Workgroup: Network Working Group
Internet-Draft: draft-irtf-cfrg-opaque-08
Published: 7 March 2022
Intended Status: Informational
Expires: 8 September 2022
Authors: D. Bourdrez
H. Krawczyk, Algorand Foundation
K. Lewi, 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
* Key Recovery Mechanism; Section 2.4
* Authenticated Key Exchange (AKE) protocol; Section 2.5

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 -09, 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 this 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:

* Evaluate(k, blinded_element): Evaluate blinded input element blinded_element using input key k, yielding output element evaluated_element. This is equivalent to the Evaluate 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, as described in [OPRF], Section 3.2. In this specification, the info parameter to DeriveKeyPair is set to "OPAQUE-DeriveKeyPair".

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.

This specification uses the OPRF mode (0x00) from [OPRF], Section 3.1.

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 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.

A Key Stretching Function (KSF) 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.

2.4. Key Recovery Method

OPAQUE relies on a key recovery mechanism for storing authentication material on the server and recovering it on the client. This material is encapsulated in an envelope, whose structure, encoding, and size must be specified by the key recovery mechanism. The size of the envelope is denoted Ne and may vary between mechanisms.

The key recovery storage mechanism takes as input a private seed and outputs an envelope. The retrieval process takes as input a private seed and envelope and outputs authentication material. The signatures for these functionalities are as follows:

* Store(private_seed): build and return an Envelope structure and the client's public key.

* Recover(private_seed, envelope): recover and return the authentication material for the AKE from the Envelope. This function raises an error if the private seed cannot be used for recovering authentication material from the input envelope.

The key recovery mechanism MUST return an error when trying to recover authentication material from an envelope with a private seed that was not used in producing the envelope.

Moreover, it MUST be compatible with the chosen AKE. For example, the key recovery mechanism specified in Section 4.1 directly recovers a private key from a seed, and the cryptographic primitive in the AKE must therefore support such a possibility.

If applications implement Section 10.9, they MUST use the same mechanism throughout their lifecycle in order to avoid activity leaks due to switching.

2.5. Authenticated Key Exchange (AKE) Protocol

OPAQUE additionally depends on a three-message Authenticated Key Exchange (AKE) protocol which satisfies the forward secrecy and KCI properties discussed in Section 10.

The AKE must define three messages AuthInit, AuthResponse and AuthFinish and provide the following functions for the client:

* Start(): Initiate the AKE by producing message AuthInit.

* ClientFinish(client_identity, client_private_key, server_identity, server_public_key, AuthInit): upon receipt of the server's response AuthResponse, complete the protocol for the client, produce AuthFinish.

The AKE protocol must provide the following functions for the server:

* Response(server_identity, server_private_key, client_identity, client_public_key, AuthInit): upon receipt of a client's request AuthInit, engage in the AKE.

* ServerFinish(AuthFinish): upon receipt of a client's final AKE message AuthFinish, complete the protocol for the server.

Both ClientFinish and ServerFinish return an error if authentication failed. In this case, clients and servers MUST NOT use any outputs from the protocol, such as session_key or export_key (defined below).

Prior to the execution of these functions, both the client and the server MUST agree on a configuration; see Section 7 for details.

This specification defines one particular AKE based on 3DH; see Section 6.4. 3DH assumes a prime-order group as described in [OPRF], Section 2.1.

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 the client recovers its authentication material and uses it to perform a mutually authenticated key exchange.

3.1. Setup

Previously to both stages, the client and server agree on a configuration, which fully specifies the cryptographic algorithm dependencies necessary to run the protocol; see Section 7 for details. The client chooses its password, and the server chooses its own 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 the same 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 part in OPAQUE that requires a server-authenticated and confidential channel, either physical, out-of-band, PKI-based, etc.

The client inputs its credentials, which includes its password and user identifier, and the server inputs its parameters, which includes 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 client 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 Record, 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):

* KE1 is composed of the CredentialRequest and AuthInit messages
* KE2 is composed of the CredentialResponse and AuthResponse messages
* KE3 represents the AuthFinish message

The CredentialRequest and CredentialResponse message contents and wire format are specified in Section 6.3, and those of AuthInit, AuthResponse and AuthFinish are specified in Section 6.4.1.

The rest of this document describes the details of these stages in detail. Section 4 describes how client credential information is generated, encoded, stored on the server on registration, and recovered on 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 key using the DeriveAuthKeyPair() function defined in Section 6.4.2.

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: 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.

```
Store

Input:
- randomized_pwd, 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, 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: client password.

* creds: client credentials, as described in Section 4.

The server inputs the following values:

* server_private_key: server private key for the AKE protocol.

* server_public_key: server public key for the AKE protocol.

* credential_identifier: unique identifier for the client's credential, generated by the server.

* oprf_seed: 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) = FinalizeRequest(response,
                                       server_identity,
                                       client_identity)

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

Section 5.2 describes details of the functions and the corresponding parameters referenced above.

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

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

data A serialized OPRF group element.

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

data A serialized OPRF group element.

server_public_key The server's encoded public key that will be used for the online authenticated key exchange 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

5.2.1. CreateRegistrationRequest

```
CreateRegistrationRequest

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

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

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

```
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.

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

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

  Create RegistrationResponse response with (evaluated_message, server_p
  return response
```

5.2.3. FinalizeRequest

To create the user record used for further authentication, the client executes the following function.

```
FinalizeRequest

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 FinalizeRequest(password, blind, response, server_identity, client_i
  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 RegistrationUpload record with (client_public_key, masking_key,
  return (record, export_key)
```

See Section 6 for details about the output export_key usage.

