

Network Working Group  
Internet-Draft  
Intended status: Informational  
Expires: 17 August 2026

S. Schlesinger  
J. Katz  
Google  
13 February 2026

Anonymous Credit Tokens  
draft-schlesinger-cfrg-act-01

## 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

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/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 17 August 2026.

## Copyright Notice

Copyright (c) 2026 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

## Table of Contents

1. Introduction . . . . .	4
1.1. Key Properties . . . . .	4
1.2. Use Cases . . . . .	5
1.3. Protocol Overview . . . . .	5
1.4. Design Goals . . . . .	6
1.5. Relation to Existing Work . . . . .	6
2. Conventions and Definitions . . . . .	7
2.1. Notation . . . . .	7
2.2. Data Types . . . . .	7
2.3. Cryptographic Parameters . . . . .	7
3. Protocol Specification . . . . .	8
3.1. System Parameters . . . . .	8
3.2. Key Generation . . . . .	10
3.3. Token Issuance . . . . .	10
3.3.1. Client: Issuance Request . . . . .	10

3.3.2.	Issuer: Issuance Response . . . . .	11
3.3.3.	Client: Token Verification . . . . .	12
3.4.	Token Spending . . . . .	13
3.4.1.	Client: Spend Proof Generation . . . . .	13
3.4.2.	Issuer: Spend Verification and Refund . . . . .	17
3.4.3.	Refund Issuance . . . . .	17
3.4.4.	Client: Refund Token Construction . . . . .	19
3.4.5.	Spend Proof Verification . . . . .	20
3.5.	Cryptographic Primitives . . . . .	21
3.5.1.	Protocol Version . . . . .	21
3.5.2.	Hash Function and Fiat-Shamir Transform . . . . .	22
3.5.3.	Encoding Functions . . . . .	23
3.5.4.	Binary Decomposition . . . . .	23
3.5.5.	Scalar Conversion . . . . .	24
4.	Protocol Messages and Wire Format . . . . .	25
4.1.	Message Encoding . . . . .	25
4.1.1.	Issuance Request Message . . . . .	25
4.1.2.	Issuance Response Message . . . . .	25
4.1.3.	Spend Proof Message . . . . .	25
4.1.4.	Refund Message . . . . .	26
4.2.	Error Responses . . . . .	26
4.3.	Protocol Flow . . . . .	26
4.3.1.	Example Usage Scenario . . . . .	27
5.	Implementation Considerations . . . . .	28
5.1.	Nullifier Management . . . . .	28
5.2.	Constant-Time Operations . . . . .	28
5.3.	Randomness Generation . . . . .	28
5.3.1.	RNG Requirements . . . . .	28
5.3.2.	Nonce Generation . . . . .	29
5.4.	Point Validation . . . . .	29
5.5.	Error Handling . . . . .	30
5.5.1.	Error Codes . . . . .	30
5.6.	Parameter Selection . . . . .	30
5.6.1.	Performance Characteristics . . . . .	30
6.	Security Considerations . . . . .	32
6.1.	Security Model and Definitions . . . . .	32
6.1.1.	Threat Model . . . . .	32
6.1.2.	Security Properties . . . . .	32
6.2.	Cryptographic Assumptions . . . . .	33
6.3.	Privacy Properties . . . . .	33
6.4.	Implementation Vulnerabilities and Mitigations . . . . .	33
6.4.1.	Critical Security Requirements . . . . .	33
6.5.	Known Attack Scenarios . . . . .	36
6.5.1.	1. Parallel Spend Attack . . . . .	36
6.5.2.	2. Balance Inflation Attack . . . . .	36
6.5.3.	3. Token Linking Attack . . . . .	37
6.6.	Protocol Composition and State Management . . . . .	37
6.6.1.	State Management Requirements . . . . .	37

