Network Working Group
Internet-Draft
Intended status: Informational
Expires: 3 September 2026

C. Yun
C. A. Wood
Apple, Inc.
A. Faz-Hernandez
Cloudflare

2 March 2026

Anonymous Rate-Limited Credentials Cryptography
draft-ietf-privacypass-arc-crypto-01

Abstract

This document specifies the Anonymous Rate-Limited Credential (ARC) protocol, a specialization of keyed-verification anonymous credentials with support for rate limiting. ARC credentials can be presented from client to server up to some fixed number of times, where each presentation is cryptographically bound to client secrets and application-specific public information, such that each presentation is unlinkable from the others as well as the original credential creation. ARC is useful in applications where a server needs to throttle or rate-limit access from anonymous clients.

About This Document

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

The latest revision of this draft can be found at https://ietf-wg-privacypass.github.io/draft-arc/draft-ietf-privacypass-arc-crypto.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-ietf-privacypass-arc-crypto/.

Discussion of this document takes place on the PRIVACYPASS Privacy Pass mailing list (mailto:privacy-pass@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/privacy-pass. Subscribe at https://www.ietf.org/mailman/listinfo/privacy-pass/.

Source for this draft and an issue tracker can be found at https://github.com/ietf-wg-privacypass/draft-arc.

Status of This Memo

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

This Internet-Draft will expire on 3 September 2026.

Copyright Notice

Table of Contents

1. Introduction

This document specifies the Anonymous Rate-Limited Credential (ARC) protocol, a specialization of keyed-verification anonymous credentials with support for rate limiting.

ARC is privately verifiable (keyed-verification), yet differs from similar token-based protocols in that each credential can be presented multiple times without violating unlinkability of different presentations. Servers issue credentials to clients that are cryptographically bound to client secrets and some public information. Afterwards, clients can present this credential to the server up to some fixed number of times, where each presentation provides proof that it was derived from a valid (previously issued) credential and bound to some public information. Each presentation is pairwise unlinkable, meaning the server cannot link any two presentations to the same client credential, nor can the server link a presentation to the preceding credential issuance flow. Notably, the maximum number of presentations from a credential is fixed by the application.

ARC is useful in settings where applications require a fixed number of zero-knowledge proofs about client secrets that can also be cryptographically bound to some public information. This capability lets servers use credentials in applications that need throttled or rate-limited access from anonymous clients.

2. Conventions and Definitions

2.1. Notation and Terminology

The following functions and notation are used throughout the document.

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

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

* random_integer_uniform(M, N): Generate a random, uniformly distributed integer R between M inclusive and N exclusive, i.e., M <= R < N.

* random_integer_uniform_excluding_set(M, N, S): Generate a random, uniformly distributed integer R between M inclusive and N exclusive, i.e., M <= R < N, such that R does not exist in the set of integers S.

All algorithms and procedures described in this document are laid out in a Python-like pseudocode. Each function takes a set of inputs and parameters and produces a set of output values. Parameters become constant values once the protocol variant and the ciphersuite are fixed.

The notation T U[N] refers to an array called U containing N items of type T. The type opaque means one single byte of uninterpreted data. Items of the array are zero-indexed and referred as U[j] such that 0 <= j < N. The notation {T} refers to a set consisting of elements of type T. For any object x, we write len(x) to denote its length in bytes.

String values such as "CredentialRequest", "CredentialResponse", "Presentation", and "Tag" are ASCII string literals.

The following terms are used throughout this document.

* Client: Protocol initiator. Creates a credential request, and uses the corresponding server response to make a credential. The client can make multiple presentations of this credential.

* Server: Computes a response to a credential request, with its server private keys. Later the server can verify the client’s presentations with its private keys. Learns nothing about the client’s secret attributes, and cannot link a client’s request/response and presentation steps.

3. Preliminaries

The construction in this document has one primary dependency:

* Group: A prime-order group implementing the API described below in Section 3.1. See Section 6 for specific instances of groups.

3.1. Prime-Order Group

In this document, we assume the construction of an additive, prime-order group Group for performing all mathematical operations. In prime-order groups, any element (other than the identity) can generate the other elements of the group. Usually, one element is fixed and defined as the group generator. In the ARC setting, there are two fixed generator elements (generatorG, generatorH). Such groups are uniquely determined by the choice of the prime p that defines the order of the group. (There may, however, exist different representations of the group for a single p. Section 6 lists specific groups which indicate both order and representation.)

The fundamental group operation is addition + with identity element I. For any elements A and B of the group, A + B = B + A is also a member of the group. Also, for any A in the group, there exists an element -A such that A + (-A) = (-A) + A = I. Scalar multiplication by r is equivalent to the repeated application of the group operation on an element A with itself r-1 times, this is denoted as r*A = A + ... + A. For any element A, p*A=I. The case when the scalar multiplication is performed on the group generator is denoted as ScalarMultGen(r). Given two elements A and B, the discrete logarithm problem is to find an integer k such that B = k*A. Thus, k is the discrete logarithm of B with respect to the base A. The set of scalars corresponds to GF(p), a prime field of order p, and are represented as the set of integers defined by {0, 1, ..., p-1}. This document uses types Element and Scalar to denote elements of the group and its set of scalars, respectively.

We now detail a number of member functions that can be invoked on a prime-order group.

* Order(): Outputs the order of the group (i.e. p).

* Identity(): Outputs the identity element of the group (i.e. I).

* Generator(): Outputs the fixed generator of the group.

* HashToGroup(x, info): Deterministically maps an array of bytes x with domain separation value info to an element of Group. The map must ensure that, for any adversary receiving R = HashToGroup(x, info), it is computationally difficult to reverse the mapping. Security properties of this function are described in [I-D.irtf-cfrg-hash-to-curve].

* HashToScalar(x, info): Deterministically maps an array of bytes x with domain separation value info to an element in GF(p). Security properties of this function are described in [I-D.irtf-cfrg-hash-to-curve], Section 10.5.

* RandomScalar(): Chooses at random a non-zero element in GF(p).

* ScalarInverse(s): Returns the inverse of input Scalar s on GF(p).

* SerializeElement(A): Maps an Element A to a canonical byte array buf of fixed length Ne.

* DeserializeElement(buf): Attempts to map a byte array buf to an Element A, and fails if the input is not the valid canonical byte representation of an element of the group. This function can raise a DeserializeError if deserialization fails or A is the identity element of the group; see Section 6 for group-specific input validation steps.

* SerializeScalar(s): Maps a Scalar s to a canonical byte array buf of fixed length Ns.

* DeserializeScalar(buf): Attempts to map a byte array buf to a Scalar s. This function can raise a DeserializeError if deserialization fails; see Section 6 for group-specific input validation steps.

For each group, there exists two distinct generators, generatorG and generatorH, generatorG = G.Generator() and generatorH = G.HashToGroup(G.SerializeElement(generatorG), "generatorH"). The group member functions GeneratorG() and GeneratorH() are shorthand for returning generatorG and generatorH, respectively.

Section 6 contains details for the implementation of this interface for different prime-order groups instantiated over elliptic curves.

4. ARC Protocol

The ARC protocol is a two-party protocol run between client and server consisting of three distinct phases:

1. Key generation. In this phase, the server generates its private and public keys to be used for the remaining phases. This phase is described in Section 4.1.

2. Credential issuance. In this phase, the client and server interact to issue the client a credential that is cryptographically bound to client secrets. This phase is described in Section 4.2.

3. Presentation. In this phase, the client uses the credential to create a "presentation" to the server, where the server learns nothing more than whether or not the presentation is valid and corresponds to some previously issued credential, without learning which credential it corresponds to. This phase is described in Section 4.3.

This protocol bears resemblance to anonymous token protocols, such as those built on Blind RSA [BLIND-RSA] and Oblivious Pseudorandom Functions [OPRFS] with one critical distinction: unlike anonymous

tokens, an anonymous credential can be used multiple times to create unlinkable presentations (up to the fixed presentation limit). This means that a single issuance invocation can drive multiple presentation invocations, whereas with anonymous tokens, each presentation invocation requires exactly one issuance invocation. As a result, credentials are generally longer lived than tokens. Applications configure the credential presentation limit after the credential is issued such that client and server agree on the limit during presentation. Servers are responsible for ensuring this limit is not exceeded. Clients that exceed the agreed-upon presentation limit break the unlinkability guarantees provided by the protocol.

The rest of this section describes the three phases of the ARC protocol.

4.1. Key Generation

In the key generation phase, the server generates its private and public keys, denoted ServerPrivateKey and ServerPublicKey, as follows.

```
Input: None
Output:
- ServerPrivateKey:
  - x0: Scalar
  - x1: Scalar
  - x2: Scalar
  - x0Blinding: Scalar
- ServerPublicKey:
  - X0: Element
  - X1: Element
  - X2: Element

Parameters
- Group G

def SetupServer():
  x0 = G.RandomScalar()
  x1 = G.RandomScalar()
  x2 = G.RandomScalar()
  x0Blinding = G.RandomScalar()
  X0 = x0 * G.GeneratorG() + x0Blinding * G.GeneratorH()
  X1 = x1 * G.GeneratorH()
  X2 = x2 * G.GeneratorH()
  return (ServerPrivateKey(x0, x1, x2, x0Blinding),
    ServerPublicKey(X0, X1, X2))
```

The server public key can be serialized as follows:

```
struct {
  uint8 X0[Ne]; // G.SerializeElement(X0)
  uint8 X1[Ne]; // G.SerializeElement(X1)
  uint8 X2[Ne]; // G.SerializeElement(X2)
} ServerPublicKey;
```

The length of this encoded response structure is NserverPublicKey = 3*Ne.

4.2. Issuance

The purpose of the issuance phase is for the client and server to cooperatively compute a credential that is cryptographically bound to the client's secrets. Clients do not choose these secrets; they are computed by the protocol.

