Workgroup: Network Working Group
Internet-Draft: draft-irtf-cfrg-opaque-00
Published: 28 September 2020
Intended Status: Informational
Expires: 1 April 2021
Authors: H. Krawczyk, Algorand Foundation; K. Lewi, Facebook; 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, along with several instantiations in different authenticated key exchange protocols.

Discussion Venues

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

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

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Password authentication is the prevalent form of authentication in the web and in many other applications. In the most common implementation, a user authenticates to a server by sending its user ID and password to the server over a TLS connection. This makes the password vulnerable to server mishandling, including accidentally logging the password or storing it in cleartext in a database. Server compromise resulting in access to these plaintext passwords is not an uncommon security incident, even among security-conscious companies. Moreover, plaintext password authentication over TLS is also vulnerable to TLS failures, including many forms of PKI attacks, certificate mishandling, termination outside the security perimeter, visibility to middle boxes, 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 user/password 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. Namely, it should only be open to inevitable attacks, such as online impersonation attempts with guessed user passwords and offline dictionary attacks upon the compromise of a server and leakage of its password file. In the latter case, the attacker learns a mapping of a user'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.

Despite the existence of multiple designs for (PKI-free) aPAKE protocols, none of these protocols are secure against pre-computation attacks. 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 salt at all or, if they do, they transmit the salt from server to user in the clear, hence losing the secrecy of the salt and its defense against pre-computation. Furthermore, transmitting the salt may require additional protocol messages.

This document describes OPAQUE, a PKI-free secure aPAKE that is secure against pre-computation attacks and capable of using a secret salt. OPAQUE provides forward secrecy (essential for protecting past communications in case of password leakage) and the ability to hide the password from the server - even during password registration. Furthermore, OPAQUE enjoys good performance and an array of additional features including the ability to increase the difficulty of offline dictionary attacks via iterated hashing or other hardening schemes, and offloading these operations to the client (that also helps against online guessing attacks); extensibility of the protocol to support storage and retrieval of user's secrets solely based on a password; and being amenable to a multi-server distributed implementation where offline dictionary attacks are not possible without breaking into a threshold of servers (such a distributed solution requires no change or awareness on the client side relative to a single-server implementation).

OPAQUE is defined and proven as the composition of two functionalities: an Oblivious PRF (OPRF) 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 6 for requirements of the OPRF and AKE protocols.) This document specifies OPAQUE instantiations based on a variety of AKE protocols, including HMQV [HMQV], 3DH [SIGNAL] and SIGMA [SIGMA]. In general, the modularity of OPAQUE's design makes it easy to integrate with additional AKE protocols, e.g., IKEv2, and with future ones such as those based on post-quantum techniques.

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

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 terms are used throughout this document to describe the operations, roles, and behaviors of OPAQUE:

* Client (U): Entity which has knowledge of a password and wishes to authenticate.

* Server (S): Entity which authenticates clients using passwords.

* (skX, pkX): An AKE key pair used in role X; skX is the private key and pkX is the public key. For example, (skU, pkU) refers to U's private and public key.

* kX: An OPRF private key used in role X. For example, kU refers to U's private OPRF key.

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

* concat(x0, ..., xN): Concatenation of byte strings. concat(0x01, 0x0203, 0x040506) = 0x010203040506.

* random(n): Generate a random byte string of length n bytes.

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

* ct_equal(a, b): Return true if a is equal to b, and false otherwise. This function is 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.

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

2. Cryptographic Protocol and Algorithm Dependencies

OPAQUE relies on the following protocols and primitives:

* Oblivious Pseudorandom Function (OPRF, [I-D.irtf-cfrg-voprf]):

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

  - Evaluate(k, M): Evaluate input M using private key k, yielding output Z.

  - Unblind(r, Z): Remove randomizer r from Z, yielding output N.

- -Finalize(x, N, info): Compute the OPRF output using input x, N, and domain separation tag info.
- -Serialize(x): Encode the OPRF group element x as a fixed-length byte string enc. The size of enc is determined by the underlying OPRF group.
- -Deserialize(enc): Decode a byte string enc into an OPRF group element x, or produce an error if enc is an invalid encoding. This is the inverse of Serialize, i.e., x = Deserialize(Serialize(x)).

*Cryptographic hash function:

- -Hash(m): Compute the cryptographic hash of input message "m".
- -Nh: The output size of the Hash function.

*Memory Hard Function (MHF):

- -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.

We also assume the existence of a function KeyGen from [I-D.irtf-cfrg-voprf], which generates an OPRF private and public key. OPAQUE only requires an OPRF private key. We write (kU, _) = KeyGen() to denote use of this function for generating secret key kU (and discarding the corresponding public key).

3. Core Protocol

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 encrypted credentials on the server. In the second stage, a client obtains those credentials, unlocks them using the user's password and subsequently uses them as input to an authenticated key exchange (AKE) protocol.

Both registration and authenticated key exchange stages require running an OPRF protocol. The latter stage additionally requires running a mutually-authenticated key-exchange protocol (AKE) using credentials recovered after the OPRF protocol completes. (The key-exchange protocol MUST satisfy forward secrecy and the KCI requirement discussed in Section 6.)

We first define the core OPAQUE protocol based on a generic OPRF, hash, and MHF function. Section 4 describes specific instantiations of OPAQUE using various AKE protocols, including: HMQV, 3DH, and

SIGMA-I. [I-D.sullivan-tls-opaque] discusses integration with TLS 1.3 [RFC8446].

