Crypto Forum
Internet-Draft
Intended status: Informational
Expires: 23 April 2026

C. Yun
C. A. Wood
Apple, Inc.
M. Raykova
S. Schlesinger
Google

20 October 2025

Anonymous Tokens with Hidden Metadata
draft-yun-cfrg-athm-00

Abstract

This document specifies the Anonymous Tokens with Hidden Metadata (ATHM) protocol, a protocol for constructing Privacy Pass like tokens with hidden metadata embedded within it unknown to the client.

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://cathieyun.github.io/draft-athm/draft-yun-cfrg-athm.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-yun-cfrg-athm/.

Discussion of this document takes place on the Crypto Forum Research Group mailing list (mailto:cfrg@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/cfrg. Subscribe at https://www.ietf.org/mailman/listinfo/cfrg/.

Source for this draft and an issue tracker can be found at https://github.com/cathieyun/draft-athm.

Status of This Memo

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

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

Copyright Notice

Table of Contents

1. Introduction

This document presents a construction for Anonymous Tokens with Hidden Metadata (ATHM). ATHM is a cryptographic primitive that enables an issuer to issue an authentication token to a client in a way that the token carries no information about the client identity except a hidden metadata value the issuer sets during issuance. The value of the hidden metadata comes from a fixed domain and the client can verify this property of the hidden metadata of the token it receives. The metadata value remains hidden with respect to any party which does not have the secret key for the ATHM construction, including the client. Only the issuer who has the secret for the construction can verify the token and read the hidden metadata value.

Similarly to public metadata proposed in the context of existing anonymous tokens schemes, the hidden metadata enables the issuer to partition clients into a number of groups proportional to the domain of the hidden metadata. During token redemption the issuer can identify which group the client belongs to based on the metadata value. The difference from public metadata is that this value of hidden metadata is only readable by the issuer who has the ATHM secret key, while public metadata embedded in the token is visible to everyone.

Anonymous tokens are used for client authentication and their meaning is to convey trust in the client. In most cases the trust assigned to a client is determined based on the outcome of fraud and spam detection processes. It is often crucial that the signals used to obtain the verdict and the verdict itself remain hidden from the adversary, who can adapt its behavior to avoid detection if it knows when and why its behavior has been identified as fraudulent. ATHM can provide the means to issue authentication tokens that provide limited hidden fraud signals while preserving strong anonymity guarantees.

2. Conventions and Definitions

2.1. Notation and Terminology

The following functions and notation are used throughout the document.

* For any object x, we write len(x) to denote its length in bytes.

* For two byte arrays x and y, write x || y to denote their concatenation.

* I2OSP(x, xLen): Converts a non-negative integer x into a byte array of specified length xLen as described in [RFC8017]. Note that this function returns a byte array in big-endian byte order.

* 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.

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.

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

The PrivateInput data type refers to inputs that are known only to the client in the protocol, whereas the PublicInput data type refers to inputs that are known to both client and server in the protocol.

The following terms are used throughout this document.

* Client: Protocol initiator. Creates an encrypted request, and uses the corresponding server encrypted issuance to make a presentation.

* Server: Computes an encrypted issuance for an encrypted 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 issuance and presentation steps.

3. Preliminaries

The construction in this document has two primary dependencies:

* Group: A prime-order group implementing the API described below in Section 3.1. See Section 4 for specific instances of groups.

* Hash: A cryptographic hash function whose output length is Nh bytes.

Section 4 specifies ciphersuites as combinations of Group and Hash.

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 KVAC 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 4 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).

* Generators(): Outputs the generator elements of the group, (generatorG, generatorH) generatorG = Group.generator generatorH = HashToGroup(SerializeElement(generatorG), "generatorH"). The group member functions GeneratorG() and GeneratorH() are shorthand for returning generatorG and generatorH, respectively.

* 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 4 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 4 for group-specific input validation steps.

Section 4 contains details for the implementation of this interface for different prime-order groups instantiated over elliptic curves. In particular, for some choices of elliptic curves, e.g., those detailed in [RFC7748], which require accounting for cofactors, Section 4 describes required steps necessary to ensure the resulting group is of prime order.

4. 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 instantiations of the following functionalities:

* 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.

* Hash: A cryptographic hash function whose output length is Nh bytes long.

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

For each ciphersuite, contextString is that which is computed in the Setup functions. Applications should take caution in using ciphersuites targeting P-256 and ristretto255.

4.1. ATHM(P-256)

This ciphersuite uses P-256 [NISTCurves] for the Group. The value of the ciphersuite identifier is "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 4.2 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].

