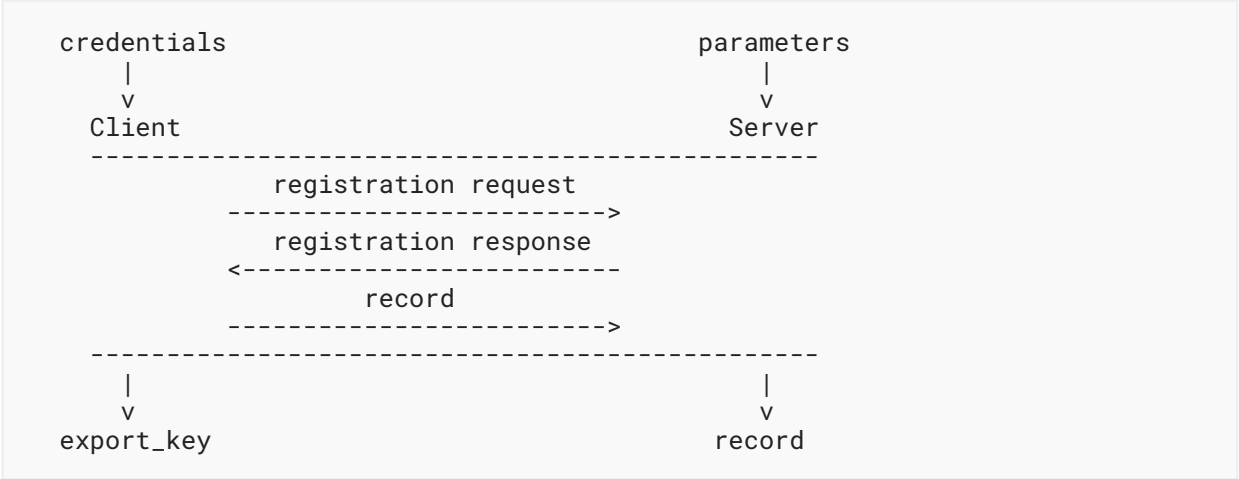

Figure 1

These messages are named `RegistrationRequest`, `RegistrationResponse`, and `RegistrationRecord`, respectively. Their contents and wire format are defined in Section 5.1.

3.3. Online Authenticated Key Exchange

In this second stage, a client obtains credentials previously registered with the server, recovers private key material using the password, and subsequently uses them as input to the AKE protocol. As in the registration phase, the client inputs its credentials, including its password and user identifier, and the server inputs its parameters and the credential file `record` corresponding to the client. The client outputs two values, an `export_key` (matching that from registration) and a `session_key`, the latter of which is the primary AKE protocol output. The server outputs a single value `session_key` that matches that of the client. Upon completion, clients and servers can use these values as needed.

The authenticated key exchange flow is shown in Figure 2:

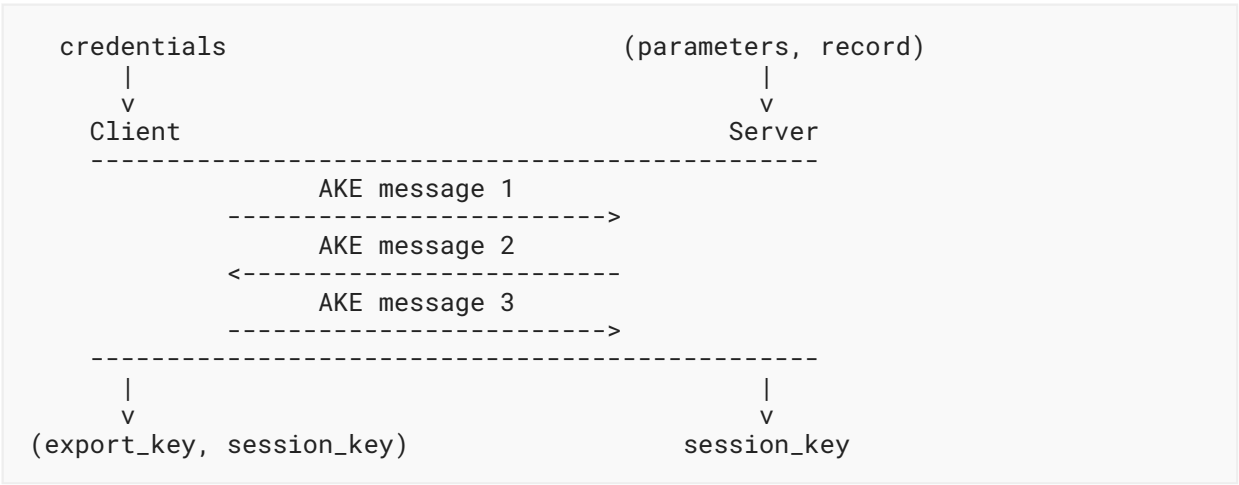

Figure 2

These messages are named `KE1`, `KE2`, and `KE3`, respectively. They carry the messages of the concurrent execution of the key recovery process (OPRF) and the authenticated key exchange (AKE). Their corresponding wire formats are specified in Section 6.1.

The rest of this document describes the specifics of these stages in detail. Section 4 describes how client credential information is generated, encoded, and stored on the server during registration and recovered during login. Section 5 describes the first registration stage of the protocol, and Section 6 describes the second authentication stage of the protocol. Section 7 describes how to instantiate OPAQUE using different cryptographic dependencies and parameters.

4. Client Credential Storage and Key Recovery

OPAQUE makes use of a structure called `Envelope` to manage client credentials. The client creates its `Envelope` on registration and sends it to the server for storage. On every login, the server sends this `Envelope` to the client so it can recover its key material for use in the AKE.

Applications may pin key material to identities if desired. If no identity is given for a party, its value **MUST** default to its public key. The following types of application credential information are considered:

`client_private_key`: The encoded client private key for the AKE protocol.

`client_public_key`: The encoded client public key for the AKE protocol.

`server_public_key`: The encoded server public key for the AKE protocol.

`client_identity`: The client identity. This is an application-specific value, e.g., an email address or an account name. If not specified, it defaults to the client's public key.

`server_identity`: The server identity. This is typically a domain name, e.g., example.com. If not specified, it defaults to the server's public key. See Section 10.3 for information about this identity.

A subset of these credential values are used in the `CleartextCredentials` structure as follows:

```
struct {
  uint8 server_public_key[Npk];
  uint8 server_identity<1..2^16-1>;
  uint8 client_identity<1..2^16-1>;
} CleartextCredentials;
```

The function CreateCleartextCredentials constructs a `CleartextCredentials` structure given application credential information.

```
CreateCleartextCredentials

Input:
- server_public_key, the encoded server public key
  for the AKE protocol.
- client_public_key, the encoded client public key
  for the AKE protocol.
- server_identity, the optional encoded server identity.
- client_identity, the optional encoded client identity.

Output:
- cleartext_credentials, a CleartextCredentials structure.

def CreateCleartextCredentials(server_public_key, client_public_key,
                               server_identity, client_identity):
  # Set identities as public keys if no
  # application-layer identity is provided
  if server_identity == nil
    server_identity = server_public_key
  if client_identity == nil
    client_identity = client_public_key

  cleartext_credentials = CleartextCredentials {
    server_public_key,
    server_identity,
    client_identity
  }

  return cleartext_credentials
```

4.1. Key Recovery

This specification defines a key recovery mechanism that uses the stretched OPRF output as a `seed` to directly derive the private and public keys using the `DeriveDiffieHellmanKeyPair()` function defined in Section 6.4.1.

4.1.1. Envelope Structure

The key recovery mechanism defines its `Envelope` as follows:

```
struct {
  uint8 envelope_nonce[Nn];
  uint8 auth_tag[Nm];
} Envelope;
```

`envelope_nonce`: A randomly sampled nonce of length `Nn` used to protect this `Envelope`.

`auth_tag`: An authentication tag protecting the contents of the `Envelope`, covering `envelope_nonce` and `CleartextCredentials`.

4.1.2. Envelope Creation

Clients create an `Envelope` at registration with the function `Store` defined below. Note that `DeriveDiffieHellmanKeyPair` in this function can fail with negligible probability. If this occurs, servers should re-run the function, sampling a new `envelope_nonce`, to completion.

```
Store

Input:
- randomized_password, a randomized password.
- server_public_key, the encoded server public key for
  the AKE protocol.
- server_identity, the optional encoded server identity.
- client_identity, the optional encoded client identity.

Output:
- envelope, the client's Envelope structure.
- client_public_key, the client's AKE public key.
- masking_key, an encryption key used by the server with the
  sole purpose of defending against client enumeration attacks.
- export_key, an additional client key.

def Store(randomized_password, server_public_key,
          server_identity, client_identity):
  envelope_nonce = random(Nn)
  masking_key = Expand(randomized_password, "MaskingKey", Nh)
  auth_key =
    Expand(randomized_password, concat(envelope_nonce, "AuthKey"),
           Nh)
  export_key =
    Expand(randomized_password, concat(envelope_nonce, "ExportKey"),
           Nh)
  seed =
    Expand(randomized_password, concat(envelope_nonce, "PrivateKey"),
           Nseed)
  (_, client_public_key) = DeriveDiffieHellmanKeyPair(seed)

  cleartext_credentials =
```

```
    CreateCleartextCredentials(server_public_key, client_public_key,
                               server_identity, client_identity)
  auth_tag =
    MAC(auth_key, concat(
      envelope_nonce,
      server_public_key,
      I2OSP(len(cleartext_credentials.server_identity), 2),
      cleartext_credentials.server_identity,
      I2OSP(len(cleartext_credentials.client_identity), 2),
      cleartext_credentials.client_identity
    ))

  envelope = Envelope {
    envelope_nonce,
    auth_tag
  }

  return (envelope, client_public_key, masking_key, export_key)
```

4.1.3. Envelope Recovery

Clients recover their `Envelope` during login with the `Recover` function defined below.

```
Recover

Input:
- randomized_password, a randomized password.
- server_public_key, the encoded server public key for the
  AKE protocol.
- envelope, the client's Envelope structure.
- server_identity, the optional encoded server identity.
- client_identity, the optional encoded client identity.

Output:
- client_private_key, the encoded client private key for the
  AKE protocol.
- cleartext_credentials, a CleartextCredentials structure.
- export_key, an additional client key.

Exceptions:
- EnvelopeRecoveryError, the Envelope fails to be recovered.

def Recover(randomized_password, server_public_key, envelope,
            server_identity, client_identity):
  auth_key =
    Expand(randomized_password, concat(envelope.nonce, "AuthKey"),
           Nh)
  export_key =
    Expand(randomized_password, concat(envelope.nonce, "ExportKey"),
           Nh)
  seed =
    Expand(randomized_password, concat(envelope.nonce, "PrivateKey"),
           Nseed)
  (client_private_key, client_public_key) =
    DeriveDiffieHellmanKeyPair(seed)

  cleartext_credentials =
    CreateCleartextCredentials(server_public_key, client_public_key,
                               server_identity, client_identity)
  expected_tag =
    MAC(auth_key, concat(envelope.nonce, cleartext_credentials))
  If !ct_equal(envelope.auth_tag, expected_tag)
    raise EnvelopeRecoveryError
  return (client_private_key, cleartext_credentials, export_key)
```

In the case of `EnvelopeRecoveryError` being raised, all previously computed intermediary values in this function **MUST** be deleted.

5. Registration

The registration process proceeds as follows. The client inputs the following values:

password: The client's password.

creds: The client credentials as described in Section 4.

The server inputs the following values:

`server_public_key`: The server public key for the AKE protocol.

`credential_identifier`: A unique identifier for the client's credential generated by the server.

`client_identity`: The optional client identity as described in Section 4.

`oprf_seed`: A seed used to derive per-client OPRF keys.

The registration protocol then runs as shown below:

```
 Client                                              Server
-------------------------------------------------------------
(request, blind) = CreateRegistrationRequest(password)

                          request
              ------------------------->

response = CreateRegistrationResponse(request,
                                      server_public_key,
                                      credential_identifier,
                                      oprf_seed)

                          response
              <-------------------------

(record, export_key) = FinalizeRegistrationRequest(password,
                                                   blind,
                                                   response,
                                                   server_identity,
                                                   client_identity)

                          record
              ------------------------->
```

Section 5.1 describes the formats for the above messages, and Section 5.2 describes details of the functions and the corresponding parameters referenced above.

At the end of this interaction, the server stores the `record` object as the credential file for each client along with the associated `credential_identifier` and `client_identity` (if different). Note that the values `oprf_seed` and `server_private_key` from the server's setup phase must also be persisted. The `oprf_seed` value **SHOULD** be used for all clients; see Section 10.9 for the justification behind this, along with a description of the exception in which applications may choose to avoid the use of a global `oprf_seed` value across clients and instead sample OPRF keys uniquely for each client. The `server_private_key` may be unique for each client.

Both client and server **MUST** validate the other party's public key before use. See Section 10.7 for more details. Upon completion, the server stores the client's credentials for later use. Moreover, the client **MAY** use the output `export_key` for further application-specific purposes; see Section 10.4.

5.1. Registration Messages

This section contains definitions of the `RegistrationRequest`, `RegistrationResponse`, and `RegistrationRecord` messages exchanged between client and server during registration.

```
struct {
  uint8 blinded_message[Noe];
} RegistrationRequest;
```

blinded_message: A serialized OPRF group element.

```
struct {
  uint8 evaluated_message[Noe];
  uint8 server_public_key[Npk];
} RegistrationResponse;
```

evaluated_message: A serialized OPRF group element.

`server_public_key`: The server's encoded public key that will be used for the online AKE stage.

```
struct {
  uint8 client_public_key[Npk];
  uint8 masking_key[Nh];
  Envelope envelope;
} RegistrationRecord;
```

`client_public_key`: The client's encoded public key corresponding to the private key `client_private_key`.

`masking_key`: An encryption key used by the server with the sole purpose of defending against client enumeration attacks.

envelope: The client's `Envelope` structure.

5.2. Registration Functions

This section contains definitions of the functions used by client and server during registration, including `CreateRegistrationRequest`, `CreateRegistrationResponse`, and `FinalizeRegistrationRequest`.

5.2.1. CreateRegistrationRequest

To begin the registration flow, the client executes the following function. This function can fail with an `InvalidInputError` error with negligible probability. A different input password is necessary in the event of this error.

```
CreateRegistrationRequest

Input:
- password, an opaque byte string containing the client's password.

Output:
- request, a RegistrationRequest structure.
- blind, an OPRF scalar value.

Exceptions:
- InvalidInputError, when Blind fails

def CreateRegistrationRequest(password):
  (blind, blinded_element) = Blind(password)
  blinded_message = SerializeElement(blinded_element)
  request = RegistrationRequest {
    blinded_message
  }
  return (request, blind)
```

5.2.2. CreateRegistrationResponse

To process the client's registration request, the server executes the following function. This function can fail with a `DeriveKeyPairError` error with negligible probability. In this case, applications can choose a new `credential_identifier` for this registration `record` and rerun this function.

```
CreateRegistrationResponse

Input:
- request, a RegistrationRequest structure.
- server_public_key, the server's public key.
- credential_identifier, an identifier that uniquely represents
  the credential.
- oprf_seed, the seed of Nh bytes used by the server to generate
  an oprf_key.

Output:
- response, a RegistrationResponse structure.

Exceptions:
- DeserializeError, when OPRF element deserialization fails.
- DeriveKeyPairError, when OPRF key derivation fails.

def CreateRegistrationResponse(request, server_public_key,
                               credential_identifier, oprf_seed):
  seed =
   Expand(oprf_seed, concat(credential_identifier, "OprfKey"), Nok)
  (oprf_key, _) = DeriveKeyPair(seed, "OPAQUE-DeriveKeyPair")

  blinded_element = DeserializeElement(request.blinded_message)
  evaluated_element = BlindEvaluate(oprf_key, blinded_element)
  evaluated_message = SerializeElement(evaluated_element)

  response = RegistrationResponse {
    evaluated_message,
    server_public_key
  }

  return response
```

5.2.3. FinalizeRegistrationRequest

To create the user `record` used for subsequent authentication and complete the registration flow, the client executes the following function.

```
FinalizeRegistrationRequest

Input:
- password, an opaque byte string containing the client's password.
- blind, an OPRF scalar value.
- response, a RegistrationResponse structure.
- server_identity, the optional encoded server identity.
- client_identity, the optional encoded client identity.

Output:
- record, a RegistrationRecord structure.
- export_key, an additional client key.

Exceptions:
- DeserializeError, when OPRF element deserialization fails.

def FinalizeRegistrationRequest(password, blind, response,
                                server_identity, client_identity):
  evaluated_element = DeserializeElement(response.evaluated_message)
  oprf_output = Finalize(password, blind, evaluated_element)

  stretched_oprf_output = Stretch(oprf_output)
  randomized_password =
    Extract("", concat(oprf_output, stretched_oprf_output))

  (envelope, client_public_key, masking_key, export_key) =
    Store(randomized_password, response.server_public_key,
          server_identity, client_identity)

  record = RegistrationRecord {
    client_public_key,
    masking_key,
    envelope
  }

  return (record, export_key)
```

See Section 6 for details about the output `export_key` usage.

