Workgroup: CFRG
Internet-Draft: draft-irtf-cfrg-vdaf-04
Published: 24 February 2023
Intended Status: Informational
Expires: 28 August 2023
Authors: R. L. Barnes (Cisco), D. Cook (ISRG), C. Patton (Cloudflare), P. Schoppmann (Google)

Verifiable Distributed Aggregation Functions

Abstract

This document describes Verifiable Distributed Aggregation Functions (VDAFs), a family of multi-party protocols for computing aggregate statistics over user measurements. These protocols are designed to ensure that, as long as at least one aggregation server executes the protocol honestly, individual measurements are never seen by any server in the clear. At the same time, VDAFs allow the servers to detect if a malicious or misconfigured client submitted an input that would result in an incorrect aggregate result.

Discussion Venues

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

Discussion of this document takes place on the Crypto Forum Research Group mailing list (cfrg@ietf.org), which is archived at https://mailarchive.ietf.org/arch/search/?email_list=cfrg.

Source for this draft and an issue tracker can be found at https://github.com/cjpatton/vdaf.

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 28 August 2023.

Copyright Notice

Copyright (c) 2023 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction
  1.1. Change Log
2. Conventions and Definitions
3. Overview
4. Definition of DAFs
  4.1. Sharding
  4.2. Preparation
  4.3. Validity of Aggregation Parameters
  4.4. Aggregation
  4.5. Unsharding
  4.6. Execution of a DAF
5. Definition of VDAFs
  5.1. Sharding
  5.2. Preparation
  5.3. Validity of Aggregation Parameters
  5.4. Aggregation
  5.5. Unsharding
  5.6. Execution of a VDAF
6. Preliminaries
  6.1. Finite Fields
    6.1.1. Auxiliary Functions
    6.1.2. FFT-Friendly Fields
    6.1.3. Parameters
  6.2. Pseudorandom Generators
    6.2.1. PrgSha3
    6.2.2. The Customization and Binder Strings
7. Prio3
  7.1. Fully Linear Proof (FLP) Systems
    7.1.1. Encoding the Input
  7.2. Construction
    7.2.1. Sharding
    7.2.2. Preparation

7.2.3. Validity of Aggregation Parameters
7.2.4. Aggregation
7.2.5. Unsharding
7.2.6. Auxiliary Functions
7.3. A General-Purpose FLP
7.3.1. Overview
7.3.2. Validity Circuits
7.3.3. Construction
7.4. Instantiations
7.4.1. Prio3Count
7.4.2. Prio3Sum
7.4.3. Prio3Histogram
8. Poplar1
8.1. Incremental Distributed Point Functions (IDPFs)
8.2. Construction
8.2.1. Client
8.2.2. Preparation
8.2.3. Validity of Aggregation Parameters
8.2.4. Aggregation
8.2.5. Unsharding
8.2.6. Auxiliary Functions
8.3. The IDPF scheme of BBCGGI21
8.3.1. Key Generation
8.3.2. Key Evaluation
8.3.3. Auxiliary Functions
8.4. Instantiation
9. Security Considerations
9.1. Requirements for the Verification Key
9.2. Requirements for the Nonce
9.3. Requirements for the Aggregation Parameters
10. IANA Considerations
11. References
11.1. Normative References
11.2. Informative References
Acknowledgments
Test Vectors
Prio3Count
Prio3Sum
Prio3Histogram
Poplar1
Sharding
Preparation, Aggregation, and Unsharding
Authors' Addresses

1. Introduction

The ubiquity of the Internet makes it an ideal platform for measurement of large-scale phenomena, whether public health trends or the behavior of computer systems at scale. There is substantial

overlap, however, between information that is valuable to measure and information that users consider private.

For example, consider an application that provides health information to users. The operator of an application might want to know which parts of their application are used most often, as a way to guide future development of the application. Specific users' patterns of usage, though, could reveal sensitive things about them, such as which users are researching a given health condition.

In many situations, the measurement collector is only interested in aggregate statistics, e.g., which portions of an application are most used or what fraction of people have experienced a given disease. Thus systems that provide aggregate statistics while protecting individual measurements can deliver the value of the measurements while protecting users' privacy.

Most prior approaches to this problem fall under the rubric of "differential privacy (DP)" [Dwo06]. Roughly speaking, a data aggregation system that is differentially private ensures that the degree to which any individual measurement influences the value of the aggregate result can be precisely controlled. For example, in systems like RAPPOR [EPK14], each user samples noise from a well-known distribution and adds it to their input before submitting to the aggregation server. The aggregation server then adds up the noisy inputs, and because it knows the distribution from whence the noise was sampled, it can estimate the true sum with reasonable precision.

Differentially private systems like RAPPOR are easy to deploy and provide a useful guarantee. On its own, however, DP falls short of the strongest privacy property one could hope for. Specifically, depending on the "amount" of noise a client adds to its input, it may be possible for a curious aggregator to make a reasonable guess of the input's true value. Indeed, the more noise the clients add, the less reliable will be the server's estimate of the output. Thus systems employing DP techniques alone must strike a delicate balance between privacy and utility.

The ideal goal for a privacy-preserving measurement system is that of secure multi-party computation (MPC): No participant in the protocol should learn anything about an individual input beyond what it can deduce from the aggregate. In this document, we describe Verifiable Distributed Aggregation Functions (VDAFs) as a general class of protocols that achieve this goal.

VDAF schemes achieve their privacy goal by distributing the computation of the aggregate among a number of non-colluding aggregation servers. As long as a subset of the servers executes the

protocol honestly, VDAFs guarantee that no input is ever accessible to any party besides the client that submitted it. At the same time, VDAFs are "verifiable" in the sense that malformed inputs that would otherwise garble the output of the computation can be detected and removed from the set of input measurements.

In addition to these MPC-style security goals, VDAFs can be composed with various mechanisms for differential privacy, thereby providing the added assurance that the aggregate result itself does not leak too much information about any one measurement.

> TODO(issue #94) Provide guidance for local and central DP and point to it here.

The cost of achieving these security properties is the need for multiple servers to participate in the protocol, and the need to ensure they do not collude to undermine the VDAF's privacy guarantees. Recent implementation experience has shown that practical challenges of coordinating multiple servers can be overcome. The Prio system [CGB17] (essentially a VDAF) has been deployed in systems supporting hundreds of millions of users: The Mozilla Origin Telemetry project [OriginTelemetry] and the Exposure Notification Private Analytics collaboration among the Internet Security Research Group (ISRG), Google, Apple, and others [ENPA].

The VDAF abstraction laid out in Section 5 represents a class of multi-party protocols for privacy-preserving measurement proposed in the literature. These protocols vary in their operational and security considerations, sometimes in subtle but consequential ways. This document therefore has two important goals:

1. Providing higher-level protocols like [DAP] with a simple, uniform interface for accessing privacy-preserving measurement schemes, and documenting relevant operational and security bounds for that interface:

    1. General patterns of communications among the various actors involved in the system (clients, aggregation servers, and the collector of the aggregate result);

    2. Capabilities of a malicious coalition of servers attempting to divulge information about client measurements; and

    3. Conditions that are necessary to ensure that malicious clients cannot corrupt the computation.

2. Providing cryptographers with design criteria that provide a clear deployment roadmap for new constructions.

This document also specifies two concrete VDAF schemes, each based on a protocol from the literature.

* The aforementioned Prio system [CGB17] allows for the privacy-preserving computation of a variety aggregate statistics. The basic idea underlying Prio is fairly simple:

  1. Each client shards its measurement into a sequence of additive shares and distributes the shares among the aggregation servers.

  2. Next, each server adds up its shares locally, resulting in an additive share of the aggregate.

  3. Finally, the aggregation servers send their aggregate shares to the data collector, who combines them to obtain the aggregate result.

  The difficult part of this system is ensuring that the servers hold shares of a valid input, e.g., the input is an integer in a specific range. Thus Prio specifies a multi-party protocol for accomplishing this task.

  In Section 7 we describe Prio3, a VDAF that follows the same overall framework as the original Prio protocol, but incorporates techniques introduced in [BBCGGI19] that result in significant performance gains.

* More recently, Boneh et al. [BBCGGI21] described a protocol called Poplar for solving the t-heavy-hitters problem in a privacy-preserving manner. Here each client holds a bit-string of length n, and the goal of the aggregation servers is to compute the set of inputs that occur at least t times. The core primitive used in their protocol is a specialized Distributed Point Function (DPF) [GI14] that allows the servers to "query" their DPF shares on any bit-string of length shorter than or equal to n. As a result of this query, each of the servers has an additive share of a bit indicating whether the string is a prefix of the client's input. The protocol also specifies a multi-party computation for verifying that at most one string among a set of candidates is a prefix of the client's input.

  In Section 8 we describe a VDAF called Poplar1 that implements this functionality.

Finally, perhaps the most complex aspect of schemes like Prio3 and Poplar1 is the process by which the client-generated measurements are prepared for aggregation. Because these constructions are based on secret sharing, the servers will be required to exchange some amount of information in order to verify the measurement is valid

and can be aggregated. Depending on the construction, this process may require multiple round trips over the network.

There are applications in which this verification step may not be necessary, e.g., when the client's software is run a trusted execution environment. To support these applications, this document also defines Distributed Aggregation Functions (DAFs) as a simpler class of protocols that aim to provide the same privacy guarantee as VDAFs but fall short of being verifiable.

> OPEN ISSUE Decide if we should give one or two example DAFs. There are natural variants of Prio3 and Poplar1 that might be worth describing.

The remainder of this document is organized as follows: Section 3 gives a brief overview of DAFs and VDAFs; Section 4 defines the syntax for DAFs; Section 5 defines the syntax for VDAFs; Section 6 defines various functionalities that are common to our constructions; Section 7 describes the Prio3 construction; Section 8 describes the Poplar1 construction; and Section 9 enumerates the security considerations for VDAFs.

1.1. Change Log

(*) Indicates a change that breaks wire compatibility with the previous draft.

04:

- Align security considerations with the security analysis of [DPRS23].

- Vdaf: Pass the nonce to the sharding algorithm.

- Vdaf: Rather than allow the application to choose the nonce length, have each implementation of the Vdaf interface specify the expected nonce length. (*)

- Prg: Split "info string" into two components: the "customization string", intended for domain separation; and the "binder string", used to bind the output to ephemeral values, like the nonce, associated with execution of a (V)DAF.

- Replace PrgAes128 with PrgSha3, an implementation of the Prg interface based on SHA-3, and use the new scheme as the default. Accordingly, replace Prio3Aes128Count with Prio3Count, Poplar1Aes128 with Poplar1, and so on. SHA-3 is a safer choice for instantiating a random oracle, which is used in the analysis of Prio3 of [DPRS23]. (*)

* Prio3, Poplar1: Ensure each invocation of the Prg uses a distinct customization string, as suggested by [DPRS23]. This is intended to make domain separation clearer, thereby simplifying security analysis. (*)

* Prio3: Replace "joint randomness hints" sent in each input share with "joint randomness parts" sent in the public share. This reduces communication overhead when the number of shares exceeds two. (*)

* Prio3: Bind nonce to joint randomness parts. This is intended to address birthday attacks on robustness pointed out by [DPRS23]. (*)

* Poplar1: Use different Prg invocations for producing the correlated randomness for inner and leaf nodes of the IDPF tree. This is intended to simplify implementations. (*)

* Poplar1: Don't bind the candidate prefixes to the verifier randomness. This is intended to improve performance, while not impacting security. According to the analysis of [DPRS23], it is necessary to restrict Poplar1 usage such that no report is aggregated more than once at a given level of the IDPF tree; otherwise, attacks on privacy may be possible. In light of this restriction, there is no added benefit of binding to the prefixes themselves. (*)

* Poplar1: During preparation, assert that all candidate prefixes are unique and appear in order. Uniqueness is required to avoid erroneously rejecting a valid report; the ordering constraint ensures the uniqueness check can be performed efficiently. (*)

* Poplar1: Increase the maximum candidate prefix count in the encoding of the aggregation parameter. (*)

* Poplar1: Bind the nonce to the correlated randomness derivation. This is intended to provide defense-in-depth by ensuring the Aggregators reject the report if the nonce does not match what the Client used for sharding. (*)

* Poplar1: Clarify that the aggregation parameter encoding is **OPTIONAL**. Accordingly, update implementation considerations around cross-aggregation state.

* IdpfPoplar: Add implementation considerations around branching on the values of control bits.

* IdpfPoplar: When decoding the the control bits in the public share, assert that the trailing bits of the final byte are all zero. (*)

03:

* Define codepoints for (V)DAFs and use them for domain separation in Prio3 and Poplar1. (*)

* Prio3: Align joint randomness computation with revised paper [BBCGGI19]. This change mitigates an attack on robustness. (*)

* Prio3: Remove an intermediate PRG evaluation from query randomness generation. (*)

* Add additional guidance for choosing FFT-friendly fields.

02:

* Complete the initial specification of Poplar1.

* Extend (V)DAF syntax to include a "public share" output by the Client and distributed to all of the Aggregators. This is to accommodate "extractable" IDPFs as required for Poplar1. (See [BBCGGI21], Section 4.3 for details.)

* Extend (V)DAF syntax to allow the unsharding step to take into account the number of measurements aggregated.

* Extend FLP syntax by adding a method for decoding the aggregate result from a vector of field elements. The new method takes into account the number of measurements.

* Prio3: Align aggregate result computation with updated FLP syntax.

* Prg: Add a method for statefully generating a vector of field elements.

* Field: Require that field elements are fully reduced before decoding. (*)

* Define new field Field255.

01:

* Require that VDAFs specify serialization of aggregate shares.

* Define Distributed Aggregation Functions (DAFs).

* Prio3: Move proof verifier check from prep_next() to prep_shares_to_prep(). (*)

* Remove public parameter and replace verification parameter with a "verification key" and "Aggregator ID".

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.

Algorithms in this document are written in Python 3. Type hints are used to define input and output types. A fatal error in a program (e.g., failure to parse one of the function parameters) is usually handled by raising an exception.

A variable with type Bytes is a byte string. This document defines several byte-string constants. When comprised of printable ASCII characters, they are written as Python 3 byte-string literals (e.g., b'some constant string').

A global constant VERSION of type Unsigned is defined, which algorithms are free to use as desired. Its value **SHALL** be 4.

This document describes algorithms for multi-party computations in which the parties typically communicate over a network. Wherever a quantity is defined that must be be transmitted from one party to another, this document prescribes a particular encoding of that quantity as a byte string.

> OPEN ISSUE It might be better to not be prescriptive about how quantities are encoded on the wire. See issue #58.

Some common functionalities:

* zeros(len: Unsigned) -> Bytes returns an array of zero bytes. The length of output **MUST** be len.

* gen_rand(len: Unsigned) -> Bytes returns an array of random bytes. The length of output **MUST** be len.

* byte(int: Unsigned) -> Bytes returns the representation of int as a byte string. The value of int **MUST** be in [0,256).

* concat(parts: Vec[Bytes]) -> Bytes returns the concatenation of the input byte strings, i.e., parts[0] || ... || parts[len(parts)-1].

*xor(left: Bytes, right: Bytes) -> Bytes returns the bitwise XOR of left and right. An exception is raised if the inputs are not the same length.

*I2OSP and OS2IP from [RFC8017], which are used, respectively, to convert a non-negative integer to a byte string and convert a byte string to a non-negative integer.

*next_power_of_2(n: Unsigned) -> Unsigned returns the smallest integer greater than or equal to n that is also a power of two.

3. Overview

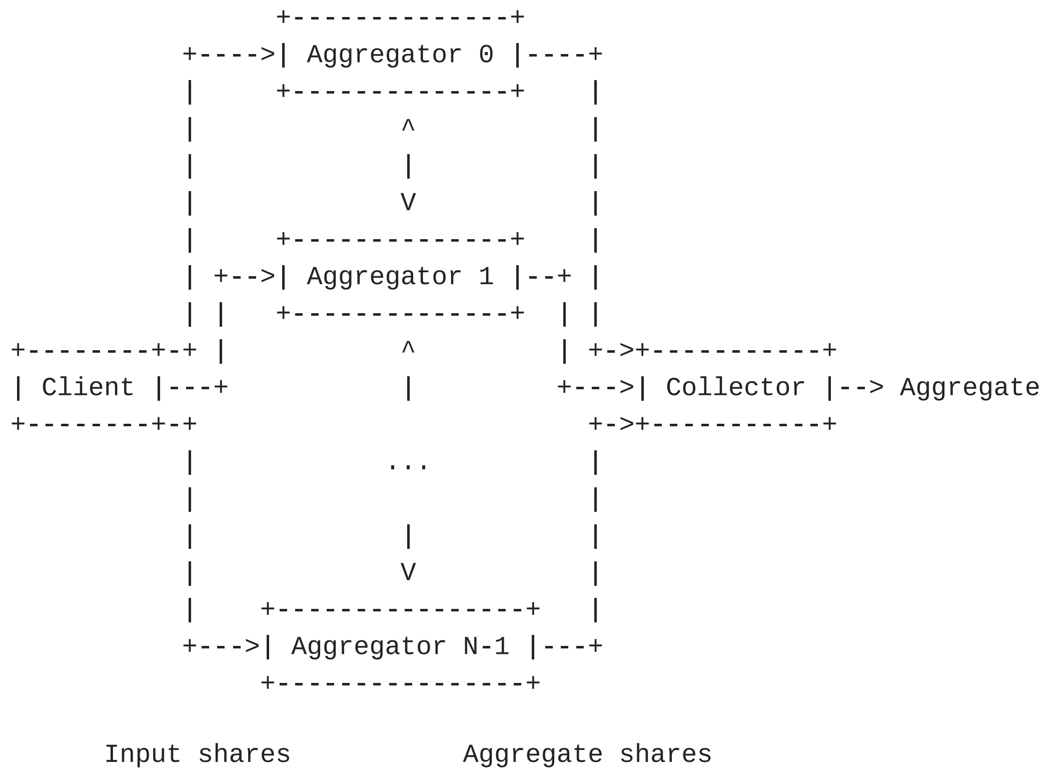

Figure 1: Overall data flow of a (V)DAF

In a DAF- or VDAF-based private measurement system, we distinguish three types of actors: Clients, Aggregators, and Collectors. The overall flow of the measurement process is as follows:

*To submit an individual measurement, the Client shards the measurement into "input shares" and sends one input share to each Aggregator. We sometimes refer to this sequence of input shares collectively as the Client's "report".

* The Aggregators convert their input shares into "output shares".

  - Output shares are in one-to-one correspondence with the input shares.

  - Just as each Aggregator receives one input share of each input, at the end of this process, each aggregator holds one output share.

  - In VDAFs, Aggregators will need to exchange information among themselves as part of the validation process.

* Each Aggregator combines the output shares across inputs in the batch to compute the "aggregate share" for that batch, i.e., its share of the desired aggregate result.

* The Aggregators submit their aggregate shares to the Collector, who combines them to obtain the aggregate result over the batch.

Aggregators are a new class of actor relative to traditional measurement systems where clients submit measurements to a single server. They are critical for both the privacy properties of the system and, in the case of VDAFs, the correctness of the measurements obtained. The privacy properties of the system are assured by non-collusion among Aggregators, and Aggregators are the entities that perform validation of Client measurements. Thus clients trust Aggregators not to collude (typically it is required that at least one Aggregator is honest), and Collectors trust Aggregators to correctly run the protocol.

Within the bounds of the non-collusion requirements of a given (V)DAF instance, it is possible for the same entity to play more than one role. For example, the Collector could also act as an Aggregator, effectively using the other Aggregator(s) to augment a basic client-server protocol.

In this document, we describe the computations performed by the actors in this system. It is up to the higher-level protocol making use of the (V)DAF to arrange for the required information to be delivered to the proper actors in the proper sequence. In general, we assume that all communications are confidential and mutually authenticated, with the exception that Clients submitting measurements may be anonymous.

4. Definition of DAFs

By way of a gentle introduction to VDAFs, this section describes a simpler class of schemes called Distributed Aggregation Functions (DAFs). Unlike VDAFs, DAFs do not provide verifiability of the computation. Clients must therefore be trusted to compute their

input shares correctly. Because of this fact, the use of a DAF is **NOT RECOMMENDED** for most applications. See Section 9 for additional discussion.

