Network Working Group
Internet-Draft
Intended status: Informational
Expires: 30 July 2026

C. Yun
C. A. Wood
Apple, Inc.

26 January 2026

Anonymous Rate-Limited Credentials Cryptography
draft-privacypass-arc-crypto-00

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-privacypass-arc-crypto.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-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

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 keys 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 are 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)

Inputs:
- requestContext: Data, context for the credential request

Outputs:
- request:
  - m1Enc: Element, first encrypted secret.
  - m2Enc: Element, second encrypted secret.
  - requestProof: 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):
  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)
  request = (m1Enc, m2Enc, requestProof)
  clientSecrets = (m1, m2, r1, r2)
  return (clientSecrets, request)
```

See Section 5.2 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 challenge[Ns];
  uint8 response0[Ns];
  uint8 response1[Ns];
  uint8 response2[Ns];
  uint8 response3[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)

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: ZKProof, a proof of correct generation of m1Enc and m2Enc.

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: 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(serverPrivateKeys, serverPublicKey, request):
  if VerifyCredentialRequestProof(request) == false:
    raise VerifyError

  b = G.RandomScalar()
```

```
  U = b * generatorG
  encUPrime = b * (serverPublicKey.X0 +
        serverPrivateKeys.x1 * request.m1Enc +
        serverPrivateKeys.x2 * request.m2Enc)
  X0Aux = b * serverPrivateKeys.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)
  return (U, encUPrime, X0Aux, X1Aux, X2Aux, HAux, responseProof)
```

The resulting response can be serialized as follows. See Section 5.3 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 challenge[Ns];
  uint8 response0[Ns];
  uint8 response1[Ns];
  uint8 response2[Ns];
  uint8 response3[Ns];
  uint8 response4[Ns];
  uint8 response5[Ns];
  uint8 response6[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: 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: 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 secret secre
ts.
  - 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 - clie
ntSecrets.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.

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

Details for each 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.
- presentationNonceSet: {Integer}, the set of nonces that have been used for this present
ation
- presentationLimit: Integer, the fixed presentation limit.

def MakePresentationState(credential, presentationContext, presentationLimit):
  return PresentationState(credential, presentationContext, [], 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)

Inputs:
state: input PresentationState
  - credential
  - presentationContext: Data (public), used for presentation tag computation.
  - presentationNonceSet: {Integer}, the set of nonces that have been used for this prese
ntation
  - presentationLimit: Integer, the fixed presentation limit.

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.
  - presentationProof: ZKProof, a proof of correct generation of the presentation.

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 presentatio
n limit for the given presentation context

def Present(state):
  if len(state.presentationNonceSet) >= state.presentationLimit:
    raise LimitExceededError

  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

  # This step mutates the state by keeping track of
  # what nonces have already been spent.
  nonce = random_integer_uniform_excluding_set(0,
    state.presentationLimit, state.presentationNonceSet)
  state.presentationNonceSet.add(nonce)

  generatorT = G.HashToGroup(presentationContext, "Tag")
  tag = (credential.m1 + nonce)^(-1) * generatorT
  V = z * credential.X1 - r * generatorG
  m1Tag = state.credential.m1 * tag

  presentationProof = MakePresentationProof(U, UPrimeCommit, m1Commit, tag, generatorT, c
redential, V, r, z, nonce, m1Tag)

  presentation = (U, UPrimeCommit, m1Commit, tag, 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 5.4 for more details on the generation of the presentation
   proof.
```

```
   struct {
     uint8 U[Ne];
     uint8 UPrimeCommit[Ne];
     uint8 m1Commit[Ne];
     uint8 tag[Ne];
     uint8 challenge[Ns];
     uint8 response0[Ns];
     uint8 response1[Ns];
     uint8 response2[Ns];
     uint8 response3[Ns];
   } Presentation
```

The length of this structure is Npresentation = 4*Ne + 5*Ns.

4.3.3. Presentation Verification

The server processes the presentation by verifying the presentation proof against server-computed values, and performing a check that the presentation conforms to the presentation limit.

```
validity, tag = VerifyPresentation(
  serverPrivateKey,
  serverPublicKey,
  requestContext,
  presentationContext,
  nonce,
  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.
- 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.
  - presentationProof: ZKProof, a proof of correct generation of the presentation.

- 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()

Exceptions:
- InvalidNonceError, raised when the nonce associated with the presentation is invalid

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

  if nonce < 0 or nonce > presentationLimit:
    raise InvalidNonceError

  generatorT = G.HashToGroup(presentationContext, "Tag")
  m1Tag = generatorT - (nonce * presentation.tag)

  validity = VerifyPresentationProof(
    serverPrivateKey,
    serverPublicKey,
    requestContext,
    presentationContext,
    presentation,
    m1Tag)

  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 describes a Schnorr proof compiler that is used for the construction of other proofs needed throughout the ARC protocol. Section 5.1 describes the compiler, and the remaining sections describe how it is used for the purposes of producing ARC proofs.

