Workgroup: Network Working Group
Internet-Draft: draft-irtf-cfrg-opaque-10
Published: 13 March 2023
Intended Status: Informational
Expires: 14 September 2023
Authors: D. Bourdrez H. Krawczyk K. Lewi
Algorand Foundation Novi Research
C. A. Wood
Cloudflare, Inc.

The OPAQUE Asymmetric PAKE Protocol

Abstract

This document describes the OPAQUE protocol, a secure asymmetric password-authenticated key exchange (aPAKE) 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.

Discussion Venues

This note is to be removed before publishing as an RFC.

Source for this draft and an issue tracker can be found at https://github.com/cfrg/draft-irtf-cfrg-opaque.

Status of This Memo

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 to cases where TLS may fail, including PKI attacks, certificate mishandling, termination outside the security perimeter, visibility to TLS-terminating intermediaries, and more.

Asymmetric (or Augmented) 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. Crucially important is 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, a PKI-free secure aPAKE that is secure against pre-computation attacks. 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 [_3DH]. Other instantiations are possible, as discussed in Appendix C, 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, and with future ones 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 draft complies with the requirements for PAKE protocols set forth in [RFC8125].

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): Generate a cryptographically secure pseudorandom byte string of length n bytes.
- 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 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 [RFC8446], Section 3.

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 PRF such that the client learns the PRF output and neither party learns the input of the other. This specification depends on the prime-order OPRF construction specified in [OPRF], draft version -20, using the OPRF mode (0x00) from [OPRF], Section 3.1.

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.
- Finalize(element, blind, evaluated_element): Finalize the OPRF evaluation using input element, random inverter blind, and evaluation output evaluated_element, yielding output oprf_output.

Moreover, the following OPRF server APIs are used:

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

* DeriveKeyPair(seed, info): Derive a private and public key pair deterministically from a seed and info parameter, as described in [OPRF], Section 3.2.

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

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

* DeserializeElement(buf): Attempt to map input byte array buf to an OPRF group element. This function can raise a DeserializeError upon failure; see [OPRF], Section 2.1 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 optional 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 collision-resistant Message Authentication Code (MAC) with the following API and parameters:

* 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, params): Apply a key stretching function with parameters params 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. 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" 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, and chooses a seed (oprf_seed) of Nh bytes for the OPRF. The server can use this single pair of keys with multiple clients and can opt to use multiple seeds (so long as they are kept consistent for each client).

3.2. Offline Registration

Registration is the only stage in OPAQUE that requires a server-authenticated and confidential channel: 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., 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 below:

```
    creds                                   parameters
      |                                         |
      v                                         v
    Client                                    Server
    ------------------------------------------------
                registration request
             ------------------------->
                registration response
             <-------------------------
                      record
             ------------------------->
   ------------------------------------------------
      |                                         |
      v                                         v
  export_key                                 record
```

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 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 below:

```
    creds                                 (parameters, record)
      |                                               |
      v                                               v
    Client                                          Server
    ------------------------------------------------
                   AKE message 1
             -------------------------->
                   AKE message 2
             <--------------------------
                   AKE message 3
             -------------------------->
   ------------------------------------------------
      |                                               |
      v                                               v
(export_key, session_key)                        session_key
```

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), and their corresponding wire formats are specified in Section 6.1.

The rest of this document describes the details 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.

Future variants of OPAQUE may use different key recovery mechanisms. See Section 4.1 for details.

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 e-mail 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 10.4 for information about this identity.

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 pr
  if server_identity == nil
    server_identity = server_public_key
  if client_identity == nil
    client_identity = client_public_key

  Create CleartextCredentials cleartext_credentials with
    (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 DeriveAuthKeyPair() function defined in Section 6.4.1.

4.1.1. Envelope Structure

The key recovery mechanism defines its Envelope as follows:

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

nonce: A unique nonce of length Nn, used to protect this Envelope.

auth_tag: An authentication tag protecting the contents of the envelope, covering the envelope nonce and CleartextCredentials.

4.1.2. Envelope Creation

Clients create an Envelope at registration with the function Store defined below. Note that DeriveAuthKeyPair 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_pwd, 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 purpos
  of defending against client enumeration attacks.
- export_key, an additional client key.

def Store(randomized_pwd, server_public_key, server_identity, client_ide
  envelope_nonce = random(Nn)
  masking_key = Expand(randomized_pwd, "MaskingKey", Nh)
  auth_key = Expand(randomized_pwd, concat(envelope_nonce, "AuthKey"), N
  export_key = Expand(randomized_pwd, concat(envelope_nonce, "ExportKey"
  seed = Expand(randomized_pwd, concat(envelope_nonce, "PrivateKey"), Ns
  (_, client_public_key) = DeriveAuthKeyPair(seed)

  cleartext_creds =
    CreateCleartextCredentials(server_public_key, client_public_key,
                               server_identity, client_identity)
  auth_tag = MAC(auth_key, concat(envelope_nonce, cleartext_creds))

  Create Envelope envelope with (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_pwd, 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 protoco
- export_key, an additional client key.

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

def Recover(randomized_pwd, server_public_key, envelope,
            server_identity, client_identity):
  auth_key = Expand(randomized_pwd, concat(envelope.nonce, "AuthKey"), N
  export_key = Expand(randomized_pwd, concat(envelope.nonce, "ExportKey"
  seed = Expand(randomized_pwd, concat(envelope.nonce, "PrivateKey"), Ns
  (client_private_key, client_public_key) = DeriveAuthKeyPair(seed)

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

5. Offline 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_private_key: The server private key for the AKE protocol.
- *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(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. 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 10.5.

