Workgroup: CFRG
Internet-Draft: draft-patton-cfrg-vdaf-00
Published: 25 October 2021
Intended Status: Informational
Expires: 28 April 2022
Authors: C. Patton (Cloudflare, Inc.)
R.L. Barnes (Cisco)
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

Copyright Notice

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 aggregated output 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 4 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 applications like [I-D.draft-gpew-priv-ppm] 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, aggregators, and measurement collectors);

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

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

2. Providing cryptographers with design criteria that allow new constructions to be easily used by applications.

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 input 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 aggregators combine their additive shares to obtain the final aggregate.

  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 6 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 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 generalization of a 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 7 we describe a VDAF called hits that implements this functionality.

The remainder of this document is organized as follows: Section 3 gives a brief overview of VDAFs; Section 4 defines the syntax for VDAFs; Section 5 defines various functionalities that are common to our constructions; Section 7 describes the hits construction; Section 6 describes the prio3 construction; and Section 8 enumerates the security considerations for VDAFs.

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. Function parameters without type hints implicitly have type Bytes, an arbitrary byte string. A fatal error in a program (e.g., failure to parse one of the function parameters) is usually handled by raising an exception.

Some common functionalities:

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

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

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

3. Overview

In a 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:

* Clients are configured with public parameters for a set of aggregators.

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

* The aggregators verify the validity of the input shares, producing a set of "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 the validation process, each aggregator holds one output share.

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

* Each aggregator combine the output shares across inputs in the batch to compute "aggregate shares", i.e., shares 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.

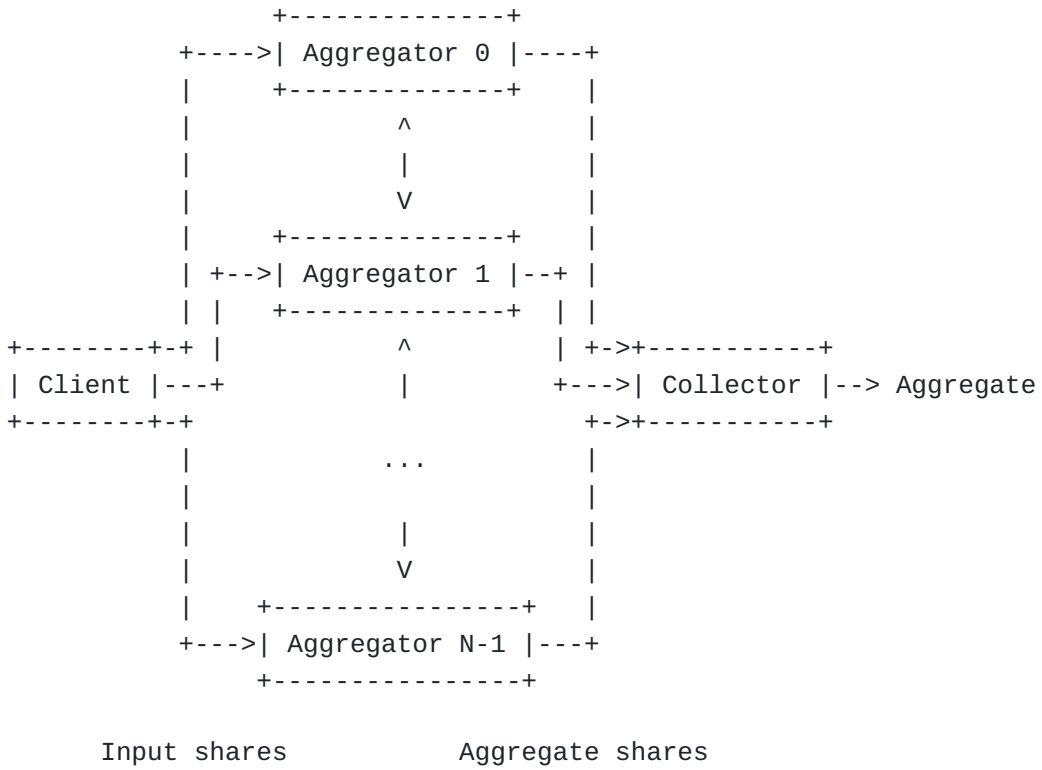

Figure 1: Overall data flow of a VDAF

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 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 inputs. Thus clients trust Aggregators not to collude (typically it is required that at least one Aggregator is honest), and Collectors trust Aggregators to properly verify Client inputs.

Within the bounds of the non-collusion requirements of a given VDAF 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 Aggregators 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 applications 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 VDAFs

A concrete VDAF specifies the algorithms involved in evaluating an aggregation function across a batch of inputs. This section specifies the interfaces of these algorithms as they would be exposed to applications.

The overall execution of a VDAF comprises the following steps:

* Setup - Generating shared parameters for the aggregators

* 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 set of output shares into an aggregate share

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

The setup algorithm is performed once for a given collection of Aggregators. Sharding and preparation are done once per measurement input. Aggregation and unsharding are done over a batch of inputs (more precisely, over the output shares recovered from those inputs).

