Workgroup: Network Working Group
Internet-Draft: draft-dss-star-02
Published: 24 October 2022
Intended Status: Standards Track
Expires: 27 April 2023
Authors: A. Davidson
Brave Software
S. K. Sahib
Brave Software
P. Snyder
Brave Software
C. A. Wood
Cloudflare

STAR: Distributed Secret Sharing for Private Threshold Aggregation Reporting

Abstract

Servers often need to collect data from clients that can be privacy-sensitive if the server is able to associate the collected data with a particular user. In this document we describe STAR, an efficient and secure threshold aggregation protocol for collecting measurements from clients by an untrusted aggregation server, while maintaining K-anonymity guarantees.

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 27 April 2023.

Copyright Notice

Table of Contents

1. Introduction

Collecting user data is often fraught with privacy issues because without adequate protections it is trivial for the server to learn sensitive information about the client contributing data. Even when the client's identity is separated from the data (for example, if the client is using the [Tor] network or [OHTTP] to upload data), it's possible for the collected data to be unique enough that the user's identity is leaked. A common solution to this problem of the measurement being user-identifying is to make sure that the measurement is only revealed to the server if there are at least K clients that have contributed the same data, thus providing K-anonymity to participating clients. Such privacy-preserving systems are referred to as threshold aggregation systems.

In this document we describe one such system, namely Distributed Secret Sharing for Private Threshold Aggregation Reporting (STAR) [STAR].

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.

The following notation is used throughout the document.

- *len(l): Outputs the length of input list l, e.g., len([1,2,3]) = 3).
- *range(a, b): Outputs a list of integers from a to b-1 in ascending order, e.g., range(1, 4) = [1,2,3].
- *pow(a, b): Outputs the integer result of a to the power of b, e.g., pow(2, 3) = 8.
- *|| denotes concatenation of byte strings, i.e., x || y denotes the byte string x, immediately followed by the byte string y, with no extra separator, yielding xy.
- *str(x): Outputs an ASCII string encoding of the integer input x, e.g., str(1) = "1".

*nil denotes an empty byte string.

In addition, the following terms are used:

Aggregation Server: An entity that would like to learn aggregated data from users.

Randomness Server: An entity that runs an oblivious pseudorandom function ([OPRF]) service that allows clients to receive pseudorandom function evaluations on their measurement and the server OPRF key, without the Randomness Server learning anything about their measurement. The clients use the output as randomness to produce the report that is then sent to the Aggregation Server.

Anonymizing Server: An entity that clients use to decouple their identity (IP address) from their messages sent to the Aggregation Server.

Client: The entity that provides user data to the system.

Measurement: The unencrypted, potentially-sensitive data that the client is asked to report.

Report: The encrypted measurement being sent by the client.

Auxiliary Data: Arbitrary data that clients may send as part of their report, but which is only revealed when at least K encrypted measurements of the same value are received.

REPORT_THRESHOLD: The minimum number of reports that an Aggregation Server needs before revealing client data. This value is chosen by the application.

3. Cryptographic Dependencies

STAR depends on the following cryptographic protocols and primitives:

*Threshold secret sharing (TSS); Section 3.1

*Oblivious Pseudorandom Function (OPRF); Section 3.2

*Key Derivation Function (KDF); Section 3.3

*Key-Committing Authenticated Encryption with Associated Data (KCAEAD); Section 3.4

This section describes the syntax for these protocols and primitives in more detail.

3.1. Threshold Secret Sharing

A threshold secret sharing scheme with the following important properties:

- *Privacy: Secret shares reveal nothing unless k = REPORT_THRESHOLD shares are combined to recover the secret.

- *Authenticity: Combining at least k = REPORT_THRESHOLD shares will only succeed if all shares correspond to the same underlying secret. Otherwise, it fails.

A threshold secret sharing scheme with these properties has the following API syntax:

- *Share(k, secret, rand): Produce a k-threshold share using randomness rand and secret, along with a commitment to the secret, each of size Nshare and Ncommitment bytes long. The value k is an integer, and secret and rand are byte strings.

- *Recover(k, share_set): Combine the secret shares in share_set, each of which correspond to the same secret share commitment, which is of size at least k, and recover the corresponding message secret. If recovery fails, this function returns an error.

- *Nshare: The size in bytes of a secret share value.

- *Ncommitment: The size in bytes of a secret share commitment value.

A threshold secret sharing scheme is built on top of the scalar field of a prime-order group G, where the order is a large prime p. The group operation for G is addition + with identity element I. For any elements A and B of the group G, A + B = B + A is also a member of G. Also, for any A in G, there exists an element -A such that A + (-A) = (-A) + A = I. Integers, taken modulo the group order p, are called scalars; arithmetic operations on scalars are implicitly performed modulo p. Since p is prime, scalars form a finite field. Scalar multiplication is equivalent to the repeated application of the group operation on an element A with itself r-1 times, denoted as ScalarMult(A, r). We denote the sum, difference, and product of two scalars using the +, -, and * operators, respectively. (Note that this means + may refer to group element addition or scalar addition, depending on types of the operands.) For any element A, ScalarMult(A, p) = I. We denote B as a fixed generator of the group. Scalar base multiplication is equivalent to the repeated application of the group operation B with itself r-1 times, this is denoted as ScalarBaseMult(r). The set of scalars corresponds to GF(p), which we refer to as the scalar field. This document uses types Element and

Scalar to denote elements of the group G and its set of scalars, respectively. We denote Scalar(x) as the conversion of integer input x to the corresponding Scalar value with the same numeric value. For example, Scalar(1) yields a Scalar representing the value 1. We denote equality comparison as == and assignment of values by =. Finally, it is assumed that group element addition, negation, and equality comparisons can be efficiently computed for arbitrary group elements.