A DAF scheme is used to compute a particular "aggregation function" over a set of measurements generated by Clients. Depending on the aggregation function, the Collector might select an "aggregation parameter" and disseminates it to the Aggregators. The semantics of this parameter is specific to the aggregation function, but in general it is used to represent the set of "queries" that can be made on the measurement set. For example, the aggregation parameter is used to represent the candidate prefixes in Poplar1 Section 8.

Execution of a DAF has four distinct stages:

* Sharding - Each Client generates input shares from its measurement and distributes them among the Aggregators.

* Preparation - Each Aggregator converts each input share into an output share compatible with the aggregation function. This computation involves the aggregation parameter. In general, each aggregation parameter may result in a different an output share.

* Aggregation - Each Aggregator combines a sequence of output shares into its aggregate share and sends the aggregate share to the Collector.

* Unsharding - The Collector combines the aggregate shares into the aggregate result.

Sharding and Preparation are done once per measurement. Aggregation and Unsharding are done over a batch of measurements (more precisely, over the recovered output shares).

A concrete DAF specifies an algorithm for the computation needed in each of these stages. The interface of each algorithm is defined in the remainder of this section. In addition, a concrete DAF defines the associated constants and types enumerated in the following table.

Parameter	**Description**
ID	Algorithm identifier for this DAF.
SHARES	Number of input shares into which each measurement is sharded
Measurement	Type of each measurement
AggParam	Type of aggregation parameter
OutShare	Type of each output share
AggResult	Type of the aggregate result

Table 1: Constants and types defined by each concrete DAF.

These types define some of the inputs and outputs of DAF methods at various stages of the computation. Observe that only the measurements, output shares, the aggregate result, and the aggregation parameter have an explicit type. All other values --- in particular, the input shares and the aggregate shares --- have type Bytes and are treated as opaque byte strings. This is because these values must be transmitted between parties over a network.

> OPEN ISSUE It might be cleaner to define a type for each value, then have that type implement an encoding where necessary. This way each method parameter has a meaningful type hint. See issue#58.

Each DAF is identified by a unique, 32-bit integer ID. Identifiers for each (V)DAF specified in this document are defined in Table 18.

4.1. Sharding

In order to protect the privacy of its measurements, a DAF Client shards its measurements into a sequence of input shares. The measurement_to_input_shares method is used for this purpose.

* Daf.measurement_to_input_shares(input: Measurement) -> (Bytes, Vec[Bytes]) is the randomized input-distribution algorithm run by each Client. It consumes the measurement and produces a "public share", distributed to each of the Aggregators, and a corresponding sequence of input shares, one for each Aggregator. The length of the output vector **MUST** be SHARES.

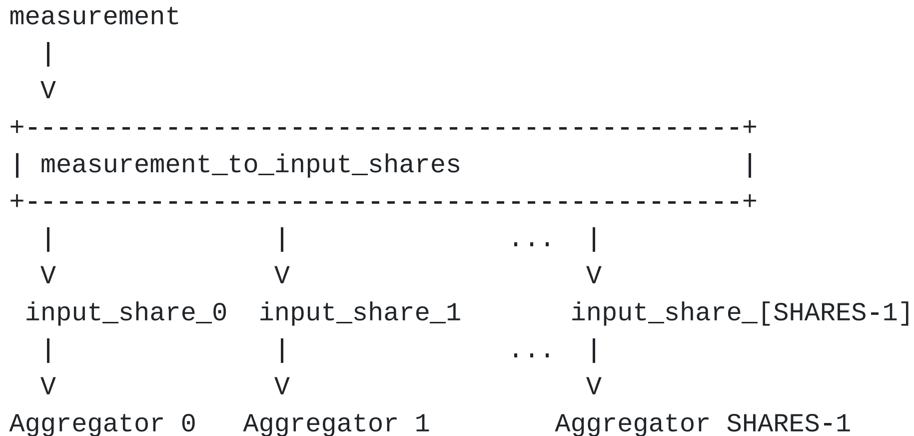

Figure 2: The Client divides its measurement into input shares and distributes them to the Aggregators.

4.2. Preparation

Once an Aggregator has received the public share and one of the input shares, the next step is to prepare the input share for aggregation. This is accomplished using the following algorithm:

* Daf.prep(agg_id: Unsigned, agg_param: AggParam, public_share: Bytes, input_share: Bytes) -> OutShare is the deterministic preparation algorithm. It takes as input the public share and one of the input shares generated by a Client, the Aggregator's unique identifier, and the aggregation parameter selected by the Collector and returns an output share.

The protocol in which the DAF is used **MUST** ensure that the Aggregator's identifier is equal to the integer in range [0, SHARES) that matches the index of input_share in the sequence of input shares output by the Client.

4.3. Validity of Aggregation Parameters

Concrete DAFs implementations **MAY** impose certain restrictions for input shares and aggregation parameters. Protocols using a DAF **MUST** ensure that for each input share and aggregation parameter agg_param, Daf.prep is only called if Daf.is_valid(agg_param, previous_agg_params) returns True, where previous_agg_params contains all aggregation parameters that have previously been used with the same input share.

DAFs **MUST** implement the following function:

* Daf.is_valid(agg_param: AggParam, previous_agg_params: Vec[AggParam]) -> Bool: Checks if the agg_param is compatible with all elements of previous_agg_params.

4.4. Aggregation

Once an Aggregator holds output shares for a batch of measurements (where batches are defined by the application), it combines them into a share of the desired aggregate result:

* Daf.out_shares_to_agg_share(agg_param: AggParam, out_shares: Vec[OutShare]) -> agg_share: Bytes is the deterministic aggregation algorithm. It is run by each Aggregator a set of recovered output shares.

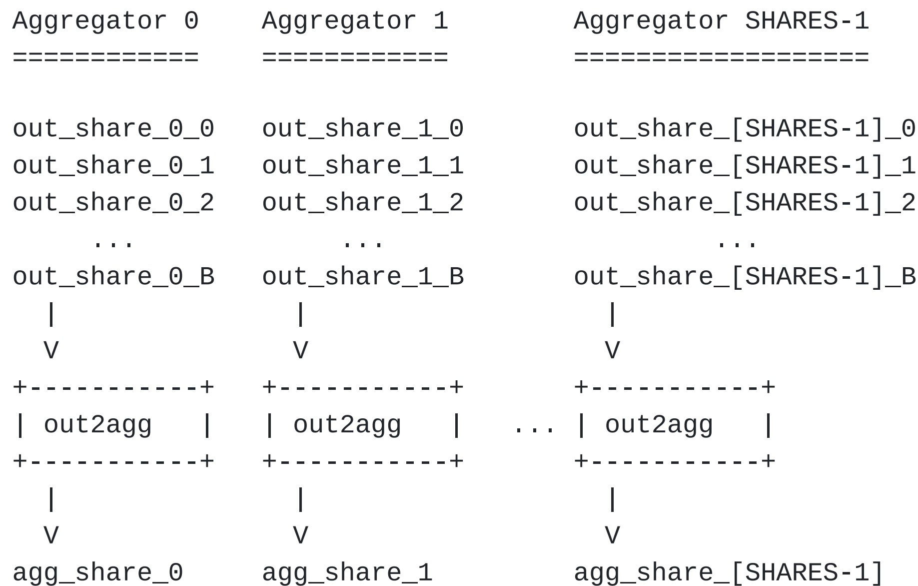

Figure 3: Aggregation of output shares. `B` indicates the number of measurements in the batch.

For simplicity, we have written this algorithm in a "one-shot" form, where all output shares for a batch are provided at the same time. Many DAFs may also support a "streaming" form, where shares are processed one at a time.

OPEN ISSUE It may be worthwhile to explicitly define the "streaming" API. See issue#47.

4.5. Unsharding

After the Aggregators have aggregated a sufficient number of output shares, each sends its aggregate share to the Collector, who runs the following algorithm to recover the following output:

* Daf.agg_shares_to_result(agg_param: AggParam, agg_shares: Vec[Bytes], num_measurements: Unsigned) -> AggResult is run by the Collector in order to compute the aggregate result from the Aggregators' shares. The length of agg_shares **MUST** be SHARES. num_measurements is the number of measurements that contributed to each of the aggregate shares. This algorithm is deterministic.

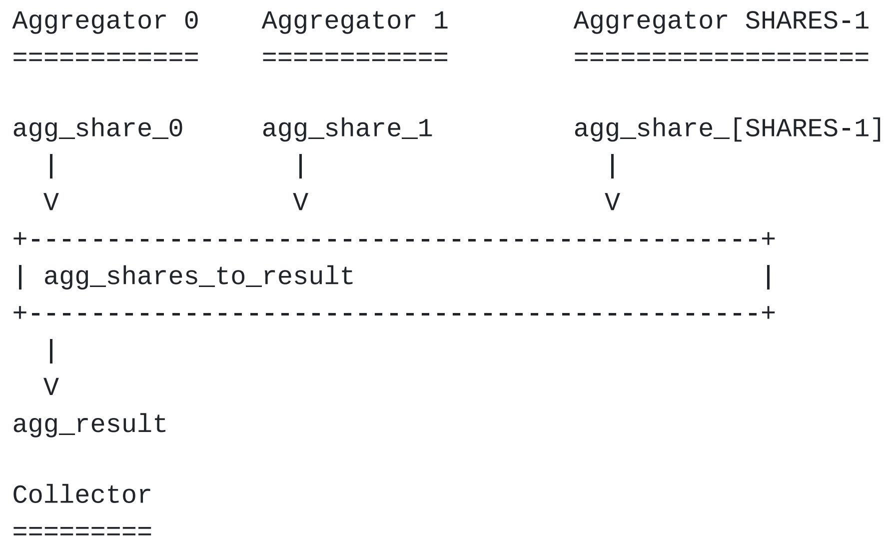

Figure 4: Computation of the final aggregate result from aggregate shares.

QUESTION Maybe the aggregation algorithms should be randomized in order to allow the Aggregators (or the Collector) to add noise for differential privacy. (See the security considerations of [DAP].) Or is this out-of-scope of this document? See https://github.com/ietf-wg-ppm/ppm-specification/issues/19.

4.6. Execution of a DAF

Securely executing a DAF involves emulating the following procedure.

```
def run_daf(Daf,
            agg_param: Daf.AggParam,
            measurements: Vec[Daf.Measurement]):
    out_shares = [ [] for j in range(Daf.SHARES) ]
    for measurement in measurements:
        # Each Client shards its measurement into input shares and
        # distributes them among the Aggregators.
        (public_share, input_shares) = \
            Daf.measurement_to_input_shares(measurement)

        # Each Aggregator prepares its input share for aggregation.
        for j in range(Daf.SHARES):
            out_shares[j].append(
                Daf.prep(j, agg_param, public_share, input_shares[j]))

    # Each Aggregator aggregates its output shares into an aggregate
    # share and sends it to the Collector.
    agg_shares = []
    for j in range(Daf.SHARES):
        agg_share_j = Daf.out_shares_to_agg_share(agg_param,
                                                  out_shares[j])
        agg_shares.append(agg_share_j)

    # Collector unshards the aggregate result.
    num_measurements = len(measurements)
    agg_result = Daf.agg_shares_to_result(agg_param, agg_shares,
                                          num_measurements)
    return agg_result
```

Figure 5: Execution of a DAF.

The inputs to this procedure are the same as the aggregation function computed by the DAF: An aggregation parameter and a sequence of measurements. The procedure prescribes how a DAF is executed in a "benign" environment in which there is no adversary and the messages are passed among the protocol participants over secure point-to-point channels. In reality, these channels need to be instantiated by some "wrapper protocol", such as [DAP], that realizes these channels using suitable cryptographic mechanisms. Moreover, some fraction of the Aggregators (or Clients) may be malicious and diverge from their prescribed behaviors. Section 9 describes the execution of the DAF in various adversarial environments and what properties the wrapper protocol needs to provide in each.

5. Definition of VDAFs

Like DAFs described in the previous section, a VDAF scheme is used to compute a particular aggregation function over a set of Client-

generated measurements. Evaluation of a VDAF involves the same four stages as for DAFs: Sharding, Preparation, Aggregation, and Unsharding. However, the Preparation stage will require interaction among the Aggregators in order to facilitate verifiability of the computation's correctness. Accommodating this interaction will require syntactic changes.

Overall execution of a VDAF comprises the following stages:

* Sharding - Computing input shares from an individual measurement

* Preparation - Conversion and verification of input shares to output shares compatible with the aggregation function being computed

* Aggregation - Combining a sequence of output shares into an aggregate share

* Unsharding - Combining a sequence of aggregate shares into an aggregate result

In contrast to DAFs, the Preparation stage for VDAFs now performs an additional task: Verification of the validity of the recovered output shares. This process ensures that aggregating the output shares will not lead to a garbled aggregate result.

The remainder of this section defines the VDAF interface. The attributes are listed in Table 2 are defined by each concrete VDAF.

Parameter	**Description**
ID	Algorithm identifier for this VDAF
VERIFY_KEY_SIZE	Size (in bytes) of the verification key (Section 5.2)
NONCE_SIZE `	Size (in bytes) of the nonce
ROUNDS	Number of rounds of communication during the Preparation stage (Section 5.2)
SHARES	Number of input shares into which each measurement is sharded (Section 5.1)
Measurement	Type of each measurement
AggParam	Type of aggregation parameter
Prep	State of each Aggregator during Preparation (Section 5.2)
OutShare	Type of each output share
AggResult	Type of the aggregate result

Table 2: Constants and types defined by each concrete VDAF.

Similarly to DAFs (see {[sec-daf}}), any output of a VDAF method that must be transmitted from one party to another is treated as an opaque byte string. All other quantities are given a concrete type.

> OPEN ISSUE It might be cleaner to define a type for each value, then have that type implement an encoding where necessary. See issue#58.

Each VDAF is identified by a unique, 32-bit integer ID. Identifiers for each (V)DAF specified in this document are defined in Table 18. The following method is defined for every VDAF:

```
def custom(Vdaf, usage: Unsigned) -> Bytes:
    return format_custom(0, Vdaf.ID, usage)
```

It is used to construct a customization string for an instance of Prg used by the VDAF. (See Section 6.2.)

5.1. Sharding

Sharding transforms a measurement into input shares as it does in DAFs (cf. Section 4.1); in addition, it takes a nonce as input and produces a public share:

* Vdaf.measurement_to_input_shares(measurement: Measurement, nonce: Bytes[Vdaf.NONCE_SIZE]) -> (Bytes, Vec[Bytes]) is the randomized input-distribution algorithm run by each Client. It consumes the measurement and the nonce and produces a public share, distributed to each of Aggregators, and the corresponding sequence of input shares, one for each Aggregator. Depending on the VDAF, the input shares may encode additional information used to verify the recovered output shares (e.g., the "proof shares" in Prio3 Section 7). The length of the output vector **MUST** be SHARES.

In order to ensure privacy of the measurement, the Client **MUST** generate the nonce using a cryptographically secure pseudorandom number generator (CSPRNG). (See Section 9 for details.)

5.2. Preparation

To recover and verify output shares, the Aggregators interact with one another over ROUNDS rounds. Prior to each round, each Aggregator constructs an outbound message. Next, the sequence of outbound messages is combined into a single message, called a "preparation message". (Each of the outbound messages are called "preparation-message shares".) Finally, the preparation message is distributed to the Aggregators to begin the next round.

An Aggregator begins the first round with its input share and it begins each subsequent round with the previous preparation message. Its output in the last round is its output share and its output in each of the preceding rounds is a preparation-message share.

This process involves a value called the "aggregation parameter" used to map the input shares to output shares. The Aggregators need to agree on this parameter before they can begin preparing inputs for aggregation.

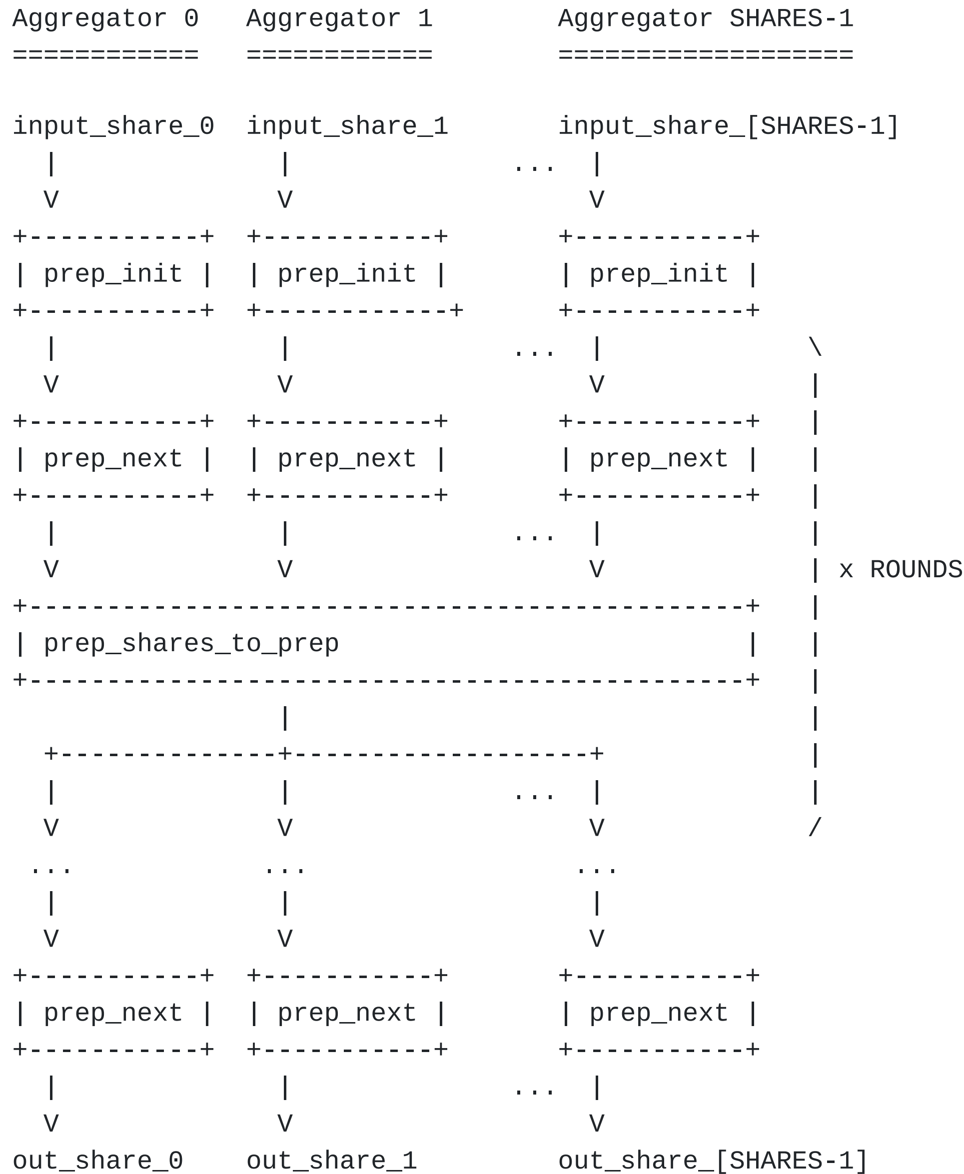

Figure 6: VDAF preparation process on the input shares for a single measurement. At the end of the computation, each Aggregator holds an output share or an error.

To facilitate the preparation process, a concrete VDAF implements the following class methods:

* Vdaf.prep_init(verify_key: Bytes[Vdaf.VERIFY_KEY_SIZE], agg_id: Unsigned, agg_param: AggParam, nonce: Bytes[Vdaf.NONCE_SIZE], public_share: Bytes, input_share: Bytes) -> Prep is the deterministic preparation-state initialization algorithm run by each Aggregator to begin processing its input share into an output share. Its inputs are the shared verification key (verify_key), the Aggregator's unique identifier (agg_id), the aggregation parameter (agg_param), the nonce provided by the environment (nonce, see Figure 7), the public share (public_share), and one of the input shares generated by the client (input_share). Its output is the Aggregator's initial preparation state.