5.1. Schnorr Compiler

The compiler specified in this section automates the Fiat-Shamir transform that is often used to transform interactive zero-knowledge proofs into non-interactive proofs such that they can be used to non-interactively prove various statements of importance in higher-level protocols, such as ARC. The compiler consists of a prover and verifier role. The prover constructs a transcript for the proof and then applies the Fiat-Shamir heuristic to generate the resulting challenge and response values. The verifier reconstructs the same transcript to verify the proof.

The prover and verifier roles are specified below in Section 5.1.1 and Section 5.1.2, respectively.

5.1.1. Prover

The prover role consists of four functions:

* AppendScalar: This function adds a scalar representation to the transcript.

* AppendElement: This function adds an element representation to the transcript.

* Constrain: This function applies an explicit constraint to the proof, where the constraint is expressed as equality between some element and a linear combination of scalar and element representations. An example constraint might be Z = aX + bY, for scalars a, b, and elements X, Y, Z.

* Prove: This function applies the Fiat-Shamir heuristic to the protocol transcript and set of constraints to produce a zero-knowledge proof that can be verified.

These functions are defined in the following sub-sections.

In addition, the prover role consists of the following state:

* label: Data, a value representing the context in which the proof will be used

* scalars: [Integer], An ordered set of representation of scalar variables to use in the proof. Each scalar has a label associated with it, stored in a list called scalar_labels.

* elements: [Integer], An ordered set of representation of element variables to use in the proof. Each element has a label associated with it, stored in a list called element_labels.

* constraints: a set of constraints, where each constraint consists of a constraint element and a linear combination of variables.

5.1.1.1. AppendScalar

```
AppendScalar(label, assignment)

Inputs:
- label: Data, Scalar variable label
- assignment: Scalar variable

Outputs:
- Integer representation of the new scalar variable

def AppendScalar(label, assignment):
  state.scalars.append(assignment)
  state.scalar_labels.append(label)
  return len(state.scalars) - 1
```

5.1.1.2. AppendElement

```
AppendElement(label, assignment)

Inputs:
- label: Data, Element variable label
- assignment: Element variable

Outputs:
- Integer representation of the new element variable

def AppendElement(label, assignment):
  state.elements.append(assignment)
  state.element_labels.append(label)
  return len(state.elements) - 1
```

5.1.1.3. Constrain

```
Constrain(result, linearCombination)

Inputs:
- result: Integer, representation of constraint element
- assignment: linear combination of scalar and element variable (representations)

def Constrain(label, linearCombination):
  state.constraints.append((result, linearCombination))
```

5.1.1.4. Prove

The Prove function is defined below.

```
Prove()

Outputs:
- ZKProof, a proof consisting of a challenge Scalar and then fixed number of response Sca
lar values

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

Exceptions:
- InvalidVariableAllocationError, raised when the prover was incorrectly configured

def Prove():
  blindings = [G.RandomScalar() for i in range(len(state.scalars))]

  blinded_elements = []
  for (constraint_point, linear_combination) in state.constraints:
    if constraint_point.index > len(state.elements):
      raise InvalidVariableAllocationError

    for (scalar_var, element_var) in linear_combination:
      if scalar_var.index > len(state.scalars):
        raise InvalidVariableAllocationError
      if element_var.index > len(state.elements):
        raise InvalidVariableAllocationError

    scalar_index = linear_combination[0][0]
    element_index = linear_combination[0][1]
    blinded_element = blindings[scalar_index] * state.elements[element_index]

    for i, pair in enumerate(linear_combination):
      if i > 0:
        scalar_index = pair[0]
        element_index = pair[1]
```

```
        blinded_element += blindings[scalar_index] * state.elements[element_index]

        blinded_elements.append(blinded_element)

  # Obtain a scalar challenge
  challenge = ComposeChallenge(state.label, state.elements, blinded_elements)

  # Compute response scalars from the challenge, scalars, and blindings.
  responses = []
  for (index, scalar) in enumerate(state.scalars):
    blinding = blindings[index]
    responses.append(blinding - challenge * scalar)

  return ZKProof(challenge, responses)
```