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 to preserve confidentiality of the envelope during login to defend 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 a InvalidInputError error with negligibile 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)
  Create RegistrationRequest request with 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, application can choose a new credential_identifier for this registration record and re-run this function.

```
CreateRegistrationResponse

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

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)

  Create RegistrationResponse response with (evaluated_message, server_p
  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_identi
  evaluated_element = DeserializeElement(response.evaluated_message)
  oprf_output = Finalize(password, blind, evaluated_element)

  stretched_oprf_output = Stretch(oprf_output, params)
  randomized_pwd = Extract("", concat(oprf_output, stretched_oprf_output

  (envelope, client_public_key, masking_key, export_key) =
    Store(randomized_pwd, response.server_public_key,
          server_identity, client_identity)
  Create RegistrationRecord record with (client_public_key, masking_key,
  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, where KE1 and KE2 include key exchange shares, e.g., DH values, sent by the client and server, respectively, and 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 10.5. The output export_key MUST NOT be used in any way before the protocol completes successfully. See Appendix B 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 = ClientInit(password)

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

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

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

   (ke3,
   session_key,
   export_key) = ClientFinish(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 defined in Section 6.4.

The server state ServerState may have the following fields:

* server_ake_state: The ServerAkeState defined in Section 6.4.

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_keyshare[Npk];
} AuthRequest;
```

client_nonce: A fresh randomly generated nonce of length Nn.

client_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_keyshare[Npk];
  uint8 server_mac[Nm];
} AuthResponse;
```

server_nonce: A fresh randomly generated nonce of length Nn.

server_keyshare: 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, 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, RecoverCredentials defined in Section 6.3

* AuthClientStart, AuthServerRespond, AuthClientFinalize, and AuthServerFinalize defined in Section 6.4.3 and Section 6.4.4

6.2.1. ClientInit

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

```
ClientInit

State:
- state, a ClientState structure.

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

Output:
- ke1, a KE1 message structure.

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

6.2.2. ServerInit

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

```
ServerInit

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 cred
- oprf_seed, the server-side seed of Nh bytes used to generate an oprf_k
- 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 ServerInit(server_identity, server_private_key, server_public_key,
               record, credential_identifier, oprf_seed, ke1, client_ide
  credential_response = CreateCredentialResponse(ke1.credential_request,
    credential_identifier, oprf_seed)
  auth_response = AuthServerRespond(server_identity, server_private_key,
    client_identity, record.client_public_key, ke1, credential_response)
  Create KE2 ke2 with (credential_response, auth_response)
  return ke2
```

6.2.3. ClientFinish

The ClientFinish 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.

```
ClientFinish

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 ClientFinish(client_identity, server_identity, ke2):
  (client_private_key, server_public_key, export_key) =
    RecoverCredentials(state.password, state.blind, ke2.credential_respo
                       server_identity, client_identity)
  (ke3, session_key) =
    AuthClientFinalize(client_identity, client_private_key, server_ident
                       server_public_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, and it, therefore, 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, 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 a InvalidInputError error with negligibile 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)
  Create CredentialRequest request with 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 a client that has yet to register).

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 cred
- oprf_seed, the server-side seed of Nh bytes used to generate an oprf_k

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, "CredentialResp
                                   Npk + Nn + Nm)
  masked_response = xor(credential_response_pad,
                        concat(server_public_key, record.envelope))
  Create CredentialResponse response with (evaluated_message, masking_no
  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 stored alongside legitimate client records. This allows servers to locate 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 client's private key for the AKE protocol.
- server_public_key, the public key of the server.
- 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, params)
  randomized_pwd = Extract("", concat(oprf_output, stretched_oprf_output

  masking_key = Expand(randomized_pwd, "MaskingKey", Nh)
  credential_response_pad = Expand(masking_key,
                                  concat(response.masking_nonce, "Crede
                                  Npk + Nn + Nm)
  concat(server_public_key, envelope) = xor(credential_response_pad,
                                             response.masked_response)
  (client_private_key, export_key) =
    Recover(randomized_pwd, server_public_key, envelope,
            server_identity, client_identity)

  return (client_private_key, server_public_key, export_key)
```

6.4. AKE Protocol

This section describes the authenticated key exchange protocol for OPAQUE using 3DH, a 3-message AKE which 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.

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

6.4.1. Key Creation

We assume the following functions to exist for all candidate groups in this setting:

* DeriveAuthKeyPair(seed): Derive a private and public authentication key pair deterministically from the input seed. This function is implemented as DeriveKeyPair(seed, "OPAQUE-DeriveAuthKeyPair"), where DeriveKeyPair is as specified in [OPRF], Section 3.2.

* GenerateAuthKeyPair(): Return a randomly generated private and public key pair. This can be implemented by invoking DeriveAuthKeyPair with Nseed random bytes as input.

* SerializeElement(element): A member function of the underlying group that maps element to a unique byte array, mirrored from the definition of the similarly-named function of the OPRF group described in [OPRF], Section 2.1.

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, re-purposed from TLS 1.3 [RFC8446].

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

Where CustomLabel is specified 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 shared context information in the transcript, such as configuration parameters or application-specific info, 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_keyshare, the corresponding field on the AuthResponse structure

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

def Preamble(client_identity, ke1, server_identity, ke2):
  preamble = concat("RFCXXXX",
                     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_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_secret, client_keyshare) = GenerateAuthKeyPair()
  Create AuthRequest auth_request with (client_nonce, client_keyshare)
  Create KE1 ke1 with (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:
- client_identity, the optional encoded client identity, which is
  set to client_public_key if not specified.
- client_private_key, the client's private key.
- server_identity, the optional encoded server identity, which is
  set to server_public_key if not specified.
- server_public_key, the server's public key.
- ke2, a KE2 message structure.

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

Exceptions:
- ServerAuthenticationError, the handshake fails.

def AuthClientFinalize(client_identity, client_private_key, server_ident
                       server_public_key, ke2):

  dh1 = SerializeElement(state.client_secret * ke2.auth_response.server_
  dh2 = SerializeElement(state.client_secret * server_public_key)
  dh3 = SerializeElement(client_private_key  * ke2.auth_response.server_
  ikm = concat(dh1, dh2, dh3)

  preamble = Preamble(client_identity,
                      state.ke1,
                      server_identity,
                      ke2.credential_response,
                      ke2.auth_response.server_nonce,
                      ke2.auth_response.server_keyshare)
  Km2, Km3, session_key = DeriveKeys(ikm, preamble)
  expected_server_mac = MAC(Km2, Hash(preamble))
  if !ct_equal(ke2.server_mac, expected_server_mac),
    raise ServerAuthenticationError
  client_mac = MAC(Km3, Hash(concat(preamble, expected_server_mac)))
  Create KE3 ke3 with 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:
- 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.
- client_identity, the optional encoded client identity, which is set to
  client_public_key if not specified.
- client_public_key, the client's public key.
- ke1, a KE1 message structure.