6.6.2. Session Management . . . . .	37
6.6.3. Version Negotiation . . . . .	37
6.7. Quantum Resistance . . . . .	38
7. IANA Considerations . . . . .	38
8. References . . . . .	38
8.1. Normative References . . . . .	38
8.2. Informative References . . . . .	38
Appendix A. Test Vectors . . . . .	39
A.1. Parameters . . . . .	39
A.2. Key Generation . . . . .	40
A.3. Issuance . . . . .	40
A.4. Spending . . . . .	40
A.5. Refund . . . . .	42
A.6. Refund Token . . . . .	42
Appendix B. Implementation Status . . . . .	43
B.1. anonymous-credit-tokens . . . . .	43
Appendix C. Terminology Glossary . . . . .	43
Appendix D. Acknowledgments . . . . .	44
Authors' Addresses . . . . .	45

## 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) help 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.

## 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\***: Unlinkability between issuance and spending; see the Security Properties section for the formal definition.
- \* **\*Security\***: Clients cannot spend more credits than they possess or use the same credits multiple times.
- \* **\*Efficiency\***: All operations should be computationally efficient, with performance characteristics suitable for high-volume web services and a large number of applications.
- \* **\*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, H4: Additional generators for commitments
- L: Bit length for credit values (configurable, must satisfy  $1 \leq L \leq 128$ )

Implementations MUST enforce  $1 \leq L \leq 128$ . See the Parameter Selection section for the rationale behind this constraint.

The generators H1, H2, H3, and H4 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, H4)

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. H4 = HashToRistretto255(seed, counter++)
7. return (H1, H2, H3, H4)