The function ComposeChallenge is defined below.

```
ComposeChallenge(label, elements, blinded_elements)

Inputs:
- label: Data, the proof label
- elements: [Element], ordered list of elements
- blinded_elements: [Element], ordered list of blinded elements

Outputs:
- challenge, Scalar

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

def ComposeChallenge(label, elements, blinded_elements):
  challenge_input = Data() # Empty Data

  for element in elements:
    serialized_element = G.SerializeElement(element)
    challenge_input += I2OSP(len(serialized_element), 2) + serialized_element

  for blinded_element in blinded_elements:
    serialized_blinded_element = G.SerializeElement(blinded_element)
    challenge_input += I2OSP(len(serialized_blinded_element), 2) + serialized_blinded_ele
ment

  return G.HashToScalar(challenge_input, label)
```

5.1.2. Verifier

The verifier role consists of four functions:

* AppendScalar: This function adds a scalar representation to the transcript.

* AppendElement: This function adds an element representation to the transcript.

* Constrain: This function applies an explicit constraint to the proof, where the constraint is expressed as equality between some element and a linear combination of scalar and element representations. An example constraint might be Z = aX + bY, for scalars a, b, and elements X, Y, Z.

* Verify: This function applies the Fiat-Shamir heuristic to verify the zero-knowledge proof.

AppendScalar and Verify are defined in the following sub-sections. AppendElement and Constrain matches the functionality used in the prover role.

5.1.2.1. AppendScalar

AppendScalar(label)

Inputs:
- label: Data, Scalar variable label

Outputs:
- Integer representation of the new scalar variable

```
def AppendScalar(label):
  state.scalar_labels.append(label)
  return len(state.scalar_labels) - 1
```

5.1.2.2. Verify

```
Verify(proof)

Inputs:
- ZKProof, a proof consisting of a challenge Scalar and then fixed number of response Sca
lar values

Outputs:
- Boolean, True if the proof is valid, False otherwise.

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

Exceptions:
- InvalidVariableAllocationError, raised when the prover was incorrectly configured

def Verify(proof):
  if len(state.elements) != len(state.element_labels):
    raise InvalidVariableAllocationError

  blinded_elements = []
  for (constraint_element, linear_combination) in state.constraints:
    if constraint_element > len(state.elements):
      raise InvalidVariableAllocationError
    for (_, element_var) in linear_combination:
      if element_var > len(state.elements):
        raise InvalidVariableAllocationError

    challenge_element = proof.challenge * state.elements[constraint_element]
    for i, pair in enumerate(linear_combination):
      challenge_element += proof.responses[pair[0]] * state.elements[pair[1]]

    blinded_elements.append(challenge_element)

  challenge = ComposeChallenge(state.label, self.elements, blinded_elements)
  return challenge == proof.challenge
```

5.2. 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

5.2.1. CredentialRequest Proof Creation

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

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.

Outputs:
- proof: ZKProof
  - 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.

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):
  prover = Prover(contextString + "CredentialRequest")

  m1Var = prover.AppendScalar("m1", m1)
  m2Var = prover.AppendScalar("m2", m2)
  r1Var = prover.AppendScalar("r1", r1)
  r2Var = prover.AppendScalar("r2", r2)

  genGVar = prover.AppendElement("genG", generatorG)
  genHVar = prover.AppendElement("genH", generatorH)
  m1EncVar = prover.AppendElement("m1Enc", m1Enc)
  m2EncVar = prover.AppendElement("m2Enc", m2Enc)

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

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

  return prover.Prove()
```

5.2.2. CredentialRequest Proof Verification

```
validity = VerifyCredentialRequestProof(request)

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

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):
  verifier = Verifier(contextString + "CredentialRequest")

  m1Var = verifier.AppendScalar("m1")
  m2Var = verifier.AppendScalar("m2")
  r1Var = verifier.AppendScalar("r1")
  r2Var = verifier.AppendScalar("r2")

  genGVar = verifier.AppendElement("genG", generatorG)
  genHVar = verifier.AppendElement("genH", generatorH)
  m1EncVar = verifier.AppendElement("m1Enc", request.m1Enc)
  m2EncVar = verifier.AppendElement("m2Enc", request.m2Enc)

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

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

  return verifier.Verify(request.proof)
```

5.3. 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))

5.3.1. CredentialResponse Proof Creation

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

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: 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.

Outputs:
- proof: ZKProof
  - 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.

