Authors: T. Geoghegan   C. Patton   E. Rescorla   C.A. Wood
         ISRG   Cloudflare   Mozilla   Cloudflare
                Privacy Preserving Measurement

Abstract

   There are many situations in which it is desirable to take
   measurements of data which people consider sensitive. In these
   cases, the entity taking the measurement is usually not interested
   in people's individual responses but rather in aggregated data.
   Conventional methods require collecting individual responses and
   then aggregating them, thus representing a threat to user privacy
   and rendering many such measurements difficult and impractical. This
   document describes a multi-party privacy preserving measurement
   (PPM) protocol which can be used to collect aggregate data without
   revealing any individual user's data.

Discussion Venues

   This note is to be removed before publishing as an RFC.

   Discussion of this document takes place on the mailing list (),
   which is archived at .

   Source for this draft and an issue tracker can be found at https://
   github.com/abetterinternet/ppm-specification.

Status of This Memo

Table of Contents

1. Introduction

This document describes a protocol for privacy preserving
measurement. The protocol is executed by a large set of clients and
a small set of servers. The servers' goal is to compute some
aggregate statistic over the clients' inputs without learning the
inputs themselves. This is made possible by distributing the
computation among the servers in such a way that, as long as at
least one of them executes the protocol honestly, no input is ever
seen in the clear by any server.

1.1. DISCLAIMER

This document is a work in progress. We have not yet settled on the
design of the protocol framework or the set of features we intend to
support.

1.2. 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 terms are used:

Aggregation function: The function computed over the users' inputs.

Aggregator: An endpoint that runs the input-validation protocol and accumulates input shares.

Batch: A set of reports that are aggregated into an output.

Batch duration: The time difference between the oldest and newest report in a batch.

Batch interval: A parameter of the collect or output-share request that specifies the time range of the reports in the batch.

Client: The endpoint from which a user sends data to be aggregated, e.g., a web browser.

Collector: The endpoint that receives the output of the aggregation function.

Input: The measurement (or measurements) emitted by a client, before any encryption or secret sharing scheme is applied.

Input share: One of the shares output by feeding an input into a secret sharing scheme. Each share is to be transmitted to one of the participating aggregators.

Input validation protocol: The protocol executed by the client and aggregators in order to validate the client's input without leaking its value to the aggregators.

Measurement: A single value (e.g., a count) being reported by a client. Multiple measurements may be grouped into a single protocol input.

Minimum batch duration: The minimum batch duration permitted for a PPM task, i.e., the minimum time difference between the oldest and newest report in a batch.

Minimum batch size: The minimum number of reports in a batch.

Leader:
A distinguished aggregator that coordinates input validation and data collection.

Output: The output of the aggregation function over a given set of reports.

Output share: The share of an output emitted by an aggregator. Output shares can be reassembled by the leader into the final output.

Proof: A value generated by the client and used by the aggregators to verify the client's input.

Report: Uploaded to the leader from the client. A report contains the secret-shared and encrypted input and proof.

Server: An aggregator.

This document uses the protocol definition language of [RFC8446].

2. Overview

The protocol is executed by a large set of clients and a small set of servers. We call the servers the *aggregators*. Each client's input to the protocol is a set of measurements (e.g., counts of some user behavior). Given the input set of measurements x_1, ..., x_n held by n users, the goal of a *privacy preserving measurement (PPM) protocol* is to compute $y = F(x\_1, ..., x\_n)$ for some aggregation function F while revealing nothing else about the measurements.

This protocol is extensible and allows for the addition of new cryptographic schemes that compute new functions. The current version supports two schemes:

  *Prio [CGB17], which allows for aggregate statistics such as sum, mean, histograms, etc. over a single value.

  *Heavy Hitters [BBCGGI21], which allows for finding the most common strings among a collection of clients (e.g., the URL of their home page) as well as counting the number of clients that hold a given string.

This protocol is designed to work with schemes that use secret sharing. Rather than send its input in the clear, each client splits its measurements into a sequence of *shares* and sends a share to each of the aggregators. This provides two important properties:

  *It's impossible to deduce the measurement without knowing *all* of the shares.

*It allows the aggregators to compute the final output by first
   aggregating up their measurements shares locally, then combining
   the results to obtain the final output.

2.1. System Architecture

{#system-architecture}

The overall system architecture is shown in Figure 1.

```
                        +------------+
                        |            |
     +--------+         |    Helper  |
     |        |         |            |
     | Client +----+    +-----^------+
     |        |    |          |
     +--------+    |          |
                   |          |
     +--------+    |    +-----v------+         +-----------+
     |        |    +----->          |         |           |
     | Client +---------->   Leader  <---------> Collector |
     |        |    +----->          |         |           |
     +--------+    |    +-----^------+         +-----------+
                   |          |
     +--------+    |          |
     |        |    |          |
     | Client +----+    +-----v------+
     |        |         |            |
     +--------+         |    Helper  |
                        |            |
                        +------------+
```
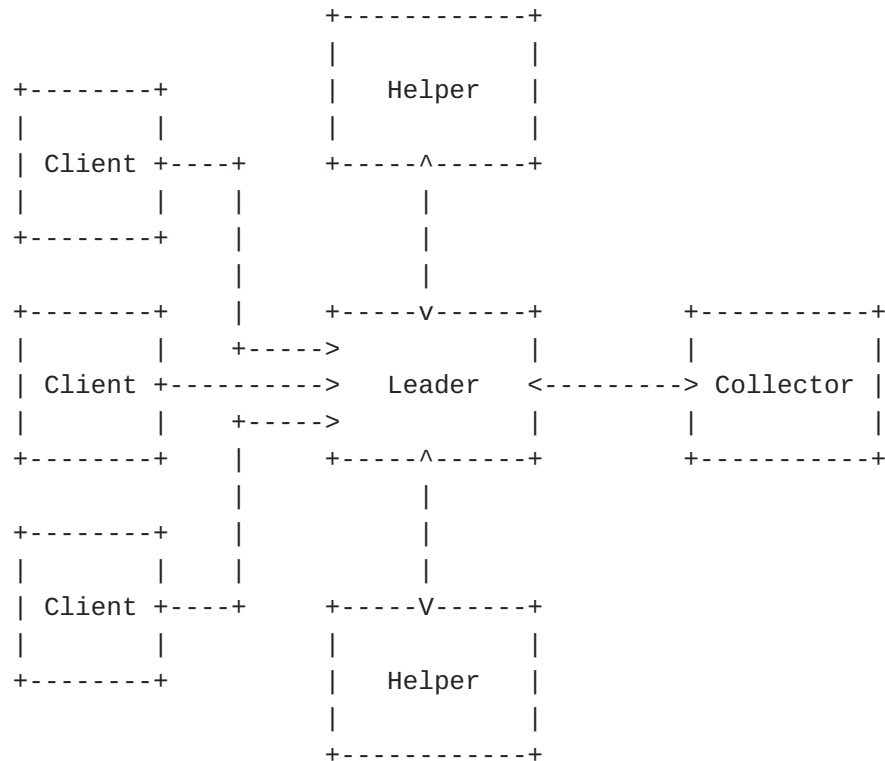
Figure 1: System Architecture

[[OPEN ISSUE: This shows two helpers, but the document only allows
one for now. https://github.com/abetterinternet/ppm-specification/
issues/117]]

The main participants in the protocol are as follows:

Collector:
The entity which wants to take the measurement and ultimately receives the results. Any given measurement will have a single collector.