3.1. Protocol messages

The OPAQUE protocol runs the OPRF protocol in two stages: registration and authenticated key exchange. A client and server exchange protocol messages in executing these stages. This section specifies the structure of these protocol messages using TLS notation (see [RFC8446], Section 3).

```
enum {
    registration_request(1),
    registration_response(2),
    registration_upload(3),
    credential_request(4),
    credential_response(5),
    (255)
} ProtocolMessageType;

struct {
    ProtocolMessageType msg_type;    /* protocol message type */
    uint24 length;                   /* remaining bytes in message */
    select (ProtocolMessage.msg_type) {
        case registration_request: RegistrationRequest;
        case registration_response: RegistrationResponse;
        case registration_upload: RegistrationUpload;
        case credential_request: CredentialRequest;
        case credential_response: CredentialResponse;
    };
} ProtocolMessage;
```

OPAQUE makes use of an additional structure Credentials to store user (client) credentials. A Credentials structure consists of secret and cleartext CredentialExtension values. Each CredentialExtension indicates the type of extension and carries the raw bytes. This specification includes extensions for OPAQUE, including:

* skU: The encoded user private key for the AKE protocol.

* pkU: The encoded user public key for the AKE protocol.

* pkS: The encoded server public key for the AKE protocol.

* idU: The user identity. This is an application-specific value, e.g., an e-mail address or normal account name.

* idS: The server identity. This is typically a domain name, e.g., example.com. See Section 3.5 for information about this identity.

Each public and private key value is an opaque byte string, specific to the AKE protocol in which OPAQUE is instantiated. For example, if used as raw public keys for TLS 1.3 [RFC8446], they may be RSA or ECDSA keys as per [RFC7250].

The full Credentials encoding is as follows.

```
enum {
  skU(1),
  pkU(2),
  pkS(3),
  idU(4),
  idS(5),
  (255)
} CredentialType;

struct {
  CredentialType type;
  CredentialData data<0..2^16-1>;
} CredentialExtension;

struct {
  CredentialExtension secret_credentials<1..2^16-1>;
  CredentialExtension cleartext_credentials<0..2^16-1>;
} Credentials;
```

secret_credentials OPAQUE credentials which require secrecy and authentication.

cleartext_credentials OPAQUE credentials which require authentication but not secrecy.

Applications MUST include skU in secret_credentials and pkS in either cleartext_credentials or secret_credentials. All other CredentialExtension values are optional. It is RECOMMENDED that applications include pkS and idS in cleartext_credentials, as this allows servers to not store redundant encryptions of these values for each user in case the server uses the same values for multiple users.

Additionally, we assume helper functions SerializeExtensions and DeserializeExtensions which translate a list of CredentialExtension structures to and from a unique byte string encoding.

OPAQUE uses an Envelope structure to encapsulate an encrypted Credentials structure. It is encoded as follows.

```
struct {
  opaque nonce[32];
  opaque ct<1..2^16-1>;
  opaque auth_data<0..2^16-1>;
} InnerEnvelope;

struct {
  InnerEnvelope contents;
  opaque auth_tag[Nh];
} Envelope;
```

nonce A unique 32-byte nonce used to protect this Envelope.

ct Encoding of encrypted and authenticated credential extensions list.

auth_data Encoding of an authenticated credential extensions list.

auth_tag Authentication tag protecting the contents of the envelope.

3.2. Offline registration stage

Registration is executed between a user U (running on a client machine) and a server S. It is assumed the server can identify the user and the client can authenticate the server during this registration phase. This is the only part in OPAQUE that requires an authenticated channel, either physical, out-of-band, PKI-based, etc. This section describes the registration flow, message encoding, and helper functions. Moreover, U has a key pair (skU, pkU) for an AKE protocol which is suitable for use with OPAQUE; See Section 3.3. (skU, pkU) may be randomly generated for the account or provided by the calling client. Clients MUST NOT use the same key pair (skU, pkU) for two different accounts.

To begin, U chooses password pwdU, and S chooses its own pair of private-public keys skS and pkS for use with the AKE. S can use the same pair of keys with multiple users. These steps can happen offline, i.e., before the registration phase. Once complete, the registration process proceeds as follows:

```
 Client (idU, pwdU, skU, pkU)                 Server (skS, pkS)
  -----------------------------------------------------------------
   request, metadata = CreateRegistrationRequest(idU, pwdU)

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

            (response, kU) = CreateRegistrationResponse(request, pkS)

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

 record = FinalizeRequest(idU, pwdU, skU, metadata, request, response)

                                  record
                              ------------------->

                                             StoreUserRecord(record)
```

Both client and server MUST validate the other party's public key before use. See Section 6.3 for more details.

3.2.1. Registration messages

```
struct {
    opaque id<0..2^16-1>;
    opaque data<1..2^16-1>;
} RegistrationRequest;
```

id An opaque string carrying the client account information, if available.

data An encoded element in the OPRF group. See [I-D.irtf-cfrg-voprf] for a description of this encoding.

```
struct {
    opaque data_blind<1..2^16-1>;
} RequestMetadata;
```

data_blind An encoded OPRF scalar element. See [I-D.irtf-cfrg-voprf] for a description of this encoding.

```
struct {
    opaque data<0..2^16-1>;
    opaque pkS<0..2^16-1>;
    CredentialType secret_types<1..255>;
    CredentialType cleartext_types<0..255>;
} RegistrationResponse;
```

data

An encoded element in the OPRF group. See [I-D.irtf-cfrg-voprf] for a description of this encoding.

pkS An encoded public key that will be used for the online authenticated key exchange stage.

```
struct {
    Envelope envelope;
    opaque pkU<0..2^16-1>;
} RegistrationUpload;
```

envelope An authenticated encoding of a Credentials structure with additional application-specific data.

pkU An encoded public key, matching the public key contained within the encrypted envelope.