```
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, encUPri
me, X0Aux, X1Aux, X2Aux, HAux):
  prover = Prover(contextString + "CredentialResponse")

  x0Var = prover.AppendScalar("x0", serverPrivateKey.x0)
  x1Var = prover.AppendScalar("x1", serverPrivateKey.x1)
  x2Var = prover.AppendScalar("x2", serverPrivateKey.x2)
  x0BlindingVar = prover.AppendScalar("x0Blinding", serverPrivateKey.x0Blinding)
  bVar = prover.AppendScalar("b", b)
  t1Var = prover.AppendScalar("t1", b * serverPrivateKey.x1)
  t2Var = prover.AppendScalar("t2", b * serverPrivateKey.x2)

  genGVar = prover.AppendElement("genG", generatorG)
  genHVar = prover.AppendElement("genH", generatorH)
  m1EncVar = prover.AppendElement("m1Enc", request.m1Enc)
  m2EncVar = prover.AppendElement("m2Enc", request.m2Enc)
  UVar = prover.AppendElement("U", U)
  encUPrimeVar = prover.AppendElement("encUPrime", encUPrime)
  X0Var = prover.AppendElement("X0", serverPublicKey.X0)
  X1Var = prover.AppendElement("X1", serverPublicKey.X1)
  X2Var = prover.AppendElement("X2", serverPublicKey.X2)
  X0AuxVar = prover.AppendElement("X0Aux", X0Aux)
  X1AuxVar = prover.AppendElement("X1Aux", X1Aux)
  X2AuxVar = prover.AppendElement("X2Aux", X2Aux)
  HAuxVar = prover.AppendElement("HAux", HAux)

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

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

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

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

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

  # 7. U = b * generatorG
  prover.Constrain(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
  prover.Constrain(encUPrimeVar, [(bVar, X0Var), (t1Var, m1EncVar), (t2Var, m2EncVar)])

  return prover.Prove()
```

5.3.2. 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: ZKProof, a proof of correct generation of U, encUPrime, server public
keys, and auxiliary points.
    - 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.
- request:
  - m1Enc: Element, first encrypted secret.
  - m2Enc: Element, second encrypted secret.
  - requestProof: 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):
  verifier = Verifier(contextString + "CredentialResponse")

  x0Var = verifier.AppendScalar("x0")
  x1Var = verifier.AppendScalar("x1")
  x2Var = verifier.AppendScalar("x2")
  x0BlindingVar = verifier.AppendScalar("x0Blinding")
  bVar = verifier.AppendScalar("b", b)
  t1Var = verifier.AppendScalar("t1")
  t2Var = verifier.AppendScalar("t2")

  genGVar = verifier.AppendElement("genG", generatorG)
  genHVar = verifier.AppendElement("genH", generatorH)
  m1EncVar = verifier.AppendElement("m1Enc", request.m1Enc)
  m2EncVar = verifier.AppendElement("m2Enc", request.m2Enc)
  UVar = verifier.AppendElement("U", response.U)
  encUPrimeVar = verifier.AppendElement("encUPrime", response.encUPrime)
  X0Var = verifier.AppendElement("X0", serverPublicKey.X0)
  X1Var = verifier.AppendElement("X1", serverPublicKey.X1)
  X2Var = verifier.AppendElement("X2", serverPublicKey.X2)
  X0AuxVar = verifier.AppendElement("X0Aux", response.X0Aux)
  X1AuxVar = verifier.AppendElement("X1Aux", response.X1Aux)
  X2AuxVar = verifier.AppendElement("X2Aux", response.X2Aux)
  HAuxVar = verifier.AppendElement("HAux", response.HAux)

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

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

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

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

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

  # 7. U = b * generatorG
  verifier.Constrain(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
  verifier.Constrain(encUPrimeVar, [(bVar, X0Var), (t1Var, m1EncVar), (t2Var, m2EncVar)])

  return verifier.Verify(response.proof)
```

5.4. Presentation Proof

The presentation proof is a proof of knowledge of (m1, r, z) used in the presentation, and a proof that the nonce used to make the tag is in the range of [0, presentationLimit).

Statements to prove:

1. m1Commit = m1 * U + z * generatorH
2. V = z * X1 - r * generatorG
3. G.HashToGroup(presentationContext, "Tag") = m1 * tag + nonce * tag
4. m1Tag = m1 * tag

5.4.1. Presentation Proof Creation

```
presentationProof = MakePresentationProof(U, UPrimeCommit, m1Commit, tag, generatorT, cre
dential, V, r, z, nonce, m1Tag)
```

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.
- m1Tag: Element, helper element for the proof.

Outputs:
- proof: ZKProof
  - 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.