  It is up to the high level protocol in which the VDAF is used to arrange for the distribution of the verification key prior to generating and processing reports. (See Section 9 for details.)

  Protocols using the VDAF **MUST** ensure that the Aggregator's identifier is equal to the integer in range [0, SHARES) that matches the index of input_share in the sequence of input shares output by the Client.

  Protocols **MUST** ensure that public share consumed by each of the Aggregators is identical. This is security critical for VDAFs such as Poplar1 that require an extractable distributed point function. (See Section 8 for details.)

* Vdaf.prep_next(prep: Prep, inbound: Optional[Bytes]) -> Union[Tuple[Prep, Bytes], OutShare] is the deterministic preparation-state update algorithm run by each Aggregator. It updates the Aggregator's preparation state (prep) and returns either its next preparation state and its message share for the current round or, if this is the last round, its output share. An exception is raised if a valid output share could not be recovered. The input of this algorithm is the inbound preparation message or, if this is the first round, None.

* Vdaf.prep_shares_to_prep(agg_param: AggParam, prep_shares: Vec[Bytes]) -> Bytes is the deterministic preparation-message pre-processing algorithm. It combines the preparation-message shares generated by the Aggregators in the previous round into the preparation message consumed by each in the next round.

In effect, each Aggregator moves through a linear state machine with ROUNDS+1 states. The Aggregator enters the first state on using the initialization algorithm, and the update algorithm advances the

Aggregator to the next state. Thus, in addition to defining the number of rounds (ROUNDS), a VDAF instance defines the state of the Aggregator after each round.

TODO Consider how to bake this "linear state machine" condition into the syntax. Given that Python 3 is used as our pseudocode, it's easier to specify the preparation state using a class.

The preparation-state update accomplishes two tasks: recovery of output shares from the input shares and ensuring that the recovered output shares are valid. The abstraction boundary is drawn so that an Aggregator only recovers an output share if it is deemed valid (at least, based on the Aggregator's view of the protocol). Another way to draw this boundary would be to have the Aggregators recover output shares first, then verify that they are valid. However, this would allow the possibility of misusing the API by, say, aggregating an invalid output share. Moreover, in protocols like Prio+ [AGJOP21] based on oblivious transfer, it is necessary for the Aggregators to interact in order to recover aggregatable output shares at all.

Note that it is possible for a VDAF to specify ROUNDS == 0, in which case each Aggregator runs the preparation-state update algorithm once and immediately recovers its output share without interacting with the other Aggregators. However, most, if not all, constructions will require some amount of interaction in order to ensure validity of the output shares (while also maintaining privacy).

OPEN ISSUE accommodating 0-round VDAFs may require syntax changes if, for example, public keys are required. On the other hand, we could consider defining this class of schemes as a different primitive. See issue#77.

5.3. Validity of Aggregation Parameters

Similar to DAFs (see Section 4.3), VDAFs **MAY** impose restrictions for input shares and aggregation parameters. Protocols using a VDAF **MUST** ensure that for each input share and aggregation parameter agg_param, the preparation phase (including Vdaf.prep_init, Vdaf.prep_next, and Vdaf.prep_shares_to_prep; see Section 5.2) is only called if Vdaf.is_valid(agg_param, previous_agg_params) returns True, where previous_agg_params contains all aggregation parameters that have previously been used with the same input share.

VDAFs **MUST** implement the following function:

* Vdaf.is_valid(agg_param: AggParam, previous_agg_params: Vec[AggParam]) -> Bool: Checks if the agg_param is compatible with all elements of previous_agg_params.

5.4. Aggregation

VDAF Aggregation is identical to DAF Aggregation (cf. Section 4.4):

* Vdaf.out_shares_to_agg_share(agg_param: AggParam, out_shares: Vec[OutShare]) -> agg_share: Bytes is the deterministic aggregation algorithm. It is run by each Aggregator over the output shares it has computed over a batch of measurement inputs.

The data flow for this stage is illustrated in Figure 3. Here again, we have the aggregation algorithm in a "one-shot" form, where all shares for a batch are provided at the same time. VDAFs typically also support a "streaming" form, where shares are processed one at a time.

5.5. Unsharding

VDAF Unsharding is identical to DAF Unsharding (cf. Section 4.5):

* Vdaf.agg_shares_to_result(agg_param: AggParam, agg_shares: Vec[Bytes], num_measurements: Unsigned) -> AggResult is run by the Collector in order to compute the aggregate result from the Aggregators' shares. The length of agg_shares **MUST** be SHARES. num_measurements is the number of measurements that contributed to each of the aggregate shares. This algorithm is deterministic.

The data flow for this stage is illustrated in Figure 4.

5.6. Execution of a VDAF

Secure execution of a VDAF involves simulating the following procedure.

```
def run_vdaf(Vdaf,
             verify_key: Bytes[Vdaf.VERIFY_KEY_SIZE],
             agg_param: Vdaf.AggParam,
             nonces: Vec[Bytes[Vdaf.NONCE_SIZE]],
             measurements: Vec[Vdaf.Measurement]):
    out_shares = []
    for (nonce, measurement) in zip(nonces, measurements):
        # Each Client shards its measurement into input shares.
        (public_share, input_shares) = \
            Vdaf.measurement_to_input_shares(measurement, nonce)

        # Each Aggregator initializes its preparation state.
        prep_states = []
        for j in range(Vdaf.SHARES):
            state = Vdaf.prep_init(verify_key, j,
                                   agg_param,
                                   nonce,
                                   public_share,
                                   input_shares[j])
            prep_states.append(state)

        # Aggregators recover their output shares.
        inbound = None
        for i in range(Vdaf.ROUNDS+1):
            outbound = []
            for j in range(Vdaf.SHARES):
                out = Vdaf.prep_next(prep_states[j], inbound)
                if i < Vdaf.ROUNDS:
                    (prep_states[j], out) = out
                outbound.append(out)
            # This is where we would send messages over the
            # network in a distributed VDAF computation.
            if i < Vdaf.ROUNDS:
                inbound = Vdaf.prep_shares_to_prep(agg_param,
                                                   outbound)

        # The final outputs of prepare phase are the output shares.
        out_shares.append(outbound)

    # Each Aggregator aggregates its output shares into an
    # aggregate share. In a distributed VDAF computation, the
    # aggregate shares are sent over the network.
    agg_shares = []
    for j in range(Vdaf.SHARES):
        out_shares_j = [out[j] for out in out_shares]
        agg_share_j = Vdaf.out_shares_to_agg_share(agg_param,
                                                   out_shares_j)
        agg_shares.append(agg_share_j)
```

```
# Collector unshards the aggregate.
num_measurements = len(measurements)
agg_result = Vdaf.agg_shares_to_result(agg_param, agg_shares,
                                       num_measurements)
return agg_result
```

Figure 7: Execution of a VDAF.

The inputs to this algorithm are the aggregation parameter, a list of measurements, and a nonce for each measurement. This document does not specify how the nonces are chosen, but security requires that the nonces be unique. See Section 9 for details. As explained in Section 4.6, the secure execution of a VDAF requires the application to instantiate secure channels between each of the protocol participants.

6. Preliminaries

This section describes the primitives that are common to the VDAFs specified in this document.

6.1. Finite Fields

Both Prio3 and Poplar1 use finite fields of prime order. Finite field elements are represented by a class Field with the following associated parameters:

* MODULUS: Unsigned is the prime modulus that defines the field.

* ENCODED_SIZE: Unsigned is the number of bytes used to encode a field element as a byte string.

A concrete Field also implements the following class methods:

* Field.zeros(length: Unsigned) -> output: Vec[Field] returns a vector of zeros. The length of output **MUST** be length.

* Field.rand_vec(length: Unsigned) -> output: Vec[Field] returns a vector of random field elements. The length of output **MUST** be length.

A field element is an instance of a concrete Field. The concrete class defines the usual arithmetic operations on field elements. In addition, it defines the following instance method for converting a field element to an unsigned integer:

* elem.as_unsigned() -> Unsigned returns the integer representation of field element elem.

Likewise, each concrete Field implements a constructor for converting an unsigned integer into a field element:

* Field(integer: Unsigned) returns integer represented as a field element. The value of integer **MUST** be less than Field.MODULUS.

Finally, each concrete Field has two derived class methods, one for encoding a vector of field elements as a byte string and another for decoding a vector of field elements.

```
def encode_vec(Field, data: Vec[Field]) -> Bytes:
    encoded = Bytes()
    for x in data:
        encoded += I2OSP(x.as_unsigned(), Field.ENCODED_SIZE)
    return encoded

def decode_vec(Field, encoded: Bytes) -> Vec[Field]:
    L = Field.ENCODED_SIZE
    if len(encoded) % L != 0:
        raise ERR_DECODE

    vec = []
    for i in range(0, len(encoded), L):
        encoded_x = encoded[i:i+L]
        x = OS2IP(encoded_x)
        if x >= Field.MODULUS:
            raise ERR_DECODE # Integer is larger than modulus
        vec.append(Field(x))
    return vec
```

Figure 8: Derived class methods for finite fields.

6.1.1. Auxiliary Functions

The following auxiliary functions on vectors of field elements are used in the remainder of this document. Note that an exception is raised by each function if the operands are not the same length.

```
# Compute the inner product of the operands.
def inner_product(left: Vec[Field], right: Vec[Field]) -> Field:
    return sum(map(lambda x: x[0] * x[1], zip(left, right)))

# Subtract the right operand from the left and return the result.
def vec_sub(left: Vec[Field], right: Vec[Field]):
    return list(map(lambda x: x[0] - x[1], zip(left, right)))

# Add the right operand to the left and return the result.
def vec_add(left: Vec[Field], right: Vec[Field]):
    return list(map(lambda x: x[0] + x[1], zip(left, right)))
```

Figure 9: Common functions for finite fields.

6.1.2. FFT-Friendly Fields

Some VDAFs require fields that are suitable for efficient computation of the discrete Fourier transform, as this allows for fast polynomial interpolation. (One example is Prio3 (Section 7) when instantiated with the generic FLP of Section 7.3.3.) Specifically, a field is said to be "FFT-friendly" if, in addition to satisfying the interface described in Section 6.1, it implements the following method:

* Field.gen() -> Field returns the generator of a large subgroup of the multiplicative group. To be FFT-friendly, the order of this subgroup **MUST** be a power of 2. In addition, the size of the subgroup dictates how large interpolated polynomials can be. It is **RECOMMENDED** that a generator is chosen with order at least 2^20.

FFT-friendly fields also define the following parameter:

* GEN_ORDER: Unsigned is the order of a multiplicative subgroup generated by Field.gen().

6.1.3. Parameters

The tables below define finite fields used in the remainder of this document.

Parameter	Value
MODULUS	2^32 * 4294967295 + 1
ENCODED_SIZE	8
Generator	7^4294967295
GEN_ORDER	2^32

Table 3: Field64, an FFT-friendly field.

Parameter	Value
MODULUS	2^66 * 4611686018427387897 + 1
ENCODED_SIZE	16
Generator	7^4611686018427387897
GEN_ORDER	2^66

Table 4: Field128, an FFT-friendly field.

Parameter	Value
MODULUS	2^255 - 19
ENCODED_SIZE	32

Table 5: Field255.

OPEN ISSUE We currently use big-endian for encoding field elements. However, for implementations of GF(2^255-19), little endian is more common. See issue#90.

6.2. Pseudorandom Generators

A pseudorandom generator (PRG) is used to expand a short, (pseudo)random seed into a long string of pseudorandom bits. A PRG suitable for this document implements the interface specified in this section.

PRGs are defined by a class Prg with the following associated parameter:

* SEED_SIZE: Unsigned is the size (in bytes) of a seed.

A concrete Prg implements the following class method:

* Prg(seed: Bytes[Prg.SEED_SIZE], custom: Bytes, binder: Bytes) constructs an instance of Prg from the given seed and customization and binder strings. (See below for definitions of these.) The seed **MUST** be of length SEED_SIZE and **MUST** be generated securely (i.e., it is either the output of gen_rand or a previous invocation of the PRG).

* prg.next(length: Unsigned) returns the next length bytes of output of PRG. If the seed was securely generated, the output can be treated as pseudorandom.

Each Prg has two derived class methods. The first is used to derive a fresh seed from an existing one. The second is used to compute a sequence of pseudorandom field elements. For each method, the seed **MUST** be of length SEED_SIZE and **MUST** be generated securely (i.e., it is either the output of gen_rand or a previous invocation of the PRG).

```
# Derive a new seed.
def derive_seed(Prg, seed: Bytes[Prg.SEED_SIZE], custom: Bytes, binder:
    prg = Prg(seed, custom, binder)
    return prg.next(Prg.SEED_SIZE)

# Output the next `length` pseudorandom elements of `Field`.
def next_vec(self, Field, length: Unsigned):
    m = next_power_of_2(Field.MODULUS) - 1
    vec = []
    while len(vec) < length:
        x = OS2IP(self.next(Field.ENCODED_SIZE))
        x &= m
        if x < Field.MODULUS:
            vec.append(Field(x))
    return vec

# Expand the input `seed` into vector of `length` field elements.
def expand_into_vec(Prg,
                    Field,
                    seed: Bytes[Prg.SEED_SIZE],
                    custom: Bytes,
                    binder: Bytes,
                    length: Unsigned):
    prg = Prg(seed, custom, binder)
    return prg.next_vec(Field, length)
```

Figure 10: Derived class methods for PRGs.

6.2.1. PrgSha3

This section describes PrgSha3, a PRG based on the Keccak permutation of SHA-3 [FIPS202]. Keccak is used in the cSHAKE128 mode of operation [SP800-185].

```
class PrgSha3(Prg):
    # Associated parameters
    SEED_SIZE = 16

    def __init__(self, seed, custom, binder):
        self.l = 0
        self.x = seed + binder
        self.s = custom

    def next(self, length: Unsigned) -> Bytes:
        self.l += length

        # Function `cSHAKE128(x, l, n, s)` is as defined in
        # [SP800-185, Section 3.3].
        #
        # Implementation note: Rather than re-generate the output
        # stream each time `next()` is invoked, most implementations
        # of SHA-3 will expose an "absorb-then-squeeze" API that
        # allows stateful handling of the stream.
        stream = cSHAKE128(self.x, self.l, b'', self.s)
        return stream[-length:]
```

Figure 11: Definition of PRG PrgSha3.

6.2.2. The Customization and Binder Strings

PRGs are used to map a seed to a finite domain, e.g., a fresh seed or a vector of field elements. To ensure domain separation, the derivation is needs to be bound to some distinguished "customization string". The customization string encodes the following values:

1. The document version (i.e.,VERSION);

2. The "class" of the algorithm using the output (e.g., VDAF);

3. A unique identifier for the algorithm; and

4. Some indication of how the output is used (e.g., for deriving the measurement shares in Prio3 Section 7).

The following algorithm is used in the remainder of this document in order to format the customization string:

```
def format_custom(algo_class: Unsigned, algo: Unsigned, usage: Unsigned)
    return I2OSP(VERSION, 1) + \
           I2OSP(algo_class, 1) + \
           I2OSP(algo, 4) + \
           I2OSP(usage, 2)
```

It is also sometimes necessary to bind the output to some ephemeral value that multiple parties need to agree on. We call this input the "binder string".

7. Prio3

This section describes Prio3, a VDAF for Prio [CGB17]. Prio is suitable for a wide variety of aggregation functions, including (but not limited to) sum, mean, standard deviation, estimation of quantiles (e.g., median), and linear regression. In fact, the scheme described in this section is compatible with any aggregation function that has the following structure:

* Each measurement is encoded as a vector over some finite field.

* Input validity is determined by an arithmetic circuit evaluated over the encoded input. (An "arithmetic circuit" is a function comprised of arithmetic operations in the field.) The circuit's output is a single field element: if zero, then the input is said to be "valid"; otherwise, if the output is non-zero, then the input is said to be "invalid".

* The aggregate result is obtained by summing up the encoded input vectors and computing some function of the sum.

At a high level, Prio3 distributes this computation as follows. Each Client first shards its measurement by first encoding it, then splitting the vector into secret shares and sending a share to each Aggregator. Next, in the preparation phase, the Aggregators carry out a multi-party computation to determine if their shares correspond to a valid input (as determined by the arithmetic circuit). This computation involves a "proof" of validity generated by the Client. Next, each Aggregator sums up its input shares locally. Finally, the Collector sums up the aggregate shares and computes the aggregate result.

This VDAF does not have an aggregation parameter. Instead, the output share is derived from the input share by applying a fixed map. See Section 8 for an example of a VDAF that makes meaningful use of the aggregation parameter.

As the name implies, Prio3 is a descendant of the original Prio construction. A second iteration was deployed in the [ENPA] system, and like the VDAF described here, the ENPA system was built from techniques introduced in [BBCGGI19] that significantly improve communication cost. That system was specialized for a particular aggregation function; the goal of Prio3 is to provide the same level of generality as the original construction.

The core component of Prio3 is a "Fully Linear Proof (FLP)" system. Introduced by [BBCGGI19], the FLP encapsulates the functionality required for encoding and validating inputs. Prio3 can be thought of as a transformation of a particular class of FLPs into a VDAF.

The remainder of this section is structured as follows. The syntax for FLPs is described in Section 7.1. The generic transformation of an FLP into Prio3 is specified in Section 7.2. Next, a concrete FLP suitable for any validity circuit is specified in Section 7.3. Finally, instantiations of Prio3 for various types of measurements are specified in Section 7.4. Test vectors can be found in Appendix "Test Vectors".

7.1. Fully Linear Proof (FLP) Systems

Conceptually, an FLP is a two-party protocol executed by a prover and a verifier. In actual use, however, the prover's computation is carried out by the Client, and the verifier's computation is distributed among the Aggregators. The Client generates a "proof" of its input's validity and distributes shares of the proof to the Aggregators. Each Aggregator then performs some a computation on its input share and proof share locally and sends the result to the other Aggregators. Combining the exchanged messages allows each Aggregator to decide if it holds a share of a valid input. (See Section 7.2 for details.)

As usual, we will describe the interface implemented by a concrete FLP in terms of an abstract base class Flp that specifies the set of methods and parameters a concrete FLP must provide.

The parameters provided by a concrete FLP are listed in Table 6.

Parameter	Description
PROVE_RAND_LEN	Length of the prover randomness, the number of random field elements consumed by the prover when generating a proof
QUERY_RAND_LEN	Length of the query randomness, the number of random field elements consumed by the verifier
JOINT_RAND_LEN	Length of the joint randomness, the number of random field elements consumed by both the prover and verifier
INPUT_LEN	Length of the encoded measurement (Section 7.1.1)
OUTPUT_LEN	Length of the aggregatable output (Section 7.1.1)
PROOF_LEN	Length of the proof
VERIFIER_LEN	Length of the verifier message generated by querying the input and proof
Measurement	Type of the measurement
AggResult	Type of the aggregate result

Parameter	Description
Field	As defined in (Section 6.1)

Table 6: Constants and types defined by a concrete FLP.

An FLP specifies the following algorithms for generating and verifying proofs of validity (encoding is described below in Section 7.1.1):

* Flp.prove(input: Vec[Field], prove_rand: Vec[Field], joint_rand: Vec[Field]) -> Vec[Field] is the deterministic proof-generation algorithm run by the prover. Its inputs are the encoded input, the "prover randomness" prove_rand, and the "joint randomness" joint_rand. The prover randomness is used only by the prover, but the joint randomness is shared by both the prover and verifier.

* Flp.query(input: Vec[Field], proof: Vec[Field], query_rand: Vec[Field], joint_rand: Vec[Field], num_shares: Unsigned) -> Vec[Field] is the query-generation algorithm run by the verifier. This is used to "query" the input and proof. The result of the query (i.e., the output of this function) is called the "verifier message". In addition to the input and proof, this algorithm takes as input the query randomness query_rand and the joint randomness joint_rand. The former is used only by the verifier. num_shares specifies how many input and proof shares were generated.

* Flp.decide(verifier: Vec[Field]) -> Bool is the deterministic decision algorithm run by the verifier. It takes as input the verifier message and outputs a boolean indicating if the input from which it was generated is valid.