Output:
- auth_response, an AuthResponse structure.

def AuthServerRespond(server_identity, server_private_key, client_identi
                      client_public_key, ke1, credential_response):
  server_nonce = random(Nn)
  (server_private_keyshare, server_keyshare) = GenerateAuthKeyPair()
  preamble = Preamble(client_identity,
                      ke1,
                      server_identity,
                      credential_response,
                      server_nonce,
                      server_keyshare)

  dh1 = SerializeElement(server_private_keyshare * ke1.auth_request.clie
  dh2 = SerializeElement(server_private_key * ke1.auth_request.client_ke
  dh3 = SerializeElement(server_private_keyshare * client_public_key)
  ikm = concat(dh1, dh2, dh3)

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

  state.expected_client_mac = MAC(Km3, Hash(concat(preamble, server_mac)
  state.session_key = session_key
  Create AuthResponse auth_response with (server_nonce, server_keyshare,
  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 "base mode" variant of [OPRF] 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 has fixed parameters, chosen by the application, and implements the interface in Section 2. Examples include Argon2id [ARGON2], scrypt [SCRYPT], and PBKDF2 [PBKDF2] with fixed parameter choices.

* 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(t=1, p=4, m=2^21), ristretto255

* P256-SHA256, HKDF-SHA-256, HMAC-SHA-256, SHA-256, Argon2id(t=1, p=4, m=2^21), P-256

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 which 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.

* Enumeration prevention: As described in Section 6.3.2.2, if servers receive a credential request for a non-existent client, they SHOULD respond with a "fake" response to prevent active client enumeration attacks. 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.

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, implementations can incorporate the client identity alongside the password as input to the OPRF. This provides additional client-side entropy which can supplement the entropy that should be introduced by the server during an honest execution of the protocol. This also provides domain separation between different clients that might otherwise share the same password.

Finally, note that 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. Implementations and deployments of OPAQUE SHOULD consider additional checks to mitigate this type of attack: for instance, by ensuring that there is a server-authenticated channel over which OPAQUE registration and login are run.

9.2. 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 KCI security and forward secrecy; see below) into a secure aPAKE protocol. In OPAQUE, the client stores a secret private key at the server 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

[[RFC EDITOR: Please delete this section before publication.]]

The specification as written here differs from the original cryptographic design in [JKX18] and the corresponding CFRG document [I-D.krawczyk-cfrg-opaque-03], 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 [JKX18].

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 [JKX18]. This change adds to the security of the protocol. [JKX18] considered the case where the envelope was constructed by the server for reasons of compatibility with previous UC modeling. An upcoming paper 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 [JKX18] 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 variant is also analyzed in the new paper referred to in the previous item. This change improves the assumption behind the protocol by getting rid of equivocality and random key robustness for the encryption function. The latter property is only required for authentication and achieved by a collision-resistant MAC. 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 authentication 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 but 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.

* 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 independent of other values in the protocol 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 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 password file 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 OPRF has

different PRF inputs, the protocol is secure. The choice of such inputs is up to the application.

The following list enumerates notable differences and refinements from the original cryptographic design in [JKX18] and the corresponding CFRG document [I-D.krawczyk-cfrg-opaque-03] that were made to make this specification suitable for interoperable implementations.

* [JKX18] 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 [I-D.irtf-cfrg-voprf], Section 2.1. This specification also delegates OPRF group choice and operations to [I-D.irtf-cfrg-voprf]. As such, the prime-order group as used in the OPRF and 3DH as specified in this document both adhere to the requirements as [JKX18].

* [JKX18] 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 the [I-D.irtf-cfrg-voprf], which is identical to the DH-OPRF functionality in [JKX18] and, concretely, uses the hash-to-curve functions in [I-D.irtf-cfrg-hash-to-curve]. All hash-to-curve methods in [I-D.irtf-cfrg-hash-to-curve] are compliant with the requirement in [JKX18], namely, that the output be a member of the prime-order group.

* [JKX18] and [I-D.krawczyk-cfrg-opaque-03] 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 C.1). Since 3DH satisfies the essential requirements for the AKE as described in [JKX18] and [I-D.krawczyk-cfrg-opaque-03], 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 in [JKX18], Figure 12 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 [JKX18]. This was done to align with best practices, such as is done for key exchange protocols like TLS 1.3 [RFC8446].

* Neither [JKX18] nor [I-D.krawczyk-cfrg-opaque-03] 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 D.

10.2. Security Analysis

Jarecki et al. [JKX18] proved the security of OPAQUE in a strong aPAKE model that ensures security against pre-computation attacks and is formulated in the Universal Composability (UC) framework [Canetti01] under the random oracle model. This assumes security of the OPRF function and the underlying key exchange protocol. In turn, the security of the OPRF protocol from [OPRF] is proven in the random oracle model under the One-More Diffie-Hellman assumption [JKKX16].

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 pre-computation 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).

10.3. Related Protocols

Despite the existence of multiple designs for (PKI-free) aPAKE protocols, none of these protocols are secure against pre-computation attacks. This includes protocols that have recent analyses in the UC model such as AuCPace [AuCPace] and SPAKE2+ [SPAKE2plus]. In particular, none of these protocols can use the standard technique against pre-computation that combines secret random values ("salt") into the one-way password mappings. Either these protocols do not use a salt at all or, if they do, they transmit the salt from server to client in the clear, hence losing the secrecy of the salt and its defense against pre-computation.

We note that as shown in [JKX18], these protocols, and any aPAKE in the model from [GMR06], can be converted into an aPAKE secure against pre-computation attacks at the expense of an additional OPRF execution.

Beyond AuCPace and SPAKE2+, the most widely deployed PKI-free aPAKE is SRP [RFC2945], which is vulnerable to pre-computation attacks, lacks proof of security, and is less efficient than OPAQUE. Moreover, SRP requires a ring as it mixes addition and multiplication operations, and thus does not work over standard elliptic curves. OPAQUE is therefore a suitable replacement for applications that use SRP.

10.4. 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.

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 (hence determined during password registration) and to 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 be 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.

10.5. Export Key Usage

The export key can be used (separately from the OPAQUE protocol) to provide confidentiality and integrity to other data which only the client should be able to process. For instance, if the server is expected to maintain any client-side secrets which require a password to access, then this export key can be used to encrypt these secrets so that they remain hidden from the server.

10.6. 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 [OPRF] 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.

10.7. 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 offline 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 at the server. Applications SHOULD select parameters that balance cost and complexity. Note that in OPAQUE, the key stretching function is executed by the client, as opposed to the server. 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 extra information about the behavior of clients that have registered with the server. There are two types of attacks we consider:

1) An attacker tries to learn whether a given client identity is registered with a server, and 2) An attacker tries to learn whether a given client identity has recently completed registration, re-registered (e.g. after a password change), or changed its identity.

OPAQUE prevents these attacks during the authentication flow. The first is prevented 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 also 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.