Client(s): The endpoints which directly take the measurement(s) and report them to the PPM system. In order to provide reasonable levels of privacy, there must be a large number of clients.

Aggregator: An endpoint which receives report shares. Each aggregator works with the other aggregators to compute the final aggregate. This protocol defines two types of aggregators: Leaders and Helpers. For each measurement, there is a single leader and helper.

Leader: The leader is responsible for coordinating the protocol. It receives the encrypted shares, distributes them to the helpers, and orchestrates the process of computing the final measurement as requested by the collector.

Helper: Helpers are responsible for executing the protocol as instructed by the leader. The protocol is designed so that helpers can be relatively lightweight, with most of the state held at the leader.

The basic unit of PPM is the "task" which represents a single measurement (though potentially taken over multiple time wndows). The definition of a task includes the following parameters:

  *The values to be measured;

  *The statistic to be computed (e.g., sum, mean, etc.);

  *The set of aggregators and necessary cryptographic keying material to use; and

  *The PPM scheme to use. This is to some extent dictated by the previous choices.

  *The minimum "batch size" of reports which can be aggregated.

  *The rate at which measurements can be taken, i.e., the "minimum batch window".

These parameters are distributed out of band to the clients and to the aggregators. Each task is identified by a unique 32-byte ID which is used to refer to it in protocol messages.

During the duration of the measurement, each client records its own value(s), packages them up into a report, and sends them to the

leader. Each share is separately encrypted for each aggregator so
that even though they pass through the leader, the leader is unable
to see or modify them. Depending on the measurement, the client may
only send one report or may send many reports over time.

The leader distributes the shares to the helpers and orchestrates
the process of verifying them (see Section 2.2) and assembling them
into a final measurement for the collector. Depending on the PPM
scheme, it may be possible to incrementally process each report as
it comes in, or may be necessary to wait until the entire batch of
reports is received.

2.2. Validating Inputs

An essential task of any data collection pipeline is ensuring that
the input data is "valid". In PPM, input validation is complicated
by the fact that none of the entities other than the client ever
sees the values for individual clients.

In order to address this problem, each PPM client generates a zero-
knowledge proof that its report is valid and attaches it to the
report. The aggregators can then jointly verify this proof prior to
incorporating the report in the aggregation and reject the report if
it cannot be verified. However, they do not learn anything about the
individual report other than that it is valid.

The specific properties attested to in the proof vary depending on
the measurement being taken. For instance, if we want to measure the
time the user took performing a given task the proof might
demonstrate that the value reported was within a certain range
(e.g., 0-60 seconds). By contrast, if we wanted to report which of a
set of N options the user select, the report might contain N
integers and the proof would demonstrate that N-1 were 0 and the
other was 1.

It is important to recognize that "validity" is distinct from
"correctness". For instance, the user might have spent 30s on a task
but the client might report 60s. This is a problem with any
measurement system and PPM does not attempt to address it; it merely
ensures that the data is within acceptable limits, so the client
could not report 10^6s or -20s.

3. Message Transport

Communications between PPM entities are carried over HTTPS
[RFC2818]. HTTPS provides server authentication and confidentiality.
In addition, report shares are encrypted directly to the aggregators
using HPKE [I-D.irtf-cfrg-hpke].

3.1. Errors

Errors can be reported in PPM both at the HTTP layer and within
challenge objects as defined in Section 7. PPM servers can return
responses with an HTTP error response code (4XX or 5XX). For
example, if the client submits a request using a method not allowed
in this document, then the server **MAY** return status code 405 (Method
Not Allowed).

When the server responds with an error status, it **SHOULD** provide
additional information using a problem document [RFC7807]. To
facilitate automatic response to errors, this document defines the
following standard tokens for use in the "type" field (within the
PPM URN namespace "urn:ietf:params:ppm:error:"):

Type	Description
unrecognizedMessage	The message type for a response was incorrect or the payload was malformed.
unrecognizedTask	An endpoint received a message with an unknown task ID.
outdatedConfig	The message was generated using an outdated configuration.

Table 1

This list is not exhaustive. The server **MAY** return errors set to a
URI other than those defined above. Servers **MUST NOT** use the PPM URN
namespace for errors not listed in the appropriate IANA registry
(see Section 7.3). Clients **SHOULD** display the "detail" field of all
errors. The "instance" value **MUST** be the endpoint to which the
request was targeted. The problem document **MUST** also include a
"taskid" member which contains the associated PPM task ID (this
value is always known, see Section 4.1).

In the remainder of this document, we use the tokens in the table
above to refer to error types, rather than the full URNs. For
example, an "error of type 'unrecognizedMessage'" refers to an error
document with "type" value
"urn:ietf:params:ppm:error:unrecognizedMessage".

This document uses the verbs "abort" and "alert with [some error
message]" to describe how protocol participants react to various
error conditions.

4. Protocol Definition

PPM has three major interactions which need to be defined:

   *Uploading reports from the client to the aggregators

*Computing the results of a given measurement

*Reporting results to the collector

We start with some basic type definitions used in other messages.

```
enum { prio(0), hits(1) } Proto;

 /* ASCII encoded URL. e.g., "https://example.com" */
opaque Url<1..2^16-1>;

Duration uint64; /* Number of seconds elapsed between two instants */

Time uint64; /* seconds elapsed since start of UNIX epoch */

/* An interval of time, where start is included and end is excluded */
struct {
  Time start;
  Time end;
} Interval;
```

4.1. Task Configuration

Prior to the start of execution of the protocol, each participant must agree on the configuration for each task. A task is uniquely identified by its task ID:

```
opaque TaskId[32];
```

A TaskId is a globally unique sequence of bytes. It is **RECOMMENDED** that this be set to a random string output by a cryptographically secure pseudorandom number generator. Each task has the following parameters associated with it:

*aggregator_endpoints: A list of URLs relative to which an aggregator's API endpoints can be found. Each endpoint's list **MUST** be in the same order. The leader's endpoint **MUST** be the first in the list. The order of the encrypted_input_shares in a Report (see Section 4.2) **MUST** be the same as the order in which aggregators appear in this list.

*collector_config: The HPKE configuration of the collector (described in Section 4.2.1). Having participants agree on this absolves collectors of the burden of operating an HTTP server. See #102 for discussion.

*max_batch_lifetime: The maximum number of times a batch of reports may be used in collect requests.

*min_batch_size: The minimum number of reports that appear in a
 batch.

*min_batch_duration: The minimum time difference between the
 oldest and newest report in a batch. This defines the boundaries
 with which the batch interval of each collect request must be
 aligned. (See Section 4.4.1.)

*protocol: named parameter identifying the core PPM protocol,
 e.g., Prio or Hits.

4.2. Uploading Reports

Clients periodically upload reports to the leader, which then
distributes the individual shares to each helper.