4.2. Random Scalar Generation

Two popular algorithms for generating a random integer uniformly distributed in the range [0, G.Order() -1] are as follows:

4.2.1. 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, return the result. If it fails, 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.

4.2.2. 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.

5. Anonymous Token with Hidden Metadata Protocol

TODO(caw): writeme

5.1. Key Generation and Context Setup

In the offline phase, the server generates a public and private key using the KeyGen routine below. The type PublicKeyProof and the function CreatePublicKeyProof are specified in Section 5.1.1. Further, they must agree on a string deploymentId and a number of buckets nBuckets. The contextString as used in HashToGroup and HashToCurve is ATHMV1-<group-name>-<nBuckets>-<deploymentId>. An example of contextString is ATHMV1-P256-4-example_deployment_id.

```
Input: None
Output:
- privateKey:
  - x: Scalar
  - y: Scalar
  - z: Scalar
  - r_x: Scalar
  - r_y: Scalar
- publicKey:
  - Z: Element
  - C_x: Element
  - C_y: Element
  - pi: PublicKeyProof

Parameters
- Group G

def KeyGen():
  x = G.RandomScalar()
  y = G.RandomNonzeroScalar()
  z = G.RandomNonzeroScalar()
  r_x = G.RandomScalar()
  r_y = G.RandomScalar()

  Z = z * G.GeneratorG()
  C_x = (x * G.GeneratorG()) + (r_x * G.GeneratorH())
  C_y = (y * G.GeneratorG()) + (r_y * G.GeneratorH())
  pi = CreatePublicKeyProof(z, Z)

  return privateKey(x, y, z, r_x, r_y), publicKey(Z, C_x, C_y, pi)
```

The output publicKey from this function is wire-encoded into publicKey = 3*Ne+2*Ns bytes as follows:

```
struct {
  uint8 Z_enc[Ne];
  uint8 C_x_enc[Ne];
  uint8 C_y_enc[Ne];
  uint8 pi_enc[Nproof];
}
```

The Z_enc, C_x_enc, and C_y_enc fields are the serialized representations of publicKey.Z, publicKey.C_x, and publicKey.C_y, respectively. The pi_enc field is the serialized PublicKeyProof, as defined below.

5.1.1. Public Key Proof

The server public key carries with it a zero-knowledge proof of knowledge of the corresponding private key. Clients verify this proof before using the public key for token issuance. The procedures for creating and verifying this proof, CreatePublicKeyProof and VerifyPublicKeyProof, are detailed below.

Input:
- z: Scalar
- Z: Element

Output:
- pi: PublicKeyProof
  - e: Scalar
  - a_z: Scalar

Parameters
- Group G

```
def CreatePublicKeyProof(z, Z):
  rho_z = G.RandomScalar()
  gamma_z = rho_z * G.GeneratorG()

  ser_genG = G.SerializeElement(G.GeneratorG())
  ser_Z = G.SerializeElement(Z)
  ser_gamma_z = G.SerializeElement(gamma_big_z)

  challenge_transcript =
    I2OSP(len(ser_genG), 2) + ser_genG +
    I2OSP(len(ser_Z), 2) + ser_Z +
    I2OSP(len(ser_gamma_z), 2) + ser_gamma_z

  e = G.HashToScalar(challenge_transcript, "KeyCommitments")
  a_z = rho_z - (e * z)

  return PublicKeyProof(e, a_z)
```

The output of CreatePublicKeyProof is a value of type PublicKeyProof, which is wire-encoded as the concatenation of the e and a_z values, both serialized using SerializeScalar, yielding the following format:

```
struct {
  uint8 e_enc[Ns];
  uint8 a_z_enc[Ns];
} PublicKeyProof;
```

The VerifyPublicKeyProof routine, which takes as input a server public key, is below.

```
Input:
- publicKey:
  - Z: Element
  - C_x: Element
  - C_y: Element
  - pi: PublicKeyProof

Output:
- verifiedPublicKey if valid, False otherwise

def VerifyPublicKeyProof(publicKey):
  pi = publicKey.pi
  gamma_z = (pi.e * publicKey.Z) +
    (pi.a_z * G.GeneratorG())

  ser_genG = G.SerializeElement(G.GeneratorG())
  ser_Z = G.SerializeElement(Z)
  ser_gamma_z = G.SerializeElement(gamma_big_z)

  challenge_transcript =
    I2OSP(len(ser_genG), 2) + ser_genG +
    I2OSP(len(ser_Z), 2) + ser_Z +
    I2OSP(len(ser_gamma_z), 2) + ser_gamma_z

  e_verify = G.HashToScalar(challenge_transcript, "KeyCommitments")
  if e_verify == e:
    return verifiedPublicKey(Z, C_x, C_y)
  return False
```

