Workgroup: CFRG
Internet-Draft: draft-irtf-cfrg-vdaf-01
Published: 26 May 2022
Intended Status: Informational
Expires: 27 November 2022
Authors: R. L. Barnes (Cisco)
C. Patton (Cloudflare)
P. Schoppmann (Google)

Verifiable Distributed Aggregation Functions

Abstract

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

Discussion Venues

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

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

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

Status of This Memo

Table of Contents

1. Introduction

The ubiquity of the Internet makes it an ideal platform for measurement of large-scale phenomena, whether public health trends or the behavior of computer systems at scale. There is substantial overlap, however, between information that is valuable to measure and information that users consider private.

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

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

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

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

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

VDAF schemes achieve their privacy goal by distributing the computation of the aggregate among a number of non-colluding aggregation servers. As long as a subset of the servers executes the protocol honestly, VDAFs guarantee that no input is ever accessible to any party besides the client that submitted it. At the same time, VDAFs are "verifiable" in the sense that malformed inputs that would otherwise garble the output of the computation can be detected and removed from the set of inputs.

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

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

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

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

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

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

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

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

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

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

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

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

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

  In Section 7 we describe Prio3, a VDAF that follows the same overall framework as the original Prio protocol, but incorporates

techniques introduced in [BBCGGI19] that result in significant performance gains.

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

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

Finally, perhaps the most complex aspect of schemes like Prio3 and Poplar1 is the process by which the client-generated measurements are prepared for aggregation. Because these constructions are based on secret sharing, the servers will be required to exchange some amount of information in order to verify the measurement is valid and can be aggregated. Depending on the construction, this process may require multiple round trips over the network.

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

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

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

1.1. Change Log

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

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 is defined, which algorithms are free to use as desired. Its value **SHALL** be b'vdaf-01'.

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).

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

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

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

3. Overview

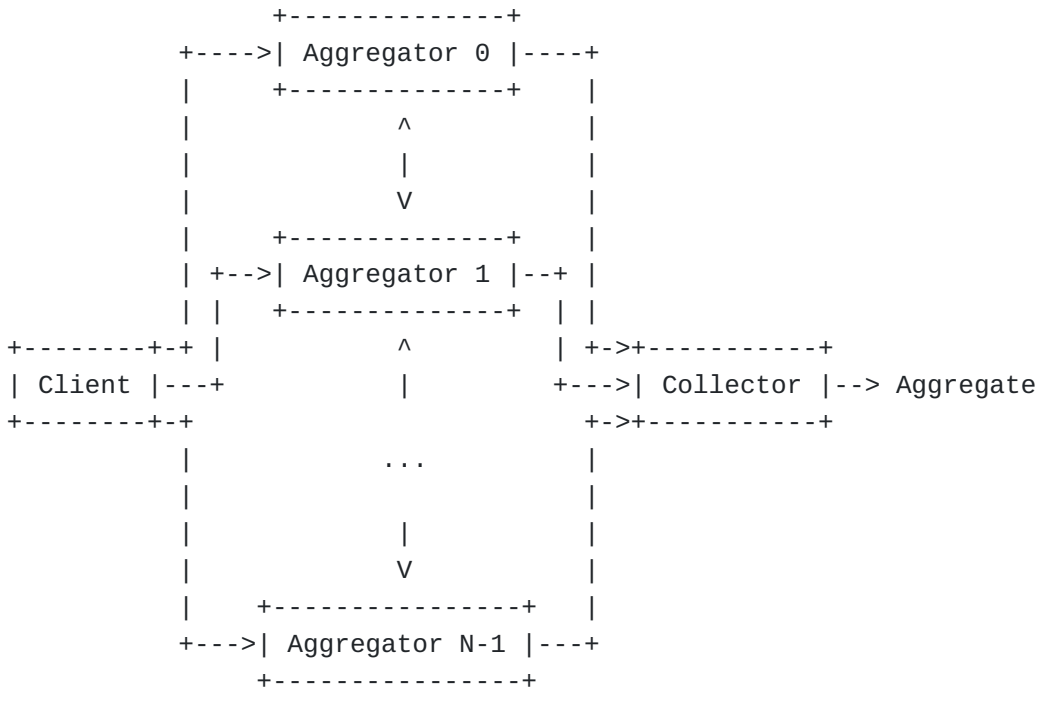

Figure 1: Overall data flow of a (V)DAF

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

* To submit an individual measurement, the Client shards the measurement into "input shares" and sends one input share to each Aggregator.

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

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

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

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

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

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

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

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

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

authenticated, with the exception that Clients submitting measurements may be anonymous.

4. Definition of DAFs

By way of a gentle introduction to VDAFs, this section describes a simpler class of schemes called Distributed Aggregation Functions (DAFs). Unlike VDAFs, DAFs do not provide verifiability of the computation. Clients must therefore be trusted to compute their input shares correctly. Because of this fact, the use of a DAF is **NOT RECOMMENDED** for most applications. See Section 9 for additional discussion.

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

Execution of a DAF has four distinct stages:

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

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

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

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

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

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

Parameter	Description
SHARES	Number of input shares into which each measurement is sharded
Measurement	Type of each measurement
AggParam	Type of aggregation parameter
OutShare	Type of each output share
AggResult	Type of the aggregate result

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

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

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

4.1. Sharding

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

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

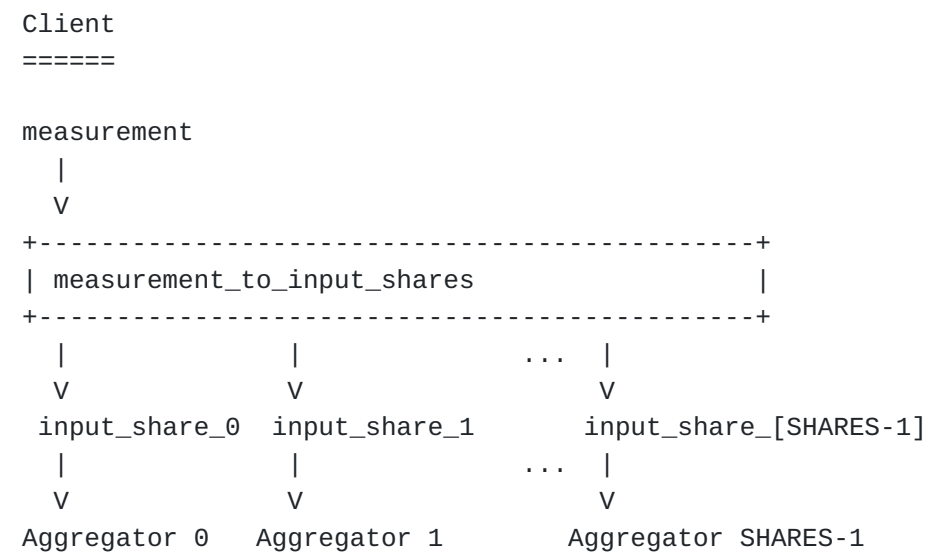

Figure 2: The Client divides its measurement into input shares and distributes them to the Aggregators.

