Workgroup: Network Working Group
Internet-Draft: draft-irtf-cfrg-opaque-07
Published: 25 October 2021
Intended Status: Informational
Expires: 28 April 2022
A   D. Bourdrez   H. Krawczyk   K. Lewi
u   Algorand Foundation   Novi Research
t
h
o
r
s
:
C.A. Wood
Cloudflare

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

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 28 April 2022.

Copyright Notice

Table of Contents

1. Introduction
  1.1. Requirements Notation
  1.2. Notation
2. Cryptographic Dependencies
  2.1. Oblivious Pseudorandom Function
  2.2. Key Derivation Function and Message Authentication Code
  2.3. Hash Functions
  2.4. Key Recovery Method
  2.5. Authenticated Key Exchange (AKE) Protocol
3. Protocol Overview
  3.1. Registration
  3.2. Online Authentication
4. Client Credential Storage and Key Recovery
  4.1. Key Recovery
    4.1.1. Envelope Structure
    4.1.2. Envelope Creation
    4.1.3. Envelope Recovery
5. Offline Registration
  5.1. Registration Messages
  5.2. Registration Functions
    5.2.1. CreateRegistrationRequest
    5.2.2. CreateRegistrationResponse
    5.2.3. FinalizeRequest
  5.3. Finalize Registration
6. Online Authenticated Key Exchange
  6.1. Client Authentication Functions
  6.2. Server Authentication Functions
  6.3. Credential Retrieval
    6.3.1. Credential Retrieval Messages
    6.3.2. Credential Retrieval Functions
  6.4. AKE Protocol
    6.4.1. AKE Messages
    6.4.2. Key Creation
    6.4.3. Key Schedule Functions
    6.4.4. 3DH Client Functions
    6.4.5. 3DH Server Functions
7. Configurations
8. Application Considerations
9. Implementation Considerations
10. Security Considerations
  10.1. Security Analysis
  10.2. Related Protocols
  10.3. Identities
  10.4. Export Key Usage
  10.5. Static Diffie-Hellman Oracles
  10.6. Input Validation

10.7. OPRF Hardening
10.8. Client Enumeration
10.9. Password Salt and Storage Implications
10.10. AKE Private Key Storage
11. IANA Considerations
12. References
12.1. Normative References
12.2. Informative References
Appendix A. Acknowledgments
Appendix B. Alternate Key Recovery Mechanisms
Appendix C. Alternate AKE Instantiations
C.1. HMQV Instantiation Sketch
C.2. SIGMA-I Instantiation Sketch
Appendix D. Test Vectors
D.1. Real Test Vectors
D.1.1. OPAQUE-3DH Real Test Vector 1
D.1.2. OPAQUE-3DH Real Test Vector 2
D.1.3. OPAQUE-3DH Real Test Vector 3
D.1.4. OPAQUE-3DH Real Test Vector 4
D.2. Fake Test Vectors
D.2.1. OPAQUE-3DH Fake Test Vector 1
D.2.2. OPAQUE-3DH Fake Test Vector 2
Authors' Addresses

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 hardening 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 [SIGNAL]. 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 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.

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

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 Authenticate Code (MAC); Section 2.2

*Cryptographic Hash Function; Section 2.3

*Memory-Hard Function (MHF); 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 key derivatio 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 uses the the OPRF defined in [I-D.irtf-cfrg-voprf], Version -08, with the following API and parameters:

*Blind(x): Convert input x into an element of the OPRF group, randomize it by some scalar r, producing M, and output (r, M).

*Evaluate(k, M, info): Evaluate input element M using private key k and public input (or metadata) info, yielding output element Z.

- *Finalize(x, r, Z, info): Finalize the OPRF evaluation using input x, random scalar r, evaluation output Z, and public input (or metadata) info, yielding output y.

- *DeriveKeyPair(seed): Derive a private and public key pair deterministically from a seed.

- *Noe: The size of a serialized OPRF group element.

- *Nok: The size of an OPRF private key.

The public input info is currently set to nil.

Note that we only need the base mode variant (as opposed to the verifiable mode variant) of the OPRF described in [I-D.irtf-cfrg-voprf]. The implementation of DeriveKeyPair based on [I-D.irtf-cfrg-voprf] is below:

```
DeriveKeyPair(seed)

Input:
- seed, pseudo-random byte sequence used as a seed.

Output:
- private_key, a private key.
- public_key, the associated public key.

Steps:
1. private_key = HashToScalar(seed, dst="OPAQUE-DeriveKeyPair")
2. public_key = ScalarBaseMult(private_key)
3. Output (private_key, public_key)
```

HashToScalar(msg, dst) is as specified in [I-D.irtf-cfrg-voprf], Section 2.1.

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 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 Memory Hard Function (MHF) is a slow and expensive cryptographic hash function with the following API:

- *Harden(msg, params): Repeatedly apply a memory-hard function with parameters params to strengthen the input msg 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 10.8, 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 [I-D.irtf-cfrg-voprf], 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. For both stages, client and server agree on a configuration, which fully specifies the cryptographic algorithm dependencies necessary to run the protocol; see Section 7 for details.

3.1. 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.

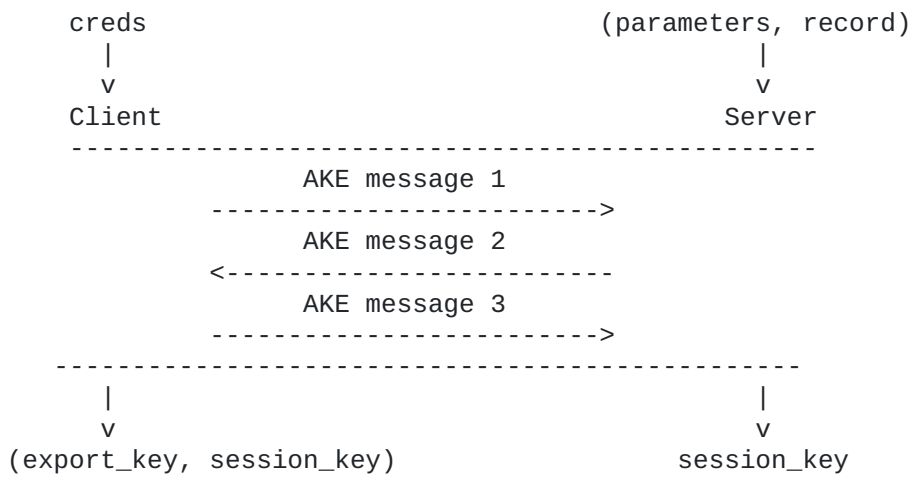
The registration flow is shown below:

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

3.2. Online Authentication

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:



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 10.3 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(server_public_key, client_public_key,
                           server_identity, client_identity)

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

Steps:
1. if server_identity == nil
2.    server_identity = server_public_key
3. if client_identity == nil
4.    client_identity = client_public_key
5. Create CleartextCredentials cleartext_credentials
   with (server_public_key, server_identity, client_identity)
6. Output cleartext_credentials
```

4.1. Key Recovery

This specification defines a key recovery mechanism that uses the hardened 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(randomized_pwd, server_public_key, server_identity, client_identit

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, a key used by the server to encrypt the
  envelope during login.
- export_key, an additional client key.

Steps:
1. envelope_nonce = random(Nn)
2. masking_key = Expand(randomized_pwd, "MaskingKey", Nh)
3. auth_key = Expand(randomized_pwd, concat(envelope_nonce, "AuthKey"),
4. export_key = Expand(randomized_pwd, concat(envelope_nonce, "ExportKey
5. seed = Expand(randomized_pwd, concat(envelope_nonce, "PrivateKey"), N
6. _, client_public_key = DeriveAuthKeyPair(seed)
7. cleartext_creds = CreateCleartextCredentials(server_public_key, clien
8. auth_tag = MAC(auth_key, concat(envelope_nonce, cleartext_creds))
9. Create Envelope envelope with (envelope_nonce, auth_tag)
10. Output (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(randomized_pwd, server_public_key, envelope,
                server_identity, client_identity)

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, when the envelope fails to be recovered

Steps:
1. auth_key = Expand(randomized_pwd, concat(envelope.nonce, "AuthKey"),
2. export_key = Expand(randomized_pwd, concat(envelope.nonce, "ExportKey
3. seed = Expand(randomized_pwd, concat(envelope.nonce, "PrivateKey"), N
4. client_private_key, client_public_key = DeriveAuthKeyPair(seed)
5. cleartext_creds = CreateCleartextCredentials(server_public_key,
                     client_public_key, server_identity, client_identit
6. expected_tag = MAC(auth_key,
                     concat(envelope.nonce, inner_env, cleartext_creds)
7. If !ct_equal(envelope.auth_tag, expected_tag),
     raise KeyRecoveryError
8. Output (client_private_key, export_key)
```

5. Offline Registration

This section describes the registration flow, message encoding, and helper functions. In a setup phase, the client chooses its password, and the server chooses its own pair of private-public AKE keys (server_private_key, server_public_key) for use with the AKE, along with a Nh-byte oprf_seed. 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). These steps can happen offline, i.e., before the registration phase.

Once complete, 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:

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 10.6 for more details. Upon completion, the server stores the client's credentials for later use. Moreover, the client MAY use the output export_key for further application-specific purposes; see Section 10.4.

5.1. Registration Messages

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

data A serialized OPRF group element.

```
struct {
  uint8 data[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 A key used by the server to preserve confidentiality of the envelope during login.

envelope The client's Envelope structure.

5.2. Registration Functions

5.2.1. CreateRegistrationRequest

```
CreateRegistrationRequest(password)

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

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

Steps:
1. (blind, M) = Blind(password)
2. Create RegistrationRequest request with M
3. Output (request, blind)
```

5.2.2. CreateRegistrationResponse

```
CreateRegistrationResponse(request, server_public_key, credential_identi
                           oprf_seed)

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.

Steps:
1. seed = Expand(oprf_seed, concat(credential_identifier, "OprfKey"), Ns
2. (oprf_key, _) = DeriveKeyPair(seed)
3. Z = Evaluate(oprf_key, request.data, nil)
4. Create RegistrationResponse response with (Z, server_public_key)
5. Output response
```

5.2.3. FinalizeRequest

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

```
FinalizeRequest(password, blind, response, server_identity, client_ident

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 encoded client identity.

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

Steps:
1. y = Finalize(password, blind, response.data, nil)
2. randomized_pwd = Extract("", concat(y, Harden(y, params)))
3. (envelope, client_public_key, masking_key, export_key) =
    Store(randomized_pwd, response.server_public_key,
                    server_identity, client_identity)
4. Create RegistrationUpload record with (client_public_key, masking_key
5. Output (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 10.8. 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 10.4. 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:

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 scalar returned by Blind(), of length Nok; 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(password)

State:
- state, a ClientState structure.

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

Output:
- ke1, a KE1 message structure.

Steps:
1. request, blind = CreateCredentialRequest(password)
2. state.blind = blind
3. ake_1 = Start(request)
4. Output KE1(request, ake_1)
```

```
ClientFinish(client_identity, server_identity, ke2)

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.

Steps:
1. (client_private_key, server_public_key, export_key) =
    RecoverCredentials(state.password, state.blind, ke2.CredentialRespon
                       server_identity, client_identity)
2. (ke3, session_key) =
    ClientFinalize(client_identity, client_private_key, server_identity,
                   server_public_key, ke2)
3. Output (ke3, session_key)
```

6.2. Server Authentication Functions

```
ServerInit(server_identity, server_private_key, server_public_key,
           record, credential_identifier, oprf_seed, ke1, client_identit

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.

Steps:
1. response = CreateCredentialResponse(ke1.request, server_public_key, r
    credential_identifier, oprf_seed)
2. ake_2 = Response(server_identity, server_private_key,
    client_identity, record.client_public_key, ke1, response)
3. Output 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 data[Noe];
} CredentialRequest;
```

data A serialized OPRF group element.

```
struct {
  uint8 data[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(password)

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

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

Steps:
1. (blind, M) = Blind(password)
2. Create CredentialRequest request with M
3. Output (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(request, server_public_key, record,
                         credential_identifier, oprf_seed)

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.

