Network Working Group
Internet-Draft
Intended status: Informational
Expires: 19 February 2026

S. Schlesinger
J. Katz
Google

18 August 2025

Anonymous Credit Tokens
draft-schlesinger-cfrg-act-00

Abstract

This document specifies Anonymous Credit Tokens (ACT), a privacy-preserving authentication protocol that enables numerical credit systems without tracking individual clients. Based on keyed-verification anonymous credentials and privately verifiable BBS-style signatures, the protocol allows issuers to grant tokens containing credits that clients can later spend anonymously with that issuer.

The protocol's key features include: (1) unlinkable transactions - the issuer cannot correlate credit issuance with spending, or link multiple spends by the same client, (2) partial spending - clients can spend a portion of their credits and receive anonymous change, and (3) double-spend prevention through cryptographic nullifiers that preserve privacy while ensuring each token is used only once.

Anonymous Credit Tokens are designed for modern web services requiring rate limiting, usage-based billing, or resource allocation while respecting user privacy. Example applications include rate limiting and API credits.

This document is a product of the Crypto Forum Research Group (CFRG) in the IRTF.

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://SamuelSchlesinger.github.io/draft-act/draft-schlesinger-cfrg-act.html. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-schlesinger-cfrg-act/.

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/SamuelSchlesinger/draft-act.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

Modern web services face a fundamental tension between operational needs and user privacy. Services need to implement rate limiting to prevent abuse, charge for API usage to sustain operations, and allocate computational resources fairly. However, traditional approaches require tracking client identities and creating detailed logs of client behavior, raising significant privacy concerns in an era of increasing data protection awareness and regulation.

Anonymous Credit Tokens (ACT) helps to resolve this tension by providing a cryptographic protocol that enables credit-based systems without client tracking. Built on keyed-verification anonymous credentials [KVAC] and privately verifiable BBS-style signatures [BBS], the protocol allows services to issue, track, and spend credits while maintaining client privacy.

1.1. Key Properties

The protocol provides four essential properties that make it suitable for privacy-preserving credit systems:

1. *Unlinkability*: The issuer cannot link credit issuance to spending, or connect multiple transactions by the same client. This property is information-theoretic, not merely computational.

2. *Partial Spending*: Clients can spend any amount up to their balance and receive anonymous change without revealing their previous or current balance, enabling flexible spending.

3. *Double-Spend Prevention*: Cryptographic nullifiers ensure each token is used only once, without linking it to issuance.

4. *Balance Privacy*: During spending, only the amount being spent is revealed, not the total balance in the token, protecting clients from balance-based profiling.

5. *Performance*: The protocol's operations are performant enough to make it useful in modern web systems. This protocol has performance characteristics which make it suitable for a large number of applications.

1.2. Use Cases

Anonymous Credit Tokens can be applied to various scenarios:

* *Rate Limiting*: Services can issue daily credit allowances that clients spend anonymously for API calls or resource access.

* *API Credits*: API providers can sell credit packages that developers use to pay for API requests without creating a detailed usage history linked to their identity. This enables:

  - Pre-paid API access without requiring credit cards for each transaction

  - Anonymous API usage for privacy-sensitive applications

  - Usage-based billing without tracking individual request patterns

  - Protection against competitive analysis through usage monitoring

1.3. Protocol Overview

The protocol involves two parties: an issuer (typically a service provider) and clients (typically users of the service). The interaction follows three main phases:

1. *Setup*: The issuer generates a key pair and publishes the public key.

2. *Issuance*: A client requests credits from the issuer. The issuer creates a blind signature on the credit value and a client-chosen nullifier, producing a credit token.

3. *Spending*: To spend credits, the client reveals a nullifier and proves possession of a valid token associated with that nullifier having sufficient balance. The issuer verifies the proof, checks the nullifier hasn't been used before, and issues a new token (which remains hidden from the issuer) for any remaining balance.

1.4. Design Goals

The protocol is designed with the following goals:

* *Privacy*: The issuer cannot link credit tokens to specific clients or link multiple transactions by the same client.

* *Security*: Clients cannot spend more credits than they possess or use the same credits multiple times.

* *Efficiency*: All operations should be computationally efficient, suitable for high-volume web services.

* *Simplicity*: The protocol should be straightforward to implement and integrate into existing systems relative to other comparable solutions.

1.5. Relation to Existing Work

This protocol builds upon several cryptographic primitives:

* *BBS Signatures* [BBS]: The core signature scheme that enables efficient proofs of possession. We use a variant that is privately verifiable, which avoids the need for pairings and makes our protocol more efficient.

* *Sigma Protocols* [ORRU-SIGMA]: The zero-knowledge proof framework used for spending proofs.

* *Fiat-Shamir Transform* [ORRU-FS]: The technique to make the interactive proofs non-interactive.

The protocol can be viewed as a specialized instantiation of keyed-verification anonymous credentials [KVAC] optimized for numerical values and partial spending.

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2.1. Notation

This document uses the following notation:

* ||: Concatenation of byte strings

* x <- S: Sampling x uniformly from the set S

* x := y: Assignment of the value y to the variable x

* [n]: The set of integers {0, 1, ..., n-1}

* |x|: The length of byte string x

* 0x prefix: Hexadecimal values

* We use additive notation for group operations, so group elements are added together like a + b and scalar multiplication of a group element by a scalar is written as a * n, with group element a and scalar n.