3.2.2. Registration functions

3.2.2.1. CreateRegistrationRequest

```
CreateRegistrationRequest(idU, pwdU)

Input:
- idU, an opaque byte string containing the user's identity
- pwdU, an opaque byte string containing the user's password

Output:
- request, a RegistrationRequest structure
- metadata, a RequestMetadata structure

Steps:
1. (r, M) = Blind(pwdU)
2. data = Serialize(M)
3. Create RegistrationRequest request with (idU, data)
4. Create RequestMetadata metadata with Serialize(r)
5. Output (request, metadata)
```

3.2.2.2. CreateRegistrationResponse

```
CreateRegistrationResponse(request, pkS)

Parameters:
- secret_credentials_list, a list of CredentialType values clients shoul
 in the secret_credentials list of their Credentials structure
- cleartext_credentials_list, a list of CredentialType values clients sh
 in the cleartext_credentials list of their Credentials structure

Input:
- request, a RegistrationRequest structure
- pkS, the server's public key

Output:
- response, a RegistrationResponse structure
- kU, Per-user OPRF key

Steps:
1. (kU, _) = KeyGen()
2. M = Deserialize(request.data)
3. Z = Evaluate(kU, M)
4. data = Z.encode()
5. Create RegistrationResponse response with
     (data, pkS, secret_credentials_list, cleartext_credentials_list)
6. Output (response, kU)
```

3.2.2.3. FinalizeRequest

```
FinalizeRequest(idU, pwdU, skU, metadata, request, response)

Parameters:
- params, the MHF parameters established out of band

Input:
- idU, an opaque byte string containing the user's identity
- pwdU, an opaque byte string containing the user's password
- skU, the user's private key
- metadata, a RequestMetadata structure
- request, a RegistrationRequest structure
- response, a RegistrationResponse structure

Output:
- upload, a RegistrationUpload structure
- export_key, an additional key

Steps:
1. Z = Deserialize(response.data)
2. N = Unblind(input.data_blind, Z)
3. y = Finalize(pwdU, N, "OPAQUE00")
4. rwdU = HKDF-Extract("rwdU", Harden(y, params))
5. Create secret_credentials with CredentialExtensions matching that
   contained in response.secret_credentials_list
6. Create cleartext_credentials with CredentialExtensions matching that
   contained in response.cleartext_credentials_list
7. pt = SerializeExtensions(secret_credentials)
8. nonce = random(32)
9. pseudorandom_pad = HKDF-Expand(rwdU, concat(nonce, "Pad"), len(pt))
10. auth_key = HKDF-Expand(rwdU, concat(nonce, "AuthKey"), Nh)
11. export_key = HKDF-Expand(rwdU, concat(nonce, "ExportKey"), Nh)
12. ct = xor(pt, pseudorandom_pad)
13. auth_data = SerializeExtensions(cleartext_credentials)
14. Create InnerEnvelope contents with (nonce, ct, auth_data)
15. t = HMAC(auth_key, contents)
16. Create Envelope envU with (contents, t)
17. Create RegistrationUpload upload with envelope value (envU, pkU)
18. Output (upload, export_key)
```

[[RFC editor: please change "OPAQUE00" to the correct RFC identifier before publication.]]

[[https://github.com/cfrg/draft-irtf-cfrg-opaque/issues/58: Should the nonce size be a parameter?]]

The inputs to HKDF-Extract and HKDF-Expand are as specified in [RFC5869]. The underlying hash function is that which is associated with the OPAQUE configuration (see Section 5).

OPAQUE security requires authentication for all CredentialExtension values, and secrecy for skU. If an application additionally requires secrecy of pkS, this value SHOULD be included in the Credentials.secret_credentials list (step 5), and MUST NOT be included in the Credentials.cleartext_credentials list. Applications may optionally include pkU, idU, or idS in the Credentials.cleartext_credentials structure, or in Credentials.secret_credentials if secrecy of these values is desired. Servers MUST specify how clients encode extensions in the Credentials structure as part of this registration phase.

The server identity idS comes from context. For example, if registering with a server within the context of a TLS connection, the identity might be the server domain name. See Section 3.5.

See Section 3.4 for details about the output export_key usage.

3.2.2.4. StoreUserRecord

The StoreUserRecord function stores the tuple (envU, pkS, skS, pkU, kU), where envU and pkU are obtained from the input RegistrationUpload message in a record associated with the user's account idU. If skS and pkS are used for multiple users, the server can store these values separately and omit them from the user's record.

3.3. Online authenticated key exchange stage

After registration, the user (through a client machine) and server run the authenticated key exchange stage of the OPAQUE protocol. This stage is composed of a concurrent OPRF and key exchange flow. The key exchange protocol is authenticated using the client and server private keys established during the offline phase; see Section 3.2. The type of keys MUST be suitable for the key exchange protocol. For example, if the key exchange protocol is 3DH, as described in Section 4.2, then the private and public keys must be Diffie-Hellman keys. At the end, the client proves the user's knowledge of the password, and both client and server agree on a mutually authenticated shared secret key.