The issuance phase of the protocol requires clients to know the server public key a priori, as well as an arbitrary, application-specific request context. It requires no other input. It consists of three distinct steps:

1. The client generates and sends a credential request to the server. This credential request contains a proof that the request is valid with respect to the client's secrets and request context. See Section 4.2.1 for details about this step.

2. The server validates the credential request. If valid, it computes a credential response with the server private keys. The response includes a proof that the credential response is valid with respect to the server keys. The server sends the response to the client. See Section 4.2.2 for details about this step.

3. The client finalizes the credential by processing the server response. If valid, this step yields a credential that can then be used in the presentation phase of the protocol. See Section 4.2.3 for details about this step.

Each of these steps is described in the following subsections.

4.2.1. Credential Request

Given a request context, the process for creating a credential request is as follows:

```
(clientSecrets, request) = CreateCredentialRequest(requestContext, rng)

Inputs:
- requestContext: Data, context for the credential request
- rng: a cryptographically secure random number generator, as defined
  in the Sigma Protocols specification.

Outputs:
- request:
  - m1Enc: Element, first encrypted secret.
  - m2Enc: Element, second encrypted secret.
  - requestProof: String (ZKProof), a proof of correct generation of
    m1Enc and m2Enc.
- clientSecrets:
  - m1: Scalar, first secret.
  - m2: Scalar, second secret.
  - r1: Scalar, blinding factor for first secret.
  - r2: Scalar, blinding factor for second secret.

Parameters:
- G: Group
- generatorG: Element, equivalent to G.GeneratorG()
- generatorH: Element, equivalent to G.GeneratorH()

def CreateCredentialRequest(requestContext, rng):
  m1 = G.RandomScalar()
  m2 = G.HashToScalar(requestContext, "requestContext")
  r1 = G.RandomScalar()
  r2 = G.RandomScalar()
  m1Enc = m1 * generatorG + r1 * generatorH
  m2Enc = m2 * generatorG + r2 * generatorH
  requestProof = MakeCredentialRequestProof(m1, m2, r1, r2,
    m1Enc, m2Enc, rng)
  request = (m1Enc, m2Enc, requestProof)
  clientSecrets = (m1, m2, r1, r2)
  return (clientSecrets, request)

  See Section 5.1 for more details on the generation of the credential
  request proof.

  The resulting request can be serialized as follows.

  struct {
    uint8 m1Enc[Ne];
    uint8 m2Enc[Ne];
    uint8 requestProof[5*Ns];
  } CredentialRequest;
```

The length of this encoded request structure is Nrequest = 2*Ne + 5*Ns.

4.2.2. Credential Response

Given a credential request and server public and private keys, the process for creating a credential response is as follows.

```
response = CreateCredentialResponse(serverPrivateKey,
  serverPublicKey, request, rng)

Inputs:
- serverPrivateKey:
  - x0: Scalar (private), server private key 0.
  - x1: Scalar (private), server private key 1.
  - x2: Scalar (private), server private key 2.
  - x0Blinding: Scalar (private), blinding value for x0.
- serverPublicKey:
  - X0: Element, server public key 0.
  - X1: Element, server public key 1.
  - X2: Element, server public key 2.
- request:
  - m1Enc: Element, first encrypted secret.
  - m2Enc: Element, second encrypted secret.
  - requestProof: String (ZKProof), a proof of correct generation
    of m1Enc and m2Enc.
- rng: a cryptographically secure random number generator, as defined
  in the Sigma Protocols specification.

Outputs:
- U: Element, a randomized generator for the response, ‘b*G‘.
- encUPrime: Element, encrypted UPrime.
- X0Aux: Element, auxiliary point for X0.
- X1Aux: Element, auxiliary point for X1.
- X2Aux: Element, auxiliary point for X2.
- HAux: Element, auxiliary point for generatorH.
- responseProof: String (ZKProof), a proof of correct generation of
  U, encUPrime, server public keys, and auxiliary points.

Parameters:
- G: Group
- generatorG: Element, equivalent to G.GeneratorG()
- generatorH: Element, equivalent to G.GeneratorH()

Exceptions:
- VerifyError, raised when response verification fails

def CreateCredentialResponse(serverPrivateKey, serverPublicKey, request, rng):
```

```
    if VerifyCredentialRequestProof(request) == false:
      raise VerifyError

    b = G.RandomScalar()
    U = b * generatorG
    encUPrime = b * (serverPublicKey.X0 +
          serverPrivateKey.x1 * request.m1Enc +
          serverPrivateKey.x2 * request.m2Enc)
    X0Aux = b * serverPrivateKey.x0Blinding * generatorH
    X1Aux = b * serverPublicKey.X1
    X2Aux = b * serverPublicKey.X2
    HAux = b * generatorH

    responseProof = MakeCredentialResponseProof(serverPrivateKey,
      serverPublicKey, request, b, U, encUPrime,
      X0Aux, X1Aux, X2Aux, HAux, rng)
    return (U, encUPrime, X0Aux, X1Aux, X2Aux, HAux, responseProof)
```

The resulting response can be serialized as follows. See Section 5.2 for more details on the generation of the credential response proof.

```
struct {
  uint8 U[Ne];
  uint8 encUPrime[Ne];
  uint8 X0Aux[Ne];
  uint8 X1Aux[Ne];
  uint8 X2Aux[Ne];
  uint8 HAux[Ne];
  uint8 responseProof[8*Ns];
} CredentialResponse
```

The length of this encoded response structure is Nresponse = 6*Ne + 8*Ns.

4.2.3. Finalize Credential

Given a credential request and response, server public keys, and the client secrets produced when creating a credential request, the process for finalizing the issuance flow and creating a credential is as follows.

```
credential = FinalizeCredential(clientSecrets,
  serverPublicKey, request, response)
```

Inputs:
- clientSecrets:
  - m1: Scalar, first secret.
  - m2: Scalar, second secret.

```
    - r1: Scalar, blinding factor for first secret.
    - r2: Scalar, blinding factor for second secret.
  - serverPublicKey: ServerPublicKey, shared with the client out-of-band
  - request:
    - m1Enc: Element, first encrypted secret.
    - m2Enc: Element, second encrypted secret.
    - requestProof: String (ZKProof), a proof of correct generation
      of m1Enc and m2Enc.
  - response:
    - U: Element, a randomized generator for the response. ‘b*G‘.
    - encUPrime: Element, encrypted UPrime.
    - X0Aux: Element, auxiliary point for X0.
    - X1Aux: Element, auxiliary point for X1.
    - X2Aux: Element, auxiliary point for X2.
    - HAux: Element, auxiliary point for generatorH.
    - responseProof: String (ZKProof), a proof of correct generation of
      U, encUPrime, server public keys, and auxiliary points.

  Outputs:
  - credential:
    - m1: Scalar, client’s first secret.
    - U: Element, a randomized generator for the response. ‘b*G‘.
    - UPrime: Element, the MAC over the server’s private keys and the
      client’s secrets.
    - X1: Element, server public key 1.

  Exceptions:
  - VerifyError, raised when response verification fails

  Parameters:
  - G: Group
  - generatorG: Element, equivalent to G.GeneratorG()
  - generatorH: Element, equivalent to G.GeneratorH()

  def FinalizeCredential(clientSecrets, serverPublicKey, request,
    response):
    if VerifyCredentialResponseProof(serverPublicKey, response,
      request) == false:
      raise VerifyError
    UPrime = response.encUPrime - response.X0Aux
      - clientSecrets.r1 * response.X1Aux
      - clientSecrets.r2 * response.X2Aux
    return (clientSecrets.m1, response.U, UPrime, serverPublicKey.X1)
```

4.3. Presentation

The purpose of the presentation phase is for the client to create a "presentation" to the server which can be verified using the server private key. This phase is non-interactive, i.e., there is no state stored between client and server in order to produce and then verify a presentation. Client and server agree upon a fixed limit of presentations in order to create and verify presentations; presentations will not verify correctly if the client and server use different limits.

This phase consists of three steps:

1. The client creates a presentation state for a given presentation context and presentation limit. This state is used to produce a fixed amount of presentations.

2. The client creates a presentation from the presentation state and sends it to the server. The presentation is cryptographically bound to the state’s presentation context, and contains proof that the presentation is valid with respect to the presentation context. Moreover, the presentation contains proof that the nonce (an integer) associated with this presentation is within the presentation limit. The nonce value used in each presentation is hidden in a Pedersen commitment, ensuring servers cannot link presentations using the nonce.

3. The server verifies the presentation with respect to the presentation context and presentation limit.

Details for each of these steps are in the following subsections.

4.3.1. Presentation State

Presentation state is used to track the number of presentations for a given credential. This state is important for ARC’s unlinkability goals: reuse of state can break unlinkability properties of credential presentations. State is initialized with a credential, presentation context, and presentation limit. It is then mutated after each presentation construction (as described in Section 4.3.2).

```
   state = MakePresentationState(credential, presentationContext,
     presentationLimit)

   Inputs:
   - credential:
     - m1: Scalar, client’s first secret.
     - U: Element, a randomized generator for the response ‘b*G‘.
     - UPrime: Element, the MAC over the server’s private keys and the
       client’s secrets.
     - X1: Element, server public key 1.
   - presentationContext: Data (public), used for presentation tag
     computation.
   - presentationLimit: Integer, the fixed presentation limit.

   Outputs:
   - credential
   - presentationContext: Data (public), used for presentation tag
     computation.
   - nextNonce: Integer, the next nonce that can be used. This
     increments by 1 for each use.
   - presentationLimit: Integer, the fixed presentation limit.

   def MakePresentationState(credential, presentationContext,
     presentationLimit):
     return PresentationState(credential, presentationContext, 0,
       presentationLimit)
```

