Workgroup: CFRG
Internet-Draft: draft-irtf-cfrg-vdaf-08
Published: 20 November 2023
Intended Status: Informational
Expires: 23 May 2024
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 measurement that would result in an invalid 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 23 May 2024.

Table of Contents

1. Introduction

[TO BE REMOVED BY RFC EDITOR: The source for this draft and and the reference implementation can be found at https://github.com/cfrg/draft-irtf-cfrg-vdaf.]

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 measurement before submitting to the aggregation server. The aggregation server then adds up the noisy measurements, 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 measurement, it may be possible for a curious aggregator to make a reasonable

guess of the measurement'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 measurement 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 measurement is ever accessible to any party besides the client that submitted it. At the same time, VDAFs are "verifiable" in the sense that malformed measurements that would otherwise garble the result of the computation can be detected and removed from the set of measurements. We refer to this property as "robustness".

In addition to these MPC-style security goals of privacy and robustness, 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 requirements, 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, documenting relevant operational and security requirements, and specifying constraints for safe usage:

    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, aggregatable value, e.g., the measurement 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 strings 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 string. 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 string.

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.

08:

* Poplar1: Bind the report nonce to the authenticator vector programmed into the IDPF. (*)

* IdpfPoplar: Modify extend() by stealing each control bit from its corresponding seed. This improves performance by reducing the number of AES calls per level from 3 to 2. The cost is a slight reduction in the concrete privacy bound. (*)

* Prio3: Add support for generating and verifying mutliple proofs per measurement. This enables a trade-off between communication cost and runtime: if more proofs are used, then a smaller field can be used without impacting robustness. (*)

* Replace SHAKE128 with TurboSHAKE128. (*)

07:

* Rename PRG to XOF ("eXtendable Output Function"). Accordingly, rename PrgSha3 to XofShake128 and PrgFixedKeyAes128 to XofFixedKeyAes128. "PRG" is a misnomer since we don't actually treat this object as a pseudorandom generator in existing security analysis.

* Replace cSHAKE128 with SHAKE128, re-implementing domain separation for the customization string using a simpler scheme. This change addresses the reality that implementations of cSHAKE128 are less common. (*)

* Define a new VDAF, called Prio3SumVec, that generalizes Prio3Sum to a vector of summands.

* Prio3Histogram: Update the codepoint and use the parallel sum optimization introduced by Prio3SumVec to reduce the proof size. (*)

* Daf, Vdaf: Rename interface methods to match verbiage in the draft.

* Daf: Align with Vdaf by adding a nonce to shard() and prep().

* Vdaf: Have prep_init() compute the first prep share. This change is intended to simplify the interface by making the input to prep_next() not optional.

* Prio3: Split sharding into two auxiliary functions, one for sharding with joint randomness and another without. This change is intended to improve readability.

* Fix bugs in the ping-pong interface discovered after implementing it.

06:

* Vdaf: Define a wrapper interface for preparation that is suitable for the "ping-pong" topology in which two Aggregators exchange messages over a request/response protocol, like HTTP, and take turns executing the computation until input from the peer is required.

* Prio3Histogram: Generalize the measurement type so that the histogram can be used more easily with discrete domains. (*)

* Daf, Vdaf: Change the aggregation parameter validation algorithm to take the set of previous parameters rather than a list. (The order of the parameters is irrelevant.)

* Daf, Vdaf, Idpf: Add parameter RAND_SIZE that specifies the number of random bytes consumed by the randomized algorithm (shard() for Daf and Vdaf and gen() for Idpf).

05:

* IdpfPoplar: Replace PrgSha3 with PrgFixedKeyAes128, a fixed-key mode for AES-128 based on a construction from [GKWWY20]. This change is intended to improve performance of IDPF evaluation. Note that the new PRG is not suitable for all applications. (*)

* Idpf: Add a binder string to the key-generation and evaluation algorithms. This is used to plumb the nonce generated by the Client to the PRG.

* Plumb random coins through the interface of randomized algorithms. Specifically, add a random input to (V)DAF sharding algorithm and IDPF key-generation algorithm and require implementations to specify the length of the random input. Accordingly, update Prio3, Poplar1, and IdpfPoplar to match the new interface. This change is intended to improve coverage of test vectors.

* Use little-endian byte-order for field element encoding. (*)

* Poplar1: Move the last step of sketch evaluation from prep_next() to prep_shares_to_prep().

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 8.

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].

* front(length: Unsigned, vec: Vec[Any]) -> (Vec[Any], Vec[Any]) splits vec into two vectors, where the first vector is made up of the first length elements of the input. I.e., (vec[:length], vec[length:]).

* 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.

* to_be_bytes(val: Unsigned, length: Unsigned) -> Bytes converts val to big-endian bytes; its value **MUST** be in range [0, 2^(8*length)). Function from_be_bytes(encoded: Bytes) -> Unsigned computes the inverse.

* to_le_bytes(val: Unsigned, length: Unsigned) -> Bytes converts val to little-endian bytes; its value **MUST** be in range [0, 2^(8*length)). Function from_le_bytes(encoded: Bytes) -> Unsigned computes the inverse.

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

* additive_secret_share(vec: Vec[Field], num_shares: Unsigned, field: type) -> Vec[Vec[Field]] takes a vector of field elements and returns multiple vectors of the same length, such that they all add up to the input vector, and each proper subset of the vectors are indistinguishable from random.

3. Overview

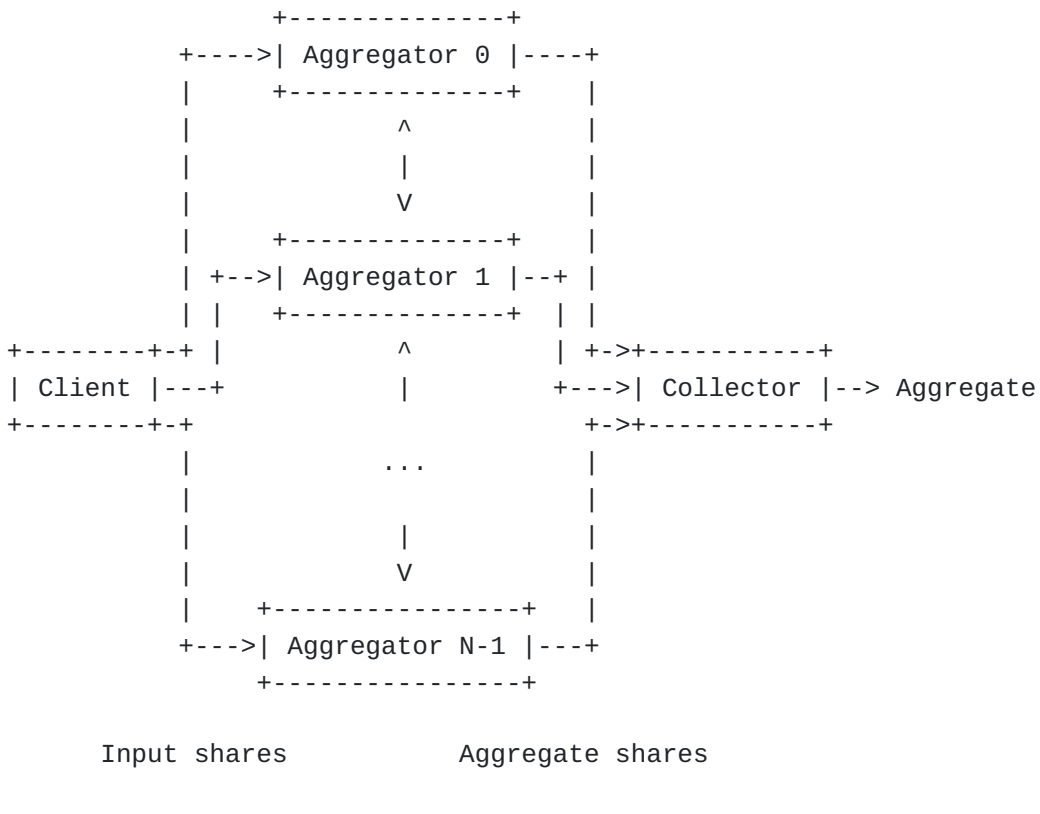

Figure 1: Overall data flow of a (V)DAF

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 refine 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 measurement, if this process succeeds, then 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 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. A 32-bit, unsigned integer.
SHARES	Number of input shares into which each measurement is sharded.
NONCE_SIZE	Size of the nonce passed by the application.
RAND_SIZE	Size of the random byte string passed to sharding algorithm.
Measurement	Type of each measurement.
PublicShare	Type of each public share.
InputShare	Type of each input share.
AggParam	Type of aggregation parameter.
OutShare	Type of each output share.
AggShare	Type of the aggregate share.
AggResult	Type of the aggregate result.

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

These types define the inputs and outputs of DAF methods at various stages of the computation. Some of these values need to be written to the network in order to carry out the computation. In particular, it is **RECOMMENDED** that concrete instantiations of the Daf interface

specify a method of encoding the PublicShare, InputShare, and AggShare.

Each DAF is identified by a unique, 32-bit integer ID. Identifiers for each (V)DAF specified in this document are defined in Table 17.

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 shard method is used for this purpose.

*Daf.shard(measurement: Measurement, nonce: bytes[Daf.NONCE_SIZE], rand: bytes[Daf.RAND_SIZE]) -> tuple[PublicShare, list[InputShare]] is the randomized sharding algorithm run by each Client. The input rand consists of the random bytes consumed by the algorithm. This value **MUST** be generated using a cryptographically secure pseudorandom number generator (CSPRNG). 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.

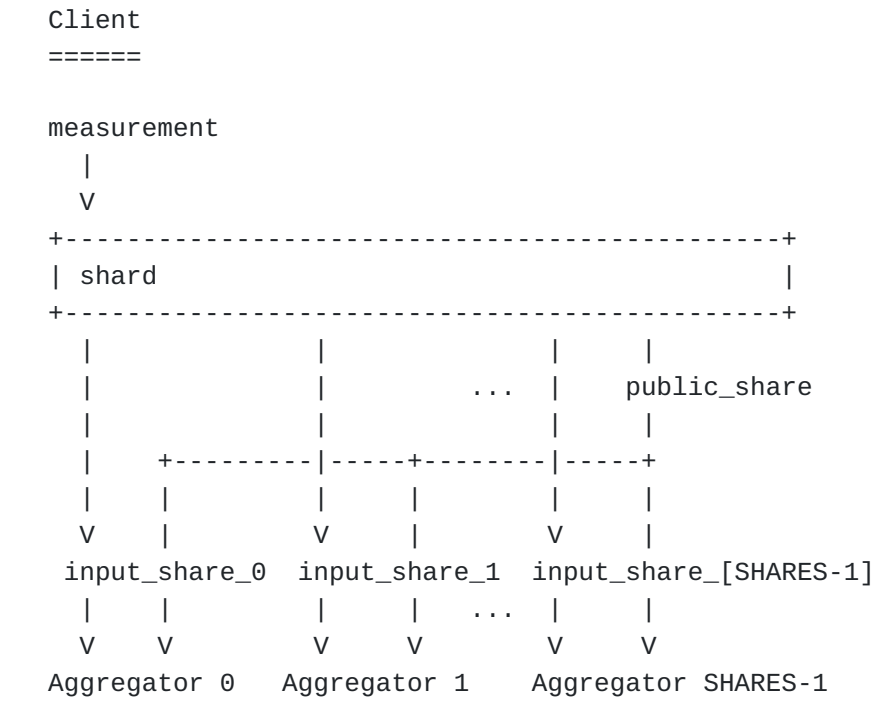

Figure 2: The Client divides its measurement into input shares and distributes them to the Aggregators. The public share is broadcast to all Aggregators.

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, nonce: bytes[NONCE_SIZE], public_share: PublicShare, input_share: InputShare) -> 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, the aggregation parameter selected by the Collector, and a nonce 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: set[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.aggregate(agg_param: AggParam, out_shares: list[OutShare]) -> AggShare is the deterministic aggregation algorithm. It is run by each Aggregator a set of recovered output shares.