Upon completion of this function, the client MUST send record to the server.

5.3. Finalize Registration

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.

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: client password.

* client_identity: client identity, as described in Section 4.

The server inputs the following values:

* server_private_key: server private for the AKE protocol.

* server_public_key: server public for the AKE protocol.

* server_identity: server identity, as described in Section 4.

* record: RegistrationUpload corresponding to the client's registration.

* credential_identifier: an identifier that uniquely represents the credential.

* oprf_seed: 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, password,
                             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 may have the following named fields:

* password, the input password; and

* blind, the random blinding inverter returned by Blind(); and

* client_ake_state, the client's AKE state if necessary.

The server state may have the following fields:

* server_ake_state, the server's AKE state if necessary.

The rest of this section describes these authenticated key exchange messages and their parameters in more detail. 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. Client Authentication Functions

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.blind = blind
  ake_1 = Start(request)
  Output KE1(request, ake_1)
```

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.CredentialRespon
                       server_identity, client_identity)
  (ke3, session_key) =
    ClientFinalize(client_identity, client_private_key, server_identity,
                   server_public_key, ke2)
  return (ke3, session_key)
```

6.2. Server Authentication Functions

```
ServerInit

Input:
- server_identity, the optional encoded server identity, which is set to
  server_public_key if nil.
- 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 encoded client identity.

Output:
- ke2, a KE2 structure.

def ServerInit(server_identity, server_private_key, server_public_key,
               record, credential_identifier, oprf_seed, ke1, client_ide
  response = CreateCredentialResponse(ke1.request, server_public_key, re
    credential_identifier, oprf_seed)
  ake_2 = Response(server_identity, server_private_key,
    client_identity, record.client_public_key, ke1, response)
  return KE2(response, ake_2)
```

Since the OPRF is a two-message protocol, KE3 has no element of the OPRF. We can therefore call the AKE's ServerFinish() directly. The ServerFinish() function MUST take KE3 as input and MUST verify the client authentication material it contains before the session_key value can be used. This verification is paramount in order to ensure forward secrecy against active attackers.

This function MUST NOT return the session_key value if the client authentication material is invalid, and may instead return an appropriate error message.

6.3. Credential Retrieval

6.3.1. Credential Retrieval Messages

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

data A serialized OPRF group element.

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

data 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

6.3.2.1. CreateCredentialRequest

```
CreateCredentialRequest

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

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

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

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 = Evaluate(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 + Ne)
  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 Ne

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 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

```
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 + Ne)
  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 AKE client state client_ake_state mentioned in Section 6 has the following named fields:

- client_secret, an opaque byte string of length Nsk; and
- ke1, a value of type KE1.

The server state server_ake_state mentioned in Section 6 has the following fields:

- expected_client_mac, an opaque byte string of length Nm; and
- session_key, an opaque byte string of length Nx.

Section 6.4.4 and Section 6.4.5 specify the inner workings of client and server functions, respectively.

6.4.1. AKE Messages

```
struct {
  uint8 client_nonce[Nn];
  uint8 client_keyshare[Npk];
} AuthInit;
```

client_nonce : A fresh randomly generated nonce of length Nn.

client_keyshare : Client ephemeral key share of fixed size Npk.

```
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 : Server ephemeral key share 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 {
  uint8 client_mac[Nm];
} AuthFinish;
```

client_mac : An authentication tag computed over the handshake transcript computed using Km2, defined below.

6.4.2. 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.

6.4.3. Key Schedule Functions

6.4.3.1. 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.
- ke2, a KE2 message 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,
                    ke2.credential_response,
                    ke2.AuthResponse.server_nonce, ke2.AuthResponse.ser
  return preamble
```

6.4.3.2. 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)
```

6.4.4. 3DH Client Functions

```
Start

Parameters:
- Nn, the nonce length.

State:
- state, a ClientState structure.

Input:
- credential_request, a CredentialRequest structure.

Output:
- ke1, a KE1 structure.

def Start(credential_request):
  client_nonce = random(Nn)
  (client_secret, client_keyshare) = GenerateAuthKeyPair()
  Create KE1 ke1 with (credential_request, client_nonce, client_keyshare
  Populate state with ClientState(client_secret, ke1)
  return (ke1, client_secret)
```

```
ClientFinalize

State:
- state, a ClientState 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 ClientFinalize(client_identity, client_private_key, server_identity,
                   server_public_key, ke2):

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

  preamble = Preamble(client_identity, state.ke1, server_identity, ke2.i
  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)
```

6.4.5. 3DH Server Functions

Response

Parameters:
- Nn, the nonce length.