4.3.2. Presentation Construction

Creating a presentation requires a credential, presentation context, and presentation limit. This process is necessarily stateful on the client since the number of times a credential is used for a given presentation context cannot exceed the presentation limit; doing so would break presentation unlinkability, as two presentations created with the same nonce can be directly compared for equality (via the "tag"). As a result, the process for creating a presentation accepts as input a presentation state and then outputs an updated presentation state.

```
newState, nonce, presentation = Present(state, rng)

Inputs:
state: input PresentationState
  - credential
  - presentationContext: Data (public), used for presentation tag
    computation.
  - nextNonce: Integer, the next nonce that can be used. This
    increments by 1 for each use.
```

```
  - presentationLimit: Integer, the fixed presentation limit.
rng: a cryptographically secure random number generator, as defined
  in the Sigma Protocol spec {{SIGMA}}.

Outputs:
- newState: updated PresentationState
- nonce: Integer, the nonce associated with this presentation.
- presentation:
  - U: Element, re-randomized from the U in the response.
  - UPrimeCommit: Element, a public key to the issued UPrime.
  - m1Commit: Element, a public key to the client secret (m1).
  - tag: Element, the tag element used for enforcing the
    presentation limit.
  - nonceCommit: Element, a Pedersen commitment to the nonce.
  - presentationProof: ZKProof, a joint proof of correct generation
    of the presentation and that the committed nonce is in
    [0, presentationLimit).

Parameters:
- G: Group
- generatorG: Element, equivalent to G.GeneratorG()
- generatorH: Element, equivalent to G.GeneratorH()

Exceptions:
- LimitExceededError, raised when the presentation count meets or
  exceeds the presentation limit for the given presentation context

def Present(state, rng):
  if state.nextNonce >= state.presentationLimit:
    raise LimitExceededError

  nonce = state.nextNonce
  # This step mutates the state by incrementing nextNonce by 1.
  state.nextNonce += 1

  a = G.RandomScalar()
  r = G.RandomScalar()
  z = G.RandomScalar()

  U = a * state.credential.U
  UPrime = a * state.credential.UPrime
  UPrimeCommit = UPrime + r * generatorG
  m1Commit = state.credential.m1 * U + z * generatorH

  # Create Pedersen commitment to the nonce
  nonceBlinding = G.RandomScalar()
  nonceCommit = G.Scalar(nonce) * generatorG + nonceBlinding * generatorH
```

```
  generatorT = G.HashToGroup(state.presentationContext, "Tag")
  tag = (state.credential.m1 + nonce)^(-1) * generatorT
  V = z * state.credential.X1 - r * generatorG

  # Generate presentation proof with integrated range proof
  presentationProof = MakePresentationProof(U, UPrimeCommit,
    m1Commit, tag, generatorT, state.credential, V, r, z, nonce,
    nonceBlinding, nonceCommit, state.presentationLimit, rng)

  presentation = (U, UPrimeCommit, m1Commit, tag, nonceCommit,
    presentationProof)

  return state, nonce, presentation
```

OPEN ISSUE: should the tag also fold in the presentation limit?

The resulting presentation can be serialized as follows. See Section 5.3 for more details on the generation of the presentation proof. The presentation proof integrates the range proof as described in Section 5.4.

```
struct {
  uint8 U[Ne];
  uint8 UPrimeCommit[Ne];
  uint8 m1Commit[Ne];
  uint8 tag[Ne];
  uint8 nonceCommit[Ne];
  PresentationProof presentationProof;
} Presentation

struct {
  uint8 D[k][Ne]; // k = ceil(log2(presentationLimit))
  uint8 challenge[Ns];
  // Variable length based on presentation variables plus range proof variables
  uint8 responses[5 + 3 * k][Ns];
} PresentationProof
```

The length of the Presentation structure is Npresentation = 5*Ne + Npresentationproof. Npresentationproof = k * Ne + (6 + 3 * k) * Ns, which includes the D commitments (k * Ne), the challenge (Ns), the response scalars for presentation variables (5 scalars: m1, z, -r, nonce, nonceBlinding), and range proof variables (3 * k scalars: b[i], s[i], s2[i] for each bit). k is the number of bits it takes to represent the presentationLimit, i.e., k = ceil(log2(presentationLimit))

4.3.3. Presentation Verification

The server processes the presentation by verifying the integrated presentation proof, which includes verification of the range proof, against server-computed values. Note that the server does not receive the raw nonce value, only the Pedersen commitment to it.

```
validity, tag = VerifyPresentation(serverPrivateKey,
  serverPublicKey, requestContext, presentationContext,
  presentation, presentationLimit)

Inputs:
- serverPrivateKey:
  - x0: Scalar (private), server private key 0.
  - x1: Scalar (private), server private key 1.
  - x2: Scalar (private), server private key 2.
  - x0Blinding: Scalar (private), blinding value for x0.
- serverPublicKey:
  - X0: Element, server public key 0.
  - X1: Element, server public key 1.
  - X2: Element, server public key 2.
- requestContext: Data, context for the credential request.
- presentationContext: Data (public), used for presentation tag
  computation.
- presentation:
  - U: Element, re-randomized from the U in the response.
  - UPrimeCommit: Element, a public key to the issued UPrime.
  - m1Commit: Element, a public key to the client secret (m1).
  - tag: Element, the tag element used for enforcing the
    presentation limit.
  - nonceCommit: Element, a Pedersen commitment to the nonce.
  - presentationProof: ZKProof, a joint proof of correct generation
    of the presentation and that the committed nonce is in
    [0, presentationLimit).
- presentationLimit: Integer, the fixed presentation limit.

Outputs:
- validity: Boolean, True if the presentation is valid,
  False otherwise.
- tag: Bytes, the value of the presentation tag used for rate
  limiting.

Parameters:
- G: Group
- generatorG: Element, equivalent to G.GeneratorG()
- generatorH: Element, equivalent to G.GeneratorH()

def VerifyPresentation(serverPrivateKey, serverPublicKey,
```

```
    requestContext, presentationContext, presentation,
    presentationLimit):

    # The presentation proof verifies the relationship between the tag,
    # m1, and the committed nonce using zero-knowledge techniques,
    # without learning the value of the nonce. It also includes
    # verification that the committed nonce is in
    # [0, presentationLimit).

    validity = VerifyPresentationProof(serverPrivateKey,
      serverPublicKey, requestContext, presentationContext,
      presentation, presentationLimit)

    return validity, presentation.tag
```

Implementation-specific steps: the server must perform a check that the tag (presentation.tag) has not previously been seen, to prevent double spending. It then stores the tag for use in future double spending checks. To reduce the overhead of performing double spend checks, the server can store and look up the tags corresponding to the associated requestContext and presentationContext values.

5. Zero-Knowledge Proofs

This section uses the Interactive Sigma Protocol [SIGMA] to create zero-knowledge proofs of knowledge for various ARC operations, and the Fiat-Shamir Transform [FIAT-SHAMIR] to make those proofs non-interactive.

5.1. CredentialRequest Proof

The request proof is a proof of knowledge of (m1, m2, r1, r2) used to generate the encrypted request. Statements to prove:

1. m1Enc = m1 * generatorG + r1 * generatorH
2. m2Enc = m2 * generatorG + r2 * generatorH

5.1.1. CredentialRequest Proof Creation

```
   requestProof = MakeCredentialRequestProof(m1, m2, r1, r2, m1Enc,
     m2Enc, rng)

   Inputs:
   - m1: Scalar, first secret.
   - m2: Scalar, second secret.
   - r1: Scalar, blinding factor for first secret.
   - r2: Scalar, blinding factor for second secret.
   - m1Enc: Element, first encrypted secret.
   - m2Enc: Element, second encrypted secret.
   - rng: a cryptographically secure random number generator, as defined
     in the Sigma Protocols specification.

   Outputs:
   - proof: String (ZKProof)

   Parameters:
   - G: Group
   - generatorG: Element, equivalent to G.GeneratorG()
   - generatorH: Element, equivalent to G.GeneratorH()
   - contextString: public input

   def MakeCredentialRequestProof(m1, m2, r1, r2, m1Enc, m2Enc, rng):
     statement = LinearRelation(G)
     [m1Var, m2Var, r1Var, r2Var] = statement.allocate_scalars(4)
     witness = [m1, m2, r1, r2]

     [genGVar, genHVar, m1EncVar, m2EncVar]
       = statement.allocate_elements(4)
     statement.set_elements([(genGVar, generatorG),
       (genHVar, generatorH), (m1EncVar, m1Enc), (m2EncVar, m2Enc)])

     # 1. m1Enc = m1 * generatorG + r1 * generatorH
     statement.append_equation(m1EncVar,
       [(m1Var, genGVar), (r1Var, genHVar)])

     # 2. m2Enc = m2 * generatorG + r2 * generatorH
     statement.append_equation(m2EncVar,
       [(m2Var, genGVar), (r2Var, genHVar)])

     session_id = contextString + "CredentialRequest"
     prover = NISchnorrProofShake128P256(session_id, statement)
     return prover.prove(witness, rng)
```

Where the request proof returned is a serialization of the following: ~~~ - challenge: Scalar, the challenge used in the proof of valid encryption. - response0: Scalar, the response corresponding to m1. - response1: Scalar, the response corresponding to m2. - response2: Scalar, the response corresponding to r1. - response3: Scalar, the response corresponding to r2. ~~~

5.1.2. CredentialRequest Proof Verification

```
validity = VerifyCredentialRequestProof(request)

Inputs:
- request:
  - m1Enc: Element, first encrypted secret.
  - m2Enc: Element, second encrypted secret.
  - requestProof: String (ZKProof), a proof of correct generation
    of m1Enc and m2Enc.

Outputs:
- validity: Boolean, True if the proof verifies correctly, False otherwise.

Parameters:
- G: group
- generatorG: Element, equivalent to G.GeneratorG()
- generatorH: Element, equivalent to G.GeneratorH()
- contextString: public input

def VerifyCredentialRequestProof(request):
  statement = LinearRelation(G)
  [m1Var, m2Var, r1Var, r2Var] = statement.allocate_scalars(4)

  [genGVar, genHVar, m1EncVar, m2EncVar]
    = statement.allocate_elements(4)
  statement.set_elements([(genGVar, generatorG),
    (genHVar, generatorH), (m1EncVar, m1Enc), (m2EncVar, m2Enc)])

  # 1. m1Enc = m1 * generatorG + r1 * generatorH
  statement.append_equation(m1EncVar,
    [(m1Var, genGVar), (r1Var, genHVar)])

  # 2. m2Enc = m2 * generatorG + r2 * generatorH
  statement.append_equation(m2EncVar,
    [(m2Var, genGVar), (r2Var, genHVar)])

  session_id = contextString + "CredentialRequest"
  verifier = NISchnorrProofShake128P256(session_id, statement)
  return verifier.verify(request.requestProof)
```