We now detail a number of member functions that can be invoked on G.

- Identity(): Outputs the group identity element I.

- RandomScalar(): Outputs a random Scalar element in GF(p), i.e., a random scalar in [0, p - 1].

- HashToScalar(x, dst): Deterministically map an array of bytes x to a Scalar element. This function is optionally parameterized by a domain separation tag dst.

- SerializeElement(A): Maps an Element A to a canonical byte array buf of fixed length Ne. This function can raise an error if A is the identity element of the group.

- DeserializeElement(buf): Attempts to map a byte array buf to an Element A, and fails if the input is not the valid canonical byte representation of an element of the group. This function can raise an error if deserialization fails or A is the identity element of the group.

- ScalarBaseMult(k): Output the scalar multiplication between Scalar k and the group generator B.

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

- DeserializeScalar(buf): Attempts to map a byte array buf to a Scalar s. This function can raise an error if deserialization fails.

[[OPEN ISSUE: specify validation steps somewhere, likely cribbing from other documents]]

3.1.1. Unverifiable Secret Sharing

This section specifies traditional (unverifiable) Shamir secret sharing (SSS) [Shamir] for implementing the sharing scheme. This functionality is implemented using ristretto255 [RISTRETTO]. Share and Recover are implemented as follows, where Nshare = 2*Nscalar and Ncommitment = 32.

```
def Share(k, secret, rand):
  # Construct the secret sharing polynomial
  poly = [G.HashToScalar(secret, str(0))]
  for i in range(1, k):
    poly.extend(G.HashToScalar(rand, str(i)))

  # Compute the secret commitment
  commitment = SHA256(secret)

  # Evaluate the polynomial at a random point
  x = G.RandomScalar()
  y = polynomial_evaluate(x, poly)

  # Construct the share
  x_enc = G.SerializeScalar(x)
  y_enc = G.SerializeScalar(y)
  share = x_enc || y_enc

  return share, commitment

def Recover(k, share_set):
  if share_set.length < k:
    raise RecoveryFailedError

  points = []
  for share in share_set:
    x = G.DeserializeScalar(share[0:Ns])
    y = G.DeserializeScalar(share[Ns:])
    points.append((x, y))

  poly = polynomial_interpolation(points)
  return poly[0]
```

The dependencies for Share and Recover are as follows:

- *polynomial_evaluate(x, poly) from [FROST], Section 4.2.1 for evaluating a given polynomial specified by poly on the input x.

- *polynomial_interpolation(points) from [FROST], Section 4.2.3 for constructing a polynomial of degree N-1 from the set points of size N and returning the coefficient list, where the 0-th coefficient of the polynomial is the first element in the output list.

3.1.2. Verifiable Secret Sharing

This section specifies Feldman's verifiable secret sharing (VSS) [Feldman] for implementing the sharing scheme. This functionality is implemented using ristretto255 [RISTRETTO]. Share and Recover are

implemented as follows, where Nshare = 2*Nscalar and Ncommitment = k*Ne, where Ne is the size of a serialized group element.

```
def Share(k, secret, rand):
  # Construct the secret sharing polynomial
  poly = [G.HashToScalar(secret, str(0))]
  for i in range(1, k):
    poly.extend(G.HashToScalar(rand, str(i)))

  # Compute the secret (and polynomial) commitment
  commitment = Commit(secret)

  # Evaluate the polynomial at a random point
  x = G.RandomScalar()
  y = polynomial_evaluate(x, poly)

  # Construct the share
  x_enc = G.SerializeScalar(x)
  y_enc = G.SerializeScalar(y)
  share = x_enc || y_enc

  return share, commitment

def Recover(k, share_set):
  if share_set.length < k:
    raise RecoveryFailedError

  points = []
  for share in share_set:
    x = G.DeserializeScalar(share[0:Ns])
    y = G.DeserializeScalar(share[Ns:])
    points.append((x, y))

  poly = polynomial_interpolation(points)
  return poly[0]
```

The helper functions polynomial_evaluate and polynomial_interpolation are as defined in the previous section. The helper function Commit is implemented as follows:

```
def Commit(poly):
  commitment = nil
  for coefficient in poly:
    C_i = G.ScalarBaseMult(coefficient)
    commitment = commitment || G.SerializeElement(C_i)
  return commitment
```

Moreover, VSS extends the syntax of SSS to add another function, Verify, that is used to check that a share is correct for a given commitment. Verify is implemented as follows.

```
def Verify(share, commitment):
  x = G.DeserializeScalar(share[0:Ns])
  y = G.DeserializeScalar(share[Ns:])
  S' = G.ScalarBaseMult(y)

  if len(commitment) % Ne != 0:
    raise Exception("Invalid commitment length")
  num_coefficients = len(commitment) % Ne
  commitments = []
  for i in range(0, num_coefficients):
    c_i = G.DeserializeElement(commitment[i*Ne:(i+1)*Ne])
    commitments.extend(c_i)

  S = G.Identity()
  for j in range(0, num_coefficients):
    S = S + G.ScalarMult(commitments[j], pow(x, j))
  return S == S'
```

3.2. Verifiable Oblivious Pseudorandom Function

A Verifiable Oblivious Pseudorandom Function (VOPRF) is a two-party protocol between client and server for computing a PRF such that the client learns the PRF output and neither party learns the input of the other. This specification depends on the prime-order VOPRF construction specified in [OPRF], draft version -10, using the VOPRF mode (0x01) from [OPRF], Section 3.1.