4.2. Preparation

Once an Aggregator has received an input share form a Client, 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, input_share: Bytes) -> OutShare is the deterministic preparation algorithm. It takes as input an input share generated by a Client, the Aggregator's unique identifier, and the aggregation parameter selected by the Collector and returns an output share.

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

4.3. Aggregation

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

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

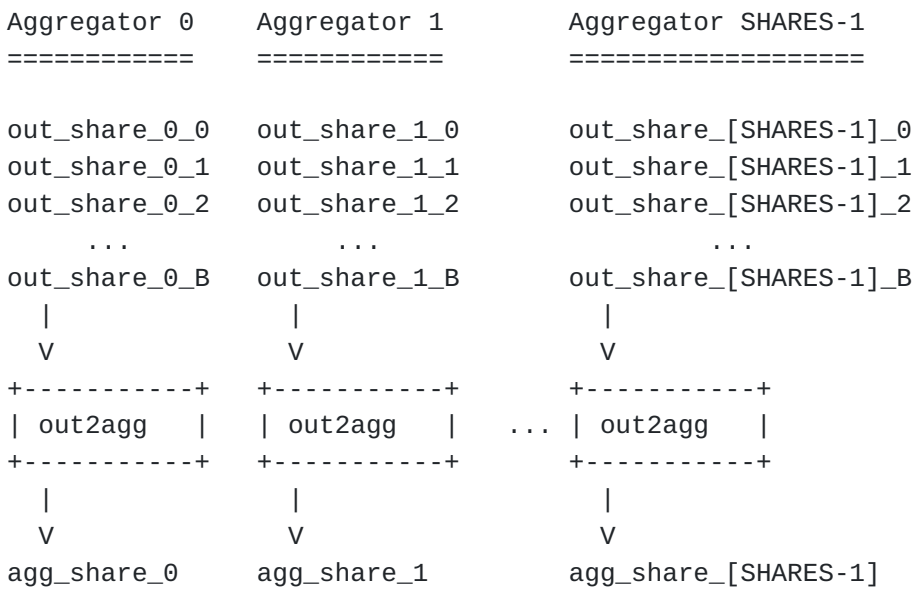

Figure 3: Aggregation of output shares. `B` indicates the number of measurements in the batch.

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

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

4.4. Unsharding

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

*Daf.agg_shares_to_result(agg_param: AggParam, agg_shares: Vec[Bytes]) -> 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. This algorithm is deterministic.

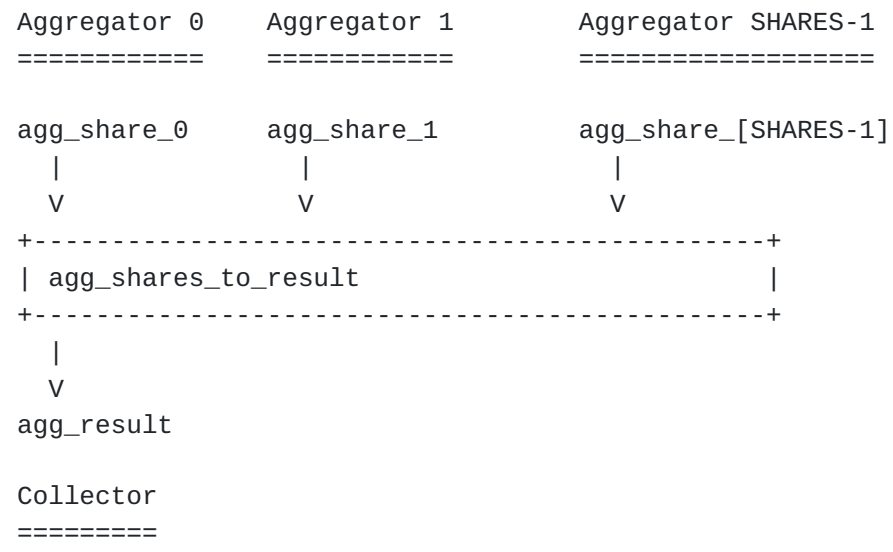

Figure 4: Computation of the final aggregate result from aggregate shares.

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.

4.5. Execution of a DAF

Securely executing a DAF involves emulating the following procedure.

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

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

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

    # Collector unshards the aggregate result.
    agg_result = Daf.agg_shares_to_result(agg_param, agg_shares)
    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
VERIFY_KEY_SIZE	Size (in bytes) of the verification key (Section 5.2)
ROUNDS	Number of rounds of communication during the Preparation stage (Section 5.2)
SHARES	Number of input shares into which each measurement is sharded (Section 5.1)
Measurement	Type of each measurement
AggParam	Type of aggregation parameter
Prep	State of each Aggregator during Preparation (Section 5.2)
OutShare	Type of each output share
AggResult	Type of the aggregate result

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

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

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

5.1. Sharding

Sharding is syntactically identical to DAFs (cf. Section 4.1):

*Vdaf.measurement_to_input_shares(measurement: Measurement) -> Vec[Bytes] is the randomized input-distribution algorithm run by each Client. It consumes the measurement and produces a 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.