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.
- 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:
- ke2, a KE2 structure.

```
def Response(server_identity, server_private_key, client_identity,
             client_public_key, ke1, credential_response):
  server_nonce = random(Nn)
  (server_private_keyshare, server_keyshare) = GenerateAuthKeyPair()
  Create inner_ke2 ike2 with (ke1.credential_response, server_nonce, ser
  preamble = Preamble(client_identity, ke1, server_identity, ike2)

  dh1 = SerializeElement(server_private_keyshare * ke1.client_keyshare)
  dh2 = SerializeElement(server_private_key * ke1.client_keyshare)
  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))
  Populate state with ServerState(expected_client_mac, session_key)
  Create KE2 ke2 with (ike2, server_mac)
  return ke2
```

```
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.

Exceptions:
- ClientAuthenticationError, the handshake fails.

def ServerFinish(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 OPRF(ristretto255, SHA-512) and OPRF(P-256, SHA-256).

* 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 Argon2 [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 OPRF(ristretto255, SHA-512), then Group SHOULD be ristretto255.

Context is the shared parameter used to construct the preamble in Section 6.4.3.1. 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:

* OPRF(ristretto255, SHA-512), HKDF-SHA-512, HMAC-SHA-512, SHA-512, Scrypt(32768,8,1), internal, ristretto255

* OPRF(P-256, SHA-256), HKDF-SHA-256, HMAC-SHA-256, SHA-256, Scrypt(32768,8,1), internal, 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 an 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 in order 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 against 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 in order 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 is 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 specifiation, 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 6.4.4.

* ClientAuthenticationError: The server failed to complete the authenticated key exchange protocol with the client; Section 6.4.5.

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 implementaton 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 equivocability 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 an 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 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 in 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 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, 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 as for domain separation between different applications of the protocol. This change was made to allow different applications to use OPAQUE without 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 have

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 using 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 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 to 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 streching 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 its 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 both 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. 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.

10.11. 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 6.4.3. 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-09, 8 February 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-voprf-09>.

[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-09, 8 February 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-voprf-09>.

[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-14, 18 February 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-hash-to-curve-14>.

[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>.

[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://scontent.whatsapp.net/v/t39.8562-34/241394876_546674233234181_8907137889500301879_n.pdf/WhatsApp_Security_Encrypted_Backups_Whitepaper.pdf?

ccb=1-5&_nc_sid=2fbf2a&_nc_ohc=Y3PFzd-3LG4AX9AdA8_&_nc_ht=scontent.whatsapp.net&oh=01_AVwwbFhPNWAn-u9VV4wqetjL2T9rX2pDmXwlk0aus4YrKA&oe=620029BC>.

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

Appendix A. Acknowledgments

The OPAQUE protocol and its analysis is 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, provided these mechanisms adhere to the requirements specified in Section 2.4. 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.AuthResponse.server_nonce, KE2.AuthResponse.server
```

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 key shares 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 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-09 of [OPRF].

D.1. Real Test Vectors

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

D.1.1.1. Configuration

```
OPRF: 0001
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: 2ed630416cb2e532804133133e7ee6836c8515752e24bb44d323fef4ea
d34cde967798f2e9784f69d233b1a6da7add58b2c95a57bc213aca920c14553ed2d83
3
credential_identifier: 31323334
password: 436f7272656374486f72736542617474657279537461706c65
envelope_nonce: 36168448f9c5ec75a8cd571370add249e99cb8a8c43f6ef05610a
c6e354642bf
masking_nonce: 13573601f2e727c90ecc19d448cf3145a662e0065f157ba524df0d
3e56ad6236
server_private_key: 51da1f6c3ea07fa00c7cbfdc1fdc70659f1a1092402da749d
938c1a6a570f103
server_public_key: 583f7bccccbc1907ae1506bac950d08266eb3b33ba452b8df7
061a390ffd736e
server_nonce: a88904fe660061c4fac7e452066b8b0f90da7d8d4a19f1cc41fb6fa
5479b467d
client_nonce: 400ceac0fbfb16005928335518be6f930a113c6c0814521262e17ec
c3cdc9f91
server_keyshare: 5cc9fd06a5917ab66a6ef5537a65525a428f768840d81a00d82a
23fc5491b53c
client_keyshare: da25553da9ac142b36332dbd487713ae6712432fb317a6e00b2b
17525bbe6912
server_private_keyshare: eb7216a0ad73af2e84aeeeeb39a9e3549f0817e1732b
5faffc5e0f5abf269e08
client_private_keyshare: a2582d86bf4476a413caa6ee0d3daf7fb6908909036e
1423170d0072aad0d00f
blind_registration: f349de058878adaf864afedb28cf6a6b0f7083a11c34f9543
9c5cb44edb7fd09
blind_login: 146538c20e42b5182766e71c26d4e3a4d1b9c493f7c94bea0bb4f9d6
31181c08
```

D.1.1.3. Intermediate Values

```
client_public_key: eabdf39b4f22d045f80477d5571bae4c40e13377bcb410c6d8
6d86eab281eb15
auth_key: ca657130e970f04883cb6e1d25414c2e6b790521d2589eedbb28f88c2a0
cd1d47a451af444604838acc7ed0eb06cd15265a8f2008f6c00a01471c30d0dce45e0
randomized_pwd: 46dac5eed750784bf22be60303312d53fd6ec61cc19bec55c136c
3629366b1916e8e6a1b09ad9e079da2aa9ce0cde3aea3f28d835b2f67c8bf6e5139e5
e3cc03
envelope: 36168448f9c5ec75a8cd571370add249e99cb8a8c43f6ef05610ac6e354
642bf72e2cdbb55ab0d0cdfcf1cfb9344d3ccbfbcb1b69f975e2e58f25749214ddb7a
11ec03ed7d3f04f05c0c822bd2a4d6cd61c7911035ce117e34f6bc4d8d27ba95
handshake_secret: 71d30b66205d8f3d35415facfa654c45c778ccb1a1522b0cc38
fe88f0eba0e47e4ffbd13cee0bdf0b4cf4b97fb50417bb799d4cfeb58471abc2302dd
264dbe9d
server_mac_key: 6ce3829a4eb1758bbbb9263da5c989b6060851fcae76d10af1a1a
17a627121cc327ac65add4a93d1f3fb289d4b741481dbcbda570a03d156a0c805e287
487db8
client_mac_key: ce946912e6fa49c11068184bdfb0c1d7cb0bb69d2d4ba15dcdc28
bba18021850d296c5c72fa68848ea4c8927d28065c4807fc8163275f0781bffccca18
72ba31
oprf_key: e4af4ba0d3e3d3340848000b77ab12e736fb1662ffbe529ec92163d37ae
26601
```