The following VOPRF client functions are used:

- *Blind(element): Create and output (blind, blinded_element), consisting of a blinded representation of input element, denoted blinded_element, along with a value to revert the blinding process, denoted blind.

- *Finalize(element, blind, evaluated_element, proof): Finalize the OPRF evaluation using input element, random inverter blind, evaluation output evaluated_element, and proof proof, yielding output oprf_output or an error upon failure.

Moreover, the following OPRF server functions are used:

- *BlindEvaluate(k, blinded_element): Evaluate blinded input element blinded_element using input key k, yielding output element evaluated_element and proof proof. This is equivalent to the Evaluate function described in [OPRF], Section 3.3.1, where k is the private key parameter.

- *DeriveKeyPair(seed, info): Derive a private and public key pair deterministically from a seed and info parameter, as described in [OPRF], Section 3.2.

Finally, this specification makes use of the following shared functions and parameters:

- SerializeElement(element): Map input element to a fixed-length byte array buf.

- DeserializeElement(buf): Attempt to map input byte array buf to an OPRF group element. This function can raise a DeserializeError upon failure; see [OPRF], Section 2.1 for more details.

- SerializeScalar(scalar): Map input scalar to a unique byte array buf of fixed length Ns bytes.

- DeserializeScalar(buf): Attempt to map input byte array buf to an OPRF scalar element. This function raise a DeserializeError upon failure; see [OPRF], Section 2.1 for more details.

- Ns: The size of a serialized OPRF scalar element output from SerializeScalar.

- Noe: The size of a serialized OPRF group element output from SerializeElement.

This specification uses the verifiable OPRF from [OPRF], Section 3 with the OPRF(ristretto255, SHA-512) as defined in [OPRF], Section 4.1.1.

3.3. Key Derivation Function

A Key Derivation Function (KDF) is a function that takes some source of initial keying material and uses it to derive one or more cryptographically strong keys. This specification uses a KDF with the following API and parameters:

- Extract(salt, ikm): Extract a pseudorandom key of fixed length Nx bytes from input keying material ikm and an optional byte string salt.

- Expand(prk, info, L): Expand a pseudorandom key prk using the optional string info into L bytes of output keying material.

- Nx: The output size of the Extract() function in bytes.

This specification uses HKDF-SHA256 [HKDF] as the KDF function, where Nx = 32.

3.4. Key-Committing Authenticated Encryption with Associated Data

A Key-Committing Authenticated Encryption with Associated Data (KCAEAD) scheme is an algorithm for encrypting and authenticating

plaintext with some additional data. It has the following API and parameters:

- *Seal(key, nonce, aad, pt): Encrypt and authenticate plaintext "pt" with associated data "aad" using symmetric key "key" and nonce "nonce", yielding ciphertext "ct" and tag "tag".

- *Open(key, nonce, aad, ct): Decrypt "ct" and tag "tag" using associated data "aad" with symmetric key "key" and nonce "nonce", returning plaintext message "pt". This function can raise an OpenError upon failure.

- *Nk: The length in bytes of a key for this algorithm.

- *Nn: The length in bytes of a nonce for this algorithm.

- *Nt: The length in bytes of the authentication tag for this algorithm.

This specification uses a KCAEAD built on AES-128-GCM [GCM], HKDF-SHA256 [HKDF], and HMAC-SHA256 [HMAC]. In particular, Nk = 16, Nn = 12, and Nt = 16. The Seal and Open functions are implemented as follows.

```
def Seal(key, nonce, aad, pt):
  key_prk = Extract(nil, key)
  aead_key = Expand(key_prk, "aead", Nk)
  hmac_key = Expand(key_prk, "hmac", 32) // 32 bytes for SHA-256

  ct = AES-128-GCM-Seal(key=aead_key, nonce=nonce, aad=aad, pt=pt)
  tag = HMAC(key=hmac_key, message=ct)
  return ct || tag

def Open(key, nonce, aad, ct_and_tag):
  key_prk = Extract(nil, key)
  aead_key = Expand(key_prk, "aead", Nk)
  hmac_key = Expand(key_prk, "hmac", 32) // 32 bytes for SHA-256

  ct || tag = ct_and_tag
  expected_tag = HMAC(key=hmac_key, message=ct)
  if !constant_time_equal(expected_tag, tag):
    raise OpenError
  pt = AES-128-GCM-Open(key=aead_key, nonce=nonce, aad=aad, ct=ct) // Th
  return pt
```

4. System Overview

In STAR, clients generate encrypted measurements and send them to a single untrusted Aggregation Server in a report. Each report is effectively a random k-out-of-n share of the client data secret,

along with some additional auxilary data. In a given amount of time, if the Aggregation Server receives the same encrypted value from k = REPORT_THRESHOLD clients, the server can recover the client data associated with each report. This ensures that clients only have their measurements revealed if they are part of a larger crowd, thereby achieving k-anonymity privacy (where k = REPORT_THRESHOLD).

Each client report is as secret as the underlying client data. That means low entropy client data values could be abused by an untrusted Aggregation Server in a dictionary attack to recover client data with fewer than REPORT_THRESHOLD honestly generated reports. To mitigate this, clients boost the entropy of their data using output from an Oblivious Pseudorandom Function (OPRF) provided by a separate, non-colluding Randomness Server.

STAR also requires use of a client Anonymizing Proxy when interacting with the Aggregation Server so that the Aggregation Server cannot link a client report to a client which generated it. This document does not require a specific type of proxy. In practice, proxies built on [OHTTP] or [Tor] suffice; see Section 5 for more details.