Steps:
1. seed = Expand(oprf_seed, concat(credential_identifier, "OprfKey"), No
2. (oprf_key, _) = DeriveKeyPair(seed)
3. Z = Evaluate(oprf_key, request.data, nil)
4. masking_nonce = random(Nn)
5. credential_response_pad = Expand(record.masking_key,
     concat(masking_nonce, "CredentialResponsePad"), Npk + Ne)
6. masked_response = xor(credential_response_pad,
                         concat(server_public_key, record.envelope))
7. Create CredentialResponse response with (Z, masking_nonce, masked_res
8. Output 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(password, blind, response,
                   server_identity, client_identity)

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.

Steps:
1. y = Finalize(password, blind, response.data, nil)
2. randomized_pwd = Extract("", concat(y, Harden(y, params)))
3. masking_key = Expand(randomized_pwd, "MaskingKey", Nh)
4. credential_response_pad = Expand(masking_key,
     concat(response.masking_nonce, "CredentialResponsePad"), Npk + Ne)
5. concat(server_public_key, envelope) = xor(credential_response_pad,
                                             response.masked_response)
6. (client_private_key, export_key) =
    Recover(randomized_pwd, server_public_key, envelope,
                    server_identity, client_identity)
7. Output (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:

- *RecoverPublicKey(private_key): Recover the public key related to the input private_key.
- *DeriveAuthKeyPair(seed): Derive a private and public authentication key pair deterministically from the input seed.
- *GenerateAuthKeyPair(): Return a randomly generated private and public key pair. This can be implemented by generating a random private key sk, then computing pk = RecoverPublicKey(sk).

The implementation of DeriveAuthKeyPair is as follows:

```
DeriveAuthKeyPair(seed)

Input:
- seed, pseudo-random byte sequence used as a seed.

Output:
- private_key, a private key.
- public_key, the associated public key.

Steps:
1. private_key = HashToScalar(seed, dst="OPAQUE-HashToScalar")
2. public_key = ScalarBaseMult(private_key)
3. Output (private_key, public_key)
```

HashToScalar(msg, dst) is as specified in [I-D.irtf-cfrg-voprf], 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(client_identity, ke1, server_identity, ke2)

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.

Steps:
1. 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
2. Output preamble
```

6.4.3.2. Shared Secret Derivation

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

```
TripleDHIKM(sk1, pk1, sk2, pk2, sk3, pk3)

Input:
- skx, scalar to be multiplied with their corresponding pkx.
- pkx, element to be multiplied with their corresponding skx.

Output:
- ikm, input key material.

Steps:
1. dh1 = SerializePublicKey(sk1 * pk1)
2. dh2 = SerializePublicKey(sk2 * pk2)
3. dh3 = SerializePublicKey(sk3 * pk3)
4. Output concat(dh1, dh2, dh3)
```

```
DeriveKeys(ikm, preamble)

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.

Steps:
1. prk = Extract("", ikm)
2. handshake_secret = Derive-Secret(prk, "HandshakeSecret", Hash(preambl
3. session_key = Derive-Secret(prk, "SessionKey", Hash(preamble))
4. Km2 = Derive-Secret(handshake_secret, "ServerMAC", "")
5. Km3 = Derive-Secret(handshake_secret, "ClientMAC", "")
6. Output (Km2, Km3, session_key)
```

6.4.4. 3DH Client Functions

```
Start(credential_request)

Parameters:
- Nn, the nonce length.

State:
- state, a ClientState structure.

Input:
- credential_request, a CredentialRequest structure.

Output:
- ke1, a KE1 structure.

Steps:
1. client_nonce = random(Nn)
2. client_secret, client_keyshare = GenerateAuthKeyPair()
3. Create KE1 ke1 with (credential_request, client_nonce, client_keyshar
4. Populate state with ClientState(client_secret, ke1)
6. Output (ke1, client_secret)
```

```
CLientFinalize(client_identity, client_private_key, server_identity,
               server_public_key, ke2)

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:
- HandshakeError, when the handshake fails

Steps:
1. ikm = TripleDHIKM(state.client_secret, ke2.server_keyshare,
    state.client_secret, server_public_key, client_private_key, ke2.serv
2. preamble = Preamble(client_identity, state.ke1, server_identity, ke2.
3. Km2, Km3, session_key = DeriveKeys(ikm, preamble)
4. expected_server_mac = MAC(Km2, Hash(preamble))
5. If !ct_equal(ke2.server_mac, expected_server_mac),
     raise HandshakeError
6. client_mac = MAC(Km3, Hash(concat(preamble, expected_server_mac))
7. Create KE3 ke3 with client_mac
8. Output (ke3, session_key)
```

6.4.5. 3DH Server Functions

```
Response(server_identity, server_private_key, client_identity,
         client_public_key, ke1, credential_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.

Steps:
1. server_nonce = random(Nn)
2. server_secret, server_keyshare = GenerateAuthKeyPair()
3. Create inner_ke2 ike2 with (ke1.credential_response, server_nonce, se
4. preamble = Preamble(client_identity, ke1, server_identity, ike2)
5. ikm = TripleDHIKM(server_secret, ke1.client_keyshare,
                     server_private_key, ke1.client_keyshare,
                     server_secret, client_public_key)
6. Km2, Km3, session_key = DeriveKeys(ikm, preamble)
7. server_mac = MAC(Km2, Hash(preamble))
8. expected_client_mac = MAC(Km3, Hash(concat(preamble, server_mac))
9. Populate state with ServerState(expected_client_mac, session_key)
10. Create KE2 ke2 with (ike2, server_mac)
11. Output ke2

ServerFinish(ke3)

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:
- HandshakeError, when the handshake fails