This section describes the message flow, encoding, and helper functions used in this stage.

```
 Client (idU, pwdU)                             Server (skS, pkS)
  -----------------------------------------------------------------
   request, metadata = CreateCredentialRequest(idU, pwdU)

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

         (response, pkU) = CreateCredentialResponse(request, pkS)

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

  creds, export_key = RecoverCredentials(pwdU, metadata, request, respon

                                 (AKE with creds)
                              <================>
```

The protocol messages below do not include the AKE protocol. Instead, OPAQUE assumes the client and server run the AKE using the credentials recovered from the OPRF protocol.

Note also that the authenticated key exchange stage can run the OPRF and AKE protocols concurrently with interleaved and combined messages (while preserving the internal ordering of messages in each protocol). In all cases, the client needs to obtain envU and rwdU (i.e., complete the OPRF protocol) before it can use its own private key skU and the server's public key pkS in the AKE. See Section 4 for examples of this integration.

3.3.1. Authenticated key exchange messages

```
struct {
    opaque id<0..2^16-1>;
    opaque data<1..2^16-1>;
} CredentialRequest;
```

id An opaque string carrying the client account information, if available. If absent, the server is assumed to have some way of ascertaining the client account information out of band.

data An encoded element in the OPRF group. See [I-D.irtf-cfrg-voprf] for a description of this encoding.

```
struct {
    opaque data<1..2^16-1>;
    opaque envelope<1..2^16-1>;
    opaque pkS<0..2^16-1>;
} CredentialResponse;
```

data

An encoded element in the OPRF group. See [I-D.irtf-cfrg-voprf] for a description of this encoding.

envelope An authenticated encoding of a Credentials structure.

pkS An encoded public key that will be used for the online authenticated key exchange stage. This field is optional.

3.3.2. Authenticated key exchange functions

3.3.2.1. CreateCredentialRequest(idU, pwdU)

```
CreateCredentialRequest(idU, pwdU)

Input:
- idU, an opaque byte string containing the user's identity
- pwdU, an opaque byte string containing the user's password

Output:
- request, an CredentialRequest structure
- metadata, a RequestMetadata structure

Steps:
1. (r, M) = Blind(pwdU)
2. data = Serialize(M)
3. Create CredentialRequest request with (idU, data)
4. Create RequestMetadata metadata with Serialize(r)
5. Output (request, metadata)
```

3.3.2.2. CreateCredentialResponse(request, pkS)

```
CreateCredentialResponse(request, pkS)

Input:
- request, an CredentialRequest structure
- pkS, public key of the server

Output:
- response, a CredentialResponse structure
- pkU, public key of the user

Steps:
1. (kU, envU, pkU) = LookupUserRecord(request.id)
2. M = Deserialize(request.data)
3. Z = Evaluate(kU, M)
4. data = Z.encode()
5. Create CredentialResponse response with (data, envU, pkS)
6. Output (response, pkU)
```

3.3.2.3. RecoverCredentials(pwdU, metadata, request, response)

```
RecoverCredentials(pwdU, metadata, request, response)

Parameters:
- params, the MHF parameters established out of band

Input:
- pwdU, an opaque byte string containing the user's password
- metadata, a RequestMetadata structure
- request, a RegistrationRequest structure
- response, a RegistrationResponse structure

Output:
- C, a Credentials structure
- export_key, an additional key

Steps:
1. Z = Deserialize(response.data)
2. N = Unblind(input.data_blind, Z)
3. y = Finalize(pwdU, N, "OPAQUE00")
4. contents = response.envelope.contents
5. nonce = contents.nonce
6. ct = contents.ct
7. rwdU = HKDF-Extract("rwdU", Harden(y, params))
8. pseudorandom_pad = HKDF-Expand(rwdU, concat(nonce, "Pad"), len(ct))
9. auth_key = HKDF-Expand(rwdU, concat(nonce, "AuthKey"), Nh)
10. export_key = HKDF-Expand(rwdU, concat(nonce, "ExportKey"), Nh)
11. expected_tag = HMAC(auth_key, contents)
12. If !ct_equal(response.envelope.auth_tag, expected_tag), raise Decryp
13. pt = xor(ct, pseudorandom_pad)
14. secret_credentials = DeserializeExtensions(pt)
15. cleartext_credentials = DeserializeExtensions(auth_data)
16. Create Credentials creds with (secret_credentials, cleartext_credent
17. Output creds, export_key
```

[[RFC editor: please change "OPAQUE00" to the correct RFC identifier before publication.]]

3.4. Export Key

In addition to Credentials, OPAQUE outputs an export_key that may be used for additional application-specific purposes. For example, one might expand the use of OPAQUE with a credential-retrieval functionality that is separate from the contents of the Credentials structure.

The exporter_key MUST NOT be used in any way before the HMAC value in the envelope is validated.

3.5. AKE Execution and Party Identities

The AKE protocol is run as part of the online authenticated key exchange flow described above. The AKE MUST authenticate the OPAQUE transcript, which consists of the encoded request and response messages exchanged during the OPRF computation and credential fetch flow.

Also, authenticated key-exchange 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 idU and idS. 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 idU to the identity the server uses to fetch envU (hence determined during password registration) and to tie idS to the server identity used by the client to initiate an offline password registration or online authenticated key exchange session. idS and idU can also be part of envU or be tied to the parties' public keys. In principle, it is possible that identities 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 user must always be those defined at time of password registration.

4. Authenticated Key Exchange Protocol Instantiations

This section describes several instantiations of OPAQUE using different AKE protocols, all of which satisfy the forward secrecy and KCI properties discussed in Section 6. For the sake of concreteness it only includes AKE protocols consisting of three messages, denoted KE1, KE2, KE3, where KE1 and KE2 include key exchange shares (DH values) sent by 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).