4.2.1. Key Configuration Request

Before the client can upload its report to the leader, it must know
the public key of each of the aggregators. These are retrieved from
each aggregator by sending a request to [aggregator]/key_config,
where [aggregator] is the aggregator's endpoint URL, obtained from
the task parameters. The aggregator responds to well-formed requests
with status 200 and an HpkeConfig value:

```
struct {
  HpkeConfigId id;
  HpkeKemId kem_id;
  HpkeKdfId kdf_id;
  HpkeAeadKdfId aead_id;
  HpkePublicKey public_key;
} HpkeConfig;

uint8 HpkeConfigId;
opaque HpkePublicKey<1..2^16-1>;
uint16 HpkeAeadId; // Defined in I-D.irtf-cfrg-hpke
uint16 HpkeKemId;  // Defined in I-D.irtf-cfrg-hpke
uint16 HpkeKdfId;  // Defined in I-D.irtf-cfrg-hpke
```

[OPEN ISSUE: Decide whether to expand the width of the id, or
support multiple cipher suites (a la OHTTP/ECH).]

The client **MUST** abort if any of the following happen for any
key_config request:

  *the client and aggregator failed to establish a secure,
   aggregator-authenticated channel;

  *the GET request failed or didn't return a valid key config; or

*the key config specifies a KEM, KDF, or AEAD algorithm the client doesn't recognize.

Aggregators **SHOULD** use HTTP caching to permit client-side caching of this resource [RFC5861]. Aggregators **SHOULD** favor long cache lifetimes to avoid frequent cache revalidation, e.g., on the order of days. Aggregators can control this cached lifetime with the Cache-Control header, as follows:

Cache-Control: max-age=86400

Clients **SHOULD** follow the usual HTTP caching [RFC7234] semantics for key configurations.

Note: Long cache lifetimes may result in clients using stale HPKE keys; aggregators **SHOULD** continue to accept reports with old keys for at least twice the cache lifetime in order to avoid rejecting reports.

4.2.2. Upload Request

Clients upload reports by using an HTTP POST to [leader]/upload, where [leader] is the first entry in the task's aggregator endpoints. The payload is structured as follows:

```
struct {
  TaskID task_id;
  Time time;
  uint64 nonce;
  Extension extensions<4..2^16-1>;
  EncryptedInputShare encrypted_input_shares<1..2^16-1>;
} Report;
```

This message is called the client's *report*. It contains the following fields:

*task_id is the task ID of the task for which the report is intended.

*time is the time at which the report was generated. This field is used by the aggregators to ensure the report appears in at most one batch. (See Section 4.4.2.)

*nonce is a random number chosen by the client generating the report. This and the timestamp field are used by the aggregators to ensure that each report appears at most once in a batch. (See Section 4.4.2.)

*extensions is a list of extensions to be included in the Upload flow; see Section 4.2.3.

*encrypted_input_shares contains the encrypted input shares of
 each of the aggregators. The order in which the encrypted input
 shares appear **MUST** match the order of the task's
 aggregator_endpoints (i.e., the first share should be the
 leader's, the second share should be for the first helper, and so
 on).

[OPEN ISSUE: consider dropping nonce altogether and relying on a
more fine-grained timestamp, subject to collision analysis]

Encrypted input shares are structured as follows:

```
struct {
  HpkeConfigId aggregator_config_id;
  opaque enc<1..2^16-1>;
  opaque payload<1..2^16-1>;
} EncryptedInputShare;
```

*aggregator_config_id is equal to HpkeConfig.id, where HpkeConfig
 is the key config of the aggregator receiving the input share.

*enc is the encapsulated HPKE context, used by the aggregator to
 decrypt its input share.

*payload is the encrypted input share.

To generate the report, the client begins by encoding its
measurements as an input for the PPM scheme and splitting it into
input shares. (Note that the structure of each input share depends
on the PPM scheme in use, its parameters, and the role of
aggregator, i.e., whether the aggregator is a leader or helper.) To
encrypt an input share, the client first generates an HPKE [I-
D.irtf-cfrg-hpke] context for the aggregator by running

```
enc, context = SetupBaseS(pk,
                          "pda input share" || task_id || server_role)
```

where pk is the aggregator's public key, task_id is Report.task_id
and server_role is a byte whose value is 0x01 if the aggregator is
the leader and 0x00 if the aggregator is the helper. enc is the
encapsulated HPKE context and context is the HPKE context used by
the client for encryption. The payload is encrypted as

```
payload = context.Seal(time || nonce || extensions, input_share)
```

where input_share is the aggregator's input share and time, nonce
and extensions are the corresponding fields of Report.

The leader responds to well-formed requests to [leader]/upload with
status 200 and an empty body. Malformed requests are handled as

described in [Section 3.1](#). Clients **SHOULD NOT** upload the same
measurement value in more than one report if the leader responds
with status 200 and an empty body.

The leader responds to requests with out-of-date HpkeConfig.id
values, indicated by EncryptedInputShare.config_id, with status 400
and an error of type 'outdatedConfig'. Clients **SHOULD** invalidate any
cached aggregator HpkeConfig and retry with a freshly generated
Report. If this retried report does not succeed, clients **MUST** abort
and discontinue retrying.

4.2.3. Upload Extensions

Each UploadReq carries a list of extensions that clients may use to
convey additional, authenticated information in the report. [OPEN
ISSUE: The extensions aren't authenticated. It's probably a good
idea to be a bit more clear about how we envision extensions being
used. Right now this includes client attestation for defeating Sybil
attacks. See issue#89.] Each extension is a tag-length encoded value
of the following form:

```
struct {
    ExtensionType extension_type;
    opaque extension_data<0..2^16-1>;
} Extension;

enum {
    TBD(0),
    (65535)
} ExtensionType;
```

"extension_type" indicates the type of extension, and
"extension_data" contains information specific to the extension.

4.3. Verifying and Aggregating Reports

Once a set of clients have uploaded their reports to the leader, the
leader can send them to the helpers to be verified and aggregated.
In order to enable the system to handle very large batches of
reports, this process can be performed incrementally. To aggregate a
set of reports, the leader sends an AggregateReq to each helper
containing those report shares. The helper then processes them
(verifying the proofs and incorporating their values into the
ongoing aggregate) and replies to the leader.

The exact structure of the aggregation flow depends on the PPM
scheme. Specifically:

  *Some PPM schemes (e.g., Prio) allow the leader to start
   aggregating reports proactively before all the reports in a batch

are received. Others (e.g., Hits) require all the reports to be
present and must be initiated by the collector.

*Processing the reports -- especially verifying the proofs -- may
require multiple round trips.

Note that it is possible to aggregate reports from one batch while
reports from the next batch are coming in.

This process is illustrated below in Figure 2. In this example, the
batch size is 20, but the leader opts to process the reports in sub-
batches of 10. Each sub-batch takes two round-trips to process. Once
both sub-batches have been processed, the leader can issue an
OutputShareReq in order to retrieve the helper's aggregated result.