2.2. Data Types

The protocol uses the following data types:

* *Scalar*: An integer modulo the group order q

* *Element*: An element of the Ristretto255 group

* *ByteString*: A sequence of bytes

2.3. Cryptographic Parameters

The protocol uses the Ristretto group [RFC9496], which provides a prime-order group abstraction over Curve25519. It would be easy to adapt this approach to using any other prime order group based on the contents of this document. The key parameters are:

* *q*: The prime order of the group (2^252 + 27742317777372353535851937790883648493)

* *G*: The standard generator of the Ristretto group

* *L*: The bit length for credit values

3. Protocol Specification

3.1. System Parameters

The protocol requires the following system parameters:

```
Parameters:
  - G: Generator of the Ristretto group
  - H1, H2, H3: Additional generators for commitments
  - L: Bit length for credit values (configurable, must satisfy L <= 252)
```

The generators H1, H2, and H3 MUST be generated deterministically from a nothing-up-my-sleeve value to ensure they are independent of each other and of G. This prevents attacks whereby malicious parameters could compromise security. Note that these generators are independent of the choice of L.

```
   GenerateParameters(domain_separator):
     Input:
       - domain_separator: ByteString identifying the deployment
     Output:
       - params: System parameters (H1, H2, H3)

     Steps:
       1. seed = BLAKE3(LengthPrefixed(domain_separator))
       2. counter = 0
       3. H1 = HashToRistretto255(seed, counter++)
       4. H2 = HashToRistretto255(seed, counter++)
       5. H3 = HashToRistretto255(seed, counter++)
       6. return (H1, H2, H3)

   HashToRistretto255(seed, counter):
     Input:
       - seed: 32-byte seed value
       - counter: Integer counter for domain separation
     Output:
       - P: A valid Ristretto255 point

     Steps:
       1. hasher = BLAKE3.new()
       2. hasher.update(LengthPrefixed(domain_separator))
       3. hasher.update(LengthPrefixed(seed))
       4. hasher.update(LengthPrefixed(counter.to_le_bytes(4)))
       5. uniform_bytes = hasher.finalize_xof(64)
       6. P = OneWayMap(uniform_bytes)
       7. return P
```

The domain_separator MUST be unique for each deployment to ensure cryptographic isolation between different services. The domain separator SHOULD follow this structured format:

```
domain_separator = "ACT-v1:" || organization || ":" || service || ":" || deployment_id ||
 ":" || version
```

Where:

* organization: A unique identifier for the organization (e.g., "example-corp", "acme-inc")

* service: The specific service or application name (e.g., "payment-api", "rate-limiter")

* deployment_id: The deployment environment (e.g., "production", "staging", "us-west-1")

* version: An ISO 8601 date (YYYY-MM-DD) indicating when parameters were generated

Example: "ACT-v1:example-corp:payment-api:production:2024-01-15"

This structured format ensures: 1. Protocol identification through the "ACT-v1:" prefix 2. Organizational namespace isolation 3. Service-level separation within organizations 4. Environment isolation (production vs staging) 5. Version tracking for parameter updates

Using generic or unstructured domain separators creates security risks through parameter collision and MUST NOT be used. When parameters need to be updated (e.g., for security reasons or protocol upgrades), a new version date MUST be used, creating entirely new parameters.

The OneWayMap function is defined in [RFC9496] Section 4.3.4, which provides a cryptographically secure mapping from uniformly random byte strings to valid Ristretto255 points.

3.2. Key Generation

The issuer generates a key pair as follows:

```
KeyGen():
  Input: None
  Output:
    - sk: Private key (Scalar)
    - pk: Public key (Group Element)

  Steps:
    1. x <- Zq
    2. W = G * x
    3. sk = x
    4. pk = W
    5. return (sk, pk)
```

3.3. Token Issuance

The issuance protocol is an interactive protocol between a client and the issuer:

3.3.1. Client: Issuance Request

```
   IssueRequest():
     Output:
       - request: Issuance request
       - state: Client state for later verification

     Steps:
       1. k <- Zq  // Nullifier (will prevent double-spending)
       2. r <- Zq  // Blinding factor
       3. K = H2 * k + H3 * r
       4. // Generate proof of knowledge of k, r
       5. k’ <- Zq
       6. r’ <- Zq
       7. K1 = H2 * k’ + H3 * r’
       8. transcript = CreateTranscript("request")
       9. AddToTranscript(transcript, K)
       10. AddToTranscript(transcript, K1)
       11. gamma = GetChallenge(transcript)
       12. k_bar = k’ + gamma * k
       13. r_bar = r’ + gamma * r
       14. request = (K, gamma, k_bar, r_bar)
       15. state = (k, r)
       16. return (request, state)
```

3.3.2. Issuer: Issuance Response

```
   Issue(sk, request, c):
     Input:
       - sk: Issuer’s private key
       - request: Client’s issuance request
       - c: Credit amount to issue (c > 0)
     Output:
       - response: Issuance response or INVALID

     Steps:
       1. Parse request as (K, gamma, k_bar, r_bar)
       2. // Verify proof of knowledge
       3. K1 = H2 * k_bar + H3 * r_bar - K * gamma
       4. transcript = CreateTranscript("request")
       5. AddToTranscript(transcript, K)
       6. AddToTranscript(transcript, K1)
       7. if GetChallenge(transcript) != gamma:
       8.     return INVALID
       9. // Create BBS signature on (c, k, r)
       10. e <- Zq
       11. A = (G + H1 * c + K) * (1/(e + sk))  // K = H2 * k + H3 * r
       12. // Generate proof of correct computation
       13. alpha <- Zq
       14. Y_A = A * alpha
       15. Y_G = G * alpha
       16. X_A = G + H1 * c + K
       17. X_G = G * e + pk
       18. transcript_resp = CreateTranscript("respond")
       19. AddToTranscript(transcript_resp, c)
       20. AddToTranscript(transcript_resp, e)
       21. AddToTranscript(transcript_resp, A)
       22. AddToTranscript(transcript_resp, X_A)
       23. AddToTranscript(transcript_resp, X_G)
       24. AddToTranscript(transcript_resp, Y_A)
       25. AddToTranscript(transcript_resp, Y_G)
       26. gamma_resp = GetChallenge(transcript_resp)
       27. z = gamma_resp * (sk + e) + alpha
       28. response = (A, e, gamma_resp, z, c)
       29. return response
```

3.3.3. Client: Token Verification

```
   VerifyIssuance(pk, request, response, state):
     Input:
       - pk: Issuer's public key
       - request: The issuance request sent
       - response: Issuer's response
       - state: Client state from request generation
     Output:
       - token: Credit token or INVALID

     Steps:
       1. Parse request as (K, gamma, k_bar, r_bar)
       2. Parse response as (A, e, gamma_resp, z, c)
       3. Parse state as (k, r)
       4. // Verify proof
       6. X_A = G + H1 * c + K
       7. X_G = G * e + pk
       8. Y_A = A * z - X_A * gamma_resp
       9. Y_G = G * z - X_G * gamma_resp
       10. transcript_resp = CreateTranscript("respond")
       11. AddToTranscript(transcript_resp, c)
       12. AddToTranscript(transcript_resp, e)
       13. AddToTranscript(transcript_resp, A)
       14. AddToTranscript(transcript_resp, X_A)
       15. AddToTranscript(transcript_resp, X_G)
       16. AddToTranscript(transcript_resp, Y_A)
       17. AddToTranscript(transcript_resp, Y_G)
       18. if GetChallenge(transcript_resp) != gamma_resp:
       19.     return INVALID
       20. token = (A, e, k, r, c)
       21. return token
```