5.2. CredentialResponse Proof

The response proof is a proof of knowledge of (x0, x1, x2, x0Blinding, b) used in the server's CredentialResponse for the client's CredentialRequest. Statements to prove:

1. X0 = x0 * generatorG + x0Blinding * generatorH
2. X1 = x1 * generatorH
3. X2 = x2 * generatorH
4. X0Aux = b * x0Blinding * generatorH
   4a. HAux = b * generatorH
   4b: X0Aux = x0Blinding * HAux (= b * x0Blinding * generatorH)
5. X1Aux = b * x1 * generatorH
   5a. X1Aux = t1 * generatorH (t1 = b * x1)
   5b. X1Aux = b * X1 (X1 = x1 * generatorH)
6. X2Aux = b * x2 * generatorH
   6a. X2Aux = b * X2 (X2 = x2 * generatorH)
   6b. X2Aux = t2 * generatorH (t2 = b * x2)
7. U = b * generatorG
8. encUPrime = b * (X0 + x1 * Enc(m1) + x2 * Enc(m2))

5.2.1. CredentialResponse Proof Creation

```
responseProof = MakeCredentialResponseProof(serverPrivateKey,
  serverPublicKey, request, b, U, encUPrime,
  X0Aux, X1Aux, X2Aux, HAux, rng)
```

Inputs:
- serverPrivateKey:
  - x0: Scalar (private), server private key 0.
  - x1: Scalar (private), server private key 1.
  - x2: Scalar (private), server private key 2.
  - x0Blinding: Scalar (private), blinding value for x0.
- serverPublicKey:
  - X0: Element, server public key 0.
  - X1: Element, server public key 1.
  - X2: Element, server public key 2.
- request:
  - m1Enc: Element, first encrypted secret.
  - m2Enc: Element, second encrypted secret.
  - requestProof: String (ZKProof), a proof of correct generation of m1Enc and m2Enc.
- encUPrime: Element, encrypted UPrime.
- X0Aux: Element, auxiliary point for X0.
- X1Aux: Element, auxiliary point for X1.
- X2Aux: Element, auxiliary point for X2.
- HAux: Element, auxiliary point for generatorH.
- rng: a cryptographically secure random number generator, as defined

```
    in the Sigma Protocols specification.

  Outputs:
  - proof: String (ZKProof)

  Parameters:
  - G: Group
  - generatorG: Element, equivalent to G.GeneratorG()
  - generatorH: Element, equivalent to G.GeneratorH()
  - contextString: public input

  def MakeCredentialResponseProof(serverPrivateKey, serverPublicKey,
    request, b, U, encUPrime, X0Aux, X1Aux, X2Aux, HAux, rng):
    statement = LinearRelation(G)
    [x0Var, x1Var, x2Var, x0BlindingVar, bVar, t1Var, t2Var]
      = statement.allocate_scalars(7)
    witness = [serverPrivateKey.x0, serverPrivateKey.x1,
      serverPrivateKey.x2, serverPrivateKey.x0Blinding, b,
      b*serverPrivateKey.x1, b*serverPrivateKey.x2]

    [genGVar, genHVar, m1EncVar, m2EncVar, UVar, encUPrimeVar, X0Var,
      X1Var, X2Var, X0AuxVar, X1AuxVar, X2AuxVar, HAuxVar]
      = statement.allocate_elements(13)
    statement.set_elements([(genGVar, generatorG),
      (genHVar, generatorH), (m1EncVar, request.m1Enc),
      (m2EncVar, request.m2Enc), (UVar, U), (encUPrimeVar, encUPrime),
      (X0Var, serverPublicKey.X0), (X1Var, serverPublicKey.X1),
      (X2Var, serverPublicKey.X2), (X0AuxVar, X0Aux),
      (X1AuxVar, X1Aux), (X2AuxVar, X2Aux), (HAuxVar, HAux)])

    # 1. X0 = x0 * generatorG + x0Blinding * generatorH
    statement.append_equation(X0Var, [(x0Var, genGVar),
      (x0BlindingVar, genHVar)])
    # 2. X1 = x1 * generatorH
    statement.append_equation(X1Var, [(x1Var, genHVar)])
    # 3. X2 = x2 * generatorH
    statement.append_equation(X2Var, [(x2Var, genHVar)])

    # 4. X0Aux = b * x0Blinding * generatorH
    # 4a. HAux = b * generatorH
    statement.append_equation(HAuxVar, [(bVar, genHVar)])
    # 4b: X0Aux = x0Blinding * HAux (= b * x0Blinding * generatorH)
    statement.append_equation(X0AuxVar, [(x0BlindingVar, HAuxVar)])

    # 5. X1Aux = b * x1 * generatorH
    # 5a. X1Aux = t1 * generatorH (t1 = b * x1)
    statement.append_equation(X1AuxVar, [(t1Var, genHVar)])
    # 5b. X1Aux = b * X1 (X1 = x1 * generatorH)
```

```
    statement.append_equation(X1AuxVar, [(bVar, X1Var)])

    # 6. X2Aux = b * x2 * generatorH
    # 6a. X2Aux = b * X2 (X2 = x2 * generatorH)
    pstatement.append_equation(X2AuxVar, [(bVar, X2Var)])
    # 6b. X2Aux = t2 * H (t2 = b * x2)
    statement.append_equation(X2AuxVar, [(t2Var, genHVar)])

    # 7. U = b * generatorG
    statement.append_equation(UVar, [(bVar, genGVar)])
    # 8. encUPrime = b * (X0 + x1 * Enc(m1) + x2 * Enc(m2))
    # simplified: encUPrime = b * X0 + t1 * m1Enc + t2 * m2Enc,
    # since t1 = b * x1 and t2 = b * x2
    statement.append_equation(encUPrimeVar, [(bVar, X0Var),
      (t1Var, m1EncVar), (t2Var, m2EncVar)])

    session_id = contextString + "CredentialResponse"
    prover = NISchnorrProofShake128P256(session_id, statement)
    return prover.prove(witness, rng)
```

Where the response proof returned is a serialization of the following: ~~~ - challenge: Scalar, the challenge used in the proof of valid response. - response0: Scalar, the response corresponding to x0. - response1: Scalar, the response corresponding to x1. - response2: Scalar, the response corresponding to x2. - response3: Scalar, the response corresponding to x0Blinding. - response4: Scalar, the response corresponding to b. - response5: Scalar, the response corresponding to t1. - response6: Scalar, the response corresponding to t2. ~~~

5.2.2. CredentialResponse Proof Verification

validity = VerifyCredentialResponseProof(serverPublicKey, response, request)

Inputs:

- serverPublicKey:
  - X0: Element, server public key 0.
  - X1: Element, server public key 1.
  - X2: Element, server public key 2.
- response:
  - U: Element, a randomized generator for the response. ‘b*G‘.
  - encUPrime: Element, encrypted UPrime.
  - X0Aux: Element, auxiliary point for X0.
  - X1Aux: Element, auxiliary point for X1.
  - X2Aux: Element, auxiliary point for X2.
  - HAux: Element, auxiliary point for generatorH.
  - responseProof: String (ZKProof), a proof of correct generation of

```
      U, encUPrime, server public keys, and auxiliary points.
  - request:
    - m1Enc: Element, first encrypted secret.
    - m2Enc: Element, second encrypted secret.
    - requestProof: String (ZKProof), a proof of correct generation of
      m1Enc and m2Enc.

  Outputs:
  - validity: Boolean, True if the proof verifies correctly,
    False otherwise.

  Parameters:
  - G: Group
  - generatorG: Element, equivalent to G.GeneratorG()
  - generatorH: Element, equivalent to G.GeneratorH()

  def VerifyCredentialResponseProof(serverPublicKey, response, request):
    statement = LinearRelation(G)
    [x0Var, x1Var, x2Var, x0BlindingVar, bVar, t1Var, t2Var]
      = statement.allocate_scalars(7)

    [genGVar, genHVar, m1EncVar, m2EncVar, UVar, encUPrimeVar, X0Var,
      X1Var, X2Var, X0AuxVar, X1AuxVar, X2AuxVar, HAuxVar]
      = statement.allocate_elements(13)
    statement.set_elements([(genGVar, generatorG),
      (genHVar, generatorH), (m1EncVar, request.m1Enc),
      (m2EncVar, request.m2Enc), (UVar, response.U),
      (encUPrimeVar, response.encUPrime), (X0Var, serverPublicKey.X0),
      (X1Var, serverPublicKey.X1), (X2Var, serverPublicKey.X2),
      (X0AuxVar, response.X0Aux), (X1AuxVar, response.X1Aux),
      (X2AuxVar, response.X2Aux), (HAuxVar, response.HAux)])

    # 1. X0 = x0 * generatorG + x0Blinding * generatorH
    statement.append_equation(X0Var, [(x0Var, genGVar),
      (x0BlindingVar, genHVar)])
    # 2. X1 = x1 * generatorH
    statement.append_equation(X1Var, [(x1Var, genHVar)])
    # 3. X2 = x2 * generatorH
    statement.append_equation(X2Var, [(x2Var, genHVar)])

    # 4. X0Aux = b * x0Blinding * generatorH
    # 4a. HAux = b * generatorH
    statement.append_equation(HAuxVar, [(bVar, genHVar)])
    # 4b: X0Aux = x0Blinding * HAux (= b * x0Blinding * generatorH)
    statement.append_equation(X0AuxVar, [(x0BlindingVar, HAuxVar)])

    # 5. X1Aux = b * x1 * generatorH
    # 5a. X1Aux = t1 * generatorH (t1 = b * x1)
```

```
    statement.append_equation(X1AuxVar, [(t1Var, genHVar)])
    # 5b. X1Aux = b * X1 (X1 = x1 * generatorH)
    statement.append_equation(X1AuxVar, [(bVar, X1Var)])

    # 6. X2Aux = b * x2 * generatorH
    # 6a. X2Aux = b * X2 (X2 = x2 * generatorH)
    pstatement.append_equation(X2AuxVar, [(bVar, X2Var)])
    # 6b. X2Aux = t2 * H (t2 = b * x2)
    statement.append_equation(X2AuxVar, [(t2Var, genHVar)])

    # 7. U = b * generatorG
    statement.append_equation(UVar, [(bVar, genGVar)])
    # 8. encUPrime = b * (X0 + x1 * Enc(m1) + x2 * Enc(m2))
    # simplified: encUPrime = b * X0 + t1 * m1Enc + t2 * m2Enc,
    # since t1 = b * x1 and t2 = b * x2
    statement.append_equation(encUPrimeVar, [(bVar, X0Var),
      (t1Var, m1EncVar), (t2Var, m2EncVar)])

    session_id = contextString + "CredentialResponse"
    verifier = NISchnorrProofShake128P256(session_id, statement)
    return verifier.verify(response.responseProof)
```