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, presentationContext
, credential, V, r, z, nonce, m1Tag)
  prover = Prover(contextString + "CredentialPresentation")

  m1Var = prover.AppendScalar("m1", credential.m1)
  zVar = prover.AppendScalar("z", z)
  rNegVar = prover.AppendScalar("-r", -r)
  nonceVar = prover.AppendScalar("nonce", nonce)

  genGVar = prover.AppendElement("genG", generatorG)
  genHVar = prover.AppendElement("genH", generatorH)
  UVar = prover.AppendElement("U", U)
  _ = prover.AppendElement("UPrimeCommit", UPrimeCommit)
  m1CommitVar = prover.AppendElement("m1Commit", m1Commit)
  VVar = prover.AppendElement("V", V)
  X1Var = prover.AppendElement("X1", credential.X1)
  tagVar = prover.AppendElement("tag", tag)
  genTVar = prover.AppendElement("genT", generatorT)
  m1TagVar = prover.AppendElement("m1Tag", m1Tag)

  # 1. m1Commit = m1 * U + z * generatorH
  prover.Constrain(m1CommitVar, [(m1Var, UVar), (zVar, genHVar)])
  # 2. V = z * X1 - r * generatorG
  prover.Constrain(VVar, [(zVar, X1Var), (rNegVar, genGVar)])
  # 3. G.HashToGroup(presentationContext, "Tag") = m1 * tag + nonce * tag
  prover.Constrain(genTVar, [(m1Var, tagVar), (nonceVar, tagVar)])
  # 4. m1Tag = m1 * tag
  prover.Constrain(m1TagVar, [(m1Var, tagVar)])

  return prover.Prove()
```

5.4.2. Presentation Proof Verification

```
validity = VerifyPresentationProof(
  serverPrivateKey,
  serverPublicKey,
  requestContext,
  presentationContext,
  presentation,
  m1Tag)

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.
  - presentationProof: ZKProof, a proof of correct generation of the presentation.
    - 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.
- m1Tag: Element, helper to validate the presentation proof.

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,
  m1Tag):

  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")

  verifier = Verifier(contextString + "CredentialPresentation")

  m1Var = verifier.AppendScalar("m1")
  zVar = verifier.AppendScalar("z")
  rNegVar = verifier.AppendScalar("-r")
  nonceVar = verifier.AppendScalar("nonce")

  genGVar = verifier.AppendElement("genG", generatorG)
  genHVar = verifier.AppendElement("genH", generatorH)
  UVar = verifier.AppendElement("U", presentation.U)
  _ = verifier.AppendElement("UPrimeCommit", presentation.UPrimeCommit)
  m1CommitVar = verifier.AppendElement("m1Commit", presentation.m1Commit)
  VVar = verifier.AppendElement("V", presentation.V)
  X1Var = verifier.AppendElement("X1", serverPublicKey.X1)
  tagVar = prover.AppendElement("tag", presentation.tag)
  genTVar = verifier.AppendElement("genT", generatorT)
  m1TagVar = prover.AppendElement("m1Tag", m1Tag)

  # 1. m1Commit = m1 * U + z * generatorH
  verifier.Constrain(m1CommitVar, [(m1Var, UVar), (zVar, genHVar)])
  # 2. V = z * X1 - r * generatorG
  verifier.Constrain(VVar, [(zVar, X1Var), (rNegVar, genGVar)])
  # 3. G.HashToGroup(presentationContext, "Tag") = m1 * tag + nonceVar * tag
  verifier.Constrain(genTVar, [(m1Var, tagVar), (nonceVar, tagVar)])
  # 4. m1Tag = m1 * tag
  prover.Constrain(m1TagVar, [(m1Var, tagVar)])

  return verifier.Verify(presentation.proof)
```

5.5. Range Proof for Arbitrary Values

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

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 v, then all bits of the bit decomposition b of the value v, and then

proving each coefficient of the bit decomposition is actually 0 or 1 and that the sum of the bits amounts to the full value v. This involves the following steps:

1. Commit to the bits of v. That is, for each bit b[i] of the bit decomposition of v, 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 computing 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}. Successfully computing a witness for any other value requires the prover to break the discrete logarithm problem.

3. Having verified the proof the above relation, the verifier checks the sum by computing

C == 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 upper_bound 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.

```
def ComputeBases(upper_bound):

Inputs:

- upper_bound: the maximum value of the range (exclusive), as integer.

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, upper_bound).