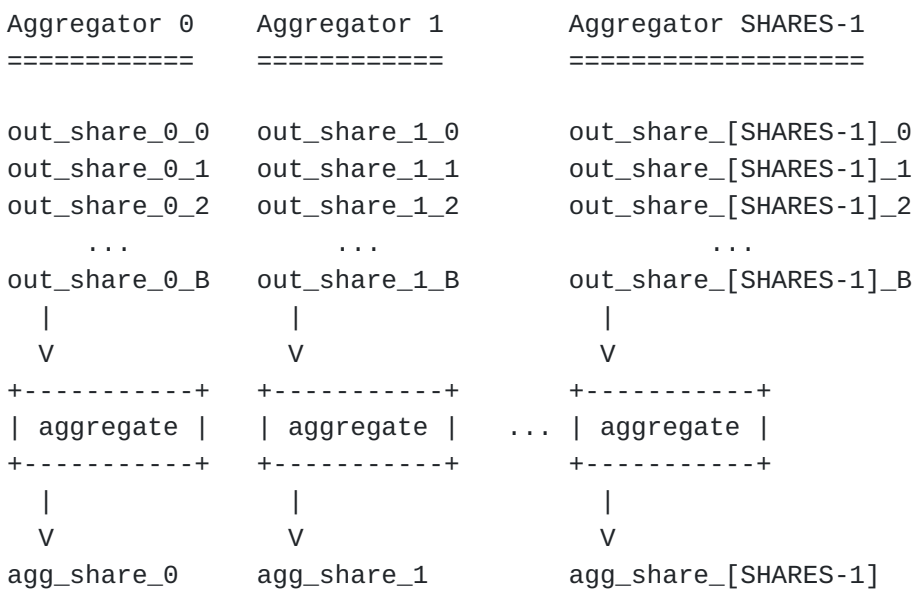

Figure 3: Aggregation of output shares. `B` indicates the number of measurements in the batch.

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.

Implementation note: For most natural DAFs (and VDAFs) it is not necessary for an Aggregator to store all output shares individually before aggregating. Typically it is possible to merge output shares into aggregate shares as they arrive, merge these into other aggregate shares, and so on. In particular, this is the case when the output shares are vectors over some finite field and aggregating them involves merely adding up the vectors element-wise. Such is the case for Prio3 Section 7 and Poplar1 Section 8.

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.unshard(agg_param: AggParam, agg_shares: list[AggShare], 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.

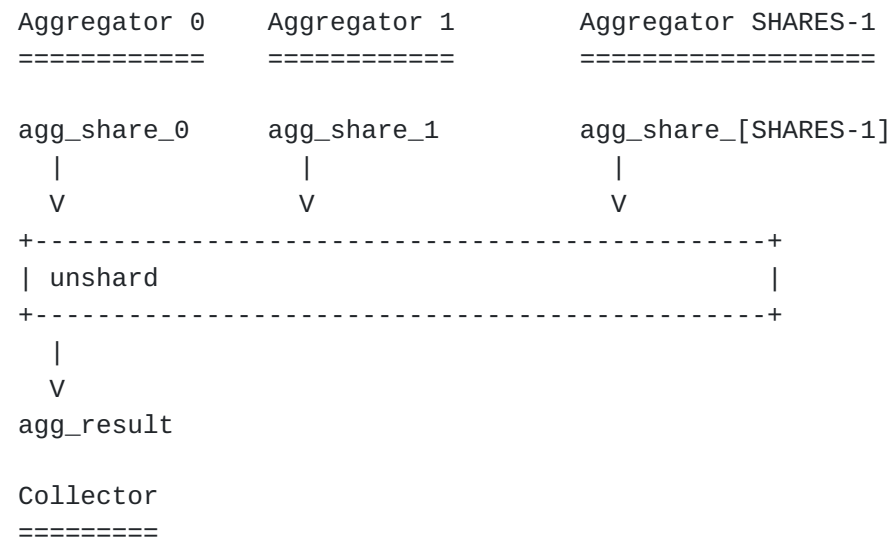

Figure 4: Computation of the final aggregate result from aggregate shares.

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: list[Daf.Measurement],
            nonces: list[bytes[Daf.NONCE_SIZE]]):
    out_shares = [[] for j in range(Daf.SHARES)]
    for (measurement, nonce) in zip(measurements, nonces):
        # Each Client shards its measurement into input shares and
        # distributes them among the Aggregators.
        rand = gen_rand(Daf.RAND_SIZE)
        (public_share, input_shares) = \
            Daf.shard(measurement, nonce, rand)

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

    # Collector unshards the aggregate result.
    num_measurements = len(measurements)
    agg_result = Daf.unshard(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).
RAND_SIZE	Size of the random byte string passed to sharding algorithm.
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.
PublicShare	Type of each public share.
InputShare	Type of each input share.
AggParam	Type of aggregation parameter.

Parameter	Description
OutShare	Type of each output share.
AggShare	Type of the aggregate share.
AggResult	Type of the aggregate result.
PrepState	Aggregator's state during preparation.
PrepShare	Type of each prep share.
PrepMessage	Type of each prep message.

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

Some of these values need to be written to the network in order to carry out the computation. In particular, it is **RECOMMENDED** that concrete instantiations of the Vdaf interface specify a method of encoding the PublicShare, InputShare, AggShare, PrepShare, and PrepMessage.

Each VDAF is identified by a unique, 32-bit integer ID. Identifiers for each (V)DAF specified in this document are defined in Table 17. The following method is defined for every VDAF:

```
def domain_separation_tag(Vdaf, usage: Unsigned) -> Bytes:
    """
    Format domain separation tag for this VDAF with the given usage.
    """
    return format_dst(0, Vdaf.ID, usage)
```

It is used to construct a domain separation tag for an instance of Xof 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.shard(measurement: Measurement, nonce: bytes[Vdaf.NONCE_SIZE], rand: bytes[Vdaf.RAND_SIZE]) -> tuple[PublicShare, list[InputShare]] is the randomized sharding algorithm run by each Client. Input rand consists of the random bytes consumed by the algorithm. 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 random bytes and nonce using a 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", or "prep message" for short. (Each of the outbound messages are called "preparation-message shares", or "prep shares" for short.) 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 prep message. Its output in the last round is its output share and its output in each of the preceding rounds is a prep 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 the measurement shares for aggregation.

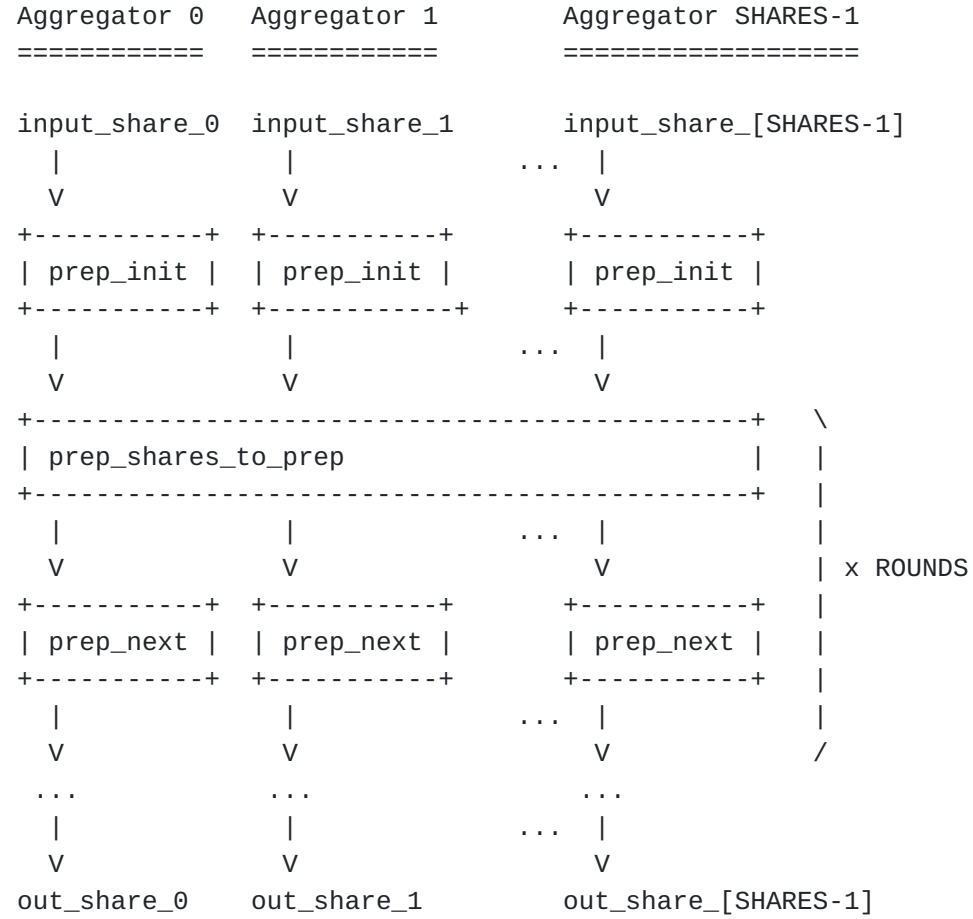

Figure 6: VDAF preparation process on the input shares for a single measurement. At the end of the computation, each Aggregator holds an output share or an error.

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

* Vdaf.prep_init(verify_key: bytes[Vdaf.VERIFY_KEY_SIZE], agg_id: Unsigned, agg_param: AggParam, nonce: bytes[Vdaf.NONCE_SIZE], public_share: PublicShare, input_share: InputShare) -> tuple[PrepState, PrepShare] 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 and initial prep share.

  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.

* Vdaf.prep_next(prep_state: PrepState, prep_msg: PrepMessage) -> Union[tuple[PrepState, PrepShare], OutShare] is the deterministic preparation-state update algorithm run by each Aggregator. It updates the Aggregator's preparation state (prep_state) 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.

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

In effect, each Aggregator moves through a linear state machine with ROUNDS 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: set[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.aggregate(agg_param: AggParam, out_shares: list[OutShare]) -> AggShare is the deterministic aggregation algorithm. It is run by each Aggregator over the output shares it has computed for a batch of measurements.

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.unshard(agg_param: AggParam, agg_shares: list[AggShare], 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: list[bytes[Vdaf.NONCE_SIZE]],
             measurements: list[Vdaf.Measurement]):
    out_shares = []
    for (nonce, measurement) in zip(nonces, measurements):
        # Each Client shards its measurement into input shares.
        rand = gen_rand(Vdaf.RAND_SIZE)
        (public_share, input_shares) = \
            Vdaf.shard(measurement, nonce, rand)

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

        # Aggregators recover their output shares.
        for i in range(Vdaf.ROUNDS-1):
            prep_msg = Vdaf.prep_shares_to_prep(agg_param,
                                                outbound)
            outbound = []
            for j in range(Vdaf.SHARES):
                out = Vdaf.prep_next(prep_states[j], prep_msg)
                (prep_states[j], out) = out
                outbound.append(out)

        # The final outputs of the prepare phase are the output shares.
        prep_msg = Vdaf.prep_shares_to_prep(agg_param,
                                            outbound)
        outbound = []
        for j in range(Vdaf.SHARES):
            out_share = Vdaf.prep_next(prep_states[j], prep_msg)
            outbound.append(out_share)
        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.aggregate(agg_param, out_shares_j)
    agg_shares.append(agg_share_j)

# Collector unshards the aggregate.
num_measurements = len(measurements)
agg_result = Vdaf.unshard(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.

5.7. Communication Patterns for Preparation

In each round of preparation, each Aggregator writes a prep share to some broadcast channel, which is then processed into the prep message using the public prep_shares_to_prep() algorithm and broadcast to the Aggregators to start the next round. In this section we describe some approaches to realizing this broadcast channel functionality in protocols that use VDAFs.

The state machine of each Aggregator is shown in Figure 8.

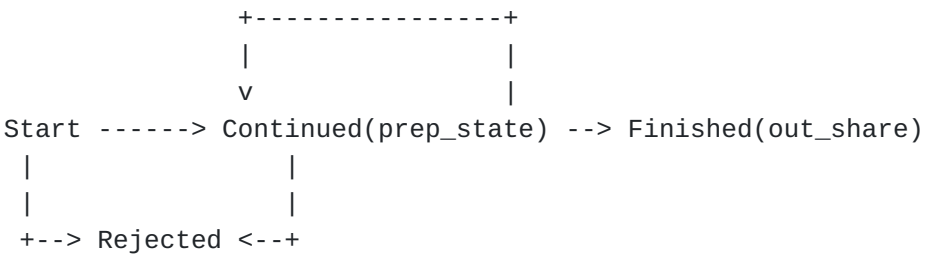

Figure 8: State machine for VDAF preparation.