6. Online Authenticated Key Exchange

The generic outline of OPAQUE with a 3-message AKE protocol includes three messages: `KE1`, `KE2`, and `KE3`. `KE1` and `KE2` include key exchange shares (e.g., DH values) sent by the client and server, respectively. `KE3` provides explicit client authentication and full forward security (without it, forward secrecy is only achieved against eavesdroppers, which is insufficient for OPAQUE security).

This section describes the online authenticated key exchange protocol flow, message encoding, and helper functions. This stage is composed of a concurrent OPRF and key exchange flow. The key exchange protocol is authenticated using the client and server credentials established

during registration; see Section 5. In the end, the client proves its knowledge of the password, and both client and server agree on (1) a mutually authenticated shared secret key and (2) any optional application information exchange during the handshake.

In this stage, the client inputs the following values:

password: The client's password.

`client_identity`: The client identity as described in Section 4.

The server inputs the following values:

`server_private_key`: The server's private key for the AKE protocol.

`server_public_key`: The server's public key for the AKE protocol.

`server_identity`: The server identity as described in Section 4.

`record`: The `RegistrationRecord` object corresponding to the client's registration.

`credential_identifier`: An identifier that uniquely represents the credential.

`oprf_seed`: The `seed` used to derive per-client OPRF keys.

The client receives two outputs: a session secret and an export key. The export key is only available to the client and may be used for additional application-specific purposes, as outlined in Section 10.4. Clients **MUST NOT** use the output `export_key` before authenticating the peer in the authenticated key exchange protocol. See Appendix A for more details about this requirement. The server receives a single output: a session secret matching the client's.

The protocol runs as shown below:

```
 Client                                                Server
-------------------------------------------------------------
 ke1 = GenerateKE1(password)

                          ke1
               ------------------------->

 ke2 = GenerateKE2(server_identity, server_private_key,
                   server_public_key, record,
                   credential_identifier, oprf_seed, ke1)

                          ke2
               <-------------------------

   (ke3,
   session_key,
   export_key) = GenerateKE3(client_identity,
                             server_identity, ke2)

                          ke3
               ------------------------->

                       session_key = ServerFinish(ke3)
```

Both client and server may use implicit internal state objects to keep necessary material for the OPRF and AKE, `client_state`, and `server_state`, respectively.

The client state `ClientState` may have the following fields:

password: The client's password.

`blind`: The random blinding inverter returned by `Blind()`.

client_ake_state: The `ClientAkeState` as defined in Section 6.4.

The server state `ServerState` may have the following fields:

server_ake_state: The `ServerAkeState` as defined in Section 6.4.

Both of these states are ephemeral and should be erased after the protocol completes.

The rest of this section describes these authenticated key exchange messages and their parameters in more detail. Section 6.1 defines the structure of the messages passed between client and server in the above setup. Section 6.2 describes details of the functions and corresponding parameters mentioned above. Section 6.3 discusses internal functions used for retrieving client credentials, and Section 6.4 discusses how these functions are used to execute the authenticated key exchange protocol.

6.1. AKE Messages

In this section, we define the `KE1`, `KE2`, and `KE3` structs that make up the AKE messages used in the protocol. `KE1` is composed of a `CredentialRequest` and `AuthRequest`, and `KE2` is composed of a `CredentialResponse` and `AuthResponse`.

```
struct {
  uint8 client_nonce[Nn];
  uint8 client_public_keyshare[Npk];
} AuthRequest;
```

client_nonce: A fresh randomly generated nonce of length `Nn`.

client_public_keyshare: A serialized client ephemeral public key of fixed size `Npk`.

```
struct {
  CredentialRequest credential_request;
  AuthRequest auth_request;
} KE1;
```

credential_request: A `CredentialRequest` structure.

auth_request: An `AuthRequest` structure.

```
struct {
  uint8 server_nonce[Nn];
  uint8 server_public_keyshare[Npk];
  uint8 server_mac[Nm];
} AuthResponse;
```

server_nonce: A fresh randomly generated nonce of length `Nn`.

`server_public_key`share: A server ephemeral public key of fixed size `Npk`, where `Npk` depends on the corresponding prime order group.

server_mac: An authentication tag computed over the handshake transcript computed using `Km2`, which is defined below.

```
struct {
  CredentialResponse credential_response;
  AuthResponse auth_response;
} KE2;
```

credential_response: A `CredentialResponse` structure.

auth_response: An `AuthResponse` structure.

```
struct {
  uint8 client_mac[Nm];
} KE3;
```

client_mac: An authentication tag computed over the handshake transcript of fixed size `Nm`, computed using `Km2`, defined below.

6.2. AKE Functions

In this section, we define the main functions used to produce the AKE messages in the protocol. Note that this section relies on definitions of subroutines defined in later sections:

- `CreateCredentialRequest`, `CreateCredentialResponse`, and `RecoverCredentials` are defined in Section 6.3.
- `AuthClientStart`, `AuthServerRespond`, `AuthClientFinalize`, and `AuthServerFinalize` are defined in Sections 6.4.3 and 6.4.4.

6.2.1. GenerateKE1

The `GenerateKE1` function begins the AKE protocol and produces the client's `KE1` output for the server.

```
GenerateKE1

State:
- state, a ClientState structure.

Input:
- password, an opaque byte string containing the client's password.

Output:
- ke1, a KE1 message structure.

def GenerateKE1(password):
  request, blind = CreateCredentialRequest(password)
  state.password = password
  state.blind = blind
  ke1 = AuthClientStart(request)
  return ke1
```

6.2.2. GenerateKE2

The `GenerateKE2` function continues the AKE protocol by processing the client's `KE1` message and producing the server's `KE2` output.

```
GenerateKE2

State:
- state, a ServerState structure.

Input:
- server_identity, the optional encoded server identity, which is
  set to server_public_key if not specified.
- server_private_key, the server's private key.
- server_public_key, the server's public key.
- record, the client's RegistrationRecord structure.
- credential_identifier, an identifier that uniquely represents
  the credential.
- oprf_seed, the server-side seed of Nh bytes used to generate
  an oprf_key.
- ke1, a KE1 message structure.
- client_identity, the optional encoded client identity, which is
  set to client_public_key if not specified.

Output:
- ke2, a KE2 structure.

def GenerateKE2(server_identity, server_private_key,
                server_public_key, record, credential_identifier,
                oprf_seed, ke1, client_identity):
  credential_response =
    CreateCredentialResponse(ke1.credential_request,
                             server_public_key, record,
                             credential_identifier, oprf_seed)
  cleartext_credentials =
    CreateCleartextCredentials(server_public_key,
                               record.client_public_key,
                               server_identity, client_identity)
  auth_response =
    AuthServerRespond(cleartext_credentials, server_private_key,
                      record.client_public_key, ke1,
                      credential_response)

  ke2 = KE2 {
    credential_response,
    auth_response
  }

  return ke2
```

6.2.3. GenerateKE3

The `GenerateKE3` function completes the AKE protocol for the client and produces the client's `KE3` output for the server, as well as the `session_key` and `export_key` outputs from the AKE.

```
GenerateKE3

State:
- state, a ClientState structure.

Input:
- client_identity, the optional encoded client identity, which is
  set to client_public_key if not specified.
- server_identity, the optional encoded server identity, which is
  set to server_public_key if not specified.
- ke2, a KE2 message structure.

Output:
- ke3, a KE3 message structure.
- session_key, the session's shared secret.
- export_key, an additional client key.

def GenerateKE3(client_identity, server_identity, ke2):
  (client_private_key, cleartext_credentials, export_key) =
    RecoverCredentials(state.password, state.blind,
                       ke2.credential_response,
                       server_identity, client_identity)
  (ke3, session_key) =
    AuthClientFinalize(cleartext_credentials,
                       client_private_key, ke2)
  return (ke3, session_key, export_key)
```

6.2.4. ServerFinish

The `ServerFinish` function completes the AKE protocol for the server, yielding the `session_key`. Since the OPRF is a two-message protocol, `KE3` has no element of the OPRF. Therefore, `KE3` invokes the AKE's `AuthServerFinalize` directly. The `AuthServerFinalize` function takes `KE3` as input and **MUST** verify the client authentication material it contains before the `session_key` value can be used. This verification is necessary to ensure forward secrecy against active attackers.

```
ServerFinish

State:
- state, a ServerState structure.

Input:
- ke3, a KE3 structure.

Output:
- session_key, the shared session secret if and only if ke3 is valid.

def ServerFinish(ke3):
  return AuthServerFinalize(ke3)
```

This function **MUST NOT** return the `session_key` value if the client authentication material is invalid and may instead return an appropriate error message, such as `ClientAuthenticationError`, which is invoked from `AuthServerFinalize`.

6.3. Credential Retrieval

This section describes the sub-protocol run during authentication to retrieve and recover the client credentials.

6.3.1. Credential Retrieval Messages

This section describes the `CredentialRequest` and `CredentialResponse` messages exchanged between client and server to perform credential retrieval.

```
struct {
  uint8 blinded_message[Noe];
} CredentialRequest;
```

blinded_message: A serialized OPRF group element.

```
struct {
  uint8 evaluated_message[Noe];
  uint8 masking_nonce[Nn];
  uint8 masked_response[Npk + Nn + Nm];
} CredentialResponse;
```

evaluated_message: A serialized OPRF group element.

masking_nonce: A nonce used for the confidentiality of the `masked_response` field.

`masked_response`: An encrypted form of the server's public key and the client's `Envelope` structure.

6.3.2. Credential Retrieval Functions

This section describes the `CreateCredentialRequest`, `CreateCredentialResponse`, and `RecoverCredentials` functions used for credential retrieval.

6.3.2.1. CreateCredentialRequest

The `CreateCredentialRequest` is used by the client to initiate the credential retrieval process, and it produces a `CredentialRequest` message and OPRF state. Like `CreateRegistrationRequest`, this function can fail with an `InvalidInputError` error with negligible probability. However, this should not occur since registration (via `CreateRegistrationRequest`) will fail when provided the same password input.

```
CreateCredentialRequest

Input:
- password, an opaque byte string containing the client's password.

Output:
- request, a CredentialRequest structure.
- blind, an OPRF scalar value.

Exceptions:
- InvalidInputError, when Blind fails

def CreateCredentialRequest(password):
  (blind, blinded_element) = Blind(password)
  blinded_message = SerializeElement(blinded_element)
  request = CredentialRequest {
    blinded_message
  }
  return (request, blind)
```

6.3.2.2. CreateCredentialResponse

The `CreateCredentialResponse` function is used by the server to process the client's `CredentialRequest` message and complete the credential retrieval process, producing a `CredentialResponse`.

There are two scenarios to handle for the construction of a `CredentialResponse` object: either the `record` for the client exists (corresponding to a properly registered client) or it was never created (corresponding to an unregistered client identity, possibly the result of an enumeration attack attempt).

In the case of an existing `record` with the corresponding identifier `credential_identifier`, the server invokes the following function to produce a `CredentialResponse`:

```
CreateCredentialResponse

Input:
- request, a CredentialRequest structure.
- server_public_key, the public key of the server.
- record, an instance of RegistrationRecord which is the server's
  output from registration.
- credential_identifier, an identifier that uniquely represents
  the credential.
- oprf_seed, the server-side seed of Nh bytes used to generate
  an oprf_key.

Output:
- response, a CredentialResponse structure.

Exceptions:
- DeserializeError, when OPRF element deserialization fails.

def CreateCredentialResponse(request, server_public_key, record,
                             credential_identifier, oprf_seed):
  seed =
    Expand(oprf_seed, concat(credential_identifier, "OprfKey"), Nok)
  (oprf_key, _) = DeriveKeyPair(seed, "OPAQUE-DeriveKeyPair")

  blinded_element = DeserializeElement(request.blinded_message)
  evaluated_element = BlindEvaluate(oprf_key, blinded_element)
  evaluated_message = SerializeElement(evaluated_element)

  masking_nonce = random(Nn)
  credential_response_pad = Expand(record.masking_key,
                                   concat(masking_nonce,
                                   "CredentialResponsePad"),
                                   Npk + Nn + Nm)
  masked_response = xor(credential_response_pad,
                        concat(server_public_key, record.envelope))

  response = CredentialResponse {
    evaluated_message,
    masking_nonce,
    masked_response
  }

  return response
```

In the case of a `record` that does not exist and if client enumeration prevention is desired, the server **MUST** respond to the credential request to fake the existence of the `record`. The server **SHOULD** invoke the `CreateCredentialResponse` function with a fake client `record` argument that is configured so that:

- `record.client_public_key` is set to a randomly generated public key of length `Npk`
- `record.masking_key` is set to a random byte string of length `Nh`
- `record.envelope` is set to the byte string consisting only of zeros of length `Nn + Nm`

It is **RECOMMENDED** that a fake client `record` is created once (e.g., as the first user `record` of the application) and then stored alongside legitimate client `records` to serve subsequent client requests. This allows servers to retrieve the `record` in a time comparable to that of a legitimate client `record`.

Note that the responses output by either scenario are indistinguishable to an adversary that is unable to guess the registered password for the client corresponding to `credential_identifier`.

6.3.2.3. RecoverCredentials

The `RecoverCredentials` function is used by the client to process the server's `CredentialResponse` message and produce the client's private key, server public key, and the `export_key`.

```
RecoverCredentials

Input:
- password, an opaque byte string containing the client's password.
- blind, an OPRF scalar value.
- response, a CredentialResponse structure.
- server_identity, The optional encoded server identity.
- client_identity, The encoded client identity.

Output:
- client_private_key, the encoded client private key for
  the AKE protocol.
- cleartext_credentials, a CleartextCredentials structure.
- export_key, an additional client key.

Exceptions:
- DeserializeError, when OPRF element deserialization fails.

def RecoverCredentials(password, blind, response,
                       server_identity, client_identity):
  evaluated_element = DeserializeElement(response.evaluated_message)

  oprf_output = Finalize(password, blind, evaluated_element)
  stretched_oprf_output = Stretch(oprf_output)

  randomized_password =
    Extract("", concat(oprf_output, stretched_oprf_output))

  masking_key = Expand(randomized_password, "MaskingKey", Nh)

  credential_response_pad =
    Expand(masking_key,
           concat(response.masking_nonce, "CredentialResponsePad"),
           Npk + Nn + Nm)

  concat(server_public_key, envelope) =
    xor(credential_response_pad, response.masked_response)

  (client_private_key, cleartext_credentials, export_key) =
    Recover(randomized_password, server_public_key, envelope,
            server_identity, client_identity)

  return (client_private_key, cleartext_credentials, export_key)
```

6.4. 3DH Protocol

This section describes the authenticated key exchange protocol for OPAQUE using 3DH, a 3-message AKE that satisfies the forward secrecy and KCI properties discussed in Section 10.

The client AKE state `ClientAkeState` mentioned in Section 6 has the following fields:

client_secret: An opaque byte string of length `Nsk`.

ke1: A value of type `KE1`.

The server AKE state `ServerAkeState` mentioned in Section 6 has the following fields:

expected_client_mac: An opaque byte string of length `Nm`.

`session_key`: An opaque byte string of length `Nx`.

Sections 6.4.3 and 6.4.4 specify the inner workings of client and server functions, respectively.

6.4.1. 3DH Key Exchange Functions

We assume the following functions exist for all Diffie-Hellman key exchange variants:

DeriveDiffieHellmanKeyPair(seed): Derive a private and public Diffie-Hellman key pair deterministically from the input `seed`. The type of the private key depends on the implementation, whereas the type of the public key is a byte string of `Npk` bytes.

DiffieHellman(k, B): A function that performs the Diffie-Hellman operation between the private input `k` and public input `B`. The output of this function is a unique, fixed-length byte string.