Preventing the second type of attack requires the server to supply a credential_identifier value for a given client identity, consistently between the registration response and credential response; see Section 5.2.2 and Section 6.3.2.2. Note that credential_identifier can be set to client_identity for simplicity.

In the event of a server compromise that results in a re-registration of credentials for all compromised clients, the oprf_seed value MUST be resampled, resulting in a change in the oprf_key value for each client. Although this change can be detected by an adversary, it is only leaked upon password rotation after the exposure of the credential files, and equally affects all registered clients.

Finally, applications must use the same key recovery mechanism when using this prevention throughout their lifecycle. The envelope size may vary between mechanisms, so a switch could then be detected.

OPAQUE does not prevent either 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.

10.10. Protecting the Registration Masking Key

The user enumeration prevention method described in this documents uses a symmetric encryption key generated by the client on registration that is sent to the server over an authenticated channel, such as one provided by TLS [RFC8446]. In the event that this channel is compromised, this encryption key could be leaked to an attacker.

One mitigation against this threat is to additionally encrypt the RegistrationRecord sent from client to server at the application layer using public key encryption, e.g., with HPKE [RFC9180]. However, the details of this mechanism are out of scope of 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 pre-computation 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 aPAKE protocol. (OPAQUE enables defense against such offline dictionary attacks by distributing the server so that an offline attack is only possible if all - or a minimal number of - servers are compromised [JKX18].) Furthermore, if the server does not sample this OPRF key 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 for enforcing password rules. Doing so invalidates this important security property of OPAQUE and is NOT RECOMMENDED. Applications should move such checks to the client. Note that limited checks at the server are possible to implement, e.g., detecting repeated passwords.

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 the 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.

11. IANA Considerations

This document makes no IANA requests.

12. References

12.1. Normative References

[I-D.irtf-cfrg-voprf] Davidson, A., Faz-Hernandez, A., Sullivan, N., and C. A. Wood, "Oblivious Pseudorandom Functions (OPRFs) using Prime-Order Groups", Work in Progress, Internet-Draft, draft-irtf-cfrg-voprf-21, 21 February 2023, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-voprf-21>.

[OPRF] Davidson, A., Faz-Hernandez, A., Sullivan, N., and C. A. Wood, "Oblivious Pseudorandom Functions (OPRFs) using Prime-Order Groups", Work in Progress, Internet-Draft, draft-irtf-cfrg-voprf-21, 21 February 2023, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-voprf-21>.

[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/rfc/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/rfc/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/rfc/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/rfc/rfc8174>.

12.2. Informative References

[ARGON2] 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/rfc/rfc9106>.

[AuCPace] Haase, B. and B. Labrique, "AuCPace: Efficient verifier-based PAKE protocol tailored for the IIoT", http://eprint.iacr.org/2018/286 , 2018.

[BG04] Brown, D. and R. Galant, "The static Diffie-Hellman problem", http://eprint.iacr.org/2004/306 , 2004.

[Boyen09] Boyen, X., "HPAKE: Password Authentication Secure against Cross-Site User Impersonation", Cryptology and Network Security (CANS) , 2009.

[Canetti01] Canetti, R., "Universally composable security: A new paradigm for cryptographic protocols", IEEE Symposium on Foundations of Computer Science (FOCS) , 2001.

[Cheon06] Cheon, J. H., "Security analysis of the strong Diffie-Hellman problem", Euroctypt 2006 , 2006.

[FIPS202] National Institute of Standards and Technology (NIST), "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", 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", WETICE , 2000.

[GMR06]
Gentry, C., MacKenzie, P., and Z, Ramzan, "A method for making password-based key exchange resilient to server compromise", CRYPTO , 2006.

[HMQV] Krawczyk, H., "HMQV: A high-performance secure Diffie-Hellman protocol", CRYPTO , 2005.

[I-D.irtf-cfrg-hash-to-curve] Faz-Hernandez, A., Scott, S., Sullivan, N., Wahby, R. S., and C. A. Wood, "Hashing to Elliptic Curves", Work in Progress, Internet-Draft, draft-irtf-cfrg-hash-to-curve-16, 15 June 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-16>.

[I-D.krawczyk-cfrg-opaque-03] "The OPAQUE Asymmetric PAKE Protocol", n.d., <https://datatracker.ietf.org/doc/html/draft-krawczyk-cfrg-opaque-03>.

[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)", IEEE European Symposium on Security and Privacy , 2016.

[JKX18] Jarecki, S., Krawczyk, H., and J. Xu, "OPAQUE: An Asymmetric PAKE Protocol Secure Against Pre-Computation Attacks", Eurocrypt , 2018.

[keyagreement] Barker, E., Chen, L., Roginsky, A., Vassilev, A., and R. Davis, "Recommendation for pair-wise key-establishment schemes using discrete logarithm cryptography", National Institute of Standards and Technology report, DOI 10.6028/nist.sp.800-56ar3, April 2018, <https://doi.org/10.6028/nist.sp.800-56ar3>.

[LGR20] Len, J., Grubbs, P., and T. Ristenpart, "Partitioning Oracle Attacks", n.d., <https://eprint.iacr.org/2020/1491.pdf>.

[PAKE-Selection] "CFRG PAKE selection process repository", n.d., <https://github.com/cfrg/pake-selection>.

[PBKDF2] Kaliski, B., "PKCS #5: Password-Based Cryptography Specification Version 2.0", RFC 2898, DOI 10.17487/

RFC2898, September 2000, <https://www.rfc-editor.org/rfc/rfc2898>.

[RFC2945] Wu, T., "The SRP Authentication and Key Exchange System", RFC 2945, DOI 10.17487/RFC2945, September 2000, <https://www.rfc-editor.org/rfc/rfc2945>.

[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/rfc/rfc5869>.

[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/rfc/rfc8017>.

[RFC8125] Schmidt, J., "Requirements for Password-Authenticated Key Agreement (PAKE) Schemes", RFC 8125, DOI 10.17487/RFC8125, April 2017, <https://www.rfc-editor.org/rfc/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/rfc/rfc8446>.

[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/rfc/rfc9180>.

[SCRYPT] 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/rfc/rfc7914>.

[SPAKE2plus] Shoup, V., "Security Analysis of SPAKE2+", http://eprint.iacr.org/2020/313 , 2020.

[WhatsAppE2E] "Security of End-to-End Encrypted Backups", n.d., <https://www.whatsapp.com/security/WhatsApp_Security_Encrypted_Backups_Whitepaper.pdf>.