Steps:
1. if !ct_equal(ke3.client_mac, state.expected_client_mac):
2.    raise HandshakeError
3. Output state.session_key
```

7. Configurations

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

- The OPRF protocol uses the "base mode" variant of [I-D.irtf-cfrg-voprf] 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 MHF 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 10.8. In settings where this attack is not a concern, servers may choose to not support this functionality.

9. Implementation Considerations

Implementations of OPAQUE should consider addressing the following:

- Clearing secrets out of memory: All private key material and intermediate values, including the outputs of the key exchange phase, should not be retained in memory after deallocation.

- Constant-time operations: 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, to support implementations which provide mitigations against client enumeration attacks.

- Deserialization checks: When parsing messages that have crossed trust boundaries (e.g. a network wire), implementations should

properly handle all error conditions covered in [I-D.irtf-cfrg-voprf] and abort accordingly.

* Additional client-side entropy: OPAQUE supports the ability to incorporate the client identity alongside the password to be 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.

* Server-authenticated channels: Note that online guessing attacks (against any Asymmetric PAKE) 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. This means that additional checks should be considered in a production deployment of OPAQUE: for instance, ensuring that there is a server-authenticated channel over which OPAQUE registration and login is run.

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. Security Analysis

Jarecki et al. [OPAQUE] 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 [I-D.irtf-cfrg-voprf] 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).

10.2. 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.

sWe note that as shown in [OPAQUE], 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.

10.3. Identities

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

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.

10.4. 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.

10.5. Static Diffie-Hellman Oracles

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

10.6. 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.

10.7. OPRF Hardening

Hardening 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 hardening 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.

10.8. 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 for an unregistered client by simulating a fake CredentialResponse as specified in Section 6.3.2.2. 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.

10.9. 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 [OPAQUE].) 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.

10.10. 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-08, 25 October 2021, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-voprf-08>.

[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.

[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.

[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>.

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

[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>.

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

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

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, MHF, EnvelopeMode, Group) tuple, where the Group matches that which is used in the OPRF. These test vectors were generated using draft-06 of [I-D.irtf-cfrg-voprf].

D.1. Real Test Vectors

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

D.1.1.1. Configuration

```
OPRF: 0001
Hash: SHA512
MHF: 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: 5c4f99877d253be5817b4b03f37b6da680b0d5671d1ec5351fa61c5d82
eab28b9de4c4e170f27e433ba377c71c49aa62ad26391ee1cac17011d8a7e9406657c
8
credential_identifier: 31323334
password: 436f7272656374486f72736542617474657279537461706c65
envelope_nonce: 71b8f14b7a1059cdadc414c409064a22cf9e970b0ffc6f1fc6fdd
539c4676775
masking_nonce: 54f9341ca183700f6b6acf28dbfe4a86afad788805de49f2d680ab
86ff39ed7f
server_private_key: 16eb9dc74a3df2033cd738bf2cfb7a3670c569d7749f284b2
b241cb237e7d10f
server_public_key: 18d5035fd0a9c1d6412226df037125901a43f4dff660c0549d
402f672bcc0933
server_nonce: f9c5ec75a8cd571370add249e99cb8a8c43f6ef05610ac6e354642b
f4fedbf69
client_nonce: 804133133e7ee6836c8515752e24bb44d323fef4ead34cde967798f
2e9784f69
server_keyshare: 6e77d4749eb304c4d74be9457c597546bc22aed699225499910f
c913b3e90712
client_keyshare: f67926bd036c5dc4971816b9376e9f64737f361ef8269c18f69f
1ab555e96d4a
server_private_keyshare: f8e3e31543dd6fc86833296726773d51158291ab9afd
666bb55dce83474c1101
client_private_keyshare: 4230d62ea740b13e178185fc517cf2c313e6908c4cd9
fb42154870ff3490c608
blind_registration: c62937d17dc9aa213c9038f84fe8c5bf3d953356db01c4d48
acb7cae48e6a504
blind_login: b5f458822ea11c900ad776e38e29d7be361f75b4d79b55ad74923299
bf8d6503
```

D.1.1.3. Intermediate Values

```
client_public_key: 60c9b59f46e93a2dc8c5dd0dd101fad1838f4c4c026691e9d1
8d3de8f2b3940d
auth_key: 72c837a116b444f86229d432ea48221327339704fd2451704766bb3d42d
10796a2be4083998a78f31f52f3d2fff6ace6b9c2fa9dae1ce64ee36cc867f6cc9e48
randomized_pwd: 024d0bc2c5e95421951227ee87d8c5488e6dc537b2bf014452edb
714bb98f0ef9590b1cca3345f2a1d0afff79967875306e07326b311662d5975b24e82
07594e
envelope: 71b8f14b7a1059cdadc414c409064a22cf9e970b0ffc6f1fc6fdd539c46
767750a343dd3f683692f4ed987ff286a4ece0813a4942e23477920608f261e1ab6f8
727f532c9fd0cde8ec492cb76efdc855da76d0b6ccbe8a4dc0ba2709d63c4517
handshake_secret: 01d81a6ba7c31a2c7c7ff4dab19db55d1f0290905645004bc2b
b8d703f00e486a34c15095df96d6524901eeac1d6c46d94a2ee390dcc5625c6b0baba
fe40e504
server_mac_key: 5ee8f006713979257342d86a1541545b59e4e628b8be4b2f01438
83cb9ce2cd9b6caded5733919739bd889b9426a03ad7c23db8b26ad13d2ad179e56dd
cfe76d
client_mac_key: 5b14da362ce9eaf3492ad315997911d9db3264f0e22e2a4cfb501
f6f336c4e5e3d448f4c1b558101563c55a31e992aab92459c67231ca68de19ab12470
4ca0f9
oprf_key: 3f76113135e6ca7e51ac5bb3e8774eb84709ad36b8907ec8f7bc3537828
71906
```

D.1.1.4. Output Values