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
```

Each component (organization, service, deployment\_id, version) MUST NOT contain the colon character ':'.

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.  $x \leftarrow \mathbb{Z}_q$   
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.  $k \leftarrow Z_q$  // Nullifier (will prevent double-spending)
2.  $r \leftarrow Z_q$  // Blinding factor
3.  $K = H2 * k + H3 * r$
4. // Generate proof of knowledge of  $k, r$
5.  $k' \leftarrow Z_q$
6.  $r' \leftarrow Z_q$
7.  $K1 = H2 * k' + H3 * r'$
8.  $\text{transcript} = \text{CreateTranscript}(\text{"request"})$
9.  $\text{AddToTranscript}(\text{transcript}, K)$
10.  $\text{AddToTranscript}(\text{transcript}, K1)$
11.  $\text{gamma} = \text{GetChallenge}(\text{transcript})$
12.  $k\_bar = k' + \text{gamma} * k$
13.  $r\_bar = r' + \text{gamma} * r$
14.  $\text{request} = (K, \text{gamma}, k\_bar, r\_bar)$
15.  $\text{state} = (k, r)$
16.  $\text{return} (\text{request}, \text{state})$

### 3.3.2. Issuer: Issuance Response

IssueResponse(sk, request, c, ctx):

Input:

- sk: Issuer's private key
- request: Client's issuance request
- c: Credit amount to issue ( $c > 0$ )
- ctx: Request context (Scalar)

Output:

- response: Issuance response or INVALID

Exceptions:

- InvalidIssuanceRequestProof, raised when the client proof verification fails

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.     raise InvalidIssuanceRequestProof
9. // Create BBS signature on (c, ctx, k, r)
10.  $e \leftarrow Zq$
11.  $A = (G + H1 * c + H4 * ctx + K) * (1/(e + sk))$  //  $K = H2 * k + H3 * r$
12. // Generate proof of correct computation
13.  $\alpha \leftarrow Zq$
14.  $Y\_A = A * \alpha$
15.  $Y\_G = G * \alpha$
16.  $X\_A = G + H1 * c + H4 * ctx + K$
17.  $X\_G = G * e + pk$
18. transcript\_resp = CreateTranscript("respond")
19. AddToTranscript(transcript\_resp, c)
20. AddToTranscript(transcript\_resp, ctx)
21. AddToTranscript(transcript\_resp, e)
22. AddToTranscript(transcript\_resp, A)
23. AddToTranscript(transcript\_resp, X\_A)
24. AddToTranscript(transcript\_resp, X\_G)
25. AddToTranscript(transcript\_resp, Y\_A)
26. AddToTranscript(transcript\_resp, Y\_G)
27. gamma\_resp = GetChallenge(transcript\_resp)
28.  $z = gamma\_resp * (sk + e) + \alpha$
29. response = (A, e, gamma\_resp, z, c, ctx)
30. 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

Exceptions:

- InvalidIssuanceResponseProof, raised when the server proof verification fails

Steps:

1. Parse request as (K, gamma, k\_bar, r\_bar)
2. Parse response as (A, e, gamma\_resp, z, c, ctx)
3. Parse state as (k, r)
4. // Verify proof
5.  $X_A = G + H1 * c + H4 * ctx + K$
6.  $X_G = G * e + pk$
7.  $Y_A = A * z - X_A * gamma\_resp$