State transitions are made when the state is acted upon by the host's local inputs and/or messages sent by the peers. The initial state is Start. The terminal states are Rejected, which indicates that the report cannot be processed any further, and Finished(out_share), which indicates that the Aggregator has recovered an output share out_share.

```
class State:
    pass

class Start(State):
    pass

class Continued(State):
    def __init__(self, prep_state):
        self.prep_state = prep_state

class Finished(State):
    def __init__(self, output_share):
        self.output_share = output_share

class Rejected(State):
    def __init__(self):
        pass
```

Note that there is no representation of the Start state as it is never instantiated in the ping-pong topology.

For convenience, the methods described in this section are defined in terms of opaque byte strings. A compatible Vdaf **MUST** specify methods for encoding public shares, input shares, prep shares, prep messages, and aggregation parameters. Minimally:

- *Vdaf.decode_public_share(encoded: bytes) -> Vdaf.PublicShare decodes a public share.

- *Vdaf.decode_input_share(agg_id: Unsigned, encoded: bytes) -> Vdaf.InputShare decodes an input share, using the aggregator ID as optional context.

- *Vdaf.encode_prep_share(prep_share: Vdaf.PrepShare) -> bytes encodes a prep share.

- *Vdaf.decode_prep_share(prep_state: Vdaf.PrepState, encoded: bytes) -> Vdaf.PrepShare decodes a prep share, using the prep state as optional context.

- *Vdaf.encode_prep_msg(prep_msg: Vdaf.PrepMessage) -> bytes encodes a prep message.

- *Vdaf.decode_prep_msg(prep_state: Vdaf.PrepState, encoded: bytes) -> Vdaf.PrepMessage decodes a prep message, using the prep state as optional decoding context.

- *Vdaf.decode_agg_param(encoded: bytes) -> Vdaf.AggParam decodes an aggregation parameter.

* Vdaf.encode_agg_param(agg_param: Vdaf.AggParam) -> bytes encodes an aggregation parameter.

Implementations of Prio3 and Poplar1 **MUST** use the encoding scheme specified in Section 7.2.7 and Section 8.2.6 respectively.

5.8. Ping-Pong Topology (Only Two Aggregators)

For VDAFs with precisely two Aggregators (i.e., Vdaf.SHARES == 2), the following "ping pong" communication pattern can be used. It is compatible with any request/response transport protocol, such as HTTP.

Let us call the initiating party the "Leader" and the responding party the "Helper". The high-level idea is that the Leader and Helper will take turns running the computation locally until input from their peer is required:

* For a 1-round VDAF (e.g., Prio3 (Section 7)), the Leader sends its prep share to the Helper, who computes the prep message locally, computes its output share, then sends the prep message to the Leader. Preparation requires just one round trip between the Leader and the Helper.

* For a 2-round VDAF (e.g., Poplar1 (Section 8)), the Leader sends its first-round prep share to the Helper, who replies with the first-round prep message and its second-round prep share. In the next request, the Leader computes its second-round prep share locally, computes its output share, and sends the second-round prep message to the Helper. Finally, the Helper computes its own output share.

* In general, each request includes the Leader's prep share for the previous round and/or the prep message for the current round; correspondingly, each response consists of the prep message for the current round and the Helper's prep share for the next round.

The Aggregators proceed in this ping-ponging fashion until a step of the computation fails (indicating the report is invalid and should be rejected) or preparation is completed. All told there there are ceil((Vdaf.ROUNDS+1)/2) requests sent.

Each message in the ping-pong protocol is structured as follows (expressed in TLS syntax as defined in Section 3 of [RFC8446]):

```
enum {
  initialize(0),
  continue(1),
  finish(2),
  (255)
} MessageType;

struct {
  MessageType type;
  select (Message.type) {
    case initialize:
      opaque prep_share<0..2^32-1>;
    case continue:
      opaque prep_msg<0..2^32-1>;
      opaque prep_share<0..2^32-1>;
    case finish:
      opaque prep_msg<0..2^32-1>;
  };
} Message;
```

These messages are used to transition between the states described in Section 5.7. They are encoded and decoded to or from byte buffers as described Section 3 of [RFC8446]) using the following routines:

* encode_ping_pong_message(message: Message) -> bytes encodes a Message into an opaque byte buffer.

* decode_pong_pong_message(encoded: bytes) -> Message decodes an opaque byte buffer into a Message, raising an error if the bytes are not a valid encoding.

The Leader's initial transition is computed with the following procedure:

```
def ping_pong_leader_init(
            Vdaf,
            vdaf_verify_key: bytes[Vdaf.VERIFY_KEY_SIZE],
            agg_param: bytes,
            nonce: bytes[Vdaf.NONCE_SIZE],
            public_share: bytes,
            input_share: bytes,
        ) -> tuple[State, bytes]:
    try:
        (prep_state, prep_share) = Vdaf.prep_init(
            vdaf_verify_key,
            0,
            Vdaf.decode_agg_param(agg_param),
            nonce,
            Vdaf.decode_public_share(public_share),
            Vdaf.decode_input_share(0, input_share),
        )
        outbound = Message.initialize(
            Vdaf.encode_prep_share(prep_share))
        return (Continued(prep_state), encode_ping_pong_message(outbound
    except:
        return (Rejected(), None)
```

The output is the State to which the Leader has transitioned and an encoded Message. If the Leader's state is Rejected, then processing halts. Otherwise, if the state is Continued, then processing continues.

The Leader sends the outbound message to the Helper. The Helper's initial transition is computed using the following procedure:

```
def ping_pong_helper_init(
            Vdaf,
            vdaf_verify_key: bytes[Vdaf.VERIFY_KEY_SIZE],
            agg_param: bytes,
            nonce: bytes[Vdaf.NONCE_SIZE],
            public_share: bytes,
            input_share: bytes,
            inbound_encoded: bytes,
        ) -> tuple[State, bytes]:
    try:
        (prep_state, prep_share) = Vdaf.prep_init(
            vdaf_verify_key,
            1,
            Vdaf.decode_agg_param(agg_param),
            nonce,
            Vdaf.decode_public_share(public_share),
            Vdaf.decode_input_share(1, input_share),
        )

        inbound = decode_ping_pong_message(inbound_encoded)

        if inbound.type != 0: # initialize
            return (Rejected(), None)

        prep_shares = [
            Vdaf.decode_prep_share(prep_state, inbound.prep_share),
            prep_share,
        ]
        return Vdaf.ping_pong_transition(
            agg_param,
            prep_shares,
            prep_state,
        )
    except:
        return (Rejected(), None)
```

Procedure ping_pong_transition() takes in the prep shares, combines them into the prep message, and computes the next prep state of the caller:

```
def ping_pong_transition(
            Vdaf,
            agg_param: Vdaf.AggParam,
            prep_shares: list[Vdaf.PrepShare],
            prep_state: Vdaf.PrepState,
         ) -> (State, bytes):
    prep_msg = Vdaf.prep_shares_to_prep(agg_param,
                                        prep_shares)
    out = Vdaf.prep_next(prep_state, prep_msg)
    if type(out) == Vdaf.OutShare:
        outbound = Message.finish(Vdaf.encode_prep_msg(prep_msg))
        return (Finished(out), encode_ping_pong_message(outbound))
    (prep_state, prep_share) = out
    outbound = Message.continue(
        Vdaf.encode_prep_msg(prep_msg),
        Vdaf.encode_prep_share(prep_share),
    )
    return (Continued(prep_state), encode_ping_pong_message(outbound))
```

The output is the State to which the Helper has transitioned and an encoded Message. If the Helper's state is Finished or Rejected, then processing halts. Otherwise, if the state is Continued, then processing continues.

Next, the Helper sends the outbound message to the Leader. The Leader computes its next state transition using the function ping_pong_leader_continued:

```
def ping_pong_leader_continued(
            Vdaf,
            agg_param: bytes,
            state: State,
            inbound_encoded: bytes,
        ) -> (State, Optional[bytes]):
    return Vdaf.ping_pong_continued(
        True,
        agg_param,
        state,
        inbound_encoded,
    )

def ping_pong_continued(
            Vdaf,
            is_leader: bool,
            agg_param: bytes,
            state: State,
            inbound_encoded: bytes,
        ) -> (State, Optional[bytes]):
    try:
        inbound = decode_ping_pong_message(inbound_encoded)

        if inbound.type == 0: # initialize
            return (Rejected(), None)

        if !isinstance(state, Continued):
            return (Rejected(), None)

        prep_msg = Vdaf.decode_prep_msg(state.prep_state, inbound.prep_m
        out = Vdaf.prep_next(state.prep_state, prep_msg)
        if type(out) == tuple[Vdaf.PrepState, Vdaf.PrepShare] \
                and inbound.type == 1:
            # continue
            (prep_state, prep_share) = out
            prep_shares = [
                Vdaf.decode_prep_share(prep_state, inbound.prep_share),
                prep_share,
            ]
            if is_leader:
                prep_shares.reverse()
            return Vdaf.ping_pong_transition(
                Vdaf.decode_agg_param(agg_param),
                prep_shares,
                prep_state,
            )
        elif type(out) == Vdaf.OutShare and inbound.type == 2:
            # finish
            return (Finished(out), None)
```

```
    else:
        return (Rejected(), None)

except:
    return (Rejected(), None)
```

If the Leader's state is Finished or Rejected, then processing halts. Otherwise, the Leader sends the outbound message to the Helper. The Helper computes its next state transition using the function ping_pong_helper_continued:

```
def ping_pong_helper_continued(
            Vdaf,
            agg_param: bytes,
            state: State,
            inbound_encoded: bytes,
        ) -> (State, Optional[bytes]):
    return Vdaf.ping_pong_continued(
        False,
        agg_param,
        state,
        inbound_encoded,
    )
```

They continue in this way until processing halts. Note that, depending on the number of rounds of preparation that are required, there may be one more message to send before the peer can also finish processing (i.e., outbound != None).

5.9. Star Topology (Any Number of Aggregators)

The ping-pong topology of the previous section is only suitable for VDAFs involving exactly two Aggregators. If more Aggregators are required, the star topology described in this section can be used instead.

> TODO Describe the Leader-emulated broadcast channel architecture that was originally envisioned for DAP. (As of DAP-05 we are going with the ping pong architecture described in the previous section.)

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.

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 += to_le_bytes(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 = from_le_bytes(encoded_x)
        if x >= Field.MODULUS:
            raise ERR_DECODE # Integer is larger than modulus
        vec.append(Field(x))
    return vec
```

Figure 9: Derived class methods for finite fields.

Finally, Field implements the following methods for representing a value as a sequence of field elements, each of which represents a bit of the input.

```
def encode_into_bit_vector(Field,
                           val: Unsigned,
                           bits: Unsigned) -> Vec[Field]:
    """
    Encode the bit representation of `val` with at most `bits` number
    of bits, as a vector of field elements.
    """
    if val >= 2 ** bits:
        # Sanity check we are able to represent `val` with `bits`
        # number of bits.
        raise ValueError("Number of bits is not enough to represent "
                         "the input integer.")
    encoded = []
    for l in range(bits):
        encoded.append(Field((val >> l) & 1))
    return encoded

def decode_from_bit_vector(Field, vec: Vec[Field]) -> Field:
    """
    Decode the field element from the bit representation, expressed
    as a vector of field elements `vec`.
    """
    bits = len(vec)
    if Field.MODULUS >> bits == 0:
        raise ValueError("Number of bits is too large to be "
                         "represented by field modulus.")
    decoded = Field(0)
    for (l, bit) in enumerate(vec):
        decoded += Field(1 << l) * bit
    return decoded
```

Figure 10: Derived class methods to encode integers into bit vector representation.

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.

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

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

Figure 11: 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	Field64	Field128	Field255
MODULUS	2^32 * 4294967295 + 1	2^66 * 4611686018427387897 + 1	2^255 - 19
ENCODED_SIZE	8	16	32
Generator	7^4294967295	7^4611686018427387897	n/a
GEN_ORDER	2^32	2^66	n/a

Table 3: Parameters for the finite fields used in this document.

6.2. Extendable Output Functions

VDAFs in this specification use extendable output functions (XOFs) to extract short, fixed-length strings we call "seeds" from long input strings and expand seeds into long output strings. We specify a single interface that is suitable for both purposes.

XOFs are defined by a class Xof with the following associated parameter and methods:

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

- *Xof(seed: Bytes[Xof.SEED_SIZE], dst: Bytes, binder: Bytes) constructs an instance of Xof from the given seed, domain separation tag, and binder string. (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 XOF).

- *xof.next(length: Unsigned) returns the next length bytes of output of xof.

Each Xof has two derived methods. The first is used to derive a fresh seed from an existing one. The second is used to compute a sequence of field elements.