As shown in [OPAQUE], OPAQUE cannot use less than three messages so the 3-message instantiations presented here are optimal in terms of number of messages. On the other hand, there is no impediment of using OPAQUE with protocols with more than 3 messages as in the case of IKEv2 (or the underlying SIGMA-R protocol [SIGMA]).

The generic outline of OPAQUE with a 3-message AKE protocol is as follows:

- C to S: credential_request, KE1

*S to C: credential_response, KE2

*C to S: KE3

Key derivation and other details of the protocol are specified by the KE scheme. We note that by the results in [OPAQUE], KE2 and KE3 should authenticate credential_request and credential_response, respectively, for binding between the underlying OPRF protocol messages and the KE session.

Next, we present three instantiations of OPAQUE - with HMQV, 3DH and SIGMA-I. [I-D.sullivan-tls-opaque] discusses integration with TLS 1.3 [RFC8446]. Note that these instantiations transmit idU in cleartext. Applications that require idU privacy should encrypt this appropriately. Mechanisms for doing so are outside the scope of this document, though may be addressed elsewhere, such as in [I-D.sullivan-tls-opaque].

OPAQUE may 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 6. This document does not specify such an instantiation. Note that such an instantiation is not quantum safe unless the OPRF and data encryption schemes are quantum safe. However, an instantiation where both AKE and data encryption are quantum safe, but the OPRF is not, would still ensure data security against future quantum attacks since breaking the OPRF does not retroactively affect the security of data transferred over a quantum-safe secure channel.

4.1. Key Schedule Utility Functions

The key derivation procedures for HMQV, 3DH, and SIGMA-I instantiations all make use of the functions below, re-purposed from TLS 1.3 [RFC8446].

```
HKDF-Expand-Label(Secret, Label, Context, Length) =
  HKDF-Expand(Secret, HkdfLabel, Length)
```

Where HkdfLabel is specified as:

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

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

HKDF uses Hash as its underlying hash function, which is the same as that which is indicated by the OPAQUE instantiation.

4.2. Instantiation with HMQV and 3DH

The integration of OPAQUE with HMQV [HMQV] leads to the most efficient instantiation of OPAQUE in terms of exponentiations count. Performance is close to optimal due to the low cost of authentication in HMQV: Just 1/6 of an exponentiation for each party over the cost of a regular DH exchange. However, HMQV is encumbered by an IBM patent, hence we also present OPAQUE with 3DH which only differs in the key derivation function at the cost of two additional exponentiations (and less resilience to the compromise of ephemeral exponents). We note that 3DH serves as a basis for the key-exchange protocol of [SIGNAL]. Importantly, many other protocols follow a similar format with differences mainly in the key derivation function. This includes the Noise family of protocols. Extensions also apply to KEM-based AKE protocols as in many post-quantum candidates.

4.2.1. HMQV and 3DH protocol messages

HMQV and 3DH are both implemented using a suitable cyclic group of prime order p. All operations in the key derivation steps in Section 4.2.2.1 and Section 4.2.2.2 are performed in this group and represented here using multiplicative notation.

OPAQUE with HMQV and OPAQUE with 3DH comprises:

- KE1 = credential_request, nonceU, info1*, idU*, epkU
- KE2 = credential_response, nonceS, info2*, epkS, Einfo2*, MAC(Km2; transcript2),
- KE3 = info3*, Einfo3*, MAC(Km3; transcript3)}

where:

- '*' denotes optional elements;
- The private and public keys of the parties in these examples are Diffie-Hellman keys, namely, pkU=g^skU and pkS=g^skS.
- credential_request and credential_response denote the online OPAQUE protocol messages (defined in Section 3.3) which carry the client and server OPRF values, respectively, as well as the envelope.
- nonceU, nonceS are fresh random nonces chosen by client and server, respectively;

* info1, info2, info3 denote optional application-specific information sent in the clear (e.g., they can include parameter negotiation, parameters for a hardening function, etc.);

* Einfo2, Einfo3 denotes optional application-specific information sent encrypted under keys Ke2, Ke3 defined below;

* idU is the user's identity used by the server to construct credential_response, which contains the server's OPRF response and envU. idU can be omitted from message KE1 if the information is available to the server in some other way;

* idS, the server's identity, is not shown explicitly, it can be part of an info field (encrypted or not), part of envU, or can be known from other context (see Section 3.5); it is used crucially for key derivation (see below);

* epkU, epkS are Diffie-Hellman ephemeral public keys chosen by user and server, respectively, which MUST be validated to be in the correct group (see Section 6.3);

* transcript2 includes the concatenation of the values credential_request, nonceU, info1*, idU*, epkU, credential_response, nonceS, info2*, epkS, Einfo2*;

* transcript3 includes the concatenation of all elements in transcript2 followed by info3*, Einfo3*;

Notes:

* The explicit concatenation of elements under transcript2 and transcript3 can be replaced with hashed values of these elements, or their combinations, using a collision-resistant hash (e.g., as in the transcript-hash of TLS 1.3 [RFC8446]).

* The inclusion of the values credential_request and credential_response under transcript2 is needed for binding the underlying OPRF execution to that of the AKE session. On the other hand, including envU in transcript2 is not mandatory for security, though done as part of including credential_response.

4.2.2. HMQV and 3DH key derivation

The above protocol requires MAC keys Km2, Km3, and optional encryption keys Ke2, Ke3, as well as generating a session key SK which is the AKE output for protecting subsequent traffic (or for generating further key material). Key derivation uses HKDF [RFC5869] with a combination of the parties static and ephemeral private-public key pairs and the parties' identities idU, idS. See Section 3.5 for more information about these identities.