D.1.1.4. Output Values

```
registration_request: ba1a2238a29a33dea928801e0257bd644f34bcc12f3e6ed
eba3a5015b45d6e33
registration_response: 1c5078bb63f7623d65926a6ef82a4ee7d1b62225d5f8a3
59f603475654f4453b583f7bccccbc1907ae1506bac950d08266eb3b33ba452b8df70
61a390ffd736e
registration_upload: eabdf39b4f22d045f80477d5571bae4c40e13377bcb410c6
d86d86eab281eb15ae21afa59b900243876169f04c46a5833b8168cd87ce9e5a5c04b
ea74eb523bdbeab479e62632bb24f6e4a16fa3ae2132fcd2d4ffcb5cafce1cc8394a8
c3eb3436168448f9c5ec75a8cd571370add249e99cb8a8c43f6ef05610ac6e354642b
f72e2cdbb55ab0d0cdfcf1cfb9344d3ccbfbcb1b69f975e2e58f25749214ddb7a11ec
03ed7d3f04f05c0c822bd2a4d6cd61c7911035ce117e34f6bc4d8d27ba95
KE1: c021ab3bca8c7c7949f7090d2af149523c5029d6c5c45b59997f8c306ccbdf75
400ceac0fbfb16005928335518be6f930a113c6c0814521262e17ecc3cdc9f91da255
53da9ac142b36332dbd487713ae6712432fb317a6e00b2b17525bbe6912
KE2: 1aaae8c352e89557d73dd57152f10983ba4871675d5307c71fc8f8d808103707
13573601f2e727c90ecc19d448cf3145a662e0065f157ba524df0d3e56ad62366311e
350706148302b24efbafa041792c5b79e78c43aa24b44c6e81dde926692d9a9095273
212a862729bad5a9e5258e7f1bf656045dc2842d331d183cc7425c1953a5cd8dd3b4e
83638980d0a85a2c2204eb8d3879421a43450d7eed4bd203b99e16526b8933fc46a62
4a1fec3caf6a5eebe2dfe9689b847716b330098638d1a88904fe660061c4fac7e4520
66b8b0f90da7d8d4a19f1cc41fb6fa5479b467d5cc9fd06a5917ab66a6ef5537a6552
5a428f768840d81a00d82a23fc5491b53ca3e00703736229ba774fb92ba77dc2a2236
e408b99cd8b8e2b0fa2a92ac132100f807b3e44ff1c60d3939ac6b6e8719a4ddf2b37
83f4650fce842ea5c63ccc19
KE3: c5cceaabc721066f2332edcc8cb70c49b8930639f31c6f3ebd8b9e232d35462e
a00bb0bcfa0b703d8b20f06d3428ee7089c299829b737f42a32a26519e33e2bb
export_key: 421f6315d5a2dd7d17eb13c596e69a4455b99209264be00181e99dedf
f76d5a5f55e9cc1340a078f8b307c9dcd95d391193b1ebf648c98378871d087620a0b
a2
session_key: fc56461df9021851b65b29169b0666e3af085c217079db4fe4881073
d9796a2a9add0878ec647f841d2e6d8aecb4d3df8fbc13970a1647b743d29fc5cc892
dab
```

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

D.1.2.1. Configuration

```
OPRF: 0001
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: 4f8c9a5c6576fe6cb958f149fec78f4d8a2875bb40615f6f44ecc2fe30
635396b708ddb7fc10fb73c4e3a9258cd9c3f6f761b2c227853b5def228c850fdbf1e
2
credential_identifier: 31323334
password: 436f7272656374486f72736542617474657279537461706c65
envelope_nonce: 39886c5188df91d7e03ab3f513b828850a017408ffdf4fe072d40
d012f55f6ac
masking_nonce: 983deeb54c9c6337fdd9e120de85343dc7887f00248f1acacc4a83
19d50e29b5
server_private_key: 7f02b3727a18c1d885605e9e09482e22555110f5d2f31a63f
7f8c17f6a985d0b
server_public_key: c0c1fba5133d9b9b5055287de8c8dea9dfbebe10d12ebdf4bd
8ed249886cc67e
server_nonce: c6d04efaee8370c45fa1dfad70201edd140cec8ed6c73b5fcd15c47
7408184fa
client_nonce: e8f5bbbaa7ad3dce15eb299eb2a5b34875ff421b1d63d7a2cfd9096
1b35150da
server_keyshare: 2c3dd46ee4b31250f28ead72fe3d8268ef89d25c9c6318189b9d
04cc729abe51
client_keyshare: 8824e44af3cdc7a29880ff532751b7ccc6a8875ac14e08964942
473de9484f7b
server_private_keyshare: 7f5fbe5a989043f533b588f3c89b21c9dc7991b89ddd
28cde4be79afdb83170f
client_private_keyshare: 9909ef87bfd10d3148a64f98e619251074345b023f19
931b1652c9934a933104
blind_registration: 45075e8ec6743c394e85e3f81ce383ddf78791d163b457fbe
c78c58c0a55050b
blind_login: 6a7637875c6c59544c262523812302dbec1fc73a01abcdbeadfe898e
54dcfe05
```