# compute bases to express the commitment as a linear combination of the bit decompositio
n
remainder=upper_bound
bases=[]
# Generate all but the last power-of-two base.
for i in range(ceil(log2(upper_bound)) - 1):
    base = 2 ** i
    remainder -= base
    bases.append((G.Scalar(base))
bases.append(remainder - 1)

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

Using the bases from ComputeBases, the function ComputeStatementAndWitnesses represents the secret value v as a linear combination of the bases, using the resulting bit representation to generate the cryptographic commitments and witness values for the range proof.

```
def ComputeStatementAndWitnesses(v, upper_bound):

Inputs:

- v: the scalar we want to prove is in range [0, upper_bound)
- r: randomness for commitment to v
- upper_bound: the maximum integer value of the range

Outputs:

- statement: proof statement for the relation
- [s,s2]: the witness for the equations appended to the statement (the bit
  decomposition, the secret shares of r, and the auxiliary witness s2. Each s2[i] is eith
er zero when
b[i] is set) or s[i] when b[i] is zero.
- C: the commitment to v
- D: the commitments to the bit decomposition of v
```

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

Exceptions:
- NumberTooBigError, raised when v is out of range

if G.ScalarToInt(v) >= upper_bound:
    raise NumberTooBigError

bases = ComputeBases(upper_bound)

# Compute bit decomposition of v.
b = []
v_remainder = G.ScalarToInt(v)
for base in bases:
    # Implementation note: In order to avoid leaking v via a timing channel, this code sh
ould be written to be constant time.
    if v_remainder >= base:
        v_remainder -= G.ScalarToInt(base)
        b.append(G.Scalar(1))
    else:
        b.append(G.Scalar(0))

# array of group elements where the i-th element corresponds to b[i] * generatorG + s * g
eneratorH
D = []
# blinding elements for Pedersen commitment, secret shares of r
s = []
# complementing blinders for proof of bit-ness
s2 = []
partial_sum = G.Scalar(0)
for i in range(len(bases) - 1):
    s.append(G.random_scalar())
    partial_sum += bases[i] * s[i]
    s2.append((G.Scalar(1) - b[i]) * s[i])
    D.append(b[i] * generatorG + s[i] * generatorH)
idx = len(bases) - 1
s[idx] = r - partial_sum
s2.append((G.Scalar(1) - b[idx]) * s[idx])
D.append(b[idx] * generatorG + s[idx] * generatorH)

# Compute the Pedersen commitment to the full value of v, using the provided r.
C = v * generatorG + r * generatorH

# start computing the linear relation
statement = LinearRelation(G)

[var_G, var_H, var_C] = statement.allocate_elements(3)
```

```
# allocate variables for decomposed statements
vars_b = statement.allocate_scalars(len(b))
# allocate blinding elements for Pedersen commitment
vars_s = statement.allocateScalars(len(b))
# allocate complementing blinders for proof of bit-ness
vars_s2 = statement.allocateScalars(len(b))
# allocate bit commitment values
vars_D = statement.allocateElements(len(b))

# Add equations proving each b[i] is in {0,1}
# For each base, we prove:
#   D[i] = b[i] * generatorG + s[i] * generatorH      (b[i] is committed in D[i])
#   b[i] * (b[i] - 1) = 0       (b[i] is 0 or 1)
for i in range(len(b)):
        statement.set_elements([(vars_D[i], D[i])])
        # add Pedersen commitment to the ith bit.
        statement.append_equation(vars_D[i], [(vars_b[i], var_G), (vars_s[i], var_H)])
        # add statement that b[i] is in {0,1}
        statement.append_equation(vars_D[i], [(vars_b[i], vars_D[i]), (vars_s2[i], var_H)
])

return (statement, [r, v, b, s, s2], [C,D])
```

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 [0, 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, nonce, presentationContext, presentationProof] are indistinguishable from all presentations made from credentials issued with the same server keys for that presentationContext, with the exception of presentations with the same nonce (since those presentations can be ascertained as being generated from different credentials, as long as the presentation tag is unique).

The indistinguishability set for those presentation elements is sum_{i=0}^c(p_i[presentationContext]) - k[presentationContext], where c is the number of credentials issued with the same server keys, p_i[presentationContext] is the number of presentations made for each of those credentials with the same presentationContext, and k is the number of presentations with the same nonce for that presentationContext. As long as the nonces are generated randomly from the range defined by the presentation limit, k[presentationContext] should be roughly equal to sum_{i=0}^c(p_i[presentationContext]) / n, where n is the presentation limit. Therefore, the indistinguishability set can be represented as sum_{i=0}^c(p_i[presentationContext])(1 - 1/n), where a larger presentation limit results in a larger indistinguishability set and therefore stronger unlinkability properties.

OPEN ISSUE: hide the nonce and replace the tag proof with a range proof built from something like Bulletproofs.

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

This section contains test vectors for the ARC ciphersuites specified in this document.

10.1. ARCV1-P256