```
def derive_seed(Xof,
                seed: Bytes[Xof.SEED_SIZE],
                dst: Bytes,
                binder: Bytes):
    """Derive a new seed."""
    xof = Xof(seed, dst, binder)
    return xof.next(Xof.SEED_SIZE)

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

def expand_into_vec(Xof,
                    Field,
                    seed: Bytes[Xof.SEED_SIZE],
                    dst: Bytes,
                    binder: Bytes,
                    length: Unsigned):
    """
    Expand the input `seed` into vector of `length` field elements.
    """
    xof = Xof(seed, dst, binder)
    return xof.next_vec(Field, length)
```

Figure 12: Derived methods for XOFs.

6.2.1. XofTurboShake128

This section describes XofTurboShake128, an XOF based on the TurboSHAKE128 [TurboSHAKE]. This XOF is **RECOMMENDED** for all use cases within VDAFs. The length of the domain separation string dst passed to XofTurboShake128 **MUST NOT** exceed 255 bytes.

```
class XofTurboShake128(Xof):
    """XOF wrapper for TurboSHAKE128."""

    # Associated parameters
    SEED_SIZE = 16

    def __init__(self, seed, dst, binder):
        self.l = 0
        self.m = to_le_bytes(len(dst), 1) + dst + seed + binder

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

        # Function `TurboSHAKE128(M, D, L)` is as defined in
        # Section 2.2 of [TurboSHAKE].
        #
        # Implementation note: Rather than re-generate the output
        # stream each time `next()` is invoked, most implementations
        # of TurboSHAKE128 will expose an "absorb-then-squeeze" API that
        # allows stateful handling of the stream.
        stream = TurboSHAKE128(self.m, 1, self.l)
        return stream[-length:]
```

Figure 13: Definition of XOF XofTurboShake128.

6.2.2. XofFixedKeyAes128

While XofTurboShake128 as described above can be securely used in all cases where a XOF is needed in the VDAFs described in this document, there are some cases where a more efficient instantiation based on fixed-key AES is possible. For now, this is limited to the XOF used inside the Idpf Section 8.1 implementation in Poplar1 Section 8.3. It is **NOT RECOMMENDED** to use this XOF anywhere else. The length of the domain separation string dst passed to XofFixedKeyAes128 **MUST NOT** exceed 255 bytes. See Security Considerations Section 9 for a more detailed discussion.

```
class XofFixedKeyAes128(Xof):
    """
    XOF based on a circular collision-resistant hash function from
    fixed-key AES.
    """

    # Associated parameters
    SEED_SIZE = 16

    def __init__(self, seed, dst, binder):
        self.length_consumed = 0

        # Use TurboSHAKE128 to derive a key from the binder string and
        # domain separation tag. Note that the AES key does not need
        # to be kept secret from any party. However, when used with
        # IdpfPoplar, we require the binder to be a random nonce.
        #
        # Implementation note: This step can be cached across XOF
        # evaluations with many different seeds.
        dst_length = to_le_bytes(len(dst), 1)
        self.fixed_key = TurboSHAKE128(dst_length + dst + binder, 2, 16)
        self.seed = seed

    def next(self, length: Unsigned) -> Bytes:
        offset = self.length_consumed % 16
        new_length = self.length_consumed + length
        block_range = range(
            int(self.length_consumed / 16),
            int(new_length / 16) + 1)
        self.length_consumed = new_length

        hashed_blocks = [
            self.hash_block(xor(self.seed, to_le_bytes(i, 16))) \
                        for i in block_range
        ]
        return concat(hashed_blocks)[offset:offset+length]

    def hash_block(self, block):
        """
        The multi-instance tweakable circular correlation-robust hash
        function of [GKWWY20] (Section 4.2). The tweak here is the key
        that stays constant for all XOF evaluations of the same Client,
        but differs between Clients.

        Function `AES128(key, block)` is the AES-128 blockcipher.
        """
        lo, hi = block[:8], block[8:]
        sigma_block = concat([hi, xor(hi, lo)])
        return xor(AES128(self.fixed_key, sigma_block), sigma_block)
```

6.2.3. The Domain Separation Tag and Binder String

XOFs 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 domain separation tag. The domain separation tag 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 domain separation tag:

```
def format_dst(algo_class: Unsigned,
               algo: Unsigned,
               usage: Unsigned) -> Bytes:
    """Format XOF domain separation tag for use within a (V)DAF."""
    return concat([
        to_be_bytes(VERSION, 1),
        to_be_bytes(algo_class, 1),
        to_be_bytes(algo, 4),
        to_be_bytes(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.

*Measurement validity is determined by an arithmetic circuit evaluated over the encoded measurement. (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

measurement is said to be "valid"; otherwise, if the output is non-zero, then the measurement is said to be "invalid".

* The aggregate result is obtained by summing up the encoded measurement 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 measurement (as determined by the arithmetic circuit). This computation involves a "proof" of validity generated by the Client. Next, each Aggregator sums up its 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 measurement 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 measurements. 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 measurement's validity and distributes shares of the proof to the Aggregators. Each Aggregator then performs some computation on its measurement 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 measurement. (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 4.

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
MEAS_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 measurement and proof
Measurement	Type of the measurement
AggResult	Type of the aggregate result
Field	As defined in (Section 6.1)

Table 4: 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(meas: 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 measurement, 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(meas: 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 measurement and proof. The result of

the query (i.e., the output of this function) is called the "verifier message". In addition to the measurement 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 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 measurement 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 measurement and proof, the query-generation algorithm outputs a share of the verifier message. Furthermore, the privacy property of the FLP system ensures that the verifier message reveals nothing about the measurement other than whether it is valid. Therefore, to decide if a measurement 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 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 measurement, is the same regardless of the number of shares.

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

```
def run_flp(flp, meas: 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(meas, prove_rand, joint_rand)

    # Shard the measurement and the proof.
    meas_shares = additive_secret_share(meas, num_shares, flp.Field)
    proof_shares = additive_secret_share(proof, num_shares, flp.Field)

    # Verifier queries the meas shares and proof shares.
    verifier_shares = [
        flp.query(
            meas_share,
            proof_share,
            query_rand,
            joint_rand,
            num_shares,
        )
        for meas_share, proof_share in zip(meas_shares, proof_shares)
    ]

    # Combine the verifier shares into the verifier.
    verifier = flp.Field.zeros(len(verifier_shares[0]))
    for verifier_share in verifier_shares:
        verifier = vec_add(verifier, verifier_share)

    # Verifier decides if the measurement is valid.
    return flp.decide(verifier)
```

Figure 14: Execution of an FLP.

The proof system is constructed so that, if meas is valid, then run_flp(Flp, meas, 1) always returns True. On the other hand, if meas 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 MEAS_LEN.

For some FLPs, the encoded measurement 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, each representing a bit of the integer (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 encoded measurement to the length of the aggregated output:

- Flp.truncate(meas: Vec[Field]) -> Vec[Field] maps an encoded measurement (e.g., the bit-encoding of the measurement) to an aggregatable output (e.g., the singleton vector containing the measurement). The length of the input **MUST** be MEAS_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].

7.1.2. Multiple proofs

To improve soundness, the prover can construct multiple unique proofs for its measurement such that the verifier will only accept the measurement once all proofs have been verified. Notably, several proofs using a smaller field can offer the same level of soundness as a single proof using a large field.

To generate these proofs for a specific measurement, the prover calls Flp.prove multiple times, each time using an independently generated prover and joint randomness string. The verifier checks each proof independently, each time with an independently generated query randomness string. It accepts the measurement only if all the decision algorithm accepts on each proof.

See Section 9.5 below for discussions on choosing the right number of proofs.

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 Xof (Section 6.2). It also has an associated constant, PROOFS, with a value within the range of [1, 256), denoting the number of FLPs generated by the Client (Section 7.1.2). The value of PROOFS is 1 unless explicitly specified.

The associated constants and types required by the Vdaf interface are defined in Table 5. The methods required for sharding, preparation, aggregation, and unsharding are described in the remaining subsections. These methods refer to constants enumerated in Table 6.

Parameter	Value
VERIFY_KEY_SIZE	Xof.SEED_SIZE
RAND_SIZE	Xof.SEED_SIZE * (1 + 2 * (SHARES - 1)) if Flp.JOINT_RAND_LEN == 0 else Xof.SEED_SIZE * (1 + 2 * (SHARES - 1) + SHARES)
NONCE_SIZE	16
ROUNDS	1
SHARES	in [2, 256)
Measurement	Flp.Measurement
AggParam	None
PublicShare	Optional[list[bytes]]
InputShare	Union[tuple[list[Flp.Field], list[Flp.Field], Optional[bytes]], tuple[bytes, bytes, Optional[bytes]]]
OutShare	list[Flp.Field]
AggShare	list[Flp.Field]
AggResult	Flp.AggResult
PrepState	tuple[list[Flp.Field], Optional[Bytes]]
PrepShare	tuple[list[Flp.Field], Optional[Bytes]]
PrepMessage	Optional[bytes]

Table 5: VDAF parameters for Prio3.

Variable	Value
USAGE_MEAS_SHARE: Unsigned	1
USAGE_PROOF_SHARE: Unsigned	2
USAGE_JOINT_RANDOMNESS: Unsigned	3
USAGE_PROVE_RANDOMNESS: Unsigned	4
USAGE_QUERY_RANDOMNESS: Unsigned	5
USAGE_JOINT_RAND_SEED: Unsigned	6
USAGE_JOINT_RAND_PART: Unsigned	7

Table 6: 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 measurement in a way that allows the Aggregators to reconstruct it from their shares. (This idea is based on the Fiat-Shamir heuristic and is described in Section 6.2.3 of [BBCGGI19].)

The sharding algorithm involves the following steps:

1. Encode the Client's measurement for the FLP

2. Shard the measurement into a sequence of measurement shares

3. Derive the joint randomness from the measurement 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

6. Return the public share, consisting of the joint randomness parts, and the input shares, each consisting of the measurement share, proof share, and blind of one of the Aggregators

As described in Section 7.1.2, the soundness of the FLP can be amplified by generating and verifying multiple FLPs. (This in turn improves the robustness of Prio3.) To support this, in Prio3:

* In step 3, derive as much joint randomness as required by PROOFS proofs

* Repeat step 4 PROOFS times, each time with a unique joint randomness

Depending on the FLP, joint randomness may not be required. In particular, when Flp.JOINT_RAND_LEN == 0, the Client does not derive the joint randomness (Step 3). The sharding algorithm is specified below.

```
def shard(Prio3, measurement, nonce, rand):
    l = Prio3.Xof.SEED_SIZE
    seeds = [rand[i:i+l] for i in range(0, Prio3.RAND_SIZE, l)]

    meas = Prio3.Flp.encode(measurement)
    if Prio3.Flp.JOINT_RAND_LEN > 0:
        return Prio3.shard_with_joint_rand(meas, nonce, seeds)
    else:
        return Prio3.shard_without_joint_rand(meas, seeds)
```

Figure 15: Input-distribution algorithm for Prio3.

It starts by splitting the randomness into seeds. It then encodes the measurement as prescribed by the FLP and calls one of two methods, depending on whether joint randomness is required by the FLP. The methods are defined in the subsections below.