D.1.2.3. Intermediate Values

```
client_public_key: ee597ed63b18ccc6e5b77ae703e3bd4cfd574650284b21c64e
c16926da7e2851
auth_key: 0fa176059cf53854c38c9841f8c5d5a756b297528729b4a4b5b7894eec9
7b1d9dacb29c337a48cbc276db45452adaeb77e1b4f4990b8d0ef4c03413d3af4a274
randomized_pwd: 15490043fad1f612a0cd72f7571f720f2e5bff138b6c0f9a3f8c7
feb028761ca6bb602028e4228cbc1bcd9b1a8dc3500a7701d9351864595a765ba6a4c
c1b2f3
envelope: 39886c5188df91d7e03ab3f513b828850a017408ffdf4fe072d40d012f5
5f6acef003030d52697e8bfd717c1db8c5ff7a2c0112d7484f1a567c942612c718b5f
010978c806fdaf6892c7ec16d50f80fa12e33daf798e96a71064c72942478bc6
handshake_secret: 540feab4c07eb9263b828c4c20a6138adb46541de1633da67ce
1393a03c1f5e04167a8b0336b45e8aa2a1c3d8c9452f9aed7b0d54545adfcaaa0aaca
35b0a573
server_mac_key: d8775d094511b77e17f4433d6cbc53f4b34b69db34a16c8feeffe
573f70175fb0e16f61ab8ccfddf3599f46ccaa95898b8cefc24c3e73d8a900ea4f0c6
bdbf86
client_mac_key: b7e1601a00b647a559a25a2f30eec8f1105ff51dbebcaa506d943
ec2032c0d85c07673c3784c1008493b8a794a1cd2ef8d4972e9472a665c3abee685f2
03f629
oprf_key: 671cd2624e173c4df9ff81295c41007bf64fe10dec3cf9fd90365040ba6
e290c
```

D.1.2.4. Output Values

```
registration_request: 3e054b6596da6f0da124baa2c095a29c3a6b48571aae699
96f0e079067ac4172
registration_response: c45804cfaa87737d2309164bf7fc0567358c9fef629afd
47a17440d7d43ee71ac0c1fba5133d9b9b5055287de8c8dea9dfbebe10d12ebdf4bd8
ed249886cc67e
registration_upload: ee597ed63b18ccc6e5b77ae703e3bd4cfd574650284b21c6
4ec16926da7e2851dffc6e3207fd7ad90fb974e8f35d17a1f60c0fc9e6cbc49375917
556413a1dac4f9c719e6f63055055276c46d5a308dd4c3f07ca3061176a7ef9200b9c
4a451239886c5188df91d7e03ab3f513b828850a017408ffdf4fe072d40d012f55f6a
cef003030d52697e8bfd717c1db8c5ff7a2c0112d7484f1a567c942612c718b5f0109
78c806fdaf6892c7ec16d50f80fa12e33daf798e96a71064c72942478bc6
KE1: 7002a52fa6c2916c49c1fff952e818e458c7f7799139b243918c97758f463a47
e8f5bbbaa7ad3dce15eb299eb2a5b34875ff421b1d63d7a2cfd90961b35150da8824e
44af3cdc7a29880ff532751b7ccc6a8875ac14e08964942473de9484f7b
KE2: 6e78c98f76160c8cb4df1d0cf3fa038a32b900a1f208901b69b7fb695c28001d
983deeb54c9c6337fdd9e120de85343dc7887f00248f1acacc4a8319d50e29b5f9f61
75f37e8a0718398038dd5159f049f6e7f96be9754907827de30738109889169846cea
a7eee3a6109334a84fd6ec3bb5d462d5b87359f1d909ca5e9b0e7b43000dba44fb4df
9f1629bbe20dd92de2972072ac4ddae968c2dadba8614afa8f0f29cd67ba8e18ced81
49290e67f772f4ff6984a1fd4f163dc2325841eb723bc6d04efaee8370c45fa1dfad7
0201edd140cec8ed6c73b5fcd15c477408184fa2c3dd46ee4b31250f28ead72fe3d82
68ef89d25c9c6318189b9d04cc729abe513ce37681b1db692d3f47e486c31c22e4390
95dc9a4155dca22a5d2e6b8a517f2f7a5d8cb0df01673030683f72a0f62bb0941350c
68d9dc7c449aaa0140bba686
KE3: 4f74844e0c86abbc9189cb03f57e807e2034bdd07f17e67233010a6cacd9ef11
0f153418cafca68e0f8f4f48234d705089f64a7b47bacd0abea3f2a574da5629
export_key: 5c54270bf510936861ea01444d70a7204a6fe1de33ca9613d41e02d30
0d1e6a90b15cabee67a0129629f6b3aac173e1483dfc43457d72fe6df6524a639f89a
1f
session_key: 391db76593cd7f7766b68de34f99b8c0253e86914dbb18177c011d3e
05d611a3a2d0ef7a2b58468c1549444f81a60afbf635d2f6f878fc63061ecc94cfb27
ba8
```