3.4. Token Spending

The spending protocol allows a client to spend s credits from a token containing c credits (where 0 < s <= c):

3.4.1. Client: Spend Proof Generation

```
ProveSpend(token, s):
  Input:
    - token: Credit token (A, e, k, r, c)
    - s: Amount to spend (0 < s <= c)
  Output:
    - proof: Spend proof
    - state: Client state for receiving change

  Steps:
    1. // Randomize the signature
```

```
    2. r1, r2 <- Zq
    3. B = G + H1 * c + H2 * k + H3 * r
    4. A' = A * (r1 * r2)
    5. B_bar = B * r1
    6. r3 = 1/r1

    7. // Generate initial proof components
    8. c' <- Zq
    9. r' <- Zq
    10. e' <- Zq
    11. r2' <- Zq
    12. r3' <- Zq

    13. // Compute first round messages
    14. A1 = A' * e' + B_bar * r2'
    15. A2 = B_bar * r3' + H1 * c' + H3 * r'

    16. // Decompose c - s into bits
    17. m = c - s
    18. (i[0], ..., i[L-1]) = BitDecompose(m)  // See Section 3.7

    19. // Create commitments for each bit
    20. k* <- Zq
    21. s[0] <- Zq
    22. Com[0] = H1 * i[0] + H2 * k* + H3 * s[0]
    23. For j = 1 to L-1:
    24.     s[j] <- Zq
    25.     Com[j] = H1 * i[j] + H3 * s[j]

    26. // Initialize range proof arrays
    27. C = array[L][2]
    28. C' = array[L][2]
    29. gamma0 = array[L]
    30. z = array[L][2]

    31. // Process bit 0 (with k* component)
    32. C[0][0] = Com[0]
    33. C[0][1] = Com[0] - H1
    34. k0' <- Zq
    35. s_prime = array[L]
    36. s_prime[0] <- Zq
    37. gamma0[0] <- Zq
    38. w0 <- Zq
    39. z[0] <- Zq

    40. if i[0] == 0:
    41.     C'[0][0] = H2 * k0' + H3 * s_prime[0]
    42.     C'[0][1] = H2 * w0 + H3 * z[0] - C[0][1] * gamma0[0]
```

```
    else:
        C’[0][0] = H2 * w0 + H3 * z[0] - C[0][0] * gamma0[0]
        C’[0][1] = H2 * k0’ + H3 * s_prime[0]

// Process remaining bits
For j = 1 to L-1:
    C[j][0] = Com[j]
    C[j][1] = Com[j] - H1
    s_prime[j] <- Zq
    gamma0[j] <- Zq
    z[j] <- Zq

    if i[j] == 0:
        C’[j][0] = H3 * s_prime[j]
        C’[j][1] = H3 * z[j] - C[j][1] * gamma0[j]
    else:
        C’[j][0] = H3 * z[j] - C[j][0] * gamma0[j]
        C’[j][1] = H3 * s_prime[j]

// Compute K’ commitment
K’ = Sum(Com[j] * 2^j for j in [L])
r* = Sum(s[j] * 2^j for j in [L])
k’ <- Zq
s’ <- Zq
C = H1 * (-c’) + H2 * k’ + H3 * s’

// Generate challenge using transcript
transcript = CreateTranscript("spend")
AddToTranscript(transcript, k)
AddToTranscript(transcript, A’)
AddToTranscript(transcript, B_bar)
AddToTranscript(transcript, A1)
AddToTranscript(transcript, A2)
For j = 0 to L-1:
    AddToTranscript(transcript, Com[j])
For j = 0 to L-1:
    AddToTranscript(transcript, C’[j][0])
    AddToTranscript(transcript, C’[j][1])
AddToTranscript(transcript, C)
gamma = GetChallenge(transcript)

// Compute responses
e_bar = -gamma * e + e’
r2_bar = gamma * r2 + r2’
r3_bar = gamma * r3 + r3’
c_bar = -gamma * c + c’
r_bar = -gamma * r + r’
```

```
// Complete range proof responses
z_final = array[L][2]
gamma0_final = array[L]

// For bit 0
if i[0] == 0:
    gamma0_final[0] = gamma - gamma0[0]
    w00 = gamma0_final[0] * k* + k0'
    w01 = w0
    z_final[0][0] = gamma0_final[0] * s[0] + s_prime[0]
    z_final[0][1] = z[0]
else:
    gamma0_final[0] = gamma0[0]
    w00 = w0
    w01 = (gamma - gamma_final[0]) * k* + k'[0]
    z_final[0][0] = z[0]
    z_final[0][1] = (gamma - gamma0_final[0]) * s[0] + s_prime[0]

// For remaining bits
For j = 1 to L-1:
    if i[j] == 0:
        gamma0_final[j] = gamma - gamma0[j]
        z_final[j][0] = gamma0_final[j] * s[j] + s_prime[j]
        z_final[j][1] = z[j]
    else:
        gamma0_final[j] = gamma0[j]
        z_final[j][0] = z[j]
        z_final[j][1] = (gamma - gamma0_final[j]) * s[j] + s_prime[j]

k_bar = gamma * k* + k'
s_bar = gamma * r* + s'

// Construct proof
proof = (k, s, A', B_bar, Com, gamma, e_bar,
         r2_bar, r3_bar, c_bar, r_bar,
         w00, w01, gamma0_final, z_final,
         k_bar, s_bar)
state = (k*, r*, m)
return (proof, state)
```