Our application requires that the FLP is "fully linear" in the sense defined in [BBCGGI19]. As a practical matter, what this property implies is that, when run on a share of the input and proof, the query-generation algorithm outputs a share of the verifier message. Furthermore, the "strong zero-knowledge" property of the FLP system ensures that the verifier message reveals nothing about the input's validity. Therefore, to decide if an input is valid, the Aggregators will run the query-generation algorithm locally, exchange verifier shares, combine them to recover the verifier message, and run the decision algorithm.

The query-generation algorithm includes a parameter num_shares that specifies the number of shares of the input and proof that were generated. If these data are not secret shared, then num_shares == 1. This parameter is useful for certain FLP constructions. For example, the FLP in Section 7.3 is defined in terms of an arithmetic circuit; when the circuit contains constants, it is sometimes

necessary to normalize those constants to ensure that the circuit's output, when run on a valid input, is the same regardless of the number of shares.

An FLP is executed by the prover and verifier as follows:

```
def run_flp(Flp, inp: Vec[Flp.Field], num_shares: Unsigned):
    joint_rand = Flp.Field.rand_vec(Flp.JOINT_RAND_LEN)
    prove_rand = Flp.Field.rand_vec(Flp.PROVE_RAND_LEN)
    query_rand = Flp.Field.rand_vec(Flp.QUERY_RAND_LEN)

    # Prover generates the proof.
    proof = Flp.prove(inp, prove_rand, joint_rand)

    # Verifier queries the input and proof.
    verifier = Flp.query(inp, proof, query_rand, joint_rand, num_shares)

    # Verifier decides if the input is valid.
    return Flp.decide(verifier)
```

Figure 12: Execution of an FLP.

The proof system is constructed so that, if input is a valid input, then run_flp(Flp, input, 1) always returns True. On the other hand, if input is invalid, then as long as joint_rand and query_rand are generated uniform randomly, the output is False with overwhelming probability.

We remark that [BBCGGI19] defines a much larger class of fully linear proof systems than we consider here. In particular, what is called an "FLP" here is called a 1.5-round, public-coin, interactive oracle proof system in their paper.

7.1.1. Encoding the Input

The type of measurement being aggregated is defined by the FLP. Hence, the FLP also specifies a method of encoding raw measurements as a vector of field elements:

* Flp.encode(measurement: Measurement) -> Vec[Field] encodes a raw measurement as a vector of field elements. The return value **MUST** be of length INPUT_LEN.

For some FLPs, the encoded input also includes redundant field elements that are useful for checking the proof, but which are not needed after the proof has been checked. An example is the "integer sum" data type from [CGB17] in which an integer in range [0, 2^k) is encoded as a vector of k field elements (this type is also defined in Section 7.4.2). After consuming this vector, all that is needed

is the integer it represents. Thus the FLP defines an algorithm for truncating the input to the length of the aggregated output:

* Flp.truncate(input: Vec[Field]) -> Vec[Field] maps an encoded input to an aggregatable output. The length of the input **MUST** be INPUT_LEN and the length of the output **MUST** be OUTPUT_LEN.

Once the aggregate shares have been computed and combined together, their sum can be converted into the aggregate result. This could be a projection from the FLP's field to the integers, or it could include additional post-processing.

* Flp.decode(output: Vec[Field], num_measurements: Unsigned) -> AggResult maps a sum of aggregate shares to an aggregate result. The length of the input **MUST** be OUTPUT_LEN. num_measurements is the number of measurements that contributed to the aggregated output.

We remark that, taken together, these three functionalities correspond roughly to the notion of "Affine-aggregatable encodings (AFEs)" from [CGB17].

7.2. Construction

This section specifies Prio3, an implementation of the Vdaf interface (Section 5). It has two generic parameters: an Flp (Section 7.1) and a Prg (Section 6.2). The associated constants and types required by the Vdaf interface are defined in Table 7. The methods required for sharding, preparation, aggregation, and unsharding are described in the remaining subsections. These methods refer to constants enumerated in Table 8.

Parameter	Value
VERIFY_KEY_SIZE	Prg.SEED_SIZE
ROUNDS	1
SHARES	in [2, 256)
Measurement	Flp.Measurement
AggParam	None
Prep	Tuple[Vec[Flp.Field], Optional[Bytes], Bytes]
OutShare	Vec[Flp.Field]
AggResult	Flp.AggResult

Table 7: VDAF parameters for Prio3.

Variable	Value
DST_MEASUREMENT_SHARE: Unsigned	1
DST_PROOF_SHARE: Unsigned	2
DST_JOINT_RANDOMNESS: Unsigned	3
DST_PROVE_RANDOMNESS: Unsigned	4

Variable	Value
DST_QUERY_RANDOMNESS: Unsigned	5
DST_JOINT_RAND_SEED: Unsigned	6
DST_JOINT_RAND_PART: Unsigned	7

Table 8: Constants used by Prio3.

7.2.1. Sharding

Recall from Section 7.1 that the FLP syntax calls for "joint randomness" shared by the prover (i.e., the Client) and the verifier (i.e., the Aggregators). VDAFs have no such notion. Instead, the Client derives the joint randomness from its input in a way that allows the Aggregators to reconstruct it from their input shares. (This idea is based on the Fiat-Shamir heuristic and is described in Section 6.2.3 of [BBCGGI19].)

The input-distribution algorithm involves the following steps:

1. Encode the Client's raw measurement as an input for the FLP
2. Shard the input into a sequence of input shares
3. Derive the joint randomness from the input shares and nonce
4. Run the FLP proof-generation algorithm using the derived joint randomness
5. Shard the proof into a sequence of proof shares

The algorithm is specified below. Notice that only one set of input and proof shares (called the "leader" shares below) are vectors of field elements. The other shares (called the "helper" shares) are represented instead by PRG seeds, which are expanded into vectors of field elements.

The definitions of constants and a few auxiliary functions are defined in Section 7.2.6.

```
def measurement_to_input_shares(Prio3, measurement, nonce):
    inp = Prio3.Flp.encode(measurement)

    # Generate measurement shares.
    leader_meas_share = inp
    k_helper_meas_shares = []
    k_helper_blinds = []
    k_joint_rand_parts = []
    for j in range(Prio3.SHARES-1):
        k_blind = gen_rand(Prio3.Prg.SEED_SIZE)
        k_share = gen_rand(Prio3.Prg.SEED_SIZE)
        helper_meas_share = Prio3.Prg.expand_into_vec(
            Prio3.Flp.Field,
            k_share,
            Prio3.custom(DST_MEASUREMENT_SHARE),
            byte(j+1),
            Prio3.Flp.INPUT_LEN
        )
        leader_meas_share = vec_sub(leader_meas_share,
                                    helper_meas_share)
        encoded = Prio3.Flp.Field.encode_vec(helper_meas_share)
        k_joint_rand_part = Prio3.Prg.derive_seed(k_blind,
            Prio3.custom(DST_JOINT_RAND_PART),
            byte(j+1) + nonce + encoded)
        k_helper_meas_shares.append(k_share)
        k_helper_blinds.append(k_blind)
        k_joint_rand_parts.append(k_joint_rand_part)
    k_leader_blind = gen_rand(Prio3.Prg.SEED_SIZE)
    encoded = Prio3.Flp.Field.encode_vec(leader_meas_share)
    k_leader_joint_rand_part = Prio3.Prg.derive_seed(k_leader_blind,
        Prio3.custom(DST_JOINT_RAND_PART),
        byte(0) + nonce + encoded)
    k_joint_rand_parts.insert(0, k_leader_joint_rand_part)

    # Compute joint randomness seed.
    k_joint_rand = Prio3.joint_rand(k_joint_rand_parts)

    # Generate the proof shares.
    prove_rand = Prio3.Prg.expand_into_vec(
        Prio3.Flp.Field,
        gen_rand(Prio3.Prg.SEED_SIZE),
        Prio3.custom(DST_PROVE_RANDOMNESS),
        b'',
        Prio3.Flp.PROVE_RAND_LEN,
    )
    joint_rand = Prio3.Prg.expand_into_vec(
        Prio3.Flp.Field,
        k_joint_rand,
        Prio3.custom(DST_JOINT_RANDOMNESS),
```

```
        b'',
        Prio3.Flp.JOINT_RAND_LEN,
    )
    proof = Prio3.Flp.prove(inp, prove_rand, joint_rand)
    leader_proof_share = proof
    k_helper_proof_shares = []
    for j in range(Prio3.SHARES-1):
        k_share = gen_rand(Prio3.Prg.SEED_SIZE)
        k_helper_proof_shares.append(k_share)
        helper_proof_share = Prio3.Prg.expand_into_vec(
            Prio3.Flp.Field,
            k_share,
            Prio3.custom(DST_PROOF_SHARE),
            byte(j+1),
            Prio3.Flp.PROOF_LEN,
        )
        leader_proof_share = vec_sub(leader_proof_share,
                                     helper_proof_share)

    # Each Aggregator's input share contains its measurement share,
    # proof share, and blind. The public share contains the
    # Aggregators' joint randomness parts.
    input_shares = []
    input_shares.append(Prio3.encode_leader_share(
        leader_meas_share,
        leader_proof_share,
        k_leader_blind,
    ))
    for j in range(Prio3.SHARES-1):
        input_shares.append(Prio3.encode_helper_share(
            k_helper_meas_shares[j],
            k_helper_proof_shares[j],
            k_helper_blinds[j],
        ))
    public_share = Prio3.encode_public_share(k_joint_rand_parts)
    return (public_share, input_shares)
```

Figure 13: Input-distribution algorithm for Prio3.

7.2.2. Preparation

This section describes the process of recovering output shares from the input shares. The high-level idea is that each Aggregator first queries its input and proof share locally, then exchanges its verifier share with the other Aggregators. The verifier shares are then combined into the verifier message, which is used to decide whether to accept.

In addition, the Aggregators must ensure that they have all used the same joint randomness for the query-generation algorithm. The joint randomness is generated by a PRG seed. Each Aggregator derives a "part" of this seed from its input share and the "blind" generated by the client. The seed is derived by hashing together the parts, so before running the query-generation algorithm, it must first gather the parts derived by the other Aggregators.

In order to avoid extra round of communication, the Client sends each Aggregator a "hint" consisting of the other Aggregators' parts of the joint randomness seed. This leaves open the possibility that the Client cheated by, say, forcing the Aggregators to use joint randomness that biases the proof check procedure some way in its favor. To mitigate this, the Aggregators also check that they have all computed the same joint randomness seed before accepting their output shares. To do so, they exchange their parts of the joint randomness along with their verifier shares.

The definitions of constants and a few auxiliary functions are defined in Section 7.2.6.

```
def prep_init(Prio3, verify_key, agg_id, _agg_param,
              nonce, public_share, input_share):
    k_joint_rand_parts = Prio3.decode_public_share(public_share)
    (meas_share, proof_share, k_blind) = \
        Prio3.decode_leader_share(input_share) if agg_id == 0 else \
        Prio3.decode_helper_share(agg_id, input_share)
    out_share = Prio3.Flp.truncate(meas_share)

    # Compute joint randomness.
    joint_rand = []
    k_corrected_joint_rand, k_joint_rand_part = None, None
    if Prio3.Flp.JOINT_RAND_LEN > 0:
        encoded = Prio3.Flp.Field.encode_vec(meas_share)
        k_joint_rand_part = Prio3.Prg.derive_seed(k_blind,
            Prio3.custom(DST_JOINT_RAND_PART),
            byte(agg_id) + nonce + encoded)
        k_joint_rand_parts[agg_id] = k_joint_rand_part
        k_corrected_joint_rand = Prio3.joint_rand(k_joint_rand_parts)
        joint_rand = Prio3.Prg.expand_into_vec(
            Prio3.Flp.Field,
            k_corrected_joint_rand,
            Prio3.custom(DST_JOINT_RANDOMNESS),
            b'',
            Prio3.Flp.JOINT_RAND_LEN,
        )

    # Query the measurement and proof share.
    query_rand = Prio3.Prg.expand_into_vec(
        Prio3.Flp.Field,
        verify_key,
        Prio3.custom(DST_QUERY_RANDOMNESS),
        nonce,
        Prio3.Flp.QUERY_RAND_LEN,
    )
    verifier_share = Prio3.Flp.query(meas_share,
                                     proof_share,
                                     query_rand,
                                     joint_rand,
                                     Prio3.SHARES)

    prep_msg = Prio3.encode_prep_share(verifier_share,
                                       k_joint_rand_part)
    return (out_share, k_corrected_joint_rand, prep_msg)

def prep_next(Prio3, prep, inbound):
    (out_share, k_corrected_joint_rand, prep_msg) = prep

    if inbound is None:
        return (prep, prep_msg)
```

```
    k_joint_rand_check = Prio3.decode_prep_msg(inbound)
    if k_joint_rand_check != k_corrected_joint_rand:
        raise ERR_VERIFY # joint randomness check failed

    return out_share

def prep_shares_to_prep(Prio3, _agg_param, prep_shares):
    verifier = Prio3.Flp.Field.zeros(Prio3.Flp.VERIFIER_LEN)
    k_joint_rand_parts = []
    for encoded in prep_shares:
        (verifier_share, k_joint_rand_part) = \
            Prio3.decode_prep_share(encoded)

        verifier = vec_add(verifier, verifier_share)

        if Prio3.Flp.JOINT_RAND_LEN > 0:
            k_joint_rand_parts.append(k_joint_rand_part)

    if not Prio3.Flp.decide(verifier):
        raise ERR_VERIFY # proof verifier check failed

    k_joint_rand_check = None
    if Prio3.Flp.JOINT_RAND_LEN > 0:
        k_joint_rand_check = Prio3.joint_rand(k_joint_rand_parts)
    return Prio3.encode_prep_msg(k_joint_rand_check)
```

Figure 14: Preparation state for Prio3.

7.2.3. Validity of Aggregation Parameters

Every input share **MUST** only be used once, regardless of the aggregation parameters used.

```
def is_valid(agg_param, previous_agg_params):
    return len(previous_agg_params) == 0
```

Figure 15: Validity of aggregation parameters for Prio3.

7.2.4. Aggregation

Aggregating a set of output shares is simply a matter of adding up the vectors element-wise.

```
def out_shares_to_agg_share(Prio3, _agg_param, out_shares):
    agg_share = Prio3.Flp.Field.zeros(Prio3.Flp.OUTPUT_LEN)
    for out_share in out_shares:
        agg_share = vec_add(agg_share, out_share)
    return Prio3.Flp.Field.encode_vec(agg_share)
```

Figure 16: Aggregation algorithm for Prio3.

7.2.5. Unsharding

To unshard a set of aggregate shares, the Collector first adds up the vectors element-wise. It then converts each element of the vector into an integer.

```
def agg_shares_to_result(Prio3, _agg_param,
                         agg_shares, num_measurements):
    agg = Prio3.Flp.Field.zeros(Prio3.Flp.OUTPUT_LEN)
    for agg_share in agg_shares:
        agg = vec_add(agg, Prio3.Flp.Field.decode_vec(agg_share))
    return Prio3.Flp.decode(agg, num_measurements)
```

Figure 17: Computation of the aggregate result for Prio3.

7.2.6. Auxiliary Functions

```
def joint_rand(Prio3, k_joint_rand_parts):
    return Prio3.Prg.derive_seed(
        zeros(Prio3.Prg.SEED_SIZE),
        Prio3.custom(DST_JOINT_RAND_SEED),
        concat(k_joint_rand_parts),
    )
```

7.2.6.1. Message Serialization

```
def encode_leader_share(Prio3,
                        meas_share,
                        proof_share,
                        k_blind):
    encoded = Bytes()
    encoded += Prio3.Flp.Field.encode_vec(meas_share)
    encoded += Prio3.Flp.Field.encode_vec(proof_share)
    if Prio3.Flp.JOINT_RAND_LEN > 0:
        encoded += k_blind
    return encoded

def decode_leader_share(Prio3, encoded):
    l = Prio3.Flp.Field.ENCODED_SIZE * Prio3.Flp.INPUT_LEN
    encoded_meas_share, encoded = encoded[:l], encoded[l:]
    meas_share = Prio3.Flp.Field.decode_vec(encoded_meas_share)
    l = Prio3.Flp.Field.ENCODED_SIZE * Prio3.Flp.PROOF_LEN
    encoded_proof_share, encoded = encoded[:l], encoded[l:]
    proof_share = Prio3.Flp.Field.decode_vec(encoded_proof_share)
    l = Prio3.Prg.SEED_SIZE
    if Prio3.Flp.JOINT_RAND_LEN == 0:
        if len(encoded) != 0:
            raise ERR_DECODE
        return (meas_share, proof_share, None)
    k_blind, encoded = encoded[:l], encoded[l:]
    if len(encoded) != 0:
        raise ERR_DECODE
    return (meas_share, proof_share, k_blind)

def encode_helper_share(Prio3,
                        k_meas_share,
                        k_proof_share,
                        k_blind):
    encoded = Bytes()
    encoded += k_meas_share
    encoded += k_proof_share
    if Prio3.Flp.JOINT_RAND_LEN > 0:
        encoded += k_blind
    return encoded

def decode_helper_share(Prio3, agg_id, encoded):
    c_meas_share = Prio3.custom(DST_MEASUREMENT_SHARE)
    c_proof_share = Prio3.custom(DST_PROOF_SHARE)
    l = Prio3.Prg.SEED_SIZE
    k_meas_share, encoded = encoded[:l], encoded[l:]
    meas_share = Prio3.Prg.expand_into_vec(Prio3.Flp.Field,
                                           k_meas_share,
                                           c_meas_share,
                                           byte(agg_id),
                                           Prio3.Flp.INPUT_LEN)
```

```
    k_proof_share, encoded = encoded[:l], encoded[l:]
    proof_share = Prio3.Prg.expand_into_vec(Prio3.Flp.Field,
                                            k_proof_share,
                                            c_proof_share,
                                            byte(agg_id),
                                            Prio3.Flp.PROOF_LEN)
    if Prio3.Flp.JOINT_RAND_LEN == 0:
        if len(encoded) != 0:
            raise ERR_DECODE
        return (meas_share, proof_share, None)
    k_blind, encoded = encoded[:l], encoded[l:]
    if len(encoded) != 0:
        raise ERR_DECODE
    return (meas_share, proof_share, k_blind)

def encode_public_share(Prio3,
                        k_joint_rand_parts):
    encoded = Bytes()
    if Prio3.Flp.JOINT_RAND_LEN > 0:
        encoded += concat(k_joint_rand_parts)
    return encoded

def decode_public_share(Prio3, encoded):
    l = Prio3.Prg.SEED_SIZE
    if Prio3.Flp.JOINT_RAND_LEN == 0:
        if len(encoded) != 0:
            raise ERR_DECODE
        return None
    k_joint_rand_parts = []
    for i in range(Prio3.SHARES):
        k_joint_rand_part, encoded = encoded[:l], encoded[l:]
        k_joint_rand_parts.append(k_joint_rand_part)
    if len(encoded) != 0:
        raise ERR_DECODE
    return k_joint_rand_parts

def encode_prep_share(Prio3, verifier, k_joint_rand):
    encoded = Bytes()
    encoded += Prio3.Flp.Field.encode_vec(verifier)
    if Prio3.Flp.JOINT_RAND_LEN > 0:
        encoded += k_joint_rand
    return encoded

def decode_prep_share(Prio3, encoded):
    l = Prio3.Flp.Field.ENCODED_SIZE * Prio3.Flp.VERIFIER_LEN
    encoded_verifier, encoded = encoded[:l], encoded[l:]
    verifier = Prio3.Flp.Field.decode_vec(encoded_verifier)
    if Prio3.Flp.JOINT_RAND_LEN == 0:
        if len(encoded) != 0:
```

```
            raise ERR_DECODE
        return (verifier, None)
    l = Prio3.Prg.SEED_SIZE
    k_joint_rand, encoded = encoded[:l], encoded[l:]
    if len(encoded) != 0:
        raise ERR_DECODE
    return (verifier, k_joint_rand)

def encode_prep_msg(Prio3, k_joint_rand_check):
    encoded = Bytes()
    if Prio3.Flp.JOINT_RAND_LEN > 0:
        encoded += k_joint_rand_check
    return encoded

def decode_prep_msg(Prio3, encoded):
    if Prio3.Flp.JOINT_RAND_LEN == 0:
        if len(encoded) != 0:
            raise ERR_DECODE
        return None
    l = Prio3.Prg.SEED_SIZE
    k_joint_rand_check, encoded = encoded[:l], encoded[l:]
    if len(encoded) != 0:
        raise ERR_DECODE
    return k_joint_rand_check
```

7.3. A General-Purpose FLP

This section describes an FLP based on the construction from in [BBCGGI19], Section 4.2. We begin in Section 7.3.1 with an overview of their proof system and the extensions to their proof system made here. The construction is specified in Section 7.3.3.