5.3. Presentation Proof

The presentation proof is a proof of knowledge of (m1, r, z, nonce, nonceBlinding) used in the presentation, as well as a proof that nonce is in the range [0, presentationLimit).

Statements to prove:

```
# The m1 commitment was correctly formed
1. m1Commit = m1 * U + z * generatorH
# Other presentation elements are consistent with the credential
2. V = z * X1 - r * generatorG
# The nonceCommit is a Pedersen commitment to nonce with blinding factor nonceBlinding
3. nonceCommit = nonce * generatorG + nonceBlinding * generatorH
# The tag was correctly computed using m1 and the nonce
4. T = m1 * tag + nonce * tag, where T = G.HashToGroup(presentationContext, "Tag")
# The nonce is in the range [0, presentationLimit)
5. constraints added by the range proof. See {#range-proof}.
```

5.3.1. Presentation Proof Creation

```
presentationProof = MakePresentationProof(U, UPrimeCommit,
  m1Commit, tag, generatorT, credential, V, r, z, nonce,
  nonceBlinding, nonceCommit, presentationLimit, rng)

Inputs:
- U: Element, re-randomized from the U in the response.
- UPrimeCommit: Element, a public key to the MACGGM output UPrime.
- m1Commit: Element, a public key to the client secret (m1).
- tag: Element, the tag element used for enforcing the presentation
  limit.
- generatorT: Element, used for presentation tag computation.
- credential:
  - m1: Scalar, client's first secret.
  - U: Element, a randomized generator for the response. 'b*G'.
  - UPrime: Element, the MAC over the server's private keys and the
    client's secrets.
  - X1: Element, server public key 1.
- V: Element, a proof helper element.
- r: Scalar (private), a randomly generated element used in
  presentation.
- z: Scalar (private), a randomly generated element used in
  presentation.
- nonce: Int, the nonce associated with the presentation.
- nonceBlinding: Scalar (private), the blinding factor for the nonce commitment.
- nonceCommit: Element, the Pedersen commitment to the nonce.
- presentationLimit: Integer, the fixed presentation limit.
- rng: a cryptographically secure random number generator, as defined
  in the Sigma Protocols specification.

Outputs:
- presentationProof: ZKProof, a joint proof covering both
  presentation and range proof
  - D: [Element], array of commitments to the bit decomposition of
    the nonce
  - challenge: Scalar, the challenge used in the proof of valid
    presentation.
  - response: [Scalar], an array of scalars for all variables
    (presentation + range proof)

Parameters:
- G: Group
- generatorG: Element, equivalent to G.GeneratorG()
- generatorH: Element, equivalent to G.GeneratorH()
- contextString: public input

def MakePresentationProof(U, UPrimeCommit, m1Commit, tag, generatorT,
  credential, V, r, z, nonce, nonceBlinding, nonceCommit,
  presentationLimit, rng):
```

```
  statement = LinearRelation(G)
  [m1Var, zVar, rNegVar, nonceVar, nonceBlindingVar] = statement.allocate_scalars(5)

  [genGVar, genHVar, UVar, UPrimeCommitVar, m1CommitVar, VVar, X1Var,
    tagVar, genTVar, nonceCommitVar] = statement.allocate_elements(10)
  statement.set_elements([(genGVar, generatorG),
    (genHVar, generatorH), (UVar, U),
    (UPrimeCommitVar, UPrimeCommit), (m1CommitVar, m1Commit),
    (VVar, V), (X1Var, credential.X1), (tagVar, tag),
    (genTVar, generatorT), (nonceCommitVar, nonceCommit)])

  # 1. m1Commit = m1 * U + z * generatorH
  statement.append_equation(m1CommitVar,
    [(m1Var, UVar),(zVar, genHVar)])
  # 2. V = z * X1 - r * generatorG
  statement.append_equation(VVar, [(zVar, X1Var), (rNegVar, genGVar)])
  # 3. nonceCommit = nonce * generatorG + nonceBlinding * generatorH
  statement.append_equation(nonceCommitVar,
    [(nonceVar, genGVar), (nonceBlindingVar, genHVar)])
  # 4. T = m1 * tag + nonce * tag, where
  #    T = G.HashToGroup(presentationContext, "Tag")
  statement.append_equation(genTVar,
    [(m1Var, tagVar), (nonceVar, tagVar)])
  # 5. Add range proof constraints
  (statement, D, rangeWitness) = MakeRangeProofHelper(statement, nonce,
    nonceBlinding, presentationLimit, genGVar, genHVar, rng)

  # Build witness vector matching the scalar allocations
  witness = [credential.m1, z, -r, nonce, nonceBlinding]
  witness.extend(rangeWitness)

  session_id = contextString + "CredentialPresentation"
  prover = NISchnorrProofShake128P256(session_id, statement)
  return (prover.prove(witness, rng), D)
```

Where the presentation proof returned is a serialization of the following: ~~~ - challenge: Scalar, the challenge used in the proof of valid presentation. - response0: Scalar, the response corresponding to m1. - response1: Scalar, the response corresponding to z. - response2: Scalar, the response corresponding to -r. - response3: Scalar, the response corresponding to nonce. ~~~

5.3.2. Presentation Proof Verification

```
validity = VerifyPresentationProof(serverPrivateKey,
  serverPublicKey, requestContext, presentationContext,
  presentation, presentationLimit)

Inputs:
- serverPrivateKey:
  - x0: Scalar (private), server private key 0.
  - x1: Scalar (private), server private key 1.
  - x2: Scalar (private), server private key 2.
  - x0Blinding: Scalar (private), blinding value for x0.
- serverPublicKey:
  - X0: Element, server public key 0.
  - X1: Element, server public key 1.
  - X2: Element, server public key 2.
- requestContext: Data, context for the credential request.
- presentationContext: Data (public), used for presentation tag
  computation.
- presentation:
  - U: Element, re-randomized from the U in the response.
  - UPrimeCommit: Element, a public key to the issued UPrime.
  - m1Commit: Element, a public key to the client secret (m1).
  - tag: Element, the tag element used for enforcing the
    presentation limit.
  - nonceCommit: Element, a Pedersen commitment to the nonce.
  - D: [Element], array of commitments to the bit decomposition of
    nonceCommit
  - presentationProof: ZKProof, a joint proof covering both
    presentation and range proof
    - challenge: Scalar, the challenge used in the proof of valid
      presentation.
    - response: [Scalar], an array of scalars for all variables
      (presentation + range proof)
- presentationLimit: Integer, the fixed presentation limit.

Outputs:
- validity: Boolean, True if the proof verifies correctly,
  False otherwise.

Parameters:
- G: Group
- generatorG: Element, equivalent to G.GeneratorG()
- generatorH: Element, equivalent to G.GeneratorH()
- contextString: public input

def VerifyPresentationProof(serverPrivateKey, serverPublicKey,
  requestContext, presentationContext, presentation,
  presentationLimit):
```

```
   m2 = G.HashToScalar(requestContext, "requestContext")
   V = serverPrivateKey.x0 * presentation.U + serverPrivateKey.x1 *
     presentation.m1Commit + serverPrivateKey.x2 * m2 *
     presentation.U - presentation.UPrimeCommit
   generatorT = G.HashToGroup(presentationContext, "Tag")

   statement = LinearRelation(G)
   [m1Var, zVar, rNegVar, nonceVar, nonceBlindingVar] =
     statement.allocate_scalars(5)

   [genGVar, genHVar, UVar, UPrimeCommitVar, m1CommitVar, VVar, X1Var,
     tagVar, genTVar, nonceCommitVar] = statement.allocate_elements(10)
   statement.set_elements([(genGVar, generatorG),
     (genHVar, generatorH), (UVar, presentation.U),
     (UPrimeCommitVar, presentation.UPrimeCommit),
     (m1CommitVar, presentation.m1Commit), (VVar, V),
     (X1Var, serverPublicKey.X1), (tagVar, presentation.tag),
     (genTVar, generatorT), (nonceCommitVar, presentation.nonceCommit)])

   # 1. m1Commit = m1 * U + z * generatorH
   statement.append_equation(m1CommitVar,
     [(m1Var, UVar),(zVar, genHVar)])
   # 2. V = z * X1 - r * generatorG
   statement.append_equation(VVar,
     [(zVar, X1Var), (rNegVar, genGVar)])
   # 3. nonceCommit = nonce * generatorG + nonceBlinding * generatorH
   statement.append_equation(nonceCommitVar,
     [(nonceVar, genGVar), (nonceBlindingVar, genHVar)])
   # 4. G.HashToGroup(presentationContext, "Tag")
   #    = m1 * tag + nonce * tag
   statement.append_equation(genTVar,
     [(m1Var, tagVar), (nonceVar, tagVar)])
   # 5. Add range proof constraints and verify the sum of the
   #    nonceCommit bit commitments
   (statement, sumValid) = VerifyRangeProofHelper(statement,
     presentation.proof.D, presentation.nonceCommit,
     presentationLimit, genGVar, genHVar)

   # Verify the joint proof
   session_id = contextString + "CredentialPresentation"
   verifier = NISchnorrProofShake128P256(session_id, statement)
   proofValid = sumValid and verifier.verify(
     presentation.presentationProof)
   return proofValid
```

5.4. Range Proof for Arbitrary Values

This section specifies a range proof to prove a secret value nonce lies in an arbitrary interval [0, presentationLimit). Before specifying the proof system, we first give a brief overview of how it works. For simplicity, assume that presentationLimit is a power of two, that is, presentationLimit = 2^k for some integer k > 0.