7.2.1.1. FLPs without joint randomness

The following method is used for FLPs that do not require joint randomness, i.e., when Flp.JOINT_RAND_LEN == 0:

```
def shard_without_joint_rand(Prio3, meas, seeds):
  k_helper_seeds, seeds = front((Prio3.SHARES-1) * 2, seeds)
  k_helper_meas_shares = [
      k_helper_seeds[i]
      for i in range(0, (Prio3.SHARES-1) * 2, 2)
  ]
  k_helper_proofs_shares = [
      k_helper_seeds[i]
      for i in range(1, (Prio3.SHARES-1) * 2, 2)
  ]
  (k_prove,), seeds = front(1, seeds)

  # Shard the encoded measurement into shares.
  leader_meas_share = meas
  for j in range(Prio3.SHARES-1):
      leader_meas_share = vec_sub(
          leader_meas_share,
          Prio3.helper_meas_share(j+1, k_helper_meas_shares[j]),
      )

  # Generate and shard each proof into shares.
  prove_rands = Prio3.prove_rands(k_prove)
  leader_proofs_share = []
  for _ in range(Prio3.PROOFS):
      prove_rand, prove_rands = front(
          Prio3.Flp.PROVE_RAND_LEN, prove_rands)
      leader_proofs_share += Prio3.Flp.prove(meas, prove_rand, [])
  for j in range(Prio3.SHARES-1):
      leader_proofs_share = vec_sub(
          leader_proofs_share,
          Prio3.helper_proofs_share(j+1, k_helper_proofs_shares[j]),
      )

  # Each Aggregator's input share contains its measurement share
  # and share of proof(s).
  input_shares = []
  input_shares.append((
      leader_meas_share,
      leader_proofs_share,
      None,
  ))
  for j in range(Prio3.SHARES-1):
      input_shares.append((
          k_helper_meas_shares[j],
          k_helper_proofs_shares[j],
          None,
      ))
  return (None, input_shares)
```

Figure 16: Sharding an encoded measurement without joint randomness.

The steps in this method are as follows:

1. Shard the encoded measurement into shares

2. Generate and shard each proof into shares

3. Encode each measurement and shares of each proof into an input share

Notice that only one pair of measurement and proof(s) share (called the "leader" shares above) are vectors of field elements. The other shares (called the "helper" shares) are represented instead by XOF seeds, which are expanded into vectors of field elements.

The methods on Prio3 for deriving the prover randomness, measurement shares, and proof shares and the methods for encoding the input shares are defined in Section 7.2.6.

7.2.1.2. FLPs with joint randomness

The following method is used for FLPs that require joint randomness, i.e., for which Flp.JOINT_RAND_LEN > 0:

```
def shard_with_joint_rand(Prio3, meas, nonce, seeds):
  k_helper_seeds, seeds = front((Prio3.SHARES-1) * 3, seeds)
  k_helper_meas_shares = [
      k_helper_seeds[i]
      for i in range(0, (Prio3.SHARES-1) * 3, 3)
  ]
  k_helper_proofs_shares = [
      k_helper_seeds[i]
      for i in range(1, (Prio3.SHARES-1) * 3, 3)
  ]
  k_helper_blinds = [
      k_helper_seeds[i]
      for i in range(2, (Prio3.SHARES-1) * 3, 3)
  ]
  (k_leader_blind,), seeds = front(1, seeds)
  (k_prove,), seeds = front(1, seeds)

  # Shard the encoded measurement into shares and compute the
  # joint randomness parts.
  leader_meas_share = meas
  k_joint_rand_parts = []
  for j in range(Prio3.SHARES-1):
      helper_meas_share = Prio3.helper_meas_share(
          j+1, k_helper_meas_shares[j])
      leader_meas_share = vec_sub(leader_meas_share,
                                  helper_meas_share)
      k_joint_rand_parts.append(Prio3.joint_rand_part(
          j+1, k_helper_blinds[j], helper_meas_share, nonce))
  k_joint_rand_parts.insert(0, Prio3.joint_rand_part(
      0, k_leader_blind, leader_meas_share, nonce))

  # Generate and shard each proof into shares.
  prove_rands = Prio3.prove_rands(k_prove)
  joint_rands = Prio3.joint_rands(
      Prio3.joint_rand_seed(k_joint_rand_parts))
  leader_proofs_share = []
  for _ in range(Prio3.PROOFS):
      prove_rand, prove_rands = front(
          Prio3.Flp.PROVE_RAND_LEN, prove_rands)
      joint_rand, joint_rands = front(
          Prio3.Flp.JOINT_RAND_LEN, joint_rands)
      leader_proofs_share += Prio3.Flp.prove(meas,
                                             prove_rand, joint_rand)
  for j in range(Prio3.SHARES-1):
      leader_proofs_share = vec_sub(
          leader_proofs_share,
          Prio3.helper_proofs_share(j+1, k_helper_proofs_shares[j]),
      )
```

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

Figure 17: Sharding an encoded measurement with joint randomness.

The difference between this procedure and previous one is that here we compute joint randomnesses joint_rands, split it into multiple joint_rand, and pass each joint_rand to the proof generationg algorithm. (In Figure 16 the joint randomness is the empty vector, [].) This requires generating an additional value, called the "blind", that is incorporated into each input share.

The joint randomness computation involves the following steps:

1. Compute a "joint randomness part" from each measurement share and blind

2. Compute a "joint randomness seed" from the joint randomness parts

3. Compute the joint randomness for each proof evaluation from the joint randomness seed

This three-step process is designed to ensure that the joint randomness does not leak the measurement to the Aggregators while preventing a malicious Client from tampering with the joint randomness in a way that allows it to break robustness. To bootstrap the required check, the Client encodes the joint randomness parts in the public share. (See Section 7.2.2 for details.)

The methods used in this computation are defined in Section 7.2.6.

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 measurement and share of proof(s) locally, then exchanges its share of verifier(s) with the other Aggregators. The shares of verifier(s) are then combined into the verifier message(s) used to decide whether to accept.

In addition, for FLPs that require joint randomness, the Aggregators must ensure that they have all used the same joint randomness for the query-generation algorithm. To do so, they collectively re-derive the joint randomness from their measurement shares just as the Client did during sharding.

In order to avoid extra round of communication, the Client sends each Aggregator a "hint" consisting of the joint randomness parts. 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 shares of verifier(s).

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 = public_share
  (meas_share, proofs_share, k_blind) = \
      Prio3.expand_input_share(agg_id, input_share)
  out_share = Prio3.Flp.truncate(meas_share)

  # Compute the joint randomness.
  joint_rand = []
  k_corrected_joint_rand, k_joint_rand_part = None, None
  if Prio3.Flp.JOINT_RAND_LEN > 0:
      k_joint_rand_part = Prio3.joint_rand_part(
          agg_id, k_blind, meas_share, nonce)
      k_joint_rand_parts[agg_id] = k_joint_rand_part
      k_corrected_joint_rand = Prio3.joint_rand_seed(
          k_joint_rand_parts)
      joint_rands = Prio3.joint_rands(k_corrected_joint_rand)

  # Query the measurement and proof share.
  query_rands = Prio3.query_rands(verify_key, nonce)
  verifiers_share = []
  for _ in range(Prio3.PROOFS):
      proof_share, proofs_share = front(
          Prio3.Flp.PROOF_LEN, proofs_share)
      query_rand, query_rands = front(
          Prio3.Flp.QUERY_RAND_LEN, query_rands)
      if Prio3.Flp.JOINT_RAND_LEN > 0:
          joint_rand, joint_rands = front(
              Prio3.Flp.JOINT_RAND_LEN, joint_rands)
      verifiers_share += Prio3.Flp.query(meas_share,
                                         proof_share,
                                         query_rand,
                                         joint_rand,
                                         Prio3.SHARES)

  prep_state = (out_share, k_corrected_joint_rand)
  prep_share = (verifiers_share, k_joint_rand_part)
  return (prep_state, prep_share)

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

  # If joint randomness was used, check that the value computed by the
  # Aggregators matches the value indicated by the Client.
  if k_joint_rand != k_corrected_joint_rand:
      raise ERR_VERIFY  # joint randomness check failed

  return out_share
```

```
def prep_shares_to_prep(Prio3, _agg_param, prep_shares):
  # Unshard the verifier shares into the verifier message.
  verifiers = Prio3.Flp.Field.zeros(
      Prio3.Flp.VERIFIER_LEN * Prio3.PROOFS)
  k_joint_rand_parts = []
  for (verifiers_share, k_joint_rand_part) in prep_shares:
      verifiers = vec_add(verifiers, verifiers_share)
      if Prio3.Flp.JOINT_RAND_LEN > 0:
          k_joint_rand_parts.append(k_joint_rand_part)

  # Verify that each proof is well-formed and the input is valid
  for _ in range(Prio3.PROOFS):
      verifier, verifiers = front(Prio3.Flp.VERIFIER_LEN, verifiers)
      if not Prio3.Flp.decide(verifier):
          raise ERR_VERIFY  # proof verifier check failed

  # Combine the joint randomness parts computed by the
  # Aggregators into the true joint randomness seed. This is
  # used in the last step.
  k_joint_rand = None
  if Prio3.Flp.JOINT_RAND_LEN > 0:
      k_joint_rand = Prio3.joint_rand_seed(k_joint_rand_parts)
  return k_joint_rand
```

Figure 18: 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 19: 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 aggregate(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 agg_share
```

Figure 20: 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 unshard(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, agg_share)
    return Prio3.Flp.decode(agg, num_measurements)
```

Figure 21: Computation of the aggregate result for Prio3.

7.2.6. Auxiliary Functions

This section defines a number of auxiliary functions referenced by the main algorithms for Prio3 in the preceding sections.

The following methods are called by the sharding and preparation algorithms.

```
def helper_meas_share(Prio3, agg_id, k_share):
    return Prio3.Xof.expand_into_vec(
        Prio3.Flp.Field,
        k_share,
        Prio3.domain_separation_tag(USAGE_MEAS_SHARE),
        byte(agg_id),
        Prio3.Flp.MEAS_LEN,
    )

def helper_proofs_share(Prio3, agg_id, k_share):
  return Prio3.Xof.expand_into_vec(
      Prio3.Flp.Field,
      k_share,
      Prio3.domain_separation_tag(USAGE_PROOF_SHARE),
      byte(Prio3.PROOFS) + byte(agg_id),
      Prio3.Flp.PROOF_LEN * Prio3.PROOFS,
  )

def expand_input_share(Prio3, agg_id, input_share):
  (meas_share, proofs_share, k_blind) = input_share
  if agg_id > 0:
      meas_share = Prio3.helper_meas_share(agg_id, meas_share)
      proofs_share = Prio3.helper_proofs_share(agg_id, proofs_share)
  return (meas_share, proofs_share, k_blind)

def prove_rands(Prio3, k_prove):
  return Prio3.Xof.expand_into_vec(
      Prio3.Flp.Field,
      k_prove,
      Prio3.domain_separation_tag(USAGE_PROVE_RANDOMNESS),
      byte(Prio3.PROOFS),
      Prio3.Flp.PROVE_RAND_LEN * Prio3.PROOFS,
  )

def query_rands(Prio3, verify_key, nonce):
  return Prio3.Xof.expand_into_vec(
      Prio3.Flp.Field,
      verify_key,
      Prio3.domain_separation_tag(USAGE_QUERY_RANDOMNESS),
      byte(Prio3.PROOFS) + nonce,
      Prio3.Flp.QUERY_RAND_LEN * Prio3.PROOFS,
  )

def joint_rand_part(Prio3, agg_id, k_blind, meas_share, nonce):
    return Prio3.Xof.derive_seed(
        k_blind,
        Prio3.domain_separation_tag(USAGE_JOINT_RAND_PART),
        byte(agg_id) + nonce + Prio3.Flp.Field.encode_vec(meas_share),
    )
```

```
def joint_rand_seed(Prio3, k_joint_rand_parts):
    """Derive the joint randomness seed from its parts."""
    return Prio3.Xof.derive_seed(
        zeros(Prio3.Xof.SEED_SIZE),
        Prio3.domain_separation_tag(USAGE_JOINT_RAND_SEED),
        concat(k_joint_rand_parts),
    )

def joint_rands(Prio3, k_joint_rand_seed):
  """Derive the joint randomness from its seed."""
  return Prio3.Xof.expand_into_vec(
      Prio3.Flp.Field,
      k_joint_rand_seed,
      Prio3.domain_separation_tag(USAGE_JOINT_RANDOMNESS),
      byte(Prio3.PROOFS),
      Prio3.Flp.JOINT_RAND_LEN * Prio3.PROOFS,
  )
```

7.2.7. Message Serialization

This section defines serialization formats for messages exchanged over the network while executing Prio3. It is **RECOMMENDED** that implementations provide serialization methods for them.

Message structures are defined following Section 3 of [RFC8446]). In the remainder we use S as an alias for Prio3.Xof.SEED_SIZE and F as an alias for Prio3.Field.ENCODED_SIZE. XOF seeds are represented as follows:

```
opaque Prio3Seed[S];
```

Field elements are encoded in little-endian byte order (as defined in Section 6.1) and represented as follows:

```
opaque Prio3Field[F];
```

7.2.7.1. Public Share

The encoding of the public share depends on whether joint randomness is required for the underlying FLP (i.e., Prio3.Flp.JOINT_RAND_LEN > 0). If joint randomness is not used, then the public share is the empty string. If joint randomness is used, then the public share encodes the joint randomness parts as follows:

```
struct {
    Prio3Seed k_joint_rand_parts[S * Prio3.SHARES];
} Prio3PublicShareWithJointRand;
```

7.2.7.2. Input share

Just as for the public share, the encoding of the input shares depends on whether joint randomness is used. If so, then each input share includes the Aggregator's blind for generating its joint randomness part.

In addition, the encoding of the input shares depends on which aggregator is receiving the message. If the aggregator ID is 0, then the input share includes the full measurement and share of proof(s). Otherwise, if the aggregator ID is greater than 0, then the measurement and shares of proof(s) are represented by XOF seeds. We shall call the former the "Leader" and the latter the "Helpers".

In total there are four variants of the input share. When joint randomness is not used, the Leader's share is structured as follows:

```
struct {
    Prio3Field meas_share[F * Prio3.Flp.MEAS_LEN];
    Prio3Field proofs_share[F * Prio3.Flp.PROOF_LEN * Prio3.PROOFS];
} Prio3LeaderShare;
```

When joint randomness is not used, the Helpers' shares are structured as follows:

```
struct {
    Prio3Seed k_meas_share;
    Prio3Seed k_proofs_share;
} Prio3HelperShare;
```

When joint randomness is used, the Leader's input share is structured as follows:

```
struct {
    Prio3LeaderShare inner;
    Prio3Seed k_blind;
} Prio3LeaderShareWithJointRand;
```

Finally, when joint randomness is used, the Helpers' shares are structured as follows:

```
struct {
    Prio3HelperShare inner;
    Prio3Seed k_blind;
} Prio3HelperShareWithJointRand;
```

7.2.7.3. Prep Share

When joint randomness is not used, the prep share is structured as follows:

```
struct {
    Prio3Field verifiers_share[F * Prio3.Flp.VERIFIER_LEN * Prio3.PROOFS
} Prio3PrepShare;
```

When joint randomness is used, the prep share includes the Aggregator's joint randomness part and is structured as follows:

```
struct {
    Prio3Field verifiers_share[F * Prio3.Flp.VERIFIER_LEN * Prio3.PROOFS
    Prio3Seed k_joint_rand_part;
} Prio3PrepShareWithJointRand;
```

7.2.7.4. Prep Message

When joint randomness is not used, the prep message is the empty string. Otherwise the prep message consists of the joint randomness seed computed by the Aggregators:

```
struct {
    Prio3Seed k_joint_rand;
} Prio3PrepMessageWithJointRand;
```

7.2.7.5. Aggregation

Aggregate shares are structured as follows:

```
struct {
    Prio3Field agg_share[F * Prio3.Flp.OUTPUT_LEN];
} Prio3AggShare;
```

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 encoded measurement: If the circuit output is zero, then the measurement is deemed valid; otherwise, if the circuit output is non-zero, then the measurement is deemed invalid. Thus the goal of the proof system is merely to

allow the verifier to evaluate the validity circuit over the measurement. For our application (Section 7), this computation is distributed among multiple Aggregators, each of which has only a share of the measurement.

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 a measurement 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 measurement 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 measurement 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 measurement 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 of [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 measurement 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(meas, r) = r * Range2(meas[0]) + ... + r^N * Range2(meas[N-1])
```

(Note that this is a special case of [BBCGGI19], Theorem 5.2.) Here meas 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 meas is valid (i.e., each meas[j] is in [0,2)), then the circuit will evaluate to 0 regardless of the value of r; but if meas[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.) This provides additional flexibility for designing circuits by allowing multiple, non-affine sub-components. For example, the following circuit is allowed:

```
C(meas, r) = r * Range2(meas[0]) + ... + r^L * Range2(meas[L-1]) + \
             r^(L+1) * Range3(meas[L]) + ... + r^N * Range3(meas[N-1])
```

where Range3(x) = x^3 - 3x^2 + 2x. This circuit checks that the first L inputs are in range [0,2) and the last N-L inputs are in range [0,3). Of course, the same circuit can be expressed using a sub-component that the gadgets have in common, namely Mul, but the resulting proof would be longer.

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
MEAS_LEN	Length of the measurement
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 7: 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).

Gadgets provide a method to evaluate their circuit on a list of inputs, eval(). The inputs can either belong to the validity circuit's field, or the polynomial ring over that field.

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 MEAS_LEN representing a measurement.

*Valid.truncate(meas: 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 methods are derived for each concrete Valid:

```
def prove_rand_len(self):
    """Length of the prover randomness."""
    return sum(g.ARITY for g in Valid.GADGETS)

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

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

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

Figure 22: 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.py.

The FLP parameters for FlpGeneric are defined in Table 8. 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
MEAS_LEN	Valid.MEAS_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 8: FLP Parameters of FlpGeneric.

7.3.3.1. Proof Generation

On input of meas, 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 meas 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 meas, 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 meas 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 measurement 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 XOF (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 XofTurboShake128 (Section 6.2.1) as its XOF. 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 eval(self, Field, inp):
    self.check_gadget_eval(inp)
    return inp[0] * inp[1]
```

The call to check_gadget_eval() raises an error if the length of the input is not equal to the gadget's ARITY parameter.

The Count validity circuit is defined as

```
def eval(self, meas, joint_rand, _num_shares):
    return self.GADGETS[0].eval(self.Field, [meas[0], meas[0]]) \
        - meas[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]
MEAS_LEN	1
OUTPUT_LEN	1
JOINT_RAND_LEN	0
Measurement	Unsigned, in range [0,2)
AggResult	Unsigned
Field	Field64 (Table 3)

Table 9: 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 XofTurboShake128 (Section 6.2.1) as its XOF. Its validity circuit, denoted Sum, uses Field128 (Table 3) 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(self, measurement):
    if 0 > measurement or measurement >= 2 ** self.MEAS_LEN:
        raise ERR_INPUT

    return self.Field.encode_into_bit_vector(measurement,
                                             self.MEAS_LEN)

def truncate(self, meas):
    return [self.Field.decode_from_bit_vector(meas)]

def decode(self, 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 eval(self, Field, inp):
    self.check_gadget_eval(inp)
    return inp[0] * inp[0] - inp[0]
```

The Sum validity circuit is defined as

```
def eval(self, meas, joint_rand, _num_shares):
    self.check_valid_eval(meas, joint_rand)
    out = self.Field(0)
    r = joint_rand[0]
    for b in meas:
        out += r * self.GADGETS[0].eval(self.Field, [b])
        r *= joint_rand[0]
    return out
```

Parameter	Value
GADGETS	[Range2]
GADGET_CALLS	[bits]
MEAS_LEN	bits
OUTPUT_LEN	1
JOINT_RAND_LEN	1
Measurement	Unsigned, in range [0, 2^bits)
AggResult	Unsigned
Field	Field128 (Table 3)

Table 10: Parameters of validity circuit Sum.

7.4.3. Prio3SumVec

This instance of Prio3 supports summing a vector of integers. It has three parameters, length, bits, and chunk_length. Each measurement is a vector of positive integers with length equal to the length

parameter. Each element of the measurement is an integer in the range [0, 2^bits). It is **RECOMMENDED** to set chunk_length to an integer near the square root of length * bits (see Section 7.4.3.1).

This instance uses XofTurboShake128 (Section 6.2.1) as its XOF. Its validity circuit, denoted SumVec, uses Field128 (Table 3) as its finite field.

Measurements are encoded as a vector of field elements with length length * bits. The field elements in the encoded vector represent all the bits of the measurement vector's elements, consecutively, in LSB to MSB order:

```
def encode(self, measurement: Vec[Unsigned]):
    if len(measurement) != self.length:
        raise ERR_INPUT

    encoded = []
    for val in measurement:
        if 0 > val or val >= 2 ** self.bits:
            raise ERR_INPUT

        encoded += self.Field.encode_into_bit_vector(val, self.bits)
    return encoded

def truncate(self, meas):
    truncated = []
    for i in range(self.length):
        truncated.append(self.Field.decode_from_bit_vector(
            meas[i * self.bits: (i + 1) * self.bits]
        ))
    return truncated

def decode(self, output, _num_measurements):
    return [x.as_unsigned() for x in output]
```

This validity circuit uses a ParallelSum gadget to achieve a smaller proof size. This optimization for "parallel-sum circuits" is described in [BBCGGI19], section 4.4. Briefly, for circuits that add up the output of multiple identical subcircuits, it is possible to achieve smaller proof sizes (on the order of O(sqrt(MEAS_LEN)) instead of O(MEAS_LEN)) by packaging more than one such subcircuit into a gadget.

The ParallelSum gadget is parameterized with an arithmetic subcircuit, and a count of how many times it evaluates that subcircuit. It takes in a list of inputs and passes them through to instances of the subcircuit in the same order. It returns the sum of the subcircuit outputs. Note that only the ParallelSum gadget itself, and not its subcircuit, participates in FlpGeneric's wire

recording during evaluation, gadget consistency proofs, and proof validation, even though the subcircuit is provided to ParallelSum as an implementation of the Gadget interface.

```
def eval(self, Field, inp):
    self.check_gadget_eval(inp)
    out = Field(0)
    for i in range(self.count):
        start_index = i * self.subcircuit.ARITY
        end_index = (i + 1) * self.subcircuit.ARITY
        out += self.subcircuit.eval(Field, inp[start_index:end_index])
    return out
```

The SumVec validity circuit checks that the encoded measurement consists of ones and zeros. Rather than use the Range2 gadget on each element, as in the Sum validity circuit, it instead uses Mul subcircuits and "free" constant multiplication and addition gates to simultaneously evaluate the same range check polynomial on each element, and multiply by a constant. One of the two Mul subcircuit inputs is equal to a measurement element multiplied by a power of the joint randomness value, and the other is equal to the same measurement element minus one. These Mul subcircuits are evaluated by a ParallelSum gadget, and the results are added up both within the ParallelSum gadget and after it.

```
def eval(self, meas, joint_rand, num_shares):
    self.check_valid_eval(meas, joint_rand)

    out = Field128(0)
    r = joint_rand[0]
    r_power = r
    shares_inv = self.Field(num_shares).inv()

    for i in range(self.GADGET_CALLS[0]):
        inputs = [None] * (2 * self.chunk_length)
        for j in range(self.chunk_length):
            index = i * self.chunk_length + j
            if index < len(meas):
                meas_elem = meas[index]
            else:
                meas_elem = self.Field(0)

            inputs[j * 2] = r_power * meas_elem
            inputs[j * 2 + 1] = meas_elem - shares_inv

            r_power *= r

        out += self.GADGETS[0].eval(self.Field, inputs)

    return out
```

Parameter	Value
GADGETS	[ParallelSum(Mul(), chunk_length)]
GADGET_CALLS	[(length * bits + chunk_length - 1) // chunk_length]
MEAS_LEN	length * bits
OUTPUT_LEN	length
JOINT_RAND_LEN	1
Measurement	Vec[Unsigned], each element in range [0, 2^bits)
AggResult	Vec[Unsigned]
Field	Field128 (Table 3)

Table 11: Parameters of validity circuit SumVec.

7.4.3.1. Selection of ParallelSum chunk length

The chunk_length parameter provides a trade-off between the arity of the ParallelSum gadget and the number of times the gadget is called. The proof length is asymptotically minimized when the chunk length is near the square root of the length of the measurement. However, the relationship between VDAF parameters and proof length is complicated, involving two forms of rounding (the circuit pads the inputs to its last ParallelSum gadget call, up to the chunk length, and FlpGeneric rounds the degree of wire polynomials -- determined by the number of times a gadget is called -- up to the next power of two). Therefore, the optimal choice of chunk_length for a concrete measurement size will vary, and must be found through trial and error. Setting chunk_length equal to the square root of the appropriate measurement length will result in proofs up to 50% larger than the optimal proof size.