8.  $Y_G = G * z - X_G * gamma\_resp$
9. transcript\_resp = CreateTranscript("respond")
10. AddToTranscript(transcript\_resp, c)
11. AddToTranscript(transcript\_resp, ctx)
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.     raise InvalidIssuanceResponseProof
20. token = (A, e, k, r, c, ctx)
21. return token

### 3.4. Token Spending

The spending protocol allows a client to spend  $s$  credits from a token containing  $c$  credits (where  $0 \leq s \leq c$ ).

Note: Spending  $s = 0$  is permitted and produces a new token with the same balance but a fresh nullifier. This "re-anonymization" operation is useful for securely transferring a token to another party: after a zero-spend, the original holder can no longer use the old nullifier, and the recipient obtains a token that is cryptographically unlinkable to the original.

#### 3.4.1. Client: Spend Proof Generation

ProveSpend(token, s):

Input:

- token: Credit token (A, e, k, r, c, ctx)
- s: Amount to spend ( $0 \leq s \leq c$ )

Output:

- proof: Spend proof
- state: Client state for receiving change

Exceptions:

- InvalidAmount: raised when  $s > c$  or  $s \geq 2^L$  or  $c \geq 2^L$

Steps:

1. // Validate inputs
2. if  $s \geq 2^L$ :
3.     raise InvalidAmount
4. if  $c \geq 2^L$ :
5.     raise InvalidAmount
6. if  $s > c$ :
7.     raise InvalidAmount
  
8. // Randomize the signature
9.  $r_1, r_2 \leftarrow Z_q$
10.  $B = G + H_1 * c + H_2 * k + H_3 * r + H_4 * ctx$
11.  $A' = A * (r_1 * r_2)$
12.  $B\_bar = B * r_1$
13.  $r_3 = 1/r_1$
  
14. // Generate initial proof components
15.  $c' \leftarrow Z_q$
16.  $r' \leftarrow Z_q$
17.  $e' \leftarrow Z_q$
18.  $r_2' \leftarrow Z_q$
19.  $r_3' \leftarrow Z_q$
  
20. // Compute first round messages
21.  $A_1 = A' * e' + B\_bar * r_2'$
22.  $A_2 = B\_bar * r_3' + H_1 * c' + H_3 * r'$
  
23. // Decompose  $c - s$  into bits
24.  $m = c - s$
25.  $(i[0], \dots, i[L-1]) = \text{BitDecompose}(m)$  // See Section 3.7
  
26. // Create commitments for each bit
27.  $k^* \leftarrow Z_q$
28.  $s[0] \leftarrow Z_q$
29.  $Com[0] = H_1 * i[0] + H_2 * k^* + H_3 * s[0]$
30. For  $j = 1$  to  $L-1$ :
31.      $s[j] \leftarrow Z_q$
32.      $Com[j] = H_1 * i[j] + H_3 * s[j]$

```

33. // Initialize range proof arrays
34. C = array[L][2]
35. C' = array[L][2]
36. gamma0 = array[L]
37. z = array[L][2]

38. // Process bit 0 (with k* component)
39. C[0][0] = Com[0]
40. C[0][1] = Com[0] - H1
41. k0' <- Zq
42. s_prime = array[L]
43. s_prime[0] <- Zq
44. gamma0[0] <- Zq
45. w0 <- Zq
46. z[0] <- Zq

47. if i[0] == 0:
48.     C'[0][0] = H2 * k0' + H3 * s_prime[0]
49.     C'[0][1] = H2 * w0 + H3 * z[0] - C[0][1] * gamma0[0]
50. else:
51.     C'[0][0] = H2 * w0 + H3 * z[0] - C[0][0] * gamma0[0]
52.     C'[0][1] = H2 * k0' + H3 * s_prime[0]