It is **RECOMMENDED** to use Elliptic Curve Diffie-Hellman for this key exchange protocol. Implementations for recommended groups in Section 7, as well as groups covered by test vectors in Appendix C, are described in the following sections.

6.4.1.1. 3DH ristretto255

This section describes the implementation of the Diffie-Hellman key exchange functions based on ristretto255 as defined in [RFC9496].

DeriveDiffieHellmanKeyPair(seed): This function is implemented as DeriveKeyPair(seed, "OPAQUE-DeriveDiffieHellmanKeyPair"), where DeriveKeyPair is as specified in Section 3.2 of [RFC9497]. The public value from DeriveKeyPair is encoded using SerializeElement from Section 2.1 of [RFC9497].

DiffieHellman(k, B): Implemented as scalar multiplication as described in [RFC9496] after decoding `B` from its encoded input using the Decode function in Section 4.3.1 of [RFC9496]. The output is then encoded using the SerializeElement function of the OPRF group described in Section 2.1 of [RFC9497].

6.4.1.2. 3DH P-256

This section describes the implementation of the Diffie-Hellman key exchange functions based on NIST P-256 as defined in [NISTCurves].

DeriveDiffieHellmanKeyPair(seed): As defined in Section 6.4.1.1.

DiffieHellman(k, B): Implemented as scalar multiplication as described in [NISTCurves] after decoding `B` from its encoded input using the compressed Octet-String-to-Elliptic-Curve-Point method according to [NISTCurves]. The output is then encoded using the SerializeElement function of the OPRF group described in Section 2.1 of [RFC9497].

6.4.1.3. 3DH Curve25519

This section describes the implementation of the Diffie-Hellman key exchange functions based on Curve25519 as defined in [RFC7748].

DeriveDiffieHellmanKeyPair(seed): This function is implemented by returning the private key `k` based on `seed` (of length `Nseed = 32` bytes) as described in Section 5 of [RFC7748], as well as the result of DiffieHellman(k, B), where B is the base point of Curve25519.

DiffieHellman(k, B): Implemented using the X25519 function in Section 5 of [RFC7748]. The output is then used raw with no processing.

6.4.2. Key Schedule Functions

This section contains functions used for the AKE key schedule.

6.4.2.1. Transcript Functions

The OPAQUE-3DH key derivation procedures make use of the functions below that are repurposed from TLS 1.3 [RFC8446].

```
Expand-Label(Secret, Label, Context, Length) =
    Expand(Secret, CustomLabel, Length)
```

Where CustomLabel is specified and encoded (following Section 3.4 of [RFC8446]) as:

```
struct {
  uint16 length = Length;
  opaque label<8..255> = "OPAQUE-" + Label;
  uint8 context<0..255> = Context;
} CustomLabel;

Derive-Secret(Secret, Label, Transcript-Hash) =
    Expand-Label(Secret, Label, Transcript-Hash, Nx)
```

Note that the `Label` parameter is not a NULL-terminated string.

OPAQUE-3DH can optionally include application-specific, shared `context` information in the transcript, such as configuration parameters or application-specific information, e.g., "appXYZ-v1.2.3".

The OPAQUE-3DH key schedule requires a `preamble`, which is computed as follows.

```
Preamble

Parameters:
- context, optional shared context information.

Input:
- client_identity, the optional encoded client identity, which is set
  to client_public_key if not specified.
- ke1, a KE1 message structure.
- server_identity, the optional encoded server identity, which is set
  to server_public_key if not specified.
- credential_response, the corresponding field on the KE2 structure.
- server_nonce, the corresponding field on the AuthResponse
  structure.
- server_public_keyshare, the corresponding field on the AuthResponse
  structure.

Output:
- preamble, the protocol transcript with identities and messages.

def Preamble(client_identity, ke1, server_identity,
             credential_response, server_nonce,
             server_public_keyshare):
  preamble = concat("OPAQUEv1-",
                    I2OSP(len(context), 2), context,
                    I2OSP(len(client_identity), 2), client_identity,
                    ke1,
                    I2OSP(len(server_identity), 2), server_identity,
                    credential_response,
                    server_nonce,
                    server_public_keyshare)
  return preamble
```

6.4.2.2. Shared Secret Derivation

The OPAQUE-3DH shared secret derived during the key exchange protocol is computed using the following helper function.

```
DeriveKeys

Input:
- ikm, input key material.
- preamble, the protocol transcript with identities and messages.

Output:
- Km2, a MAC authentication key.
- Km3, a MAC authentication key.
- session_key, the shared session secret.

def DeriveKeys(ikm, preamble):
  prk = Extract("", ikm)
  handshake_secret =
    Derive-Secret(prk, "HandshakeSecret",Hash(preamble))
  session_key =
    Derive-Secret(prk, "SessionKey", Hash(preamble))
  Km2 = Derive-Secret(handshake_secret, "ServerMAC", "")
  Km3 = Derive-Secret(handshake_secret, "ClientMAC", "")
  return (Km2, Km3, session_key)
```

6.4.3. 3DH Client Functions

The `AuthClientStart` function is used by the client to create a `KE1` structure.

```
AuthClientStart

Parameters:
- Nn, the nonce length.

State:
- state, a ClientAkeState structure.

Input:
- credential_request, a CredentialRequest structure.

Output:
- ke1, a KE1 structure.

def AuthClientStart(credential_request):
  client_nonce = random(Nn)
  client_keyshare_seed = random(Nseed)
  (client_secret, client_public_keyshare) =
    DeriveDiffieHellmanKeyPair(client_keyshare_seed)

  auth_request = AuthRequest {
    client_nonce,
    client_public_keyshare
  }

  ke1 = KE1 {
    credential_request,
    auth_request
  }

  state.client_secret = client_secret
  state.ke1 = ke1
  return ke1
```

The `AuthClientFinalize` function is used by the client to create a `KE3` message and output `session_key` using the server's `KE2` message and recovered credential information.

```
AuthClientFinalize

State:
- state, a ClientAkeState structure.

Input:
- cleartext_credentials, a CleartextCredentials structure.
- client_private_key, the client's private key.
- ke2, a KE2 message structure.

Output:
- ke3, a KE3 structure.
- session_key, the shared session secret.

Exceptions:
- ServerAuthenticationError, the handshake fails.

def AuthClientFinalize(cleartext_credentials,
                       client_private_key, ke2):

  dh1 = DiffieHellman(state.client_secret,
                      ke2.auth_response.server_public_keyshare)
  dh2 = DiffieHellman(state.client_secret,
                      cleartext_credentials.server_public_key)
  dh3 = DiffieHellman(client_private_key,
                      ke2.auth_response.server_public_keyshare)
  ikm = concat(dh1, dh2, dh3)

  preamble = Preamble(cleartext_credentials.client_identity,
                      state.ke1,
                      cleartext_credentials.server_identity,
                      ke2.credential_response,
                      ke2.auth_response.server_nonce,
                      ke2.auth_response.server_public_keyshare)
  Km2, Km3, session_key = DeriveKeys(ikm, preamble)
  expected_server_mac = MAC(Km2, Hash(preamble))
  if !ct_equal(ke2.auth_response.server_mac, expected_server_mac),
    raise ServerAuthenticationError
  client_mac = MAC(Km3, Hash(concat(preamble, expected_server_mac)))
  ke3 = KE3 {
    client_mac
  }
  return (ke3, session_key)
```

6.4.4. 3DH Server Functions

The `AuthServerRespond` function is used by the server to process the client's `KE1` message and public credential information to create a `KE2` message.

```
AuthServerRespond

Parameters:
- Nn, the nonce length.

State:
```

```
- state, a ServerAkeState structure.

Input:
- cleartext_credentials, a CleartextCredentials structure.
- server_private_key, the server's private key.
- client_public_key, the client's public key.
- ke1, a KE1 message structure.

Output:
- auth_response, an AuthResponse structure.

def AuthServerRespond(cleartext_credentials, server_private_key,
                      client_public_key, ke1, credential_response):
  server_nonce = random(Nn)
  server_keyshare_seed = random(Nseed)
  (server_private_keyshare, server_public_keyshare) =
    DeriveDiffieHellmanKeyPair(server_keyshare_seed)
  preamble = Preamble(cleartext_credentials.client_identity,
                      ke1,
                      cleartext_credentials.server_identity,
                      credential_response,
                      server_nonce,
                      server_public_keyshare)

  dh1 = DiffieHellman(server_private_keyshare,
                      ke1.auth_request.client_public_keyshare)
  dh2 = DiffieHellman(server_private_key,
                      ke1.auth_request.client_public_keyshare)
  dh3 = DiffieHellman(server_private_keyshare,
                      client_public_key)
  ikm = concat(dh1, dh2, dh3)

  Km2, Km3, session_key = DeriveKeys(ikm, preamble)
  server_mac = MAC(Km2, Hash(preamble))

  state.expected_client_mac =
    MAC(Km3, Hash(concat(preamble, server_mac)))
  state.session_key = session_key

  auth_response = AuthResponse {
    server_nonce,
    server_public_keyshare,
    server_mac
  }

  return auth_response
```

The `AuthServerFinalize` function is used by the server to process the client's `KE3` message and output the final `session_key`.

```
AuthServerFinalize

State:
- state, a ServerAkeState structure.

Input:
- ke3, a KE3 structure.

Output:
- session_key, the shared session secret if and only if ke3 is valid.

Exceptions:
- ClientAuthenticationError, the handshake fails.

def AuthServerFinalize(ke3):
  if !ct_equal(ke3.client_mac, state.expected_client_mac):
    raise ClientAuthenticationError
  return state.session_key
```

7. Configurations

An OPAQUE-3DH configuration is a tuple (OPRF, KDF, MAC, Hash, KSF, Group, Context) such that the following conditions are met:

- The OPRF protocol uses the `modeOPRF` configuration in Section 3.1 of [RFC9497] and implements the interface in Section 2. Examples include ristretto255-SHA512 and P256-SHA256.
- The KDF, MAC, and Hash functions implement the interfaces in Section 2. Examples include HKDF [RFC5869] for the KDF, HMAC [RFC2104] for the MAC, and SHA-256 and SHA-512 for the Hash functions. If an extensible output function such as SHAKE128 [FIPS202] is used, then the output length `Nh` **MUST** be chosen to align with the target security level of the OPAQUE configuration. For example, if the target security parameter for the configuration is 128 bits, then `Nh` **SHOULD** be at least 32 bytes.
- The KSF is determined by the application and implements the interface in Section 2. As noted, collision resistance is required. Examples for KSF include Argon2id [RFC9106], scrypt [RFC7914], and PBKDF2 [RFC8018] with fixed parameter choices. See Section 8 for more information about this choice of function.
- The Group mode identifies the group used in the OPAQUE-3DH AKE. This **SHOULD** match that of the OPRF. For example, if the OPRF is ristretto255-SHA512, then Group **SHOULD** be ristretto255.

Context is the shared parameter used to construct the `preamble` in Section 6.4.2.1. This parameter **SHOULD** include any application-specific configuration information or parameters that are needed to prevent cross-protocol or downgrade attacks.

Absent an application-specific profile, the following configurations are **RECOMMENDED**:

- ristretto255-SHA512, HKDF-SHA-512, HMAC-SHA-512, SHA-512, Argon2id(S = zeroes(16), p = 4, T = `Nh`, m = 2^21, t = 1, v = 0x13, K = nil, X = nil, y = 2), ristretto255
- P256-SHA256, HKDF-SHA-256, HMAC-SHA-256, SHA-256, Argon2id(S = zeroes(16), p = 4, T = `Nh`, m = 2^21, t = 1, v = 0x13, K = nil, X = nil, y = 2), P-256
- P256-SHA256, HKDF-SHA-256, HMAC-SHA-256, SHA-256, scrypt(S = zeroes(16), N = 32768, r = 8, p = 1, dkLen = 32), P-256

The above recommended configurations target 128-bit security.

Future configurations may specify different combinations of dependent algorithms with the following considerations:

1. The size of AKE public and private keys -- `Npk` and `Nsk`, respectively -- must adhere to the output length limitations of the KDF Expand function. If HKDF is used, this means `Npk`, `Nsk` <= 255 * `Nx`, where `Nx` is the output size of the underlying hash function. See [RFC5869] for details.
2. The output size of the Hash function **SHOULD** be long enough to produce a key for MAC of suitable length. For example, if MAC is HMAC-SHA256, then `Nh` could be 32 bytes.

8. Application Considerations

Beyond choosing an appropriate configuration, there are several parameters that applications can use to control OPAQUE:

- Credential identifier: As described in Section 5, this is a unique handle to the client's credential being stored. In applications where there are alternate client identities that accompany an account, such as a username or email address, this identifier can be set to those alternate values. For simplicity, applications may choose to set `credential_identifier` to be equal to `client_identity`. Applications **MUST NOT** use the same credential identifier for multiple clients.
- Context information: As described in Section 7, applications may include a shared `context` string that is authenticated as part of the handshake. This parameter **SHOULD** include any configuration information or parameters that are needed to prevent cross-protocol or downgrade attacks. This `context` information is not sent over the wire in any key exchange messages. However, applications may choose to send it alongside key exchange messages if needed for their use case.
- Client and server identities: As described in Section 4, clients and servers are identified with their public keys by default. However, applications may choose alternate identities that are pinned to these public keys. For example, servers may use a domain name instead of a public key as their identifier. Absent alternate notions of identity, applications **SHOULD** set these identities to nil and rely solely on public key information.

- Configuration and envelope updates: Applications may wish to update or change their configuration or other parameters that affect the client's `RegistrationRecord` over time. Some reasons for changing these are to use different cryptographic algorithms, e.g., a different KSF with improved parameters, or to update key material that is cryptographically bound to the `RegistrationRecord`, such as the server's public key (`server_public_key`). Any such change will require users to reregister to create a new `RegistrationRecord`. Supporting these types of updates can be helpful for applications that anticipate such changes in their deployment setting.
- Password hardening parameters: Key stretching is done to help prevent password disclosure in the event of server compromise; see Section 10.8. There is no ideal or default set of parameters, though relevant specifications for KSFs give some reasonable defaults.
- Enumeration prevention: If servers receive a credential request for a non-existent client, they **SHOULD** respond with a "fake" response to prevent active client enumeration attacks as described in Section 6.3.2.2. Servers that implement this mitigation **SHOULD** use the same configuration information (such as the `oprf_seed`) for all clients; see Section 10.9. In settings where this attack is not a concern, servers may choose to not support this functionality.
- Handling password changes: In the event of a password change, the client and server can run the registration phase using the new password as a fresh instance (ensuring to resample all random values). The resulting registration `record` can then replace the previous `record` corresponding to the client's old password registration.

9. Implementation Considerations

This section documents considerations for OPAQUE implementations. This includes implementation safeguards and error handling considerations.

9.1. Implementation Safeguards

Certain information created, exchanged, and processed in OPAQUE is sensitive. Specifically, all private key material and intermediate values, along with the outputs of the key exchange phase, are all secret. Implementations should not retain these values in memory when no longer needed. Moreover, all operations, particularly the cryptographic and group arithmetic operations, should be constant-time and independent of the bits of any secrets. This includes any conditional branching during the creation of the credential response as needed to mitigate client enumeration attacks.

As specified in Section 5 and Section 6, OPAQUE only requires the client password as input to the OPRF for registration and authentication. However, if `client_identity` can be bound to the client's registration `record` (i.e., the identity will not change during the lifetime of the `record`), then an implementation **SHOULD** incorporate `client_identity` alongside the password as input to the OPRF. Furthermore, it is **RECOMMENDED** to incorporate `server_identity` alongside the password as input to the OPRF. These additions provide domain separation for clients and servers; see Section 10.2.

9.2. Handling Online Guessing Attacks

Online guessing attacks (against any aPAKE) can be done from both the client side and the server side. In particular, a malicious server can attempt to simulate honest responses to learn the client's password. While this constitutes an exhaustive online attack (as expensive as a guessing attack from the client side), it can be mitigated when the channel between client and server is authenticated, e.g., using server-authenticated TLS. In such cases, these online attacks are limited to clients and the authenticated server itself. Moreover, such a channel provides privacy of user information, including identity and envelope values.

Additionally, note that a client participating in the online login stage will learn whether or not authentication is successful after receiving the `KE2` message. This means that the server should treat any client which fails to send a subsequent `KE3` message as an authentication failure. This can be handled in applications that wish to track authentication failures by, for example, assuming by default that any client authentication attempt is a failure unless a `KE3` message is received by the server and passes `ServerFinish` without error.