3.4.2. Issuer: Spend Verification and Refund

```
VerifyAndRefund(sk, proof):
  Input:
    - sk: Issuer’s private key
    - proof: Client’s spend proof
  Output:
    - refund: Refund for remaining credits or INVALID

  Steps:
    1. Parse proof and extract nullifier k
    2. // Check nullifier hasn’t been used
    3. if k in used_nullifiers:
    4.     return INVALID
    5. // Verify the proof (see Section 3.5.2)
    6. if not VerifySpendProof(sk, proof):
    7.     return INVALID
    8. // Record nullifier
    9. used_nullifiers.add(k)
    10. // Issue refund for remaining balance
    11. K’ = Sum(Com[j] * 2^j for j in [L])
    12. refund = IssueRefund(sk, K’)
    13. return refund
```

3.4.3. Refund Issuance

After verifying a spend proof, the issuer creates a refund token for the remaining balance:

```
   IssueRefund(sk, K'):
     Input:
       - sk: Issuer's private key
       - K': Commitment to remaining balance and new nullifier
     Output:
       - refund: Refund response

     Steps:
       1. // Create new BBS signature on remaining balance
       2. e* <- Zq
       3. X_A* = G + K'
       4. A* = X_A* * (1/(e* + sk))

       5. // Generate proof of correct computation
       6. alpha <- Zq
       7. Y_A = A* * alpha
       8. Y_G = G * alpha
       9. X_G = G * e* + pk

       10. // Create challenge using transcript
       11. transcript = CreateTranscript("refund")
       12. AddToTranscript(transcript, e*)
       13. AddToTranscript(transcript, A*)
       14. AddToTranscript(transcript, X_A*)
       15. AddToTranscript(transcript, X_G)
       16. AddToTranscript(transcript, Y_A)
       17. AddToTranscript(transcript, Y_G)
       18. gamma = GetChallenge(transcript)

       19. // Compute response
       20. z = gamma * (sk + e*) + alpha

       21. refund = (A*, e*, gamma, z)
       22. return refund
```

3.4.4. Client: Refund Token Construction

The client verifies the refund and constructs a new credit token:

```
   ConstructRefundToken(pk, spend_proof, refund, state):
     Input:
       - pk: Issuer's public key
       - spend_proof: The spend proof sent to issuer
       - refund: Issuer's refund response
       - state: Client state (k*, r*, m)
     Output:
       - token: New credit token or INVALID

     Steps:
       1. Parse refund as (A*, e*, gamma, z)
       2. Parse state as (k*, r*, m)

       3. // Reconstruct commitment
       4. K' = Sum(spend_proof.Com[j] * 2^j for j in [L])
       5. X_A* = G + K'
       6. X_G = G * e* + pk

       7. // Verify proof
       8. Y_A = A* * z + X_A* * (-gamma)
       9. Y_G = G * z + X_G * (-gamma)

       10. // Check challenge using transcript
       11. transcript = CreateTranscript("refund")
       12. AddToTranscript(transcript, e*)
       13. AddToTranscript(transcript, A*)
       14. AddToTranscript(transcript, X_A*)
       15. AddToTranscript(transcript, X_G)
       16. AddToTranscript(transcript, Y_A)
       17. AddToTranscript(transcript, Y_G)
       18. if GetChallenge(transcript) != gamma:
       19.     return INVALID

       20. // Construct new token
       21. token = (A*, e*, k*, r*, m)
       22. return token
```

3.4.5. Spend Proof Verification

The issuer verifies a spend proof as follows:

```
VerifySpendProof(sk, proof):
  Input:
    - sk: Issuer's private key
    - proof: Spend proof from client
  Output:
    - valid: Boolean indicating if proof is valid
```

```
  Steps:
    1. Parse proof as (k, s, A', B_bar, Com, gamma, e_bar,
                       r2_bar, r3_bar, c_bar, r_bar, w00, w01,
                       gamma0, z, k_bar, s_bar)

    2. // Check A' is not identity
    3. if A' == Identity:
    4.     return false

    5. // Compute issuer's view of signature
    6. A_bar = A' * sk
    7. H1_prime = G + H2 * k

    8. // Verify sigma protocol
    9. A1 = A' * e_bar + B_bar * r2_bar - A_bar * gamma
    10. A2 = B_bar * r3_bar + H1 * c_bar + H3 * r_bar - H1_prime * gamma

    11. // Initialize arrays for range proof verification
    12. gamma1 = array[L]
    13. C = array[L][2]
    14. C' = array[L][2]

    15. // Process bit 0 (with k* component)
    16. gamma1[0] = gamma - gamma0[0]
    17. C[0][0] = Com[0]
    18. C[0][1] = Com[0] - H1
    19. C'[0][0] = H2 * w00 + H3 * z[0][0] - C[0][0] * gamma0[0]
    20. C'[0][1] = H2 * w01 + H3 * z[0][1] - C[0][1] * gamma1[0]

    21. // Verify remaining bits
    22. For j = 1 to L-1:
    23.     gamma1[j] = gamma - gamma0[j]
    24.     C[j][0] = Com[j]
    25.     C[j][1] = Com[j] - H1
    26.     C'[j][0] = H3 * z[j][0] - C[j][0] * gamma0[j]
    27.     C'[j][1] = H3 * z[j][1] - C[j][1] * gamma1[j]

    28. // Verify final commitment
    29. K' = Sum(Com[j] * 2^j for j in [L])
    30. Com_total = H1 * s + K'
    31. C_final = H1 * (-c_bar) + H2 * k_bar + H3 * s_bar - Com_total * gamma

    32. // Recompute challenge using transcript
    33. transcript = CreateTranscript("spend")
    34. AddToTranscript(transcript, k)
    35. AddToTranscript(transcript, A')
    36. AddToTranscript(transcript, B_bar)
    37. AddToTranscript(transcript, A1)
```

```
    AddToTranscript(transcript, A2)
    For j = 0 to L-1:
        AddToTranscript(transcript, Com[j])
    For j = 0 to L-1:
        AddToTranscript(transcript, C'[j][0])
        AddToTranscript(transcript, C'[j][1])
    AddToTranscript(transcript, C_final)
    gamma_check = GetChallenge(transcript)

    // Verify challenge matches
    if gamma != gamma_check:
        return false

    return true
```

3.5. Cryptographic Primitives

3.5.1. Protocol Version

The protocol version string for domain separation is:

PROTOCOL_VERSION = "curve25519-ristretto anonymous-credits v1.0"

This version string MUST be used consistently across all implementations for interoperability. The curve specification is included to prevent cross-curve attacks and ensure implementations using different curves cannot accidentally interact.