[_3DH] "Simplifying OTR deniability", https://signal.org/blog/simplifying-otr-deniability , 2016.

Appendix A. Acknowledgments

The OPAQUE protocol and its analysis is the joint work of the author with Stanislaw Jarecki and Jiayu Xu. We are indebted to the OPAQUE reviewers during CFRG's aPAKE selection process, particularly Julia Hesse and Bjorn Tackmann. This draft has benefited from comments by multiple people. Special thanks to Richard Barnes, Dan Brown, Eric Crockett, Paul Grubbs, Fredrik Kuivinen, Payman Mohassel, Jason Resch, Greg Rubin, and Nick Sullivan.

Appendix B. 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 C. Alternate AKE Instantiations

It is possible to instantiate OPAQUE with other AKEs, such as HMQV [HMQV] and 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 is quantum-safe, but the OPRF is not, would still ensure the confidentiality of application data encrypted under session_key (or a key derived from it) with a quantum-safe encryption function.

C.1. HMQV Instantiation Sketch

An HMQV instantiation would work similar 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_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.

C.2. SIGMA-I Instantiation Sketch

A 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.

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 D. Test Vectors

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

Similarly, each fake test vector in Appendix D.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 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. These test vectors were generated using draft-10 of [OPRF].

D.1. Real Test Vectors

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

D.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
```

D.1.1.2. Input Values

```
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
server_keyshare: c8c39f573135474c51660b02425bca633e339cec4e1acc69c94d
d48497fe4028
client_keyshare: 0c3a00c961fead8a16f818929cc976f0475e4f723519318b96f4
947a7a5f9663
server_private_keyshare: 2e842960258a95e28bcfef489cffd19d8ec99cc1375d
840f96936da7dbb0b40d
client_private_keyshare: 22c919134c9bdd9dc0c5ef3450f18b54820f43f646a9
5223bf4a85b2018c2001
blind_registration: 76cfbfe758db884bebb33582331ba9f159720ca8784a2a070
a265d9c2d6abe01
blind_login: 6ecc102d2e7a7cf49617aad7bbe188556792d4acd60a1a8a8d2b65d4
b0790308
```

D.1.1.3. Intermediate Values

```
client_public_key: 2ec892bdbf9b3e2ea834be9eb11f5d187e64ba661ec041c0a3
b66db8b7d6cc30
auth_key: 6cd32316f18d72a9a927a83199fa030663a38ce0c11fbaef82aa9003773
0494fc555c4d49506284516edd1628c27965b7555a4ebfed2223199f6c67966dde822
randomized_pwd: aac48c25ab036e30750839d31d6e73007344cb1155289fb7d329b
eb932e9adeea73d5d5c22a0ce1952f8aba6d66007615cd1698d4ac85ef1fcf150031d
1435d9
envelope: ac13171b2f17bc2c74997f0fce1e1f35bec6b91fe2e12dbd323d23ba7a3
8dfecb9dbe7d48cf714fc3533becab6faf60b783c94d258477eb74ecc453413bf61c5
3fd58f0fb3c1175410b674c02e1b59b2d729a865b709db3dc4ee2bb45703d5a8
handshake_secret: 562564da0d4efdc73cb6efbb454388dabfa5052d4e7e83f4d02
40c5afd8352881e762755c2f1a9110e36b05fe770f0f48658489c9730dcd365e6c2d4
049c8fe3
server_mac_key: 59473632c53a647f9f4ab4d6c3b81e241dd9cb19ca05f0eabed7e
593f0407ff57e7f060621e5e48d5291be600a1959fbecbc26d4a7157bd227a993c37b
645f73
client_mac_key: f2d019bad603b45b2ac50376279a0a37d097723b5405aa4fb20a5
9f60cdbdd52ec043372cedcdbbdb634c54483e1be51a88d13a5798180acb84c10b129
7069fd
oprf_key: 5d4c6a8b7c7138182afb4345d1fae6a9f18a1744afbcc3854f8f5a2b4b4
c6d05
```

D.1.1.4. Output Values

```
registration_request: 5059ff249eb1551b7ce4991f3336205bde44a105a032e74
7d21bf382e75f7a71
registration_response: 7408a268083e03abc7097fc05b587834539065e86fb0c7
b6342fcf5e01e5b019b2fe7af9f48cc502d016729d2fe25cdd433f2c4bc904660b2a3
82c9b79df1a78
registration_upload: 2ec892bdbf9b3e2ea834be9eb11f5d187e64ba661ec041c0
a3b66db8b7d6cc301ac5844383c7708077dea41cbefe2fa15724f449e535dd7dd562e
66f5ecfb95864eadddec9db5874959905117dad40a4524111849799281fefe3c51fa8
2785c5ac13171b2f17bc2c74997f0fce1e1f35bec6b91fe2e12dbd323d23ba7a38dfe
cb9dbe7d48cf714fc3533becab6faf60b783c94d258477eb74ecc453413bf61c53fd5
8f0fb3c1175410b674c02e1b59b2d729a865b709db3dc4ee2bb45703d5a8
KE1: c4dedb0ba6ed5d965d6f250fbe554cd45cba5dfcce3ce836e4aee778aa3cd44d
da7e07376d6d6f034cfa9bb537d11b8c6b4238c334333d1f0aebb380cae6a6cc0c3a0
0c961fead8a16f818929cc976f0475e4f723519318b96f4947a7a5f9663
KE2: 7e308140890bcde30cbcea28b01ea1ecfbd077cff62c4def8efa075aabcbb471
38fe59af0df2c79f57b8780278f5ae47355fe1f817119041951c80f612fdfc6dd6ec6
0bcdb26dc455ddf3e718f1020490c192d70dfc7e403981179d8073d1146a4f9aa1ced
4e4cd984c657eb3b54ced3848326f70331953d91b02535af44d9fe0610f003be80cb2
098357928c8ea17bb065af33095f39d4e0b53b1687f02d522d96bad4ca354293d5c40
1177ccbd302cf565b96c327f71bc9eaf2890675d2fbb71cd9960ecef2fe0d0f749498
6fa3d8b2bb01963537e60efb13981e138e3d4a1c8c39f573135474c51660b02425bca
633e339cec4e1acc69c94dd48497fe40287f33611c2cf0eef57adbf48942737d9421e
6b20e4b9d6e391d4168bf4bf96ea57aa42ad41c977605e027a9ef706a349f4b2919fe
3562c8e86c4eeecf2f9457d4
KE3: df9a13cd256091f90f0fcb2ef6b3411e4aebff07bb0813299c0ec7f5dedd33a7
681231a001a82f1dece1777921f42abfeee551ee34392e1c9743c5cc1dc1ef8c
export_key: 1ef15b4fa99e8a852412450ab78713aad30d21fa6966c9b8c9fb3262a
970dc62950d4dd4ed62598229b1b72794fc0335199d9f7fcc6eaedde92cc04870e63f
16
session_key: 8a0f9f4928fc0c3b5bb261c4b7b3997600405424a8128632e85a5667
b4b742484ed791933971be6d3fcf2b23c56b8e8f7e7edcae19a03b8fd87f5999fce12
9d2
```

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

D.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
```