The output verifiedPublicKey from this function is wire-encoded into verifiedPublicKey = 3*Ne bytes as follows:

```
struct {
  uint8 Z_enc[Ne];
  uint8 C_x_enc[Ne];
  uint8 C_y_enc[Ne];
}
```

5.2. ATHM Protocol

The ATHM Protocol is a two-party protocol between a client and server where they interact to compute

```
token = F(privateKey, publicKey, hiddenMetadata)
```

where privateKey and publicKey contains the server's private and public keys, respectively (and generated as described in Section 5.1), and hiddenMetadata is an application-specific value known only to the server.

The protocol begins with the client making a token request using the verified server public key.

```
verifiedPublicKey = VerifyPublicKeyProof(publicKey, pi)
(context, request) = TokenRequest(verifiedPublicKey)
```

The client then sends request to the server. If the request is well-formed, the server computes a token response with the server private keys and hidden metadata. The response includes a proof that the token response is valid with respect to the server keys, and a maximum number of buckets for the hidden metadata.

```
response = CredentialResponse(privateKey, publicKey, request, hiddenMetadata, nBuckets)
```

The server sends the response to the client. The client processes the response by verifying the response proof. If the proof verifies correctly, the client computes a token from its context and the server response:

```
token = FinalizeToken(context, verifiedPublicKey, request, response, nBuckets)
```

When the client presents the token to the server for redemption, the server verifies it using its private keys, as follows:

```
hiddenMetadata = VerifyToken(privateKey, token, nBuckets)
```

This process fails if the token is invalid, and otherwise returns the hidden metadata associated with the token during issuance.

Shown graphically, the protocol runs as follows:

In the remainder of this section, we specify the TokenRequest, TokenResponse, FinalizeToken, and VerifyToken functions that are used in this protocol.

5.3. TokenRequest

The TokenRequest function is defined below.

```
Inputs:
- verifiedPublicKey:
  - Z: Element
  - C_x: Element
  - C_y: Element

Outputs:
- context:
  - r: Scalar
  - tc: Scalar
- request:
  - T: Element

Parameters:
- G: Group

def TokenRequest(client):
  r = G.RandomScalar()
  tc = G.RandomScalar()
  T = (r * G.GeneratorG()) + (tc * verifiedPublicKey.Z)
  return context(r, tc), request(T)
```

The output request from this function is wire-encoded into Nrequest=Ne bytes as follows:

```
struct {
  uint8 T_enc[Ne];
}
```

The T_enc field is the serialized representation of request.T.

5.4. TokenResponse

The TokenResponse function is defined below.

```
   Inputs:
   - privateKey:
     - x: Scalar
     - y: Scalar
     - z: Scalar
     - r_x: Scalar
     - r_y: Scalar
   - publicKey:
     - Z: Element
     - C_x: Element
     - C_y: Element
     - pi: PublicKeyProof
   - request:
     - T: Element
   - hiddenMetadata: Integer
   - nBuckets: Integer

   Outputs:
   - response:
     - U: Element
     - V: Element
     - ts: Scalar
     - pi: IssuanceProof

   Parameters:
   - G: Group

   def TokenResponse(privateKey, publicKey, request, hiddenMetadata, nBuckets):
     ts = G.RandomScalar()
     d = G.RandomNonzeroScalar()

     U = d * G.GeneratorG()
     X = privateKey.x * GeneratorG()
     Y = privateKey.y * GeneratorG()
     V = d * (X + (hiddenMetadata * Y) + (ts * publicKey.Z) + request.T)

     pi = CreateIssuanceProof(privateKey, publicKey, hiddenMetadata, nBuckets, d, U, V, t
s, T)

     return response(U, V, ts, pi)
```

The output response from this function is wire-encoded into Nresponse=Ne+Ne+Ns+Nproof bytes as follows:

```
struct {
  uint8 U_enc[Ne];
  uint8 V_enc[Ne];
  uint8 ts_enc[Ns];
  uint8 pi_enc[Nproof];
}
```

The U_enc, V_enc, and ts_enc fields are the serialized representations of response.U, response.V, and response.ts, respectively. The pi_enc field is the serialized IssuanceProof, as defined below.

The CreateIssuanceProof function is defined below.