To prove a value lies in [0,(2^k)-1), we prove it has a valid k-bit representation. This is proven by committing to the full value nonce, then all bits of the bit decomposition b of the value nonce, and then proving each coefficient of the bit decomposition is actually 0 or 1 and that the sum of the bits multiplied by their associated bases equals the full value nonce. This involves the following steps:

1. Commit to the bits of nonce. That is, for each bit b[i] of the k-bit decomposition of nonce, let D[i] = b[i] * generatorG + s[i] * generatorH, where s[i] is a blinding scalar.

2. Prove that b[i] is in {0,1} by proving the algebraic relation b[i] * (b[i]-1) == 0 holds. This quadratic relation can be linearized by adding an auxilary witness s2[i] and adding the linear relation D[i] = b[i] * D[i] + s2[i] * generatorH to the equation system. A valid witness s2[i] can only be computed by the prover if b[i] is in {0,1}, and is computed as s2[i] = (1 - b[i]) * s[i]. Successfully computing a witness for any other value, while satisfying the linear relation constraints, requires the prover to break the discrete logarithm problem.

3. In addition to verifying the proof of the above relation, the verifier checks that the sum of the bit commitments is equal to the sum of the commitment to nonce:

```
nonceCommit = D[0] * 2^0 + D[1] * 2^1 + D[2] * 2^2 + ... + D[k-1] * 2^{k-1}
```

The third step is verified outside of the proof by adding the commitments homomorphically.

To support the general case, where presentationLimit is not necessarily a power of two, we extend the range proof for arbitrary ranges by decomposing the range up to the second highest power of two and adding an additional, non-binary range that covers the remaining range. This is detailed in ComputeBases below.

```
bases = ComputeBases(presentationLimit)

Inputs:
- presentationLimit: Integer, the maximum value of the range (exclusive).

Outputs:
- bases: an array of Scalar bases to represent elements, sorted in descending order. A ba
se is
  either a power of two or a unique remainder that can be used to represent any integer
  in [0, presentationLimit).

def ComputeBases(presentationLimit):
  # compute bases to express the commitment as a linear combination of the bit decomposit
ion
  remainder = presentationLimit
  bases = []
  k = ceil(log2(presentationLimit))
  # Generate all but the last power-of-two base.
  for i in range(k - 1):
      base = 2 ** i
      remainder -= base
      bases.append(base)
  bases.append(remainder - 1) # add non-binary base to close the gap

  # call sorted on array to ensure the additional base is in correct order
  return sorted(bases, reverse=True)
```

Note that by extending the range proof for arbitrary ranges, we are changing the bases used for decomposition and therefore introducing the potential for multiple valid decompositions of a value (the nonce). Implementations compliant with this specification MUST follow the canonical decomposition defined in Section 5.4.1.

5.4.1. Range Proof Creation

Using the bases from ComputeBases, the function MakeRangeProofHelper represents the secret nonce as a linear combination of the bases, using the resulting bit representation to generate the cryptographic commitments and witness values for the range proof. This helper function is called from within MakePresentationProof to add range proof constraints to the presentation proof statement.

```
(prover, D, rangeWitness) = MakeRangeProofHelper(prover, nonce, nonceBlinding,
                                                 presentationLimit, genGVar, genHVar, rn
g)

Inputs:
- prover: Prover statement to which constraints will be added
- nonce: Integer, the nonce value to prove is in range
- nonceBlinding: Scalar, the blinding factor for the nonce commitment
- presentationLimit: Integer, the maximum value of the range (exclusive).
```

```
- genGVar: Integer, variable index for generator G
- genHVar: Integer, variable index for generator H
- rng: a cryptographically secure random number generator, as defined
  in the Sigma Protocols specification.

Outputs:
- prover: Modified prover statement with range proof constraints added
- D: [Element], array of commitments to the bit decomposition of nonceCommit
- rangeWitness: [Scalar], witness values for the range proof scalars, in the
  order they were allocated: b[0], ..., b[n-1], s[0], ..., s[n-1],
  s2[0], ..., s2[n-1]

Parameters:
- G: Group
- generatorG: Element, equivalent to G.GeneratorG()
- generatorH: Element, equivalent to G.GeneratorH()

def MakeRangeProofHelper(statement, nonce, nonceBlinding, presentationLimit,
                         genGVar, genHVar):

  # Compute bit decomposition and commitments
  bases = ComputeBases(presentationLimit)

  # Compute bit decomposition of nonce
  b = []
  remainder = nonce
  # must run in constant-time (branching depends on secret value)
  for base in bases:
    bitValue = 1 if (remainder >= base) else 0
    remainder -= bitValue * base
    b.append(G.Scalar(bitValue))

  # Compute commitments to bits
  D = []
  s = []
  s2 = []
  partial_sum = G.Scalar(0)
  for i in range(len(bases) - 1):
    s.append(G.RandomScalar())
    partial_sum += bases[i] * s[i]
    s2.append((G.Scalar(1) - b[i]) * s[i])
    D.append(b[i] * generatorG + s[i] * generatorH)
  # Blinding value for the last bit commitment is chosen strategically
  # so that all the bit commitments will sum up to nonceCommit.
  idx = len(bases) - 1
  s.append(G.ScalarInverse(bases[idx]) * (nonceBlinding - partial_sum))
  s2.append((G.Scalar(1) - b[idx]) * s[idx])
  D.append(b[idx] * generatorG + s[idx] * generatorH)
```

```
  # Allocate scalar variables (b, s, s2 for each bit)
  num_bits = len(b)
  vars_b = statement.allocate_scalars(num_bits)
  vars_s = statement.allocate_scalars(num_bits)
  vars_s2 = statement.allocate_scalars(num_bits)

  # Allocate and set element variables for bit commitments D
  vars_D = statement.allocate_elements(num_bits)
  statement.set_elements([(vars_D[i], D[i]) for i in range(num_bits)])

  # Add constraints proving each b[i] is in {0,1}
  for i in range(len(b)):
    # D[i] = b[i] * generatorG + s[i] * generatorH
    statement.append_equation(vars_D[i], [(vars_b[i], genGVar), (vars_s[i], genHVar)])
    # D[i] = b[i] * D[i] + s2[i] * generatorH (proves b[i] is in {0,1})
    statement.append_equation(vars_D[i], [(vars_b[i], vars_D[i]), (vars_s2[i], genHVar)])

  # Build witness array: all b values, then all s values, then all s2 values
  rangeWitness = []
  rangeWitness.extend(b)
  rangeWitness.extend(s)
  rangeWitness.extend(s2)

  return (statement, D, rangeWitness)
```

5.4.2. Range Proof Verification

```
(verifier, sumValid) = VerifyRangeProofHelper(verifier, D, nonceCommit, presentationLimit
,
                                              genGVar, genHVar)
```

Inputs:
- verifier: Verifier statement to which constraints will be added
- D: [Element], array of commitments to the bit decomposition of nonceCommit
- nonceCommit: Element, the Pedersen commitment to the nonce
- presentationLimit: Integer, the maximum value of the range (exclusive).
- genGVar: Integer, variable index for generator G
- genHVar: Integer, variable index for generator H

Outputs:
- verifier: Modified verifier statement with range proof constraints added
- validity: Boolean, True if sum(bases[i] * D[i]) == nonceCommit, False otherwise

Parameters:
- G: Group
- generatorG: Element, equivalent to G.GeneratorG()
- generatorH: Element, equivalent to G.GeneratorH()

```
def VerifyRangeProofHelper(statement, D, nonceCommit, presentationLimit,
```

```
                            genGVar, genHVar):

  bases = ComputeBases(presentationLimit)
  num_bits = len(bases)

  # Allocate scalar variables (b, s, s2 for each bit)
  vars_b = statement.allocate_scalars(num_bits)
  vars_s = statement.allocate_scalars(num_bits)
  vars_s2 = statement.allocate_scalars(num_bits)

  # Allocate and set element variables for bit commitments D
  vars_D = statement.allocate_elements(num_bits)
  statement.set_elements([(vars_D[i], D[i]) for i in range(num_bits)])

  # Add constraints proving each b[i] is in {0,1}
  for i in range(num_bits):
    # D[i] = b[i] * generatorG + s[i] * generatorH
    statement.append_equation(vars_D[i], [(vars_b[i], genGVar), (vars_s[i], genHVar)])
    # D[i] = b[i] * D[i] + s2[i] * generatorH
    statement.append_equation(vars_D[i], [(vars_b[i], vars_D[i]), (vars_s2[i], genHVar)])

  # Verify the sum check: nonceCommit == sum(bases[i] * D[i])
  # This is done explicitly by computing the sum homomorphically
  sum_D = G.Identity()
  for i in range(len(bases)):
    sum_D = sum_D + bases[i] * D[i]

  sumValid = (sum_D == nonceCommit)
  return (statement, sumValid)
```

6. Ciphersuites

A ciphersuite (also referred to as 'suite' in this document) for the protocol wraps the functionality required for the protocol to take place. The ciphersuite should be available to both the client and server, and agreement on the specific instantiation is assumed throughout.