HMQV and 3DH use the following key schedule for computing Km2, Km3, Ke2, Ke3, and SK:

```
  HKDF-Extract(salt=0, IKM)
      |
      +--> Derive-Secret(., "handshake secret", info) = handshake_secret
      |
      +--> Derive-Secret(., "session secret", info) = SK
```

From handshake_secret, Km2, Km3, Ke2, and Ke3 are computed as follows:

```
Km2 = HKDF-Expand-Label(handshake_secret, "client mac", "", Hash.length)
Km3 = HKDF-Expand-Label(handshake_secret, "server mac", "", Hash.length)
Ke2 = HKDF-Expand-Label(handshake_secret, "client enc", "", key_length)
Ke3 = HKDF-Expand-Label(handshake_secret, "server enc", "", key_length)
```

key_length is the length of the key required for the AKE handshake encryption algorithm.

Values IKM and info are defined for each instantiation in the following sections.

4.2.2.1. HMQV key derivation

The HKDF input parameter info is computed as follows:

```
info = "HMQV keys" || I2OSP(len(nonceU), 2) || nonceU
                   || I2OSP(len(nonceS), 2) || nonceS
                   || I2OSP(len(idU), 2) || idU
                   || I2OSP(len(idS), 2) || idS
```

The input parameter IKM is Khmqv, where Khmqv is computed by the client as follows:

1. u' = (eskU + u*skU) mod p
2. Khmqv = (epkS * pkS^s)^u'

Hash is the same hash function used in the main OPAQUE protocol for key derivation. Its output length must be at least the length of the group order p.

Likewise, servers compute Khmqv as follows:

1. s' = (eskS + s*skS) mod p
2. Khmqv = (epkU * pkU^u)^s'

In both cases, u is computed as follows:

```
hashInput = I2OSP(len(epkU), 2) || epkU ||
            I2OSP(len(info), 2) || info ||
            I2OSP(len("client"), 2) || "client"
u = Hash(hashInput) mod (len(p)-1)
```

Likewise, s is computed as follows:

```
hashInput = I2OSP(len(epkS), 2) || epkS ||
            I2OSP(len(info), 2) || info ||
            I2OSP(len("server"), 2) || "server"
s = Hash(hashInput) mod (len(p)-1)
```

4.2.2.2. 3DH key derivation

The HKDF input parameter info is computed as follows:

```
info = "3DH keys" || I2OSP(len(nonceU), 2) || nonceU
                  || I2OSP(len(nonceS), 2) || nonceS
                  || I2OSP(len(idU), 2) || idU
                  || I2OSP(len(idS), 2) || idS
```

The input parameter IKM is K3dh, where K3dh is the concatenation of three DH values computed by the client as follows:

```
K3dh = epkS^eskU || pkS^eskU || epkS^skU
```

Likewise, K3dh is computed by the server as follows:

```
K3dh = epkU^eskS || epkU^skS || pkU^eskS
```

4.3. Instantiation with SIGMA-I

We show the integration of OPAQUE with the 3-message SIGMA-I protocol [SIGMA]. This is an example of a signature-based protocol and also serves as a basis for integration of OPAQUE with TLS 1.3 as specified in [I-D.sullivan-tls-opaque]. This specification can be extended to the 4-message SIGMA-R protocol as used in IKEv2.

4.3.1. SIGMA protocol messages

OPAQUE with SIGMA-I comprises:

- KE1 = credential_request, nonceU, info1*, idU*, epkU

- KE2 = credential_response, nonceS, info2*, epkS*, *Einfo2*, Sign(skS; transcript2-), MAC(Km2; idS),

- KE3 = info3*, Einfo3*, Sign(skU; transcript3-), MAC(Km3; idU)}

See explanation of fields in Section 4.2.1. In addition, for the signed material, transcript2- is defined similarly to transcript2, however if transcript2 includes information that identifies the user, such information can be eliminated in transcript2- (this is advised if signing user's identification information by the server is deemed to have adverse privacy consequences). Similarly, transcript3- is defined as transcript3 with server identification information removed if so desired.

4.3.2. SIGMA key derivation

The key schedule for computing Km2, Km3, Ke2, Ke3, and SK is the same as specified in Section 4.2.2. The HKDF input parameter info is computed as follows:

```
info = "SIGMA-I keys" || I2OSP(len(nonceU), 2) || nonceU
                      || I2OSP(len(nonceS), 2) || nonceS
                      || I2OSP(len(idU), 2) || idU
                      || I2OSP(len(idS), 2) || idS
```

The input parameter IKM is Ksigma, where Ksigma is computed by clients as epkS^eskU and by servers as epkU^eskS.

5. Configurations

An OPAQUE configuration is a tuple (OPRF, Hash, MHF, AKE). The OPAQUE OPRF protocol is drawn from [I-D.irtf-cfrg-voprf]. The following OPRF ciphersuites supports are supported:

*OPRF(curve25519, SHA-512)

*OPRF(curve448, SHA-512)

*OPRF(P-256, SHA-512)

*OPRF(P-384, SHA-512)

*OPRF(P-521, SHA-512)

The OPAQUE hash function is that which is associated with the OPRF variant. For the variants specified here, only SHA-512 is supported.