Inputs:
- privateKey:
  - x: Scalar
  - y: Scalar
  - z: Scalar
  - r_x: Scalar
  - r_y: Scalar
- publicKey:
  - Z: Element
  - C_x: Element
  - C_y: Element
  - pi: PublicKeyProof
- hiddenMetadata: Integer
- nBuckets: Integer
- d: Scalar
- U: Element
- V: Element
- ts: Scalar
- T: Element

Output:
- pi: IssuanceProof
  - C: Element
  - e: [Scalar]
  - a: [Scalar]
  - a_d: Scalar
  - a_rho: Scalar
  - a_w: Scalar

Parameters:
- G: Group

```
def CreateIssuanceProof(privateKey, publicKey, hiddenMetadata, nBuckets, d, U, V, ts,
T):
  e_vec = [G.RandomScalar() for i in range(nBuckets)]
```

```
    e_vec[hiddenMetadata] = Scalar(0) # zero for now - will be calculated and set later.
    a_vec = [G.RandomScalar() for i in range(nBuckets)]
    a_vec[hiddenMetadata] = Scalar(0) # zero for now - will be calculated and set later.
    r_mu = G.RandomScalar()
    r_d = G.RandomScalar()
    r_rho = G.RandomScalar()
    r_w = G.RandomScalar()
    mu = G.RandomScalar()

    C = hiddenMetadata * publicKey.C_y + mu * G.GeneratorH()
    C_vec = []
    for i in range(nBuckets):
      if i == hiddenMetadata:
        # This is the actual C[i] value for the "correct" slot (i == hiddenMetadata)
        C_vec.append(r_mu * G.GeneratorH())
      else:
        # This is the simulated C[i] value for the "incorrect" slot (i != hiddenMetadata
)
        C_vec.append(a_vec[i] * G.GeneratorH() - e_vec[i] * (C - (i * pk.C_y)))
    C_d = r_d * U
    C_rho = (r_d * V) + (r_rho * G.GeneratorH())
    C_w = (r_d * V) + (r_w * G.GeneratorG())

    ser_genG = G.SerializeElement(G.GeneratorG())
    ser_genH = G.SerializeElement(G.GeneratorH())
    ser_C_x = G.SerializeElement(publicKey.C_x)
    ser_C_y = G.SerializeElement(publicKey.C_y)
    ser_Z = G.SerializeElement(publicKey.Z)
    ser_U = G.SerializeElement(U)
    ser_V = G.SerializeElement(V)
    ser_ts = G.SerializeScalar(ts)
    ser_T = G.SerializeElement(T)
    ser_C = G.SerializeElement(C)
    ser_C_d = G.SerializeElement(C_d)
    ser_C_rho = G.SerializeElement(C_rho)
    ser_C_w = G.SerializeElement(C_w)

    challenge_transcript = \
      I2OSP(len(ser_genG), 2) + ser_genG + \
      I2OSP(len(ser_genH), 2) + ser_genH + \
      I2OSP(len(ser_C_x), 2) + ser_C_x + \
      I2OSP(len(ser_C_y), 2) + ser_C_y + \
      I2OSP(len(ser_Z), 2) + ser_Z + \
      I2OSP(len(ser_U), 2) + ser_U + \
      I2OSP(len(ser_V), 2) + ser_V + \
      I2OSP(len(ser_ts), 2) + ser_ts + \
      I2OSP(len(ser_T), 2) + ser_T + \
      I2OSP(len(ser_C), 2) + ser_C
    for i in range(nBuckets):
```

```
      ser_C_i = G.SerializeElement(C_vec[i])
      challenge_transcript.append(I2OSP(len(ser_C_i), 2) + ser_C_i)
    challenge_transcript.append( \
      I2OSP(len(ser_C_d), 2) + ser_C_d + \
      I2OSP(len(ser_C_rho), 2) + ser_C_rho + \
      I2OSP(len(ser_C_w), 2) + ser_C_w
    )

    e = G.HashToScalar(challenge_transcript, "TokenResponseProof")
    # Set the correct e_vec[hiddenMetadata] value.
    e_vec[hiddenMetadata] = e - sum(e_vec)

    d_inv = G.ScalarInverse(d)
    rho = -(privateKey.r_x + (hiddenMetadata * privateKey.r_y) + mu)
    w = privateKey.x + (hiddenMetadata * privateKey.y) + (ts * privateKey.z)

    # Set the correct a_vec[hiddenMetadata] value.
    a_vec[hiddenMetadata] = r_mu + (e_vec[hiddenMetadata] * mu)
    a_d = r_d - (e * d_inv)
    a_rho = r_rho + (e * rho)
    a_w = r_w + (e * w)

    return IssuanceProof(C, e_vec, a_vec, a_d, a_rho, a_w)
```