A ciphersuite contains an instantiation of the following functionality:

* Group: A prime-order Group exposing the API detailed in Section 3.1, with the generator element defined in the corresponding reference for each group. Each group also specifies HashToGroup, HashToScalar, and serialization functionalities. For HashToGroup, the domain separation tag (DST) is constructed in accordance with the recommendations in [I-D.irtf-cfrg-hash-to-curve], Section 3.1. For HashToScalar, each group specifies an integer order that is used in reducing integer values to a member of the corresponding scalar field.

This section includes an initial set of ciphersuites with supported groups. It also includes implementation details for each ciphersuite, focusing on input validation.

6.1. ARC(P-256)

This ciphersuite uses P-256 [NISTCurves] for the Group. The value of the ciphersuite identifier is "P256". The value of contextString is "ARCV1-P256".

* Group: P-256 (secp256r1) [NISTCurves]

  - Order(): Return 0xffffffff00000000ffffffffffffffffbce6faada7179e84f3b9cac2fc632551.

  - Identity(): As defined in [NISTCurves].

  - Generator(): As defined in [NISTCurves].

  - RandomScalar(): Implemented by returning a uniformly random Scalar in the range [1, G.Order() - 1]. Refer to Section 6.2 for implementation guidance.

  - HashToGroup(x, info): Use hash_to_curve with suite P256_XMD:SHA-256_SSWU_RO_ [I-D.irtf-cfrg-hash-to-curve], input x, and DST = "HashToGroup-" || contextString || info.

  - HashToScalar(x, info): Use hash_to_field from [I-D.irtf-cfrg-hash-to-curve] using L = 48, expand_message_xmd with SHA-256, input x and DST = "HashToScalar-" || contextString || info, and prime modulus equal to Group.Order().

  - ScalarInverse(s): Returns the multiplicative inverse of input Scalar s mod Group.Order().

- SerializeElement(A): Implemented using the compressed Elliptic-Curve-Point-to-Octet-String method according to [SEC1]; Ne = 33.

- DeserializeElement(buf): Implemented by attempting to deserialize a 33-byte array to a public key using the compressed Octet-String-to-Elliptic-Curve-Point method according to [SEC1], and then performs partial public-key validation as defined in section 5.6.2.3.4 of [KEYAGREEMENT]. This includes checking that the coordinates of the resulting point are in the correct range, that the point is on the curve, and that the point is not the point at infinity. Additionally, this function validates that the resulting element is not the group identity element. If these checks fail, deserialization returns an InputValidationError error.

- SerializeScalar(s): Implemented using the Field-Element-to-Octet-String conversion according to [SEC1]; Ns = 32.

- DeserializeScalar(buf): Implemented by attempting to deserialize a Scalar from a 32-byte string using Octet-String-to-Field-Element from [SEC1]. This function can fail if the input does not represent a Scalar in the range [1, G.Order() - 1].

6.2. Random Scalar Generation

Two popular algorithms for generating a random integer uniformly distributed in the range [1, G.Order() -1] are as follows:

6.2.1. Rejection Sampling

Generate a random byte array with Ns bytes, and attempt to map to a Scalar by calling DeserializeScalar in constant time. If it succeeds and is non-zero, return the result. Otherwise, try again with another random byte array, until the procedure succeeds. Failure to implement DeserializeScalar in constant time can leak information about the underlying corresponding Scalar.

As an optimization, if the group order is very close to a power of 2, it is acceptable to omit the rejection test completely. In particular, if the group order is p, and there is an integer b such that |p - 2^b| is less than 2^(b/2), then RandomScalar can simply return a uniformly random integer of at most b bits.

6.2.2. Random Number Generation Using Extra Random Bits

Generate a random byte array with L = ceil(((3 * ceil(log2(G.Order()))) / 2) / 8) bytes, and interpret it as an integer; reduce the integer modulo G.Order() and return the result. See [I-D.irtf-cfrg-hash-to-curve], Section 5 for the underlying derivation of L.

7. Security Considerations

For arguments about correctness, unforgeability, anonymity, and blind issuance of the ARC protocol, see the "Formal Security Definitions for Keyed-Verification Anonymous Credentials" in [KVAC].

This section elaborates on unlinkability properties for ARC and other implementation details necessary for these properties to hold.

7.1. Credential Request Unlinkability

Client credential requests are constructed such that the server cannot distinguish between any two credential requests from the same client and two requests from different clients. We refer to this property as issuance unlinkability. This property is achieved by the way the credential requests are constructed. In particular, each credential request consists of two Pedersen commitments with fresh blinding factors, which are used to commit to a freshly generated client secret and request context. The resulting request is therefore statistically hiding, and independent from other requests from the same client. More details about this unlinkability property can be found in [KVAC] and [REVISITING_KVAC].

7.2. Credential Issuance Unlinkability

The server commitment to x0 is defined as X0 = x0 * G.generatorG() + x0Blinding * G.GeneratorH(), following the definitions in [KVAC]. This is computationally binding to the secret key x0. This means that unless the discrete log is broken, the credentials issued under one server commitment X0, X1, ... will all be issued under the same private keys x0, x1, ...

However, an adversary breaking the discrete log (e.g., a quantum adversary) can find pairs (x0, x0Blinding) and (x0', x0Blinding') both committing to X0 and use them to issue different credentials. This capability would let the adversary partitioning the client anonymity set by linking clients to the underlying secret used for credential issuance, i.e., x0 or x0'. This requires an active attack and therefore is not an immediate concern.

Statistical anonymity is possible by committing to x0 and x0Blinding separately, as in [REVISITING_KVAC]. However, the security of this construction requires additional analysis.

7.3. Presentation Unlinkability

Client credential presentations are constructed so that all presentations are indistinguishable, even if coming from the same user. We refer to this property as presentation unlinkability. This property is achieved by the way the credential presentations are constructed. The presentation elements [U, UPrimeCommit, m1Commit] are indistinguishable from all other presentations made from credentials issued with the same server keys, as detailed in [KVAC].

The indistinguishability set for these presentation elements is sum_{i=0}^c(p_i), where c is the number of credentials issued with the same server keys, and p_i is the number of presentations made for each of those credentials.

The presentation elements [tag, nonceCommit, presentationContext, presentationProof, rangeProof] are indistinguishable from all presentations made from credentials issued with the same server keys for that presentationContext. The nonce is never revealed to the server since it is hidden within a Pedersen commitment. The range proof ensures the committed nonce is within the valid range [0, presentationLimit) without revealing its value. This provides strong unlinkability properties: the server cannot link presentations based on nonce values, as the nonce commitment uses a fresh random blinding factor for each presentation.

The indistinguishability set for these presentation elements is sum_{i=0}^c(p_i[presentationContext]), where c is the number of credentials issued with the same server keys and p_i[presentationContext] is the number of presentations made for each of those credentials with the same presentationContext. Unlike protocols where nonces are revealed, presentations can not be linked by comparing nonce values, resulting in maximum unlinkability within the presentation context.

7.4. Timing Leaks

To ensure no information is leaked during protocol execution, all operations that use secret data MUST run in constant time. This includes all prime-order group operations and proof-specific operations that operate on secret data, including proof generation and verification.

8. Alternatives considered

ARC uses the MACGGM algebraic MAC as its underlying primitive, as detailed in [KVAC] and [REVISITING_KVAC]. This offers the benefit of having a lower credential size than MACDDH, which is an alternative algebraic MAC detailed in [KVAC].

The BBS anonymous credential scheme, as detailed in [BBS] and its variants, is efficient and publicly verifiable, but requires pairings for verification. This is problematic for adoption because pairings are not supported as widely in software and hardware as non-pairing elliptic curves.

It is possible to construct a keyed-verification variant of BBS which doesn't use pairings, as discussed in [BBDT17] and [REVISITING_KVAC]. However these keyed-verification BBS variants require more analysis, proofs of security properties, and review to be considered mature enough for safe deployment.

9. IANA Considerations

This document has no IANA actions.

10. Test Vectors

10.1. Seeded PRNG

For interoperability, the random number generator used for test vectors is implemented using the SeededPRNG defined in Section A.1 of [SIGMA].

The following sections contain test vectors for the ARC ciphersuites specified in this document.