[[https://github.com/cfrg/draft-irtf-cfrg-opaque/issues/59: Consider SHA-256 for the Curve25519 OPRF suite - SHA-512 is excessive]]

The OPAQUE MHFs include Argon2 [I-D.irtf-cfrg-argon2], scrypt [RFC7914], and PBKDF2 [RFC2898] with suitable parameter choices. These may be constant values or set at the time of password registration and stored at the server. In the latter case, the server communicates these parameters to the client during login.

The OPAQUE AKE protocols are those which are specified in Section 4. Future specifications (such as [I-D.sullivan-tls-opaque]) MAY introduce other AKE instantiations.

[[https://github.com/cfrg/draft-irtf-cfrg-opaque/issues/60: Should we have a registry for configurations?]]

6. Security Considerations

OPAQUE is defined and proven as the composition of two functionalities: An Oblivious PRF (OPRF) and an authenticated key-exchange (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 user 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 TLS. The latter aspect is of prime importance as the use of OPAQUE with TLS constitutes a major security improvement relative to the standard password-over-TLS practice. At the same time, the combination with TLS builds OPAQUE as a fully functional secure communications protocol and can help provide privacy to account information sent by the user to the server prior to authentication.

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 user 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).

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 of 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].

Very few aPAKE protocols have been proven formally, and those proven were analyzed in a weak security model that allows for pre-computation attacks (e.g., [GMR06]). This is not just a formal issue: these protocols are actually vulnerable to such attacks. This includes protocols that have recent analyses in the UC model such as AuCPace [AuCPace] and SPAKE2+ [SPAKE2plus]. We 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.

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).

6.1. Configuration Choice

Best practices regarding implementation of cryptographic schemes apply to OPAQUE. Particular care needs to be given to the implementation of the OPRF regarding testing group membership and avoiding timing and other side channel leakage in the hash-to-curve mapping. Drafts [I-D.irtf-cfrg-hash-to-curve] and [I-D.irtf-cfrg-voprf] have detailed instantiation and implementation guidance.

6.2. 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 the case that q-1 or q+1, where q is the order of the group G, has a t-bit divisor, they show an attack that calls the OPRF on 2^t chosen inputs and reduces security by t/2 bits, i.e., it can find the OPRF key in time 2^{q/2-t/2} and 2^{q/2-t/2} memory. For typical curves, the attack requires an infeasible number of calls and/or results in insignificant security loss (*). Moreover, in the OPAQUE application, these attacks are completely impractical as the number of calls to the function translates to an equal number of failed authentication attempts by a *single* user. For example, one would need a billion impersonation attempts to reduce security by 15 bits and a trillion to reduce it by 20 bits - and most curves will not even allow for such attacks in the first place (note that this theoretical loss of security is with respect to computing discrete logarithms, not in reducing the password strength).

(*) Some examples (courtesy of Dan Brown): For P-384, 2^90 calls reduce security from 192 to 147 bits; for NIST P-256 the options are 6-bit reduction with 2153 OPRF calls, about 14 bit reduction with 187 million calls and 20 bits with a trillion calls. For Curve25519, attacks are completely infeasible (require over 2^100 calls) but its twist form allows an attack with 25759 calls that reduces security by 7 bits and one with 117223 calls that reduces security by 8.4 bits.

6.3. 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. For X25519 and X448, validation is as described in [RFC7748]. In particular, where applicable, endpoints MUST check whether the Diffie-Hellman shared secret is the all-zero value and abort if so.

6.4. User authentication versus Authenticated Key Exchange

OPAQUE provides PAKE (password-based authenticated key exchange) functionality in the client-server setting. While in the case of user identification, wherein the focus is often on authentication, we stress that the key exchange element is essential. Indeed, in most cases, user authentication enforces some policy, and the key exchange step is essential for binding this enforcement to the authentication step. Skipping the key exchange part is analogous to carefully checking a visitor's credential at the door and then leaving the door open for others to enter freely.

6.5. OPRF Hardening

Hardening the output of the OPRF greatly increases the cost of an offline attack upon the compromise of the password file at the server. Applications SHOULD select parameters that balance cost and complexity.

6.6. User enumeration

User enumeration refers to attacks where the attacker tries to learn whether a given user identity is registered with a server. Preventing such attack requires the server to act with unknown user identities in a way that is indistinguishable from its behavior with existing users. Here we suggest a way to implement such defense, namely, a way for simulating the values beta and envU for non-existing users. Note that if the same pair of user identity idU and value alpha is received twice by the server, the response needs to be the same in both cases (since this would be the case for real users). For protection against this attack, one would apply the encryption function in the construction of envU to all the key material in envU, namely, cleartext_credentials will be empty. The server S will have two keys MK, MK' for a PRF f (this refers to a regular PRF such as HMAC or CMAC). Upon receiving a pair of user identity idU and value alpha for a non-existing user idU, S computes kU=f(MK; idU) and kU'=f(MK'; idU) and responds with values beta=alpha^kU and envU, where the latter is computed as follows. rwdU is set to kU' and AEenv is set to the all-zero string (of the length of a regular envU plaintext). Care needs to be taken to avoid side channel leakage (e.g., timing) from helping differentiate these operations from a regular server response. The above requires changes to the server-side implementation but not to the protocol itself or the client side.