The output pi from this function is wire-encoded into Nproof = Ne+(3+2*nBuckets)*Ns bytes as follows:

```
struct {
  uint8 C_enc[Ne];
  [uint8] e_0_enc[Ns]...e_nBuckets_enc[Ns];
  [uint8] a_0_enc[Ns]...a_nBuckets_enc[Ns];
  uint8 a_d_enc[Ns];
  uint8 a_rho_enc[Ns];
  uint8 a_w_enc[Ns];
}
```

The fields in this structure are the serialized representations of the contents of pi, e.g., C_enc is the serialized representation of pi.C and e_0_enc is the serialized representation of e[0].

5.4.1. FinalizeToken

The FinalizeToken function is defined below. Internally, the client verifies the token response proof. FinalizeToken fails if this proof is invalid.

```
Inputs:
- context:
  - r: Scalar
  - tc: Scalar
- verifiedPublicKey:
  - Z: Element
  - C_x: Element
  - C_y: Element
- request:
  - T: Element
- response:
  - U: Element
  - V: Element
  - ts: Scalar
  - pi: IssuanceProof
- nBuckets: Integer

Outputs:
- token:
  - t: Scalar
  - P: Element
  - Q: Element

Parameters:
- G: Group

Exceptions:
- VerifyError, raised when response proof verification fails

def FinalizeToken(context, verifiedPublicKey, request, response, nBuckets):
  if VerifyIssuanceProof(verifiedPublicKey, request, response, nBuckets) == false:
    raise VerifyError

  c = G.RandomNonzeroScalar()
  P = c * response.U
  Q = c * (response.V - (context.r * response.U))
  t = context.tc + response.ts

  return token(t, P, Q)
```

The resulting token can be serialized as the concatenation of t, P, and Q serialized using their respective serialization functions, yielding the following struct:

```
   struct {
     uint8 t_enc[Ns];
     uint8 P_enc[Ne];
     uint8 Q_enc[Ne];
   }
```

The VerifyIssuanceProof function is defined below.

Inputs:
- verifiedPublicKey:
  - Z: Element
  - C_x: Element
  - C_y: Element
- T: Element
- response:
  - U: Element
  - V: Element
  - ts: Scalar
  - pi: IssuanceProof
- nBuckets: Integer

Output:
- True if valid, false otherwise

Parameters:
- G: Group

```
   def VerifyIssuanceProof(verifiedPublicKey, T, response, nBuckets):
     pi = response.pi

     C_vec = []
     for i in range(nBuckets):
       C_i = pi.a_vec[i] * G.GeneratorH - (pi.e_vec[i] * (pi.C - (i * verifiedPublicKey.C
_y))

     e = sum(pi.e_vec)

     C_d = (response.pi.a_d * response.U) + (e * G.GeneratorG())

     C_rho = (response.pi.a_d * response.V)
         + (response.pi.a_rho * G.GeneratorH())
         + (e * (verifiedPublicKey.C_x + response.pi.C + (response.ts * verifiedPublicKey
.Z) + T))

     C_w = (response.pi.a_d * response.V)
         + (response.pi.a_w * G.GeneratorG())
         + (e * T)

     ser_genG = G.SerializeElement(G.GeneratorG())
     ser_genH = G.SerializeElement(G.GeneratorH())
```

```
    ser_C_x = G.SerializeElement(verifiedPublicKey.C_x)
    ser_C_y = G.SerializeElement(verifiedPublicKey.C_y)
    ser_Z = G.SerializeElement(verifiedPublicKey.Z)
    ser_U = G.SerializeElement(response.U)
    ser_V = G.SerializeElement(response.V)
    ser_ts = G.SerializeScalar(response.ts)
    ser_T = G.SerializeElement(T)
    ser_C = G.SerializeElement(pi.C)
    ser_C_d = G.SerializeElement(C_d)
    ser_C_rho = G.SerializeElement(C_rho)
    ser_C_w = G.SerializeElement(C_w)

    challenge_transcript = \
      I2OSP(len(ser_genG), 2) + ser_genG + \
      I2OSP(len(ser_genH), 2) + ser_genH + \
      I2OSP(len(ser_C_x), 2) + ser_C_x + \
      I2OSP(len(ser_C_y), 2) + ser_C_y + \
      I2OSP(len(ser_Z), 2) + ser_Z + \
      I2OSP(len(ser_U), 2) + ser_U + \
      I2OSP(len(ser_V), 2) + ser_V + \
      I2OSP(len(ser_ts), 2) + ser_ts + \
      I2OSP(len(ser_T), 2) + ser_T + \
      I2OSP(len(ser_C), 2) + ser_C + \
    for i in range(nBuckets):
      ser_C_i = G.SerializeElement(pi.C_vec[i])
      challenge_transcript.append(I2OSP(len(ser_C_i), 2) + ser_C_i)
    challenge_transcript.append( \
      I2OSP(len(ser_C_d), 2) + ser_C_d + \
      I2OSP(len(ser_C_rho), 2) + ser_C_rho + \
      I2OSP(len(ser_C_w), 2) + ser_C_w
    )

    e_verify = G.HashToScalar(challenge_transcript, "TokenResponseProof")
    return e == e_verify
```