```
registration_request: ac7a6330f91d1e5c87365630c7be58641885d59ffe4d3f8
a49c094271993331d
registration_response: 5c7d3c70cf7478ead859bb879b37cce78baef3b9d81e04
f4c790ce25f2830e2e18d5035fd0a9c1d6412226df037125901a43f4dff660c0549d4
02f672bcc0933
registration_upload: 60c9b59f46e93a2dc8c5dd0dd101fad1838f4c4c026691e9
d18d3de8f2b3940d7981498360f8f276df1dfb852a93ec4f4a0189dec5a96363296a6
93fc8a51fb052ae8318dac48be7e3c3cd290f7b8c12b807617b7f9399417deed00158
281ac771b8f14b7a1059cdadc414c409064a22cf9e970b0ffc6f1fc6fdd539c467677
50a343dd3f683692f4ed987ff286a4ece0813a4942e23477920608f261e1ab6f8727f
532c9fd0cde8ec492cb76efdc855da76d0b6ccbe8a4dc0ba2709d63c4517
KE1: e4e7ce5bf96ddb2924faf816774b26a0ec7a6dd9d3a5bced1f4a3675c3cfd14c
804133133e7ee6836c8515752e24bb44d323fef4ead34cde967798f2e9784f69f6792
6bd036c5dc4971816b9376e9f64737f361ef8269c18f69f1ab555e96d4a
KE2: 1af11be29a90322dc16462d0861b1eb617611fe2f05e5e9860c164592d4f7f62
54f9341ca183700f6b6acf28dbfe4a86afad788805de49f2d680ab86ff39ed7f76011
9ed2f12f6ec4983f2c598068057af146fd09133c75b229145b7580d53cac4ba581155
2e6786837a3e03d9f7971df0dad4a04fd6a6d4164101c91137a87f4afde7dae72daf2
620082f46413bbb3071767d549833bcc523acc645b571a66318b0b1f8bf4b23de3542
8373aa1d3a45c1e89eff88f03f9446e5dfc23b6f8394f9c5ec75a8cd571370add249e
99cb8a8c43f6ef05610ac6e354642bf4fedbf696e77d4749eb304c4d74be9457c5975
46bc22aed699225499910fc913b3e907120638f222a1a08460f4e40d0686830d3d608
ce89789489161438bf6809dbbce3a6ddb0ce8702576843b58465d6cedd4e965f3f81b
92992ecec0e2137b66eff0b4
KE3: 1c0c743ff88f1a4ff07350eef61e899ae25d7fb23d555926b218bac4c1963071
5038c56cca247630be8a8e66f3ff18b89c1bc97e1e2192fd7f14f2f60ed084a3
export_key: 8408f92d282c7f4b0f5462e5206bd92937a4d53b0dcdef90afffd015c
5dee44dc4dc5ad35d1681c97e2b66de09203ac359a69f1d45f8c97dbc907589177ccc
24
session_key: 05d03f4143e5866844f7ae921d3b48f3d611e930a6c4be0993a98290
085110c5a27a2e5f92aeed861b90de068a51a952aa75bf97589be7c7104a4c30cc357
506
```

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

D.1.2.1. Configuration

```
OPRF: 0001
Hash: SHA512
MHF: 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: db5c1c16e264b8933d5da56439e7cfed23ab7287b474fe3cdcd58df089
a365a426ea849258d9f4bc13573601f2e727c90ecc19d448cf3145a662e0065f157ba
5
credential_identifier: 31323334
password: 436f7272656374486f72736542617474657279537461706c65
envelope_nonce: d0c7b0f0047682bd87a87e0c3553b9bcdce7e1ae3348570df20bf
2747829b2d2
masking_nonce: 30635396b708ddb7fc10fb73c4e3a9258cd9c3f6f761b2c227853b
5def228c85
server_private_key: eeb2fcc794f98501b16139771720a0713a2750b9e528adfd3
662ad56a7e19b04
server_public_key: 8aa90cb321a38759fc253c444f317782962ca18d33101eab2c
8cda04405a181f
server_nonce: 3fa57f7ef652185f89114109f5a61cc8c9216fdd7398246bb7a0c20
e2fbca2d8
client_nonce: a6bcd29b5aecc3507fc1f8f7631af3d2f5105155222e48099e5e608
5d8c1187a
server_keyshare: ae070cdffe5bb4b1c373e71be8e7d8f356ee5de37881533f1039
7bcd84d35445
client_keyshare: 642e7eecf19b804a62817486663d6c6c239396f709b663a4350c
da67d025687a
server_private_keyshare: 0974010a8528b813f5b33ae0d791df88516c8839c152
b030697637878b2d8b0a
client_private_keyshare: 03b52f066898929f4aca48014b2b97365205ce691ee3
444b0a7cecec3c7efb01
blind_registration: a66ffb41ccf1194a8d7dda900f8b6b0652e4c7fac4610066f
e0489a804d3bb05
blind_login: e6f161ac189e6873a19a54efca4baa0719e801e336d929d35ca28b5b
4f60560e
```

D.1.2.3. Intermediate Values

```
client_public_key: 3036af4744effe59eb7ee5db0ebcb653bd4a1c7ad0c56c78af
1288f1e8538d1c
auth_key: 8820ff275662bf91d4ebcca74c9b90913eb3ee8151047926ad754da823e
98800db56a79c68b44d76ad906d26ed8b9e25d8ea862cfc6c2f0da86c623f6a24961a
randomized_pwd: 16decdbba6912903b7ae38de7040a79ebc59c9fbbac04add8a710
0ff8aedbb9530c4e664bd08b2689a607e99923e80563a8379ddfdb37801718ed043fb
7bca07
envelope: d0c7b0f0047682bd87a87e0c3553b9bcdce7e1ae3348570df20bf274782
9b2d24117867ef8aa569ed6fa8ad1b3749b0df472d431ce92da7775e44623d6c36f7e
9396d16ac58060704e9d42b37f09642ed7ee49008b4b81dc65d282ddcec0ab97
handshake_secret: eef528fc4e46c387ef2bd68c06eb135ea18d743bcb632233594
b213eb4eefaa4010b0f00785a204c77456f9007d8f7645fdc0c6795f486150a53fe17
e6608179
server_mac_key: ff6eeffbe0e88a966bdb39fa4fc933a488c0b802f913a7a6950d1
c2eda0fc6acce6fdf7b060aa0be02efddb5a127e4dd893d3666e9b54d6fd6c85f52c9
13138e
client_mac_key: 4babec778469dfed06cd89c6673b8511f337e27ce565a2fd68cb6
2a04df035fabc2097ba33018a4cd858dc399a0563950d7ce8ffe26c223328023e4c70
a16c8e
oprf_key: 531b0c7b0a3f90060c28d3d96ef5fecf56e25b8e4bf71c14bc770804c3f
b4507
```

D.1.2.4. Output Values