> OPEN ISSUE We're not yet sure if specifying this general-purpose FLP is desirable. It might be preferable to specify specialized FLPs for each data type that we want to standardize, for two reasons. First, clear and concise specifications are likely easier to write for specialized FLPs rather than the general one. Second, we may end up tailoring each FLP to the measurement type in a way that improves performance, but breaks compatibility with the general-purpose FLP.
>
> In any case, we can't make this decision until we know which data types to standardize, so for now, we'll stick with the general-purpose construction. The reference implementation can be found at https://github.com/cfrg/draft-irtf-cfrg-vdaf/tree/main/poc.
>
> OPEN ISSUE Chris Wood points out that the this section reads more like a paper than a standard. Eventually we'll want to work this into something that is readily consumable by the CFRG.

7.3.1. Overview

In the proof system of [BBCGGI19], validity is defined via an arithmetic circuit evaluated over the input: If the circuit output is zero, then the input is deemed valid; otherwise, if the circuit output is non-zero, then the input is deemed invalid. Thus the goal of the proof system is merely to allow the verifier to evaluate the validity circuit over the input. For our application (Section 7), this computation is distributed among multiple Aggregators, each of which has only a share of the input.

Suppose for a moment that the validity circuit C is affine, meaning its only operations are addition and multiplication-by-constant. In particular, suppose the circuit does not contain a multiplication gate whose operands are both non-constant. Then to decide if an input x is valid, each Aggregator could evaluate C on its share of x locally, broadcast the output share to its peers, then combine the output shares locally to recover C(x). This is true because for any SHARES-way secret sharing of x it holds that

```
C(x_shares[0] + ... + x_shares[SHARES-1]) =
    C(x_shares[0]) + ... + C(x_shares[SHARES-1])
```

(Note that, for this equality to hold, it may be necessary to scale any constants in the circuit by SHARES.) However this is not the

case if C is not-affine (i.e., it contains at least one multiplication gate whose operands are non-constant). In the proof system of [BBCGGI19], the proof is designed to allow the (distributed) verifier to compute the non-affine operations using only linear operations on (its share of) the input and proof.

To make this work, the proof system is restricted to validity circuits that exhibit a special structure. Specifically, an arithmetic circuit with "G-gates" (see [BBCGGI19], Definition 5.2) is composed of affine gates and any number of instances of a distinguished gate G, which may be non-affine. We will refer to this class of circuits as 'gadget circuits' and to G as the "gadget".

As an illustrative example, consider a validity circuit C that recognizes the set L = set([0], [1]). That is, C takes as input a length-1 vector x and returns 0 if x[0] is in [0,2) and outputs something else otherwise. This circuit can be expressed as the following degree-2 polynomial:

```
C(x) = (x[0] - 1) * x[0] = x[0]^2 - x[0]
```

This polynomial recognizes L because x[0]^2 = x[0] is only true if x[0] == 0 or x[0] == 1. Notice that the polynomial involves a non-affine operation, x[0]^2. In order to apply [BBCGGI19], Theorem 4.3, the circuit needs to be rewritten in terms of a gadget that subsumes this non-affine operation. For example, the gadget might be multiplication:

```
Mul(left, right) = left * right
```

The validity circuit can then be rewritten in terms of Mul like so:

```
C(x[0]) = Mul(x[0], x[0]) - x[0]
```

The proof system of [BBCGGI19] allows the verifier to evaluate each instance of the gadget (i.e., Mul(x[0], x[0]) in our example) using a linear function of the input and proof. The proof is constructed roughly as follows. Let C be the validity circuit and suppose the gadget is arity-L (i.e., it has L input wires.). Let wire[j-1,k-1] denote the value of the jth wire of the kth call to the gadget during the evaluation of C(x). Suppose there are M such calls and fix distinct field elements alpha[0], ..., alpha[M-1]. (We will require these points to have a special property, as we'll discuss in Section 7.3.1.1; but for the moment it is only important that they are distinct.)

The prover constructs from wire and alpha a polynomial that, when evaluated at alpha[k-1], produces the output of the kth call to the gadget. Let us call this the "gadget polynomial". Polynomial evaluation is linear, which means that, in the distributed setting,

the Client can disseminate additive shares of the gadget polynomial that the Aggregators then use to compute additive shares of each gadget output, allowing each Aggregator to compute its share of C(x) locally.

There is one more wrinkle, however: It is still possible for a malicious prover to produce a gadget polynomial that would result in C(x) being computed incorrectly, potentially resulting in an invalid input being accepted. To prevent this, the verifier performs a probabilistic test to check that the gadget polynomial is well-formed. This test, and the procedure for constructing the gadget polynomial, are described in detail in Section 7.3.3.

7.3.1.1. Extensions

The FLP described in the next section extends the proof system [BBCGGI19], Section 4.2 in three ways.

First, the validity circuit in our construction includes an additional, random input (this is the "joint randomness" derived from the input shares in Prio3; see Section 7.2). This allows for circuit optimizations that trade a small soundness error for a shorter proof. For example, consider a circuit that recognizes the set of length-N vectors for which each element is either one or zero. A deterministic circuit could be constructed for this language, but it would involve a large number of multiplications that would result in a large proof. (See the discussion in [BBCGGI19], Section 5.2 for details). A much shorter proof can be constructed for the following randomized circuit:

```
C(inp, r) = r * Range2(inp[0]) + ... + r^N * Range2(inp[N-1])
```

(Note that this is a special case of [BBCGGI19], Theorem 5.2.) Here inp is the length-N input and r is a random field element. The gadget circuit Range2 is the "range-check" polynomial described above, i.e., Range2(x) = x^2 - x. The idea is that, if inp is valid (i.e., each inp[j] is in [0,2)), then the circuit will evaluate to 0 regardless of the value of r; but if inp[j] is not in [0,2) for some j, the output will be non-zero with high probability.

The second extension implemented by our FLP allows the validity circuit to contain multiple gadget types. (This generalization was suggested in [BBCGGI19], Remark 4.5.) For example, the following circuit is allowed, where Mul and Range2 are the gadgets defined above (the input has length N+1):

```
C(inp, r) = r * Range2(inp[0]) + ... + r^N * Range2(inp[N-1]) + \
            2^0 * inp[0]       + ... + 2^(N-1) * inp[N-1]     - \
            Mul(inp[N], inp[N])
```

Finally, [BBCGGI19], Theorem 4.3 makes no restrictions on the choice of the fixed points alpha[0], ..., alpha[M-1], other than to require that the points are distinct. In this document, the fixed points are chosen so that the gadget polynomial can be constructed efficiently using the Cooley-Tukey FFT ("Fast Fourier Transform") algorithm. Note that this requires the field to be "FFT-friendly" as defined in Section 6.1.2.

7.3.2. Validity Circuits

The FLP described in Section 7.3.3 is defined in terms of a validity circuit Valid that implements the interface described here.

A concrete Valid defines the following parameters:

Parameter	Description
GADGETS	A list of gadgets
GADGET_CALLS	Number of times each gadget is called
INPUT_LEN	Length of the input
OUTPUT_LEN	Length of the aggregatable output
JOINT_RAND_LEN	Length of the random input
Measurement	The type of measurement
AggResult	Type of the aggregate result
Field	An FFT-friendly finite field as defined in Section 6.1.2

Table 9: Validity circuit parameters.

Each gadget G in GADGETS defines a constant DEGREE that specifies the circuit's "arithmetic degree". This is defined to be the degree of the polynomial that computes it. For example, the Mul circuit in Section 7.3.1 is defined by the polynomial Mul(x) = x * x, which has degree 2. Hence, the arithmetic degree of this gadget is 2.

Each gadget also defines a parameter ARITY that specifies the circuit's arity (i.e., the number of input wires).

A concrete Valid provides the following methods for encoding a measurement as an input vector, truncating an input vector to the length of an aggregatable output, and converting an aggregated output to an aggregate result:

* Valid.encode(measurement: Measurement) -> Vec[Field] returns a vector of length INPUT_LEN representing a measurement.

* Valid.truncate(input: Vec[Field]) -> Vec[Field] returns a vector of length OUTPUT_LEN representing an aggregatable output.

*Valid.decode(output: Vec[Field], num_measurements: Unsigned) -> AggResult returns an aggregate result.

Finally, the following class methods are derived for each concrete Valid:

```
# Length of the prover randomness.
def prove_rand_len(Valid):
    return sum(map(lambda g: g.ARITY, Valid.GADGETS))

# Length of the query randomness.
def query_rand_len(Valid):
    return len(Valid.GADGETS)

# Length of the proof.
def proof_len(Valid):
    length = 0
    for (g, g_calls) in zip(Valid.GADGETS, Valid.GADGET_CALLS):
        P = next_power_of_2(1 + g_calls)
        length += g.ARITY + g.DEGREE * (P - 1) + 1
    return length

# Length of the verifier message.
def verifier_len(Valid):
    length = 1
    for g in Valid.GADGETS:
        length += g.ARITY + 1
    return length
```

Figure 18: Derived methods for validity circuits.

7.3.3. Construction

This section specifies FlpGeneric, an implementation of the Flp interface (Section 7.1). It has as a generic parameter a validity circuit Valid implementing the interface defined in Section 7.3.2.

> NOTE A reference implementation can be found in https://github.com/cfrg/draft-irtf-cfrg-vdaf/blob/main/poc/flp_generic.sage.

The FLP parameters for FlpGeneric are defined in Table 10. The required methods for generating the proof, generating the verifier, and deciding validity are specified in the remaining subsections.

In the remainder, we let [n] denote the set {1, ..., n} for positive integer n. We also define the following constants:

*Let H = len(Valid.GADGETS)

* For each i in [H]:

  - Let G_i = Valid.GADGETS[i]

  - Let L_i = Valid.GADGETS[i].ARITY

  - Let M_i = Valid.GADGET_CALLS[i]

  - Let P_i = next_power_of_2(M_i+1)

  - Let alpha_i = Field.gen()^(Field.GEN_ORDER / P_i)

Parameter	**Value**
PROVE_RAND_LEN	Valid.prove_rand_len() (see Section 7.3.2)
QUERY_RAND_LEN	Valid.query_rand_len() (see Section 7.3.2)
JOINT_RAND_LEN	Valid.JOINT_RAND_LEN
INPUT_LEN	Valid.INPUT_LEN
OUTPUT_LEN	Valid.OUTPUT_LEN
PROOF_LEN	Valid.proof_len() (see Section 7.3.2)
VERIFIER_LEN	Valid.verifier_len() (see Section 7.3.2)
Measurement	Valid.Measurement
Field	Valid.Field

Table 10: FLP Parameters of FlpGeneric.

7.3.3.1. Proof Generation

On input inp, prove_rand, and joint_rand, the proof is computed as follows:

1. For each i in [H] create an empty table wire_i.

2. Partition the prover randomness prove_rand into sub-vectors seed_1, ..., seed_H where len(seed_i) == L_i for all i in [H]. Let us call these the "wire seeds" of each gadget.

3. Evaluate Valid on input of inp and joint_rand, recording the inputs of each gadget in the corresponding table. Specifically, for every i in [H], set wire_i[j-1,k-1] to the value on the jth wire into the kth call to gadget G_i.

4. Compute the "wire polynomials". That is, for every i in [H] and j in [L_i], construct poly_wire_i[j-1], the jth wire polynomial for the ith gadget, as follows:

   * Let w = [seed_i[j-1], wire_i[j-1,0], ..., wire_i[j-1,M_i-1]].

*Let padded_w = w + Field.zeros(P_i - len(w)).

NOTE We pad w to the nearest power of 2 so that we can use FFT for interpolating the wire polynomials. Perhaps there is some clever math for picking wire_inp in a way that avoids having to pad.

*Let poly_wire_i[j-1] be the lowest degree polynomial for which poly_wire_i[j-1](alpha_i^k) == padded_w[k] for all k in [P_i].

5. Compute the "gadget polynomials". That is, for every i in [H]:

   *Let poly_gadget_i = G_i(poly_wire_i[0], ..., poly_wire_i[L_i-1]). That is, evaluate the circuit G_i on the wire polynomials for the ith gadget. (Arithmetic is in the ring of polynomials over Field.)

The proof is the vector proof = seed_1 + coeff_1 + ... + seed_H + coeff_H, where coeff_i is the vector of coefficients of poly_gadget_i for each i in [H].

7.3.3.2. Query Generation

On input of inp, proof, query_rand, and joint_rand, the verifier message is generated as follows:

1. For every i in [H] create an empty table wire_i.

2. Partition proof into the sub-vectors seed_1, coeff_1, ..., seed_H, coeff_H defined in Section 7.3.3.1.

3. Evaluate Valid on input of inp and joint_rand, recording the inputs of each gadget in the corresponding table. This step is similar to the prover's step (3.) except the verifier does not evaluate the gadgets. Instead, it computes the output of the kth call to G_i by evaluating poly_gadget_i(alpha_i^k). Let v denote the output of the circuit evaluation.

4. Compute the wire polynomials just as in the prover's step (4.).

5. Compute the tests for well-formedness of the gadget polynomials. That is, for every i in [H]:

   *Let t = query_rand[i]. Check if t^(P_i) == 1: If so, then raise ERR_ABORT and halt. (This prevents the verifier from inadvertently leaking a gadget output in the verifier message.)

   *Let y_i = poly_gadget_i(t).

*For each j in [0,L_i) let x_i[j-1] = poly_wire_i[j-1](t).

The verifier message is the vector verifier = [v] + x_1 + [y_1] + ... + x_H + [y_H].

7.3.3.3. Decision

On input of vector verifier, the verifier decides if the input is valid as follows:

1. Parse verifier into v, x_1, y_1, ..., x_H, y_H as defined in Section 7.3.3.2.

2. Check for well-formedness of the gadget polynomials. For every i in [H]:

   *Let z = G_i(x_i). That is, evaluate the circuit G_i on x_i and set z to the output.

   *If z != y_i, then return False and halt.

3. Return True if v == 0 and False otherwise.

7.3.3.4. Encoding

The FLP encoding and truncation methods invoke Valid.encode, Valid.truncate, and Valid.decode in the natural way.

7.4. Instantiations

This section specifies instantiations of Prio3 for various measurement types. Each uses FlpGeneric as the FLP (Section 7.3) and is determined by a validity circuit (Section 7.3.2) and a PRG (Section 6.2). Test vectors for each can be found in Appendix "Test Vectors".

NOTE Reference implementations of each of these VDAFs can be found in https://github.com/cfrg/draft-irtf-cfrg-vdaf/blob/main/poc/vdaf_prio3.sage.

7.4.1. Prio3Count

Our first instance of Prio3 is for a simple counter: Each measurement is either one or zero and the aggregate result is the sum of the measurements.

This instance uses PrgSha3 (Section 6.2.1) as its PRG. Its validity circuit, denoted Count, uses Field64 (Table 3) as its finite field. Its gadget, denoted Mul, is the degree-2, arity-2 gadget defined as

```
def Mul(x, y):
    return x * y
```

The validity circuit is defined as

```
def Count(inp: Vec[Field64]):
    return Mul(inp[0], inp[0]) - inp[0]
```

The measurement is encoded and decoded as a singleton vector in the natural way. The parameters for this circuit are summarized below.

Parameter	**Value**
GADGETS	[Mul]
GADGET_CALLS	[1]
INPUT_LEN	1
OUTPUT_LEN	1
JOINT_RAND_LEN	0
Measurement	Unsigned, in range [0,2)
AggResult	Unsigned
Field	Field64 (Table 3)

Table 11: Parameters of validity circuit Count.

7.4.2. Prio3Sum

The next instance of Prio3 supports summing of integers in a pre-determined range. Each measurement is an integer in range [0, 2^bits), where bits is an associated parameter.

This instance of Prio3 uses PrgSha3 (Section 6.2.1) as its PRG. Its validity circuit, denoted Sum, uses Field128 (Table 4) as its finite field. The measurement is encoded as a length-bits vector of field elements, where the lth element of the vector represents the lth bit of the summand:

```
def encode(Sum, measurement: Integer):
    if 0 > measurement or measurement >= 2^Sum.INPUT_LEN:
        raise ERR_INPUT

    encoded = []
    for l in range(Sum.INPUT_LEN):
        encoded.append(Sum.Field((measurement >> l) & 1))
    return encoded

def truncate(Sum, inp):
    decoded = Sum.Field(0)
    for (l, b) in enumerate(inp):
        w = Sum.Field(1 << l)
        decoded += w * b
    return [decoded]

def decode(Sum, output, _num_measurements):
    return output[0].as_unsigned()
```

The validity circuit checks that the input consists of ones and zeros. Its gadget, denoted Range2, is the degree-2, arity-1 gadget defined as

```
def Range2(x):
    return x^2 - x
```

The validity circuit is defined as

```
def Sum(inp: Vec[Field128], joint_rand: Vec[Field128]):
    out = Field128(0)
    r = joint_rand[0]
    for x in inp:
        out += r * Range2(x)
        r *= joint_rand[0]
    return out
```

Parameter	Value
GADGETS	[Range2]
GADGET_CALLS	[bits]
INPUT_LEN	bits
OUTPUT_LEN	1
JOINT_RAND_LEN	1
Measurement	Unsigned, in range [0, 2^bits)
AggResult	Unsigned
Field	Field128 (Table 4)

Table 12: Parameters of validity circuit Sum.

7.4.3. Prio3Histogram

This instance of Prio3 allows for estimating the distribution of the measurements by computing a simple histogram. Each measurement is an arbitrary integer and the aggregate result counts the number of measurements that fall in a set of fixed buckets.

This instance of Prio3 uses PrgSha3 (Section 6.2.1) as its PRG. Its validity circuit, denoted Histogram, uses Field128 (Table 4) as its finite field. The measurement is encoded as a one-hot vector representing the bucket into which the measurement falls (let bucket denote a sequence of monotonically increasing integers):

```
def encode(Histogram, measurement: Integer):
    boundaries = buckets + [Infinity]
    encoded = [Field128(0) for _ in range(len(boundaries))]
    for i in range(len(boundaries)):
        if measurement <= boundaries[i]:
            encoded[i] = Field128(1)
            return encoded

def truncate(Histogram, inp: Vec[Field128]):
    return inp

def decode(Histogram, output: Vec[Field128], _num_measurements):
    return [bucket_count.as_unsigned() for bucket_count in output]
```

The validity circuit uses Range2 (see Section 7.4.2) as its single gadget. It checks for one-hotness in two steps, as follows:

```
def Histogram(inp: Vec[Field128],
              joint_rand: Vec[Field128],
              num_shares: Unsigned):
    # Check that each bucket is one or zero.
    range_check = Field128(0)
    r = joint_rand[0]
    for x in inp:
        range_check += r * Range2(x)
        r *= joint_rand[0]

    # Check that the buckets sum to 1.
    sum_check = -Field128(1) * Field128(num_shares).inv()
    for b in inp:
        sum_check += b

    out = joint_rand[1]   * range_check + \
          joint_rand[1]^2 * sum_check
    return out
```

Note that this circuit depends on the number of shares into which the input is sharded. This is provided to the FLP by Prio3.

Parameter	Value
GADGETS	[Range2]
GADGET_CALLS	[buckets + 1]
INPUT_LEN	buckets + 1
OUTPUT_LEN	buckets + 1
JOINT_RAND_LEN	2
Measurement	Integer
AggResult	Vec[Unsigned]
Field	Field128 (Table 4)

Table 13: Parameters of validity circuit Histogram.