5.2. Preparation

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

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

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

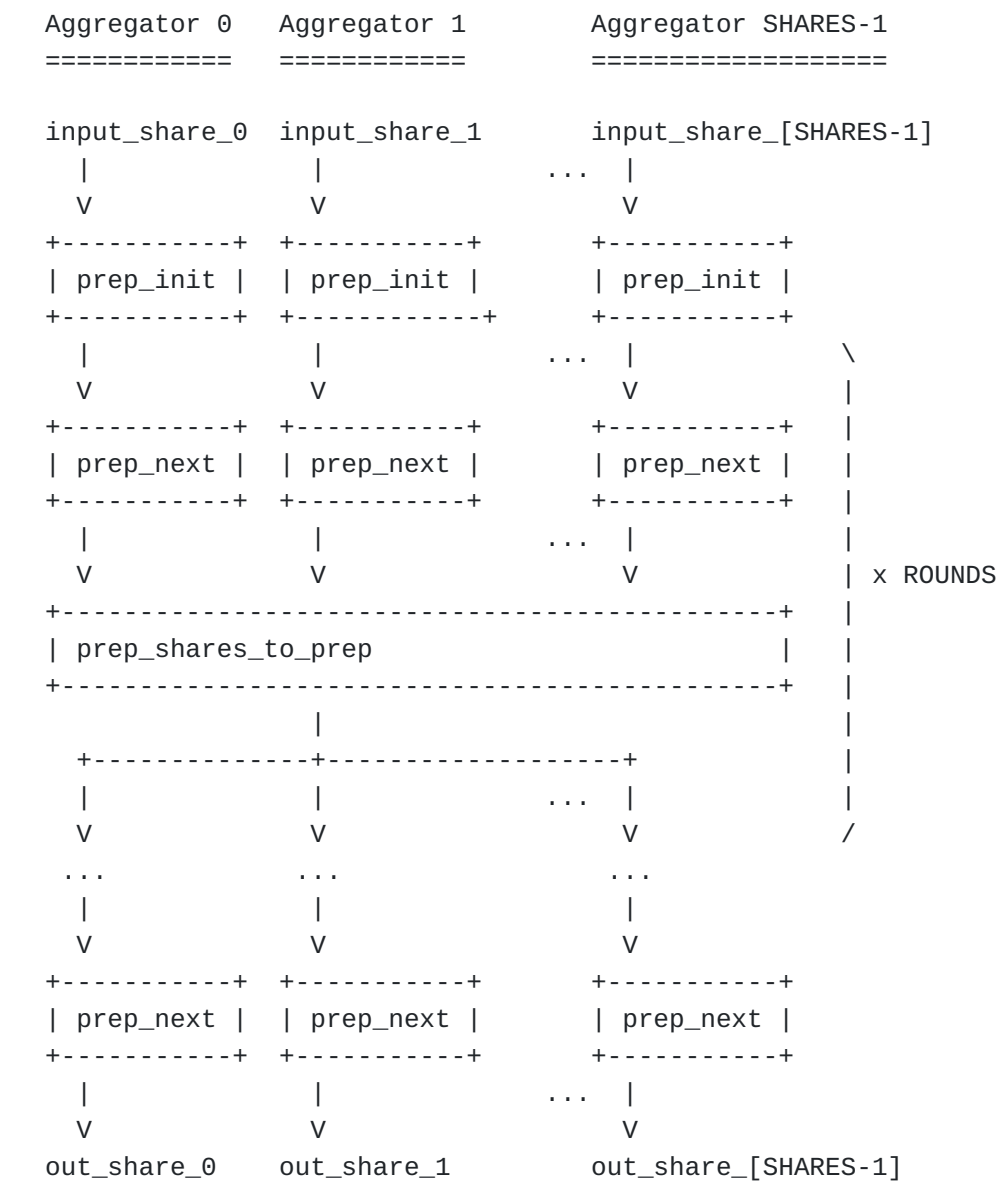

Figure 6: VDAF preparation process on the input shares for a single measurement. At the end of the computation, each Aggregator holds an output share or an error.

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

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

generated by the client (input_share). Its output is the Aggregator's initial preparation state.

The length of verify_key **MUST** be VERIFY_KEY_SIZE. It is up to the high level protocol in which the VDAF is used to arrange for the distribution of the verification key among the Aggregators prior to the start of this phase of VDAF evaluation.

OPEN ISSUE What security properties do we need for this key exchange? See issue#18.

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.

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

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

In effect, each Aggregator moves through a linear state machine with ROUNDS+1 states. The Aggregator enters the first state on using the initialization algorithm, and the update algorithm advances the Aggregator to the next state. Thus, in addition to defining the number of rounds (ROUNDS), a VDAF instance defines the state of the Aggregator after each round.

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

The preparation-state update accomplishes two tasks: recovery of output shares from the input shares and ensuring that the recovered output shares are valid. The abstraction boundary is drawn so that an Aggregator only recovers an output share if it is deemed valid (at least, based on the Aggregator's view of the protocol). Another way to draw this boundary would be to have the Aggregators recover output shares first, then verify that they are valid. However, this

would allow the possibility of misusing the API by, say, aggregating an invalid output share. Moreover, in protocols like Prio+ [AGJOP21] based on oblivious transfer, it is necessary for the Aggregators to interact in order to recover aggregatable output shares at all.

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

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

5.3. Aggregation

VDAF Aggregation is identical to DAF Aggregation (cf. Section 4.3):

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

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

5.4. Unsharding

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

* Vdaf.agg_shares_to_result(agg_param: AggParam, agg_shares: Vec[Bytes]) -> 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. This algorithm is deterministic.

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

5.5. Execution of a VDAF

Secure execution of a VDAF involves simulating the following procedure.

```
def run_vdaf(Vdaf,
             agg_param: Vdaf.AggParam,
             nonces: Vec[Bytes],
             measurements: Vec[Vdaf.Measurement]):
    # Generate the long-lived verification key.
    verify_key = gen_rand(Vdaf.VERIFY_KEY_SIZE)

    out_shares = []
    for (nonce, measurement) in zip(nonces, measurements):
        # Each Client shards its measurement into input shares.
        input_shares = Vdaf.measurement_to_input_shares(measurement)

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

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

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

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

```
# Collector unshards the aggregate.
agg_result = Vdaf.agg_shares_to_result(agg_param, agg_shares)
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 4.5, the secure execution of a VDAF requires the application to instantiate secure channels between each of the protocol participants.

6. Preliminaries

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

6.1. Finite Fields

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

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

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

A concrete Field also implements the following class methods:

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

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

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

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

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

* Field(integer: Unsigned) returns integer represented as a field element. If integer >= Field.MODULUS, then integer is first reduced modulo Field.MODULUS.

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

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

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

    vec = []
    for i in range(0, len(encoded), L):
        encoded_x = encoded[i:i+L]
        x = Field(OS2IP(encoded_x))
        vec.append(x)
    return vec
```

Figure 8: Derived class methods for finite fields.

6.1.1. Auxiliary Functions

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

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

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

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

Figure 9: Common functions for finite fields.

6.1.2. FFT-Friendly Fields

Some VDAFs require fields that are suitable for efficient computation of the discrete Fourier transform. (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.

FFT-friendly fields also define the following parameter:

* GEN_ORDER: Unsigned is the order of a multiplicative subgroup generated by Field.gen(). This value **MUST** be a power of 2.

6.1.3. Parameters

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

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

Table 3: Field64, an FFT-friendly field.

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

Table 4: Field128, an FFT-friendly field.

6.2. Pseudorandom Generators

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

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

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

A concrete Prg implements the following class method:

*Prg(seed: Bytes, info: Bytes) constructs an instance of Prg from the given seed and info string. The seed **MUST** be of length SEED_SIZE and **MUST** be generated securely (i.e., it is either the output of gen_rand or a previous invocation of the PRG). The info string is used for domain separation.

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

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

```
# Derive a new seed.
def derive_seed(Prg, seed: Bytes, info: Bytes) -> bytes:
    prg = Prg(seed, info)
    return prg.next(Prg.SEED_SIZE)

# Expand a seed into a vector of Field elements.
def expand_into_vec(Prg,
                    Field,
                    seed: Bytes,
                    info: Bytes,
                    length: Unsigned):
    m = next_power_of_2(Field.MODULUS) - 1
    prg = Prg(seed, info)
    vec = []
    while len(vec) < length:
        x = OS2IP(prg.next(Field.ENCODED_SIZE))
        x &= m
        if x < Field.MODULUS:
            vec.append(Field(x))
    return vec
```

Figure 10: Derived class methods for PRGs.

6.2.1. PrgAes128

OPEN ISSUE Phillipp points out that a fixed-key mode of AES may be more performant (https://eprint.iacr.org/2019/074.pdf). See issue#32.

Our first construction, PrgAes128, converts a blockcipher, namely AES-128, into a PRG. Seed expansion involves two steps. In the first step, CMAC [RFC4493] is applied to the seed and info string to get a fresh key. In the second step, the fresh key is used in CTR-mode to produce a key stream for generating the output. A fixed initialization vector (IV) is used.