5.4.2. VerifyToken

The VerifyToken token is defined below.

```
Inputs:
- privateKey:
  - x: Scalar
  - y: Scalar
  - z: Scalar
  - r_x: Scalar
  - r_y: Scalar
- token:
  - t: Scalar
  - P: Element
  - Q: Element
- nBuckets: Integer

Outputs:
- hiddenMetadata: Integer

Parameters:
- G: Group

Exceptions:
- VerifyError, raised when token verification fails

def VerifyToken(privateKey, token, nBuckets):
  if (token.P == token.P + token.P) || (token.Q == token.Q + token.Q):
    raise VerifyError # P and Q should not be zero

  iMatch = -1
  for i in range(nBuckets):
    Q_i = (privateKey.x + token.t * privateKey.z + i * privateKey.y) * token.P
    if token.Q == Q_i:
      if iMatch != -1:
        raise VerifyError # Multiple metadata values match
      iMatch = i

  if iMatch != -1:
    return iMatch
  else:
    raise VerifyError # No metadata values match
```

6. Security Considerations

The work of [CDV22] proves the following properties for the ATHM construction presented here: unforgeability, unlinkability and privacy of the metadata. Unforgeability guarantees that only an issuer with a valid secret key can generate new tokens. Unlinkability states that tokens with the same metadata are indistinguishable to the redeemer, i.e. the redeemer does not have any advatage in guessing which issuance session (among those assigned

the same metadata) a paricular redeemed token comes from. Finally, privacy of the metadata guarantees that any party who does not know the secret verification key cannot learn any information about the hidden metadata of a token.

The only place where the construction presented in this document differs from the ATHM construcion presented in [CDV22] is the fact that we generalize that construction to more than one bit for the hidden metadata. This requires that the ZK proof provided during issuance to the client proves a new bound on the number of embedded bits in the token. The proofs of unfogeability and the privacy of the hidden metadata bits do not depend on the range of the metadata. The unlinkability proof accounts for the fact that the issuer can partition the clients into a larger than two number of groups, since unlinkability only holds among tokens with the same metadata.

7. IANA Considerations

This document has no IANA actions.

8. References

8.1. 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>.

8.2. Informative References

[CDV22] "Anonymous Tokens with Stronger Metadata Bit Hiding 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.186-4, 2013, <https://doi.org/10.6028/nist.fips.186-4>.

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

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

Acknowledgments

Thanks to Melissa Chase for discussion about the ATHM paper. Thanks also to Tommy Pauly, Phillipp Schoppmann and Ghous Amjad for collaboration on the ATHM specifications.

Test Vectors

This section contains test vectors for the ATHM ciphersuites specified in this document. The test vectors were generated from the reference implementation located at https://github.com/google/anonymous-tokens/pull/45/commits/8701431d31c2f49774526fac23bf7a2d24864314. All byte strings, except deployment_id, are encoded in hexadecimal.