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

D.1.3.1. Configuration

```
OPRF: 0003
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: 380d78c283bf98e26334038293e47865922a3b54d3722d8e9ced1c8729
c42f5a
credential_identifier: 31323334
password: 436f7272656374486f72736542617474657279537461706c65
envelope_nonce: a994c5c01c1855151c467aa331d70f59d9bb63e9afa1e314672a9
c7c6e460d5d
masking_nonce: 848bdf20ed725f0fa3b58e7d8f3eab2a0aace261f61193c7f85709
e9794357fb
server_private_key: 63b448daf85853343c35ec32253326810d0d88f0936c712d3
e901b42cb792f37
server_public_key: 02217c73e50ebf9f8ea0e080a2ecbaf594ca7d5828984e8d5d
455d42ac8531e4f1
server_nonce: 84ff1f2a310fe428d9de5819bf63b3942dbe09f991ca0cf545e33a8
fa17ab9c6
client_nonce: 72721898ef81cc0a76a0b5508f2f7bb817e86f1dd05ca013190a686
02c7af25f
server_keyshare: 0212d788fc5776bd88b7aa01e72ad0d147d8c8a3d9e47d94ca79
10e29f11297b34
client_keyshare: 03a51c7c3d3a69f5217c0f8de4efa242b0cf4ba35cc67c820e57
b69e7a4f53cd69
server_private_keyshare: b3c02a66ef9a72d48cca6c1f9afc1fedea22567b0868
140b482123652ea37c7f
client_private_keyshare: 5d25f85613f5838cd7c6b1697f27bb5e8018e88ecfc5
3891529278c47239f8ff
blind_registration: 7b5d31d5e3ebdb127f92416a3cbcda76e24b2be8d08c79074
a5520292916911b
blind_login: 47401b35db40bdc28cd90b502b3390d3cfea5814c105ca7b460cdf8a
7012c76d
```

D.1.3.3. Intermediate Values

```
client_public_key: 030068ab6e722bb6593382a86becf60ed8290650402470c21d
c90bd0ea9da0f19b
auth_key: 8813ac116d6d46df161221d53ba5ec3bd68baca857c9ee8e3eecb7fc162
08fe0
randomized_pwd: f19aa5337d0ef8c7f728787df75f9abb6a0d06c854960d0646844
c8d68dcc3f2
envelope: a994c5c01c1855151c467aa331d70f59d9bb63e9afa1e314672a9c7c6e4
60d5d6465561f86591334921a1c4402ecbfab336a9945ce398848eff0990b44f4b6a0
handshake_secret: 019dee3711ac01beb7674207a7ed2814f67658d10c52cd71d71
6b87e4204d9a0
server_mac_key: 922a759956ae32addb64e55343677c08538eeddcba9a8b4e861f2
1e9c3849d5c
client_mac_key: ba64564e701ed14b35c6c0d124f6dc98ee1a138c40b4267079363
7419654cd28
oprf_key: 7ad47c7aa69dc3700c91449472d4bd09b15543683560870c7dd21b78398
0f7bf
```

D.1.3.4. Output Values

```
registration_request: 0347ed9a28ccf8baae3b312837378fbd4f994bf601a2522
0bc404102bd1cd9e4a0
registration_response: 027818306df41ac75916146c9d0f06d842e83f232a61da
40b660ee5d670cf77b8202217c73e50ebf9f8ea0e080a2ecbaf594ca7d5828984e8d5
d455d42ac8531e4f1
registration_upload: 030068ab6e722bb6593382a86becf60ed8290650402470c2
1dc90bd0ea9da0f19b4344705e052a843c4cced8ce7c87478555cff1323fc64063301
9423a19455e53a994c5c01c1855151c467aa331d70f59d9bb63e9afa1e314672a9c7c
6e460d5d6465561f86591334921a1c4402ecbfab336a9945ce398848eff0990b44f4b
6a0
KE1: 0226bc3aeccce9c813eaec852599fe76eafe611467a054e738441d4a3b7922aa
ba72721898ef81cc0a76a0b5508f2f7bb817e86f1dd05ca013190a68602c7af25f03a
51c7c3d3a69f5217c0f8de4efa242b0cf4ba35cc67c820e57b69e7a4f53cd69
KE2: 02745cdd4d8336647d5de1715fff6a639b8799e3c6ad951faae59203f4bd97b7
89848bdf20ed725f0fa3b58e7d8f3eab2a0aace261f61193c7f85709e9794357fbd6b
dad9096bf4fa824a2e78e8f36209c9a7fbac3ccd0d56c2b6ea9a0cca3ec7691594eaa
bafbcb4b8b32b65dd8e9fe7e903d9639d67787a2ef7d88d06d257f791eaa59fb7a8b3
d8ec4186c6707b2942dc6ef990e8b958d79c27587f73d371a7cc884ff1f2a310fe428
d9de5819bf63b3942dbe09f991ca0cf545e33a8fa17ab9c60212d788fc5776bd88b7a
a01e72ad0d147d8c8a3d9e47d94ca7910e29f11297b344439d99f9408b8047da08d4d
6ea017e571a26a9a1d80440ed9e4793684dd463d
KE3: a453c142682a3247cea48735543911b07c7498c1c3a7908b8b60c8e1fb90adf1
export_key: feda4a04aa974c1ef9c9d047eb2909ee175851f1c0f5ba37929673f0e
46235e4
session_key: 3585c6e3365b8ad1daa5fd7c3878de2930e6d844bdc8fb13f09debce
b82fde22
```