```
registration_request: d81b76a8a78b8b0758f7ceffaa5c3cb4ac76c0517759ad8
077ed87857e585f79
registration_response: a48835aa277db6d7d501addbd431100a548867e3f1ee6f
d6ae4aacd817a66e4c8aa90cb321a38759fc253c444f317782962ca18d33101eab2c8
cda04405a181f
registration_upload: 3036af4744effe59eb7ee5db0ebcb653bd4a1c7ad0c56c78
af1288f1e8538d1cedbc931daab2331b192808768f149499a04c6dffa4eae66a6e0d3
399547c8b9e9a743a3cd20f08ce07adf84b27c9ca879d730bcc41823cbd60411fbde6
c7faf2d0c7b0f0047682bd87a87e0c3553b9bcdce7e1ae3348570df20bf2747829b2d
24117867ef8aa569ed6fa8ad1b3749b0df472d431ce92da7775e44623d6c36f7e9396
d16ac58060704e9d42b37f09642ed7ee49008b4b81dc65d282ddcec0ab97
KE1: 8a32b2985d824b0e42b7d3c5091774acd64386f8a762678422f0b5cbabeda12b
a6bcd29b5aecc3507fc1f8f7631af3d2f5105155222e48099e5e6085d8c1187a642e7
eecf19b804a62817486663d6c6c239396f709b663a4350cda67d025687a
KE2: da642966461f20090d1e8d6b1f63ea70dc94fc6e0ea0bad46d011e906cc03c03
30635396b708ddb7fc10fb73c4e3a9258cd9c3f6f761b2c227853b5def228c8594543
768891810f779604eb9e07dcd37635def358e2f4531f464a4e0b3726c150d7872785c
9b6a22f00fe3527d9e938d4b503047484723585ee390925ab9d97e30f0860caef1243
0459d8ca24e5ff1a2029c363ed00f2f3cd09ead304f217290d8915183c2667959d420
175bfca3bbec3d603844ca0d5b5892888f0de19dc3b83fa57f7ef652185f89114109f
5a61cc8c9216fdd7398246bb7a0c20e2fbca2d8ae070cdffe5bb4b1c373e71be8e7d8
f356ee5de37881533f10397bcd84d354454f08b6c37449cf70cac0babb85d5302dc59
a0ae16b2e54865642b8bb985f48444d49ad89a6a0707dd46c2d53b8b73dff46ac7176
a6167f39818f605e3c39d22c
KE3: b9487ca4b1308ce593d765739992e19c10d63c47f4f2d3eb4bfd0ffa101b6959
114b4f6051305652e0f48ad219a696f3f12fad685f8d6e371dddc10fda2ec87e
export_key: 258d525e93a07c17dd9e41afc4fbfe316152afad02c54a6d3d201fd77
487903143ca2ef27718a1e48b2ade5dc614b027b8a46fd334b701df5d385aaef2b1bd
16
session_key: 021c0c3f15940f3e898f2925949aa8bc262248fae7b9ed7d33a2900e
866548ed24760c2244a2c14bfc196a00ffd66ebf54839850b101bc5e617c37ccad45a
68a
```

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

D.1.3.1. Configuration

```
OPRF: 0003
Hash: SHA256
MHF: 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: 77bfc065218c9a5593c952161b93193f025b3474102519e6984fa64831
0dd1bf
credential_identifier: 31323334
password: 436f7272656374486f72736542617474657279537461706c65
envelope_nonce: 2527e48c983deeb54c9c6337fdd9e120de85343dc7887f00248f1
acacc4a8319
masking_nonce: cb792f3657240ce5296dd5633e7333531009c11ee6ab46b6111f15
6d96a160b2
server_private_key: 87ef09986545b295e8f5bbbaa7ad3dce15eb299eb2a5b3487
5ff421b1d63d7a3
server_public_key: 025b95a6add1f2f3d038811b5ad3494bed73b1e2500d8dadec
592d88406e25c2f2
server_nonce: 8018e88ecfc53891529278c47239f8fe6f1be88972721898ef81cc0
a76a0b550
client_nonce: 967fcded96ed46986e60fcbdf985232639f537377ca3fcf07ad4899
56b2e9019
server_keyshare: 0242bc29993976185dacf6be815cbfa923aac80fad8b7f020c9d
4f18e0b6867a17
client_keyshare: 03358b4eae039953116889466bfddeb40168e39ed83809fd5f0d
5f2de9c5234398
server_private_keyshare: b1c0063e442238bdd89cd62b4c3ad31f016b68085d25
f85613f5838cd7c6b16a
client_private_keyshare: 10256ab078bc1edbaf79bee4cd28dd9db89179dcc921
9bc8f388b533f5439099
blind_registration: d50e29b581d716c3c05c4a0d6110b510cb5c9959bee817fde
b1eabd7ccd74fee
blind_login: 503d8495c6d04efaee8370c45fa1dfad70201edd140cec8ed6c73b5f
cd15c478
```

D.1.3.3. Intermediate Values

```
client_public_key: 030f9b896400f6efd57c69a41b05ffedc456f041cb54a2ab56
8f5595c586070708
auth_key: 4e01ca008eb4f84b8cee1b84b3abfaeb4f2c7fb41d2c8ad0f4fe89d74e6
f0fc5
randomized_pwd: c741d0a042e653ee4ccf24648aee4e3b4c500cc28feb3a72eea0f
24f69006693
envelope: 2527e48c983deeb54c9c6337fdd9e120de85343dc7887f00248f1acacc4
a83190f798f947d61d060cb102e5eeb9bd698bec5d1e1b6788860ec7c2d2e590121b0
handshake_secret: 78bedd3ee950e1795ddeca4e0d4f4267a971ace52e6f876d9b2
c8a349ec2be2a
server_mac_key: c8e62b9aee6ae6e2199db70f16631a302e9269f27d5f6ef954572
f8ca05f8d01
client_mac_key: 31e3581fcfbb7d6b10b5cf78399fb844ab7afe42cf94f8b72178a
1618711bb25
oprf_key: d153d662a1e7dd4383837aa7125685d2be6f8041472ecbfd610e46952a6
a24f1
```

D.1.3.4. Output Values