```
// ServerKey
x0 = 3338fa65ec36e0290022b48eb562889d89dbfa691d1cde91517fa222ed7ad36
4
x1 = f9db001266677f62c095021db018cd8cbb55941d4073698ce45c405d1348b7b
1
x2 = 350e8040f828bf6ceca27405420cdf3d63cb3aef005f40ba51943c802687796
3
xb = fd293126bb49a6d793cd77d7db960f5692fec3b7ec07602c60cd32aee595dff
d
X0 = 0232b5e93dc2ff489c20a986a84757c5cc4512f057e1ea92011a26d3ad2c562
88d
X1 = 03c413230a9bd956718aa46138a33f774f4c708d61c1d6400d404243049d4a3
1dc
X2 = 02db00f6f8e6d235786a120017bd356fe1c9d09069d3ac9352cc9be10ef1505
a55

// CredentialRequest
Blinding_0 = 0000000000000000000000000000000000000000000000000000000
000000001
Blinding_1 = 0000000000000000000000000000000000000000000000000000000
000000002
Blinding_2 = 0000000000000000000000000000000000000000000000000000000
000000003
Blinding_3 = 0000000000000000000000000000000000000000000000000000000
000000004
```

```
request_context = 746573742072657175657374206f6e74657874
m1 = eedfe7939e2382934ab5b0f76aae44124955d2c5ebf9b41d88786259c34692d
2
m2 = 911fb315257d9ae29d47ecb48c6fa27074dee6860a0489f8db6ac9a486be6a3
e
r1 = 008035081690bfde3b1e68b91443c22cc791d244340fe957d5aa44d7313740d
f
r2 = d59c5e6ff560cc597c2b8ca25256c720bceca2ab03921492c5e9e4ad3b55800
2
m1_enc = 03b8f11506a5302424143573e087fa20195cb5e893a67ef354eae3a78e2
63c54e4
m2_enc = 03f1ae4d7b78ba8030bd63859d4f4a909395c52bda34716b6620a2fdd52
b336fc9
proof = 0f361327abbc724ff0d37db365065bc4bd60e18125842bb4c03a7e5a632a
1e95e74dcc440fcb9fb39106922e0d2544e6c82ca710abf35e8b10bf5d61296c9adb
7d683eaed9a76a755b73f2b4b6e763a7c7883ce4b5c21bd02cd96b9af18cfb227f1a
cb4ead77c85049d291ed7841405610843f163e9cc2f6a8869111582324cd32bf1300
0c129d274ccf5386cb90e839916d5dff7eade18e3eabec415f613911

// CredentialResponse
Blinding_0 = 0000000000000000000000000000000000000000000000000000000
000000001
Blinding_1 = 0000000000000000000000000000000000000000000000000000000
000000002
Blinding_2 = 0000000000000000000000000000000000000000000000000000000
000000003
Blinding_3 = 0000000000000000000000000000000000000000000000000000000
000000004
Blinding_4 = 0000000000000000000000000000000000000000000000000000000
000000005
Blinding_5 = 0000000000000000000000000000000000000000000000000000000
000000006
Blinding_6 = 0000000000000000000000000000000000000000000000000000000
000000007
b = e699140babbe599f7dd8f6e3e8f615e5f201d1c2b0bc2f821f19e80a0a0e1e7b
U = 033ee1ebbcff622bc26b10932ed1eb147226d832048fb2337dc0ad7722cb0748
3d
enc_U_prime = 035b8e09ce8776f1a2c7ef8610c9a6a39936c5666ab8b28d6629d3
685056716482
X0_aux = 02d453c121324114367906bd11ffc3b6e6a77b75382497279b1a60ab841
2c1dec6
X1_aux = 03b0e4b1f376c6207bf34efda46ce54b132a20b90bc28b9152f3e441fe2
b508b63
X2_aux = 0327369efcb7577abaeb7b56940e6e042126900bdf8bd8944c0adbb7be3
ad98e2a
H_aux = 03d3cd09eeb8d19716586a49260c69309c495a717a36cad3381f6c02ac80
b70e64
proof = dd4596175db0b4273fcdff330370d2b5e7a4bf92bf518141f4553af37ef0
```

```
   e1260cb8312affc2462800adba102117448b449985d1704d8afd0df9ac708231561d
   ca56faae325cb56b0a9e8ad07bdc6ce90f6e7430090e970a7240e289218de7a17672
   bea9a66187d102ffef976fb01af69d8d3aa3156a5a4223dc6d08b8ce9f1d2639a2ed
   c7052404bf1410adf6c41465bd687e3dfa5372ea71f804b56d947bae9482e5707f42
   dbe35f8b0e11b4a0d27a5a01e1b9a75b66d82b7945eb0b002ee400bebcdc4c3133f8
   04b22bd2d771762058cc35a5033365d2e15150fe46d3b0e98e18ee55f0451b0b1714
   20f73592292e4ff50603c1f0d7769dbd090936090f63