The overall architecture is shown in Figure 1, where msg is the measurement and aux is auxiliary data associated with a given client. The output of the interaction is a data value msg shared amongst REPORT_THRESHOLD honest clients and a list of additional auxiliary data values associated with each of the REPORT_THRESHOLD client reports, denoted <aux>.

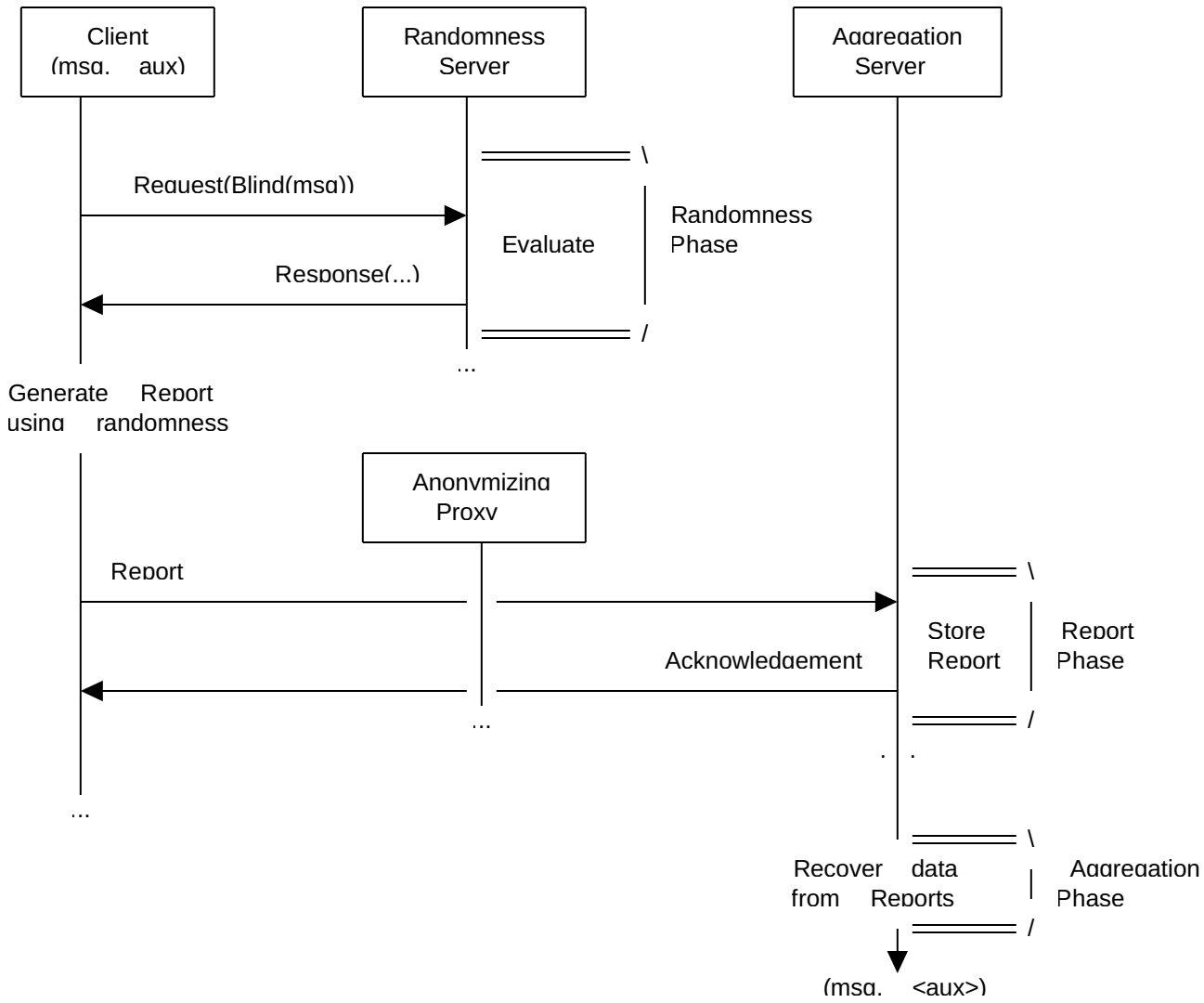

Figure 1: System Architecture

In the following subsections, we describe each of the phases of STAR in more detail.

4.1. Randomness Phase

The randomness sampled from a client data **MUST** be a deterministic function of the measurement. Clients sample this randomness by running an OPRF protocol with the Randomness Server. This section describes how the Randomness Server is configured and then how clients interact with it for computing the randomness.

4.1.1. Configuration

STAR clients are configured with a Randomness Server URI and the Randomness Server public key pkR. Clients use this URI to send HTTP messages to the Randomness Server to complete the protocol. As an example, the Randomness Server URI might be https://randomness.example.

The Randomness Server only needs to configure an OPRF key pair per epoch. This is done as follows:

```
seed = random(32)
(skR, pkR) = DeriveKeyPair(seed, "STAR")
```

4.1.2. Randomness Protocol

This procedure works as follows. Let msg be the client's measurement to be used for deriving the randomness rand.