8. Poplar1

This section specifies Poplar1, a VDAF for the following task. Each Client holds a string of length BITS and the Aggregators hold a set of l-bit strings, where l <= BITS. We will refer to the latter as the set of "candidate prefixes". The Aggregators' goal is to count how many inputs are prefixed by each candidate prefix.

This functionality is the core component of the Poplar protocol [BBCGGI21]. At a high level, the protocol works as follows.

1. Each Client splits its input string into input shares and sends one share to each Aggregator.

2. The Aggregators agree on an initial set of candidate prefixes, say 0 and 1.

3. The Aggregators evaluate the VDAF on each set of input shares and aggregate the recovered output shares. The aggregation parameter is the set of candidate prefixes.

4. The Aggregators send their aggregate shares to the Collector, who combines them to recover the counts of each candidate prefix.

5. Let H denote the set of prefixes that occurred at least t times. If the prefixes all have length BITS, then H is the set of t-heavy-hitters. Otherwise compute the next set of candidate prefixes as follows. For each p in H, add add p || 0 and p || 1 to the set. Repeat step 3 with the new set of candidate prefixes.

Poplar1 is constructed from an "Incremental Distributed Point Function (IDPF)", a primitive described by [BBCGGI21] that generalizes the notion of a Distributed Point Function (DPF) [GI14]. Briefly, a DPF is used to distribute the computation of a "point function", a function that evaluates to zero on every input except at a programmable "point". The computation is distributed in such a way that no one party knows either the point or what it evaluates to.

An IDPF generalizes this "point" to a path on a full binary tree from the root to one of the leaves. It is evaluated on an "index" representing a unique node of the tree. If the node is on the programmed path, then function evaluates to to a non-zero value; otherwise it evaluates to zero. This structure allows an IDPF to provide the functionality required for the above protocol, while at the same time ensuring the same degree of privacy as a DPF.

Poplar1 composes an IDPF with the "secure sketching" protocol of [BBCGGI21]. This protocol ensures that evaluating a set of input shares on a unique set of candidate prefixes results in shares of a "one-hot" vector, i.e., a vector that is zero everywhere except for one element, which is equal to one.

The remainder of this section is structured as follows. IDPFs are defined in Section 8.1; a concrete instantiation is given Section 8.3. The Poplar1 VDAF is defined in Section 8.2 in terms of a generic IDPF. Finally, a concrete instantiation of Poplar1 is specified in Section 8.4; test vectors can be found in Appendix "Test Vectors".

8.1. Incremental Distributed Point Functions (IDPFs)

An IDPF is defined over a domain of size 2^BITS, where BITS is constant defined by the IDPF. Indexes into the IDPF tree are encoded as integers in range [0, 2^BITS). The Client specifies an index alpha and a vector of values beta, one for each "level" L in range [0, BITS). The key generation algorithm generates one IDPF "key" for each Aggregator. When evaluated at level L and index 0 <= prefix < 2^L, each IDPF key returns an additive share of beta[L] if prefix is the L-bit prefix of alpha and shares of zero otherwise.

An index x is defined to be a prefix of another index y as follows. Let LSB(x, N) denote the least significant N bits of positive integer x. By definition, a positive integer 0 <= x < 2^L is said to be the length-L prefix of positive integer 0 <= y < 2^BITS if LSB(x, L) is equal to the most significant L bits of LSB(y, BITS), For example, 6 (110 in binary) is the length-3 prefix of 25 (11001), but 7 (111) is not.

Each of the programmed points beta is a vector of elements of some finite field. We distinguish two types of fields: One for inner nodes (denoted Idpf.FieldInner), and one for leaf nodes (Idpf.FieldLeaf). (Our instantiation of Poplar1 (Section 8.4) will use a much larger field for leaf nodes than for inner nodes. This is to ensure the IDPF is "extractable" as defined in [BBCGGI21], Definition 1.)

A concrete IDPF defines the types and constants enumerated in Table 14. In the remainder we write Idpf.Vec as shorthand for the type Union[Vec[Vec[Idpf.FieldInner]], Vec[Vec[Idpf.FieldLeaf]]]. (This type denotes either a vector of inner node field elements or leaf node field elements.) The scheme is comprised of the following algorithms:

* Idpf.gen(alpha: Unsigned, beta_inner: Vec[Vec[Idpf.FieldInner]], beta_leaf: Vec[Idpf.FieldLeaf]) -> (Bytes, Vec[Bytes]) is the randomized IDPF-key generation algorithm. Its inputs are the index alpha and the values beta. The value of alpha **MUST** be in range [0, 2^BITS). The output is a public part that is sent to all Aggregators and a vector of private IDPF keys, one for each aggregator.

* Idpf.eval(agg_id: Unsigned, public_share: Bytes, key: Bytes, level: Unsigned, prefixes: Vec[Unsigned]) -> Idpf.Vec is the deterministic, stateless IDPF-key evaluation algorithm run by each Aggregator. Its inputs are the Aggregator's unique identifier, the public share distributed to all of the Aggregators, the Aggregator's IDPF key, the "level" at which to evaluate the IDPF, and the sequence of candidate prefixes. It returns the share of the value corresponding to each candidate prefix.

  The output type depends on the value of level: If level < Idpf.BITS-1, the output is the value for an inner node, which has type Vec[Vec[Idpf.FieldInner]]; otherwise, if level == Idpf.BITS-1, then the output is the value for a leaf node, which has type Vec[Vec[Idpf.FieldLeaf]].

  The value of level **MUST** be in range [0, BITS). The indexes in prefixes **MUST** all be distinct and in range [0, 2^level).

  Applications **MUST** ensure that the Aggregator's identifier is equal to the integer in range [0, SHARES) that matches the index of key in the sequence of IDPF keys output by the Client.

In addition, the following method is derived for each concrete Idpf:

```
def current_field(Idpf, level):
    return Idpf.FieldInner if level < Idpf.BITS-1 \
                else Idpf.FieldLeaf
```

Finally, an implementation note. The interface for IDPFs specified here is stateless, in the sense that there is no state carried between IDPF evaluations. This is to align the IDPF syntax with the VDAF abstraction boundary, which does not include shared state across across VDAF evaluations. In practice, of course, it will often be beneficial to expose a stateful API for IDPFs and carry the state across evaluations. See Section 8.3 for details.

Parameter	**Description**
SHARES	Number of IDPF keys output by IDPF-key generator
BITS	Length in bits of each input string
VALUE_LEN	Number of field elements of each output value
KEY_SIZE	Size in bytes of each IDPF key
FieldInner	Implementation of Field (Section 6.1) used for values of inner nodes
FieldLeaf	Implementation of Field used for values of leaf nodes
Prg	Implementation of Prg (Section 6.2)

Table 14: Constants and types defined by a concrete IDPF.

8.2. Construction

This section specifies Poplar1, an implementation of the Vdaf interface (Section 5). It is defined in terms of any Idpf (Section 8.1) for which Idpf.SHARES == 2 and Idpf.VALUE_LEN == 2. The associated constants and types required by the Vdaf interface are defined in Table 15. The methods required for sharding, preparation, aggregation, and unsharding are described in the remaining subsections. These methods make use of constants defined in Table 16.

Parameter	**Value**
VERIFY_KEY_SIZE	Idpf.Prg.SEED_SIZE
ROUNDS	2
SHARES	2
Measurement	Unsigned
AggParam	Tuple[Unsigned, Vec[Unsigned]]
Prep	Tuple[Bytes, Unsigned, Idpf.Vec]
OutShare	Idpf.Vec
AggResult	Vec[Unsigned]

Table 15: VDAF parameters for Poplar1.

Variable	Value
DST_SHARD_RAND: Unsigned	1
DST_CORR_INNER: Unsigned	2
DST_CORR_LEAF: Unsigned	3
DST_VERIFY_RAND: Unsigned	4

Table 16: Constants used by Poplar1.

8.2.1. Client

The client's input is an IDPF index, denoted alpha. The programmed IDPF values are pairs of field elements (1, k) where each k is chosen at random. This random value is used as part of the secure sketching protocol of [BBCGGI21], Appendix C.4. After evaluating their IDPF key shares on a given sequence of candidate prefixes, the sketching protocol is used by the Aggregators to verify that they hold shares of a one-hot vector. In addition, for each level of the tree, the prover generates random elements a, b, and c and computes

```
A = -2*a + k
B = a^2 + b - k*a + c
```

and sends additive shares of a, b, c, A and B to the Aggregators. Putting everything together, the input-distribution algorithm is defined as follows. Function encode_input_shares is defined in Section 8.2.6.

```
def measurement_to_input_shares(Poplar1, measurement, nonce):
    prg = Poplar1.Idpf.Prg(gen_rand(Poplar1.Idpf.Prg.SEED_SIZE),
                           Poplar1.custom(DST_SHARD_RAND), b'')

    # Construct the IDPF values for each level of the IDPF tree.
    # Each "data" value is 1; in addition, the Client generates
    # a random "authenticator" value used by the Aggregators to
    # compute the sketch during preparation. This sketch is used
    # to verify the one-hotness of their output shares.
    beta_inner = [
        [Poplar1.Idpf.FieldInner(1), k] \
            for k in prg.next_vec(Poplar1.Idpf.FieldInner,
                                  Poplar1.Idpf.BITS - 1) ]
    beta_leaf = [Poplar1.Idpf.FieldLeaf(1)] + \
        prg.next_vec(Poplar1.Idpf.FieldLeaf, 1)

    # Generate the IDPF keys.
    (public_share, keys) = \
        Poplar1.Idpf.gen(measurement, beta_inner, beta_leaf)

    # Generate correlated randomness used by the Aggregators to
    # compute a sketch over their output shares. PRG seeds are
    # used to encode shares of the `(a, b, c)` triples.
    # (See [BBCGGI21, Appendix C.4].)
    corr_seed = [
        gen_rand(Poplar1.Idpf.Prg.SEED_SIZE),
        gen_rand(Poplar1.Idpf.Prg.SEED_SIZE),
    ]

    corr_offsets = vec_add(
        Poplar1.Idpf.Prg.expand_into_vec(
            Poplar1.Idpf.FieldInner,
            corr_seed[0],
            Poplar1.custom(DST_CORR_INNER),
            byte(0) + nonce,
            3 * (Poplar1.Idpf.BITS-1),
        ),
        Poplar1.Idpf.Prg.expand_into_vec(
            Poplar1.Idpf.FieldInner,
            corr_seed[1],
            Poplar1.custom(DST_CORR_INNER),
            byte(1) + nonce,
            3 * (Poplar1.Idpf.BITS-1),
        ),
    )
    corr_offsets += vec_add(
        Poplar1.Idpf.Prg.expand_into_vec(
            Poplar1.Idpf.FieldLeaf,
            corr_seed[0],
```

```
            Poplar1.custom(DST_CORR_LEAF),
            byte(0) + nonce,
            3,
        ),
        Poplar1.Idpf.Prg.expand_into_vec(
            Poplar1.Idpf.FieldLeaf,
            corr_seed[1],
            Poplar1.custom(DST_CORR_LEAF),
            byte(1) + nonce,
            3,
        ),
    )

    # For each level of the IDPF tree, shares of the `(A, B)`
    # pairs are computed from the corresponding `(a, b, c)`
    # triple and authenticator value `k`.
    corr_inner = [[], []]
    for level in range(Poplar1.Idpf.BITS):
        Field = Poplar1.Idpf.current_field(level)
        k = beta_inner[level][1] if level < Poplar1.Idpf.BITS - 1 \
            else beta_leaf[1]
        (a, b, c), corr_offsets = corr_offsets[:3], corr_offsets[3:]
        A = -Field(2) * a + k
        B = a^2 + b - a * k + c
        corr1 = prg.next_vec(Field, 2)
        corr0 = vec_sub([A, B], corr1)
        if level < Poplar1.Idpf.BITS - 1:
            corr_inner[0] += corr0
            corr_inner[1] += corr1
        else:
            corr_leaf = [corr0, corr1]

    # Each input share consists of the Aggregator's IDPF key
    # and a share of the correlated randomness.
    return (public_share,
            Poplar1.encode_input_shares(
                keys, corr_seed, corr_inner, corr_leaf))
```

Figure 19: The input-distribution algorithm for Poplar1.

8.2.2. Preparation

The aggregation parameter encodes a sequence of candidate prefixes. When an Aggregator receives an input share from the Client, it begins by evaluating its IDPF share on each candidate prefix, recovering a data_share and auth_share for each. The Aggregators use these and the correlation shares provided by the Client to verify that the sequence of data_share values are additive shares of a one-hot vector.

Aggregators **MUST** ensure the candidate prefixes are all unique and appear in lexicographic order. (This is enforced in the definition of prep_init() below.) Uniqueness is necessary to ensure the refined measurement (i.e., the sum of the output shares) is in fact a one-hot vector. Otherwise, sketch verification might fail, causing the Aggregators to erroneously reject a report that is actually valid. Note that enforcing the order is not strictly necessary, but this does allow uniqueness to be determined more efficiently.

The algorithms below make use of the auxiliary function decode_input_share() defined in Section 8.2.6.

```
def prep_init(Poplar1, verify_key, agg_id, agg_param,
              nonce, public_share, input_share):
    (level, prefixes) = agg_param
    (key, corr_seed, corr_inner, corr_leaf) = \
        Poplar1.decode_input_share(input_share)
    Field = Poplar1.Idpf.current_field(level)

    # Ensure that candidate prefixes are all unique and appear in
    # lexicographic order.
    for i in range(1,len(prefixes)):
        if prefixes[i-1] >= prefixes[i]:
            raise ERR_INPUT # out-of-order prefix

    # Evaluate the IDPF key at the given set of prefixes.
    value = Poplar1.Idpf.eval(
        agg_id, public_share, key, level, prefixes)

    # Get shares of the correlated randomness for computing the
    # Aggregator's share of the sketch for the given level of the IDPF
    # tree.
    if level < Poplar1.Idpf.BITS - 1:
        corr_prg = Poplar1.Idpf.Prg(corr_seed,
                                    Poplar1.custom(DST_CORR_INNER),
                                    byte(agg_id) + nonce)
        # Fast-forward the PRG state to the current level.
        corr_prg.next_vec(Field, 3 * level)
    else:
        corr_prg = Poplar1.Idpf.Prg(corr_seed,
                                    Poplar1.custom(DST_CORR_LEAF),
                                    byte(agg_id) + nonce)
    (a_share, b_share, c_share) = corr_prg.next_vec(Field, 3)
    (A_share, B_share) = corr_inner[2*level:2*(level+1)] \
        if level < Poplar1.Idpf.BITS - 1 else corr_leaf

    # Compute the Aggregator's first round of the sketch. These are
    # called the "masked input values" [BBCGGI21, Appendix C.4].
    verify_rand_prg = Poplar1.Idpf.Prg(verify_key,
        Poplar1.custom(DST_VERIFY_RAND),
        nonce + I2OSP(level, 2))
    verify_rand = verify_rand_prg.next_vec(Field, len(prefixes))
    sketch_share = [a_share, b_share, c_share]
    out_share = []
    for (i, r) in enumerate(verify_rand):
        (data_share, auth_share) = value[i]
        sketch_share[0] += data_share * r
        sketch_share[1] += data_share * r^2
        sketch_share[2] += auth_share * r
        out_share.append(data_share)
```

```
    prep_mem = sketch_share \
                + [A_share, B_share, Field(agg_id)] \
                + out_share
    return (b'ready', level, prep_mem)

def prep_next(Poplar1, prep_state, opt_sketch):
    (step, level, prep_mem) = prep_state
    Field = Poplar1.Idpf.current_field(level)

    # Aggregators exchange masked input values (step (3.)
    # of [BBCGGI21, Appendix C.4]).
    if step == b'ready' and opt_sketch == None:
        sketch_share, prep_mem = prep_mem[:3], prep_mem[3:]
        return ((b'sketch round 1', level, prep_mem),
                Field.encode_vec(sketch_share))

    # Aggregators exchange evaluated shares (step (4.)).
    elif step == b'sketch round 1' and opt_sketch != None:
        prev_sketch = Field.decode_vec(opt_sketch)
        if len(prev_sketch) == 0:
            prev_sketch = Field.zeros(3)
        elif len(prev_sketch) != 3:
            raise ERR_INPUT # prep message malformed
        (A_share, B_share, agg_id), prep_mem = \
            prep_mem[:3], prep_mem[3:]
        sketch_share = [
            agg_id * (prev_sketch[0]^2 \
                        - prev_sketch[1]
                        - prev_sketch[2]) \
                + A_share * prev_sketch[0] \
                + B_share
        ]
        return ((b'sketch round 2', level, prep_mem),
                Field.encode_vec(sketch_share))

    elif step == b'sketch round 2' and opt_sketch != None:
        prev_sketch = Field.decode_vec(opt_sketch)
        if len(prev_sketch) == 0:
            prev_sketch = Field.zeros(1)
        elif len(prev_sketch) != 1:
            raise ERR_INPUT # prep message malformed
        if prev_sketch[0] != Field(0):
            raise ERR_VERIFY
        return prep_mem # Output shares

    raise ERR_INPUT # unexpected input

def prep_shares_to_prep(Poplar1, agg_param, prep_shares):
    if len(prep_shares) != 2:
```

```
    raise ERR_INPUT # unexpected number of prep shares
(level, _) = agg_param
Field = Poplar1.Idpf.current_field(level)
sketch = vec_add(Field.decode_vec(prep_shares[0]),
                 Field.decode_vec(prep_shares[1]))
if sketch == Field.zeros(len(sketch)):
    # In order to reduce communication overhead, let the
    # empty string denote the zero vector of the required
    # length.
    return b''
return Field.encode_vec(sketch)
```

Figure 20: Preparation state for Poplar1.

8.2.3. Validity of Aggregation Parameters

Aggregation parameters are valid for a given input share if no aggregation parameter with the same level has been used with the same input share before. The whole preparation phase **MUST NOT** be run more than once for a given combination of input share and level.

```
def is_valid(agg_param, previous_agg_params):
    (level, _) = agg_param
    return all(
        level != other_level
        for (other_level, _) in previous_agg_params
    )
```

Figure 21: Validity of aggregation parameters for Poplar1.

8.2.4. Aggregation

Aggregation involves simply adding up the output shares.

```
def out_shares_to_agg_share(Poplar1, agg_param, out_shares):
    (level, prefixes) = agg_param
    Field = Poplar1.Idpf.current_field(level)
    agg_share = Field.zeros(len(prefixes))
    for out_share in out_shares:
        agg_share = vec_add(agg_share, out_share)
    return Field.encode_vec(agg_share)
```

Figure 22: Aggregation algorithm for Poplar1.

8.2.5. Unsharding

Finally, the Collector unshards the aggregate result by adding up the aggregate shares.

```
def agg_shares_to_result(Poplar1, agg_param,
                         agg_shares, _num_measurements):
    (level, prefixes) = agg_param
    Field = Poplar1.Idpf.current_field(level)
    agg = Field.zeros(len(prefixes))
    for agg_share in agg_shares:
        agg = vec_add(agg, Field.decode_vec(agg_share))
    return list(map(lambda x: x.as_unsigned(), agg))
```

Figure 23: Computation of the aggregate result for Poplar1.

8.2.6. Auxiliary Functions

8.2.6.1. Message Serialization

This section defines methods for serializing input shares, as required by the Vdaf interface. Optional serialization of the aggregation parameter is also specified below.