9.3. Error Considerations

Some functions included in this specification are fallible. For example, the authenticated key exchange protocol may fail because the client's password was incorrect or the authentication check failed, yielding an error. The explicit errors generated throughout this specification, along with conditions that lead to each error, are as follows:

- `EnvelopeRecoveryError`: The `Envelope Recover` function failed to produce any authentication key material; Section 4.1.3.
- ServerAuthenticationError: The client failed to complete the authenticated key exchange protocol with the server; Section 6.4.3.
- `ClientAuthenticationError`: The server failed to complete the authenticated key exchange protocol with the client; Section 6.4.4.

Beyond these explicit errors, OPAQUE implementations can produce implicit errors. For example, if protocol messages sent between client and server do not match their expected size, an implementation should produce an error. More generally, if any protocol message received from the peer is invalid, perhaps because the message contains an invalid public key (indicated by the AKE DeserializeElement function failing) or an invalid OPRF element (indicated by the OPRF DeserializeElement), then an implementation should produce an error.

The errors in this document are meant as a guide for implementors. They are not an exhaustive list of all the errors an implementation might emit. For example, an implementation might run out of memory.

10. Security Considerations

OPAQUE is defined as the composition of two functionalities: an OPRF and an AKE protocol. It can be seen as a "compiler" for transforming any AKE protocol (with Key Compromise Impersonation (KCI) security and forward secrecy; see Section 10.2) into a secure aPAKE protocol. In OPAQUE, the client derives a private key during password registration and retrieves this key each time it needs to authenticate to the server. The OPRF security properties ensure that only the correct password can unlock the private key while at the same time avoiding potential offline guessing attacks. This general composability property provides great flexibility and enables a variety of OPAQUE instantiations, from optimized performance to integration with existing authenticated key exchange protocols such as TLS.

10.1. Notable Design Differences

The specification as written here differs from the original cryptographic design in [JKX18Full] and the corresponding CFRG document [Krawczyk20], both of which were used as input to the CFRG PAKE competition. This section describes these differences, including their motivation and explanation as to why they preserve the provable security of OPAQUE based on [JKX18Full].

The following list enumerates important functional differences that were made as part of the protocol specification process to address application or implementation considerations.

- Clients construct envelope contents without revealing the password to the server, as described in Section 5, whereas the servers construct envelopes in [JKX18Full]. This change adds to the security of the protocol. [JKX18Full] considered the case where the envelope was constructed by the server for reasons of compatibility with previous Universal Composability (UC) security modeling. [HJKW23] analyzes the registration phase as specified in this document. This change was made to support registration flows where the client chooses the password and wishes to keep it secret from the server, and it is compatible with the variant in [JKX18Full] that was originally analyzed.
- Envelopes do not contain encrypted credentials. Instead, envelopes contain information used to derive client private key material for the AKE. This change improves the assumption behind the protocol by getting rid of equivocality and random key robustness for the encryption function. The random-key robustness property defined in Section 2.2 is only needed for the MAC function. This change was made for two reasons. First, it reduces the number of bytes stored in envelopes, which is a helpful improvement for large applications of OPAQUE with many registered users. Second, it removes the need for client applications to generate private keys during registration. Instead, this responsibility is handled by OPAQUE, thereby simplifying the client interface to the protocol.
- Envelopes are masked with a per-user masking key as a way of preventing client enumeration attacks. See Section 10.9 for more details. This extension is not needed for the security of OPAQUE as an aPAKE protocol, but is only used to provide a defense against

enumeration attacks. In the analysis, the masking key can be simulated as a (pseudo) random key. This change was made to support real-world use cases where client or user enumeration is a security (or privacy) risk.

- Per-user OPRF keys are derived from a client identity and cross-user PRF `seed` as a mitigation against client enumeration attacks. See Section 10.9 for more details. The analysis of OPAQUE assumes OPRF keys of different users are independently random or pseudorandom. Deriving these keys via a single PRF (i.e., with a single cross-user key) applied to users' identities satisfies this assumption. This change was made to support real-world use cases where client or user enumeration is a security (or privacy) risk. Note that the derivation of the OPRF key via a PRF keyed by `oprf_seed` and applied to the unique `credential_identifier` ensures the critical requirement of the per-user OPRF keys being unique per client.
- The protocol outputs an export key for the client in addition to a shared session key that can be used for application-specific purposes. This key is a pseudorandom value derived from the client password (among other values) and has no influence on the security analysis (it can be simulated with a random output). This change was made to support more application use cases for OPAQUE, such as the use of OPAQUE for end-to-end encrypted backups; see [WhatsAppE2E].
- The AKE protocol describes a 3-message protocol where the third message includes client authentication material that the server is required to verify. This change (from the original 2-message protocol) was made to provide explicit client authentication and full forward security. The 3-message protocol is analyzed in [JKX18Full].
- The protocol admits optional application-layer client and server identities. In the absence of these identities, the client and server are authenticated against their public keys. Binding authentication to identities is part of the AKE part of OPAQUE. The type of identities and their semantics are application-dependent and independent of the protocol analysis. This change was made to simplify client and server interfaces to the protocol by removing the need to specify additional identities alongside their corresponding public authentication keys when not needed.
- The protocol admits application-specific `context` information configured out-of-band in the AKE transcript. This allows domain separation between different application uses of OPAQUE. This is a mechanism for the AKE component and is best practice for domain separation between different applications of the protocol. This change was made to allow different applications to use OPAQUE without the risk of cross-protocol attacks.
- Servers use a separate identifier for computing OPRF evaluations and indexing into the registration `record` storage called the `credential_identifier`. This allows clients to change their application-layer identity (`client_identity`) without inducing server-side changes, e.g., by changing an email address associated with a given account. This mechanism is part of the derivation of OPRF keys via a single PRF. As long as the derivation of different OPRF keys from a single PRF has different PRF inputs, the protocol is secure. The choice of such inputs is up to the application.
- [JKX18Full] comments on a defense against offline dictionary attacks upon server compromise or honest-but-curious servers. The authors suggest implementing the OPRF phase as a threshold OPRF [TOPPSS], effectively forcing an attacker to act online or control

at least t key shares (of the total n), where t is the threshold number of shares necessary to recombine the secret OPRF key. Only then would an attacker be able to run an offline dictionary attack. This implementation only affects the server and changes nothing for the client. Furthermore, if the threshold OPRF servers holding these keys are separate from the authentication server, then recovering all n shares would still not suffice to run an offline dictionary attack without access to the client `record` database. However, this mechanism is out of scope for this document.

The following list enumerates notable differences and refinements from the original cryptographic design in [JKX18Full] and the corresponding CFRG document [Krawczyk20] that were made to make this specification suitable for interoperable implementations.

- [JKX18Full] used a generic prime-order group for the DH-OPRF and HMQV operations, and includes necessary prime-order subgroup checks when receiving attacker-controlled values over the wire. This specification instantiates the prime-order group used for 3DH using prime-order groups based on elliptic curves as described in Section 2.1 of [RFC9497]. This specification also delegates OPRF group choice and operations to Section 4 of [RFC9497]. As such, the prime-order group as used in the OPRF and 3DH as specified in this document both adhere to the requirements in [JKX18Full].
- [JKX18Full] specified DH-OPRF (see Appendix B) to instantiate the OPRF functionality in the protocol. A critical part of DH-OPRF is the hash-to-group operation, which was not instantiated in the original analysis. However, the requirements for this operation were included. This specification instantiates the OPRF functionality based on Section 3.3.1 of [RFC9497], which is identical to the DH-OPRF functionality in [JKX18Full] and, concretely, uses the hash-to-curve functions in [RFC9380]. All hash-to-curve methods in [RFC9380] are compliant with the requirement in [JKX18Full], namely, that the output be a member of the prime-order group.
- [JKX18Full] and [Krawczyk20] both used HMQV as the AKE for the protocol. However, this document fully specifies 3DH instead of HMQV (though a sketch for how to instantiate OPAQUE using HMQV is included in Appendix B.1). Since 3DH satisfies the essential requirements for the AKE protocol as described in [JKX18Full] and [Krawczyk20], as recalled in Section 10.2, this change preserves the overall security of the protocol. 3DH was chosen for its simplicity and ease of implementation.
- The DH-OPRF and HMQV instantiation of OPAQUE as shown in Figure 12 [JKX18Full] uses a different transcript than that which is described in this specification. In particular, the key exchange transcript specified in Section 6.4 is a superset of the transcript as defined in [JKX18Full]. This was done to align with best practices, like what is done for key exchange protocols like TLS 1.3 [RFC8446].
- Neither [JKX18Full] nor [Krawczyk20] included wire format details for the protocol, which is essential for interoperability. This specification fills this gap by including such wire format details and corresponding test vectors; see Appendix C.

10.2. Security Analysis

Jarecki et al. [JKX18Full] proved the security of OPAQUE (modulo the design differences outlined in Section 10.1) in a strong aPAKE model that ensures security against precomputation attacks and is formulated in the UC framework [Canetti01] under the random oracle model. This assumes security of the OPRF function and the underlying key exchange protocol.

OPAQUE's design builds on a line of work initiated in the seminal paper of Ford and Kaliski [FK00] and is based on the HPAKE protocol of Xavier Boyen [Boyen09] and the (1,1)-PPSS protocol from Jarecki et al. [JKKX16]. None of these papers considered security against precomputation attacks or presented a proof of aPAKE security (not even in a weak model).

The KCI property required from AKE protocols for use with OPAQUE states that knowledge of a party's private key does not allow an attacker to impersonate others to that party. This is an important security property achieved by most public-key-based AKE protocols, including protocols that use signatures or public key encryption for authentication. It is also a property of many implicitly authenticated protocols, e.g., HMQV, but not all of them. We also note that key exchange protocols based on shared keys do not satisfy the KCI requirement, hence they are not considered in the OPAQUE setting. We note that KCI is needed to ensure a crucial property of OPAQUE. Even upon compromise of the server, the attacker cannot impersonate the client to the server without first running an exhaustive dictionary attack. Another essential requirement from AKE protocols for use in OPAQUE is to provide forward secrecy (against active attackers).

In [JKX18Full], security is proven for one instance (i.e., one key) of the OPAQUE protocol, and without batching. There is currently no security analysis available for the OPAQUE protocol described in this document in a setting with multiple server keys or batching.

As stated in Section 9.1, incorporating `client_identity` adds domain separation, particularly against servers that choose the same OPRF key for multiple clients. The `client_identity` as input to the OPRF also acts as a key identifier that would be required for a proof of the protocol in the multi-key setting; the OPAQUE analysis in [JKX18Full] assumes single server-key instances. Adding `server_identity` to the OPRF input provides domain separation for clients that reuse the same `client_identity` across different server instantiations.

10.3. Identities

AKE protocols generate keys that need to be uniquely and verifiably bound to a pair of identities. In the case of OPAQUE, those identities correspond to `client_identity` and `server_identity`. Thus, it is essential for the parties to agree on such identities, including an agreed bit representation of these identities as needed.

Note that the method of transmission of `client_identity` from client to server is outside the scope of this protocol and it is up to an application to choose how this identity should be delivered (for instance, alongside the first OPAQUE message or agreed upon before the OPAQUE protocol begins).

Applications may have different policies about how and when identities are determined. A natural approach is to tie `client_identity` to the identity the server uses to fetch the envelope (determined during password registration) and tie `server_identity` to the server identity used by the client to initiate an offline password registration or online authenticated key exchange session. `server_identity` and `client_identity` can also be part of the envelope or tied to the parties' public keys. In principle, identities may change across different sessions as long as there is a policy that can establish if the identity is acceptable or not to the peer. However, we note that the public keys of both the server and the client must always be those defined at the time of password registration.

The client identity (`client_identity`) and server identity (`server_identity`) are optional parameters that are left to the application to designate as aliases for the client and server. If the application layer does not supply values for these parameters, then they will be omitted from the creation of the envelope during the registration stage. Furthermore, they will be substituted with `client_identity` = `client_public_key` and `server_identity` = `server_public_key` during the authenticated key exchange stage.

The advantage of supplying a custom `client_identity` and `server_identity` (instead of simply relying on a fallback to `client_public_key` and `server_public_key`) is that the client can then ensure that any mappings between `client_identity` and `client_public_key` (and `server_identity` and `server_public_key`) are protected by the authentication from the envelope. Then, the client can verify that the `client_identity` and `server_identity` contained in its envelope match the `client_identity` and `server_identity` supplied by the server.

However, if this extra layer of verification is unnecessary for the application, then simply leaving `client_identity` and `server_identity` unspecified (and using `client_public_key` and `server_public_key` instead) is acceptable.

10.4. Export Key Usage

The export key can be used (separately from the OPAQUE protocol) to provide confidentiality and integrity to other data that only the client should be able to process. For instance, if the client wishes to store secrets with a third party, then this export key can be used by the client to encrypt these secrets so that they remain hidden from a passive adversary that does not have access to the server's secret keys or the client's password.

10.5. Static Diffie-Hellman Oracles

While one can expect the practical security of the OPRF function (namely, the hardness of computing the function without knowing the key) to be in the order of computing discrete logarithms or solving Diffie-Hellman, Brown and Gallant [BG04] and Cheon [Cheon06] show an attack that slightly improves on generic attacks. For typical curves, the attack requires an infeasible number of calls to the OPRF or results in insignificant security loss; see Section 7.2.3 of [RFC9497] for more information. For OPAQUE, these attacks are particularly impractical as they translate into an infeasible number of failed authentication attempts directed at individual users.

10.6. Random-Key Robust MACs

The random-key robustness property for a MAC states that, given two random keys k1 and k2, it is infeasible to find a message m such that MAC(k1, m) = MAC(k2, m). Note that in general, not every MAC function is key-robust. In particular, GMAC (which underlies GCM) does not satisfy key-robustness, whereas HMAC with a collision-resistant hash function does satisfy key-robustness.

An application can choose to use a non-key-robust MAC within the AKE portion of the protocol described in Section 6.4, but it **MUST** use a key-robust MAC for the creation of the `auth_tag` parameter in Section 4.1.2.

10.7. Input Validation

Both client and server **MUST** validate the other party's public key(s) used for the execution of OPAQUE. This includes the keys shared during the registration phase, as well as any keys shared during the online key agreement phase. The validation procedure varies depending on the type of key. For example, for OPAQUE instantiations using 3DH with P-256, P-384, or P-521 as the underlying group, validation is as specified in Section 5.6.2.3.4 of [keyagreement]. This includes checking that the coordinates are in the correct range, that the point is on the curve, and that the point is not the point at infinity. Additionally, validation **MUST** ensure the Diffie-Hellman shared secret is not the point at infinity.

10.8. OPRF Key Stretching

Applying a key stretching function to the output of the OPRF greatly increases the cost of an offline attack upon the compromise of the credential file on the server. Applications **SHOULD** select parameters for the KSF that balance cost and complexity across different client implementations and deployments. Note that in OPAQUE, the key stretching function is executed by the client as opposed to the server in common password hashing scenarios. This means that applications must consider a tradeoff between the performance of the protocol on clients (specifically low-end devices) and protection against offline attacks after a server compromise.

10.9. Client Enumeration

Client enumeration refers to attacks where the attacker tries to learn whether a given user identity is registered with a server or whether a reregistration or change of password was performed for that user. OPAQUE counters these attacks by requiring servers to act with unregistered client identities in a way that is indistinguishable from their behavior with existing registered clients. Servers do this by simulating a fake `CredentialResponse` as specified in Section 6.3.2.2 for unregistered users and encrypting `CredentialResponse` using a masking key. In this way, real and fake `CredentialResponse` messages are indistinguishable from one another. Implementations must also take care to avoid side-channel leakage (e.g., timing attacks) from helping differentiate these operations from a regular server response. Note that this may introduce possible abuse vectors since the server's cost of generating a `CredentialResponse` is less than that of the client's cost of generating a `CredentialRequest`. Server implementations

may choose to forego the construction of a simulated credential response message for an unregistered client if these client enumeration attacks can be mitigated through other application-specific means or are otherwise not applicable for their threat model.

OPAQUE does not prevent this type of attack during the registration flow. Servers necessarily react differently during the registration flow between registered and unregistered clients. This allows an attacker to use the server's response during registration as an oracle for whether a given client identity is registered. Applications should mitigate against this type of attack by rate limiting or otherwise restricting the registration flow.