Clients first generate the a context for invoking the OPRF protocol as follows:

```
client_context = SetupVOPRFClient(0x0001, pkR) // OPRF(ristretto255, SHA
```

Clients then blind their measurement using this context as follows:

```
(blinded, blinded_element) = client_context.Blind(msg)
```

Clients then compute randomness_request = OPRF.SerializeElement(blinded_element) and send it to the Randomness Server URI in a HTTP POST message using content type "application/star-randomness-request". An example request is shown below.

```
:method = POST
:scheme = https
:authority = randomness.example
:path = /
accept = application/star-randomness-response
content-type = application/star-randomness-request
content-length = Noe

<Bytes containing a serialized blinded element>
```

Upon receipt, the Randomness Server evaluates and returns a response. It does so by first creating a context for running the ORPF protocol as follows:

```
server_context = SetupVOPRFServer(0x0001, skR, pkR) // OPRF(ristretto255
```

Here, skR and pkR are private and public keys generated as described in Section 4.1.1.

The Randomness Server then computes blinded_element = OPRF.DeserializeElement(randomness_request). If this fails, the Randomness Server returns an error in a 4xx response to the client. Otherwise, the server computes:

```
evaluated_element, proof = server_context.BlindEvaluate(sk, blinded_elem
```

The Randomness Server then serializes the evaluation output and proof to produce a randomness response as follows:

```
evaluated_element_enc = OPRF.SerializeElement(evaluated_element)
proof_enc = OPRF.SerializeScalar(proof[0]) || OPRF.SerializeScalar(proof
randomness_response = evaluated_element_enc || proof_enc
```

This response is then sent to the client using the content type "application/star-randomness-response". An example response is below.

```
:status = 200
content-type = application/star-randomness-response
content-length = Noe

<Bytes containing randomness_response>
```

Upon receipt, the client computes parses randomness_response to recover the evaluated element and proof as follows:

```
evaluated_element_enc || proof_enc = parse(randomness_response)
evaluated_element = OPRF.DeserializeElement(evaluated_element_enc)
proof = [OPRF.DeserializeScalar(proof_enc[0:Ns]), OPRF.DeserializeScalar
```

If any of these steps fail, the client aborts the protocol. Otherwise, the client finalizes the OPRF protocol to compute the output rand as follows:

```
rand = client_context.Finalize(msg, blind, evaluated_element, proof)
```

4.2. Reporting Phase

In the reporting phase, the client uses its measurement msg with auxiliary data aux and its derived randomness rand to produce a report for the Aggregation Server.

4.2.1. Reporting Configuration

The reporting phase requires the Aggregation Server to be configured with a URI for accepting reports. As an example, the Aggregation

Server URI might be https://aggregator.example. The Aggregation Server is both an Oblivious HTTP Target and Oblivious Gateway Resource.

Clients are also configured with an Anonymizing Proxy that clients can use to send proxy reports to the Aggregation Server. The exact type of proxy is not specified here. See Section 5 for more details.

4.2.2. Reporting Protocol

This reporting protocol works as follows. First, the client stretches rand into three values key_seed and share_coins, and additionally derives an KCAEAD key and nonce from key_seed.

```
// Randomness derivation
rand_prk = Extract(nil, rand)
key_seed = Expand(rand_prk, "key_seed", 16)
share_coins = Expand(rand_prk, "share_coins", 16)

// Symmetric encryption key derivation
key_prk = Extract(nil, key_seed)
key = Expand(key_prk, "key", Nk)
nonce = Expand(key_prk, "nonce", Nn)
```

The client then generates a secret share of key_seed using share_coins as randomness as follows:

```
random_share, share_commitment = Share(REPORT_THRESHOLD, key_seed, share
```

The client then encrypts msg and aux using the KCAEAD key and nonce as follows:

```
report_data = len(msg, 4) || msg || len(aux, 4) || aux
encrypted_report = Seal(key, nonce, nil, report_data)
```

The function len(x, n) encodes the length of input x as an n-byte big-endian integer.

Finally, the client constructs a report consisting of encrypted_report and random_share, as well as share_commitment, and sends this to the Anonymizing Server in the subsequent epoch, i.e., after the Randomness Server has rotated its OPRF key.

```
struct {
  opaque encrypted_report<1..2^16-1>;
  opaque random_share[Nshare];
  opaque share_commitment[Ncommitment];
} Report;
```

Specifically, Clients send a Report to the Aggregation Server using an HTTP POST message with content type "application/star-report". An example message is below.

```
:method = POST
:scheme = https
:authority = aggregator.example
:path = /
content-type = application/star-report
content-length = <Length of body>

<Bytes containing a Report>
```

This message is sent to the Aggregation Server through the Anonymizing Proxy. See Section 5 for different types of proxy options.

4.3. Aggregation Phase

Aggregation is the final phase of STAR. It happens offline and does not require any communication between different STAR entities. It proceeds as follows. First, the Aggregation Server groups reports together based on their share_commitment value. If applicable, the Aggregation Server also verifies that each share commitment is correct, i.e., by invoking the Verify function on each share and share_commitment pair in candidate set of reports. Let report_set denote a set of at least REPORT_THRESHOLD reports that have a matching share_commitment value.

Given this set, the Aggregation Server begins by running the secret share recovery algoritm as follows:

```
key_seed = Recover(report_set)
```

If this fails, the Aggregation Server chooses a new candidate report share set and reruns the aggregation process. See Section 6.3 for more details.

Otherwise, the Aggregation Server derives the same KCAEAD key and nonce from key_seed to decrypt each of the report ciphertexts in report_set.

```
key_prk = Extract(nil, key_seed)
key = Expand(key_prk, "key", Nk)
nonce = Expand(key_prk, "nonce", Nn)
```

Each report ciphertext is decrypted as follows:

```
report_data = Open(key, nonce, nil, ct)
```

The message msg and auxiliary data aux are then parsed from report_data.

If this fails for any report, the Aggregation Server chooses a new candidate report share set and reruns the aggregation process. Otherwise, the Aggregation Server then outputs the msg and aux values for the corresponding reports.