Implementation note: The aggregation parameter includes the level of the IDPF tree and the sequence of indices to evaluate. For implementations that perform per-report caching across executions of the VDAF, this may be more information than is strictly needed. In particular, it may be sufficient to convey which indices from the previous execution will have their children included in the next. This would help reduce communication overhead.

```
def encode_input_shares(Poplar1, keys,
                        corr_seed, corr_inner, corr_leaf):
    input_shares = []
    for (key, seed, inner, leaf) in zip(keys,
                                        corr_seed,
                                        corr_inner,
                                        corr_leaf):
        encoded = Bytes()
        encoded += key
        encoded += seed
        encoded += Poplar1.Idpf.FieldInner.encode_vec(inner)
        encoded += Poplar1.Idpf.FieldLeaf.encode_vec(leaf)
        input_shares.append(encoded)
    return input_shares

def decode_input_share(Poplar1, encoded):
    l = Poplar1.Idpf.KEY_SIZE
    key, encoded = encoded[:l], encoded[l:]
    l = Poplar1.Idpf.Prg.SEED_SIZE
    corr_seed, encoded = encoded[:l], encoded[l:]
    l = Poplar1.Idpf.FieldInner.ENCODED_SIZE \
        * 2 * (Poplar1.Idpf.BITS - 1)
    encoded_corr_inner, encoded = encoded[:l], encoded[l:]
    corr_inner = Poplar1.Idpf.FieldInner.decode_vec(
        encoded_corr_inner)
    l = Poplar1.Idpf.FieldLeaf.ENCODED_SIZE * 2
    encoded_corr_leaf, encoded = encoded[:l], encoded[l:]
    corr_leaf = Poplar1.Idpf.FieldLeaf.decode_vec(
        encoded_corr_leaf)
    if len(encoded) != 0:
        raise ERR_INPUT
    return (key, corr_seed, corr_inner, corr_leaf)

def encode_agg_param(Poplar1, level, prefixes):
    if level > 2^16 - 1:
        raise ERR_INPUT # level too deep
    if len(prefixes) > 2^32 - 1:
        raise ERR_INPUT # too many prefixes
    encoded = Bytes()
    encoded += I2OSP(level, 2)
    encoded += I2OSP(len(prefixes), 4)
    packed = 0
    for (i, prefix) in enumerate(prefixes):
        packed |= prefix << ((level+1) * i)
    l = floor(((level+1) * len(prefixes) + 7) / 8)
    encoded += I2OSP(packed, l)
    return encoded

def decode_agg_param(Poplar1, encoded):
```

```
encoded_level, encoded = encoded[:2], encoded[2:]
level = OS2IP(encoded_level)
encoded_prefix_count, encoded = encoded[:4], encoded[4:]
prefix_count = OS2IP(encoded_prefix_count)
l = floor(((level+1) * prefix_count + 7) / 8)
encoded_packed, encoded = encoded[:l], encoded[l:]
packed = OS2IP(encoded_packed)
prefixes = []
m = 2^(level+1) - 1
for i in range(prefix_count):
    prefixes.append(packed >> ((level+1) * i) & m)
if len(encoded) != 0:
    raise ERR_INPUT
return (level, prefixes)
```

8.3. The IDPF scheme of *[BBCGGI21]*

TODO(issue#32) Consider replacing the generic Prg object here with some fixed-key mode for AES (something along the lines of ia.cr/2019/074). This would allow us to take advantage of hardware acceleration, which would significantly improve performance. We use SHA-3 primarily to instantiate random oracles, but the random oracle model may not be required for IDPF. More investigation is needed.

In this section we specify a concrete IDPF, called IdpfPoplar, suitable for instantiating Poplar1. The scheme gets its name from the name of the protocol of [BBCGGI21].

TODO We should consider giving IdpfPoplar a more distinctive name.

The constant and type definitions required by the Idpf interface are given in Table 17.

Parameter	**Value**
SHARES	2
BITS	any positive integer
VALUE_LEN	any positive integer
KEY_SIZE	Prg.SEED_SIZE
FieldInner	Field64 (Table 3)
FieldLeaf	Field255 (Table 5)
Prg	any implementation of Prg (Section 6.2)

Table 17: Constants and type definitions for IdpfPoplar.

8.3.1. Key Generation

TODO Describe the construction in prose, beginning with a gentle introduction to the high level idea.

The description of the IDPF-key generation algorithm makes use of auxiliary functions extend(), convert(), and encode_public_share() defined in Section 8.3.3. In the following, we let Field2 denote the field GF(2).

```
def gen(IdpfPoplar, alpha, beta_inner, beta_leaf):
    if alpha >= 2^IdpfPoplar.BITS:
        raise ERR_INPUT # alpha too long
    if len(beta_inner) != IdpfPoplar.BITS - 1:
        raise ERR_INPUT # beta_inner vector is the wrong size

    init_seed = [
        gen_rand(IdpfPoplar.Prg.SEED_SIZE),
        gen_rand(IdpfPoplar.Prg.SEED_SIZE),
    ]

    seed = init_seed.copy()
    ctrl = [Field2(0), Field2(1)]
    correction_words = []
    for level in range(IdpfPoplar.BITS):
        Field = IdpfPoplar.current_field(level)
        keep = (alpha >> (IdpfPoplar.BITS - level - 1)) & 1
        lose = 1 - keep
        bit = Field2(keep)

        (s0, t0) = IdpfPoplar.extend(seed[0])
        (s1, t1) = IdpfPoplar.extend(seed[1])
        seed_cw = xor(s0[lose], s1[lose])
        ctrl_cw = (
            t0[0] + t1[0] + bit + Field2(1),
            t0[1] + t1[1] + bit,
        )

        x0 = xor(s0[keep], ctrl[0].conditional_select(seed_cw))
        x1 = xor(s1[keep], ctrl[1].conditional_select(seed_cw))
        (seed[0], w0) = IdpfPoplar.convert(level, x0)
        (seed[1], w1) = IdpfPoplar.convert(level, x1)
        ctrl[0] = t0[keep] + ctrl[0] * ctrl_cw[keep]
        ctrl[1] = t1[keep] + ctrl[1] * ctrl_cw[keep]

        b = beta_inner[level] if level < IdpfPoplar.BITS-1 \
                else beta_leaf
        if len(b) != IdpfPoplar.VALUE_LEN:
            raise ERR_INPUT # beta too long or too short

        w_cw = vec_add(vec_sub(b, w0), w1)
        # Implementation note: Here we negate the correction word if
        # the control bit `ctrl[1]` is set. We avoid branching on the
        # value in order to reduce leakage via timing side channels.
        mask = Field(1) - Field(2) * Field(ctrl[1].as_unsigned())
        for i in range(len(w_cw)):
            w_cw[i] *= mask

        correction_words.append((seed_cw, ctrl_cw, w_cw))
```

```
public_share = IdpfPoplar.encode_public_share(correction_words)
return (public_share, init_seed)
```

Figure 24: IDPF-key generation algorithm of IdpfPoplar.

8.3.2. Key Evaluation

TODO Describe in prose how IDPF-key evaluation algorithm works.

The description of the IDPF-evaluation algorithm makes use of auxiliary functions extend(), convert(), and decode_public_share() defined in Section 8.3.3.

```
def eval(IdpfPoplar, agg_id, public_share, init_seed,
         level, prefixes):
    if agg_id >= IdpfPoplar.SHARES:
        raise ERR_INPUT # invalid aggregator ID
    if level >= IdpfPoplar.BITS:
        raise ERR_INPUT # level too deep
    if len(set(prefixes)) != len(prefixes):
        raise ERR_INPUT # candidate prefixes are non-unique

    correction_words = IdpfPoplar.decode_public_share(public_share)
    out_share = []
    for prefix in prefixes:
        if prefix >= 2^(level+1):
            raise ERR_INPUT # prefix too long

        # The Aggregator's output share is the value of a node of
        # the IDPF tree at the given `level`. The node's value is
        # computed by traversing the path defined by the candidate
        # `prefix`. Each node in the tree is represented by a seed
        # (`seed`) and a set of control bits (`ctrl`).
        seed = init_seed
        ctrl = Field2(agg_id)
        for current_level in range(level+1):
            bit = (prefix >> (level - current_level)) & 1

            # Implementation note: Typically the current round of
            # candidate prefixes would have been derived from
            # aggregate results computed during previous rounds. For
            # example, when using `IdpfPoplar` to compute heavy
            # hitters, a string whose hit count exceeded the given
            # threshold in the last round would be the prefix of each
            # `prefix` in the current round. (See [BBCGGI21,
            # Section 5.1].) In this case, part of the path would
            # have already been traversed.
            #
            # Re-computing nodes along previously traversed paths is
            # wasteful. Implementations can eliminate this added
            # complexity by caching nodes (i.e., `(seed, ctrl)`
            # pairs) output by previous calls to `eval_next()`.
            (seed, ctrl, y) = IdpfPoplar.eval_next(seed, ctrl,
                correction_words[current_level], current_level, bit)
        out_share.append(y if agg_id == 0 else vec_neg(y))
    return out_share

# Compute the next node in the IDPF tree along the path determined by
# a candidate prefix. The next node is determined by `bit`, the bit
# of the prefix corresponding to the next level of the tree.
#
# TODO Consider implementing some version of the optimization
```

```
# discussed at the end of [BBCGGI21, Appendix C.2]. This could on
# average reduce the number of AES calls by a constant factor.
def eval_next(IdpfPoplar, prev_seed, prev_ctrl,
              correction_word, level, bit):
    Field = IdpfPoplar.current_field(level)
    (seed_cw, ctrl_cw, w_cw) = correction_word
    (s, t) = IdpfPoplar.extend(prev_seed)
    s[0] = xor(s[0], prev_ctrl.conditional_select(seed_cw))
    s[1] = xor(s[1], prev_ctrl.conditional_select(seed_cw))
    t[0] += ctrl_cw[0] * prev_ctrl
    t[1] += ctrl_cw[1] * prev_ctrl

    next_ctrl = t[bit]
    (next_seed, y) = IdpfPoplar.convert(level, s[bit])
    # Implementation note: Here we add the correction word to the
    # output if `next_ctrl` is set. We avoid branching on the value of
    # the control bit in order to reduce side channel leakage.
    mask = Field(next_ctrl.as_unsigned())
    for i in range(len(y)):
        y[i] += w_cw[i] * mask

    return (next_seed, next_ctrl, y)
```

Figure 25: IDPF-evaluation generation algorithm of IdpfPoplar.

8.3.3. Auxiliary Functions

```
def extend(IdpfPoplar, seed):
    prg = IdpfPoplar.Prg(seed, format_custom(1, 0, 0), b'')
    s = [
        prg.next(IdpfPoplar.Prg.SEED_SIZE),
        prg.next(IdpfPoplar.Prg.SEED_SIZE),
    ]
    b = OS2IP(prg.next(1))
    t = [Field2(b & 1), Field2((b >> 1) & 1)]
    return (s, t)

def convert(IdpfPoplar, level, seed):
    prg = IdpfPoplar.Prg(seed, format_custom(1, 0, 1), b'')
    next_seed = prg.next(IdpfPoplar.Prg.SEED_SIZE)
    Field = IdpfPoplar.current_field(level)
    w = prg.next_vec(Field, IdpfPoplar.VALUE_LEN)
    return (next_seed, w)

def encode_public_share(IdpfPoplar, correction_words):
    encoded = Bytes()
    control_bits = list(itertools.chain.from_iterable(
        cw[1] for cw in correction_words
    ))
    encoded += pack_bits(control_bits)
    for (level, (seed_cw, _, w_cw)) \
        in enumerate(correction_words):
        Field = IdpfPoplar.current_field(level)
        encoded += seed_cw
        encoded += Field.encode_vec(w_cw)
    return encoded

def decode_public_share(IdpfPoplar, encoded):
    l = floor((2*IdpfPoplar.BITS + 7) / 8)
    encoded_ctrl, encoded = encoded[:l], encoded[l:]
    control_bits = unpack_bits(encoded_ctrl, 2 * IdpfPoplar.BITS)
    correction_words = []
    for level in range(IdpfPoplar.BITS):
        Field = IdpfPoplar.current_field(level)
        ctrl_cw = (
            control_bits[level * 2],
            control_bits[level * 2 + 1],
        )
        l = IdpfPoplar.Prg.SEED_SIZE
        seed_cw, encoded = encoded[:l], encoded[l:]
        l = Field.ENCODED_SIZE * IdpfPoplar.VALUE_LEN
        encoded_w_cw, encoded = encoded[:l], encoded[l:]
        w_cw = Field.decode_vec(encoded_w_cw)
        correction_words.append((seed_cw, ctrl_cw, w_cw))
    leftover_bits = encoded_ctrl[-1] >> (
        ((IdpfPoplar.BITS + 3) % 4 + 1) * 2
```

```
)
if leftover_bits != 0 or len(encoded) != 0:
    raise ERR_DECODE
return correction_words
```

Figure 26: Helper functions for IdpfPoplar.

Here, pack_bits() takes a list of bits, packs each group of eight bits into a byte, in LSB to MSB order, padding the most significant bits of the last byte with zeros as necessary, and returns the byte array. unpack_bits() performs the reverse operation: it takes in a byte array and a number of bits, and returns a list of bits, extracting eight bits from each byte in turn, in LSB to MSB order, and stopping after the requested number of bits. If the byte array has an incorrect length, or if unused bits in the last bytes are not zero, it throws an error.

8.4. Instantiation

By default, Poplar1 is instantiated with IdpfPoplar (VALUE_LEN == 2) and PrgSha3 (Section 6.2.1). This VDAF is suitable for any positive value of BITS. Test vectors can be found in Appendix "Test Vectors".

9. Security Considerations

VDAFs have two essential security goals:

1. Privacy: An attacker that controls the network, the Collector, and a subset of Clients and Aggregators learns nothing about the measurements of honest Clients beyond what it can deduce from the aggregate result.

2. Robustness: An attacker that controls the network and a subset of Clients cannot cause the Collector to compute anything other than the aggregate of the measurements of honest Clients.

Formal definitions of privacy and robustness can be found in [DPRS23]. A VDAF is the core cryptographic primitive of a protocol that achieves the above privacy and robustness goals. It is not sufficient on its own, however. The application will need to assure a few security properties, for example:

* Securely distributing the long-lived parameters, in particular the verification key.

* Establishing secure channels:

  - Confidential and authentic channels among Aggregators, and between the Aggregators and the Collector; and

  - Confidential and Aggregator-authenticated channels between Clients and Aggregators.

* Enforcing the non-collusion properties required of the specific VDAF in use.

In such an environment, a VDAF provides the high-level privacy property described above: The Collector learns only the aggregate measurement, and nothing about individual measurements aside from what can be inferred from the aggregate result. The Aggregators learn neither individual measurements nor the aggregate result. The Collector is assured that the aggregate statistic accurately reflects the inputs as long as the Aggregators correctly executed their role in the VDAF.

On their own, VDAFs do not mitigate Sybil attacks [Dou02]. In this attack, the adversary observes a subset of input shares transmitted by a Client it is interested in. It allows the input shares to be processed, but corrupts and picks bogus measurements for the remaining Clients. Applications can guard against these risks by adding additional controls on report submission, such as client authentication and rate limits.

VDAFs do not inherently provide differential privacy [Dwo06]. The VDAF approach to private measurement can be viewed as complementary to differential privacy, relying on non-collusion instead of statistical noise to protect the privacy of the inputs. It is possible that a future VDAF could incorporate differential privacy features, e.g., by injecting noise before the sharding stage and removing it after unsharding.

9.1. Requirements for the Verification Key

The Aggregators are responsible for exchanging the verification key in advance of executing the VDAF. Any procedure is acceptable as long as the following conditions are met:

1. To ensure robustness of the computation, the Aggregators **MUST NOT** reveal the verification key to the Clients. Otherwise, a malicious Client might be able to exploit knowledge of this key to craft an invalid report that would be accepted by the Aggregators.

2. To ensure privacy of the measurements, the Aggregators **MUST** commit to the verification key prior to processing reports generated by Clients. Otherwise, a malicious Aggregator may be able to craft a verification key that, for a given report, causes an honest Aggregator to leak information about the measurement during preparation.

Meeting these conditions is required in order to leverage security analysis in the framework of [DPRS23]. Their definition of robustness allows the attacker, playing the role of a cohort of malicious Clients, to submit arbitrary reports to the Aggregators and eavesdrop on their communications as they process them. Security

in this model is achievable as long as the verification key is kept secret from the attacker.

The privacy definition of [DPRS23] considers an active attacker that controls the network and a subset of Aggregators; in addition, the attacker is allowed to choose the verification key used by each honest Aggregator over the course of the experiment. Security is achievable in this model as long as the key is picked at the start of the experiment, prior to any reports being generated. (The model also requires nonces to be generated at random; see Section 9.2 below.)

Meeting these requirements is relatively straightforward. For example, the Aggregators may designate one of their peers to generate the verification key and distribute it to the others. To assure Clients of key commitment, the Clients and (honest) Aggregators could bind reports to a shared context string derived from the key. For instance, the "task ID" of DAP [DAP] could be set to the hash of the verification key; then as long as honest Aggregators only consume reports for the task indicated by the Client, forging a new key after the fact would reduce to finding collisions in the underlying hash function. (Keeping the key secret from the Clients would require the hash function to be one-way.) However, since rotating the key implies rotating the task ID, this scheme would not allow key rotation over the lifetime of a task.

9.2. Requirements for the Nonce

The sharding and preparation steps of VDAF execution depend on a nonce associated with the Client's report. To ensure privacy of the underlying measurement, the Client **MUST** generate this nonce using a CSPRNG. This is required in order to leverage security analysis for the privacy definition of [DPRS23], which assumes the nonce is chosen at random prior to generating the report.

Other security considerations may require the nonce to be non-repeating. For example, to achieve differential privacy it is necessary to avoid "over exposing" a measurement by including it too many times in a single batch or across multiple batches. It is **RECOMMENDED** that the nonce generated by the Client be used by the Aggregators for replay protection.

9.3. Requirements for the Aggregation Parameters

As described in Section 4.3 and Section 5.3 respectively, DAFs and VDAFs may impose restrictions on the re-use of input shares. This is to ensure that correlated randomness provided by the Client through the input share is not used more than once, which might compromise confidentiality of the Client's measurements.

Protocols that make use of VDAFs therefore **MUST** call Vdaf.is_valid on the set of all aggregation parameters used for a Client's input share, and only proceed with the preparation and aggregation phases if that function call returns True.

10. IANA Considerations

A codepoint for each (V)DAF in this document is defined in the table below. Note that 0xFFFF0000 through 0xFFFFFFFF are reserved for private use.

Value	Scheme	Type	Reference
0x00000000	Prio3Count	VDAF	Section 7.4.1
0x00000001	Prio3Sum	VDAF	Section 7.4.2
0x00000002	Prio3Histogram	VDAF	Section 7.4.3
0x00000003 to 0x00000FFF	reserved for Prio3	VDAF	n/a
0x00001000	Poplar1	VDAF	Section 8.4
0xFFFF0000 to 0xFFFFFFFF	reserved	n/a	n/a

Table 18: Unique identifiers for (V)DAFs.

TODO Add IANA considerations for the codepoints summarized in Table 18.

OPEN ISSUE Currently the scheme includes the PRG. This means that we need bits of the codepoint to differentiate between PRGs. We could instead make the PRG generic (e.g., Prio3Count(Aes128) instead of Prio3Aes128Count) and define a separate codepoint.

11. References

11.1. Normative References

[FIPS202] "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", NIST FIPS PUB 202 , August 2015.

[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>.

[RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <https://www.rfc-editor.org/rfc/rfc8017>.

[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>.

[SP800-185]
"SHA-3 Derived Functions: cSHAKE, KMAC, TupleHash and ParallelHash", NIST Special Publication 800-185 , December 2016.

11.2. Informative References

[AGJOP21] Addanki, S., Garbe, K., Jaffe, E., Ostrovsky, R., and A. Polychroniadou, "Prio+: Privacy Preserving Aggregate Statistics via Boolean Shares", 2021, <https://ia.cr/2021/576>.

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

[BBCGGI21] Boneh, D., Boyle, E., Corrigan-Gibbs, H., Gilboa, N., and Y. Ishai, "Lightweight Techniques for Private Heavy Hitters", IEEE S&P 2021 , 2021, <https://ia.cr/2021/017>.

[CGB17] Corrigan-Gibbs, H. and D. Boneh, "Prio: Private, Robust, and Scalable Computation of Aggregate Statistics", NSDI 2017 , 2017, <https://dl.acm.org/doi/10.5555/3154630.3154652>.

[DAP] Geoghegan, T., Patton, C., Rescorla, E., and C. A. Wood, "Distributed Aggregation Protocol for Privacy Preserving Measurement", Work in Progress, Internet-Draft, draft-ietf-ppm-dap-03, 9 December 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-ppm-dap-03>.