In order to allow the helpers to retain minimal state, the helper
can attach a state parameter to its response, with the leader
returning the state value in the next request, thus offloading the
state to the leader. This state value **MUST** be cryptographically
protected as described in Section 4.3.1.2.


```
Leader                                                      Helper

AggregateReq (Reports 1-10) ------------------------------->  \
<-------------------------------- AggregateResp (State 1)  | Reports
AggregateReq (continued, State 1)    -------------------->  | 10-11
<-------------------------------- AggregateResp (State 2)  /


AggregateReq (Reports 11-20, State 2) --------------------->  \
<-------------------------------- AggregateResp (State 3)  | Reports
AggregateReq (continued, State 3) ------------------------->  | 20-21
<-------------------------------- AggregateResp (State 4) /

OutputShareReq (State 4) ----------------------------------->
<------------------------------- OutputShareResp (Result)
```
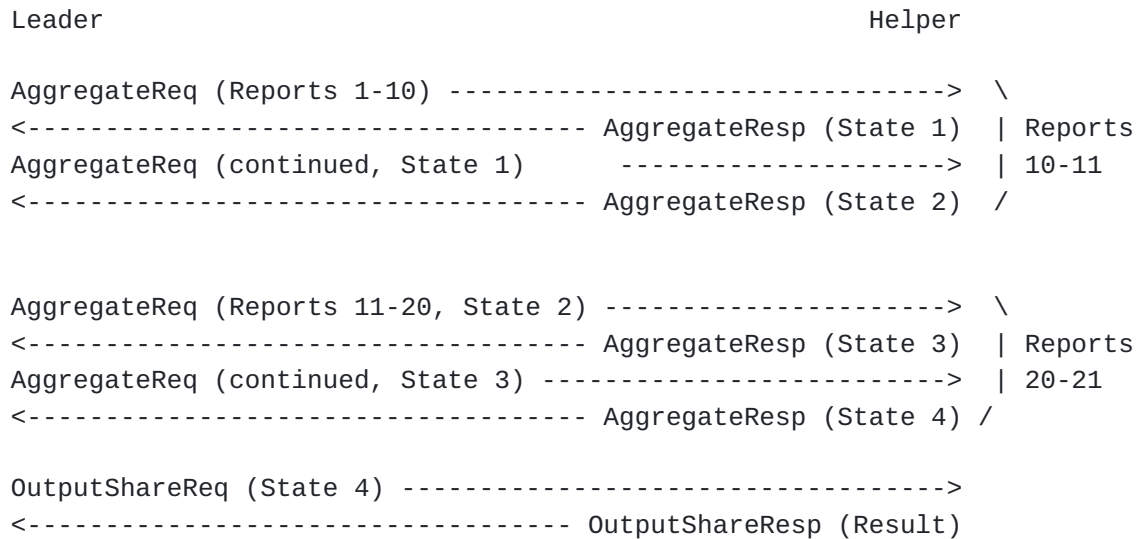
Figure 2: Aggregation Process (batch size=20)

[OPEN ISSUE: Should there be an indication of whether a given
AggregateReq is a continuation of a previous sub-batch?]

[TODO: Decide if and how the collector's request is authenticated.]

4.3.1. Aggregate Request

The AggregateReq request is used by the leader to send a set of
reports to the helper. These reports **MUST** all be associated with the
same PPM task. [[OPEN ISSUE: And the same batch, right?]]

For each aggregator endpoint [aggregator] in AggregateReq.task_id's
parameters except its own, the leader sends a POST request to
[aggregator]/aggregate with the following message:

```
struct {
  TaskID task_id;
  opaque helper_state<0..2^16>;
  AggregateSubReq seq<1..2^24-1>;
} AggregateReq;
```

The structure contains the PPM task, an opaque *helper state* string,
and a sequence of *sub-requests*, each corresponding to a unique
client report. Sub-requests are structured as follows:

```
struct {
  Time time;                          // Equal to Report.time.
  uint64 nonce;                       // Equal to Report.nonce.
  Extension extensions<4..2^16-1>; // Equal to Report.extensions.
  EncryptedInputShare helper_share;
  select (protocol) { // Protocol for the PPM task
    case prio: PrioAggregateSubReq;
    case hits: HitsAggregateSubReq;
  }
} AggregateSubReq;
```

The time, nonce, and extensions fields have the same value as those
in the report uploaded by the client. Similarly, the helper_share
field is the EncryptedInputShare from the Report whose index in
Report.encrypted_input_shares is equal to the index of [aggregator]
in the task's aggregator endpoints. [OPEN ISSUE: We usually only
need to send this in the first aggregate request. Shall we exclude
it in subsequent requests somehow?] The remainder of the structure
is dedicated to the protocol-specific request parameters.

In order to provide replay protection, the leader is required to
send aggregate sub-requests in ascending order, where the ordering
on sub-requests is determined by the algorithm defined in Section
4.4.2. Specifically, the leader constructs its request such that:

   *each sub-request follows the previous sub-request; and

   *the first sub-request follows the last sub-request in the
    previous aggregate request.

The helper handles well-formed requests as follows. (As usual,
malformed requests are handled as described in Section 3.1.) It
first looks for PPM parameters corresponding to
AggregateReq.task_id. It then filters out out-of-order sub-requests
by ignoring any sub-request that does not follow the previous one
(See Section 4.4.2.)

The response is an HTTP 200 OK with a body consisting of the
helper's updated state and a sequence of *sub-responses*, where each
sub-response corresponds to the sub-request in the same position in
AggregateReq. The structure of each sub-response is specific to the
PPM protocol:

```
struct {
  opaque helper_state<0..2^16>;
  AggregateSubResp seq<1..2^24-1>;
} AggregateResp;

struct {
  Time time;      // Equal to AggregateSubReq.time.
  uint64 nonce;  // Equal to AggregateSubReq.nonce.
  select (protocol) { // Protocol for the PPM task
    case prio: PrioAggregateSubResp;
    case hits: HitsAggregateSubResp;
  }
} AggregateSubResp;
```

The helper handles each sub-request AggregateSubReq as follows. It
first looks up the HPKE config and corresponding secret key
associated with helper_share.config_id. If not found, then the sub-
response consists of an "unrecognized config" alert. [TODO: We'll
want to be more precise about what this means. See issue#57.] Next,
it attempts to decrypt the payload with the following procedure:

```
context = SetupBaseR(helper_share.enc, sk,
                     "pda input share" || task_id || server_role)
input_share = context.Open(time || nonce || extensions, helper_share)
```

where sk is the HPKE secret key, task_id is AggregateReq.task_id and
server_role is the role of the server (0x01 for the leader and 0x00
for the helper). time, nonce and extensions are obtained from the
corresponding fields in AggregateSubReq. If decryption fails, then
the sub-response consists of a "decryption error" alert. [See
issue#57.] Otherwise, the helper handles the request for its
plaintext input share input_share and updates its state as specified
by the PPM protocol.

After processing all of the sub-requests, the helper encrypts its
updated state and constructs its response to the aggregate request.

4.3.1.1. Leader State

The leader is required to issue aggregate requests in order, but
reports are likely to arrive out-of-order. The leader **SHOULD** store
reports for a time period proportional to the batch window before
including them in an aggregate request. Failure to do so may result
in out-of-order reports being dropped by the helper. The leader **MUST**

NOT accept reports whose timestamps are in the future. Implementors **MAY** provide for some small leeway, usually no more than a few minutes, to account for clock skew.

4.3.1.2. Helper State

The helper state is an optional parameter of an aggregate request that the helper can use to carry state across requests. At least part of the state will usually need to be encrypted in order to protect user privacy. However, the details of precisely how the state is encrypted and the information that it carries is up to the helper implementation.

4.3.2. Output Share Request

Once the aggregators have verified at least as many reports as required for the PPM task, the leader issues an *output share request* to each helper. The helper responds to this request by extracting its output share from its state and encrypting it under the collector's HPKE public key.