4.4. Auxiliary data

In Figure 1, aux refers to auxiliary or additional data that may be sent by clients, and is distinct from the measurement data protected by the K-anonymity guarantee. Auxiliary data is only revealed when the k-condition is met but, importantly, is not part of the k-condition itself. This data might be unique to some or all of the submissions, or omitted entirely. This can even be the actual measured value itself. For example: if we're measuring tabs open on a client, then the measurement being sent can be "city: Vancouver" and the aux data can be "7" for a particular client. The idea being, that we only reveal all the measurements once we know that there are at least K clients with city: Vancouver.

5. Anonymizing Proxy Options

The Anonymizing Proxy can be instantiated using [OHTTP], [Tor], or even a TCP-layer proxy. The choice of which proxy to use depends on the application threat model. The fundamental requirement is that the Anonymizing Proxy hide the client IP address and any other unique client information from the Aggregation Server.

In general, there are two ways clients could implement the proxy: at the application layer, e.g., via [OHTTP], or at the connection or transport layer, e.g., via [Tor] or similar systems. We describe each below.

5.1. Application-Layer Proxy

An application-layer proxy hides client identifying information from the Aggregation Server via application-layer intermediation. [OHTTP] is the **RECOMMENDED** option for an application-layer proxy. [OHTTP] ensures that a network adversary between the client and Anonymizing Proxy cannot link reports sent to the Aggregation Server (up to what is possible by traffic analysis).

OHTTP consists of four entities: client, Oblivious Relay Resource, Oblivious Gateway Resource, and Target Resource. In this context, the Target Resource is the Aggregation Server. The Aggregation Server can also act as the Oblvious Gateway Resource. Clients are configured with the URI of the Oblivious Relay Resource, and use this to forward requests to a Oblivious Gateway Resource. The

Oblivious Gateway Resource then forwards requests to the Target as required.

5.2. Connection-Layer Proxy

A connection-layer proxy hides client identifying information from the Aggregation Server via connection-layer intermediation. [Tor] is perhaps the most commonly known example of such a proxy. Clients can use Tor to connect to and send reports to the Aggregation Server. Other examples of connection-layer proxies include CONNECT-based HTTPS proxies, used in systems like Private Relay [PrivateRelay] and TCP-layer proxies. TCP proxies only offer weak protection in practice since an adversary capable of eavesdropping on ingress and egress connections from the Anonymizing Proxy can trivially link data together.

6. Security Considerations

This section contains security considerations for the draft.

6.1. Randomness Sampling

Deterministic randomness **MUST** be sampled by clients to construct their STAR report, as discussed in Section 4.2. This randomness CANNOT be derived locally, and **MUST** be sampled from the Randomness Server (that runs an [OPRF] service).

For best-possible security, the Randomness Server **SHOULD** sample and use a new OPRF key for each time epoch t, where the length of epochs is determined by the application. The previous OPRF key that was used in epoch t-1 can be safely deleted. As discussed in Section 6.5, shorter epochs provide more protection from Aggregation Server attacks, but also reduce the window in which data collection occurs (and hence reduce the possibility that we will have enough reports to decrypt) while increasing the reporting latency.

In this model, for further security, clients **SHOULD** sample their randomness in epoch t and then send it to the Aggregation Server in t+1 (after the Randomness Server has rotated their secret key). This prevents the Aggregation Server from launching queries after receiving the client reports (Section 6.5). It is also **RECOMMENDED** that the Randomness Server runs in verifiable mode, which allows clients to verify the randomness that they are being served [OPRF].

6.2. Oblivious Submission

The reports being submitted to an Aggregation Server in STAR **MUST** be detached from client identity. This is to ensure that the Aggregation Server does not learn exactly what each client submits, in the event that their measurement is revealed. This is achieved

through the use of an Anonymizing Server, which is an OHTTP Oblivious Relay Resource. This server **MUST NOT** collude with the Aggregation Server. All the client responsibilities mentioned in section 7.1 of [OHTTP] apply.

The OHTTP Relay Resource and Randomness Server **MAY** be combined into a single entity, since client reports are protected by a TLS connection between the client and the Aggregation Server. Therefore, OHTTP support can be enabled without requiring any additional non-colluding parties. In this mode, the Randomness Server **SHOULD** allow two endpoints: (1) to evaluate the VOPRF functionality that provides clients with randomness, and (2) to proxy client reports to the Aggregation Server. However, this increases the privacy harm in case of collusion; see Section 6.5.3.

If configured otherwise, clients can upload reports to the Aggregation Server using an existing anonymizing proxy service such as [Tor]. However, use of OHTTP is likely to be the most efficient way to achieve oblivious submission.

6.3. Malicious Clients

Malicious clients can perform a denial-of-service attacks on the system by sending bogus reports to the Aggregation Server. There are several types of bogus reports:

* Reports with invalid shares, or corrupt reports. These are reports that will yield the incorrect secret when combined by the Aggregation Server.

* Reports with invalid ciphertext, or garbage reports. These are reports that contain an encryption of the wrong measurement value (msg).

Corrupt reports can be mitigated by using a verifiable secret sharing scheme, such as the one described in Section 3.1.2, and verifying that the share commitments are correct for each share. This ensures that each share in a report set corresponds to the same secret.

Garbage reports cannot easily be mitigated unless the Aggregation Server has a way to confirm that the recovered secret is correct for a given measurement value (msg). This might be done by allowing the Aggregation Server to query the Randomness Server on values of its choosing, but this opens the door to dictionary attacks.