Note that the preparation step performs two functions: Verification and conversion. Conversion translates input shares into output shares that are compatible with the aggregation function. Verification ensures that aggregating the recovered output shares will not lead to a garbled aggregate result.

A concrete VDAF scheme specifies implementations of these algorithms. In addition, a VDAF specifies the following constants:

- SHARES: Unsigned is the number of Aggregators for which the VDAF is defined.

- ROUNDS: Unsigned is the number of rounds of communication among the Aggregators before they recover output shares from a single set of input shares.

4.1. Setup

Before execution of the VDAF can begin, it is necessary to distribute long-lived parameters to the Client and Aggregators. The long-lived parameters are generated by the following algorithm:

- vdaf_setup() -> (public_param, verify_params: Vec[Bytes]) is the randomized setup algorithm used to generate the public parameter used by the Clients (public_param) and the verification parameters used by the Aggregators (verify_params). The length of the latter **MUST** be SHARES. The parameters are generated once and reused across multiple VDAF evaluations. In general, an Aggregator's verification parameter is considered secret and **MUST NOT** be revealed to the Clients, Collector or other Aggregators.

4.2. Sharding

In order to protect the privacy of its measurements, a VDAF client divides its measurements into "input shares". The measurement_to_input_shares method is executed by the client to produce these shares. One share is sent to each aggregator.

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

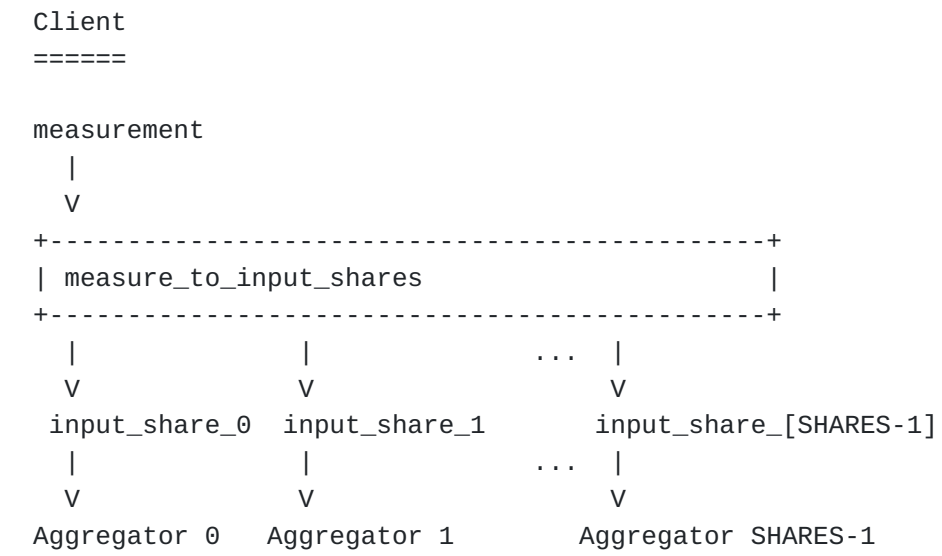

Figure 2: The Client divides its measurement input into input shares and distributes them to the Aggregators.

4.3. Preparation

To recover and verify the validity of output shares, the Aggregators interact with one another over ROUND rounds. In the first round, each aggregator produces a single output based on its input share. In subsequent rounds, each aggregator is given the outputs from all aggregators in the previous round. After the final round, either all aggregators hold an output share (if the output shares are valid) or all aggregators report an error (if they are not).

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.