```
class PrgAes128:

    SEED_SIZE: Unsigned = 16

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

        # Use CMAC as a pseudorandom function to derive a key.
        self.key = AES128-CMAC(seed, info)

    def next(self, length):
        self.length_consumed += length

        # CTR-mode encryption of the all-zero string of the desired
        # length and using a fixed, all-zero IV.
        stream = AES128-CTR(key, zeros(16), zeros(self.length_consumed))
        return stream[-length:]
```

Figure 11: Definition of PRG PrgAes128.

7. Prio3

NOTE This construction has not undergone significant security analysis.

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

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

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

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

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

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

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

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

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

7.1. Fully Linear Proof (FLP) Systems

Conceptually, an FLP is a two-party protocol executed by a prover and a verifier. In actual use, however, the prover's computation is carried out by the Client, and the verifier's computation is distributed among the Aggregators. The Client generates a "proof" of its input's validity and distributes shares of the proof to the Aggregators. Each Aggregator then performs some a computation on its input share and proof share locally and sends the result to the

other Aggregators. Combining the exchanged messages allows each Aggregator to decide if it holds a share of a valid input. (See Section 7.2 for details.)

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

The parameters provided by a concrete FLP are listed in Table 5.

Parameter	Description
PROVE_RAND_LEN	Length of the prover randomness, the number of random field elements consumed by the prover when generating a proof
QUERY_RAND_LEN	Length of the query randomness, the number of random field elements consumed by the verifier
JOINT_RAND_LEN	Length of the joint randomness, the number of random field elements consumed by both the prover and verifier
INPUT_LEN	Length of the encoded measurement (Section 7.1.1)
OUTPUT_LEN	Length of the aggregatable output (Section 7.1.1)
PROOF_LEN	Length of the proof
VERIFIER_LEN	Length of the verifier message generated by querying the input and proof
Measurement	Type of the measurement
Field	As defined in (Section 6.1)

Table 5: Constants and types defined by a concrete FLP.

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

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

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

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

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

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

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

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

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

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

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

Figure 12: Execution of an FLP.

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

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

7.1.1. Encoding the Input

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

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

For some FLPs, the encoded input also includes redundant field elements that are useful for checking the proof, but which are not needed after the proof has been checked. An example is the "integer sum" data type from [CGB17] in which an integer in range [0, 2^k) is encoded as a vector of k field elements (this type is also defined in Section 7.4). After consuming this vector, all that is needed is the integer it represents. Thus the FLP defines an algorithm for truncating the input to the length of the aggregated output:

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

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

7.2. Construction

This section specifies Prio3, an implementation of the Vdaf interface (Section 5). It has two generic parameters: an Flp (Section 7.1) and a Prg (Section 6.2). The associated constants and types required by the Vdaf interface are defined in Table 6. The methods required for sharding, preparation, aggregation, and unsharding are described in the remaining subsections.

Parameter	Value
VERIFY_KEY_SIZE	Prg.SEED_SIZE
ROUNDS	1
SHARES	in [2, 255)
Measurement	Flp.Measurement
AggParam	None
Prep	Tuple[Vec[Flp.Field], Optional[Bytes], Bytes]
OutShare	Vec[Flp.Field]

Parameter	Value
AggResult	Vec[Unsigned]

Table 6: Associated parameters for the Prio3 VDAF.

7.2.1. Sharding

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

The input-distribution algorithm involves the following steps:

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

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

The code refers to a pair of auxiliary functions for encoding the shares. These are called encode_leader_share and encode_helper_share respectively and they are described in Section 7.2.5.

```
def measurement_to_input_shares(Prio3, measurement):
    # Domain separation tag for PRG info string
    dst = VERSION + b' prio3'
    inp = Prio3.Flp.encode(measurement)
    k_joint_rand = zeros(Prio3.Prg.SEED_SIZE)

    # Generate input shares.
    leader_input_share = inp
    k_helper_input_shares = []
    k_helper_blinds = []
    k_helper_hints = []
    for j in range(Prio3.SHARES-1):
        k_blind = gen_rand(Prio3.Prg.SEED_SIZE)
        k_share = gen_rand(Prio3.Prg.SEED_SIZE)
        helper_input_share = Prio3.Prg.expand_into_vec(
            Prio3.Flp.Field,
            k_share,
            dst + byte(j+1),
            Prio3.Flp.INPUT_LEN
        )
        leader_input_share = vec_sub(leader_input_share,
                                     helper_input_share)
        encoded = Prio3.Flp.Field.encode_vec(helper_input_share)
        k_hint = Prio3.Prg.derive_seed(k_blind,
                                       byte(j+1) + encoded)
        k_joint_rand = xor(k_joint_rand, k_hint)
        k_helper_input_shares.append(k_share)
        k_helper_blinds.append(k_blind)
        k_helper_hints.append(k_hint)
    k_leader_blind = gen_rand(Prio3.Prg.SEED_SIZE)
    encoded = Prio3.Flp.Field.encode_vec(leader_input_share)
    k_leader_hint = Prio3.Prg.derive_seed(k_leader_blind,
                                          byte(0) + encoded)
    k_joint_rand = xor(k_joint_rand, k_leader_hint)

    # Finish joint randomness hints.
    for j in range(Prio3.SHARES-1):
        k_helper_hints[j] = xor(k_helper_hints[j], k_joint_rand)
    k_leader_hint = xor(k_leader_hint, k_joint_rand)

    # Generate the proof shares.
    prove_rand = Prio3.Prg.expand_into_vec(
        Prio3.Flp.Field,
        gen_rand(Prio3.Prg.SEED_SIZE),
        dst,
        Prio3.Flp.PROVE_RAND_LEN
    )
    joint_rand = Prio3.Prg.expand_into_vec(
        Prio3.Flp.Field,
```

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

input_shares = []
input_shares.append(Prio3.encode_leader_share(
    leader_input_share,
    leader_proof_share,
    k_leader_blind,
    k_leader_hint,
))
for j in range(Prio3.SHARES-1):
    input_shares.append(Prio3.encode_helper_share(
        k_helper_input_shares[j],
        k_helper_proof_shares[j],
        k_helper_blinds[j],
        k_helper_hints[j],
    ))
return input_shares
```

Figure 13: Input-distribution algorithm for Prio3.