In the absence of protocol-level mitigations, Aggregation Servers can limit the impact of malicious clients by using higher-layer defences such as identity-based certification [Sybil].

6.4. Malicious Aggregation Server

6.4.1. Dictionary Attacks

The Aggregation Server may attempt to launch a dictionary attack against the client measurement, by repeatedly launching queries against the Randomness Server for measurements of its choice. This is mitigated by the fact that the Randomness Server regularly rotates the VOPRF key that they use, which reduces the window in which this attack can be launched (Section 6.1). Note that such attacks can also be limited in scope by maintaining out-of-band protections against entities that attempt to launch large numbers of queries in short time periods.

6.4.2. Sybil Attacks

By their very nature, attacks where a malicious Aggregation Server injects clients into the system that send reports to try and reveal data from honest clients are an unavoidable consequence of building any threshold aggregation system. This system cannot provide comprehensive protection against such attacks. The time window in which such attacks can occur is restricted by rotating the VOPRF key (Section 6.1). Such attacks can also be limited in scope by using higher-layer defences such as identity-based certification [Sybil].

6.5. Leakage and Failure Model

6.5.1. Size of Anonymity Set

Client reports immediately leak deterministic tags that are derived from the VOPRF output that is evaluated over client measurement. This has the immediate impact that the size of the anonymity set for each received measurement (i.e. which clients share the same measurement) is revealed, even if the measurement is not revealed. As long as client reports are sent via an [OHTTP] Relay Resource, then the leakage derived from the anonymity sets themselves is significantly reduced. However, it may still be possible to use this leakage to reduce a client's privacy, and so care should be taken to not construct situations where counts of measurement subsets are likely to lead to deanonymization of clients or their data.

6.5.2. Collusion between Aggregation and Randomness Servers

Finally, note that if the Aggregation and Randomness Servers collude and jointly learn the VOPRF key, then the attack above essentially becomes an offline dictionary attack. As such, client security is not completely lost when collusion occurs, which represents a safer mode of failure when compared with Prio and Poplar.

6.5.3. Collusion between Aggregation Server and Anonymizing Proxy

As mentioned in Section 6.2, systems that depend on a relaying server to remove linkage between client reports and client identity rely on the assumption of non-collusion between the relay and the server processing the client reports. Given that STAR depends on such a system for guaranteeing that the Aggregation Server does not come to know which client submitted the STAR report (once decrypted), the same collusion risk applies.

It's worth mentioning here for completeness sake that if the OHTTP Relay Resource and Randomness Server are combined into a single entity as mentioned in Section 6.2, then this worsens the potential leakage in case of collusion: if the entities responsible for the Aggregation Server and the Randomness Server collude as described in Section 6.5.2, this results in the Aggregation Server in effect colluding with the anonymizing proxy.

7. Comparisons with other Systems

[[EDITOR NOTE: for information/discussion: consider removing before publication]]

7.1. Private Heavy-Hitter Discovery

STAR is similar in nature to private heavy-hitter discovery protocols, such as Poplar [Poplar]. In such systems, the Aggregation Server reveals the set of client measurements that are shared by at least K clients. STAR allows a single untrusted server to perform the aggregation process, as opposed to Poplar which requires two non-colluding servers that communicate with each other.

As a consequence, the STAR protocol is orders of magnitude more efficient than the Poplar approach, with respect to computational, network-usage, and financial metrics. Therefore, STAR scales much better for large numbers of client submissions. See the [STAR] paper for more details on efficiency comparisons with the Poplar approach.

7.2. General Aggregation

In comparison to general aggregation protocols like Prio [Prio], the STAR protocol provides a more constrained set of functionality. However, STAR is significantly more efficient for the threshold aggregation functionality, requires only a single Aggregation Server, and is not limited to only processing numerical data types.

7.3. Protocol Leakage

As we discuss in Section 6.5, STAR leaks deterministic tags derived from the client measurement that reveal which (and how many) clients

share the same measurements, even if the measurements themselves are not revealed. This also enables an online dictionary attack to be launched by the Aggregation Server by sending repeated VOPRF queries to the Randomness Server as discussed in Section 6.4.1.

The leakage of Prio is defined as whatever is leaked by the function that the aggregation computes. The leakage in Poplar allows the two Aggregation Servers to learn all heavy-hitting prefixes of the eventual heavy-hitting strings that are output. Note that in Poplar it is also possible to launch dictionary attacks of a similar nature to STAR by launching a Sybil attack [Sybil] that explicitly injects multiple measurements that share the same prefix into the aggregation. This attack would result in the aggregation process learning more about client inputs that share those prefixes.

Finally, note that under collusion, the STAR security model requires the adversary to launch an offline dictionary attack against client measurements. In Prio and Poplar, security is immediately lost when collusion occurs.

7.4. Support for auxiliary data

It should be noted that clients can send auxiliary data (Section 4.4) that is revealed only when the aggregation including their measurement succeeds (i.e. K-1 other clients send the same value). Such data is supported by neither Prio, nor Poplar.

8. IANA Considerations

8.1. Protocol Message Media Types

This specification defines the following protocol messages, along with their corresponding media types types:

- Randomness request Section 4.1: "application/star-randomness-request"
- Randomness response Section 4.1: "application/star-randomness-response"
- Report Section 4.2: "application/star-report"

The definition for each media type is in the following subsections.

Protocol message format evolution is supported through the definition of new formats that are identified by new media types.

IANA [shall update / has updated] the "Media Types" registry at https://www.iana.org/assignments/media-types with the registration information in this section for all media types listed above.

[OPEN ISSUE: Solicit review of these allocations from domain experts.]