```
registration_request: 037aa042e317344246ebb94c38fe9989e01f7265413ade1
f7ffaa706a81f58cf19
registration_response: 03c0b3e621cadf1a56aa48305e3101efedb6248157708c
7ba70af396fa62d29bf7025b95a6add1f2f3d038811b5ad3494bed73b1e2500d8dade
c592d88406e25c2f2
registration_upload: 030f9b896400f6efd57c69a41b05ffedc456f041cb54a2ab
568f5595c5860707085e76cb3c849637cfd386d9cc762050a476a58da7c24b8a39084
4689d8d6482bd2527e48c983deeb54c9c6337fdd9e120de85343dc7887f00248f1aca
cc4a83190f798f947d61d060cb102e5eeb9bd698bec5d1e1b6788860ec7c2d2e59012
1b0
KE1: 0320fee3e9c08dfd30d00ce524cee6595d9bd7387629efa0cb9eba1ba82ec465
13967fcded96ed46986e60fcbdf985232639f537377ca3fcf07ad489956b2e9019033
58b4eae039953116889466bfddeb40168e39ed83809fd5f0d5f2de9c5234398
KE2: 03f629c1a3a5a3dc83af63c52d3bd58bbd78d5054caee7731381e967a7c381fa
20cb792f3657240ce5296dd5633e7333531009c11ee6ab46b6111f156d96a160b22c1
7f819537c821604229b8c07798c56f14b5104729a1336f153510f58ea921758f8a486
13ec4ee3e5675dc8be14776c0bb6458bf0d3f76dd24af8b43b49c8fbfcb5229c0bbe3
a37c440bdca76ce404b215ceb8842e95e81138416e161ea02c2648018e88ecfc53891
529278c47239f8fe6f1be88972721898ef81cc0a76a0b5500242bc29993976185dacf
6be815cbfa923aac80fad8b7f020c9d4f18e0b6867a1764573de6cf3b1b7737e7e56a
181fe0ec8754940adce33c4712bd35e7e9e08e7c
KE3: d9108b70e4ff4955911162ed1cec6df65c880aad120bbf10fd7f32eea71b1a04
export_key: 086cd26a64f469f2d22ab0b5f0c524b10321c4019018b004d0f8383c0
24059be
session_key: 36d1125dbf5ea45568e586645841efb6c5f53d357cdffb79edf1bb8d
b0b843a9
```

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

D.1.4.1. Configuration

```
OPRF: 0003
Hash: SHA256
MHF: 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: 482123652ea37c7e4a0f9f1984ff1f2a310fe428d9de5819bf63b3942d
be09f9
credential_identifier: 31323334
password: 436f7272656374486f72736542617474657279537461706c65
envelope_nonce: 75c245690f9669a9af5699e8b23d6d1fa9e697aeb4526267d942b
842e4426e42
masking_nonce: 5947586f69259e0708bdfab794f689eec14c7deb7edde68c816451
56cf278f21
server_private_key: c728ebf47b1c65594d77dab871872dba848bdf20ed725f0fa
3b58e7d8f3eab2b
server_public_key: 029a2c6097fbbcf3457fe3ff7d4ef8e89dab585a67dfed0905
c9f104d909138bae
server_nonce: 581ac468101aee528cc6b69daac7a90de8837d49708e76310767cbe
4af18594d
client_nonce: 46498f95ec7986f0602019b3fbb646db87a2fdbc12176d4f7ab74fa
5fadace60
server_keyshare: 022aa8746ab4329d591296652d44f6dfb04470103311bacd7ad5
1060ef5abac41b
client_keyshare: 02a9f857ad3eabe09047049e8b8cee72feea2acb7fc487777c0b
22d3add6a0e0c0
server_private_keyshare: 48a5baa24274d5acc5e007a44f2147549ac8dd675564
2638f1029631944beed4
client_private_keyshare: 161e3aaa50f50e33344022969d17d9cf4c88b7a9eec4
c36bf64de079abb6dc7b
blind_registration: 9280e203ef27d9ef0d1d189bb3c02a66ef9a72d48cca6c1f9
afc1fedea22567c
blind_login: 4308682dc1bdab92ff91bb1a5fc5bc084223fe4369beddca3f1640a6
645455ad
```

D.1.4.3. Intermediate Values

```
client_public_key: 03ce71710d0d366e44e4a7e92cb111fc41353d4244cac1ce4d
8a622acaab9effc6
auth_key: b894fa35f63413029fcc70e80a0d1b59d1c90c3c255bfb11cf7b58fb136
d2aee
randomized_pwd: 0588794becaf8f5fee7921cb467e4ce8b3c048e7b42d815ed306d
ef278c231d3
envelope: 75c245690f9669a9af5699e8b23d6d1fa9e697aeb4526267d942b842e44
26e42cb65c94629db9811649cd4f3ff92e5d2c67f7486203ea5e471f2655f363f9f19
handshake_secret: 8a2547abef351fc1f94fb19a886c2e5ca16aba3b2bfe0b4a8cc
086dd47b62c08
server_mac_key: fa7c99e15ca1036738b9b48799515be78e471a2d06c3c3920d6a3
703d11c0360
client_mac_key: d480fde6de5e91a08179d9780bf6db0d1b959ae2fa394c09acdc6
07b993410c2
oprf_key: f14e1fc34ba1218bfd3f7373f036889bf4f35a8fbc9e8c9c07ccf2d2388
79d9c
```

D.1.4.4. Output Values

```
registration_request: 02baa002c856f4b0d49542dcb1391f240f836178702f835
819fd221bcf9b6e9eec
registration_response: 03864f4590c09b4c4155f0cbb731c5aab554ab1bc930c3
28e7a58bd6227933d54f029a2c6097fbbcf3457fe3ff7d4ef8e89dab585a67dfed090
5c9f104d909138bae
registration_upload: 03ce71710d0d366e44e4a7e92cb111fc41353d4244cac1ce
4d8a622acaab9effc66c5d2844e32ed930c56080fa523c15ec6d85f7db1bbd02c4692
14b31e27f6c5775c245690f9669a9af5699e8b23d6d1fa9e697aeb4526267d942b842
e4426e42cb65c94629db9811649cd4f3ff92e5d2c67f7486203ea5e471f2655f363f9
f19
KE1: 038469dadcb23317fa577317079c82bad1e20be41c783cd0ecad6bef3de1b16b
1446498f95ec7986f0602019b3fbb646db87a2fdbc12176d4f7ab74fa5fadace6002a
9f857ad3eabe09047049e8b8cee72feea2acb7fc487777c0b22d3add6a0e0c0
KE2: 036297ebd0b53dabaae6377cb1c3ba1bdd942a67a5ce019b363f26cd11ae3707
ac5947586f69259e0708bdfab794f689eec14c7deb7edde68c81645156cf278f21308
4ce22d007db399a17af864b5ea826f4086f3d477ce236cacf7867de174692940b103b
367ccb8b5aee6ef352079bf95c5961442cf400432de4d904815d1a8a20f64f3e8447b
82c27f4c9b798769db0fb5ab8d29ea0ee54c1e371105388a7ae7c581ac468101aee52
8cc6b69daac7a90de8837d49708e76310767cbe4af18594d022aa8746ab4329d59129
6652d44f6dfb04470103311bacd7ad51060ef5abac41bfa6b8e732462d3de6bdb3ef3
edcf4595b478a6704d578fde4eaf922e1c1e8504
KE3: cd11b70f1ed59d101ec20a73745d3d654c3772236ed2c365a730ef8ee51da6d2
export_key: 8e1eb57bcde2d58d805b16fa045811679c68b0ec2817b9ac61786786a
9032837
session_key: b1f3da97388d6171719c3e2281e88da75b68d6945189f460db841cc6
92f7e164
```

D.2. Fake Test Vectors

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

D.2.1.1. Configuration