7.2.2. Preparation

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

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

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

> NOTE This optimization somewhat diverges from Section 6.2.3 of [BBCGGI19]. Security analysis is needed.

The algorithms required for preparation are defined as follows. These algorithms make use of encoding and decoding methods defined in Section 7.2.5.

```
def prep_init(Prio3,
              verify_key, agg_id, _agg_param, nonce, input_share):
    # Domain separation tag for PRG info string
    dst = VERSION + b'prio3'

    (input_share, proof_share, k_blind, k_hint) = \
        Prio3.decode_leader_share(input_share) if agg_id == 0 else \
        Prio3.decode_helper_share(dst, agg_id, input_share)

    out_share = Prio3.Flp.truncate(input_share)

    k_query_rand = Prio3.Prg.derive_seed(verify_key, byte(255) + nonce)
    query_rand = Prio3.Prg.expand_into_vec(
        Prio3.Flp.Field,
        k_query_rand,
        dst,
        Prio3.Flp.QUERY_RAND_LEN
    )
    joint_rand, k_joint_rand, k_joint_rand_share = [], None, None
    if Prio3.Flp.JOINT_RAND_LEN > 0:
        encoded = Prio3.Flp.Field.encode_vec(input_share)
        k_joint_rand_share = Prio3.Prg.derive_seed(
            k_blind, byte(agg_id) + encoded)
        k_joint_rand = xor(k_hint, k_joint_rand_share)
        joint_rand = Prio3.Prg.expand_into_vec(
            Prio3.Flp.Field,
            k_joint_rand,
            dst,
            Prio3.Flp.JOINT_RAND_LEN
        )
    verifier_share = Prio3.Flp.query(input_share,
                                     proof_share,
                                     query_rand,
                                     joint_rand,
                                     Prio3.SHARES)

    prep_msg = Prio3.encode_prep_share(verifier_share,
                                       k_joint_rand_share)
    return (out_share, k_joint_rand, prep_msg)

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

    if inbound is None:
        return (prep, prep_msg)

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

```
    return out_share

def prep_shares_to_prep(Prio3, _agg_param, prep_shares):
    verifier = Prio3.Flp.Field.zeros(Prio3.Flp.VERIFIER_LEN)
    k_joint_rand_check = zeros(Prio3.Prg.SEED_SIZE)
    for encoded in prep_shares:
        (verifier_share, k_joint_rand_share) = \
            Prio3.decode_prep_share(encoded)

        verifier = vec_add(verifier, verifier_share)

        if Prio3.Flp.JOINT_RAND_LEN > 0:
            k_joint_rand_check = xor(k_joint_rand_check,
                                     k_joint_rand_share)

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

    return Prio3.encode_prep_msg(k_joint_rand_check)
```

Figure 14: Preparation state for Prio3.

7.2.3. Aggregation

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

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

Figure 15: Aggregation algorithm for Prio3.

7.2.4. Unsharding

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

```
def agg_shares_to_result(Prio3, _agg_param, agg_shares):
    agg = Prio3.Flp.Field.zeros(Prio3.Flp.OUTPUT_LEN)
    for agg_share in agg_shares:
        agg = vec_add(agg, Prio3.Flp.Field.decode_vec(agg_share))
    return list(map(lambda x: x.as_unsigned(), agg))
```

Figure 16: Computation of the aggregate result for Prio3.

7.2.5. Auxiliary Functions

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

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

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

def decode_helper_share(Prio3, dst, agg_id, encoded):
    l = Prio3.Prg.SEED_SIZE
    k_input_share, encoded = encoded[:l], encoded[l:]
    input_share = Prio3.Prg.expand_into_vec(Prio3.Flp.Field,
                                            k_input_share,
                                            dst + byte(agg_id),
                                            Prio3.Flp.INPUT_LEN)
```

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

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

def decode_prep_share(Prio3, encoded):
    l = Prio3.Flp.Field.ENCODED_SIZE * Prio3.Flp.VERIFIER_LEN
    encoded_verifier, encoded = encoded[:l], encoded[l:]
    verifier = Prio3.Flp.Field.decode_vec(encoded_verifier)
    k_joint_rand = None
    if Prio3.Flp.JOINT_RAND_LEN > 0:
        l = Prio3.Prg.SEED_SIZE
        k_joint_rand, encoded = encoded[:l], encoded[l:]
    if len(encoded) != 0:
        raise ERR_DECODE
    return (verifier, k_joint_rand)

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

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

Figure 17: Helper functions required for Prio3.

7.3. A General-Purpose FLP

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

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

7.3.1. Overview

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

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

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

(Note that, for this equality to hold, it may be necessary to scale any constants in the circuit by SHARES.) However this is not the case if C is not-affine (i.e., it contains at least one multiplication gate whose operands are non-constant). In the proof system of [BBCGGI19], the proof is designed to allow the (distributed) verifier to compute the non-affine operations using only linear operations on (its share of) the input and proof.

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

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

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

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

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

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

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

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

The prover constructs from wire and alpha a polynomial that, when evaluated at alpha[k-1], produces the output of the kth call to the

gadget. Let us call this the "gadget polynomial". Polynomial evaluation is linear, which means that, in the distributed setting, the Client can disseminate additive shares of the gadget polynomial that the Aggregators then use to compute additive shares of each gadget output, allowing each Aggregator to compute its share of C(x) locally.

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

7.3.1.1. Extensions

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

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

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

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

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

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

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

7.3.2. Validity Circuits

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

A concrete Valid defines the following parameters:

Parameter	Description
GADGETS	A list of gadgets
GADGET_CALLS	Number of times each gadget is called
INPUT_LEN	Length of the input
OUTPUT_LEN	Length of the aggregatable output
JOINT_RAND_LEN	Length of the random input
Measurement	The type of measurement
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).

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

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

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

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

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

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

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

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

Figure 18: Derived methods for validity circuits.

7.3.3. Construction

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

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

The FLP parameters for FlpGeneric are defined in Table 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
INPUT_LEN	Valid.INPUT_LEN
OUTPUT_LEN	Valid.OUTPUT_LEN
PROOF_LEN	Valid.proof_len() (see Section 7.3.2)
VERIFIER_LEN	Valid.verifier_len() (see Section 7.3.2)
Measurement	Valid.Measurement
Field	Valid.Field

Table 8: FLP Parameters of FlpGeneric.