D.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
server_keyshare: c8c39f573135474c51660b02425bca633e339cec4e1acc69c94d
d48497fe4028
client_keyshare: 0c3a00c961fead8a16f818929cc976f0475e4f723519318b96f4
947a7a5f9663
server_private_keyshare: 2e842960258a95e28bcfef489cffd19d8ec99cc1375d
840f96936da7dbb0b40d
client_private_keyshare: 22c919134c9bdd9dc0c5ef3450f18b54820f43f646a9
5223bf4a85b2018c2001
blind_registration: 76cfbfe758db884bebb33582331ba9f159720ca8784a2a070
a265d9c2d6abe01
blind_login: 6ecc102d2e7a7cf49617aad7bbe188556792d4acd60a1a8a8d2b65d4
b0790308
```

D.1.2.3. Intermediate Values

```
client_public_key: 2ec892bdbf9b3e2ea834be9eb11f5d187e64ba661ec041c0a3
b66db8b7d6cc30
auth_key: 6cd32316f18d72a9a927a83199fa030663a38ce0c11fbaef82aa9003773
0494fc555c4d49506284516edd1628c27965b7555a4ebfed2223199f6c67966dde822
randomized_pwd: aac48c25ab036e30750839d31d6e73007344cb1155289fb7d329b
eb932e9adeea73d5d5c22a0ce1952f8aba6d66007615cd1698d4ac85ef1fcf150031d
1435d9
envelope: ac13171b2f17bc2c74997f0fce1e1f35bec6b91fe2e12dbd323d23ba7a3
8dfec1ac902dc5589e9a5f0de56ad685ea8486210ef41449cd4d8712828913c5d2b68
0b2b3af4a26c765cff329bfb66d38ecf1d6cfa9e7a73c222c6efe0d9520f7d7c
handshake_secret: bc2abaa979af9cbb6859856b7d5d201a038fbdfa7e10f11d131
d3f8f6fc3b263bde4db6d2d9207d4648ff80415a276d5f157f9d37a3eade559db2e5f
3fa026b2
server_mac_key: 2420461c589866700b08c8818cbf390c872629a14cf32a264dad3
375f85f33188c8f04bdb71880b2d4613187a0e416808ab62b45858b88319882602371
ef5f75
client_mac_key: 156e4ab0b9f71ef994bbbb73928e6d14d7335cf9561f113d61ac6
b41fab35f9c72fe827d3c4d7dd91d8398ee619810e4f9286e6b32f329eb6b1476ce18
fa8500
oprf_key: 5d4c6a8b7c7138182afb4345d1fae6a9f18a1744afbcc3854f8f5a2b4b4
c6d05
```

D.1.2.4. Output Values

```
registration_request: 5059ff249eb1551b7ce4991f3336205bde44a105a032e74
7d21bf382e75f7a71
registration_response: 7408a268083e03abc7097fc05b587834539065e86fb0c7
b6342fcf5e01e5b019b2fe7af9f48cc502d016729d2fe25cdd433f2c4bc904660b2a3
82c9b79df1a78
registration_upload: 2ec892bdbf9b3e2ea834be9eb11f5d187e64ba661ec041c0
a3b66db8b7d6cc301ac5844383c7708077dea41cbefe2fa15724f449e535dd7dd562e
66f5ecfb95864eadddec9db5874959905117dad40a4524111849799281fefe3c51fa8
2785c5ac13171b2f17bc2c74997f0fce1e1f35bec6b91fe2e12dbd323d23ba7a38dfe
c1ac902dc5589e9a5f0de56ad685ea8486210ef41449cd4d8712828913c5d2b680b2b
3af4a26c765cff329bfb66d38ecf1d6cfa9e7a73c222c6efe0d9520f7d7c
KE1: c4dedb0ba6ed5d965d6f250fbe554cd45cba5dfcce3ce836e4aee778aa3cd44d
da7e07376d6d6f034cfa9bb537d11b8c6b4238c334333d1f0aebb380cae6a6cc0c3a0
0c961fead8a16f818929cc976f0475e4f723519318b96f4947a7a5f9663
KE2: 7e308140890bcde30cbcea28b01ea1ecfbd077cff62c4def8efa075aabcbb471
38fe59af0df2c79f57b8780278f5ae47355fe1f817119041951c80f612fdfc6dd6ec6
0bcdb26dc455ddf3e718f1020490c192d70dfc7e403981179d8073d1146a4f9aa1ced
4e4cd984c657eb3b54ced3848326f70331953d91b02535af44d9fea502150b67fe367
95dd8914f164e49f81c7688a38928372134b7dccd50e09f8fed9518b7b2f94835b3c4
fe4c8475e7513f20eb97ff0568a39caee3fd6251876f71cd9960ecef2fe0d0f749498
6fa3d8b2bb01963537e60efb13981e138e3d4a1c8c39f573135474c51660b02425bca
633e339cec4e1acc69c94dd48497fe4028c463164503598ea84fab9005b9cd51b7bb3
206fb22a412e8a86b9cb6ffca18f5ea6b4c24fdc94865e8bf74248e6be15b85b16041
40ffad2175f9518452d381af
KE3: a86ece659d90525e2476aa1756d313b067581cb7b0643b97be6b8ab8d0f10843
57e514ecfaff9dc18f6cca37da630545f0048393f16bc175eb819653ebc45b60
export_key: 1ef15b4fa99e8a852412450ab78713aad30d21fa6966c9b8c9fb3262a
970dc62950d4dd4ed62598229b1b72794fc0335199d9f7fcc6eaedde92cc04870e63f
16
session_key: 0968e91efeb702d6aa09023a9a79803332d8bd3442a79b8ad09490b9
267161013bf475bed945238a5e976ef7d7de7ff41ae30439fe2fc39758fb3e56f2683
e60
```

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

D.1.3.1. 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
```