   // Credential
   m1 = eedfe7939e2382934ab5b0f76aae44124955d2c5ebf9b41d88786259c34692d
   2
   U = 033ee1ebbcff622bc26b10932ed1eb147226d832048fb2337dc0ad7722cb0748
   3d
   U_prime = 02637fe04cc143281ee607bd8f898e670293dce44a2840b9cbb9e0d1fc
   7a2b29b4
   X1 = 03c413230a9bd956718aa46138a33f774f4c708d61c1d6400d404243049d4a3
   1dc

   // Presentation1
   Blinding_0 = 0000000000000000000000000000000000000000000000000000000
   000000001
   Blinding_1 = 0000000000000000000000000000000000000000000000000000000
   000000002
   Blinding_2 = 0000000000000000000000000000000000000000000000000000000
   000000003
   Blinding_3 = 0000000000000000000000000000000000000000000000000000000
   000000004
   presentation_context = 7465737420707265736e746174696f6e20636f6e746
   57874
   a = b78e57df8f0a95d102ff12bbb97e15ed35c23e54f9b4483d30b76772ee60d886
   r = 42252210dd60ddbbf1a57e3b144e26dd693b7644a9626a8c36896ede53d12930
   z = f5a4bbcf14e55e357df9f5ccb5ded37b2b14bc2e1a68e31f86416f0606ee75d1
   U = 032704f22133d2ec70f9e6f4bbf64c582220b666f2e2c1d37c3f8995a2a5568c
   7e
   U_prime_commit = 03533cf1b2fd53a0716e02425eb42e4c55835aa6b2992d364cb
   a70810d0f8aeb51
   m1_commit = 03e412408579105213ed10b6447c85bcd672ba73ecae1e21c463d0df
   4ef7beb814
   nonce = 0x0
   tag = 031a774fd87a8f18f6420bea43cf5425e7426eec8ba7b8df5c13dc05f10ec6
   52d9
   proof = a558da5f17c04adcb0898827aaded14be1dc612dcd12b0579c11bb387ce9
   ae4b7dbcb3bbe413caaaf754d99e5a342abb7e0041458d670f4b58eda37e745a6752
   95d7a7b86248141d6547b53d793e5c77896ec4dc8dd438ab66d9c8b43ef6b060938a
   1ca793057b154970ebc3c7ec3a23134e0852d0041f9098ce77311e5b5eca00000000
   0000000000000000000000000000000000000000000000000000000000000004

   // Presentation2
```

```
Blinding_0 = 0000000000000000000000000000000000000000000000000000000000
000001
Blinding_1 = 0000000000000000000000000000000000000000000000000000000000
000002
Blinding_2 = 0000000000000000000000000000000000000000000000000000000000
000003
Blinding_3 = 0000000000000000000000000000000000000000000000000000000000
000004
presentation_context = 7465737420707265736e746174696f6e20636f6e746
57874
a = 95bcf45150a61f5c44a6cfbf343cd9e0f593f127b6f49bec0f9b20f0550504a2
r = d7ed72750b6d366ed0febdc539b52d89434f468a578c59d7ca9015b7da240ad6
z = 91eedfb168c556ff5ca3b89d047f482c9279b47f584aab6c7f895f7674251771
U = 035fd233dee2c147155c6008ea64941b6ff7b315aced12531468f2e27bf22e3e
f0
U_prime_commit = 02434af337b87fd21d1e3d950aebfc8033a3d2e9dd2bb8b9e79
53488078754496d
m1_commit = 02a578fd3a84eb5b657367b02de39b45fd48ab7781ef8f94efe60127
4a5ded2a07
nonce = 0x1
tag = 03084fe6fff0ecc7c33ef5c49b492dda38083f52e9a2b70b88f3d4b4ba7b50
afba
proof = 050965cde906fc4723333b100ce0fd9f7b026315f1db16984d4cccb2bc4a
a65eb7a17f5b8dfe4f14d40006506ee5fb323e829dd4cb9dc3c455b2e04dd691600a
ec3cc3f1939198a80acb78b7f90b3bff769cab890f33e4d69b7c302d21ad35ec457d
048d3ed7d13ee82c3c0aac2129ad0c8375cf29cd8ea3948a16b9247b1cc5faf69a31
16f903b9dcccc4eff31f026041e49797b53c87eca66cfe1040187ef7
```

11. Acknowledgments

The authors would like to acknowledge helpful conversations with Tommy Pauly about rate limiting and Privacy Pass integration.

12. References

12.1. Normative References

[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-00, 8 August 2025, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-sigma-protocols-00>.

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