7.3.3.1. Proof Generation

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

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

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

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

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

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

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

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

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

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

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

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

7.3.3.2. Query Generation

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

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

2. Partition proof into the subvectors seed_1, coeff_1, ..., seed_H, coeff_H defined in Section 7.3.3.1.

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

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

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

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

*Let y_i = poly_gadget_i(t).

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

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

7.3.3.3. Decision

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

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

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

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

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

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

7.3.3.4. Encoding

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

7.4. Instantiations

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

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

7.4.1. Prio3Aes128Count

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 PrgAes128 (Section 6.2.1) as its PRG. Its validity circuit, denoted Count, uses Field64 (Table 3) as its finite field. Its gadget, denoted Mul, is the degree-2, arity-2 gadget defined as

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

The validity circuit is defined as

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

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

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

Table 9: Parameters of validity circuit Count.

7.4.2. Prio3Aes128Sum

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 PrgAes128 (Section 6.2.1) as its PRG. Its validity circuit, denoted Sum, uses Field128 (Table 4) as its finite field. The measurement is encoded as a length-bits vector of field elements, where the lth element of the vector represents the lth bit of the summand:

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

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

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

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

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

The validity circuit is defined as

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

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

Table 10: Parameters of validity circuit Sum.

7.4.3. Prio3Aes128Histogram

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

This instance of Prio3 uses PrgAes128 (Section 6.2.1) as its PRG. Its validity circuit, denoted Histogram, uses Field128 (Table 4) as its finite field. The measurement is encoded as a one-hot vector representing the bucket into which the measurement falls (let bucket denote a sequence of monotonically increasing integers):

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

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

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

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

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

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

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

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

Table 11: Parameters of validity circuit Histogram.

8. Poplar1

TODO Update this section in light of removing the public parameter and replacing the verification parameter.

NOTE The spec for Poplar1 is still a work-in-progress. A partial implementation can be found at https://github.com/divviup/libprio-rs/blob/main/src/vdaf/poplar1.rs. The verification logic is nearly complete, however as of this draft the code is missing the IDPF. An implementation of the IDPF can be found at https://github.com/google/distributed_point_functions/.

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

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

1. Each Clients runs the input-distribution algorithm on its n-bit string and sends an input share to each Aggregator.

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

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

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

5. Let H denote the set of prefixes that occurred at least t times. If the prefixes all have length BITS, then H is the set

of t-heavy-hitters. Otherwise compute the next set of candidate prefixes as follows. For each p in H, add add p || 0 and p || 1 to the set. Repeat step 3 with the new set of candidate prefixes.

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

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

Our VDAF 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.

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. The Client specifies an index alpha and a pair of values beta, one for each "level" 1 <= l <= BITS. The key generation generates two IDPF keys, one for each Aggregator. When evaluated at index 0 <= x < 2^l, each IDPF share returns an additive share of beta[l] if x is the l-bit prefix of alpha and shares of zero otherwise.

> CP What does it mean for x to be the l-bit prefix of alpha? We need to be a bit more precise here.
>
> CP Why isn't the domain size actually 2^(BITS+1), i.e., the number of nodes in a binary tree of height BITS (excluding the root)?

Each beta[l] is a pair of elements of a finite field. Each level **MAY** have different field parameters. Thus a concrete IDPF specifies associated types Field[1], Field[2], ..., and Field[BITS] defining, respectively, the field parameters at level 1, level 2, ..., and level BITS.

An IDPF is comprised of the following algorithms (let type Value[l] denote (Field[l], Field[l]) for each level l):

- idpf_gen(alpha: Unsigned, beta: (Value[1], ..., Value[BITS])) -> key: (IDPFKey, IDPFKey) is the randomized key-generation algorithm run by the client. Its inputs are the index alpha and the values beta. The value of alpha **MUST** be in range [0, 2^BITS).

- IDPFKey.eval(l: Unsigned, x: Unsigned) -> value: Value[l]) is deterministic, stateless key-evaluation algorithm run by each Aggregator. It returns the value corresponding to index x. The value of l **MUST** be in [1, BITS] and the value of x **MUST** be in range [2^(l-1), 2^l).

A concrete IDPF specifies a single associated constant:

- BITS: Unsigned is the length of each Client input.

A concrete IDPF also specifies the following associated types:

- Field[l] for each level 1 <= l <= BITS. Each defines the same methods and associated constants as Field in Section 7.

Note that IDPF construction of [BBCGGI21] uses one field for the inner nodes of the tree and a different, larger field for the leaf nodes. See [BBCGGI21], Section 4.3.

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.

8.2. Construction

The VDAF involves two rounds of communication (ROUNDS == 2) and is defined for two Aggregators (SHARES == 2).

8.2.1. Setup

The verification parameter is a symmetric key shared by both Aggregators. This VDAF has no public parameter.

```
def vdaf_setup():
  k_verify_init = gen_rand(SEED_SIZE)
  return (None, [(0, k_verify_init), (1, k_verify_init)])
```

Figure 19: The setup algorithm for poplar1.

8.2.1.1. Client

The client's input is an IDPF index, denoted alpha. The 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]. After evaluating their IDPF key shares on the set 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*a + b - k*a + c
```

and sends additive shares of a, b, c, A and B to the Aggregators. Putting everything together, the input-distribution algorithm is defined as follows. Function encode_input_share is defined in Section 8.2.5.

```
def measurement_to_input_shares(_, alpha):
  if alpha < 2**BITS: raise ERR_INVALID_INPUT

  # Prepare IDPF values.
  beta = []
  correlation_shares_0, correlation_shares_1 = [], []
  for l in range(1,BITS+1):
    (k, a, b, c) = Field[l].rand_vec(4)

    # Construct values of the form (1, k), where k
    # is a random field element.
    beta += [(1, k)]

    # Create secret shares of correlations to aid
    # the Aggregators' computation.
    A = -2*a+k
    B = a*a + b - a * k + c
    correlation_share = Field[l].rand_vec(5)
    correlation_shares_1.append(correlation_share)
    correlation_shares_0.append(
      [a, b, c, A, B] - correlation_share)

  # Generate IDPF shares.
  (key_0, key_1) = idpf_gen(alpha, beta)

  input_shares = [
    encode_input_share(key_0, correlation_shares_0),
    encode_input_share(key_1, correlation_shares_1),
  ]

  return input_shares
```

Figure 20: The input-distribution algorithm for poplar1.

TODO It would be more efficient to represent the shares of a, b, and c using PRG seeds as suggested in [BBCGGI21].

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 pair of vectors of field elements data_share and auth_share, The Aggregators use auth_share and the correlation shares provided by the Client to verify that their data_share vectors are additive shares of a one-hot vector.

CP Consider adding aggregation parameter as input to k_verify_rand derivation.