D.1.3.2. 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
server_keyshare: 020e67941e94deba835214421d2d8c90de9b0f7f925d11e2032c
e19b1832ae8e0f
client_keyshare: 03493f36ca12467d1f5eaaabea67ca31377c4869c1e9a62346b6
f01a991624b95d
server_private_keyshare: 9addab838c920fa7044f3a46b91ecaea24b0e7203992
8ee7d4c37a5b9bc17349
client_private_keyshare: 89d5a7e18567f255748a86beac13913df755a5adf776
d69e143147b545d22134
blind_registration: 411bf1a62d119afe30df682b91a0a33d777972d4f2daa4b34
ca527d597078153
blind_login: c497fddf6056d241e6cf9fb7ac37c384f49b357a221eb0a802c989b9
942256c1
```

D.1.3.3. Intermediate Values

```
client_public_key: 02dc91b178ba2c4bbf9b9403fca25457b906a7f507e59b6e70
3031e09114ba2be0
auth_key: 5bd4be1602516092dc5078f8d699f5721dc1720a49fb80d8e5c16377abd
0987b
randomized_pwd: 06be0a1a51d56557a3adad57ba29c5510565dcd8b5078fa319151
b9382258fb0
envelope: a921f2a014513bd8a90e477a629794e89fec12d12206dde662ebdcf6567
0e51fe155412cb432898eda63529c3b2633521f770cccbd25d7548a4e20665a45e65a
handshake_secret: c59197dd9269abfdb3037ea1c203a97627e2c0aa142000d1c3f
06a2c8713077d
server_mac_key: a431a5c1d3cb5772cbc66af0c2851e23dd9ad153a0c8b99081c7d
0d543173fde
client_mac_key: 7329ffd54df21db5532fce8794fca78b505fef9397aad28a424f6
ea3f97c51ca
oprf_key: 2dfb5cb9aa1476093be74ca0d43e5b02862a05f5d6972614d7433acdc66
f7f31
```

D.1.3.4. Output Values

```
registration_request: 029e949a29cfa0bf7c1287333d2fb3dc586c41aa652f507
0d26a5315a1b50229f8
registration_response: 0350d3694c00978f00a5ce7cd08a00547e4ab5fb5fc2b2
f6717cdaa6c89136efef035f40ff9cf88aa1f5cd4fe5fd3da9ea65a4923a5594f84fd
9f2092d6067784874
registration_upload: 02dc91b178ba2c4bbf9b9403fca25457b906a7f507e59b6e
703031e09114ba2be07f0ed53532d3ae8e505ecc70d42d2b814b6b0e48156def71ea0
29148b2803aafa921f2a014513bd8a90e477a629794e89fec12d12206dde662ebdcf6
5670e51fe155412cb432898eda63529c3b2633521f770cccbd25d7548a4e20665a45e
65a
KE1: 037342f0bcb3ecea754c1e67576c86aa90c1de3875f390ad599a26686cdfee6e
07ab3d33bde0e93eda72392346a7a73051110674bbf6b1b7ffab8be4f91fdaeeb1034
93f36ca12467d1f5eaaabea67ca31377c4869c1e9a62346b6f01a991624b95d
KE2: 0246da9fe4d41d5ba69faa6c509a1d5bafd49a48615a47a8dd4b0823cc147648
1138fe59af0df2c79f57b8780278f5ae47355fe1f817119041951c80f612fdfc6d2f0
c547f70deaeca54d878c14c1aa5e1ab405dec833777132eea905c2fbb12504a67dcbe
0e66740c76b62c13b04a38a77926e19072953319ec65e41f9bfd2ae2687bd3348bfe3
3cb0bb9864fdb3b307f7dd68a17f3f150074a0bfc830ab889717d71cd9960ecef2fe0
d0f7494986fa3d8b2bb01963537e60efb13981e138e3d4a1020e67941e94deba83521
4421d2d8c90de9b0f7f925d11e2032ce19b1832ae8e0fb5166145361a2c344d9737dd
5c826fede3bbfafa418ad379ce4fa65fbb15db6e
KE3: 272d04758b2b436bf0239ba7b9bd0a1686a9b6542ceaaf08732054beda956498
export_key: c3c9a1b0e33ac84dd83d0b7e8af6794e17e7a3caadff289fbd9dc769a
853c64b
session_key: a224790a010afc0a3f37e23c1b7a5cb7f9e73e3d9a924116510d97d8
0e2a1e0c
```

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

D.1.4.1. 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
```

D.1.4.2. 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
server_keyshare: 020e67941e94deba835214421d2d8c90de9b0f7f925d11e2032c
e19b1832ae8e0f
client_keyshare: 03493f36ca12467d1f5eaaabea67ca31377c4869c1e9a62346b6
f01a991624b95d
server_private_keyshare: 9addab838c920fa7044f3a46b91ecaea24b0e7203992
8ee7d4c37a5b9bc17349
client_private_keyshare: 89d5a7e18567f255748a86beac13913df755a5adf776
d69e143147b545d22134
blind_registration: 411bf1a62d119afe30df682b91a0a33d777972d4f2daa4b34
ca527d597078153
blind_login: c497fddf6056d241e6cf9fb7ac37c384f49b357a221eb0a802c989b9
942256c1
```

D.1.4.3. Intermediate Values

```
client_public_key: 02dc91b178ba2c4bbf9b9403fca25457b906a7f507e59b6e70
3031e09114ba2be0
auth_key: 5bd4be1602516092dc5078f8d699f5721dc1720a49fb80d8e5c16377abd
0987b
randomized_pwd: 06be0a1a51d56557a3adad57ba29c5510565dcd8b5078fa319151
b9382258fb0
envelope: a921f2a014513bd8a90e477a629794e89fec12d12206dde662ebdcf6567
0e51f4d7773a36a208a866301dbb2858e40dc5638017527cf91aef32d3848eebe0971
handshake_secret: 0ee4a82c4a34992f72bfbcb5d2ce64044477dfe200b9d8c92bf
1759b219b3485
server_mac_key: 77ebd7511216a51e9c2f3368ce6c1e40513f24b6f42085ef18e7f
737b427aab5
client_mac_key: e48e2064cf570dbd18eb42550d4459c58ac4ae4e28881d1aefbab
d668f7f1df9
oprf_key: 2dfb5cb9aa1476093be74ca0d43e5b02862a05f5d6972614d7433acdc66
f7f31
```

D.1.4.4. Output Values