3.5.2. Hash Function and Fiat-Shamir Transform

The protocol uses BLAKE3 [BLAKE3] as the underlying hash function for the Fiat-Shamir transform [ORRU-FS]. Following the sigma protocol framework [ORRU-SIGMA], challenges are generated using a transcript that accumulates all protocol messages:

```
   CreateTranscript(label):
     Input:
       - label: ASCII string identifying the proof type
     Output:
       - transcript: A new transcript object

     Steps:
       1. hasher = BLAKE3.new()
       2. hasher.update(LengthPrefixed(PROTOCOL_VERSION))
       3. hasher.update(LengthPrefixed(Encode(H1)))
       4. hasher.update(LengthPrefixed(Encode(H2)))
       5. hasher.update(LengthPrefixed(Encode(H3)))
       6. hasher.update(LengthPrefixed(label))
       7. return transcript with hasher

   AddToTranscript(transcript, value):
     Input:
       - transcript: Existing transcript
       - value: Element or Scalar to add

     Steps:
       1. encoded = Encode(value)
       2. transcript.hasher.update(LengthPrefixed(encoded))

   GetChallenge(transcript):
     Input:
       - transcript: Completed transcript
     Output:
       - challenge: Scalar challenge value

     Steps:
       1. hash = transcript.hasher.output(64)  // 64 bytes of output
       3. challenge = from_little_endian_bytes(hash) mod q
       4. return challenge
```

This approach ensures:

* Domain separation through the label and protocol version

* Inclusion of all public parameters to prevent parameter substitution attacks

* Proper ordering with length prefixes to prevent ambiguity

* Deterministic challenge generation from the complete transcript

3.5.3. Encoding Functions

Elements and scalars are encoded as follows:

```
Encode(value):
  Input:
    - value: Element or Scalar
  Output:
    - encoding: ByteString

  Steps:
    1. If value is an Element:
    2.     return value.compress()  // 32 bytes, compressed Ristretto point
    3. If value is a Scalar:
    4.     return value.to_bytes_le()  // 32 bytes, little-endian
```

The following function provides consistent length-prefixing for hash inputs:

```
LengthPrefixed(data):
  Input:
    - data: ByteString to be length-prefixed
  Output:
    - prefixed: ByteString with length prefix

  Steps:
    1. length = len(data)
    2. return length.to_be_bytes(8) || data  // 8-byte big-endian length prefix
```

Note: Implementations MAY use standard serialization formats (e.g. CBOR) for complex structures, but MUST ensure deterministic encoding for hash inputs.

3.5.4. Binary Decomposition

To decompose a scalar into its binary representation:

```
BitDecompose(s):
  Input:
    - s: Scalar value
  Output:
    - bits: Array of L scalars (each 0 or 1)

  Steps:
    1. bytes = s.to_bytes_le()  // 32 bytes, little-endian
    2. For i = 0 to L-1:
    3.     byte_index = i / 8
    4.     bit_position = i % 8
    5.     bit = (bytes[byte_index] >> bit_position) & 1
    6.     bits[i] = Scalar(bit)
    7. return bits
```

Note: This algorithm produces bits in LSB-first order (i.e., bits[0] is the least significant bit). The algorithm works for any L < 252, as the scalar is represented in 32 bytes (256 bits), which accommodates the full range of the Ristretto group order.

3.5.5. Scalar Conversion

Converting between credit amounts and scalars:

```
   CreditToScalar(amount):
     Input:
       - amount: Integer credit amount (0 <= amount < 2^L)
     Output:
       - s: Scalar representation

     Steps:
       1. if amount >= 2^L:
       2.     return ERROR
       3. return Scalar(amount)

   ScalarToCredit(s):
     Input:
       - s: Scalar value
     Output:
       - amount: Integer credit amount or ERROR

     Steps:
       1. bytes = s.to_bytes_le()
       2. // Check high bytes are zero
       3. For i = 16 to 31:
       4.     if bytes[i] != 0:
       5.         return ERROR
       6. amount = bytes[0..15] as u128
       7. return amount
```

4. Protocol Messages and Wire Format

4.1. Message Encoding

All protocol messages SHOULD be encoded using deterministic CBOR (RFC 8949) for interoperability. The following sections define the structure of each message type.

4.1.1. Issuance Request Message

```
IssuanceRequestMsg = {
    1: bstr,  ; K (compressed Ristretto point, 32 bytes)
    2: bstr,  ; gamma (scalar, 32 bytes)
    3: bstr,  ; k_bar (scalar, 32 bytes)
    4: bstr   ; r_bar (scalar, 32 bytes)
}
```

4.1.2. Issuance Response Message

```
IssuanceResponseMsg = {
    1: bstr,  ; A (compressed Ristretto point, 32 bytes)
    2: bstr,  ; e (scalar, 32 bytes)
    3: bstr,  ; gamma_resp (scalar, 32 bytes)
    4: bstr,  ; z (scalar, 32 bytes)
    5: bstr   ; c (scalar, 32 bytes)
}
```

4.1.3. Spend Proof Message

```
SpendProofMsg = {
    1: bstr,           ; k (nullifier, 32 bytes)
    2: bstr,           ; s (spend amount, 32 bytes)
    3: bstr,           ; A' (compressed point, 32 bytes)
    4: bstr,           ; B_bar (compressed point, 32 bytes)
    5: [* bstr],       ; Com array (L compressed points)
    6: bstr,           ; gamma (scalar, 32 bytes)
    7: bstr,           ; e_bar (scalar, 32 bytes)
    8: bstr,           ; r2_bar (scalar, 32 bytes)
    9: bstr,           ; r3_bar (scalar, 32 bytes)
    10: bstr,          ; c_bar (scalar, 32 bytes)
    11: bstr,          ; r_bar (scalar, 32 bytes)
    12: bstr,          ; w00 (scalar, 32 bytes)
    13: bstr,          ; w01 (scalar, 32 bytes)
    14: [* bstr],      ; gamma0 array (L scalars)
    15: [* [bstr, bstr]], ; z array (L pairs of scalars)
    16: bstr,          ; k_bar (scalar, 32 bytes)
    17: bstr           ; s_bar (scalar, 32 bytes)
}
```