[Dou02] Douceur, J., "The Sybil Attack", IPTPS 2002 , 2002, <https://doi.org/10.1007/3-540-45748-8_24>.

[DPRS23] Davis, H., Patton, C., Rosulek, M., and P. Schoppmann, "Verifiable Distributed Aggregation Functions", n.d., <https://ia.cr/2023/130>.

[Dwo06] Dwork, C., "Differential Privacy", ICALP 2006 , 2006, <https://link.springer.com/chapter/10.1007/11787006_1>.

[ENPA] "Exposure Notification Privacy-preserving Analytics (ENPA) White Paper", 2021, <https://covid19-static.cdn-apple.com/applications/covid19/current/static/contact-tracing/pdf/ENPA_White_Paper.pdf>.

[EPK14] Erlingsson, Ú., Pihur, V., and A. Korolova, "RAPPOR: Randomized Aggregatable Privacy-Preserving Ordinal

Response", CCS 2014 , 2014, <https://dl.acm.org/doi/10.1145/2660267.2660348>.

[GI14] Gilboa, N. and Y. Ishai, "Distributed Point Functions and Their Applications", EUROCRYPT 2014 , 2014, <https://link.springer.com/chapter/10.1007/978-3-642-55220-5_35>.

[OriginTelemetry] "Origin Telemetry", 2020, <https://firefox-source-docs.mozilla.org/toolkit/components/telemetry/collection/origin.html>.

Acknowledgments

The security considerations in Section 9 are based largely on the security analysis of [DPRS23]. Thanks to Hannah Davis and Mike Rosulek, who lent their time to developing definitions and security proofs.

Thanks to Henry Corrigan-Gibbs, Armando Faz-Hernández, Simon Friedberger, Tim Geoghegan, Mariana Raykova, Jacob Rothstein, and Christopher Wood for useful feedback on and contributions to the spec.

Test Vectors

> NOTE Machine-readable test vectors can be found at https://github.com/cfrg/draft-irtf-cfrg-vdaf/tree/main/poc/test_vec.

Test vectors cover the generation of input shares and the conversion of input shares into output shares. Vectors specify the verification key, measurements, aggregation parameter, and any parameters needed to construct the VDAF. (For example, for Prio3Sum, the user specifies the number of bits for representing each summand.)

Byte strings are encoded in hexadecimal To make the tests deterministic, gen_rand() was replaced with a function that returns the requested number of 0x01 octets.

Prio3Count


```
verify_key: "01010101010101010101010101010101"
upload_0:
  measurement: 1
  nonce: "01010101010101010101010101010101"
  public_share: >-
  input_share_0: >-
    d8700d17841945acb2891eaadb82b14c71098bcf4a708cf5bfff34c19f1fcb389524
    33062a90142b69f3de7e58566e86
  input_share_1: >-
    0101010101010101010101010101010101010101010101010101010101010101
  round_0:
    prep_share_0: >-
      bc5ed32148ccdfe8149c04ac5b2e26bbb74c2c01d26b8e9a4f26452740c8b6e9
    prep_share_1: >-
      43a12cddb7332019fa93469841dbb1aba89845e53745d13445a31850c99c4ea8
    prep_message: >-
  out_share_0:
    - d8700d17841945ac
  out_share_1:
    - 278ff2e77be6ba56
agg_share_0: >-
  d8700d17841945ac
agg_share_1: >-
  278ff2e77be6ba56
agg_result: 1
```

Prio3Sum

```
bits: 8
verify_key: "01010101010101010101010101010101"
upload_0:
  measurement: 100
  nonce: "01010101010101010101010101010101"
  public_share: >-
    905f744fdd73d539b852697c311bcc977e2eef2723dabfbae02c93ce1f196ae8
  input_share_0: >-
    9129dfc509b86e7947f238dd725ae3cc8393f640c44fd72eb0dd39bf28fbbf8e9455
    f069ca3089119d684ae29c280e954061350c04c90916f27895fe374ebd6badc69ad0
    e1fda3e02a85750c9a27b4f34a4dd09ce0c2cef5599c0a0a40c2c3fdea329eaa925f
    a1a65b7f947e9ec0b395412e56245f4d9fa5993f3e6a7263d71d73ac5dbb78a3104f
    23f0c67893b390b1b14a6211c54651645548d145213de5c87acd21d6218248991394
    708b9d823c9d9d7e16996f32d4578c2ecec9b0c2673c4d1ee156ffac2415369c1583
    21482ba0d8fa021f3c00185a99c74d619637f463e6d5bb22bb45cf3d43a9ba332148
    a61b42c26567563b8d9d612a8e85be68dd542838a2e1bd03e471ccf5514e333b3837
    0a41c8df749dd43a86ee2708c1797dd10fb2d655700c3d74e4c9a9726eafa553035d
    846162f981925e83d8dac7f08784339c3b89f54461ae764239bb521b6e3419ebe8d5
    6638968479a53a9ec5453b3f94dad0e3b41e71bc273222733428a18f3e66bdb42081
    55aa11215ce9c91bbf7664e10992cc2d8c30aa0abf524b6fc545a599a43b66d676a0
    2be73e6fc8c24e0ad17eceba6e081d8d8e7882eae163cc2afcc1e76cb7b3936168e0
    a71ce3f73999e7769ec43108075657cc36f5108a0b1713777bc6b561fed05dd0869d
    596c8cc079628e94440df17b63486d5b1e6af79dd1378f45bbdf68497d064e0c9f63
    b768bdd7e0761ec441b1a529e742d4e13f4755cdc99569236592dae907aab4ba5304
    cb7224469fb8bd76f2b348026d5d6312fc8c52f891461b08c5b05fdae2e1c745ce94
    63b92efb59a12cca026c6afb195322ef69c5eb452788742075e41231991d324f24f6
    21f6b79d6da2472438b455379856f7623ca9a4419d61eababea02634010101010101
    01010101010101010101
  input_share_1: >-
    0101010101010101010101010101010101010101010101010101010101010101010101
    0101010101010101010101010101
  round_0:
    prep_share_0: >-
      b11ad355381629c48921bae8ff502c7e9b214980f51d58fd3b7293542ddb43f83f
      f15e16a0d7b38ed62357f1dbae34d4905f744fdd73d539b852697c311bcc97
    prep_share_1: >-
      4ee52caac7e9d61f76de451700afd383f622613281a91f1113882cd7646c207484
      da515e63d02452fe32316648fd84277e2eef2723dabfbae02c93ce1f196ae8
    prep_message: >-
      fab4374d7e6549b8ae110e76b4eb315d
  out_share_0:
    - 36a9b3cc584eefb20e6f6eca8c54caaf
  out_share_1:
    - c9564c33a7b11031f190913573ab35b6
agg_share_0: >-
  36a9b3cc584eefb20e6f6eca8c54caaf
agg_share_1: >-
  c9564c33a7b11031f190913573ab35b6
agg_result: 100
```

Prio3Histogram


```
buckets: [1, 10, 100]
verify_key: "01010101010101010101010101010101"
upload_0:
  measurement: 50
  nonce: "01010101010101010101010101010101"
  public_share: >-
    d995029335b0240dbdca1cbfb1211ecc67fa8edb6f0f74882a626d290baa199c
  input_share_0: >-
    22d711113962f5ffe2f3119db262a76940036dd11c6ac7a0ea844da70ed1cb575ada
    ee861cb1075340a455a13bbadd6270e2ab78349dbf2f7aca71d9a77efb01ccff433b
    195d4e1f143bd7bb6f3502dde041a22ef5c262c67f79bc61f0364dedcd549842a047
    2c4edf77a3860b96444fd00e20d1e9b6fe42013e7104b608d1c7f5c5752f4f95b614
    de92bc377c1b7ee8c96bf3de4eca547a691b86463b94e94d0b5b44bcfc47a54cd0f9
    6391e9559ab065af59873f4a8cf9dd55597c02245c4dd5343414855127a44b579b40
    e5fa3a941bb0b65cd4403291d03ed043aae2a0ec43b0950a7d1c28fc80090f5cb834
    f8df88c7dbdfa1e7503aef1d2a1a3d41de25091ddb2984b5a89d4a576564a93fb529
    5844aa9360be4b164f8ab9c064a640afea30eaef8ee4522b1b3340d0819f4ad40f3f
    809b746f999c34c40564fd87448001010101010101010101010101010101
  input_share_1: >-
    0101010101010101010101010101010101010101010101010101010101010101010101
    01010101010101010101010101010101
  round_0:
    prep_share_0: >-
      2d3f3200d6afb340b09dfc30b92ea4146ad4f5e2677515ed1c4e6ebcc8b6809ef5
      762a4ee715e20e05b219f8529dbfc1d995029335b0240dbdca1cbfb1211ecc
    prep_share_1: >-
      d2c0cdff29504ca34f6203cf46d15bed9083e4170397a5da2d55754b9a7bbf3803
      0a54eebf540cd5ff5e6b07d451ce7067fa8edb6f0f74882a626d290baa199c
    prep_message: >-
      c3885ffa3a60ba7f6413596d328cfd27
  out_share_0:
    - 22d711113962f5ffe2f3119db262a769
    - 40036dd11c6ac7a0ea844da70ed1cb57
    - 5adaee861cb1075340a455a13bbadd62
    - 70e2ab78349dbf2f7aca71d9a77efb01
  out_share_1:
    - dd28eeeec69d09e41d0cee624d9d5898
    - bffc922ee3953843157bb258f12e34aa
    - a5251179e34ef890bf5baa5ec44522a0
    - 8f1d5487cb6240b485358e2658810500
agg_share_0: >-
  22d711113962f5ffe2f3119db262a76940036dd11c6ac7a0ea844da70ed1cb575adaee
  861cb1075340a455a13bbadd6270e2ab78349dbf2f7aca71d9a77efb01
agg_share_1: >-
  dd28eeeec69d09e41d0cee624d9d5898bffc922ee3953843157bb258f12e34aaa52511
  79e34ef890bf5baa5ec44522a08f1d5487cb6240b485358e2658810500
agg_result: [0, 0, 1, 0]
```

Poplar1

Sharding

bits: 4
verify_key: "01010101010101010101010101010101"
agg_param: (0, [0, 1])
upload_0:
  measurement: 13
  nonce: "01010101010101010101010101010101"
  public_share: >-
    9a00000000000000000000000000000000ffffffff000000002925acd3fdadd65000
    000000000000000000000000000000ffffffff00000000a30477d49c635897000000
    00000000000000000000000000ffffffff00000000be1923f14c176f080000000000
    000000000000000000000007ffffffffffffffffffffffffffffffffffffffffffff
    ffffffffffffffffec2f05a6f445eda5468c2b3ee83fd3031e06bef0befa154c75ef
    08d68e559203fc
  input_share_0: >-
    01010101010101010101010101010101010101010101010101010101010101012829
    0055e1399e093ce757ec6febc3bc89a955aab0739c173a2d54cfe984f90bc9170ed9
    fa7be3d005feb7b32d1d87867f965e8e09e8dedee5fead6898a353e42d243e7b64e6
    db53a4d1e004d7dd23880d06ea70ceb5fc414eb742e39ddd30ca69689aff04bee551
    8956f318ccd148a8
  input_share_1: >-
    0101010101010101010101010101010101010101010101010101010101010101b84f
    b52b53729cff7f30537d822e188c24f328a9cbcf8a0c5dec0ba7935e426335334440
    0524965ab9925911875b5354328f383f015d117fda7cc41787a4ee1596864e0b15e3
    529a3834658b103b521e1d2a46275be79a4a5766d3bd9e7d22fe000404c9866dd3ec
    a0b381328f8305a3

Preparation, Aggregation, and Unsharding

```
verify_key: "01010101010101010101010101010101"
agg_param: (0, [0, 1])
upload_0:
  round_0:
    prep_share_0: >-
      45d4d76583ff3d4bfadb0a1a136a6df5139a62a553367029
    prep_share_1: >-
      a3f8aa69980f13c2749bf9b3c23e21e19e39198fb3f8e2ed
    prep_message: >-
      e9cd81cf1c0e510d6f7703ced5a88fd5b1d37c35072f5316
  round_1:
    prep_share_0: >-
      23568c2a012e01c9
    prep_share_1: >-
      dca973d4fed1fe38
    prep_message: >-
  out_share_0:
    - fcf3039f72d70136
    - 7a930346dcd33ac4
  out_share_1:
    - 030cfc5f8d28fecb
    - 856cfcb8232cc53e
agg_share_0: >-
  fcf3039f72d701367a930346dcd33ac4
agg_share_1: >-
  030cfc5f8d28fecb856cfcb8232cc53e
agg_result: [0, 1]
```

```
verify_key: "01010101010101010101010101010101"
agg_param: (1, [0, 1, 2, 3])
upload_0:
  round_0:
    prep_share_0: >-
      7ff2aa9b62d4699bb13b5d89054dc718ef0ef340d31e71e7
    prep_share_1: >-
      152587e270e3ad1f7166f605136db06284feef9e0c60ec4a
    prep_message: >-
      9518327dd3b816ba22a2538f18bb7779740de2dfdf7f5e30
  round_1:
    prep_share_0: >-
      edbf9effc5523ad6
    prep_share_1: >-
      124060ff3aadc52b
    prep_message: >-
  out_share_0:
    - e6b4675d5645143d
    - 6751646bdd854e4f
    - 782cd9dd3b220cbe
    - 07af78ced3d6e89f
  out_share_1:
    - 194b98a1a9baebc4
    - 98ae9b93227ab1b2
    - 87d32621c4ddf343
    - f85087302c291763
agg_share_0: >-
  e6b4675d5645143d6751646bdd854e4f782cd9dd3b220cbe07af78ced3d6e89f
agg_share_1: >-
  194b98a1a9baebc498ae9b93227ab1b287d32621c4ddf343f85087302c291763
agg_result: [0, 0, 0, 1]
```

```
verify_key: "01010101010101010101010101010101"
agg_param: (2, [0, 2, 4, 6])
upload_0:
  round_0:
    prep_share_0: >-
      6d956ea3b01727f03e3c4b1cad1461c3fae8fc676763d8ef
    prep_share_1: >-
      9b8bfdb242f67c7aed868f197e62d46f4450ac65b089a5b6
    prep_message: >-
      09216c56f30da4692bc2da372b7736313f39a8ce17ed7ea4
  round_1:
    prep_share_0: >-
      a22d3f45a76cf4f3
    prep_share_1: >-
      5dd2c0b958930b0e
    prep_message: >-
  out_share_0:
    - bb4d5c8b44fea90f
    - 01c9989f7fa25aef
    - 46f355d9bc179269
    - 50600db413c684ce
  out_share_1:
    - 44b2a373bb0156f2
    - fe36675f805da512
    - b90caa2543e86d98
    - af9ff24aec397b34
agg_share_0: >-
  bb4d5c8b44fea90f01c9989f7fa25aef46f355d9bc17926950600db413c684ce
agg_share_1: >-
  44b2a373bb0156f2fe36675f805da512b90caa2543e86d98af9ff24aec397b34
agg_result: [0, 0, 0, 1]
```

```
verify_key: "01010101010101010101010101010101"
agg_param: (3, [1, 3, 5, 7, 9, 13, 15])
upload_0:
  round_0:
    prep_share_0: >-
      21acceb054f9fe345aa353803f2debb5a307dad06b26dc6a7c7b79a1a665448b20
      88a892dad00cba3ab6ab9b9283b734937b281810100158daca6188dbbc4e155e3a
      0697f2291a6f723b6085dedf9a23817a554358e9f191aaa2a2425c938238
    prep_share_1: >-
      11fbc7788b2224efb1c760840c8ef96f400e6762e8bcfa000823120f2fb12e3838
      7ab34acd8cbbb78016048aaf737e167c8b7660581df35eb1af7b95de07d29526de
      fc2f23c59aff034e7feec503ae2e064db27e2c0f57ce905cfba9fcc3dab0
    prep_message: >-
      33a89628e01c23240c6ab4044bbce524e316423353e3d66a849e8bb0d61672c359
      035bdda85cc871baccb02641f7354b10069e78682df4b78c79dd1eb9c420aa0519
      02c715eeb56e7589e074a3e3485187c807c184f949603aff9dec59575cfb
  round_1:
    prep_share_0: >-
      3dc137d56f1e1eb3f09ea7300d66f972d74efd3f1efd667bd6079dc9b16179d6
    prep_share_1: >-
      423ec82a90e1e14c0f6158cff299068d28b102c0e102998429f862364e9e8617
    prep_message: >-
  out_share_0:
    - 5ea259ad47a621cc04bc4804ecc9d302ccd2225b37e82770912154e4c0495f6b
    - 6f8fb25f78167a93352cd07ca7bd6ea4f12f457b2968a62f4fc721fd12946030
    - 066e1fbff795f0eab7023023e8662c0d01804d169505f1f58bb005b1ae8d7906
    - 241a08089dd8e7c318987d598d3fe8cb2c7870b6d3a46228307f5cc50ee1b81b
    - 16d9502d5eb5ba9bd5e5c55e0385b35ee3a944585225848fd5cb82ce78ea0e77
    - 2ccf7b85827ba848284a28c82e5b8bb9c591b4b21628f374c3fc2cf79e3e8560
    - 5e16bb7dbf1ca16bf038dbc8702e4a1ee79074856c05d742e3d43e6f94d8022c
  out_share_1:
    - 215da652b859de33fb43b7fb13362cfd332ddda4c817d88f6edeab1b3fb6a082
    - 10704da087e9856ccad32f835842915b0ed0ba84d69759d0b038de02ed6b9fbd
    - 7991e040086a0f1548fdcfdc1799d3f2fe7fb2e96afa0e0a744ffa4e517286e7
    - 5be5f7f76227183ce76782a672c01734d3878f492c5b9dd7cf80a33af11e47d2
    - 6926afd2a14a45642a1a3aa1fc7a4ca11c56bba7adda7b702a347d318715f176
    - 5330847a7d8457b7d7b5d737d1a474463a6e4b4de9d70c8b3c03d30861c17a8e
    - 21e9448240e35e940fc724378fd1b5e1186f8b7a93fa28bd1c2bc1906b27fdc1
agg_share_0: >-
  5ea259ad47a621cc04bc4804ecc9d302ccd2225b37e82770912154e4c0495f6b6f8fb2
  5f78167a93352cd07ca7bd6ea4f12f457b2968a62f4fc721fd12946030066e1fbff795
  f0eab7023023e8662c0d01804d169505f1f58bb005b1ae8d7906241a08089dd8e7c318
  987d598d3fe8cb2c7870b6d3a46228307f5cc50ee1b81b16d9502d5eb5ba9bd5e5c55e
  0385b35ee3a944585225848fd5cb82ce78ea0e772ccf7b85827ba848284a28c82e5b8b
  b9c591b4b21628f374c3fc2cf79e3e85605e16bb7dbf1ca16bf038dbc8702e4a1ee790
  74856c05d742e3d43e6f94d8022c
agg_share_1: >-
  215da652b859de33fb43b7fb13362cfd332ddda4c817d88f6edeab1b3fb6a08210704d
  a087e9856ccad32f835842915b0ed0ba84d69759d0b038de02ed6b9fbd7991e040086a
```

```
  0f1548fdcfdc1799d3f2fe7fb2e96afa0e0a744ffa4e517286e75be5f7f76227183ce7
  6782a672c01734d3878f492c5b9dd7cf80a33af11e47d26926afd2a14a45642a1a3aa1
  fc7a4ca11c56bba7adda7b702a347d318715f1765330847a7d8457b7d7b5d737d1a474
  463a6e4b4de9d70c8b3c03d30861c17a8e21e9448240e35e940fc724378fd1b5e1186f
  8b7a93fa28bd1c2bc1906b27fdc1
agg_result: [0, 0, 0, 0, 0, 1, 0]
```

Authors' Addresses

Richard L. Barnes
Cisco

Email: rlb@ipv.sx

David Cook
ISRG

Email: divergentdave@gmail.com

Christopher Patton
Cloudflare

Email: chrispatton+ietf@gmail.com

Phillipp Schoppmann
Google

Email: schoppmann@google.com