ATHM(P-256)


```
[
  {
    "procedure": "params",
    "args": {},
    "output": {
      "deployment_id": "test_vector_deployment_id",
      "generator_g": "036b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296",
      "generator_h": "02361fc6831d3796a82612dffb231ec67253b2f69dbb124c9a0f9917b4e3180d03",
      "n_buckets": "4"
    }
  },
  {
    "procedure": "key_gen",
    "args": {
      "rng_seed": "0101010101010101010101010101010101010101010101010101010101010101"
    },
    "output": {
```

```
        "key_id": "027defbe3a76d47f76e8e1296ddbadf8faeb91852a5964d7986ad974441dfc1c",
        "private_key": "023f37203a2476c42566a61cc55c3ca875dbb4cc41c0deb789f8e7bf881836381ecc3686b60ee3b84b6c7d321d70d5c06e9dac63a4d0a79d731b17c0d04d030d01274dd1ee5216c204fb698daea45b52e98b6f0fdd046dcc3a86bb079e36f024147e4b875d59a9ef432b8e45b04a98c4b19dc8c7475f5dce4259b4ca2dd67282b478b8702c1d2569fe52e5d7dbadec6223cd10fd4b504dabac7fff23a37363d1",
        "public_key": "032b80024ee818d709196780a84affe08789f571529fccc1acce07eea2df2fc65f035f6a6eb84bea9c1361119462bda863c47c3b43e53e9e5e7a8796ba1b924d093e03b7f16df0ad82cbd8bd6a04ed427107adf5a3d4ba840e9b341a3badc1d9b7fe19",
        "public_key_proof": "28ccf67d4ad339a45a8435b660160fef113a157221ce6f4a5c6f584a58d4509152852fcc950dd7495211dbefd34b733a730e5a2085e04f847e3c1af4355c59fe"
      }
    },
    {
      "procedure": "token_request",
      "args": {
        "public_key": "032b80024ee818d709196780a84affe08789f571529fccc1acce07eea2df2fc65f035f6a6eb84bea9c1361119462bda863c47c3b43e53e9e5e7a8796ba1b924d093e03b7f16df0ad82cbd8bd6a04ed427107adf5a3d4ba840e9b341a3badc1d9b7fe19",
        "public_key_proof": "28ccf67d4ad339a45a8435b660160fef113a157221ce6f4a5c6f584a58d4509152852fcc950dd7495211dbefd34b733a730e5a2085e04f847e3c1af4355c59fe",
        "rng_seed": "0202020202020202020202020202020202020202020202020202020202020202"
      },
      "output": {
        "token_context": "f6a12ca8ffc30a66ca140ccc7276336115819361186d3f535dd99f8eaaca8fce7f82dd63f4f75c33da444b72372be3aa43c0027a076bf9675eb7932695d127a4",
        "token_request": "030a1b41e492728f4f59d368761e1ff568536ad2987fd5be6017c5043ed25b67ea"
      }
    },
    {
      "procedure": "token_response",
      "args": {
        "hidden_metadata": "3",
        "private_key": "023f37203a2476c42566a61cc55c3ca875dbb4cc41c0deb789f8e7bf881836381ecc3686b60ee3b84b6c7d321d70d5c06e9dac63a4d0a79d731b17c0d04d030d01274dd1ee5216c204fb698daea45b52e98b6f0fdd046dcc3a86bb079e36f024147e4b875d59a9ef432b8e45b04a98c4b19dc8c7475f5dce4259b4ca2dd67282b478b8702c1d2569fe52e5d7dbadec6223cd10fd4b504dabac7fff23a37363d1",
        "public_key": "032b80024ee818d709196780a84affe08789f571529fccc1acce07eea2df2fc65f035f6a6eb84bea9c1361119462bda863c47c3b43e53e9e5e7a8796ba1b924d093e03b7f16df0ad82cbd8bd6a04ed427107adf5a3d4ba840e9b341a3badc1d9b7fe19",
        "rng_seed": "0303030303030303030303030303030303030303030303030303030303030303",
        "token_request": "030a1b41e492728f4f59d368761e1ff568536ad2987fd5be6017c5043ed25b67ea"
      },
      "output": {