For each aggregator endpoint [aggregator] in the parameters associated with CollectReq.task_id (see Section 4.4) except its own, the leader sends a POST request to [aggregator]/output_share with the following message:

```
struct {
  TaskID task_id;
  Interval batch_interval;
  opaque helper_state<0..2^16>;
} OutputShareReq;
```

  *task_id is the task ID associated with the PPM parameters.

  *batch_interval is the batch interval of the request.

  *helper_state is the helper's state, which is carried across
   requests from the leader.

To respond to an output share request, the helper first looks up the PPM parameters associated with task task_id. Then, using the procedure in Section 4.4.1, it ensures that the request meets the requirements of the batch parameters. If so, it aggregates all valid input shares that fall in the batch interval into an output share. The format of the output share is specific to the PPM protocol:

```
struct {
  select (protocol) { // Protocol for CollectReq.task_id
    case prio: PrioOutputShare;
    case hits: HitsOutputShare;
  }
} OutputShare;
```

Next, the helper encrypts the output share output_share under the collector's public key as follows:

```
enc, context = SetupBaseS(pk,
                          "pda output share" || task_id || server_role)
encrypted_output_share = context.Seal(batch_interval, output_share)
```

where pk is the HPKE public key encoded by the collector's HPKE key configuration, task_id is OutputShareReq.task_id and server_role is the role of the server (0x01 for the leader and 0x00 for the helper). output_share is the serialized OutputShare, and batch_interval is obtained from the OutputShareReq.

This encryption prevents the leader from learning the actual result, as it only has its own share and not the helper's share, which is encrypted for the collector. The helper responds to the collector with HTTP status 200 OK and a body consisting of the following structure:

```
struct {
  HpkeConfigId collector_hpke_config_id;
  opaque enc<1..2^16-1>;
  opaque payload<1..2^16>;
} EncryptedOutputShare;
```

   *collector_hpke_config_id is collector_config.id from the task parameters corresponding to CollectReq.task_id.

   *enc is the encapsulated HPKE context, used by the collector to decrypt the output share.

   *payload is an encrypted OutputShare.

The leader uses the helper's output share response to respond to the collector's collect request (see [Section 4.4](#)).

4.4. Collecting Results

The collector uses CollectReq to ask the leader to collect and return the results for a given PPM task over a given time period. To make a collect request, the collector issues a POST request to [leader]/collect, where [leader] is the leader's endpoint URL. The body of the request is structured as follows:

```
struct {
  TaskID task_id;
  Interval batch_interval;
  select (protocol) { // Protocol corresponding to task_id
    case prio: PrioCollectReq;
    case hits: HitsCollectReq;
  }
} CollectReq;
```

The named parameters are:

*task_id, the PPM task ID.

*batch_interval, the request's batch interval.

The remainder of the message is dedicated to the protocol-specific request parameters.

Depending on the PPM scheme and how the leader is configured, the collect request may cause the leader to send a series of aggregate requests to the helpers in order to compute their share of the output. Alternately, the leader may already have made these requests and can respond immediately. In either case it responds to the collector's request as follows.

It begins by checking that the request meets the requirements of the batch parameters using the procedure in Section 4.4.1. If so, it obtains the helper's encrypted output share for the batch interval by sending an output share request to the helper as described in Section 4.3.2. (This request may too have been made in advance.)

Next, the leader computes its own output share by aggregating all of the valid input shares that fall within the batch interval. Finally, it responds with HTTP status 200 and a body consisting of a CollectResp message:

[OPEN ISSUE: What happens if this all takes a really long time.]

[TODO: Decide if and how the collector's request is authenticated.]

```
struct {
  EncryptedOutputShare shares<1..2^16-1>;
} CollectResp;
```

*shares is a vector of EncryptedOutputShares, as described in Section 4.3.2, except that for the leader's share, the task_id and batch_interval used to encrypt the OutputShare are obtained from the CollectReq.

[OPEN ISSUE: Describe how intra-protocol errors yield collect errors
(see issue#57). For example, how does a leader respond to a collect
request if the helper drops out?]

4.4.1. Validating Batch Parameters

Before an aggregator responds to a collect request or output share
request, it must first check that the request does not violate the
parameters associated with the PPM task. It does so as described
here.

First the aggregator checks that the request's batch interval
respects the boundaries defined by the PPM task's parameters.
Namely, it checks that both batch_interval.start and
batch_interval.end are divisible by min_batch_duration and that
batch_interval.end - batch_interval.start >= min_batch_duration.
Unless both these conditions are true, it aborts and alerts the peer
with "invalid batch interval".

Next, the aggregator checks that the request respects the generic
privacy parameters of the PPM task. Let X denote the set of input
shares the aggregator has validated and which fall in the batch
interval of the request.

  *If len(X) < min_batch_size, then the aggregator aborts and alerts
   the peer with "insufficient batch size".

  *The aggregator keeps track of the number of times each input
   share was added to the batch of an output share request. If any
   input share in X was added to at least max_batch_lifetime
   previous batches, then the helper aborts and alerts the peer with
   "request exceeds the batch's privacy budget".

4.4.2. Anti-replay

Using a client-provided report multiple times within a single batch,
or using the same report in multiple batches, may allow a server to
learn information about the client's measurement, violating the
privacy property of PPM. To prevent such replay attacks, this
specification defines a total ordering on reports that aggregators
can use to ensure that reports are aggregated once.

Aggregate requests are ordered as follows: We say that a report R2
follows report R1 if either R2.time > R1.time or R2.time == R1.time
and R2.nonce > R1.nonce. If R2.time < R1.time, or R2.time == R1.time
but R2.nonce <= R1.nonce, then we say that R2 does not follow R1.

To prevent replay attacks, each aggregator ensures that each report
it aggregates follows the previously aggregated report. To prevent
the adversary from tampering with the ordering of reports, honest

clients incorporate the ordering-sensitive parameters (time, nonce) into the AAD for HPKE encryption. Note that this strategy may result in dropping reports that happen to have the same timestamp and nonce value.

Aggregators prevent the same report from being used in multiple batches (except as required by the protocol) by only responding to valid collect requests, as described in [Section 4.4.1](#).

5. Operational Considerations

PPM protocols have inherent constraints derived from the tradeoff between privacy guarantees and computational complexity. These tradeoffs influence how applications may choose to utilize services implementing the specification.

5.1. Protocol participant capabilities

The design in this document has different assumptions and requirements for different protocol participants, including clients, aggregators, and collectors. This section describes these capabilities in more detail.

5.1.1. Client capabilities

Clients have limited capabilities and requirements. Their only inputs to the protocol are (1) the parameters configured out of band and (2) a measurement. Clients are not expected to store any state across any upload flows, nor are they required to implement any sort of report upload retry mechanism. By design, the protocol in this document is robust against individual client upload failures since the protocol output is an aggregate over all inputs.

5.1.2. Aggregator capabilities

Helpers and leaders have different operational requirements. The design in this document assumes an operationally competent leader, i.e., one that has no storage or computation limitations or constraints, but only a modestly provisioned helper, i.e., one that has computation, bandwidth, and storage constraints. By design, leaders must be at least as capable as helpers, where helpers are generally required to:

  *Support the collect protocol, which includes validating and aggregating reports; and

  *Publish and manage an HPKE configuration that can be used for the upload protocol.