There is one form of leakage that the above allows and whose prevention would require a change in OPAQUE. Note that an attacker that tests a idU (and same alpha) twice and receives different responses can conclude that either the user registered with the service between these two activations or that the user was registered before but changed its password in between the activations (assuming the server changes kU at the time of a password change). In any case, this indicates that idU is a registered user at the time of the second activation. To conceal this information, S can implement the derivation of kU as kU=f(MK; idU) also for registered users. Hiding changes in envU, however, requires a change in the protocol. Instead of sending envU as is, S would send an encryption of envU under a key that the user derives from the OPRF result (similarly to rwdU) and that S stores during password registration. During the authenticated key exchange stage, the user will derive this key from the OPRF result, will use it to decrypt envU, and continue with the regular protocol. If S uses a randomized encryption, the encrypted envU will look each time as a fresh random string, hence S can simulate the encrypted envU also for non-existing users.

Note that the first case above does not change the protocol so its implementation is a server's decision (the client side is not changed). The second case, requires changes on the client side so it changes OPAQUE itself.

[[https://github.com/cfrg/draft-irtf-cfrg-opaque/issues/22: Should this variant be documented/standardized?]]

6.7. 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 password to the server, even during registration. Note that a corrupted server can run an exhaustive offline dictionary attack to validate guesses for the user's password; this is inevitable in any aPAKE protocol. (OPAQUE enables a 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].)

Some applications may require learning the user'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.

7. Performance Considerations

The computational cost of OPAQUE is determined by the cost of the OPRF, the cost of a regular Diffie-Hellman exchange, and the cost of authenticating such exchange. In an elliptic-curve implementation of the OPRF, the cost for the client is two exponentiations (one or two of which can be fixed base) and one hashing-into-curve operation [I-D.irtf-cfrg-hash-to-curve]; for the server, it is just one exponentiation. The cost of a Diffie-Hellman exchange is as usual two exponentiations per party (one of which is fixed-base). Finally, the cost of authentication per party depends on the specific AKE protocol: it is just 1/6 of an exponentiation with HMQV, two exponentiations for 3DH, and it is one signature generation and verification in the case of SIGMA and TLS 1.3. These instantiations preserve the number of messages in the underlying AKE protocol except in implementations such as [I-D.sullivan-tls-opaque] where an additional round trip is required to provide privacy to account information.

8. IANA Considerations

This document makes no IANA requests.

9. References

9.1. Normative References

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

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

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

9.2. Informative References

[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.

[FK00] Ford, W. and B.S. Kaliski, Jr, "Server-assisted generation of a strong secret from a password", WETICE , 2000.

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

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

[I-D.irtf-cfrg-argon2] Biryukov, A., Dinu, D., Khovratovich, D., and S. Josefsson, "The memory-hard Argon2 password hash and proof-of-work function", Work in Progress, Internet-Draft, draft-irtf-cfrg-argon2-12, 8 September 2020, <http://www.ietf.org/internet-drafts/draft-irtf-cfrg-argon2-12.txt>.

[I-D.irtf-cfrg-hash-to-curve]
Faz-Hernandez, A., Scott, S., Sullivan, N., Wahby, R., and C. Wood, "Hashing to Elliptic Curves", Work in Progress, Internet-Draft, draft-irtf-cfrg-hash-to-curve-09, 29 June 2020, <http://www.ietf.org/internet-drafts/draft-irtf-cfrg-hash-to-curve-09.txt>.

[I-D.irtf-cfrg-voprf]
Davidson, A., Sullivan, N., and C. Wood, "Oblivious Pseudorandom Functions (OPRFs) using Prime-Order Groups", Work in Progress, Internet-Draft, draft-irtf-cfrg-

voprf-04, 13 July 2020, <http://www.ietf.org/internet-drafts/draft-irtf-cfrg-voprf-04.txt>.

[I-D.sullivan-tls-opaque]
Sullivan, N., Krawczyk, H., Friel, O., and R. Barnes, "Usage of OPAQUE with TLS 1.3", Work in Progress, Internet-Draft, draft-sullivan-tls-opaque-00, 11 March 2019, <http://www.ietf.org/internet-drafts/draft-sullivan-tls-opaque-00.txt>.

[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", DOI 10.6028/nist.sp.800-56ar3, National Institute of Standards and Technology report, April 2018, <https://doi.org/10.6028/nist.sp.800-56ar3>.

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

[RFC2898] Kaliski, B., "PKCS #5: Password-Based Cryptography Specification Version 2.0", RFC 2898, DOI 10.17487/RFC2898, September 2000, <https://www.rfc-editor.org/info/rfc2898>.

[RFC2945] Wu, T., "The SRP Authentication and Key Exchange System", RFC 2945, DOI 10.17487/RFC2945, September 2000, <https://www.rfc-editor.org/info/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/info/rfc5869>.

[RFC7250] Wouters, P., Ed., Tschofenig, H., Ed., Gilmore, J., Weiler, S., and T. Kivinen, "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport

Layer Security (DTLS)", RFC 7250, DOI 10.17487/RFC7250, June 2014, <https://www.rfc-editor.org/info/rfc7250>.

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

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

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

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

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

[SIGMA] Krawczyk, H., "SIGMA: The SIGn-and-MAc approach to authenticated Diffie-Hellman and its use in the IKE protocols", CRYPTO , 2003.

[SIGNAL] "Signal recommended cryptographic algorithms", https://signal.org/docs/specifications/doubleratchet/#recommended-cryptographic-algorithms , 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 Stas 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, Kevin Lewi, Payman Mohassel, Jason Resch, Greg Rubin, Nick Sullivan.

Authors' Addresses

Hugo Krawczyk
Algorand Foundation

Email: hugokraw@gmail.com

Kevin Lewi
Facebook

Email: lewi.kevin.k@gmail.com

Christopher A. Wood
Cloudflare

Email: caw@heapingbits.net