```
class PrepState:
  def __init__(verify_param, agg_param, nonce, input_share):
    (self.l, self.candidate_prefixes) = decode_indexes(agg_param)
    (self.idpf_key,
     self.correlation_shares) = decode_input_share(input_share)
    (self.party_id, k_verify_init) = verify_param
    self.k_verify_rand = get_key(k_verify_init, nonce)
    self.step = 'ready'

  def next(self, inbound: Optional[Bytes]):
    l = self.l
    (a_share, b_share, c_share,
     A_share, B_share) = correlation_shares[l-1]

    if self.step == 'ready' and inbound == None:
      # Evaluate IDPF on candidate prefixes.
      data_share, auth_share = [], []
      for x in self.candidate_prefixes:
        value = self.idpf_key.eval(l, x)
        data_share.append(value[0])
        auth_share.append(value[1])

      # Prepare first sketch verification message.
      r = Prg.expand_into_vec(
        Field[l], self.k_verify_rand, len(data_share))
      verifier_share_1 = [
         a_share + inner_product(data_share, r),
         b_share + inner_product(data_share, r * r),
         c_share + inner_product(auth_share, r),
      ]

      self.output_share = data_share
      self.step = 'sketch round 1'
      return verifier_share_1

    elif self.step == 'sketch round 1' and inbound != None:
      verifier_1 = Field[l].decode_vec(inbound)
      verifier_share_2 = [
        (verifier_1[0] * verifier_1[0] \
         - verifier_1[1] \
         - verifier_1[2]) * self.party_id \
        + A_share * verifier_1[0] \
        + B_share
      ]

      self.step = 'sketch round 2'
      return Field[l].encode_vec(verifier_share_2)

    elif self.step == 'sketch round 2' and inbound != None:
```

```
      verifier_2 = Field[l].decode_vec(inbound)
      if verifier_2 != 0: raise ERR_INVALID
      return Field[l].encode_vec(self.output_share)

    else: raise ERR_INVALID_STATE

def prep_shares_to_prep(agg_param, inbound: Vec[Bytes]):
  if len(inbound) != 2:
    raise ERR_INVALID_INPUT

  (l, _) = decode_indexes(agg_param)
  verifier = Field[l].decode_vec(inbound[0]) + \
             Field[l].decode_vec(inbound[1])

  return Field[l].encode_vec(verifier)
```

Figure 21: Preparation state for poplar1.

8.2.3. Aggregation

```
def out_shares_to_agg_share(agg_param, output_shares: Vec[Bytes]):
  (l, candidate_prefixes) = decode_indexes(agg_param)
  if len(output_shares) != len(candidate_prefixes):
    raise ERR_INVALID_INPUT

  agg_share = Field[l].zeros(len(candidate_prefixes))
  for output_share in output_shares:
    agg_share += Field[l].decode_vec(output_share)

  return Field[l].encode_vec(agg_share)
```

Figure 22: Aggregation algorithm for poplar1.

8.2.4. Unsharding

```
def agg_shares_to_result(agg_param, agg_shares: Vec[Bytes]):
  (l, _) = decode_indexes(agg_param)
  if len(agg_shares) != 2:
    raise ERR_INVALID_INPUT

  agg = Field[l].decode_vec(agg_shares[0]) + \
        Field[l].decode_vec(agg_shares[1]J)

  return Field[l].encode_vec(agg)
```

Figure 23: Computation of the aggregate result for poplar1.

8.2.5. Helper Functions

TODO Specify the following functionalities:

*encode_input_share is used to encode an input share, consisting of an IDPF key share and correlation shares.

*decode_input_share is used to decode an input share.

*decode_indexes(encoded: Bytes) -> (l: Unsigned, indexes: Vec[Unsigned]) decodes a sequence of indexes, i.e., candidate indexes for IDPF evaluation. The value of l **MUST** be in range [1, BITS] and indexes[i] **MUST** be in range [2^(l-1), 2^l) for all i. An error is raised if encoded cannot be decoded.

9. Security Considerations

NOTE: This is a brief outline of the security considerations. This section will be filled out more as the draft matures and security analyses are completed.

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.

Note that, to achieve robustness, it is important to ensure that the verification key distributed to the Aggregators (verify_key, see Section 8.2.1) is never revealed to the Clients.

It is also possible to consider a stronger form of robustness, where the attacker also controls a subset of Aggregators (see [BBCGGI19], Section 6.3). To satisfy this stronger notion of robustness, it is necessary to prevent the attacker from sharing the verification key with the Clients. It is therefore **RECOMMENDED** that the Aggregators generate verify_key only after a set of Client inputs has been collected for verification, and re-generate them for each such set of inputs.

In order to achieve robustness, the Aggregators **MUST** ensure that the nonces used to process the measurements in a batch are all unique.

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.

* 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 inputs for the remaining Clients. Applications can guard against these risks by adding additional controls on measurement 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.

10. IANA Considerations

This document makes no request of IANA.

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>.

[RFC4493] Song, JH., Poovendran, R., Lee, J., and T. Iwata, "The AES-CMAC Algorithm", RFC 4493, DOI 10.17487/RFC4493, June 2006, <https://www.rfc-editor.org/rfc/rfc4493>.

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

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

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-00, 4 May 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-ppm-dap-00>.

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

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

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

[EPK14] Erlingsson, Ú., Pihur, V., and A. Korolova, "RAPPOR: Randomized Aggregatable Privacy-Preserving Ordinal Response", CCS 2014 , 2014, <https://dl.acm.org/doi/10.1145/2660267.2660348>.

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

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

Acknowledgments

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

Test Vectors

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

Test vectors cover the generation of input shares and the conversion of input shares into output shares. Vectors specify the verification key, measurements, aggregation parameter, and any parameters needed to construct the VDAF. (For example, for Prio3AesSum, the user specifies the number of bits for representing each summand.)

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

Prio3Aes128Count

```
verify_key: "01010101010101010101010101010101"
upload_0:
  measurement: 1
  nonce: "01010101010101010101010101010101"
  input_share_0: >-
    ae5483343eb35a52fcb36a62271a7ddb47f09d0ea2c6613807f84ac2e16814c82bca
    bdc9db5080fdf4f4f778734644fc
  input_share_1: >-
    01010101010101010101010101010101010101010101010101010101010101010101
  round_0:
    prep_share_0: >-
      22ce013d3aaa7e7574ed01fe1d074cd845dfbbbc5901cabd487d4e2e228274cc
    prep_share_1: >-
      dd31fec1c555818c51ab7ccac14ca5b00aae1c33d835c76dfa9406011a92a8e9
    prep_message: >-
  out_share_0:
    - 12561809521056635474
  out_share_1:
    - 5884934548357948848
agg_share_0: >-
  ae5483343eb35a52
agg_share_1: >-
  51ab7ccac14ca5b0
agg_result: [1]
```

Prio3Aes128Sum