7.4.4. Prio3Histogram

This instance of Prio3 allows for estimating the distribution of some quantity by computing a simple histogram. Each measurement increments one histogram bucket, out of a set of fixed buckets. (Bucket indexing begins at 0.) For example, the buckets might quantize the real numbers, and each measurement would report the bucket that the corresponding client's real-numbered value falls into. The aggregate result counts the number of measurements in each bucket.

This instance of Prio3 uses XofTurboShake128 (Section 6.2.1) as its XOF. Its validity circuit, denoted Histogram, uses Field128 (Table 3) as its finite field. It has two parameters, length, the number of histogram buckets, and chunk_length, which is used by by a circuit optimization described below. It is **RECOMMENDED** to set chunk_length to an integer near the square root of length (see Section 7.4.3.1).

The measurement is encoded as a one-hot vector representing the bucket into which the measurement falls:

```
def encode(self, measurement):
    encoded = [self.Field(0)] * self.length
    encoded[measurement] = self.Field(1)
    return encoded

def truncate(self, meas):
    return meas

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

The Histogram validity circuit checks for one-hotness in two steps, by checking that the encoded measurement consists of ones and zeros, and by checking that the sum of all elements in the encoded measurement is equal to one. All the individual checks are combined together in a random linear combination.

As in the SumVec validity circuit (Section 7.4.3), the first part of the validity circuit uses the ParallelSum gadget to perform range checks while achieving a smaller proof size. The ParallelSum gadget uses Mul subcircuits to evaluate a range check polynomial on each element, and includes an additional constant multiplication. One of the two Mul subcircuit inputs is equal to a measurement element multiplied by a power of the first joint randomness value, and the other is equal to the same measurement element minus one. The results are added up both within the ParallelSum gadget and after it.

```
def eval(self, meas, joint_rand, num_shares):
    self.check_valid_eval(meas, joint_rand)

    # Check that each bucket is one or zero.
    range_check = self.Field(0)
    r = joint_rand[0]
    r_power = r
    shares_inv = self.Field(num_shares).inv()
    for i in range(self.GADGET_CALLS[0]):
        inputs = [None] * (2 * self.chunk_length)
        for j in range(self.chunk_length):
            index = i * self.chunk_length + j
            if index < len(meas):
                meas_elem = meas[index]
            else:
                meas_elem = self.Field(0)

            inputs[j * 2] = r_power * meas_elem
            inputs[j * 2 + 1] = meas_elem - shares_inv

            r_power *= r

        range_check += r * self.GADGETS[0].eval(self.Field, inputs)

    # Check that the buckets sum to 1.
    sum_check = -shares_inv
    for b in meas:
        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 measurement is sharded. This is provided to the FLP by Prio3.

Parameter	Value
GADGETS	[ParallelSum(Mul(), chunk_length)]
GADGET_CALLS	[(length + chunk_length - 1) // chunk_length]
MEAS_LEN	length
OUTPUT_LEN	length
JOINT_RAND_LEN	2
Measurement	Unsigned
AggResult	Vec[Unsigned]
Field	Field128 (Table 3)

Table 12: 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 measurements are prefixed by each candidate prefix.

This functionality is the core component of the Poplar protocol [BBCGGI21], which was designed to compute the heavy hitters over a set of input strings. At a high level, the protocol works as follows.

1. Each Client splits its 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, e.g., for each p in H, 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 the function evaluates to a non-zero value; otherwise it evaluates to zero. This structure allows an IDPF to provide the functionality required for the above protocol: To compute the hit count for an index, just evaluate each set of IDPF shares at that index and add up the results.

Consider the sub-tree constructed from a set of input strings and a target threshold t by including all indices that prefix at least t of the input strings. We shall refer to this structure as the "prefix tree" for the batch of inputs and target threshold. To compute the t-heavy hitters for a set of inputs, the Aggregators and Collector first compute the prefix tree, then extract the heavy hitters from the leaves of this tree. (Note that the prefix tree may leak more information about the set than the heavy hitters themselves; see Section 9.3.1 for details.)

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 FieldInner), and one for leaf nodes (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 13. In the remainder we write Output as shorthand for the type Union[list[list[FieldInner]], list[list[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: list[list[FieldInner]], beta_leaf: list[FieldLeaf], binder: bytes, rand: bytes[Idpf.RAND_SIZE]) -> tuple[bytes, list[bytes]] is the randomized IDPF-key generation algorithm. (Input rand consists of the random bytes it consumes.) Its inputs are the index alpha the values beta, and a binder string. 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. The binder string is used to derive the key in the underlying XofFixedKeyAes128 XOF that is used for expanding seeds at each level. It **MUST** be chosen uniformly at random by the Client (see Section 9.2).

  TODO(issue #255) Decide whether to treat the public share as an opaque byte string or to replace it with an explicit type.

* Idpf.eval(agg_id: Unsigned, public_share: bytes, key: bytes, level: Unsigned, prefixes: tuple[Unsigned, ...], binder: Bytes) -> Output 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, the sequence of candidate prefixes, and a binder string. It returns the share of the value corresponding to each candidate prefix.

  The output type (i.e., Output) depends on the value of level: If level < Idpf.BITS-1, the output is the value for an inner node, which has type list[list[Idpf.FieldInner]]; otherwise, if level == Idpf.BITS-1, then the output is the value for a leaf node, which has type list[list[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
RAND_SIZE	Size of the random string consumed by the IDPF-key generator. Equal to twice the XOF's seed size.
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
Output	Alias of Union[list[list[FieldInner]], list[list[FieldLeaf]]]
FieldVec	Alias of Union[list[FieldInner], list[FieldLeaf]]

Table 13: 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 and an implementation of Xof (Section 6.2). The associated constants and types required by the Vdaf interface are defined in Table 14. The methods required for sharding, preparation, aggregation, and unsharding are described in the remaining subsections. These methods make use of constants defined in Table 15.

Parameter	Value
VERIFY_KEY_SIZE	Xof.SEED_SIZE
RAND_SIZE	Xof.SEED_SIZE * 3 + Idpf.RAND_SIZE
NONCE_SIZE	16
ROUNDS	2
SHARES	2
Measurement	Unsigned
AggParam	Tuple[Unsigned, Tuple[Unsigned, ...]]

Parameter	Value
PublicShare	bytes (IDPF public share)
InputShare	tuple[bytes, bytes, list[Idpf.FieldInner], list[Idpf.FieldLeaf]]
OutShare	Idpf.FieldVec
AggShare	Idpf.FieldVec
AggResult	Vec[Unsigned]
PrepState	tuple[bytes, Unsigned, Idpf.FieldVec]
PrepShare	Idpf.FieldVec
PrepMessage	Optional[Idpf.FieldVec]

Table 14: VDAF parameters for Poplar1.

Variable	Value
USAGE_SHARD_RAND: Unsigned	1
USAGE_CORR_INNER: Unsigned	2
USAGE_CORR_LEAF: Unsigned	3
USAGE_VERIFY_RAND: Unsigned	4

Table 15: Constants used by Poplar1.

8.2.1. Client

The Client's measurement is interpreted as 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 sharding algorithm is defined as follows.

```
def shard(Poplar1, measurement, nonce, rand):
    l = Poplar1.Xof.SEED_SIZE

    # Split the random input into random input for IDPF key
    # generation, correlated randomness, and sharding.
    if len(rand) != Poplar1.RAND_SIZE:
        raise ERR_INPUT # unexpected length for random input
    idpf_rand, rand = front(Poplar1.Idpf.RAND_SIZE, rand)
    seeds = [rand[i:i+l] for i in range(0,3*l,l)]
    corr_seed, seeds = front(2, seeds)
    (k_shard,), seeds = front(1, seeds)

    xof = Poplar1.Xof(
        k_shard,
        Poplar1.domain_separation_tag(USAGE_SHARD_RAND),
        nonce,
    )

    # 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 xof.next_vec(Poplar1.Idpf.FieldInner,
                              Poplar1.Idpf.BITS - 1)
    ]
    beta_leaf = [Poplar1.Idpf.FieldLeaf(1)] + \
        xof.next_vec(Poplar1.Idpf.FieldLeaf, 1)

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

    # Generate correlated randomness used by the Aggregators to
    # compute a sketch over their output shares. XOF seeds are
    # used to encode shares of the `(a, b, c)` triples.
    # (See [BBCGGI21, Appendix C.4].)
    corr_offsets = vec_add(
        Poplar1.Xof.expand_into_vec(
            Poplar1.Idpf.FieldInner,
            corr_seed[0],
            Poplar1.domain_separation_tag(USAGE_CORR_INNER),
            byte(0) + nonce,
            3 * (Poplar1.Idpf.BITS-1),
```

```
    ),
    Poplar1.Xof.expand_into_vec(
        Poplar1.Idpf.FieldInner,
        corr_seed[1],
        Poplar1.domain_separation_tag(USAGE_CORR_INNER),
        byte(1) + nonce,
        3 * (Poplar1.Idpf.BITS-1),
    ),
)
corr_offsets += vec_add(
    Poplar1.Xof.expand_into_vec(
        Poplar1.Idpf.FieldLeaf,
        corr_seed[0],
        Poplar1.domain_separation_tag(USAGE_CORR_LEAF),
        byte(0) + nonce,
        3,
    ),
    Poplar1.Xof.expand_into_vec(
        Poplar1.Idpf.FieldLeaf,
        corr_seed[1],
        Poplar1.domain_separation_tag(USAGE_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 = xof.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.
input_shares = list(zip(keys, corr_seed, corr_inner, corr_leaf))
return (public_share, input_shares)
```

Figure 23: The sharding 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.

```
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) = 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, nonce)

    # 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_xof = Poplar1.Xof(
            corr_seed,
            Poplar1.domain_separation_tag(USAGE_CORR_INNER),
            byte(agg_id) + nonce,
        )
        # Fast-forward the XOF state to the current level.
        corr_xof.next_vec(Field, 3 * level)
    else:
        corr_xof = Poplar1.Xof(
            corr_seed,
            Poplar1.domain_separation_tag(USAGE_CORR_LEAF),
            byte(agg_id) + nonce,
        )
    (a_share, b_share, c_share) = corr_xof.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_xof = Poplar1.Xof(
        verify_key,
        Poplar1.domain_separation_tag(USAGE_VERIFY_RAND),
        nonce + to_be_bytes(level, 2),
    )
    verify_rand = verify_rand_xof.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 = [A_share, B_share, Field(agg_id)] + out_share
        return ((b'sketch round 1', level, prep_mem),
                sketch_share)

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

        if step == b'sketch round 1':
            if prev_sketch == None:
                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),
                    sketch_share)

        elif step == b'sketch round 2':
            if prev_sketch == None:
                return prep_mem  # Output shares
            else:
                raise ERR_INPUT  # prep message malformed

        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(prep_shares[0], prep_shares[1])
        if len(sketch) == 3:
            return sketch
        elif len(sketch) == 1:
            if sketch == Field.zeros(1):
```

```
        # In order to reduce communication overhead, let `None`
        # denote a successful sketch verification.
        return None
    else:
        raise ERR_VERIFY  # sketch verification failed
else:
    raise ERR_INPUT  # unexpected input length
```

Figure 24: 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 25: Validity of aggregation parameters for Poplar1.

8.2.4. Aggregation

Aggregation involves simply adding up the output shares.

```
def aggregate(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 agg_share
```

Figure 26: Aggregation algorithm for Poplar1.

8.2.5. Unsharding

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

```
def unshard(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, agg_share)
    return list(map(lambda x: x.as_unsigned(), agg))
```

Figure 27: Computation of the aggregate result for Poplar1.

8.2.6. Message Serialization

This section defines serialization formats for messages exchanged over the network while executing Poplar1. It is **RECOMMENDED** that implementations provide serialization methods for them.

Message structures are defined following Section 3 of [RFC8446]). In the remainder we use S as an alias for Poplar1.Xof.SEED_SIZE, Fi as an alias for Poplar1.Idpf.FieldInner and Fl as an alias for Poplar1.Idpf.FieldLeaf. XOF seeds are represented as follows:

```
opaque Poplar1Seed[S];
```

Elements of the inner field are encoded in little-endian byte order (as defined in Section 6.1) and are represented as follows:

```
opaque Poplar1FieldInner[Fi];
```

Likewise, elements of the leaf field are encoded in little-endian byte order (as defined in Section 6.1) and are represented as follows:

```
opaque Poplar1FieldLeaf[Fl];
```

8.2.6.1. Public Share

The public share is equal to the IDPF public share, which is a byte string. (See Section 8.1.)