In addition, for each PPM task, helpers are required to:

  *Implement some form of batch-to-report index, as well as inter-
   and intra-batch replay mitigation storage, which includes some
   way of tracking batch report size with optional support for state
   offloading. Some of this state may be used for replay attack
   mitigation. The replay mitigation strategy is described in
   [Section 4.4.2](#).

Beyond the minimal capabilities required of helpers, leaders are
generally required to:

  *Support the upload protocol and store reports; and

  *Track batch report size during each collect flow and request
   encrypted output shares from helpers.

In addition, for each PPM task, leaders are required to:

  *Implement and store state for the form of inter- and intra-batch
   replay mitigation in [Section 4.4.2](#); and

  *Store helper state.

5.1.3. Collector capabilities

   Collectors statefully interact with aggregators to produce an
   aggregate output. Their input to the protocol is the task
   parameters, configured out of band, which include the corresponding
   batch window and size. For each collect invocation, collectors are
   required to keep state from the start of the protocol to the end as
   needed to produce the final aggregate output.

   Collectors must also maintain state for the lifetime of each task,
   which includes key material associated with the HPKE key
   configuration.

5.2. Data resolution limitations

   Privacy comes at the cost of computational complexity. While affine-
   aggregatable encodings (AFEs) can compute many useful statistics,
   they require more bandwidth and CPU cycles to account for finite-
   field arithmetic during input-validation. The increased work from
   verifying inputs decreases the throughput of the system or the
   inputs processed per unit time. Throughput is related to the
   verification circuit's complexity and the available compute-time to
   each aggregator.

   Applications that utilize proofs with a large number of
   multiplication gates or a high frequency of inputs may need to limit

inputs into the system to meet bandwidth or compute constraints.
Some methods of overcoming these limitations include choosing a
better representation for the data or introducing sampling into the
data collection methodology.

[[TODO: Discuss explicit key performance indicators, here or
elsewhere.]]

5.3. Aggregation utility and soft batch deadlines

A soft real-time system should produce a response within a deadline
to be useful. This constraint may be relevant when the value of an
aggregate decreases over time. A missed deadline can reduce an
aggregate's utility but not necessarily cause failure in the system.

An example of a soft real-time constraint is the expectation that
input data can be verified and aggregated in a period equal to data
collection, given some computational budget. Meeting these deadlines
will require efficient implementations of the input-validation
protocol. Applications might batch requests or utilize more
efficient serialization to improve throughput.

Some applications may be constrained by the time that it takes to
reach a privacy threshold defined by a minimum number of input
shares. One possible solution is to increase the reporting period so
more samples can be collected, balanced against the urgency of
responding to a soft deadline.

5.4. Protocol-specific optimizations

Not all PPM tasks have the same operational requirements, so the
protocol is designed to allow implementations to reduce operational
costs in certain cases.

5.4.1. Reducing storage requirements

In general, the aggregators are required to keep state for all valid
reports for as long as collect requests can be made for them. In
particular, the aggregators must store a batch as long as the batch
has not been queried more than max_batch_lifetime times. However, it
is not always necessary to store the reports themselves. For schemes
like Prio in which the input-validation protocol is only run once
per input share, each aggregator only needs to store the aggregate
output share for each possible batch interval, along with the number
of times the output share was used in a batch. (The helper may store
its output shares in its encrypted state, thereby offloading this
state to the leader.) This is due to the requirement that the batch
interval respect the boundaries defined by the PPM parameters. (See
Section 4.4.1.)

6. Security Considerations

Prio assumes a powerful adversary with the ability to compromise an
unbounded number of clients. In doing so, the adversary can provide
malicious (yet truthful) inputs to the aggregation function. Prio
also assumes that all but one server operates honestly, where a
dishonest server does not execute the protocol faithfully as
specified. The system also assumes that servers communicate over
secure and mutually authenticated channels. In practice, this can be
done by TLS or some other form of application-layer authentication.

In the presence of this adversary, Prio provides two important
properties for computing an aggregation function F:

1. Privacy. The aggregators and collector learn only the output of
   F computed over all client inputs, and nothing else.

2. Robustness. As long as the aggregators execute the input-
   validation protocol correctly, a malicious client can skew the
   output of F only by reporting false (untruthful) input. The
   output cannot be influenced in any other way.

There are several additional constraints that a Prio deployment must
satisfy in order to achieve these goals:

1. Minimum batch size. The aggregation batch size has an obvious
   impact on privacy. (A batch size of one hides nothing of the
   input.)

2. Aggregation function choice. Some aggregation functions leak
   slightly more than the function output itself.

[TODO: discuss these in more detail.]

6.1. Threat model

In this section, we enumerate the actors participating in the Prio
system and enumerate their assets (secrets that are either
inherently valuable or which confer some capability that enables
further attack on the system), the capabilities that a malicious or
compromised actor has, and potential mitigations for attacks enabled
by those capabilities.

This model assumes that all participants have previously agreed upon
and exchanged all shared parameters over some unspecified secure
channel.

6.1.1. Client/user

6.1.1.1. Assets

1. Unshared inputs. Clients are the only actor that can ever see the original inputs.

2. Unencrypted input shares.

6.1.1.2. Capabilities

1. Individual users can reveal their own input and compromise their own privacy.

2. Clients (that is, software which might be used by many users of the system) can defeat privacy by leaking input outside of the Prio system.

3. Clients may affect the quality of aggregations by reporting false input.

   *Prio can only prove that submitted input is valid, not that it is true. False input can be mitigated orthogonally to the Prio protocol (e.g., by requiring that aggregations include a minimum number of contributions) and so these attacks are considered to be outside of the threat model.

4. Clients can send invalid encodings of input.

6.1.1.3. Mitigations

1. The input validation protocol executed by the aggregators prevents either individual clients or coalitions of clients from compromising the robustness property.

2. If aggregator output satisifes differential privacy [Section 6.5](#), then all records not leaked by malicious clients are still protected.

6.1.2. Aggregator

6.1.2.1. Assets

1. Unencrypted input shares.

2. Input share decryption keys.

3. Client identifying information.

4. Output shares.

5. Aggregator identity.

6.1.2.2. Capabilities

1. Aggregators may defeat the robustness of the system by emitting bogus output shares.

2. If clients reveal identifying information to aggregators (such as a trusted identity during client authentication), aggregators can learn which clients are contributing input.

    1. Aggregators may reveal that a particular client contributed input.

    2. Aggregators may attack robustness by selectively omitting inputs from certain clients.

        *For example, omitting submissions from a particular geographic region to falsely suggest that a particular localization is not being used.

3. Individual aggregators may compromise availability of the system by refusing to emit output shares.

4. Input validity proof forging. Any aggregator can collude with a malicious client to craft a proof that will fool honest aggregators into accepting invalid input.

5. Aggregators can count the total number of input shares, which could compromise user privacy (and differential privacy [Section 6.5](#)) if the presence or absence of a share for a given user is sensitive.

6.1.2.3. Mitigations

1. The linear secret sharing scheme employed by the client ensures that privacy is preserved as long as at least one aggregator does not reveal its input shares.

2. If computed over a sufficient number of input shares, output shares reveal nothing about either the inputs or the participating clients.