```
bits: 8
verify_key: "01010101010101010101010101010101"
upload_0:
  measurement: 100
  nonce: "01010101010101010101010101010101"
  input_share_0: >-
    ae5483353eb35a3371beec8f796e9afd086cb72d05a83a3dbefbe273acb0410787b1
    afba2065df5389011fd8963091e3004fa07fc91018af378da47c89abf1bd85047e40
    874e2cdc5f3bc48f363b89f746770a402a777bed31b5a10c7319b3908d72de0c6512
    15ba78d3cf681e07c564c0b4a9a4508df645bad8fef61e3ddf37fcb36a63271a7dbe
    47f09d0ea2c661396ad006d8915d149ad88f9b1cdb86e1d13d683c359b7ac899a245
    4316051e4e235dfd566f3459c336826555ed7f1baabf241e9a697d458912f3bd3778
    225e832b78cd4f17e57c9b9678cf6043894aff0d0f2e06828982ac3493ae5ded0c98
    86ea13d52bc0f209dc2f4e676c42b95b548a413f67b03ff18e9e6b699338400e9dff
    a800563abb495364acffc17126bf0bf8ff3c5caba82333e91352e03c637d44dc4db1
    59a1b19d8db4d5a3fce356f6f2fca4adc9bcf65bec8d4d962b2b40f7ea413aead099
    79d4958707bf4098bb28829b79e381aaad8f69b7f2e6c159bbcb342ef7df2d9c56a9
    06b171ab61b025b7c19aad8de495a8a97af2baab6d1240d30df417d1cc0fe7a90ada
    ad8115924c0987fe1d16abe0c8a3c297d58a3112b818df72a10a41b34aa6b4ae370b
    1340a6085c8dcd597eead5d2584fdb160f0a086a56ea6a7736666ae34d3012fdb2c2
    4af3d4b2a6ae735edfe837eaab1309eaa2d8273e7dbfe0fd4166d545ce8354e1237b
    48456715d12e38d02cd64c96b9daa01a2281d8a930817088c648b7c115e1550ada14
    b6072ada49be3c7e3f184db2461160d29937caa97db6020a5598063f1dc05653d1d3
    80b34e923bd7170eeeb811bfc3ce12c1df55cf552e986a823743fac4723a48bda6c1
    8ffec653c1f182890197e9fc74631dcbd0283c4258933c03aee9404f010101010101
    0101010101010101010101fb0c701f7c07b9407a4a7b77d1ea017e
  input_share_1: >-
    0101010101010101010101010101010101010101010101010101010101010101010101
    0101010101010101010101010101012d7667bffd0f81b078896503385f6f13
  round_0:
    prep_share_0: >-
      9f7aca77f790b930b46e8cd786ff1a239aa00e7aaaa734cc2bbcb121eb7c5bc00a
      ef22fd95a24cd1a0054bde0dba06062d7667bffd0f81b078896503385f6f13
    prep_share_1: >-
      60853588086f46b34b9173287900e5de2becb1bdb8a8009d2cdc258674f08e8157
      e3a202a38282c20e220e733ab61e4cfb0c701f7c07b9407a4a7b77d1ea017e
    prep_message: >-
      d67a17a0810838f002c31e74e9b56e6d
  out_share_0:
    - 242787699414660215404830418280405596120
  out_share_1:
    - 97494667506278247542035355087495170189
agg_share_0: >-
  b6a735c5636efee29c0c1455e0c0f7d8
agg_share_1: >-
  4958ca3a9c91010163f3ebaa1f3f088d
agg_result: [100]
```

Prio3Aes128Histogram

```
buckets: [1, 10, 100]
verify_key: "01010101010101010101010101010101"
upload_0:
  measurement: 50
  nonce: "01010101010101010101010101010101"
  input_share_0: >-
    ae5483353eb35a3371beec8f796e9afd086cb72d05a83a3dbefbe273acb0410787b1
    afba2065df5389011fd8963091e3004fa07fc91018af378da47c89abf1bdfcb36a63
    271a7dbe47f09d0ea2c6613956dfe44e1302160dd2ade0205aa0409225caf0f966df
    97691568169000ef0af27c0985636e34889bc3fef4df192d7ead56e0dd51187bdc66
    62505cbd2962843cf2a1929642367f32058c531a6c611d76441e4ba82d136ba4aab1
    6f2a612df63678d42d527e59d8a0b4cb2f07ed8aaf04199819a25fad1b8cad62fb2e
    c5a9bd78b2e013a50250c8bd44a15ad7d5edac35a58bed81a4088c72430afbd6fe34
    635a737cb7c4d29ffc9947b6b0fb8f3fdede9d8bd495b4d47e8400bded8aa53e4a5a
    2d063c6091c29613e044082b0555ce74c45b823aa8c5804aacdd3dc92a6ac0058755
    7770972dcdc37eefb42eef43a1b401010101010101010101010101010101d5bf864d
    e68bac19204e29697bf9504d
  input_share_1: >-
    01010101010101010101010101010101010101010101010101010101010101010101
    0101010101010101010101010101018e2e553b5e45c62e0ced57ec947c8627
  round_0:
    prep_share_0: >-
      93b9dd4a3b46d8941fe7524a5cf1cd47ff8ee9c0c2e9b8230b1b940b665263b7b1
      c4b370652a333ee774ec9cd379b6e78e2e553b5e45c62e0ced57ec947c8627
    prep_share_1: >-
      6c4622b5c4b9274fe018adb5a30e32baa310ddd3e5ed87892dca520d1bed7d0299
      8e38190652ba60a225e19211d77e22d5bf864de68bac19204e29697bf9504d
    prep_message: >-
      5b91d376b8ce6a372ca37e85ef85d66a
  out_share_0:
    - 231724485416847873323492487111470127869
    - 111983072749766693877657441957482498 63
    - 180368380143069850478496598824148046307
    - 413446761563421317675646300023681469
  out_share_1:
    - 108557881504090589623373286256430638340
    - 329084059645961793559100029172152516346
    - 159913986777868612468369174543752719903
    - 339868920159375041629190127067877084740
agg_share_0: >-
  ae5483353eb35a3371beec8f796e9afd086cb72d05a83a3dbefbe273acb0410787b1af
  ba2065df5389011fd8963091e3004fa07fc91018af378da47c89abf1bd
agg_share_1: >-
  51ab7ccac14ca5b08e41137086916504f79348d2fa57c5a641041d8c534fbefa784e50
  45df9a209076fee02769cf6e1fffb05f8036efe734c8725b8376540e44
agg_result: [0, 0, 1, 0]
```

Authors' Addresses

Richard L. Barnes
Cisco

Email: rlb@ipv.sx

Christopher Patton
Cloudflare

Email: chrispatton+ietf@gmail.com

Phillipp Schoppmann
Google

Email: schoppmann@google.com