8.2.6.2. Input Share

Each input share is structured as follows:

```
struct {
    opaque idpf_key[Poplar1.Idpf.KEY_SIZE];
    Poplar1Seed corr_seed;
    Poplar1FieldInner corr_inner[Fi * 2 * (Poplar1.Idpf.BITS - 1)];
    Poplar1FieldLeaf corr_leaf[Fl * 2];
} Poplar1InputShare;
```

8.2.6.3. Prep Share

Encoding of the prep share depends on the round of sketching: if the first round, then each sketch share has three field elements; if the second round, then each sketch share has one field element. The field that is used depends on the level of the IDPF tree specified by the aggregation parameter, either the inner field or the leaf field.

For the first round and inner field:

```
struct {
    Poplar1FieldInner sketch_share[Fi * 3];
} Poplar1PrepShareRoundOneInner;
```

For the first round and leaf field:

```
struct {
    Poplar1FieldLeaf sketch_share[Fl * 3];
} Poplar1PrepShareRoundOneLeaf;
```

For the second round and inner field:

```
struct {
    Poplar1FieldInner sketch_share;
} Poplar1PrepShareRoundTwoInner;
```

For the second round and leaf field:

```
struct {
    Poplar1FieldLeaf sketch_share;
} Poplar1PrepShareRoundTwoLeaf;
```

8.2.6.4. Prep Message

Likewise, the structure of the prep message for Poplar1 depends on the sketching round and field. For the first round and inner field:

```
struct {
    Poplar1FieldInner[Fi * 3];
} Poplar1PrepMessageRoundOneInner;
```

For the first round and leaf field:

```
struct {
    Poplar1FieldLeaf sketch[Fl * 3];
} Poplar1PrepMessageRoundOneLeaf;
```

Note that these messages have the same structures as the prep shares for the first round.

The second-round prep message is the empty string. This is because the sketch shares are expected to sum to a particular value if the output shares are valid; we represent a successful preparation with the empty string and otherwise return an error.

8.2.6.5. Aggregate Share

The encoding of the aggregate share depends on whether the inner or leaf field is used, and the number of candidate prefixes. Both of these are determined by the aggregation parameter.

Let prefix_count denote the number of candidate prefixes. For the inner field:

```
struct {
    Poplar1FieldInner agg_share[Fi * prefix_count];
} Poplar1AggShareInner;
```

For the leaf field:

```
struct {
    Poplar1FieldLeaf agg_share[Fl * prefix_count];
} Poplar1AggShareLeaf;
```

8.2.6.6. Aggregation Parameter

The aggregation parameter is encoded as follows:

> TODO(issue #255) Express the aggregation parameter encoding in TLS syntax. Decide whether to RECOMMEND this encoding, and if so, add it to test vectors.

```
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 += to_be_bytes(level, 2)
    encoded += to_be_bytes(len(prefixes), 4)
    packed = 0
    for (i, prefix) in enumerate(prefixes):
        packed |= prefix << ((level+1) * i)
    l = ((level+1) * len(prefixes) + 7) // 8
    encoded += to_be_bytes(packed, l)
    return encoded

def decode_agg_param(Poplar1, encoded):
    encoded_level, encoded = encoded[:2], encoded[2:]
    level = from_be_bytes(encoded_level)
    encoded_prefix_count, encoded = encoded[:4], encoded[4:]
    prefix_count = from_be_bytes(encoded_prefix_count)
    l = ((level+1) * prefix_count + 7) // 8
    encoded_packed, encoded = encoded[:l], encoded[l:]
    packed = from_be_bytes(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, tuple(prefixes))
```

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.

8.3. The IDPF scheme of *[BBCGGI21]*

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 16.

IdpfPoplar requires a XOF for deriving the output shares, as well as a variety of other artifacts used internally. For performance reasons, we instantiate this object using XofFixedKeyAes128 (Section 6.2.2). See Section 9.4 for justification of this choice.

Parameter	Value
SHARES	2
BITS	any positive integer
VALUE_LEN	any positive integer
KEY_SIZE	Xof.SEED_SIZE
FieldInner	Field64 (Table 3)
FieldLeaf	Field255 (Table 3)

Table 16: 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, binder, rand):
    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
    if len(rand) != IdpfPoplar.RAND_SIZE:
        raise ERR_INPUT # unexpected length for random input

    init_seed = [
        rand[:XofFixedKeyAes128.SEED_SIZE],
        rand[XofFixedKeyAes128.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], binder)
        (s1, t1) = IdpfPoplar.extend(seed[1], binder)
        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, binder)
        (seed[1], w1) = IdpfPoplar.convert(level, x1, binder)
        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 28: 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, binder):
    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,
                binder,
            )
        out_share.append(y if agg_id == 0 else vec_neg(y))
    return out_share
```

```
def eval_next(IdpfPoplar, prev_seed, prev_ctrl,
              correction_word, level, bit, binder):
    """
    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.
    """

    Field = IdpfPoplar.current_field(level)
    (seed_cw, ctrl_cw, w_cw) = correction_word
    (s, t) = IdpfPoplar.extend(prev_seed, binder)
    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], binder)
    # 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 29: IDPF-evaluation generation algorithm of IdpfPoplar.

8.3.3. Auxiliary Functions

```
def extend(IdpfPoplar, seed, binder):
    xof = XofFixedKeyAes128(seed, format_dst(1, 0, 0), binder)
    s = [
        bytearray(xof.next(XofFixedKeyAes128.SEED_SIZE)),
        bytearray(xof.next(XofFixedKeyAes128.SEED_SIZE)),
    ]
    # Use the least significant bits as the control bit correction,
    # and then zero it out. This gives effectively 127 bits of
    # security, but reduces the number of AES calls needed by 1/3.
    t = [Field2(s[0][0] & 1), Field2(s[1][0] & 1)]
    s[0][0] &= 0xFE
    s[1][0] &= 0xFE
    return (s, t)

def convert(IdpfPoplar, level, seed, binder):
    xof = XofFixedKeyAes128(seed, format_dst(1, 0, 1), binder)
    next_seed = xof.next(XofFixedKeyAes128.SEED_SIZE)
    Field = IdpfPoplar.current_field(level)
    w = xof.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 = (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 = XofFixedKeyAes128.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))
if len(encoded) != 0:
    raise ERR_DECODE
return correction_words
```

Figure 30: 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 XofTurboShake128 (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.

9.3.1. Additional Privacy Considerations

Aggregating a batch of reports multiple times, each time with a different aggregation parameter, could result in information leakage beyond what is used by the application.

For example, when Poplar1 is used for heavy hitters, the Aggregators learn not only the heavy hitters themselves, but also the prefix tree (as defined in Section 8) computed along the way. Indeed, this leakage is inherent to any construction that uses an IDPF (Section 8.1) in the same way. Depending on the distribution of the measurements, the prefix tree can leak a significant amount of information about unpopular inputs. For instance, it is possible (though perhaps unlikely) for a large set of non-heavy-hitter values to share a common prefix, which would be leaked by a prefix tree with a sufficiently small threshold.

The only known, general-purpose approach to mitigating this leakage is via differential privacy.

> TODO(issue #94) Describe (or point to some description of) the central DP mechanism for Poplar described in [BBCGGI21].

9.4. Requirements for XOFs

As described in Section 6.2, our constructions rely on eXtendable Output Functions (XOFs). In the security analyses of our protocols, these are usually modeled as random oracles. XofTurboShake128 is designed to be indifferentiable from a random oracle [MRH04], making it a suitable choice for most situations.

The one exception is the Idpf implementation IdpfPoplar Section 8.3. Here, a random oracle is not needed to prove privacy, since the analysis of [BBCGGI21], Proposition 1, only requires a Pseudorandom Generator (PRG). As observed in [GKWY20], a PRG can be instantiated from a correlation-robust hash function H. Informally, correlation robustness requires that for a random r, H(xor(r, x)) is computationally indistinguishable from a random function of x. A PRG can therefore be constructed as

PRG(r) = H(xor(r, 1)) || H(xor(r, 2)) || ...`

since each individual hash function evaluation is indistinguishable from a random function.

Our construction at Section 6.2.2 implements a correlation-robust hash function using fixed-key AES. For security, it assumes that AES with a fixed key can be modeled as a random permutation [GKWY20]. Additionally, we use a different AES key for every client, which in the ideal cipher model leads to better concrete security [GKWWY20].

We note that for robustness, the analysis of [BBCGGI21] still assumes a random oracle to make the Idpf extractable. While XofFixedKeyAes128 has been shown to be differentiable from a random oracle [GKWWY20], there are no known attacks exploiting this difference. We also stress that even if the Idpf is not extractable, Poplar1 guarantees that every client can contribute to at most one prefix among the ones being evaluated by the helpers.

9.5. Choosing the Number of Proofs to Use for Prio3

TODO Add guidance for choosing PROOFS (Section 7.1.2) for Prio3. In particular when we go for a smaller field for a given circuit. See this for details.

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	Prio3SumVec	VDAF	Section 7.4.3
0x00000003	Prio3Histogram	VDAF	Section 7.4.4
0x00000004 to 0x00000FFF	reserved for Prio3	VDAF	n/a
0x00001000	Poplar1	VDAF	Section 8.4
0xFFFF0000 to 0xFFFFFFFF	reserved	n/a	n/a

Table 17: Unique identifiers for (V)DAFs.

TODO Add IANA considerations for the codepoints summarized in Table 17.

11. References

11.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/

RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

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

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

[TurboSHAKE] Viguier, B., Wong, D., Van Assche, G., Dang, Q., and J. Daemen, "KangarooTwelve and TurboSHAKE", Work in Progress, Internet-Draft, draft-irtf-cfrg-kangarootwelve-11, 20 June 2023, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-kangarootwelve-11>.

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-08, 23 October 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-ppm-dap-08>.

[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>.

[GKWWY20] Guo, C., Katz, J., Wang, X., Weng, C., and Y. Yu, "Better concrete security for half-gates garbling (in the multi-instance setting)", CRYPTO 2020 , 2020, <https://link.springer.com/chapter/10.1007/978-3-030-56880-1_28>.

[GKWY20] Guo, C., Katz, J., Wang, X., and Y. Yu, "Efficient and Secure Multiparty Computation from Fixed-Key Block Ciphers", S&P 2020 , 2020, <https://eprint.iacr.org/2019/074>.

[MRH04] Maurer, U., Renner, R., and C. Holenstein, "Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology", In TCC 2004: Theory of Cryptography, pages 21-39, DOI 10.1007/978-3-540-24638-1_2, February 2004, <https://doi.org/10.1007/978-3-540-24638-1_2>.

[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 Junye Chen, Henry Corrigan-Gibbs, Armando Faz-Hernández, Simon Friedberger, Tim Geoghegan, Albert Liu, Brandon Pitman, Mariana Raykova, Jacob Rothstein, Shan Wang, Xiao Wang, and Christopher Wood for useful feedback on and contributions to the spec.

Test Vectors

[TO BE REMOVED BY RFC EDITOR: 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, the random inputs of randomized algorithms were fixed to the byte sequence starting with 0, incrementing by 1, and wrapping at 256:

```
0, 1, 2, ..., 255, 0, 1, 2, ...
```

Prio3Count

TODO Copy the machine readable vectors from the source repository (https://github.com/cfrg/draft-irtf-cfrg-vdaf/tree/main/poc/test_vec) and format them for humans.

Prio3Sum

TODO Copy the machine readable vectors from the source repository (https://github.com/cfrg/draft-irtf-cfrg-vdaf/tree/main/poc/test_vec) and format them for humans.

Prio3SumVec

TODO Copy the machine readable vectors from the source repository (https://github.com/cfrg/draft-irtf-cfrg-vdaf/tree/main/poc/test_vec) and format them for humans.

Prio3Histogram

TODO Copy the machine readable vectors from the source repository (https://github.com/cfrg/draft-irtf-cfrg-vdaf/tree/main/poc/test_vec) and format them for humans.

Poplar1

TODO Copy the machine readable vectors from the source repository (https://github.com/cfrg/draft-irtf-cfrg-vdaf/tree/main/poc/test_vec) and format them for humans.

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