8.1.1. "application/star-randomness-request" media type

Type name: application

Subtype name: star-randomness-request

Required parameters: N/A

Optional parameters: None

Encoding considerations: only "8bit" or "binary" is permitted

Security considerations: see Section 6

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information:

- **Magic number(s):** N/A
- **Deprecated alias names for this type:** N/A
- **File extension(s):** N/A
- **Macintosh file type code(s):** N/A

Person and email address to contact for further information: see Authors' Addresses section

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

8.1.2. "application/star-report" media type

Type name: application

Subtype name: star-report

Required parameters:
N/A

Optional parameters: None

Encoding considerations: only "8bit" or "binary" is permitted

Security considerations: see Section 6

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information:

Magic number(s): N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information: see Authors' Addresses section

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

8.1.3. "application/star-randomness-response" media type

Type name: application

Subtype name: star-randomness-response

Required parameters: N/A

Optional parameters: None

Encoding considerations: only "8bit" or "binary" is permitted

Security considerations: see Section 6

Interoperability considerations:
N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information:

Magic number(s): N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information: see Authors' Addresses section

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG

9. References

9.1. Normative References

[FROST] Connolly, D., Komlo, C., Goldberg, I., and C. A. Wood, "Two-Round Threshold Schnorr Signatures with FROST", Work in Progress, Internet-Draft, draft-irtf-cfrg-frost-11, 7 October 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-frost-11>.

[GCM] Dworkin, M., "Recommendation for block cipher modes of operation :: GaloisCounter Mode (GCM) and GMAC", National Institute of Standards and Technology report, DOI 10.6028/nist.sp.800-38d, 2007, <https://doi.org/10.6028/nist.sp.800-38d>.

[HKDF] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <https://www.rfc-editor.org/rfc/rfc5869>.

[HMAC] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <https://www.rfc-editor.org/rfc/rfc2104>.

[OPRF] Davidson, A., Faz-Hernández, A., Sullivan, N., and C. A. Wood, "Oblivious Pseudorandom Functions (OPRFs) using Prime-Order Groups", Work in Progress, Internet-Draft, draft-irtf-cfrg-voprf-14, 6 October 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-voprf-14>.

[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>.

[RISTRETTO] de Valence, H., Grigg, J., Hamburg, M., Lovecruft, I., Tankersley, G., and F. Valsorda, "The ristretto255 and decaf448 Groups", Work in Progress, Internet-Draft, draft-irtf-cfrg-ristretto255-decaf448-04, 14 October 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-ristretto255-decaf448-04>.

9.2. Informative References

[ADSS] Bellare, M., Dai, W., and P. Rogaway, "Reimagining Secret Sharing: Creating a Safer and More Versatile Primitive by Adding Authenticity, Correcting Errors, and Reducing Randomness Requirements", 27 June 2020, <https://eprint.iacr.org/2020/800>.

[Brave] "Brave Browser", n.d., <https://brave.com>.

[Feldman] Feldman, P., "A practical scheme for non-interactive verifiable secret sharing", 28th Annual Symposium on Foundations of Computer Science (sfcs 1987), DOI 10.1109/sfcs.1987.4, October 1987, <https://doi.org/10.1109/sfcs.1987.4>.

[OHTTP] Thomson, M. and C. A. Wood, "Oblivious HTTP", Work in Progress, Internet-Draft, draft-ietf-ohai-ohttp-05, 26

September 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-ohai-ohttp-05>.

[Poplar] Boneh, D., Boyle, E., Corrigan-Gibbs, H., Gilboa, N., and Y. Ishai, "Lightweight Techniques for Private Heavy Hitters", 4 January 2022, <https://eprint.iacr.org/2021/017>.

[Prio] Geoghegan, T., Patton, C., Rescorla, E., and C. A. Wood, "Privacy Preserving Measurement", Work in Progress, Internet-Draft, draft-gpew-priv-ppm-01, 7 March 2022, <https://datatracker.ietf.org/doc/html/draft-gpew-priv-ppm-01>.

[PrivateRelay] "iCloud Private Relay Overview", 2021, <https://www.apple.com/icloud/docs/iCloud_Private_Relay_Overview_Dec2021.pdf>.

[SGCM] Saarinen, M.-J. O., "SGCM: The Sophie Germain Counter Mode", 4 November 2011, <https://eprint.iacr.org/2011/326>.

[Shamir] Shamir, A., "How to share a secret", 1 November 1979, <https://dl.acm.org/doi/10.1145/359168.359176>.

[STAR] Davidson, A., Snyder, P., Quirk, E., Genereux, J., Haddadi, H., and B. Livshits, "STAR: Distributed Secret Sharing for Private Threshold Aggregation Reporting", 10 April 2022, <https://arxiv.org/abs/2109.10074>.

[Sybil] Douceur, J., "The Sybil Attack", 10 October 2002, <https://link.springer.com/chapter/10.1007/3-540-45748-8_24>.

[Tor] Dingledine, R., Mathewson, N., and P. Syverson, "Tor: The Second-Generation Onion Router", 2004, <https://svn-archive.torproject.org/svn/projects/design-paper/tor-design.pdf>.

Acknowledgments

The authors would like to thank the authors of the original [STAR] paper, which forms the basis for this document.

Authors' Addresses

Alex Davidson
Brave Software

Email: alex.davidson92@gmail.com

Shivan Kaul Sahib
Brave Software

Email: shivankaulsahib@gmail.com

Peter Snyder
Brave Software

Email: pes@brave.com

Christopher A. Wood
Cloudflare

Email: caw@heapingbits.net