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

D.1.4.1. Configuration

```
OPRF: 0003
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: b19c2b0ccd8ba22218b6c772e19c4174dc8f436b55b69a4fd701d69873
dacfeb
credential_identifier: 31323334
password: 436f7272656374486f72736542617474657279537461706c65
envelope_nonce: 3f1640a6645455ac63788ee075c245690f9669a9af5699e8b23d6
d1fa9e697ae
masking_nonce: f1029631944beed3594c283c581ac468101aee528cc6b69daac7a9
0de8837d49
server_private_key: 31ae68b478bfc59f5ef534d4e0092e8ef1bfe338aaa4b65c0
563d42fe20626a0
server_public_key: 025cbe5babe2fb2b94ee2527bcdc66fd3a62f4b7e724bdb3ef
4a41cfee527434f3
server_nonce: df174426b40de97e2fabc448b1f4ab66a1a3149df447696d2838463
8319c3819
client_nonce: a2912bab9b6a62cddf7d5e3209a2859e5947586f69259e0708bdfab
794f689ee
server_keyshare: 02ad94bd9a2bb46d8e8ea26ae480a24e2825f58560a20d583a3c
c5078849bdfb8b
client_keyshare: 038744dec9da18441e1ef78ff9b2e5d62c713e56eee7aa326a9b
e577365f919d6c
server_private_keyshare: 711c04899739c0620dc94323d026011ac6def373c257
5400d4018ae26bb2437c
client_private_keyshare: 708e76310767cbe4af18594dfcd436216c2658300d05
18d56d002be476bd06c8
blind_registration: b4526267d942b842e4426e429d05ea84aa6ab34552f0c4a3b
9efdcacbf50daec
blind_login: 12176d4f7ab74fa5fadace604308682dc1bdab92ff91bb1a5fc5bc08
4223fe44
```

D.1.4.3. Intermediate Values

```
client_public_key: 02bef9b16c148b03218e5f8b01a4b52d5cea4a51ddbc76743a
13ba2fa5d1631b33
auth_key: 6dd41a206a9f6a75e02e80e7bb4e696ee2ba68e01e1f96c65e1afc9556a
e1ec8
randomized_pwd: 95847555e29a90ecd2af4e26343be5c65e1c347f1d921be48ba9d
f4e61fad23d
envelope: 3f1640a6645455ac63788ee075c245690f9669a9af5699e8b23d6d1fa9e
697aebc8f31f964da8a9c11a21b359f7522ae50bc02c85362e7dbf051bdf3bf113d98
handshake_secret: 19b7cd9abb29842a0786aec00a574d29f6080cf128840c4867e
2077ba430b621
server_mac_key: 700c2b8c1797a44829e511aef4c66a49ea43aa5ef2847e4993946
798d8d7cbdf
client_mac_key: dfc3af348aa8baf3e95eb904fdfdb11cf8606b961f14dfd5d6881
7658b4d7178
oprf_key: 0a663cda294cc97edada43ff06235a23ac53bf55c439ecc664c01e44738
74e65
```

D.1.4.4. Output Values

```
registration_request: 024cd26832a141c12564716b57b3101d281193c3a2cfaf4
b4d0217b98c69a6e356
registration_response: 034b85dfb783b81cfdcc2255b6ba440479439c17e5f566
690de0dff23ab08bb153025cbe5babe2fb2b94ee2527bcdc66fd3a62f4b7e724bdb3e
f4a41cfee527434f3
registration_upload: 02bef9b16c148b03218e5f8b01a4b52d5cea4a51ddbc7674
3a13ba2fa5d1631b33f8295762da035ff51f6ae4c07fac29e73b900f39a6be5a222d1
43e466282bff13f1640a6645455ac63788ee075c245690f9669a9af5699e8b23d6d1f
a9e697aebc8f31f964da8a9c11a21b359f7522ae50bc02c85362e7dbf051bdf3bf113
d98
KE1: 03ff69ee0b845955eafc817acf721fdecccc94977c4aa0841ec33bf5060375e3
a4a2912bab9b6a62cddf7d5e3209a2859e5947586f69259e0708bdfab794f689ee038
744dec9da18441e1ef78ff9b2e5d62c713e56eee7aa326a9be577365f919d6c
KE2: 026c5dfb5840f9e18b49a2553083bf600b23d73a5352a289223d2a1175d36c9f
b0f1029631944beed3594c283c581ac468101aee528cc6b69daac7a90de8837d49b76
72e17bcd95b17b4d599321921a0ff1d3783624edb14480c018bf39e7a7bab84752a79
7ced451076a5542cf5b0b8433d2b8cd5ceaf9ef7c5f9c1ac13a3e9ff6242362de7710
c5106109ea6a6889388a62a44c932e225c18d649bb44df09b0fffdf174426b40de97e
2fabc448b1f4ab66a1a3149df447696d28384638319c381902ad94bd9a2bb46d8e8ea
26ae480a24e2825f58560a20d583a3cc5078849bdfb8b3d44038b2e740b28519d83f3
8b58cfc221a4421bca2eb74efd05f5c31b34190b
KE3: 6ad96f7b8b167c3482babae788482ddd2ef417eff9ad5617ae49f6c35613b723
export_key: 7284190bd6a6e175cf38846f1374b5f81a481200f774482d89bdb93e0
3674f15
session_key: 4114f3f9ddb7d6f84fc479ab1cbf2e6470540e814b75329661d22fc5
5a8eadff
```