Finally, applications that do not require protection against client enumeration attacks can choose to derive independent OPRF keys for different clients. The advantage to using independently-derived OPRF keys is that the server avoids keeping the `oprf_seed` value across different clients, which, if leaked, would compromise the security for all clients reliant on `oprf_seed` as noted in [DL24].

10.10. Protecting the Registration Masking Key

The user enumeration prevention method described in this document uses a symmetric encryption key, `masking_key`, generated and sent to the server by the client during registration. This requires a confidential channel between client and server during registration, e.g., using TLS [RFC8446]. If the channel is only authenticated (this is a requirement for correct identification of the parties), a confidential channel can be established using public-key encryption, e.g., with HPKE [RFC9180]. However, the details of this mechanism are out of scope for this document.

10.11. Password Salt and Storage Implications

In OPAQUE, the OPRF key acts as the secret `salt` value that ensures the infeasibility of precomputation attacks. No extra `salt` value is needed. Also, clients never disclose their passwords to the server, even during registration. Note that a corrupted server can run an exhaustive offline dictionary attack to validate guesses for the client's password; this is inevitable in any (single-server) aPAKE protocol. It can be avoided in the case of OPAQUE by resorting to a multi-server threshold OPRF implementation, e.g., [TOPPSS]. Furthermore, if the server does not sample the PRF `seed` with sufficiently high entropy, or if it is not kept hidden from an adversary, then any derivatives from the client's password may also be susceptible to an offline dictionary attack to recover the original password.

Some applications may require learning the client's password to enforce password rules. Doing so invalidates this important security property of OPAQUE and is **NOT RECOMMENDED** unless it is not possible for applications to move such checks to the client. Note that limited checks at the server are possible to implement, e.g., detecting repeated passwords upon reregistrations or password change.

In general, passwords should be selected with sufficient entropy to avoid being susceptible to recovery through dictionary attacks, both online and offline.

10.12. AKE Private Key Storage

Server implementations of OPAQUE do not need access to the raw AKE private key. They only require the ability to compute shared secrets as specified in Section 6.4.2. Thus, applications may store the server AKE private key in a Hardware Security Module (HSM) or similar. Upon compromise of `oprf_seed` and client envelopes, this would prevent an attacker from using this data to mount a server spoofing attack. Supporting implementations need to consider allowing separate AKE and OPRF algorithms in cases where the HSM is incompatible with the OPRF algorithm.

10.13. Client Authentication Using Credentials

For scenarios in which the client has access to private state that can be persisted across registration and login, the client can back up the `randomized_password` variable (as computed in Section 5.2.3) so that upon a future login attempt, the client can authenticate to the server using `randomized_password` instead of the original password. This can be achieved by supplying an arbitrary password as input to `CreateCredentialRequest` in the login phase, and then using `randomized_password` from the backup in `RecoverCredentials` (invoked by `GenerateKE3`) rather than computing it from the password.

This provides an advantage over the regular authentication flow for login in that if `randomized_password` is compromised, an adversary cannot use this value to successfully impersonate the server to the client during login. The drawback is that it is only applicable to settings where `randomized_password` can be treated as a credential that can be stored securely after registration and retrieved upon login.

11. IANA Considerations

This document has no IANA actions.

12. References

12.1. Normative References

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <https://www.rfc-editor.org/info/rfc2104>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/info/rfc4086>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC9497] Davidson, A., Faz-Hernandez, A., Sullivan, N., and C. A. Wood, "Oblivious Pseudorandom Functions (OPRFs) Using Prime-Order Groups", RFC 9497, DOI 10.17487/RFC9497, December 2023, <https://www.rfc-editor.org/info/rfc9497>.

12.2. Informative References

[BG04] Brown, D. and R. Galant, "The Static Diffie-Hellman Problem", Cryptology ePrint Archive, Paper 2004/306, 2004, <https://eprint.iacr.org/2004/306>.

[Boyen09] Boyen, X., "HPAKE: Password Authentication Secure against Cross-Site User Impersonation", Cryptology and Network Security (CANS 2009), Lecture Notes in Computer Science, vol. 5888, pp. 279-298, DOI 10.1007/978-3-642-10433-6_19, 2009, <https://doi.org/10.1007/978-3-642-10433-6_19>.

[Canetti01] Canetti, R., "Universally composable security: A new paradigm for cryptographic protocols", 42nd IEEE Symposium on Foundations of Computer Science, pp. 136-145, DOI 10.1109/SFCS.2001.959888, 2001, <https://doi.org/10.1109/SFCS.2001.959888>.

[Cheon06] Cheon, J. H., "Security Analysis of the Strong Diffie-Hellman Problem", Advances in Cryptology - EUROCRYPT 2006, Lecture Notes in Computer Science, vol. 4004, pp. 1-11, DOI 10.1007/11761679_1, 2006, <https://doi.org/10.1007/11761679_1>.

[DL24] Dayanikli, D. and A. Lehmann, "(Strong) aPAKE Revisited: Capturing Multi-User Security and Salting", Cryptology ePrint Archive, Paper 2024/756, 2024, <https://eprint.iacr.org/2024/756>.

[FIPS202] NIST, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", NIST FIPS 202, DOI 10.6028/NIST.FIPS.202, August 2015, <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf>.

[FK00] Ford, W. and B. S. Kaliski, Jr, "Server-assisted generation of a strong secret from a password", IEEE 9th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE 2000), pp. 176-180, DOI 10.1109/ENABL.2000.883724, 2000, <https://doi.org/10.1109/ENABL.2000.883724>.

[HJKW23] Hesse, J., Jarecki, S., Krawczyk, H., and C. Wood, "Password-Authenticated TLS via OPAQUE and Post-Handshake Authentication", Advances in Cryptology – EUROCRYPT 2023, Lecture Notes in Computer Science, vol. 14008, pp. 98-127, DOI 10.1007/978-3-031-30589-4_4, 2023, <https://doi.org/10.1007/978-3-031-30589-4_4>.

[HMQV] Krawczyk, H., "HMQV: A High-Performance Secure Diffie-Hellman Protocol", Cryptology ePrint Archive, Paper 2005/176, 2005, <https://eprint.iacr.org/2005/176>.

[JKKX16] Jarecki, S., Kiayias, A., Krawczyk, H., and J. Xu, "Highly-Efficient and Composable Password-Protected Secret Sharing (Or: How to Protect Your Bitcoin Wallet Online)", 2016 IEEE European Symposium on Security and Privacy (EuroS&P), pp. 276-291, DOI 10.1109/EuroSP.2016.30, 2016, <https://doi.org/10.1109/EuroSP.2016.30>.

[JKX18Full] Jarecki, S., Krawczyk, H., and J. Xu, "OPAQUE: An Asymmetric PAKE Protocol Secure Against Pre-Computation Attacks", Cryptology ePrint Archive, Paper 2018/163, 2018, <https://eprint.iacr.org/2018/163>.

[keyagreement] Barker, E., Chen, L., Roginsky, A., Vassilev, A., and R. Davis, "Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography", DOI 10.6028/nist.sp.800-56ar3, NIST SP 800-56Ar3, April 2018, <https://doi.org/10.6028/nist.sp.800-56ar3>.

[Krawczyk20] Krawczyk, H., "The OPAQUE Asymmetric PAKE Protocol", Work in Progress, Internet-Draft, draft-krawczyk-cfrg-opaque-06, 19 June 2020, <https://datatracker.ietf.org/doc/html/draft-krawczyk-cfrg-opaque-06>.

[LGR20] Len, J., Grubbs, P., and T. Ristenpart, "Partitioning Oracle Attacks", Cryptology ePrint Archive, Paper 2020/1491, 2021, <https://eprint.iacr.org/2020/1491.pdf>.

[NISTCurves] NIST, "Digital Signature Standard (DSS)", NIST FIPS 186-5, DOI 10.6028/nist.fips.186-5, 2013, <https://doi.org/10.6028/nist.fips.186-5>.

[RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/info/rfc5869>.

[RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <https://www.rfc-editor.org/info/rfc7748>.

[RFC7914] Percival, C. and S. Josefsson, "The scrypt Password-Based Key Derivation Function", RFC 7914, DOI 10.17487/RFC7914, August 2016, <https://www.rfc-editor.org/info/rfc7914>.

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <https://www.rfc-editor.org/info/rfc8017>.

[RFC8018] Moriarty, K., Ed., Kaliski, B., and A. Rusch, "PKCS #5: Password-Based Cryptography Specification Version 2.1", RFC 8018, DOI 10.17487/RFC8018, January 2017, <https://www.rfc-editor.org/info/rfc8018>.

[RFC8125] Schmidt, J., "Requirements for Password-Authenticated Key Agreement (PAKE) Schemes", RFC 8125, DOI 10.17487/RFC8125, April 2017, <https://www.rfc-editor.org/info/rfc8125>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

[RFC9106] Biryukov, A., Dinu, D., Khovratovich, D., and S. Josefsson, "Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications", RFC 9106, DOI 10.17487/RFC9106, September 2021, <https://www.rfc-editor.org/info/rfc9106>.

[RFC9180] Barnes, R., Bhargavan, K., Lipp, B., and C. Wood, "Hybrid Public Key Encryption", RFC 9180, DOI 10.17487/RFC9180, February 2022, <https://www.rfc-editor.org/info/rfc9180>.

[RFC9380] Faz-Hernandez, A., Scott, S., Sullivan, N., Wahby, R. S., and C. A. Wood, "Hashing to Elliptic Curves", RFC 9380, DOI 10.17487/RFC9380, August 2023, <https://www.rfc-editor.org/info/rfc9380>.

[RFC9496] de Valence, H., Grigg, J., Hamburg, M., Lovecruft, I., Tankersley, G., and F. Valsorda, "The ristretto255 and decaf448 Groups", RFC 9496, DOI 10.17487/RFC9496, December 2023, <https://www.rfc-editor.org/info/rfc9496>.

[SIGMA-I] Krawczyk, H., "SIGMA: The 'SIGn-and-MAc' Approach to Authenticated Diffie-Hellman and its Use in the IKE Protocols", 2003, <https://www.iacr.org/cryptodb/archive/2003/CRYPTO/1495/1495.pdf>.

[TOPPSS] Jarecki, S., Kiayias, A., Krawczyk, H., and J. Xu, "TOPPSS: Cost-Minimal Password-Protected Secret Sharing based on Threshold OPRF", Applied Cryptology and Network Security - ACNS 2017, Lecture Notes in Computer Science, vol. 10355, pp. 39-58, DOI 10.1007/978-3-319-61204-1_3, 2017, <https://doi.org/10.1007/978-3-319-61204-1_3>.

[TripleDH] Marlinspike, M., "Simplifying OTR deniability", Signal Blog, 27 July 2013, <https://signal.org/blog/simplifying-otr-deniability>.

[WhatsAppE2E] WhatsApp, "Security of End-to-End Encrypted Backups", WhatsApp Security Whitepaper, 10 September 2021, <https://www.whatsapp.com/security/WhatsApp_Security_Encrypted_Backups_Whitepaper.pdf>.

Appendix A. Alternate Key Recovery Mechanisms

Client authentication material can be stored and retrieved using different key recovery mechanisms. Any key recovery mechanism that encrypts data in the envelope **MUST** use an authenticated encryption scheme with random key-robustness (or key-committing). Deviating from the key-robustness requirement may open the protocol to attacks, e.g., [LGR20]. This specification enforces this property by using a MAC over the envelope contents.

We remark that `export_key` for authentication or encryption requires no special properties from the authentication or encryption schemes as long as `export_key` is used only after authentication material is successfully recovered, i.e., after the MAC in `RecoverCredentials` passes verification.

Appendix B. Alternate AKE Instantiations

It is possible to instantiate OPAQUE with other AKEs, such as HMQV [HMQV] and SIGMA-I [SIGMA-I]. HMQV is similar to 3DH but varies in its key schedule. SIGMA-I uses digital signatures rather than static DH keys for authentication. Specification of these instantiations is left to future documents. A sketch of how these instantiations might change is included in the next subsection for posterity.

OPAQUE may also be instantiated with any post-quantum (PQ) AKE protocol that has the message flow above and security properties (KCI resistance and forward secrecy) outlined in Section 10. Note that such an instantiation is not quantum-safe unless the OPRF is quantum-safe. However, an OPAQUE instantiation where the AKE protocol is quantum-safe, but the OPRF is not, would still ensure the confidentiality and integrity of application data encrypted under `session_key` (or a key derived from it) with a quantum-safe encryption function. However, the only effect of a break of the OPRF by a future quantum attacker would be the ability of this attacker to run at that time an exhaustive dictionary attack against the old user's password and only for users whose envelopes were harvested while in use (in the case of OPAQUE run over a TLS channel with the server, harvesting such envelopes requires targeted active attacks).

B.1. HMQV Instantiation Sketch

An HMQV instantiation would work similarly to OPAQUE-3DH, differing primarily in the key schedule [HMQV]. First, the key schedule `preamble` value would use a different constant prefix -- "HMQV" instead of "3DH" -- as shown below.

```
preamble = concat("HMQV",
                  I2OSP(len(client_identity), 2), client_identity,
                  KE1,
                  I2OSP(len(server_identity), 2), server_identity,
                  KE2.credential_response,
                  KE2.auth_response.server_nonce,
                  KE2.auth_response.server_public_keyshare)
```

Second, the `IKM` derivation would change. Assuming HMQV is instantiated with a cyclic group of prime order p with bit length L, clients would compute `IKM` as follows:

```
u' = (eskU + u \* skU) mod p
IKM = (epkS \* pkS^s)^u'
```

Likewise, servers would compute `IKM` as follows:

```
s' = (eskS + s \* skS) mod p
IKM = (epkU \* pkU^u)^s'
```

In both cases, `u` would be computed as follows:

```
hashInput = concat(I2OSP(len(epkU), 2), epkU,
                   I2OSP(len(info), 2), info,
                   I2OSP(len("client"), 2), "client")
u = Hash(hashInput) mod L
```

Likewise, `s` would be computed as follows:

```
hashInput = concat(I2OSP(len(epkS), 2), epkS,
                   I2OSP(len(info), 2), info,
                   I2OSP(len("server"), 2), "server")
s = Hash(hashInput) mod L
```

Hash is the same hash function used in the main OPAQUE protocol for key derivation. Its output length (in bits) must be at least L.

Both parties should perform validation (as in Section 10.7) on each other's public keys before computing the above parameters.

B.2. SIGMA-I Instantiation Sketch

A SIGMA-I [SIGMA-I] instantiation differs more drastically from OPAQUE-3DH since authentication uses digital signatures instead of Diffie-Hellman. In particular, both `KE2` and `KE3` would carry a digital signature, computed using the server and client private keys established during registration, respectively, as well as a MAC, where the MAC is computed as in OPAQUE-3DH but it also covers the identity of the sender.

The key schedule would also change. Specifically, the key schedule `preamble` value would use a different constant prefix -- "SIGMA-I" instead of "3DH" -- and the `IKM` computation would use only the ephemeral public keys exchanged between client and server.

Appendix C. Test Vectors

This section contains real and fake test vectors for the OPAQUE-3DH specification. Each real test vector in Appendix C.1 specifies the configuration information, protocol inputs, intermediate values computed during registration and authentication, and protocol outputs.

Similarly, each fake test vector in Appendix C.2 specifies the configuration information, protocol inputs, and protocol outputs computed during the authentication of an unknown or unregistered user. Note that `masking_key`, `client_private_key`, and `client_public_key` are used as additional inputs as described in Section 6.3.2.2. `client_public_key` is used as the fake `record`'s public key, and `masking_key` is used for the fake `record`'s masking key parameter.

All values are encoded in hexadecimal strings. The configuration information includes the (OPRF, Hash, KSF, KDF, MAC, Group, Context) tuple, where the Group matches that which is used in the OPRF. The KSF used for each test vector is the identity function (denoted Identity), which returns as output the input message supplied to the function without any modification, i.e., `msg` = Stretch(msg).