4.1.4. Refund Message

```
RefundMsg = {
    1: bstr,  ; A* (compressed Ristretto point, 32 bytes)
    2: bstr,  ; e* (scalar, 32 bytes)
    3: bstr,  ; gamma (scalar, 32 bytes)
    4: bstr   ; z (scalar, 32 bytes)
}
```

4.2. Error Responses

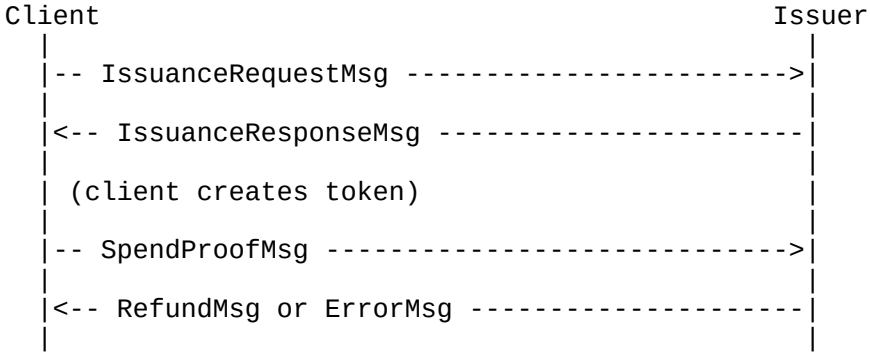
Error responses SHOULD use the following format:

```
ErrorMsg = {
    1: uint,   ; error_code
    2: tstr    ; error_message (for debugging only)
}
```

Error codes are defined in Section 5.3.

4.3. Protocol Flow

The complete protocol flow with message types:



4.3.1. Example Usage Scenario

Consider an API service that sells credits in bundles of 1000:

1. *Purchase*: Alice buys 1000 API credits
    * Alice generates a random nullifier k and blinding factor r
    * Alice sends IssuanceRequestMsg to the service
    * Service creates a BBS signature on (1000, k, r) and returns it
    * Alice now has a token worth 1000 credits
2. *First API Call*: Alice makes an API call costing 50 credits
    * Alice creates a SpendProofMsg proving she has 扮 · 50 credits
    * Alice reveals nullifier k to prevent double-spending
    * Service verifies the proof and records k as used
    * Service issues a RefundMsg for a new token worth 950 credits
    * Alice generates new nullifier k’ for the refund token
3. *Subsequent Calls*: Alice continues using the API
    * Each call repeats the spend/refund process

* Each new token has a fresh nullifier

* The service cannot link Alice’s calls together

This example demonstrates how the protocol maintains privacy while preventing double-spending and enabling flexible partial payments.

5. Implementation Considerations

5.1. Nullifier Management

Implementations MUST maintain a persistent database of used nullifiers to prevent double-spending. The nullifier storage requirements grow linearly with the number of spent tokens. Implementations MAY use the following strategies to manage storage:

1. *Expiration*: If tokens have expiration dates, old nullifiers can be pruned.

2. *Sharding*: Nullifiers can be partitioned across multiple databases.

3. *Bloom Filters*: Probabilistic data structures can reduce memory usage with a small false-positive rate.

5.2. Constant-Time Operations

To prevent timing attacks, implementations MUST use constant-time operations for:

* Scalar arithmetic

* Point operations

* Conditional selections in range proofs

In particular, the range proof generation MUST use constant-time conditional selection when choosing between bit values 0 and 1. The following pattern should be used:

```
ConstantTimeSelect(condition, value_if_true, value_if_false):
  // Returns value_if_true if condition is true (1),
  // value_if_false if condition is false (0)
  // Must execute in constant time regardless of condition
```

This is critical in the range proof generation where bit values must not leak through timing channels.

5.3. Randomness Generation

The security of the protocol critically depends on the quality of random number generation. Implementations MUST use cryptographically secure random number generators (CSPRNGs) for:

* Private key generation
* Blinding factors (r, k)
* Proof randomness (nonces)

5.3.1. RNG Requirements

1. *Entropy Source*: Use OS-provided entropy (e.g., /dev/urandom on Unix systems)
2. *Fork Safety*: Reseed after fork() to prevent nonce reuse
3. *Backtracking Resistance*: Use forward-secure PRNGs when possible

5.3.2. Nonce Generation

Following [ORRU-SIGMA], nonces (the randomness used in proofs) MUST be generated with extreme care:

1. *Fresh Randomness*: Generate new nonces for every proof
2. *No Reuse*: Never reuse nonces across different proofs
3. *Full Entropy*: Use the full security parameter (256 bits) of randomness
4. *Zeroization*: Clear nonces from memory after use

WARNING: Leakage of even a few bits of a nonce can allow complete recovery of the witness (secret values). Implementations MUST use constant-time operations and secure memory handling for all nonce-related computations.

5.4. Point Validation

All Ristretto points received from external sources MUST be validated:

1. *Deserialization*: Verify the point deserializes to a valid Ristretto point

2. *Non-Identity*: Verify the point is not the identity element

3. *Subgroup Check*: Ristretto guarantees prime-order subgroup membership

Example validation:

```
ValidatePoint(P):
  1. If P fails to deserialize:
  2.     return INVALID
  3. If P == Identity:
  4.     return INVALID
  5. // Ristretto ensures prime-order subgroup membership
  6. return VALID
```

All implementations MUST validate points at these locations:

* When receiving K in issuance request

* When receiving A in issuance response

* When receiving A’ and B_bar in spend proof

* When receiving Com[j] commitments in spend proof

* When receiving A* in refund response

5.5. Error Handling

Implementations SHOULD NOT provide detailed error messages that could leak information about the verification process. A single INVALID response should be returned for all verification failures.

5.5.1. Error Codes

While detailed error messages should not be exposed to untrusted parties, implementations MAY use the following internal error codes:

* INVALID_PROOF: Proof verification failed

* NULLIFIER_REUSE: Double-spend attempt detected

* MALFORMED_REQUEST: Request format is invalid

* INVALID_AMOUNT: Credit amount exceeds maximum (2^L - 1)

5.6. Parameter Selection

Implementations MUST choose L based on their maximum credit requirements and performance constraints. Note that L MUST be less than 252 to fit within the Ristretto group order.