```
OPRF: 0001
Hash: SHA512
MHF: 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: 98ee70b2c51d3e89d9c08b00889a1fa8f3947a48dac9ad994e946f408a
2c31250ee34f9d04a7d85661bab11c67048ecfb7a68c657a3df87cff3d09c6af9912a
1
credential_identifier: 31323334
masking_nonce: 7cb33db5ba8082e4f4bfb830e8e3f525b0ddcb70469b34224758d7
25ce53ac76
client_private_key: 21c97ffc56be5d93f86441023b7c8a4b629399933b3f845f5
852e8c716a60408
client_public_key: 5cc46fdc0337a684e126f8663deacc67872a7daffc75312a1d
6377783935f932
server_private_key: 030c7634ae24b9c6d4e2c07176719e7a246850b8e019f1c71
a23af6bdb847b0b
server_public_key: 1ac449e9cdd633788069cca1aaea36ea359d7c2d493b254e5f
fe8d64212dcc59
server_nonce: cae1f4fee4ee4ba509fda550ea0421a85762305b1db20e37f4539b2
327d37b80
server_keyshare: 5e5c0ac2904c7d9bf38f99e0050594e484b4d8ded8038ef6e0c1
41a985fa6b35
server_private_keyshare: a4abffe3bef8082b78323ea4507fbb0ce8105ca62b38
1919a35767deaa699709
masking_key: 077adba76f768fd0979f8dc006ca297e7954ebf0e81a893021ee24ac
c35e1a3f4b5e0366c15771133082ec21035ae0ef0d8bcd0e59d26775ae953b9552fdf
bf2
KE1: 88303c5318f93d39bb8afde6df62593869ba4eec265b980e3843c013401e6c5a
8837b6c0709160251cbebe0d55e4423554c45da7a8952367cf336eb623379e80dae2f
1e0cd79b733131d499fb9e77efe0f235d73c1f920bdc5816259ad3a7429
```

D.2.1.3. Output Values

```
KE2: 8a003351892efcf8615128a241e2bf091433fab5a080d7512b156f53e8602a20
7cb33db5ba8082e4f4bfb830e8e3f525b0ddcb70469b34224758d725ce53ac76094c0
aa800d9a0884392e4efbc0479e3cb84a38c9ead879f1ff755ad762c06812b9858f82c
9722acc61b8eb1d156bc994839bf9ed8a760615258d23e0f94fa2cffadc655ed0d6ff
6914066427366019d4e6989b65d13e38e8edc5ae6f82aa1b6a46bfe6ca0256c64d0cf
db50a3eb7676e1d212e155e152e3bbc9d1fae3c679aacae1f4fee4ee4ba509fda550e
a0421a85762305b1db20e37f4539b2327d37b805e5c0ac2904c7d9bf38f99e0050594
e484b4d8ded8038ef6e0c141a985fa6b35ad4627117ba6a8cfe2a7c9d100800a62c84
aacc83ed786d722921ee7037abf71b4af7381cdc3d40c4d9e4fc9f6dc2bb2fc15c8e3
aa9eca8a83332841dda4524f
```

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

D.2.2.1. Configuration

```
OPRF: 0003
Hash: SHA256
MHF: 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: f7664fae89be455ee3350b04a85eab390b2dc63256fbd311d8de944b45
b859e6
credential_identifier: 31323334
masking_nonce: 21cd364318a92b2afbfccea5d80d337f07defe40d92673a52f3844
058f5d949a
client_private_key: 41ffab7c86e2b0916361fb6a69f9a097e3ef2f83f8fd5f95c
c79432eabf3e020
client_public_key: 0251bc2a7e0cb7c043eec5ee7d1b769b69f85b0fa19d1ae907
5416e93fa01689de
server_private_key: 61764783412278e6ce3c6c66f1995a2a30b5824be6a6d31ca
d35a578ec3d9353
server_public_key: 03727dd31712275905b1a3cca3bbb33bc71034a1d0c3801be0
20541933dd497f18
server_nonce: 2b772c1eb569cc2b57741bf3be630e377c8245b11d0b6ad1fe1d606
490c27208
server_keyshare: 02a59205c836a2ab86e19dbd9a417818052179e9a5c99221e2d1
d8a780dfe4734d
server_private_keyshare: e8c25741b201c2ba00abe390e5a3933a75efdb71b50e
1e0087cc7235f6f9448a
masking_key: 5bb4d884375d7dcbd562a62190cc569ccc809cff9d5aa5e176d48e96
46b558eb
KE1: 0320dd7cff999858fb63be5d11db9c3fafbacbedb775303324d8859bfb31f6dc
dba91c9485d74c9010185f462ce1eec52f588a8e392f36915849b6bfcb6bd5b904037
6a35db8f7e582569dba2e573c4af1462f91c59a9bdee253ed13f60108746252
```

D.2.2.3. Output Values

```
KE2: 03cac8c1654bba83a122227e503e5e5d1a094def98d6835be289421cdc08d549
a121cd364318a92b2afbfccea5d80d337f07defe40d92673a52f3844058f5d949a604
39294e7567fc29643e0d5c8799d0dffbbfc8609558b982012fa90aef2ce52b1ffdd8f
96bda49f5306ae346cd745812d3a953ff94712e4ed0acc67c99b432860e337fe3234b
ba88415ac55368b938106cca4049b5c13496fe167d3a092bd990e2b772c1eb569cc2b
57741bf3be630e377c8245b11d0b6ad1fe1d606490c2720802a59205c836a2ab86e19
dbd9a417818052179e9a5c99221e2d1d8a780dfe4734d04a0d4911decc97ece7f24af
58f767090bf16677af9468a4026efbab99877399
```

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

Email: caw@heapingbits.net