53. // Process remaining bits
54. For j = 1 to L-1:
55.     C[j][0] = Com[j]
56.     C[j][1] = Com[j] - H1
57.     s_prime[j] <- Zq
58.     gamma0[j] <- Zq
59.     z[j] <- Zq
60.
61.     if i[j] == 0:
62.         C'[j][0] = H3 * s_prime[j]
63.         C'[j][1] = H3 * z[j] - C[j][1] * gamma0[j]
64.     else:
65.         C'[j][0] = H3 * z[j] - C[j][0] * gamma0[j]
66.         C'[j][1] = H3 * s_prime[j]

67. // Compute K' commitment
68. K' = Sum(Com[j] * 2^j for j in [L])
69. r* = Sum(s[j] * 2^j for j in [L])
70. k' <- Zq
71. s' <- Zq
72. C_final = H1 * (-c') + H2 * k' + H3 * s'

73. // Generate challenge using transcript
74. transcript = CreateTranscript("spend")
75. AddToTranscript(transcript, k)

```

```

76. AddToTranscript(transcript, ctx)
77. AddToTranscript(transcript, A')
78. AddToTranscript(transcript, B_bar)
79. AddToTranscript(transcript, A1)
80. AddToTranscript(transcript, A2)
81. For j = 0 to L-1:
82.     AddToTranscript(transcript, Com[j])
83. For j = 0 to L-1:
84.     AddToTranscript(transcript, C'[j][0])
85.     AddToTranscript(transcript, C'[j][1])
86. AddToTranscript(transcript, C_final)
87. gamma = GetChallenge(transcript)

88. // Compute responses
89. e_bar = -gamma * e + e'
90. r2_bar = gamma * r2 + r2'
91. r3_bar = gamma * r3 + r3'
92. c_bar = -gamma * c + c'
93. r_bar = -gamma * r + r'

94. // Complete range proof responses
95. z_final = array[L][2]
96. gamma0_final = array[L]
97.
98. // For bit 0
99. if i[0] == 0:
100.     gamma0_final[0] = gamma - gamma0[0]
101.     w00 = gamma0_final[0] * k* + k0'
102.     w01 = w0
103.     z_final[0][0] = gamma0_final[0] * s[0] + s_prime[0]
104.     z_final[0][1] = z[0]
105. else:
106.     gamma0_final[0] = gamma0[0]
107.     w00 = w0
108.     w01 = (gamma - gamma0_final[0]) * k* + k0'
109.     z_final[0][0] = z[0]
110.     z_final[0][1] = (gamma - gamma0_final[0]) * s[0] + s_prime[0]

111. // For remaining bits
112. For j = 1 to L-1:
113.     if i[j] == 0:
114.         gamma0_final[j] = gamma - gamma0[j]
115.         z_final[j][0] = gamma0_final[j] * s[j] + s_prime[j]
116.         z_final[j][1] = z[j]
117.     else:
118.         gamma0_final[j] = gamma0[j]
119.         z_final[j][0] = z[j]
120.         z_final[j][1] = (gamma - gamma0_final[j]) * s[j] + s_prime[j]

```

```

121. k_bar = gamma * k* + k'
122. s_bar = gamma * r* + s'