```
registration_request: 029e949a29cfa0bf7c1287333d2fb3dc586c41aa652f507
0d26a5315a1b50229f8
registration_response: 0350d3694c00978f00a5ce7cd08a00547e4ab5fb5fc2b2
f6717cdaa6c89136efef035f40ff9cf88aa1f5cd4fe5fd3da9ea65a4923a5594f84fd
9f2092d6067784874
registration_upload: 02dc91b178ba2c4bbf9b9403fca25457b906a7f507e59b6e
703031e09114ba2be07f0ed53532d3ae8e505ecc70d42d2b814b6b0e48156def71ea0
29148b2803aafa921f2a014513bd8a90e477a629794e89fec12d12206dde662ebdcf6
5670e51f4d7773a36a208a866301dbb2858e40dc5638017527cf91aef32d3848eebe0
971
KE1: 037342f0bcb3ecea754c1e67576c86aa90c1de3875f390ad599a26686cdfee6e
07ab3d33bde0e93eda72392346a7a73051110674bbf6b1b7ffab8be4f91fdaeeb1034
93f36ca12467d1f5eaaabea67ca31377c4869c1e9a62346b6f01a991624b95d
KE2: 0246da9fe4d41d5ba69faa6c509a1d5bafd49a48615a47a8dd4b0823cc147648
1138fe59af0df2c79f57b8780278f5ae47355fe1f817119041951c80f612fdfc6d2f0
c547f70deaeca54d878c14c1aa5e1ab405dec833777132eea905c2fbb12504a67dcbe
0e66740c76b62c13b04a38a77926e19072953319ec65e41f9bfd2ae268d7f10604202
1c80300e4c6f585980cf39fc51a4a6bba41b0729f9b240c729e5671cd9960ecef2fe0
d0f7494986fa3d8b2bb01963537e60efb13981e138e3d4a1020e67941e94deba83521
4421d2d8c90de9b0f7f925d11e2032ce19b1832ae8e0fdca637d2a5390f4c809a67b4
6977c536fe9f643f703178a17a413d14e4bb523c
KE3: 298cd0077d018f122bc95d706e5fef06537814c567f08d5e40b0c0ae918f9287
export_key: c3c9a1b0e33ac84dd83d0b7e8af6794e17e7a3caadff289fbd9dc769a
853c64b
session_key: 0c59872e9bcdde274f4f52f6ba0fd1acca211d6eb7db98677b457a73
9ef1f0d8
```

D.2. Fake Test Vectors

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

D.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
```

D.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
server_keyshare: 5236e2e06d49f0b496db2a786f6ee1016f15b4fd6c0dbd95d6b1
17055d914157
server_private_keyshare: 6d8fba9741a357584770f85294430bce2252fe212a8a
372152a73c7ffe414503
masking_key: 39ebd51f0e39a07a1c2d2431995b0399bca9996c5d10014d6ebab445
3dc10ce5cef38ed3df6e56bfff40c2d8dd4671c2b4cf63c3d54860f31fe40220d690b
b71
KE1: b0a26dcaca2230b8f5e4b1bcab9c84b586140221bb8b2848486874b0be448905
42d4e61ed3f8d64cdd3b9d153343eca15b9b0d5e388232793c6376bd2d9cfd0a0e4ed
8bcc15f3dd01a30365c97c0c0de0a3dd3fbf5d3cbec55fb6ac1d3bf740f
```

D.2.1.3. Output Values

```
KE2: 928f79ad8df21963e91411b9f55165ba833dea918f441db967cdc09521d22925
9c035896a043e70f897d87180c543e7a063b83c1bb728fbd189c619e27b6e5a632b5a
b1bff96636144faa4f9f9afaac75dd88ea99cf5175902ae3f3b2195693f165f11929b
a510a5978e64dcdabecbd7ee1e4380ce270e58fea58e6462d92964a1aaef72698bca1
c673baeb04cc2bf7de5f3c2f5553464552d3a0f7698a9ca7f9c5e70c6cb1f706b2f17
5ab9d04bbd13926e816b6811a50b4aafa9799d5ed7971e10f6eeab2a7a420bf09da9b
27a4639645622c46358de9cf7ae813055ae2d125236e2e06d49f0b496db2a786f6ee1
016f15b4fd6c0dbd95d6b117055d914157cb5e11625c701e642293ad32bfcf88da653
c9b6e71efc8a89607fd46ed5e7b9bf7cc7dbb997a4fd41194a04bcd0c5d88052e080a
2f02c68d8d9e9c0ce15c92ff
```

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

D.2.2.1. 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
```

D.2.2.2. 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
server_keyshare: 03f42965d5bcba2a590a49eb2418061effe40b5c29a34b8e5163
e0ef32044b2e4c
server_private_keyshare: 1a2a0ff27f3ca75221378a2a21fe5222ce0b439452f8
70475857a34197ba8f6d
masking_key: caecc6ccb4cae27cb54d8f3a1af1bac52a3d53107ce08497cdd362b1
992e4e5e
KE1: 0396875da2b4f7749bba411513aea02dc514a48d169d8a9531bd61d3af3fa9ba
ae42d4e61ed3f8d64cdd3b9d153343eca15b9b0d5e388232793c6376bd2d9cfd0a039
94d4f1221bfd205063469e92ea4d492f7cc76a327223633ab74590c30cf7285
```

D.2.2.3. Output Values

KE2: 0201198dcd13f9792eb75dcfa815f61b049abfe2e3e9456d4bbbceec5f442efd
049c035896a043e70f897d87180c543e7a063b83c1bb728fbd189c619e27b6e5a6fac
da65ce0a97b9085e7af07f61fd3fdd046d257cbf2183ce8766090b8041a8bf28d79dd
4c9031ddc75bb6ddb4c291e639937840e3d39fc0d5a3d6e7723c09f7945df485bcf9a
efe3fe82d149e84049e259bb5b33d6a2ff3b25e4bfb7eff0962821e10f6eeab2a7a42
0bf09da9b27a4639645622c46358de9cf7ae813055ae2d1203f42965d5bcba2a590a4
9eb2418061effe40b5c29a34b8e5163e0ef32044b2e4c196137813ed8ec48627f0b0d
90d9427f4ec137f8360769df167c25836eae5d91

Authors' Addresses

Daniel Bourdrez

Email: d@bytema.re

Hugo Krawczyk
Algorand Foundation

Email: hugokraw@gmail.com

Kevin Lewi
Novi Research

Email: lewi.kevin.k@gmail.com

Christopher A. Wood
Cloudflare, Inc.

Email: caw@heapingbits.net