C.1. Real Test Vectors

C.1.1. OPAQUE-3DH Real Test Vector 1

C.1.1.1. Configuration

```
OPRF: ristretto255-SHA512
Hash: SHA512
KSF: Identity
KDF: HKDF-SHA512
MAC: HMAC-SHA512
Group: ristretto255
Context: 4f50415155452d504f43
Nh: 64
Npk: 32
Nsk: 32
Nm: 64
Nx: 64
Nok: 32
```

C.1.1.2. Input Values

```
oprf_seed: f433d0227b0b9dd54f7c4422b600e764e47fb503f1f9a0f0a47c6606b0
54a7fdc65347f1a08f277e22358bbabe26f823fca82c7848e9a75661f4ec5d5c1989e
f
credential_identifier: 31323334
password: 436f7272656374486f7273654261747465727953746 1706c65
envelope_nonce: ac13171b2f17bc2c74997f0fce1e1f35bec6b91fe2e12dbd323d2
3ba7a38dfec
masking_nonce: 38fe59af0df2c79f57b8780278f5ae47355fe1f817119041951c80
f612fdfc6d
server_private_key: 47451a85372f8b3537e249d7b54188091fb18edde78094b43
e2ba42b5eb89f0d
server_public_key: b2fe7af9f48cc502d016729d2fe25cdd433f2c4bc904660b2a
382c9b79df1a78
server_nonce: 71cd9960ecef2fe0d0f7494986fa3d8b2bb01963537e60efb13981e
138e3d4a1
client_nonce: da7e07376d6d6f034cfa9bb537d11b8c6b4238c334333d1f0aebb38
0cae6a6cc
client_keyshare_seed: 82850a697b42a505f5b68fcdafce8c31f0af2b581f063cf
1091933541936304b
server_keyshare_seed: 05a4f54206eef1ba2f615bc0aa285cb22f26d1153b5b40a
1e85ff80da12f982f
blind_registration: 76cfbfe758db884bebb33582331ba9f159720ca8784a2a070
a265d9c2d6abe01
blind_login: 6ecc102d2e7a7cf49617aad7bbe188556792d4acd60a1a8a8d2b65d4
b0790308
```

C.1.1.3. Intermediate Values

```
client_public_key: 76a845464c68a5d2f7e442436bb1424953b17d3e2e289ccbac
cafb57ac5c3675
auth_key: 6cd32316f18d72a9a927a83199fa030663a38ce0c11fbaef82aa9003773
0494fc555c4d49506284516edd1628c27965b7555a4ebfed2223199f6c67966dde822
randomized_password: aac48c25ab036e30750839d31d6e73007344cb1155289fb7
d329beb932e9adeea73d5d5c22a0ce1952f8aba6d66007615cd1698d4ac85ef1fcf15
0031d1435d9
envelope: ac13171b2f17bc2c74997f0fce1e1f35bec6b91fe2e12dbd323d23ba7a3
8dfec634b0f5b96109c198a8027da51854c35bee90d1e1c781806d07d49b76de6a28b
8d9e9b6c93b9f8b64d16dddd9c5bfb5fea48ee8fd2f75012a8b308605cdd8ba5
handshake_secret: 81263cb85a0cfa12450f0f388de4e92291ec4c7c7a0878b6245
50ff528726332f1298fc6cc822a432c89504347c7a2ccd70316ae3da6a15e0399e6db
3f7c1b12
server_mac_key: 0d36b26cfe38f51f804f0a9361818f32ee1ce2a4e5578653b5271
84af058d3b2d8075c296fd84d24677913d1baa109290cd81a13ed383f9091a3804e65
298dfc
client_mac_key: 91750adbac54a5e8e53b4c233cc8d369fe83b0de1b6a3cd85575e
eb0bb01a6a90a086a2cf5fe75fff2a9379c30ba9049510a33b5b0b1444a88800fc3ee
e2260d
oprf_key: 5d4c6a8b7c7138182afb4345d1fae6a9f18a1744afbcc3854f8f5a2b4b4
c6d05
```

C.1.1.4. Output Values

```
registration_request: 5059ff249eb1551b7ce4991f3336205bde44a105a032e74
7d21bf382e75f7a71
registration_response: 7408a268083e03abc7097fc05b587834539065e86fb0c7
b6342fcf5e01e5b019b2fe7af9f48cc502d016729d2fe25cdd433f2c4bc904660b2a3
82c9b79df1a78
registration_upload: 76a845464c68a5d2f7e442436bb1424953b17d3e2e289ccb
accafb57ac5c36751ac5844383c7708077dea41cbefe2fa15724f449e535dd7dd562e
66f5ecfb95864eadddec9db5874959905117dad40a4524111849799281fefe3c51fa8
2785c5ac13171b2f17bc2c74997f0fce1e1f35bec6b91fe2e12dbd323d23ba7a38dfe
c634b0f5b96109c198a8027da51854c35bee90d1e1c781806d07d49b76de6a28b8d9e
9b6c93b9f8b64d16dddd9c5bfb5fea48ee8fd2f75012a8b308605cdd8ba5
KE1: c4dedb0ba6ed5d965d6f250fbe554cd45cba5dfcce3ce836e4aee778aa3cd44d
da7e07376d6d6f034cfa9bb537d11b8c6b4238c334333d1f0aebb380cae6a6cc6e29b
ee50701498605b2c085d7b241ca15ba5c32027dd21ba420b94ce60da326
KE2: 7e308140890bcde30cbcea28b01ea1ecfbd077cff62c4def8efa075aabcbb471
38fe59af0df2c79f57b8780278f5ae47355fe1f817119041951c80f612fdfc6dd6ec6
0bcdb26dc455ddf3e718f1020490c192d70dfc7e403981179d8073d1146a4f9aa1ced
4e4cd984c657eb3b54ced3848326f70331953d91b02535af44d9fedc80188ca46743c
52786e0382f95ad85c08f6afcd1ccfbff95e2bdeb015b166c6b20b92f832cc6df01e0
b86a7efd92c1c804ff865781fa93f2f20b446c8371b671cd9960ecef2fe0d0f749498
6fa3d8b2bb01963537e60efb13981e138e3d4a1c4f62198a9d6fa9170c42c3c71f197
1b29eb1d5d0bd733e40816c91f7912cc4a660c48dae03e57aaa38f3d0cffcfc21852e
bc8b405d15bd6744945ba1a93438a162b6111699d98a16bb55b7bdddfe0fc5608b23d
a246e7bd73b47369169c5c90
KE3: 4455df4f810ac31a6748835888564b536e6da5d9944dfea9e34defb9575fe5e2
661ef61d2ae3929bcf57e53d464113d364365eb7d1a57b629707ca48da18e442
export_key: 1ef15b4fa99e8a852412450ab78713aad30d21fa6966c9b8c9fb3262a
970dc62950d4dd4ed62598229b1b72794fc0335199d9f7fcc6eaedde92cc04870e63f
16
session_key: 42afde6f5aca0cfa5c163763fbad55e73a41db6b41bc87b8e7b62214
a8eedc6731fa3cb857d657ab9b3764b89a84e91ebcb4785166fbb02cedfcbdfda215b
96f
```

C.1.2. OPAQUE-3DH Real Test Vector 2

C.1.2.1. Configuration

```
OPRF: ristretto255-SHA512
Hash: SHA512
KSF: Identity
KDF: HKDF-SHA512
MAC: HMAC-SHA512
Group: ristretto255
Context: 4f50415155452d504f43
Nh: 64
Npk: 32
Nsk: 32
Nm: 64
Nx: 64
Nok: 32
```

C.1.2.2. Input Values

```
client_identity: 616c696365
server_identity: 626f62
oprf_seed: f433d0227b0b9dd54f7c4422b600e764e47fb503f1f9a0f0a47c6606b0
54a7fdc65347f1a08f277e22358bbabe26f823fca82c7848e9a75661f4ec5d5c1989e
f
credential_identifier: 31323334
password: 436f7272656374486f72736542617474657279537461706c65
envelope_nonce: ac13171b2f17bc2c74997f0fce1e1f35bec6b91fe2e12dbd323d2
3ba7a38dfec
masking_nonce: 38fe59af0df2c79f57b8780278f5ae47355fe1f817119041951c80
f612fdfc6d
server_private_key: 47451a85372f8b3537e249d7b54188091fb18edde78094b43
e2ba42b5eb89f0d
server_public_key: b2fe7af9f48cc502d016729d2fe25cdd433f2c4bc904660b2a
382c9b79df1a78
server_nonce: 71cd9960ecef2fe0d0f7494986fa3d8b2bb01963537e60efb13981e
138e3d4a1
client_nonce: da7e07376d6d6f034cfa9bb537d11b8c6b4238c334333d1f0aebb38
0cae6a6cc
client_keyshare_seed: 82850a697b42a505f5b68fcdafce8c31f0af2b581f063cf
1091933541936304b
server_keyshare_seed: 05a4f54206eef1ba2f615bc0aa285cb22f26d1153b5b40a
1e85ff80da12f982f
blind_registration: 76cfbfe758db884bebb33582331ba9f159720ca8784a2a070
a265d9c2d6abe01
blind_login: 6ecc102d2e7a7cf49617aad7bbe188556792d4acd60a1a8a8d2b65d4
b0790308
```

C.1.2.3. Intermediate Values

```
client_public_key: 76a845464c68a5d2f7e442436bb1424953b17d3e2e289ccbac
cafb57ac5c3675
auth_key: 6cd32316f18d72a9a927a83199fa030663a38ce0c11fbaef82aa9003773
0494fc555c4d49506284516edd1628c27965b7555a4ebfed2223199f6c67966dde822
randomized_password: aac48c25ab036e30750839d31d6e73007344cb1155289fb7
d329beb932e9adeea73d5d5c22a0ce1952f8aba6d66007615cd1698d4ac85ef1fcf15
0031d1435d9
envelope: ac13171b2f17bc2c74997f0fce1e1f35bec6b91fe2e12dbd323d23ba7a3
8dfec1ac902dc5589e9a5f0de56ad685ea8486210ef41449cd4d8712828913c5d2b68
0b2b3af4a26c765cff329bfb66d38ecf1d6cfa9e7a73c222c6efe0d9520f7d7c
handshake_secret: 5e723bed1e5276de2503419eba9da61ead573109c4012268323
98c7e08155b885bfe7bc93451f9d887a0c1d0c19233e40a8e47b347a9ac3907f94032
a4cff64f
server_mac_key: dad66bb9251073d17a13f8e5500f36e5998e3cde520ca0738e708
5af62fd97812eb79a745c94d0bf8a6ac17f980cf435504cf64041eeb6bb237796d2c7
f81e9a
client_mac_key: f816fe2914f7c5b29852385615d7c7f31ac122adf202d7ccd4976
06d7aabd48930323d1d02b1cc9ecd456c4de6f46c7950becb18bffd921dd5876381b5
486ffe
oprf_key: 5d4c6a8b7c7138182afb4345d1fae6a9f18a1744afbcc3854f8f5a2b4b4
c6d05
```

C.1.2.4. Output Values

```
registration_request: 5059ff249eb1551b7ce4991f3336205bde44a105a032e74
7d21bf382e75f7a71
registration_response: 7408a268083e03abc7097fc05b587834539065e86fb0c7
b6342fcf5e01e5b019b2fe7af9f48cc502d016729d2fe25cdd433f2c4bc904660b2a3
82c9b79df1a78
registration_upload: 76a845464c68a5d2f7e442436bb1424953b17d3e2e289ccb
accafb57ac5c36751ac5844383c7708077dea41cbefe2fa15724f449e535dd7dd562e
66f5ecfb95864eadddec9db5874959905117dad40a4524111849799281fefe3c51fa8
2785c5ac13171b2f17bc2c74997f0fce1e1f35bec6b91fe2e12dbd323d23ba7a38dfe
c1ac902dc5589e9a5f0de56ad685ea8486210ef41449cd4d8712828913c5d2b680b2b
3af4a26c765cff329bfb66d38ecf1d6cfa9e7a73c222c6efe0d9520f7d7c
KE1: c4dedb0ba6ed5d965d6f250fbe554cd45cba5dfcce3ce836e4aee778aa3cd44d
da7e07376d6d6f034cfa9bb537d11b8c6b4238c334333d1f0aebb380cae6a6cc6e29b
ee50701498605b2c085d7b241ca15ba5c32027dd21ba420b94ce60da326
KE2: 7e308140890bcde30cbcea28b01ea1ecfbd077cff62c4def8efa075aabcbb471
38fe59af0df2c79f57b8780278f5ae47355fe1f817119041951c80f612fdfc6dd6ec6
0bcdb26dc455ddf3e718f1020490c192d70dfc7e403981179d8073d1146a4f9aa1ced
4e4cd984c657eb3b54ced3848326f70331953d91b02535af44d9fea502150b67fe367
95dd8914f164e49f81c7688a38928372134b7dccd50e09f8fed9518b7b2f94835b3c4
fe4c8475e7513f20eb97ff0568a39caee3fd6251876f71cd9960ecef2fe0d0f749498
6fa3d8b2bb01963537e60efb13981e138e3d4a1c4f62198a9d6fa9170c42c3c71f197
1b29eb1d5d0bd733e40816c91f7912cc4a292371e7809a9031743e943fb3b56f51de9
03552fc91fba4e7419029951c3970b2e2f0a9dea218d22e9e4e0000855bb6421aa361
0d6fc0f4033a6517030d4341
KE3: 7a026de1d6126905736c3f6d92463a08d209833eb793e46d0f7f15b3e0f62c76
43763c02bbc6b8d3d15b63250cae98171e9260f1ffa789750f534ac11a0176d5
export_key: 1ef15b4fa99e8a852412450ab78713aad30d21fa6966c9b8c9fb3262a
970dc62950d4dd4ed62598229b1b72794fc0335199d9f7fcc6eaedde92cc04870e63f
16
session_key: ae7951123ab5befc27e62e63f52cf472d6236cb386c968cc47b7e34f
866aa4bc7638356a73cfce92becf39d6a7d32a1861f12130e824241fe6cab34fbd471
a57
```

C.1.3. OPAQUE-3DH Real Test Vector 3

C.1.3.1. Configuration

```
OPRF: ristretto255-SHA512
Hash: SHA512
KSF: Identity
KDF: HKDF-SHA512
MAC: HMAC-SHA512
Group: curve25519
Context: 4f50415155452d504f43
Nh: 64
Npk: 32
Nsk: 32
Nm: 64
Nx: 64
Nok: 32
```

C.1.3.2. Input Values

```
oprf_seed: a78342ab84d3d30f08d5a9630c79bf311c31ed7f85d9d4959bf492ec67
a0eec8a67dfbf4497248eebd49e878aab173e5e4ff76354288fdd53e949a5f7c9f7f1
b
credential_identifier: 31323334
password: 436f7272656374486f72736542617474657279537461706c65
envelope_nonce: 40d6b67fdd7da7c49894750754514dbd2070a407166bd2a5237cc
a9bf44d6e0b
masking_nonce: 38fe59af0df2c79f57b8780278f5ae47355fe1f817119041951c80
f612fdfc6d
server_private_key: c06139381df63bfc91c850db0b9cfbec7a62e86d80040a41a
a7725bf0e79d564
server_public_key: a41e28269b4e97a66468cc00c5a57753e192e1527669897706
88aa90486ef031
server_nonce: 71cd9960ecef2fe0d0f7494986fa3d8b2bb01963537e60efb13981e
138e3d4a1
client_nonce: da7e07376d6d6f034cfa9bb537d11b8c6b4238c334333d1f0aebb38
0cae6a6cc
client_keyshare_seed: 82850a697b42a505f5b68fcdafce8c31f0af2b581f063cf
1091933541936304b
server_keyshare_seed: 05a4f54206eef1ba2f615bc0aa285cb22f26d1153b5b40a
1e85ff80da12f982f
blind_registration: c575731ffe1cb0ca5ba63b42c4699767b8b9ab78ba39316ee
04baddb2034a70a
blind_login: 6ecc102d2e7a7cf49617aad7bbe188556792d4acd60a1a8a8d2b65d4
b0790308
```

C.1.3.3. Intermediate Values

```
client_public_key: 0936ea94ab030ec332e29050d266c520e916731a052d05ced7
e0cfe751142b48
auth_key: 7e880ab484f750e80e6f839d975aff476070ce65066d85ea62523d1d576
4739d91307fac47186a4ab935e6a5c7f70cb47faa9473311947502c022cc67ae9440c
randomized_password: 3a602c295a9c323d9362fe286f104567ed6862b25dbe30fa
da844f19e41cf40047424b7118e15dc2c1a815a70fea5c8de6c30aa61440cd4b4b5e8
f3963fbb2e1
envelope: 40d6b67fdd7da7c49894750754514dbd2070a407166bd2a5237cca9bf44
d6e0b20c1e81fef28e92e897ca8287d49a55075b47c3988ff0fff367d79a3e350ccac
150b4a3ff48b4770c8e84e437b3d4e68d2b95833f7788f7eb93fa6a8afb85ecb
handshake_secret: 178c8c15e025252380c3edb1c6ad8ac52573b38d536099e2f86
5786f5e31c642608550c0c6f281c37ce259667dd72768af31630e0eb36f1096a2e642
1c2aa163
server_mac_key: f3c6a8e069c54bb0d8905139f723c9e22f5c662dc08848243a665
4c8223800019b9823523d84da2ef67ca1c14277630aace464c113be8a0a658c39e181
a8bb71
client_mac_key: b1ee7ce52dbd0ab72872924ff11596cb196bbabfc319e74aca78a
de54a0f74dd15dcf5621f6d2e79161b0c9b701381d494836dedbb86e584a65b34267a
370e01
oprf_key: 62ef7f7d9506a14600c34f642aaf6ef8019cc82a6755db4fded5248ea14
6030a
```