3. Clients can ensure that aggregate counts are non-sensitive by generating input independently of user behavior. For example, a client should periodically upload a report even if the event that the task is tracking has not occurred, so that the absence of reports cannot be distinguished from their presence.

4. Bogus inputs can be generated that encode "null" shares that do
   not affect the aggregate output, but mask the total number of
   true inputs.

   *Either leaders or clients can generate these inputs to mask
    the total number from non-leader aggregators or all the
    aggregators, respectively.

   *In either case, care must be taken to ensure that bogus
    inputs are indistinguishable from true inputs (metadata,
    etc), especially when constructing timestamps on reports.

[OPEN ISSUE: Define what "null" shares are. They should be defined
such that inserting null shares into an aggregation is effectively a
no-op. See issue#98.]

6.1.3. Leader

The leader is also an aggregator, and so all the assets,
capabilities and mitigations available to aggregators also apply to
the leader.

6.1.3.1. Capabilities

1. Input validity proof verification. The leader can forge proofs
   and collude with a malicious client to trick aggregators into
   aggregating invalid inputs.

   *This capability is no stronger than any aggregator's ability
    to forge validity proof in collusion with a malicious
    client.

2. Relaying messages between aggregators. The leader can
   compromise availability by dropping messages.

   *This capability is no stronger than any aggregator's ability
    to refuse to emit output shares.

3. Shrinking the anonymity set. The leader instructs aggregators
   to construct output parts and so could request aggregations
   over few inputs.

6.1.3.2. Mitigations

1. Aggregators enforce agreed upon minimum aggregation thresholds
   to prevent deanonymizing.

2. If aggregator output satisfies differential privacy [Section
   6.5](#), then genuine records are protected regardless of the size
   of the anonymity set.

6.1.4. Collector

6.1.4.1. Capabilities

1. Advertising shared configuration parameters (e.g., minimum
   thresholds for aggregations, joint randomness, arithmetic
   circuits).

2. Collectors may trivially defeat availability by discarding
   output shares submitted by aggregators.

3. Known input injection. Collectors may collude with clients to
   send known input to the aggregators, allowing collectors to
   shrink the effective anonymity set by subtracting the known
   inputs from the final output. Sybil attacks [Dou02] could be
   used to amplify this capability.

6.1.4.2. Mitigations

1. Aggregators should refuse shared parameters that are trivially
   insecure (i.e., aggregation threshold of 1 contribution).

2. If aggregator output satisfies differential privacy Section
   6.5, then genuine records are protected regardless of the size
   of the anonymity set.

6.1.5. Aggregator collusion

If all aggregators collude (e.g. by promiscuously sharing
unencrypted input shares), then none of the properties of the system
hold. Accordingly, such scenarios are outside of the threat model.

6.1.6. Attacker on the network

We assume the existence of attackers on the network links between
participants.

6.1.6.1. Capabilities

1. Observation of network traffic. Attackers may observe messages
   exchanged between participants at the IP layer.

   1. The time of transmission of input shares by clients could
      reveal information about user activity.

      *For example, if a user opts into a new feature, and the
       client immediately reports this to aggregators, then
       just by observing network traffic, the attacker can
       infer what the user did.

2. Observation of message size could allow the attacker to
      learn how much input is being submitted by a client.

         *For example, if the attacker observes an encrypted
          message of some size, they can infer the size of the
          plaintext, plus or minus the cipher block size. From
          this they may be able to infer which aggregations the
          user has opted into or out of.

   2. Tampering with network traffic. Attackers may drop messages or
      inject new messages into communications between participants.

6.1.6.2. Mitigations

   1. All messages exchanged between participants in the system
      should be encrypted.

   2. All messages exchanged between aggregators, the collector and
      the leader should be mutually authenticated so that network
      attackers cannot impersonate participants.

   3. Clients should be required to submit inputs at regular
      intervals so that the timing of individual messages does not
      reveal anything.

   4. Clients should submit dummy inputs even for aggregations the
      user has not opted into.

   [[OPEN ISSUE: The threat model for Prio --- as it's described in the
   original paper and [BBCGGI19] --- considers **either** a malicious
   client (attacking soundness) **or** a malicious subset of aggregators
   (attacking privacy). In particular, soundness isn't guaranteed if
   any one of the aggregators is malicious; in theory it may be
   possible for a malicious client and aggregator to collude and break
   soundness. Is this a contingency we need to address? There are
   techniques in [BBCGGI19] that account for this; we need to figure
   out if they're practical.]]

6.2. Client authentication or attestation

   [TODO: Solve issue#89]

6.3. Anonymizing proxies

   Client reports can contain auxiliary information such as source IP,
   HTTP user agent or in deployments which use it, client
   authentication information, which could be used by aggregators to
   identify participating clients or permit some attacks on robustness.
   This auxiliary information could be removed by having clients submit
   reports to an anonymizing proxy server which would then use Oblivous

HTTP [I-D.thomson-http-oblivious] to forward inputs to the PPM
leader, without requiring any server participating in PPM to be
aware of whatever client authentication or attestation scheme is in
use.

6.4. Batch parameters

An important parameter of a PPM deployment is the minimum batch
size. If an aggregation includes too few inputs, then the outputs
can reveal information about individual participants. Aggregators
use the batch size field of the shared task parameters to enforce
minimum batch size during the collect protocol, but server
implementations may also opt out of participating in a PPM task if
the minimum batch size is too small. This document does not specify
how to choose minimum batch sizes.

The PPM parameters also specify the maximum number of times a report
can be used. Some protocols, such as Hits, require reports to be
used in multiple batches spanning multiple collect requests.

6.5. Differential privacy

Optionally, PPM deployments can choose to ensure their output F
achieves differential privacy [Vad16]. A simple approach would
require the aggregators to add two-sided noise (e.g. sampled from a
two-sided geometric distribution) to outputs. Since each aggregator
is adding noise independently, privacy can be guaranteed even if all
but one of the aggregators is malicious. Differential privacy is a
strong privacy definition, and protects users in extreme
circumstances: Even if an adversary has prior knowledge of every
input in a batch except for one, that one record is still formally
protected.

[OPEN ISSUE: While parameters configuring the differential privacy
noise (like specific distributions / variance) can be agreed upon
out of band by the aggregators and collector, there may be benefits
to adding explicit protocol support by encoding them into task
parameters.]

6.6. Robustness in the presence of malicious servers

Most PPM protocols, including Prio and Hits, are robust against
malicious clients, but are not robust against malicious servers. Any
aggregator can simply emit bogus output shares and undetectably
spoil aggregates. If enough aggregators were available, this could
be mitigated by running the protocol multiple times with distinct
subsets of aggregators chosen so that no aggregator appears in all
subsets and checking all the outputs against each other. If all the
protocol runs do not agree, then participants know that at least one
aggregator is defective, and it may be possible to identify the

defector (i.e., if a majority of runs agree, and a single aggregator appears in every run that disagrees). See [#22](#22) for discussion.

6.7. Infrastructure diversity

Prio deployments should ensure that aggregators do not have common dependencies that would enable a single vendor to reassemble inputs. For example, if all participating aggregators stored unencrypted input shares on the same cloud object storage service, then that cloud vendor would be able to reassemble all the input shares and defeat privacy.