D.2. Fake Test Vectors

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

D.2.1.1. Configuration

```
OPRF: 0001
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: 28885f5b834484836667b5ffb0ecf900c07c55d70e9894af0231f52c54
dd29cccdae5fae5b60c92fa3cd7e6f042429c7c9e946f5351292fa08f4e99e395c30a
9
credential_identifier: 31323334
masking_nonce: 59d26775ae953b9552fdfbf2ab6f469f2f153f9a88aacb7ed434ae
d9fd7ac1ab
client_private_key: 11e4c3344def24f8f55f46f9b72584b36ce931e2a11299afc
6093dff0fbf470f
client_public_key: 020693a36a55d62c38bd2d5f1aaeac2a918e90e1df44a12f48
ce800f7f6e5764
server_private_key: f5002870ce2c5117d0ada53bf11fd7144f72510098b8d477f
ba67a07e5d1640b
server_public_key: 463499017daf13c3915d866656576a8920e15aaf860568d68d
4e1edbc5452802
server_nonce: 7954ebf0e81a893021ee24acc35e1a3f4b5e0366c15771133082ec2
1035ae0ef
server_keyshare: 42a889e6b6d90e31ce452e2ecf4d14ec0c5f5205981d828ae380
90fdcae8bc25
server_private_keyshare: 8d39667010ba488071c889447a547931809e3723b66e
33cf672395a8b48b980a
masking_key: 1bad1c8b6ad879e348e15bd698ee70b2c51d3e89d9c08b00889a1fa8
f3947a48dac9ad994e946f408a2c31250ee34f9d04a7d85661bab11c67048ecfb7a68
c65
KE1: 943a149cf304878367fa2dce5cb30eac23cfd1358e5cc0efdbd4361a9e7bd72d
c26fead2a8b3d5910e25fd29402530b5c7e852585f843f3b939993624b8a7c3b58106
2b0e8e90db4798adbb49581f016034e0855b6d6199aceb56a71c9bd4866
```

D.2.1.3. Output Values

```
KE2: d0b9756ee8cdf900c4120b84b2fcb9c1961b4272fa7d393a33ffa273587f547a
59d26775ae953b9552fdfbf2ab6f469f2f153f9a88aacb7ed434aed9fd7ac1abd4709
ea9ca3ff1bd89374fc6132b7c027593dc7f0ce02da216dde6e90c9a75da99aea9e1e6
c8de0cdd29d54df90584fa83be96c22436b80aed30ea658c79c40cee00730b9d866c2
db24145be6911b530631f5e279ee3fb0b801c3d0c0c3c7de54c745f219fde845a5b9c
415facd45b670dea221104a2a73dab32a0cd951d20a67954ebf0e81a893021ee24acc
35e1a3f4b5e0366c15771133082ec21035ae0ef42a889e6b6d90e31ce452e2ecf4d14
ec0c5f5205981d828ae38090fdcae8bc258ce81724ca613428f82a6f9376f03904f34
dd85794caaaafb55abedfddd35e785e5f7543a8b52964b290869bdd9b786d2c412e50
72784ee403eee9acfa8a5b9e
```

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

D.2.2.1. Configuration

```
OPRF: 0003
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: 94384ca183c8e6f639ab29b5d2a81ef4305df9a67cb33db5ba8082e4f4
bfb830
credential_identifier: 31323334
masking_nonce: 375d7dcbd562a62190cc569ccc809cff9d5aa5e176d48e9646b558
eb41ffab7c
client_private_key: 9ecb5dc678e429e1a01ad6fe5d45301484d12c2a2cf2278fa
c0a0a2cf96eef57
client_public_key: 035951b821e6e1e449933fdba30c7e2e8b6e8f42f4c7a54c80
010a339e72cb2253
server_private_key: 7154525469c4fbae6c9907f4ff26a6386c0a4077f512138e2
203f247d56cbe91
server_public_key: 02acadb2750e036bfcbdb3c5aacad0f55c832631cc8f8e26a6
bc65f7e53525ae79
server_nonce: e0d04374ad9a276620c681abfca7bdb432e63509e5ec96ed2ec5542
f6fc7db23
server_keyshare: 02ffe33c6d938b4d10afeb4ad5ba108ad228317ecab3d6a78a3b
4e2494dc7ec8fb
server_private_keyshare: 034ce43e75362936d67055acf8301fbb910e2afd8769
c4334721fc4ff6bab1d7
masking_key: 1db20e37f4539b2327d37b80c00b98a2cfea9156e5e889b4efa3556e
9f0d24ce
KE1: 0258384d63ae4bbddde6d00d41b0e7174695ff6234563e16fc284aa589c7de93
f9b8bb2700cdd47e339d95404519f2fb3da58c93d84cbb4d51de6757a31919382b026
30e46a94b7f8f66071d24794c37f605055c098afc04d637caf9b1bc714bd15c
```

D.2.2.3. Output Values

```
KE2: 02d0866ee88ab8dc2eb5e39e859e55fa96dca50dc2d280e66dc747f21a14c015
f9375d7dcbd562a62190cc569ccc809cff9d5aa5e176d48e9646b558eb41ffab7c7d6
5027823b4a13e0a19c738eed6ccf3ee141697d93ffe7192a32d1cf803557cc1627fe1
1ccc933dffb662c2d8bbaa97c375287cb172d942c0f0252be71c74f367bd69bc17a7a
7d1e5e25dd1528e81a65f3b8a266c45f0dbf66486c1c65749ff06e0d04374ad9a2766
20c681abfca7bdb432e63509e5ec96ed2ec5542f6fc7db2302ffe33c6d938b4d10afe
b4ad5ba108ad228317ecab3d6a78a3b4e2494dc7ec8fb120aed0e35ab8f67a2a723fe
bf5e5f590d57c08245419972555a59b058240c46
```

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