C.1.3.4. Output Values

```
registration_request: 26f3dbfd76b8e5f85b4da604f42889a7d4b1bc919f65538
1a67de02c59fd5436
registration_response: 506e8f1b89c098fb89b5b6210a05f7898cafdaea221761
e8d5272fc39e0f9f08a41e28269b4e97a66468cc00c5a57753e192e15276698977068
8aa90486ef031
registration_upload: 0936ea94ab030ec332e29050d266c520e916731a052d05ce
d7e0cfe751142b486d23c6ed818882f9bdfdcf91389fcbc0b7a3faf92bd0bd6be4a1e
7730277b694fc7c6ba327fbe786af18487688e0f7c148bbd54dc2fc80c28e7a976d9e
f53c3540d6b67fdd7da7c49894750754514dbd2070a407166bd2a5237cca9bf44d6e0
b20c1e81fef28e92e897ca8287d49a55075b47c3988ff0fff367d79a3e350ccac150b
4a3ff48b4770c8e84e437b3d4e68d2b95833f7788f7eb93fa6a8afb85ecb
KE1: c4dedb0ba6ed5d965d6f250fbe554cd45cba5dfcce3ce836e4aee778aa3cd44d
da7e07376d6d6f034cfa9bb537d11b8c6b4238c334333d1f0aebb380cae6a6cc10a83
b9117d3798cb2957fbdb0268a0d63dbf9d66bde5c00c78affd80026c911
KE2: 9a0e5a1514f62e005ea098b0d8cf6750e358c4389e6add1c52aed9500fa19d00
38fe59af0df2c79f57b8780278f5ae47355fe1f817119041951c80f612fdfc6d22cc3
1127d6f0096755be3c3d2dd6287795c317aeea10c9485bf4f419a786642c19a8f151c
eb5e8767d175248c62c017de94057398d28bf0ed00d1b50ee4f812fd9afddf98af8cd
58067ca43b0633b6cadd0e9d987f89623fed4d3583bdf6910c425600e90dab3c6b351
3188a465461a67f6bbc47aeba808f7f7e2c6d66f5c3271cd9960ecef2fe0d0f749498
6fa3d8b2bb01963537e60efb13981e138e3d4a141f55f0bef355cfb34ccd468fdacad
75865ee7efef95f4cb6c25d477f720502676f06a3b806da262139bf3fa76a1090b94d
ac78bc3bc6f8747d5b35acf94eff3ec2ebe7d49b8cf16be64120b279fe92664e47be5
da7e60f08f12e91192652f79
KE3: 550e923829a544496d8316c490da2b979b78c730dd75be3a17f237a26432c19f
bba54b6a0467b1c22ecbd6794bc5fa5b04215ba1ef974c6b090baa42c5bb984f
export_key: 9dec51d6d0f6ce7e4345f10961053713b07310cc2e45872f57bbd2fe5
070fdf0fb5b77c7ddaa2f3dc5c35132df7417ad7fefe0f690ad266e5a54a21d045c9c
38
session_key: fd2fdd07c1bcc88e81c1b1d1de5ad62dfdef1c0b8209ff9d671e1fac
55ce9c34d381c1fb2703ff53a797f77daccbe33047ccc167b8105171e10ec962eea20
3aa
```

C.1.4. OPAQUE-3DH Real Test Vector 4

C.1.4.1. Configuration

```
OPRF: ristretto255-SHA512
Hash: SHA512
KSF: Identity
KDF: HKDF-SHA512
MAC: HMAC-SHA512
Group: curve25519
Context: 4f50415155452d504f43
Nh: 64
Npk: 32
Nsk: 32
Nm: 64
Nx: 64
Nok: 32
```

C.1.4.2. Input Values

```
client_identity: 616c696365
server_identity: 626f62
oprf_seed: a78342ab84d3d30f08d5a9630c79bf311c31ed7f85d9d4959bf492ec67
a0eec8a67dfbf4497248eebd49e878aab173e5e4ff76354288fdd53e949a5f7c9f7f1
b
credential_identifier: 31323334
password: 436f7272656374486f72736542617474657279537461706c65
envelope_nonce: 40d6b67fdd7da7c49894750754514dbd2070a407166bd2a5237cc
a9bf44d6e0b
masking_nonce: 38fe59af0df2c79f57b8780278f5ae47355fe1f817119041951c80
f612fdfc6d
server_private_key: c06139381df63bfc91c850db0b9cfbec7a62e86d80040a41a
a7725bf0e79d564
server_public_key: a41e28269b4e97a66468cc00c5a57753e192e1527669897706
88aa90486ef031
server_nonce: 71cd9960ecef2fe0d0f7494986fa3d8b2bb01963537e60efb13981e
138e3d4a1
client_nonce: da7e07376d6d6f034cfa9bb537d11b8c6b4238c334333d1f0aebb38
0cae6a6cc
client_keyshare_seed: 82850a697b42a505f5b68fcdafce8c31f0af2b581f063cf
1091933541936304b
server_keyshare_seed: 05a4f54206eef1ba2f615bc0aa285cb22f26d1153b5b40a
1e85ff80da12f982f
blind_registration: c575731ffe1cb0ca5ba63b42c4699767b8b9ab78ba39316ee
04baddb2034a70a
blind_login: 6ecc102d2e7a7cf49617aad7bbe188556792d4acd60a1a8a8d2b65d4
b0790308
```

C.1.4.3. Intermediate Values

```
client_public_key: 0936ea94ab030ec332e29050d266c520e916731a052d05ced7
e0cfe751142b48
auth_key: 7e880ab484f750e80e6f839d975aff476070ce65066d85ea62523d1d576
4739d91307fac47186a4ab935e6a5c7f70cb47faa9473311947502c022cc67ae9440c
randomized_password: 3a602c295a9c323d9362fe286f104567ed6862b25dbe30fa
da844f19e41cf40047424b7118e15dc2c1a815a70fea5c8de6c30aa61440cd4b4b5e8
f3963fbb2e1
envelope: 40d6b67fdd7da7c49894750754514dbd2070a407166bd2a5237cca9bf44
d6e0bb4c0eab6143959a650c5f6b32acf162b1fbe95bb36c5c4f99df53865c4d3537d
69061d80522d772cd0efdbe91f817f6bf7259a56e20b4eb9cbe9443702f4b759
handshake_secret: 13e7dc6afa5334b9dfffe26bee3caf744ef4add176caee464cd
eb3d37303b90de35a8bf095df84471ac77d705f12fe232f1571de1d6a001d3e808998
73a142dc
server_mac_key: a58135acfb2bde92d506cf59119729a6404ad94eba294e4b52a63
baf58cfe03f21bcf735222c7f2c27a60bd958be7f6aed50dc03a78f64e7ae4ac1ff07
1b95aa
client_mac_key: 1e1a8ba156aadc4a302f707d2193c9dab477b355f430d450dd407
ce40dc75613f76ec33dec494f8a6bfdcf951eb060dac33e6572c693954fe92e33730c
9ab0a2
oprf_key: 62ef7f7d9506a14600c34f642aaf6ef8019cc82a6755db4fded5248ea14
6030a
```

C.1.4.4. Output Values

```
registration_request: 26f3dbfd76b8e5f85b4da604f42889a7d4b1bc919f65538
1a67de02c59fd5436
registration_response: 506e8f1b89c098fb89b5b6210a05f7898cafdaea221761
e8d5272fc39e0f9f08a41e28269b4e97a66468cc00c5a57753e192e15276698977068
8aa90486ef031
registration_upload: 0936ea94ab030ec332e29050d266c520e916731a052d05ce
d7e0cfe751142b486d23c6ed818882f9bdfdcf91389fcbc0b7a3faf92bd0bd6be4a1e
7730277b694fc7c6ba327fbe786af18487688e0f7c148bbd54dc2fc80c28e7a976d9e
f53c3540d6b67fdd7da7c49894750754514dbd2070a407166bd2a5237cca9bf44d6e0
bb4c0eab6143959a650c5f6b32acf162b1fbe95bb36c5c4f99df53865c4d3537d6906
1d80522d772cd0efdbe91f817f6bf7259a56e20b4eb9cbe9443702f4b759
KE1: c4dedb0ba6ed5d965d6f250fbe554cd45cba5dfcce3ce836e4aee778aa3cd44d
da7e07376d6d6f034cfa9bb537d11b8c6b4238c334333d1f0aebb380cae6a6cc10a83
b9117d3798cb2957fbdb0268a0d63dbf9d66bde5c00c78affd80026c911
KE2: 9a0e5a1514f62e005ea098b0d8cf6750e358c4389e6add1c52aed9500fa19d00
38fe59af0df2c79f57b8780278f5ae47355fe1f817119041951c80f612fdfc6d22cc3
1127d6f0096755be3c3d2dd6287795c317aeea10c9485bf4f419a786642c19a8f151c
eb5e8767d175248c62c017de94057398d28bf0ed00d1b50ee4f812699bff7663be3c5
d59de94d8e7e58817c7da005b39c25d25555c929e1c5cf6c1b82837b1367c839aab56
a422c0d97719426a79a16f9869cf852100597b23b5a071cd9960ecef2fe0d0f749498
6fa3d8b2bb01963537e60efb13981e138e3d4a141f55f0bef355cfb34ccd468fdacad
75865ee7efef95f4cb6c25d477f72050267cc22c87edbf3ecaca64cb33bc60dc3bfc5
51e365f0d46a7fed0e09d96f9afbb48868f5bb3c3e05a86ed8c9476fc22c58306c5a2
91be34388e09548ba9d70f39
KE3: d16344e791c3f18594d22ba068984fa18ec1e9bead662b75f66826ffd627932f
cd1ec40cd01dcf5f63f4055ebe45c7717a57a833aad360256cf1e1c20c0eae1c
export_key: 9dec51d6d0f6ce7e4345f10961053713b07310cc2e45872f57bbd2fe5
070fdf0fb5b77c7ddaa2f3dc5c35132df7417ad7fefe0f690ad266e5a54a21d045c9c
38
session_key: f6116d3aa0e4089a179713bad4d98ed5cb57e5443cae8d36ef78996f
a60f3dc6e9fcdd63c001596b06dbc1285d80211035cc0e485506b3f7a650cbf78c5bf
fc9
```

C.1.5. OPAQUE-3DH Real Test Vector 5

C.1.5.1. Configuration

```
OPRF: P256-SHA256
Hash: SHA256
KSF: Identity
KDF: HKDF-SHA256
MAC: HMAC-SHA256
Group: P256_XMD:SHA-256_SSWU_RO_
Context: 4f50415155452d504f43
Nh: 32
Npk: 33
Nsk: 32
Nm: 32
Nx: 32
Nok: 32
```

C.1.5.2. Input Values

```
oprf_seed: 62f60b286d20ce4fd1d64809b0021dad6ed5d52a2c8cf27ae6582543a0
a8dce2
credential_identifier: 31323334
password: 436f7272656374486f72736542617474657279537461706c65
envelope_nonce: a921f2a014513bd8a90e477a629794e89fec12d12206dde662ebd
cf65670e51f
masking_nonce: 38fe59af0df2c79f57b8780278f5ae47355fe1f817119041951c80
f612fdfc6d
server_private_key: c36139381df63bfc91c850db0b9cfbec7a62e86d80040a41a
a7725bf0e79d5e5
server_public_key: 035f40ff9cf88aa1f5cd4fe5fd3da9ea65a4923a5594f84fd9
f2092d6067784874
server_nonce: 71cd9960ecef2fe0d0f7494986fa3d8b2bb01963537e60efb13981e
138e3d4a1
client_nonce: ab3d33bde0e93eda72392346a7a73051110674bbf6b1b7ffab8be4f
91fdaeeb1
client_keyshare_seed: 633b875d74d1556d2a2789309972b06db21dfcc4f5ad51d
7e74d783b7cfab8dc
server_keyshare_seed: 05a4f54206eef1ba2f615bc0aa285cb22f26d1153b5b40a
1e85ff80da12f982f
blind_registration: 411bf1a62d119afe30df682b91a0a33d777972d4f2daa4b34
ca527d597078153
blind_login: c497fddf6056d241e6cf9fb7ac37c384f49b357a221eb0a802c989b9
942256c1
```

C.1.5.3. Intermediate Values

```
client_public_key: 03b218507d978c3db570ca994aaf36695a731ddb2db272c817
f79746fc37ae5214
auth_key: 5bd4be1602516092dc5078f8d699f5721dc1720a49fb80d8e5c16377abd
0987b
randomized_password: 06be0a1a51d56557a3adad57ba29c5510565dcd8b5078fa3
19151b9382258fb0
envelope: a921f2a014513bd8a90e477a629794e89fec12d12206dde662ebdcf6567
0e51fad30bbcfc1f8eda0211553ab9aaf26345ad59a128e80188f035fe4924fad67b8
handshake_secret: 83a932431a8f25bad042f008efa2b07c6cd0faa8285f335b636
3546a9f9b235f
server_mac_key: 13e928581febfad28855e3e7f03306d61bd69489686f621535d44
a1365b73b0d
client_mac_key: afdc53910c25183b08b930e6953c35b3466276736d9de2e9c5efa
f150f4082c5
oprf_key: 2dfb5cb9aa1476093be74ca0d43e5b02862a05f5d6972614d7433acdc66
f7f31
```

C.1.5.4. Output Values

```
registration_request: 029e949a29cfa0bf7c1287333d2fb3dc586c41aa652f507
0d26a5315a1b50229f8
registration_response: 0350d3694c00978f00a5ce7cd08a00547e4ab5fb5fc2b2
f6717cdaa6c89136efef035f40ff9cf88aa1f5cd4fe5fd3da9ea65a4923a5594f84fd
9f2092d6067784874
registration_upload: 03b218507d978c3db570ca994aaf36695a731ddb2db272c8
17f79746fc37ae52147f0ed53532d3ae8e505ecc70d42d2b814b6b0e48156def71ea0
29148b2803aafa921f2a014513bd8a90e477a629794e89fec12d12206dde662ebdcf6
5670e51fad30bbcfc1f8eda0211553ab9aaf26345ad59a128e80188f035fe4924fad6
7b8
KE1: 037342f0bcb3ecea754c1e67576c86aa90c1de3875f390ad599a26686cdfee6e
07ab3d33bde0e93eda72392346a7a73051110674bbf6b1b7ffab8be4f91fdaeeb1022
ed3f32f318f81bab80da321fecab3cd9b6eea11a95666dfa6beeaab321280b6
KE2: 0246da9fe4d41d5ba69faa6c509a1d5bafd49a48615a47a8dd4b0823cc147648
1138fe59af0df2c79f57b8780278f5ae47355fe1f817119041951c80f612fdfc6d2f0
c547f70deaeca54d878c14c1aa5e1ab405dec833777132eea905c2fbb12504a67dcbe
0e66740c76b62c13b04a38a77926e19072953319ec65e41f9bfd2ae26837b6ce688bf
9af2542f04eec9ab96a1b9328812dc2f5c89182ed47fead61f09f71cd9960ecef2fe0
d0f7494986fa3d8b2bb01963537e60efb13981e138e3d4a103c1701353219b53acf33
7bf6456a83cefed8f563f1040b65afbf3b65d3bc9a19b50a73b145bc87a157e8c58c0
342e2047ee22ae37b63db17e0a82a30fcc4ecf7b
KE3: e97cab4433aa39d598e76f13e768bba61c682947bdcf9936035e8a3a3ebfb66e
export_key: c3c9a1b0e33ac84dd83d0b7e8af6794e17e7a3caadff289fbd9dc769a
853c64b
session_key: 484ad345715ccce138ca49e4ea362c6183f0949aaaa1125dc3bc3f80
876e7cd1
```