10.2. ARCV1-P256

```
// ServerKey
x0 = 1008f2c706ae2157c75e41b2d75695c7bf480d0632a1ef447036cafe4cabb02
1
x1 = 526e009578f6f25fdec992343f09f5e6c58489c31fcf8a934bbaf85797121bd
d
x2 = 549075ccd3d1c36b3546725c43e71943414409a23b980b2c47a3fc2b9c37679
b
xb = 7276533ce3c89f04a007c2e8aa7d2e3b36829d0eaab5631347d8336c2da09a8
e
X0 = 03bad54cc48293ef3472ac1ada55c9c9fdb3eb99ee47369bbe1d3ce46b300cd
7b3
X1 = 02a0323862a05707d76862bfa8477eed468441ceae14c8fb1659e0b3020b8a2
```

```
   4e1
   X2 = 031d16ef08ede5a347e94a8eca071bec7bedb9d8ba943d24bde912a4e1578e5
   29b

   // CredentialRequest
   request_context = 7465737420726571756573742063 6f6e74657874
   m1 = 141c4ca5e614af8e5e323eb47a7e7673ebb67caf49dfa8e109f45f231227f7a
   0
   m2 = 911fb315257d9ae29d47ecb48c6fa27074dee6860a0489f8db6ac9a486be6a3
   e
   r1 = 5c183d2dea942eb2780afb90cfd94983ae6575d60e350021c8c93008ac50397
   3
   r2 = 044d4a5b5daf00dd1fb4444ca2f8c3facc95d537d5ad0e0a2815c912e98a431
   d
   m1_enc = 033fe5d950712f711e5d292d68f804fad4c35fb7f3f1866516448647d4a
   ab12590
   m2_enc = 026502a833ed1d972ee27175e750b1719adee12726c653125887c0d32b1
   f3747ab
   proof = 2a088673e302502a3dc80d6100a1bb709083ac7b31da34f9a7c52e7cfeaa
   2ea30b7341133086e64b79dfc6cdac9f348ddbed0b087746f0167ea238d3ddf17e61
   3880b73e85f499c7eddc6555355ea71487b49862400091b5b32cb219d7104f571306
   bc6f2487bab299bb2e9a1078dee94d83b6536ed570f8114ee9c97b8b602bfacbeb37
   64f6a22915a19c24895a6bf7048c663337f7690f0182a1f866586d9e

   // CredentialResponse
   b = 9ac9d836ef405f4c6c1de4de18d210c929a8dc786c95e3eac3a828cc19e1636e
   U = 021cf52318c97c33472cc8fb42a5b5a774f83c3b36e6c782209d53e5945d99a4
   93
   enc_U_prime = 02ae23020d5427c7f785a72d77c24997f955e66ab7c378c334b7c2
   59dabdf572d7
   X0_aux = 031523abe64e436e65e592abdae322dc556fcbea707757e18d4160ba57d
   574cd87
   X1_aux = 023cc3b53807f6e0082b675794ae9f6b370483ca5a3e6d688c3b81f2fdb
   6d4ec00
   X2_aux = 0329dc7c93f8a231a1f16ec69f0fba446e022ce69945b20f37386a7fda3
   e573b79
   H_aux = 0389746891b6dbf062511619eae7d72ae87630bea1e277a925708fdfef83
   63a1d4
   proof = ec342aee0d481435379ea6bbe919edd5d2eb9c12198a083e0e899da1f14d
   bc46a8048f5a12c5cae21e5f5949fe08d1c15c266c63544615400def4ce9a6cf8aee
   32052ced26e7a9d854f2c45ea23ffea0f6bf977f6155d412991abc0e2d1ad8350412
   9c1ac8319b2a45940c52c4b41bde80969313641b9cb727445e20b44d0ea884e9b180
   cd152442883038b97d72772201f281d76a18d22e374bd989accd7654806739916242
   8c4d25daf1b7f68f3580a38cc4564a88f28494649064500f06c5b946dde032a389f8
   fe337605627ce91a92c20db911100a2c7c42ae15fde5a5cbd9d078b819a804235931
   92c40d70ce77f1a6d377770fe5c05781782bd1eaa43f

   // Credential
```

```
m1 = 141c4ca5e614af8e5e323eb47a7e7673ebb67caf49dfa8e109f45f231227f7a
0
U = 021cf52318c97c33472cc8fb42a5b5a774f83c3b36e6c782209d53e5945d99a4
93
U_prime = 02646199272c28911165b4d1c5f4ffbd8a83f686948fd4c7250e28c81d
bfecd354
X1 = 02a0323862a05707d76862bfa8477eed468441ceae14c8fb1659e0b3020b8a2
4e1

// Presentation1
presentation_context = 7465737420707265736e746174696f6e20636f6e746
57874
a = a3c469d2d55062b463b17f45acd2fb17c038b18df4c8d6c9c745866ba961de9a
r = 54925f70e9ec2128114c6ae8bfc6e1a2914a8fdd383e5ff03d8c2992edd081a9
z = 9ed3c3ddb1b5ff64e55b1d0a4f58eee67351fea7de16baf644e80f4d0949cf5f
U = 0216af8901c1ad38a703bf9003fabea440b411b4f072fd23b5254cb17d1b5bf3
3d
U_prime_commit = 03140f8e6f6c5eab3d03a7fba5d542362a9bc00a89d80caa505
1b4e4446b0b01f3
m1_commit = 0214d0297c21120d621cc6fed75852569de3cbf0bd9f5a8a812cf6b0
24bf51e627
nonce = 0x0
nonce_blinding = ed8b643e3c8ba74cc417b5bbfc4f42bee6dcd2d6c5ea48fb227
3afec7b6505a4
nonce_commit = 032326abcd4eb2fd1a47053ec9ce1aab3ee91e98373d610e9752a
7d16a5c1e38d8
tag = 0281428e61688f4e7989dbe8dab170705c81b294c4a73b785a0754712fc968
eb40
D_0 = 032326abcd4eb2fd1a47053ec9ce1aab3ee91e98373d610e9752a7d16a5c1e
38d8
proof = 032326abcd4eb2fd1a47053ec9ce1aab3ee91e98373d610e9752a7d16a5c
1e38d8946f5f0b44e34f826b41ec59a4e2dcfaa826b8a39cc278e10b1b02b5dbaafd
b6e789639885a8d2d69269a9fea55830f1d7e1fd0a771183b7b4eebe5e03e0c0255d
1ba614de7e31d4f46eb93a24e0ffe9864b002527109a516a10dc1ad718b8d984efd1
6ab245d7a5dfabe2d0027e23796981422b19c2821a831cb46a8e9b8b566bbdb55b64
9021bf2f777b9130c2e375f560eee4691d04bd38e9571d9451257858d9128002a2f8
908d7e4521510a2185244fa533e2502b61e502fd157d974f91acc4f2ba0d724f2bfd
182d5df4d038e74b5c35cc7c4aa7622c2682e040877eebcc18fe822cc6abab5d3adc
9db836991d3d1ecf699658245b8b0756946ba0d7756b433aae3b476ccbc2186b2fe2
ecc2fe0da30df264802829254df8196a8307f0

// Presentation2
presentation_context = 7465737420707265736e746174696f6e20636f6e746
57874
a = ae14ddaf96907f2fee72069664e1883fee4582cefcfbb2f3fae380c317018ab2
r = f994bf66d0c7943ce97331da186e231281b691eb271c7c524ff9f8bc7804b41d
z = fa27efc5066bca91121642d629477eb1c7812fa9c473b30dea3eaba8a1731568
U = 0357e53851143e7cc34311bdba0d44d4d3c9192180434ce247b8766232b5de1e
```

```
08
U_prime_commit = 02bad8dc9b0179dff7a1d63d03d92810520085cbc41b65b667d
3cbe2203eb7c544
m1_commit = 02455589d2b92a24e49ff8c2e8287f6eeb05cbfddc16aba66dfe9ab9
7702bc3c35
nonce = 0x1
nonce_blinding = 90b8387fe4145c2d47a0f042c26119939bcbcc8c2c32f81d103
4db3958b9af39
nonce_commit = 0363d6bd2969b64a42354ba896be33a4abce479261d7dec0001fa
1af7fbdeecb41
tag = 02ad6c293325d0c2c388c8b2240b6d8ab9e52395297ef5921fb78ace6a1274
b03b
D_0 = 0363d6bd2969b64a42354ba896be33a4abce479261d7dec0001fa1af7fbdee
cb41
proof = 0363d6bd2969b64a42354ba896be33a4abce479261d7dec0001fa1af7fbd
eecb417c59300e0aafb0d58e3f85423030401dabe5dd39566924f07e99cae5b3be62
f45e736890857cfe0950c22c93c52d56e6ade5f5a1c1d486e9261e7788b745438701
15370c46e62e376b17844a287bbb6722dc5e5848fdbd8d19d259ec1cec83851a1ef4
a21dadefeef3f5222eb19361facbb2ec3ba640aef22cd5700a17ea17fc3ece772f5b
5cca1e119bf32cfa7f3459c2184d6d8c777d281c91b416187ee949c9557fece8afb0
ac785c7b8c4854e622f8b005daf5c0682cfdc2d900150087bae0090d44b7bac130c7
f4067bb8b3374b159106d0c03e30f9577063de0b52d15d1f5f41328b335ee23d10ca
7dc4e0717bd9e919e3f6f580e594b3c48b358baa7a320ec9e1019260efe2cbf6e9cb
a6871ffee3b5566d5e865c729c5e1d48529559
```

11. Acknowledgments

The authors would like to acknowledge helpful conversations with Tommy Pauly about rate limiting and Privacy Pass integration, as well as Lena Heimberger for specifying the range proof.

12. References

12.1. Normative References

[FIAT-SHAMIR]
Orr湛, M., "Fiat-Shamir Transformation", Work in Progress, Internet-Draft, draft-irtf-cfrg-fiat-shamir-01, 20 October 2025, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-fiat-shamir-01>.

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

[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, DOI 10.6028/nist.sp.800-56ar3, April 2018, <https://doi.org/10.6028/nist.sp.800-56ar3>.

[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>.

[SIGMA] Orr湛, M. and C. Yun, "Interactive Sigma Proofs", Work in Progress, Internet-Draft, draft-irtf-cfrg-sigma-protocols-01, 20 October 2025, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-sigma-protocols-01>.

12.2. Informative References

[BBDT17] "Improved Algebraic MACs and Practical Keyed-Verification Anonymous Credentials", n.d., <https://link.springer.com/chapter/10.1007/978-3-319-69453-5_20>.

[BBS] "Short Group Signatures", n.d., <https://eprint.iacr.org/2004/174>.

[BLIND-RSA]
Denis, F., Jacobs, F., and C. A. Wood, "RSA Blind Signatures", RFC 9474, DOI 10.17487/RFC9474, October 2023, <https://www.rfc-editor.org/rfc/rfc9474>.

[KVAC] "Keyed-Verification Anonymous Credentials from Algebraic MACs", n.d., <https://eprint.iacr.org/2013/516>.

[NISTCurves]
"Digital Signature Standard (DSS)", National Institute of Standards and Technology (U.S.), DOI 10.6028/nist.fips.186-5, February 2023, <https://doi.org/10.6028/nist.fips.186-5>.

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

[REVISITING_KVAC]
"Revisiting Keyed-Verification Anonymous Credentials", n.d., <https://eprint.iacr.org/2024/1552>.

[SEC1] Standards for Efficient Cryptography Group (SECG), "SEC 1: Elliptic Curve Cryptography", <https://www.secg.org/sec1-v2.pdf>.

Authors' Addresses

Cathie Yun
Apple, Inc.
Email: cathieyun@gmail.com

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

Armando Faz-Hernandez
Cloudflare
Email: armfazh@cloudflare.com