123. // Construct proof
124. proof = (k, s, ctx, A', B_bar, Com, gamma, e_bar,
125.         r2_bar, r3_bar, c_bar, r_bar,
126.         w00, w01, gamma0_final, z_final,
127.         k_bar, s_bar)
128. state = (k*, r*, m, ctx)
129. return (proof, state)

```

#### 3.4.2. Issuer: Spend Verification and Refund

VerifyAndRefund(sk, proof, t):

Input:

- sk: Issuer's private key
- proof: Client's spend proof
- t: Credits to return to the client ( $0 \leq t \leq s$ ,  $t < 2^L$ )

Output:

- refund: Refund for remaining credits

Exceptions:

- DoubleSpendError: raised when the nullifier has been used before
- InvalidSpendProof: raised when the spend proof verification fails

Steps:

1. Parse proof and extract nullifier k
2. // Check nullifier hasn't been used
3. if k in used\_nullifiers:
4.     raise DoubleSpendError
5. // Verify the proof (see Section 3.5.2)
6. if not VerifySpendProof(sk, proof):
7.     raise InvalidSpendProof
8. // Record nullifier
9. used\_nullifiers.add(k)
10. // Issue refund for remaining balance
11.  $K' = \text{Sum}(\text{Com}[j] * 2^j \text{ for } j \text{ in } [L])$
12. refund = IssueRefund(sk, K', proof.ctx, proof.s, t)
13. return refund

#### 3.4.3. Refund Issuance

After verifying a spend proof, the issuer creates a refund token for the remaining balance. The issuer may optionally return  $t$  credits (where  $0 \leq t \leq s$ ) back to the client via a partial credit return. This enables pre-authorization patterns where the client holds  $s$  credits but only  $t$  are returned unused. The resulting token will have  $c - s + t$  credits. Use  $t = 0$  to consume the full spend amount:

IssueRefund(sk, K', ctx, s, t):

Input:

- sk: Issuer's private key
- K': Commitment to remaining balance and new nullifier
- ctx: Request context from the spend proof
- s: The spend amount from the proof
- t: Credits to return to the client ( $0 \leq t \leq s$ ,  $t < 2^L$ )

Output:

- refund: Refund response

Exceptions:

- InvalidAmount: raised when  $t > s$  or  $t \geq 2^L$

Steps:

1. // Validate partial return amount
2. if  $t \geq 2^L$ :
3.     raise InvalidAmount
4. if  $t > s$ :
5.     raise InvalidAmount
6. // Create new BBS signature on remaining balance + partial return
7.  $e^* \leftarrow Zq$
8.  $X\_A^* = G + K' + H1 * t + H4 * ctx$
9.  $A^* = X\_A^* * (1/(e^* + sk))$
10. // Generate proof of correct computation
11.  $\alpha \leftarrow Zq$
12.  $Y\_A = A^* * \alpha$
13.  $Y\_G = G * \alpha$
14.  $X\_G = G * e^* + pk$
15. // Create challenge using transcript
16. transcript = CreateTranscript("refund")
17. AddToTranscript(transcript,  $e^*$ )
18. AddToTranscript(transcript, t)
19. AddToTranscript(transcript, ctx)
20. AddToTranscript(transcript,  $A^*$ )
21. AddToTranscript(transcript,  $X\_A^*$ )
22. AddToTranscript(transcript,  $X\_G$ )
23. AddToTranscript(transcript,  $Y\_A$ )
24. AddToTranscript(transcript,  $Y\_G$ )
25.  $\gamma = \text{GetChallenge}(\text{transcript})$
26. // Compute response
27.  $z = \gamma * (sk + e^*) + \alpha$
28. refund = ( $A^*$ ,  $e^*$ ,  $\gamma$ , z, t)
29. 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 ( $k^*$ ,  $r^*$ ,  $m$ ,  $ctx$ )

Output:

- token: New credit token or INVALID

Exceptions:

- InvalidRefundProof: When the refund proof verification fails

Steps:

1. Parse refund as ( $A^*$ ,  $e^*$ ,  $\gamma$ ,  $z$ ,  $t$ )
2. Parse state as ( $k^*$ ,  $r^*$ ,  $m$ ,  $ctx$ )
3. // Reconstruct commitment with partial return
4.  $K' = \text{Sum}(\text{spend\_proof.Com}[j] * 2^j \text{ for } j \text{ in } [L])$
5.  $X\_A^* = G + K' + H1 * t + H4 * ctx$
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.  $\text{transcript} = \text{CreateTranscript}(\text{"refund"})$
12.  $\text{AddToTranscript}(\text{transcript}, e^*)$
13.  $\text{AddToTranscript}(\text{transcript}, t)$
14.  $\text{AddToTranscript}(\text{transcript}, ctx)$
15.  $\text{AddToTranscript}(\text{transcript}, A^*)$
16.  $\text{AddToTranscript}(\text{transcript}, X\_A^*)$
17.  $\text{AddToTranscript}(\text{transcript}, X\_G)$
18.  $\text{AddToTranscript}(\text{transcript}, Y\_A)$
19.  $\text{AddToTranscript}(\text{transcript}, Y\_G)$
20. if  $\text{GetChallenge}(\text{transcript}) \neq \gamma$ :
21.     raise InvalidRefundProof
22. // Construct new token with remaining balance + partial return
23.  $\text{token} = (A^*, e^*, k^*, r^*, m + t, ctx)$
24. 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

Exceptions:

- IdentityPointError: raised when A' is the identity
- InvalidClientSpendProof: raised when the challenge does not match the reconstruction

n

Steps:

1. Parse proof as (k, s, ctx, 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.     raise IdentityPointError
5. // Compute issuer's view of signature
6. A\_bar = A' \* sk
7. H1\_prime = G + H2 \* k + H4 \* ctx
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, ctx)
36. AddToTranscript(transcript, A')
37. AddToTranscript(transcript, B_bar)
38. AddToTranscript(transcript, A1)
39. AddToTranscript(transcript, A2)
40. For j = 0 to L-1:
41.     AddToTranscript(transcript, Com[j])
42. For j = 0 to L-1:
43.     AddToTranscript(transcript, C'[j][0])
44.     AddToTranscript(transcript, C'[j][1])
45. AddToTranscript(transcript, C_final)
46. gamma_check = GetChallenge(transcript)

47. // Verify challenge matches
48. if gamma != gamma_check:
49.     raise InvalidVerifySpendProof

50. 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(Encode(H4)))
7. hasher.update(LengthPrefixed(label))
8. 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
2. challenge = from\_little\_endian\_bytes(hash) mod q
3. 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). See Section 3.1 for constraints on L.

### 3.5.5. Scalar Conversion

Converting between credit amounts and scalars:

```
CreditToScalar(amount):
  Input:
    - amount: Integer credit amount ( $0 \leq \text{amount} < 2^L$ )
  Output:
    - s: Scalar representation
  Exceptions:
    - AmountTooBigError: raised when the amount exceeds  $2^L$ 

  Steps:
    1. if amount  $\geq 2^L$ :
    2.     return AmountTooBigError
    3. return Scalar(amount)
```

```
ScalarToCredit(s):
  Input:
    - s: Scalar value
  Output:
    - amount: Integer credit amount or ERROR
  Exceptions:
    - ScalarOutOfRangeError: raised when the scalar value is  $\geq 2^L$ 

  Steps:
    1. amount = s as integer // Interpret little-endian scalar bytes as integer
    2. if amount  $\geq 2^L$ :
    3.     return ScalarOutOfRangeError
    4. 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. Decoders MUST reject messages containing unknown CBOR map keys. 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)  
  6: bstr  ; ctx (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)
  18: bstr           ; ctx (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)
  5: bstr  ; t (partial return, scalar, 32 bytes)
}

```

#### 4.2. Error Responses

Error responses SHOULD use the following format:

```

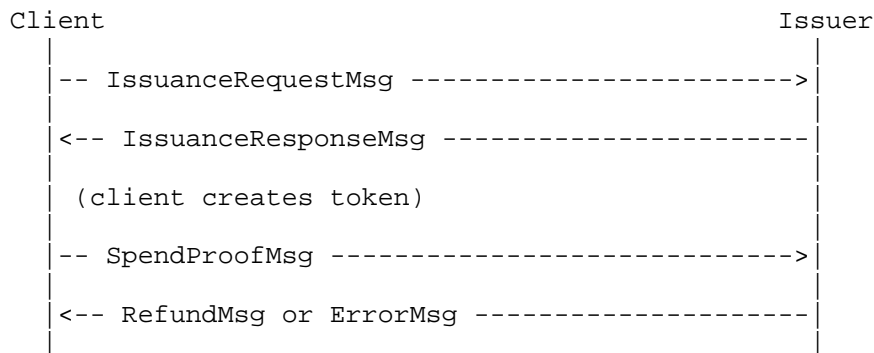
ErrorMsg = {
  1: uint, ; error_code
  2: tstr  ; error_message (for debugging only)
}

```

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 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. **WARNING**: false positives cause legitimate spends to be rejected. Bloom filters **MUST NOT** be the sole nullifier check; a positive result **MUST** be confirmed against authoritative storage before rejecting a spend.

### 5.2. Constant-Time Operations

Implementations **MUST** use constant-time operations for all secret-dependent computations. See the Security Considerations section for detailed requirements and mitigations.

### 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 is invalid (exceeds  $2^L - 1$ , spend exceeds balance, or partial return exceeds spend)

## 5.6. Parameter Selection

Implementations MUST choose L based on their maximum credit requirements and performance constraints. See Section 3.1 for constraints on L.

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

### 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	Hashes
Client Request	$17 + 4L$	$27 + 8L$	$13 + 5L$	$12 + 3L$	1
Issuer Response	$16 + 4L$	$24 + 5L$	$4 + L$	1	1
Client Credit Token Construction	3	5	$L$	$L$	1

Table 2

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

## \* \*Storage\*:

Component	Size
Token size	192 bytes (6 $\times$ 32 bytes)
Spend proof size	$32 \times (14 + 4L)$ 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 issuer 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.
4. *\*Context Privacy\**: The request context ( $ctx$ ) is revealed in the clear during spending. If the issuer assigns distinct  $ctx$  values per issuance, the resulting token chain (issuance, spend, refund, subsequent spends) becomes linkable through the shared  $ctx$  value. This is by design for application-level context binding, but deployments that require full unlinkability MUST use a shared  $ctx$  across all clients within the same context (e.g., per-service or per-epoch), not per-client values. The  $ctx$  value persists across refunds: a token produced by a refund inherits the  $ctx$  of the original token.

## 6.4. Implementation Vulnerabilities and Mitigations

### 6.4.1. 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

6.5. Known Attack Scenarios

6.5.1. 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.

6.5.2. 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.

### 6.5.3. 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.

## 6.6. Protocol Composition and State Management

### 6.6.1. 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.

### 6.6.2. Session Management

Each protocol session (issuance or spend/refund) MUST use fresh randomness. See the Randomness Generation section for detailed RNG requirements.

### 6.6.3. 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.

## 6.7. 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.

The following test vector was generated deterministically using a ChaCha20 RNG seeded with the bytes 00 01 02 ... 1e 1f and L=8. The domain separator is "ACT-v1:test:vectors:v0:2025-01-01", credit amount c=100, spend amount s=30, partial return t=10, and ctx=0. Values labelled \*\_cbor are the CBOR wire-format encodings (Section 4) of each protocol message, displayed in hexadecimal.

Implementations SHOULD verify they can deserialize these CBOR messages and that a full protocol run with the same deterministic RNG produces identical output.

### A.1. Parameters

```
domain_separator: "ACT-v1:test:vectors:v0:2025-01-01"
L: 8
c: 100
s: 30
t: 10
ctx: 0000000000000000000000000000000000000000000000000000000000000000
```

## A.2. Key Generation

sk\_cbor:

```
a201582036e5b43419551a92c809a995a3d2c817a86ce8f5dd973b06fe9cb5a3
f012870b0258204aceeb1d507e50957db46b6bcd374614b8ea080cbbc77ad060
666bf5788c8121
```

pk\_cbor:

```
58204aceeb1d507e50957db46b6bcd374614b8ea080cbbc77ad060666bf5788c
8121
```

## A.3. Issuance

preissuance\_cbor:

```
a20158206102398efee33b886f4bb7042b897d83db59b71a05aff76e9b633b87
cade7d0002582069e5d557cb6094acfa586118e602e90aa6fe6cbabd4571eeb0
d2f63b8c8a8f07
```

issuance\_request\_cbor:

```
a4015820aa9315999f76c89406fe743dc7ff12e8fab85871f8c36987c6ec25ee
ca2cd84e025820811880b9160decfb41006af6c39056c9b0c139f7acf647fb5
b0b22486039504035820c319066c466ef36d08809279c02dac8430c119fae886
7f0c235cd6f6e4514c0f0458206af5dcb3e7138eb2a0f5b523054e05137b558f
1bb6711b10b689e3565c241506
```

issuance\_response\_cbor:

```
a60158201eeda51d75404be1bdd06c31aa72bbd38470a4717e732ccf372b91bc
77161a0e0258204778cf09b14bf78e89e5ef5bcb523863d4e70f9d84aelfbe75
778e60a92c290e035820e94bd324b71702e9f29a239b3a064caa4c46d85d9693
0f75bc39eca6211d2c0f04582029002c4fa8a9f71b8c015fb7869ad64a0fc4e0
50c7c4955ef6ebff27f722890e0558206400000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
```

credit\_token\_cbor:

```
a60158201eeda51d75404be1bdd06c31aa72bbd38470a4717e732ccf372b91bc
77161a0e0258204778cf09b14bf78e89e5ef5bcb523863d4e70f9d84aelfbe75
778e60a92c290e03582069e5d557cb6094acfa586118e602e90aa6fe6cbabd45
71eeb0d2f63b8c8a8f070458206102398efee33b886f4bb7042b897d83db59b7
1a05aff76e9b633b87cade7d0005582064000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000
```

## A.4. Spending



```
5184c3bd8bc8d7a3f33941a3e8126850fbc795d4809dfef5f363a30ff2ee0482
58204c7c14d9881aa291a136ab4979d832cf5c80b75c8fe00ca976f65b3d4c9d
150c5820150b6c20f4b894b4d491a252b4956c72e880639b7880f25f6ad0cba1
6dbd020f8258207b2bc0d260f1f3ac32dfa6643a721a845faf9fe8e5de52751f
5d824fa34e770f58209fe39299e57caa120937957a8957cd6c25f14958ea8e7b
6333a2ac501bc144068258201f19d99a4e9955cc464fe48912f51399358bc678
4a22312a95233e592dd1070558204c666f3adfaa3806b290247faa6950bb74e2
70c9155f5004d407d21f223c0802825820d0b1d956f1b1a4ffb3c1a447bcb22a
0e5679b70ca98357ce0dd5b6670bdc480e582017636a9a14caef700e53785bcb
bd98380ac45d53af06107e99911fbf4ccf24001058201c7ad54f635ebd976c1e
4fc275d93ce2e0981bbcaacb3745deda0be61cfe9f07115820c8451b222fdf76
abc8a76dd90d1b5526de7e510809750825eea0cce6a4fe1c0212582000000000
0000000000000000000000000000000000000000000000000000000000000000
```

prerefund\_cbor:

```
a40158200f9288d8ef1360d8ef4967e041bf09a716c093956464370d30dfe283
2be71b06025820ebada4fb4050db92729a58f0ae585f76154103a2ef2166c401
12638f006d280b03582046000000000000000000000000000000000000000000
0000000000000000000000004582000000000000000000000000000000000000
00000000000000000000000000000000
```

#### A.5. Refund

refund\_cbor:

```
a5015820880974b47fd0d4d06333e2f047abc4420992bd903ed44dae86199a54
361f9c540258208a0977b088e9d17a637f71a013c67774648f0da03b141404ae
678a0e5e090b04035820fdcd645c0d6e13905fff07e56d63465e4cc585f3c247
8500c96cd361a4ad01070458202c9f3110e53540738100e7e636949ce7ac08bf
b4ac6867fb72ac6ec847a2f90e0558200a000000000000000000000000000000
00000000000000000000000000000000
```

#### A.6. Refund Token

```
refund_token_cbor:
  a6015820880974b47fd0d4d06333e2f047abc4420992bd903ed44dae86199a54
  361f9c540258208a0977b088e9d17a637f71a013c67774648f0da03b141404ae
  678a0e5e090b04035820ebada4fb4050db92729a58f0ae585f76154103a2ef21
  66c40112638f006d280b0458200f9288d8ef1360d8ef4967e041bf09a716c093
  956464370d30dfe2832be71b06055820500000000000000000000000000000
  000000000000000000000000000000000000000000000000000000000000
  000000000000000000000000000000000000000000000000000000000000

refund_token_credits:
  500000000000000000000000000000000000000000000000000000000000

refund_token_nullifier:
  ebada4fb4050db92729a58f0ae585f76154103a2ef2166c40112638f006d280b

remaining_balance: 80
```

## 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 ( $A, e, k, r, c, \text{ctx}$ ).

**\*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