The bit length L is configurable and determines the range of credit values (0 to 2^L - 1). The choice of L involves several trade-offs:

1. *Range*: Larger L supports higher credit values

2. *Performance*: Proof size and verification time scale linearly with L

3. *Security*: L must be less than the bit length of the group order (252 bits for Ristretto)

The implementation MUST enforce L < 252 to ensure proper scalar arithmetic within the group order.

5.6.1. Performance Characteristics

The protocol has the following computational complexity:

Notation for Operations:

* *Group Operations*: Point additions in the Ristretto255 group (e.g., P + Q)

* *Group Exponentiations*: Scalar multiplication of group elements (e.g., P * s)

* *Scalar Additions/Multiplications*: Arithmetic operations modulo the group order q

* *Issuance*:

Operation	Group Operations	Group Exponentiations	Scalar Additions	Scalar Multiplications	Hashes
Client Request	2	4	2	1	1
Issuer Response	5	8	3	1	2
Client Credit Token Construction	5	5	0	0	1

Table 1

* *Spending*:

Operation	Group Operations	Group Exponentiations	Scalar Additions	Scalar Multiplications
Client Request	17 + 4L	27 + 8L	13 + 5L	12 + 3L
Issuer Response	16 + 4L	24 + 5L	4 + L	1
Client Credit Token Construction	3	5	L	L

Table 2

Note: L is the configurable bit length for credit values.

* *Storage*:

Component	Size
Token size	160 bytes (5 テ 32 bytes)
Spend proof size	32 × (14 + 4L) bytes
Nullifier database entry	32 bytes per spent token

Table 3

Note: Token size is independent of L.

6. Security Considerations

6.1. Security Model and Definitions

6.1.1. Threat Model

We consider a setting with:

* Multiple issuers who can operate independently, though malicious issuers may collude with each other

* Potentially malicious clients who may attempt to spend more credits than they should (whether by forging tokens, spending more credits than a token has, or double-spending a token)

6.1.2. Security Properties

The protocol provides the following security guarantees:

1. *Unforgeability*: For an honest isser I, no probabilistic polynomial-time (PPT) adversary controlling a set of malicious clients and other malicious issuers can spend more credits than have been issued by I.

2. *Anonymity/Unlinkability*: For an honest client C, no adversary controlling a set of malicious issuers and other malicious clients can link a token issuance/refund to C with a token spend by C. This property is information-theoretic in nature.

6.2. Cryptographic Assumptions

Security relies on:

1. *The q-SDH Assumption* in the Ristretto255 group. We refer to [TZ23] for the formal definition.

2. *Random Oracle Model*: The BLAKE3 hash function H is modeled as a random oracle.

6.3. Privacy Properties

The protocol provides the following privacy guarantees:

1. *Unlinkability*: The issuer cannot link a token issuance/refund to a later spend of that token.

However, the protocol does NOT provide:

1. *Network-Level Privacy*: IP addresses and network metadata can still link transactions.

2. *Amount Privacy*: The spent amount s is revealed to the issuer.

3. *Timing Privacy*: Transaction timing patterns could potentially be used for correlation.

6.4. Security Properties

The protocol ensures:

1. *Unforgeability*: Clients cannot spend more credits than they have been issued by the issuer.

6.5. Implementation Vulnerabilities and Mitigations

6.5.1. Critical Security Requirements

1. *RNG Failures*: Weak randomness can completely break the protocol’s security.

   Attack Vector: Predictable or repeated nonces in proofs can allow complete recovery of secret values including private keys and token contents.

   Mitigations:

   * MUST use cryptographically secure RNGs (e.g., OS-provided entropy sources)

   * MUST reseed after fork() operations to prevent nonce reuse

* MUST implement forward-secure RNG state management

* SHOULD use separate RNG instances for different protocol components

* MUST zeroize RNG state on process termination

2. *Timing Attacks*: Variable-time operations can leak information about secret values.

   Attack Vector: Timing variations in scalar arithmetic or bit operations can reveal secret bit patterns, potentially exposing credit balances or allowing token forgery.

   Mitigations:

   * MUST use constant-time scalar arithmetic libraries

   * MUST use constant-time conditional selection for range proof conditionals

   * MUST avoid early-exit conditions based on secret values

   * Critical constant-time operations include:

     - Scalar multiplication and addition

     - Binary decomposition in range proofs

     - Conditional assignments based on secret bits

     - Challenge verification comparisons

3. *Nullifier Database Attacks*: Corruption or manipulation of the nullifier database enables double-spending.

   Attack Vectors:

   * Database corruption allowing nullifier deletion

   * Race conditions in concurrent nullifier checks

   Mitigations:

   * MUST use ACID-compliant database transactions

   * MUST check nullifier uniqueness within the same transaction as insertion

* SHOULD implement append-only audit logs for nullifier operations

* MUST implement proper database backup and recovery procedures

4. *Eavesdropping/Message Modification Attacks*: A network-level adversary can copy spend proofs or modify messages sent between an honest client and issuer.

   Attack Vectors:

   * Eavesdropping and copying of proofs

   * Message modifications causing protocol failure

   Mitigations:

   * Client and issuer MUST use TLS 1.3 or above when communicating.

5. *State Management Vulnerabilities*: Improper state handling can lead to security breaches.

   Attack Vectors:

   * State confusion between protocol sessions

   * Memory disclosure of sensitive state

   * Incomplete state cleanup

   Mitigations:

   * MUST use separate state objects for each protocol session

   * MUST zeroize all sensitive data (keys, nonces, intermediate values) after use

   * SHOULD use memory protection mechanisms (e.g., mlock) for sensitive data

   * MUST implement proper error handling that doesn’t leak state information

   * SHOULD use explicit state machines for protocol flow

6. *Concurrency and Race Conditions*: Parallel operations can introduce vulnerabilities.

Attack Vectors:

* TOCTOU (Time-of-check to time-of-use) vulnerabilities in nullifier checking

* Race conditions in balance updates

* Concurrent modification of shared state

Mitigations:

* MUST use appropriate locking for all shared resources

* MUST perform nullifier check and insertion atomically

* SHOULD document thread-safety guarantees

* MUST ensure atomic read-modify-write for all critical operations

6.6. Known Attack Scenarios

6.6.1. 1. Parallel Spend Attack

Scenario: A malicious client attempts to spend the same token multiple times by initiating parallel spend operations before any nullifier is recorded.