        "token_response": "0245db306e323313840a20233ad432aae1be784f96aa940b628d56914249ce4b8c0250ccea1818d4ecf3026ebca1845fbff1cd0089868bbfdecac1c77b9ef2b00480385553aa23a24b14d8bbc2dff606277f444e049797ae7e0404e3a9ba0ecef2fb02d84ab8d7b059374064295a5b2c084c1814a4a33635769a6a196cb99c33122b6affb9e4ee66f5a203f5a31575fd70251b1cd922bcfaa87fc7c071df4412854ba49b424c1a1cded001b3ff8f0bc5306fa9950d236ce7075d9f702c3f0cd5546f8900a60c8dfef9f6633994aa4ed8b1ebd9111bcf05c62b39e9f7426edc7aab3fe679ff5c180ef849b41bd7033353818dd3e442431a1cb2c16992bdadef048a3673a43052fa4d37c9836e498619b9fc1584381292d0b1ec6f952fc6bab8035f6b30478648a01804edaba38b5cf910a932567c48e78b724d57400edd6f98cd565835bb1f733e33dc6335d9283d288ada574131ba288318476415651c571eee8430a75cae55cb231a35b67334ab9c30110d0bc6ce1d3654d4c44bbef74c3d572d432c7b4d06f693bd73505a93a56ab5190b18484c9e7231037fddfa45e1bcbb488def7d54110bbbe0a2a4799a6a8ccff13b1b347fe6867132b1dec0901f7bad11d7c2f8446cffc0f19d60db17c0db5770fdc83f70cd691de8b68289efafac5b098f63"
      }
    },
    {
      "procedure": "finalize_token",
      "args": {
```

```
        "public_key": "032b80024ee818d709196780a84affe08789f571529fccc1acce07eea2df2fc65
f035f6a6eb84bea9c1361119462bda863c47c3b43e53e9e5e7a8796ba1b924d093e03b7f16df0ad82cbd8bd6a
04ed427107adf5a3d4ba840e9b341a3badc1d9b7fe19",
        "rng_seed": "0404040404040404040404040404040404040404040404040404040404040404",
        "token_context": "f6a12ca8ffc30a66ca140ccc72763361158193611 86d3f535dd99f8eaaca8f
ce7f82dd63f4f75c33da444b72372be3aa43c0027a076bf9675eb7932695d127a4",
        "token_request": "030a1b41e492728f4f59d368761e1ff568536ad2987fd5be6017c5043ed25b
67ea",
        "token_response": "0245db306e323313840a20233ad432aae1be784f96aa940b628d56914249c
e4b8c0250ccea1818d4ecf3026ebca1845fbff1cd0089868bbfdecac1c77b9ef2b00480385553aa23a24b14d8
bbc2dff606277f444e049797ae7e0404e3a9ba0ecef2fb02d84ab8d7b059374064295a5b2c084c1814a4a3363
5769a6a196cb99c33122b6affb9e4ee66f5a203f5a31575fd70251b1cd922bcfaa87fc7c071df4412854ba49b
424c1a1cded001b3ff8f0bc5306fa9950d236ce7075d9f702c3f0cd5546f8900a60c8dfef9f6633994aa4ed8b
1ebd9111bcf05c62b39e9f7426edc7aab3fe679ff5c180ef849b41bd7033353818dd3e442431a1cb2c16992bd
adef048a3673a43052fa4d37c9836e498619b9fc1584381292d0b1ec6f952fc6bab8035f6b30478648a01804e
daba38b5cf910a932567c48e78b724d57400edd6f98cd565835bb1f733e33dc6335d9283d288ada574131ba28
8318476415651c571eee8430a75cae55cb231a35b67334ab9c30110d0bc6ce1d3654d4c44bbef74c3d572d432
c7b4d06f693bd73505a93a56ab5190b18484c9e7231037fddfa45e1bcbb488def7d54110bbbe0a2a4799a6a8c
cff13b1b347fe6867132b1dec0901f7bad11d7c2f8446cffc0f19d60db17c0db5770fdc83f70cd691de8b6828
9efafac5b098f63"
      },
      "output": {
        "token": "b7d8310e1899a748b3000e522d320b29880e07119f1a776b639b3ce0a4a01a9f02123d
0c125de3d122577b335a8f6616d735e9400b60dcde57eff056e9cbbd2b3c03a062c5b41507e1fe089d0c3b413
2d84ece1d4a9dfc0bf4f0588d660f25bdf6cf"
      }
    },
    {
      "procedure": "verify_token",
      "args": {
        "private_key": "023f37203a2476c42566a61cc55c3ca875dbb4cc41c0deb789f8e7bf88183638
1ecc3686b60ee3b84b6c7d321d70d5c06e9dac63a4d0a79d731b17c0d04d030d01274dd1ee5216c204fb698da
ea45b52e98b6f0fdd046dcc3a86bb079e36f024147e4b875d59a9ef432b8e45b04a98c4b19dc8c7475f5dce42
59b4ca2dd67282b478b8702c1d2569fe52e5d7dbadec6223cd10fd4b504dabac7fff23a37363d1",
```

```
        "token": "b7d8310e1899a748b3000e522d320b29880e07119f1a776b639b3ce0a4a01a9f02123d
0c125de3d122577b335a8f6616d735e9400b60dcde57eff056e9cbbd2b3c03a062c5b41507e1fe089d0c3b413
2d84ece1d4a9dfc0bf4f0588d660f25bdf6cf"
      },
      "output": {
        "hidden_metadata": "3"
      }
    }
  ]
```


Authors' Addresses

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

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

Mariana Raykova
Google
Email: marianar@google.com

Samuel Schlesinger
Google
Email: sgschlesinger@gmail.com