C.1.6. OPAQUE-3DH Real Test Vector 6

C.1.6.1. Configuration

```
OPRF: P256-SHA256
Hash: SHA256
KSF: Identity
KDF: HKDF-SHA256
MAC: HMAC-SHA256
Group: P256_XMD:SHA-256_SSWU_RO_
Context: 4f50415155452d504f43
Nh: 32
Npk: 33
Nsk: 32
Nm: 32
Nx: 32
Nok: 32
```

C.1.6.2. Input Values

```
client_identity: 616c696365
server_identity: 626f62
oprf_seed: 62f60b286d20ce4fd1d64809b0021dad6ed5d52a2c8cf27ae6582543a0
a8dce2
credential_identifier: 31323334
password: 436f7272656374486f72736542617474657279537461706c65
envelope_nonce: a921f2a014513bd8a90e477a629794e89fec12d12206dde662ebd
cf65670e51f
masking_nonce: 38fe59af0df2c79f57b8780278f5ae47355fe1f817119041951c80
f612fdfc6d
server_private_key: c36139381df63bfc91c850db0b9cfbec7a62e86d80040a41a
a7725bf0e79d5e5
server_public_key: 035f40ff9cf88aa1f5cd4fe5fd3da9ea65a4923a5594f84fd9
f2092d6067784874
server_nonce: 71cd9960ecef2fe0d0f7494986fa3d8b2bb01963537e60efb13981e
138e3d4a1
client_nonce: ab3d33bde0e93eda72392346a7a73051110674bbf6b1b7ffab8be4f
91fdaeeb1
client_keyshare_seed: 633b875d74d1556d2a2789309972b06db21dfcc4f5ad51d
7e74d783b7cfab8dc
server_keyshare_seed: 05a4f54206eef1ba2f615bc0aa285cb22f26d1153b5b40a
1e85ff80da12f982f
blind_registration: 411bf1a62d119afe30df682b91a0a33d777972d4f2daa4b34
ca527d597078153
blind_login: c497fddf6056d241e6cf9fb7ac37c384f49b357a221eb0a802c989b9
942256c1
```

C.1.6.3. Intermediate Values

```
client_public_key: 03b218507d978c3db570ca994aaf36695a731ddb2db272c817
f79746fc37ae5214
auth_key: 5bd4be1602516092dc5078f8d699f5721dc1720a49fb80d8e5c16377abd
0987b
randomized_password: 06be0a1a51d56557a3adad57ba29c5510565dcd8b5078fa3
19151b9382258fb0
envelope: a921f2a014513bd8a90e477a629794e89fec12d12206dde662ebdcf6567
0e51f4d7773a36a208a866301dbb2858e40dc5638017527cf91aef32d3848eebe0971
handshake_secret: 80bdcc498f22de492e90ee8101fcc7c101e158dd49c77f7c283
816ae329ed62f
server_mac_key: 0f82432fbdb5b90daf27a91a3acc42299a9590dba1b77932c2207
b4cb3d4a157
client_mac_key: 7f629eb0b1b69979b07ca1f564b3e92ed22f07569fd1d11725d93
e46731fbe71
oprf_key: 2dfb5cb9aa1476093be74ca0d43e5b02862a05f5d6972614d7433acdc66
f7f31
```

C.1.6.4. Output Values

```
registration_request: 029e949a29cfa0bf7c1287333d2fb3dc586c41aa652f507
0d26a5315a1b50229f8
registration_response: 0350d3694c00978f00a5ce7cd08a00547e4ab5fb5fc2b2
f6717cdaa6c89136efef035f40ff9cf88aa1f5cd4fe5fd3da9ea65a4923a5594f84fd
9f2092d6067784874
registration_upload: 03b218507d978c3db570ca994aaf36695a731ddb2db272c8
17f79746fc37ae52147f0ed53532d3ae8e505ecc70d42d2b814b6b0e48156def71ea0
29148b2803aafa921f2a014513bd8a90e477a629794e89fec12d12206dde662ebdcf6
5670e51f4d7773a36a208a866301dbb2858e40dc5638017527cf91aef32d3848eebe0
971
KE1: 037342f0bcb3ecea754c1e67576c86aa90c1de3875f390ad599a26686cdfee6e
07ab3d33bde0e93eda72392346a7a73051110674bbf6b1b7ffab8be4f91fdaeeb1022
ed3f32f318f81bab80da321fecab3cd9b6eea11a95666dfa6beeaab321280b6
KE2: 0246da9fe4d41d5ba69faa6c509a1d5bafd49a48615a47a8dd4b0823cc147648
1138fe59af0df2c79f57b8780278f5ae47355fe1f817119041951c80f612fdfc6d2f0
c547f70deaeca54d878c14c1aa5e1ab405dec833777132eea905c2fbb12504a67dcbe
0e66740c76b62c13b04a38a77926e19072953319ec65e41f9bfd2ae268d7f10604202
1c80300e4c6f585980cf39fc51a4a6bba41b0729f9b240c729e5671cd9960ecef2fe0
d0f7494986fa3d8b2bb01963537e60efb13981e138e3d4a103c1701353219b53acf33
7bf6456a83cefed8f563f1040b65afbf3b65d3bc9a19b84922c7e5d074838a8f27859
2c53f61fb59f031e85ad480c0c71086b871e1b24
KE3: 46833578cee137775f6be3f01b80748daac5a694101ad0e9e7025480552da56a
export_key: c3c9a1b0e33ac84dd83d0b7e8af6794e17e7a3caadff289fbd9dc769a
853c64b
session_key: 27766fabd8dd88ff37fbd0ef1a491e601d10d9f016c2b28c4bd1b0fb
7511a3c3
```

C.2. Fake Test Vectors

C.2.1. OPAQUE-3DH Fake Test Vector 1

C.2.1.1. Configuration

```
OPRF: ristretto255-SHA512
Hash: SHA512
KSF: Identity
KDF: HKDF-SHA512
MAC: HMAC-SHA512
Group: ristretto255
Context: 4f50415155452d504f43
Nh: 64
Npk: 32
Nsk: 32
Nm: 64
Nx: 64
Nok: 32
```

C.2.1.2. Input Values

```
client_identity: 616c696365
server_identity: 626f62
oprf_seed: 743fc168d1f826ad43738933e5adb23da6fb95f95a1b069f0daa0522d0
a78b617f701fc6aa46d3e7981e70de7765dfcd6b1e13e3369a582eb8dc456b10aa53b
0
credential_identifier: 31323334
masking_nonce: 9c035896a043e70f897d87180c543e7a063b83c1bb728fbd189c61
9e27b6e5a6
client_private_key: 2b98980aa95ab53a0f39f0291903d2fdf04b00c167f081416
9922df873002409
client_public_key: 84f43f9492e19c22d8bdaa4447cc3d4db1cdb5427a9f852c47
07921212c36251
server_private_key: c788585ae8b5ba2942b693b849be0c0426384e41977c18d2e
81fbe30fd7c9f06
server_public_key: 825f832667480f08b0c9069da5083ac4d0e9ee31b49c4e0310
031fea04d52966
server_nonce: 1e10f6eeab2a7a420bf09da9b27a4639645622c46358de9cf7ae813
055ae2d12
client_keyshare_seed: a270dc715dc2b4612bc7864312a05c3e9788ee1bad1f276
d1e15bdeb4c355e94
server_keyshare_seed: 360b0937f47d45f6123a4d8f0d0c0814b6120d840ebb8bc
5b4f6b62df07f78c2
masking_key: 39ebd51f0e39a07a1c2d2431995b0399bca9996c5d10014d6ebab445
3dc10ce5cef38ed3df6e56bfff40c2d8dd4671c2b4cf63c3d54860f31fe40220d690b
b71
KE1: b0a26dcaca2230b8f5e4b1bcab9c84b586140221bb8b2848486874b0be448905
42d4e61ed3f8d64cdd3b9d153343eca15b9b0d5e388232793c6376bd2d9cfd0ab641d
7f20a245a09f1d4dbb6e301661af7f352beb0791d055e48d3645232f77f
```

C.2.1.3. Output Values

```
KE2: 928f79ad8df21963e91411b9f55165ba833dea918f441db967cdc09521d22925
9c035896a043e70f897d87180c543e7a063b83c1bb728fbd189c619e27b6e5a632b5a
b1bff96636144faa4f9f9afaac75dd88ea99cf5175902ae3f3b2195693f165f11929b
a510a5978e64dcdabecbd7ee1e4380ce270e58fea58e6462d92964a1aaef72698bca1
c673baeb04cc2bf7de5f3c2f5553464552d3a0f7698a9ca7f9c5e70c6cb1f706b2f17
5ab9d04bbd13926e816b6811a50b4aafa9799d5ed7971e10f6eeab2a7a420bf09da9b
27a4639645622c46358de9cf7ae813055ae2d1298251c5ba55f6b0b2d58d9ff0c88fe
4176484be62a96db6e2a8c4d431bd1bf27fe6c1d0537603835217d42ebf7b25819827
32e74892fd28211b31ed33863f0beaf75ba6f59474c0aaf9d78a60a9b2f4cd24d7ab5
4131b3c8efa192df6b72db4c
```

C.2.2. OPAQUE-3DH Fake Test Vector 2

C.2.2.1. Configuration

```
OPRF: ristretto255-SHA512
Hash: SHA512
KSF: Identity
KDF: HKDF-SHA512
MAC: HMAC-SHA512
Group: curve25519
Context: 4f50415155452d504f43
Nh: 64
Npk: 32
Nsk: 32
Nm: 64
Nx: 64
Nok: 32
```

C.2.2.2. Input Values

```
client_identity: 616c696365
server_identity: 626f62
oprf_seed: 66e650652a8266b2205f31fdd68adeb739a05b5e650b19e7edc75e734a
1296d6088188ca46c31ae8ccbd42a52ed338c06e53645387a7efbc94b6a0449526155
e
credential_identifier: 31323334
masking_nonce: 9c035896a043e70f897d87180c543e7a063b83c1bb728fbd189c61
9e27b6e5a6
client_private_key: 288bf63470199221847bb035d99f96531adf8badd14cb1571
b48f7a506649660
client_public_key: 3c64a3153854cc9f0c23aab3c1a19106ec8bab4730736d1d00
3880a1d5a59005
server_private_key: 30fbe7e830be1fe8d2187c97414e3826040cbe49b893b6422
9bab5e85a588846
server_public_key: 78b3040047ff26572a7619617601a61b9c81899bee92f00cfc
aa5eed96863555
server_nonce: 1e10f6eeab2a7a420bf09da9b27a4639645622c46358de9cf7ae813
055ae2d12
client_keyshare_seed: a270dc715dc2b4612bc7864312a05c3e9788ee1bad1f276
d1e15bdeb4c355e94
server_keyshare_seed: 360b0937f47d45f6123a4d8f0d0c0814b6120d840ebb8bc
5b4f6b62df07f78c2
masking_key: 79ad2621b0757a447dff7108a8ae20a068ce67872095620f415ea611
c9dcc04972fa359538cd2fd6528775ca775487b2b56db642049b8a90526b975a38484
c6a
KE1: b0a26dcaca2230b8f5e4b1bcab9c84b586140221bb8b2848486874b0be448905
42d4e61ed3f8d64cdd3b9d153343eca15b9b0d5e388232793c6376bd2d9cfd0ac059b
7ba2aec863933ae48816360c7a9022e83d822704f3b0b86c0502a66e574
```

C.2.2.3. Output Values

```
KE2: 6606b6fedbb33f19a81a1feb5149c600fe77252f58acd3080d7504d3dad4922f
9c035896a043e70f897d87180c543e7a063b83c1bb728fbd189c619e27b6e5a67db39
8c0f65d8c298eac430abdae4c80e82b552fb940c00f0cbcea853c0f96c1c15099f3d4
b0e83ecc249613116d605b8d77bb68bdf76994c2bc507e2dcae4176f00afed68ad25c
f3040a0e991acece31ca532117f5c12816997372ff031ad04ebcdce06c501da24e7b4
db95343456e2ed260895ec362694230a1fa20e24a9c71e10f6eeab2a7a420bf09da9b
27a4639645622c46358de9cf7ae813055ae2d122d9055eb8f83e1b497370adad5cc2a
417bf9be436a792def0c7b7ccb92b9e275d7c663104ea4655bd70570d975c05351655
d55fbfb392286edb55600a23b55ce18f8c60e0d1960c960412dd08eabc81ba7ca8ae2
b04aad65462321f51c298010
```

C.2.3. OPAQUE-3DH Fake Test Vector 3

C.2.3.1. Configuration

```
OPRF: P256-SHA256
Hash: SHA256
KSF: Identity
KDF: HKDF-SHA256
MAC: HMAC-SHA256
Group: P256_XMD:SHA-256_SSWU_RO_
Context: 4f50415155452d504f43
Nh: 32
Npk: 33
Nsk: 32
Nm: 32
Nx: 32
Nok: 32
```

C.2.3.2. Input Values

```
client_identity: 616c696365
server_identity: 626f62
oprf_seed: bb1cd59e16ac09bc0cb6d528541695d7eba2239b1613a3db3ade77b362
80f725
credential_identifier: 31323334
masking_nonce: 9c035896a043e70f897d87180c543e7a063b83c1bb728fbd189c61
9e27b6e5a6
client_private_key: d423b87899fc61d014fc8330a4e26190fcfa470a3afe59243
24294af7dbbc1dd
client_public_key: 03b81708eae026a9370616c22e1e8542fe9dbebd36ce8a2661
b708e9628f4a57fc
server_private_key: 34fbe7e830be1fe8d2187c97414e3826040cbe49b893b6422
9bab5e85a5888c7
server_public_key: 0221e034c0e202fe883dcfc96802a7624166fed4cfcab4ae30
cf5f3290d01c88bf
server_nonce: 1e10f6eeab2a7a420bf09da9b27a4639645622c46358de9cf7ae813
055ae2d12
client_keyshare_seed: a270dc715dc2b4612bc7864312a05c3e9788ee1bad1f276
d1e15bdeb4c355e94
server_keyshare_seed: 360b0937f47d45f6123a4d8f0d0c0814b6120d840ebb8bc
5b4f6b62df07f78c2
masking_key: caecc6ccb4cae27cb54d8f3a1af1bac52a3d53107ce08497cdd362b1
992e4e5e
KE1: 0396875da2b4f7749bba411513aea02dc514a48d169d8a9531bd61d3af3fa9ba
ae42d4e61ed3f8d64cdd3b9d153343eca15b9b0d5e388232793c6376bd2d9cfd0a021
47a6583983cc9973b5082db5f5070890cb373d70f7ac1b41ed2305361009784
```

C.2.3.3. Output Values

```
KE2: 0201198dcd13f9792eb75dcfa815f61b049abfe2e3e9456d4bbbceec5f442efd
049c035896a043e70f897d87180c543e7a063b83c1bb728fbd189c619e27b6e5a6fac
da65ce0a97b9085e7af07f61fd3fdd046d257cbf2183ce8766090b8041a8bf28d79dd
4c9031ddc75bb6ddb4c291e639937840e3d39fc0d5a3d6e7723c09f7945df485bcf9a
efe3fe82d149e84049e259bb5b33d6a2ff3b25e4bfb7eff0962821e10f6eeab2a7a42
0bf09da9b27a4639645622c46358de9cf7ae813055ae2d12023f82bbb24e75b8683fd
13b843cd566efae996cd0016cffdcc24ee2bc937d026f80144878749a69565b433c10
40aff67e94f79345de888a877422b9bbe21ec329
```

Acknowledgments

We are indebted to the OPAQUE reviewers during CFRG's aPAKE selection process, particularly Julia Hesse and Bjorn Tackmann. This document has benefited from comments by multiple people. Special thanks to Richard Barnes, Dan Brown, Matt Campagna, Eric Crockett, Paul Grubbs, Fredrik Kuivinen, Stefan Marsiske, Payman Mohassel, Marta Mularczyk, Jason Resch, Greg Rubin, and Nick Sullivan. Hugo Krawczyk wishes to thank his OPAQUE co-authors Stas Jarecki and Jiayu Xu, without whom this work would have not been possible.

Authors' Addresses

Daniel Bourdrez
Email: d@bytema.re

Hugo Krawczyk
AWS
Email: hugokraw@gmail.com

Kevin Lewi
Meta
Email: lewi.kevin.k@gmail.com

Christopher A. Wood
Cloudflare, Inc.
Email: caw@heapingbits.net