Prevention: The issuer MUST ensure atomic nullifier checking and recording within a single database transaction. Network-level rate limiting can provide additional protection.

6.6.2. 2. Balance Inflation Attack

Scenario: An attacker attempts to create a proof claiming to have more credits than actually issued by manipulating the range proof.

Prevention: The cryptographic soundness of the range proof prevents this attack.

6.6.3. 3. Token Linking Attack

Scenario: An issuer attempts to link transactions by analyzing patterns in nullifiers, amounts, or timing.

Prevention: Nullifiers are cryptographically random and unlinkable. However, implementations MAY add random delays and amount obfuscation where possible.

6.7. Protocol Composition and State Management

6.7.1. State Management Requirements

Before they make a spend request or an issue request, the client MUST store their private state (the nullifier, the blinding factor, and the new balance) durably.

For the issuer, the spend and refund operations MUST be treated as an atomic transaction. However, even more is required. If a nullifier associated with a given spend is persisted to the database, clients MUST be able to access the associated refund. If they cannot access this, then they can lose access to the rest of their credits. For performance reasons, an issuer SHOULD automatically clean these up after some expiry, but if they do so, they MUST inform the client of this policy so the client can ensure they can retry to retrieve the rest of their credits in time. Issuers MAY implement functionality to notify the issuer that the refund request was processed, so they can delete the refund record. It is not clear that this is worth the cost relative to just cleaning them up in bulk at some specified expiration date, however if you are memory constrained this could be useful.

6.7.2. Session Management

Each protocol session (issuance or spend/refund) MUST:

* Use fresh randomness

* Not reuse any random values across sessions

6.7.3. Version Negotiation

To support protocol evolution, implementations MAY include version negotiation in the initial handshake. All parties MUST agree on the protocol version before proceeding.

6.8. Quantum Resistance

This protocol is NOT quantum-resistant. The discrete logarithm problem can be solved efficiently by quantum computers using Shor’s algorithm. Organizations requiring long-term security should consider post-quantum alternatives. However, user privacy is preserved even in the presence of a cryptographically relevant quantum computer.

7. IANA Considerations

This document has no IANA actions.

8. References

8.1. Normative References

[BLAKE3] "BLAKE3: One Function, Fast Everywhere", 9 January 2020, <https://github.com/BLAKE3-team/BLAKE3-specs/blob/master/blake3.pdf>.

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

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

[RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <https://www.rfc-editor.org/rfc/rfc8949>.

[RFC9380] Faz-Hernandez, A., Scott, S., Sullivan, N., Wahby, R. S., and C. A. Wood, "Hashing to Elliptic Curves", RFC 9380, DOI 10.17487/RFC9380, August 2023, <https://www.rfc-editor.org/rfc/rfc9380>.

[RFC9496] de Valence, H., Grigg, J., Hamburg, M., Lovecruft, I., Tankersley, G., and F. Valsorda, "The ristretto255 and decaf448 Groups", RFC 9496, DOI 10.17487/RFC9496, December 2023, <https://www.rfc-editor.org/rfc/rfc9496>.

8.2. Informative References

[BBS] "Short Group Signatures", 2004, <https://crypto.stanford.edu/~dabo/pubs/papers/groupsigs.pdf>.

[KVAC] "Keyed-Verification Anonymous Credentials", 2014, <https://eprint.iacr.org/2013/516.pdf>.

[ORRU-FS] "The Fiat-Shamir Transform", 19 January 2025, <https://mmaker.github.io/draft-zkproof-sigma-protocols/draft-orru-zkproof-fiat-shamir.html>.

[ORRU-SIGMA]
"Sigma Protocols", 19 January 2025,
<https://www.ietf.org/archive/id/draft-orru-zkproof-sigma-protocols-00.txt>.

[RFC9474] 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>.

[TZ23] "Revisiting BBS Signatures", 2023, <https://eprint.iacr.org/2023/275>.

Appendix A. Test Vectors

This appendix provides test vectors for implementers to verify their implementations. All values are encoded in hexadecimal.

TODO

Appendix B. Implementation Status

This section records the status of known implementations of the protocol defined by this specification at the time of posting of this Internet-Draft, and is based on a proposal described in RFC 7942.

B.1. anonymous-credit-tokens

Organization: Google

Description: Reference implementation in Rust

Maturity: Beta

Coverage: Complete protocol implementation

License: Apache 2.0

Contact: sgschlesinger@gmail.com

URL: https://github.com/SamuelSchlesinger/anonymous-credit-tokens

Appendix C. Terminology Glossary

This glossary provides quick definitions of key terms used throughout this document:

ACT (Anonymous Credit Tokens): The privacy-preserving authentication protocol specified in this document.

Blind Signature: A cryptographic signature where the signer signs a message without seeing its content.

Refund: The refund issued for the remaining balance after a partial spend.

Credit: A numerical unit of authorization that can be spent by clients.

Domain Separator: A unique string used to ensure cryptographic isolation between different deployments.

Element: A point in the Ristretto255 elliptic curve group.

Issuer: The entity that creates and signs credit tokens.

Nullifier: A unique value revealed during spending that prevents double-spending of the same token.

Partial Spending: The ability to spend less than the full value of a token and receive change.

Scalar: An integer modulo the group order q, used in cryptographic operations.

Sigma Protocol: An interactive zero-knowledge proof protocol following a commit-challenge-response pattern.

Token: A cryptographic credential containing a BBS signature and associated data (A, e, k, r, c).

Unlinkability: The property that transactions cannot be correlated with each other or with token issuance.

Appendix D. Acknowledgments

The authors would like to thank the Crypto Forum Research Group for their valuable feedback and suggestions. Special thanks to the contributors who provided implementation guidance and security analysis.

This work builds upon the foundational research in anonymous credentials and zero-knowledge proofs by numerous researchers in the cryptographic community, particularly the work on BBS signatures by Boneh, Boyen, and Shacham, and keyed-verification anonymous credentials by Chase, Meiklejohn, and Zaverucha.

Authors' Addresses

Samuel Schlesinger
Google
Email: samschlesinger@google.com

Jonathan Katz
Google
Email: jkcrypto@google.com