6.8. System requirements

6.8.1. Data types

7. IANA Considerations

7.1. Protocol Message Media Types

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

  *HpkeConfig [Section 4.1](#section-4.1): "application/ppm-hpke-config"

  *Report [Section 4.2.2](#section-4.2.2): "message/ppm-report"

  *AggregateReq [Section 4.3.1](#section-4.3.1): "message/ppm-aggregate-req"

  *AggregateResp [Section 4.3.1](#section-4.3.1): "message/ppm-aggregate-resp"

  *OutputShareReq [Section 4.3.2](#section-4.3.2): "message/ppm-output-share-req"

  *OutputShareResp [Section 4.3.2](#section-4.3.2): "message/ppm-output-share-resp"

  *CollectReq [Section 4.4](#section-4.4): "message/ppm-collect-req"

  *CollectResp [Section 4.4](#section-4.4): "message/ppm-collect-req"

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.]

7.1.1. "application/ppm-hpke-config" media type

Type name: application

Subtype name: ppm-hpke-config

Required parameters: N/A

Optional parameters: None

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

Security considerations: see Section 4.1

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

7.1.2. "message/ppm-report" media type

Type name: message

Subtype name: ppm-report

Required parameters: N/A

Optional parameters:
                        None

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

Security considerations: see Section 4.2.2

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

7.1.3. "message/ppm-aggregate-req" media type

Type name: message

Subtype name: ppm-aggregate-req

Required parameters: N/A

Optional parameters: None

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

Security considerations: see Section 4.3.1

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

7.1.4. "message/ppm-aggregate-resp" media type

Type name: application

Subtype name: ppm-aggregate-resp

Required parameters: N/A

Optional parameters: None

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

Security considerations: see Section 4.3.1

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
```

7.1.5. "message/ppm-output-share-req" media type

```
Type name:  application

Subtype name:  ppm-output-share-req

Required parameters:  N/A

Optional parameters:  None

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

Security considerations:  see Section 4.3.2

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

7.1.6. "message/ppm-output-share-resp" media type

Type name: application

Subtype name: ppm-output-share-resp

Required parameters: N/A

Optional parameters: None

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

Security considerations: see [Section 4.3.2](#)

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

7.1.7. "message/ppm-collect-req" media type

Type name: application

Subtype name: ppm-collect-req

Required parameters: N/A

Optional parameters: None

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

Security considerations: see Section 4.4

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

7.1.8. "message/ppm-collect-req" media type

Type name: application

Subtype name:
                    ppm-collect-req

   Required parameters: N/A

   Optional parameters: None

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

   Security considerations: see [Section 4.4](#)

   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

7.2. Upload Extension Registry

   This document requests creation of a new registry for extensions to
   the Upload protocol. This registry should contain the following
   columns:

   [TODO: define how we want to structure this registry when the time
   comes]

7.3. URN Sub-namespace for PPM (urn:ietf:params:ppm)

The following value [will be/has been] registered in the "IETF URN
Sub- namespace for Registered Protocol Parameter Identifiers"
registry, following the template in [RFC3553]:

Registry name: ppm

Specification: [[THIS DOCUMENT]]

Repository: http://www.iana.org/assignments/ppm

Index value: No transformation needed.

Initial contents: The types and descriptions in the table in Section
3.1 above, with the Reference field set to point to this
specification.

8. Acknowledgements

The text in Section 3 is based extensively on [RFC8555]

9. References

9.1. Normative References

[I-D.irtf-cfrg-hpke] Barnes, R. L., Bhargavan, K., Lipp, B., and C.
            A. Wood, "Hybrid Public Key Encryption", Work in
            Progress, Internet-Draft, draft-irtf-cfrg-hpke-12, 2
            September 2021, <https://datatracker.ietf.org/doc/html/
            draft-irtf-cfrg-hpke-12>.

[I-D.thomson-http-oblivious] Thomson, M. and C. A. Wood, "Oblivious
            HTTP", Work in Progress, Internet-Draft, draft-thomson-
            http-oblivious-02, 24 August 2021, <https://
            datatracker.ietf.org/doc/html/draft-thomson-http-
            oblivious-02>.

[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>.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, DOI 10.17487/
            RFC2818, May 2000, <https://www.rfc-editor.org/rfc/
            rfc2818>.

[RFC3553] Mealling, M., Masinter, L., Hardie, T., and G. Klyne, "An
            IETF URN Sub-namespace for Registered Protocol

          Parameters", BCP 73, RFC 3553, DOI 10.17487/RFC3553, June
          2003, <https://www.rfc-editor.org/rfc/rfc3553>.

[RFC5861] Nottingham, M., "HTTP Cache-Control Extensions for Stale
          Content", RFC 5861, DOI 10.17487/RFC5861, May 2010,
          <https://www.rfc-editor.org/rfc/rfc5861>.

[RFC7234] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke,
          Ed., "Hypertext Transfer Protocol (HTTP/1.1): Caching",
          RFC 7234, DOI 10.17487/RFC7234, June 2014, <https://
          www.rfc-editor.org/rfc/rfc7234>.

[RFC7807] Nottingham, M. and E. Wilde, "Problem Details for HTTP
          APIs", RFC 7807, DOI 10.17487/RFC7807, March 2016,
          <https://www.rfc-editor.org/rfc/rfc7807>.

[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>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS)
          Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446,
          August 2018, <https://www.rfc-editor.org/rfc/rfc8446>.

9.2. Informative References

[BBCGGI19] Boneh, D., Boyle, E., Corrigan-Gibbs, H., Gilboa, N.,
          and Y. Ishai, "Zero-Knowledge Proofs on Secret-Shared
          Data via Fully Linear PCPs", 5 January 2021, <https://
          eprint.iacr.org/2019/188>.

[BBCGGI21] Boneh, D., Boyle, E., Corrigan-Gibbs, H., Gilboa, N.,
          and Y. Ishai, "Lightweight Techniques for Private Heavy
          Hitters", 5 January 2021, <https://eprint.iacr.org/
          2021/017>.

[CGB17]   Corrigan-Gibbs, H. and D. Boneh, "Prio: Private, Robust,
          and Scalable Computation of Aggregate Statistics", 14
          March 2017, <https://crypto.stanford.edu/prio/paper.pdf>.

[Dou02]   Douceur, J., "The Sybil Attack", 10 October 2022,
          <https://link.springer.com/chapter/
          10.1007/3-540-45748-8_24>.

[RFC8555] Barnes, R., Hoffman-Andrews, J., McCarney, D., and J.
          Kasten, "Automatic Certificate Management Environment

(ACME)", RFC 8555, DOI 10.17487/RFC8555, March 2019,
<https://www.rfc-editor.org/rfc/rfc8555>.

[Vad16]   Vadhan, S., "The Complexity of Differential Privacy", 9
           August 2016, <https://privacytools.seas.harvard.edu/
           files/privacytools/files/complexityprivacy_1.pdf>.

Authors' Addresses

    Tim Geoghegan
    ISRG

    Email: timgeog+ietf@gmail.com

    Christopher Patton
    Cloudflare

    Email: chrispatton+ietf@gmail.com

    Eric Rescorla
    Mozilla

    Email: ekr@rtfm.com

    Christopher A. Wood
    Cloudflare

    Email: caw@heapingbits.net