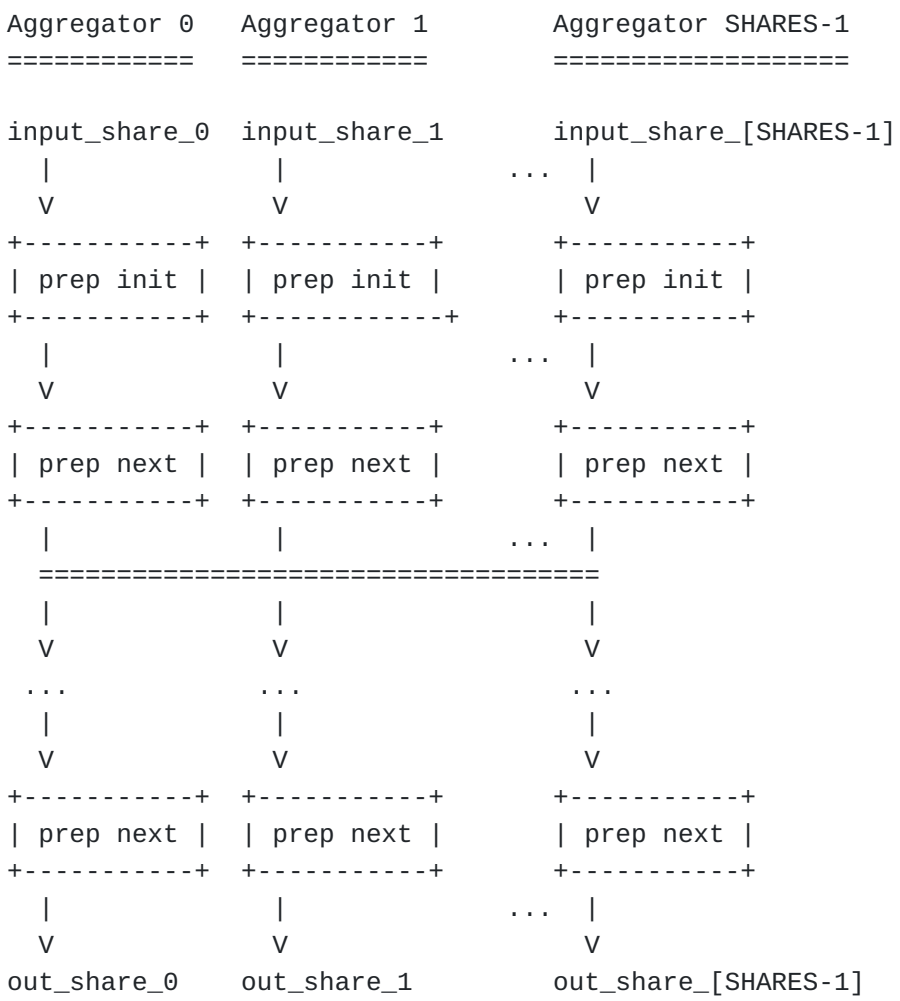

Figure 3: VDAF preparation process on the input shares for a single input. The === lines indicate the sharing of one round's outputs as the inputs to the next round. At the end of the protocol, each aggregator holds an output share or an error.

*PrepState(verify_param, agg_param, nonce, input_share) 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 aggregator's verification parameter (verify_param), the aggregation parameter (agg_param), the nonce provided by the environment (nonce, see Figure 6), and one of the input shares generated by the client (input_share). Its outputs is the Aggregator's initial preparation state.

*PrepState.next(inbound: Vec[Bytes]) -> outbound is the deterministic preparation-state update algorithm run by each Aggregator. It updates the Aggregator's preparation state (an instance of PrepState) and returns either its outbound message 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 sequence of

inbound messages from the previous round or, if this is the first round, an empty vector.

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 that are essential to most schemes: recovery of output shares from the input shares, and a multi-party computation carried out by the Aggregators to ensure that their output shares are valid. The VDAF 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 woulds 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 some protocols, like Prio+ [AGJOP21] it is necessary for the Aggregators to interact in order to recover 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).

4.4. Aggregation

Once an aggregator holds validated 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. This algorithm is performed locally at each Aggregator, without communication with the other Aggregators.

* output_to_aggregate_shares(agg_param, output_shares) is the deterministic aggregation algorithm. It is run by each Aggregator over the output shares it has computed over a batch of measurement inputs.

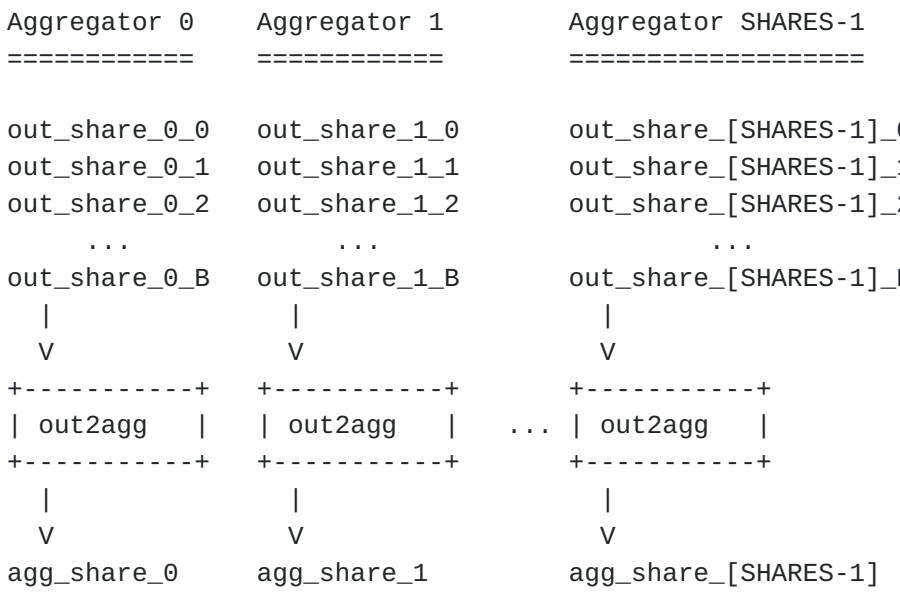

Figure 4: Aggregation of output shares. `B` indicates the number of measurements in the batch.

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

4.5. Unsharding

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

*aggregate_shares_to_result(agg_param, agg_shares: Vec[Bytes]) -> agg_result 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.

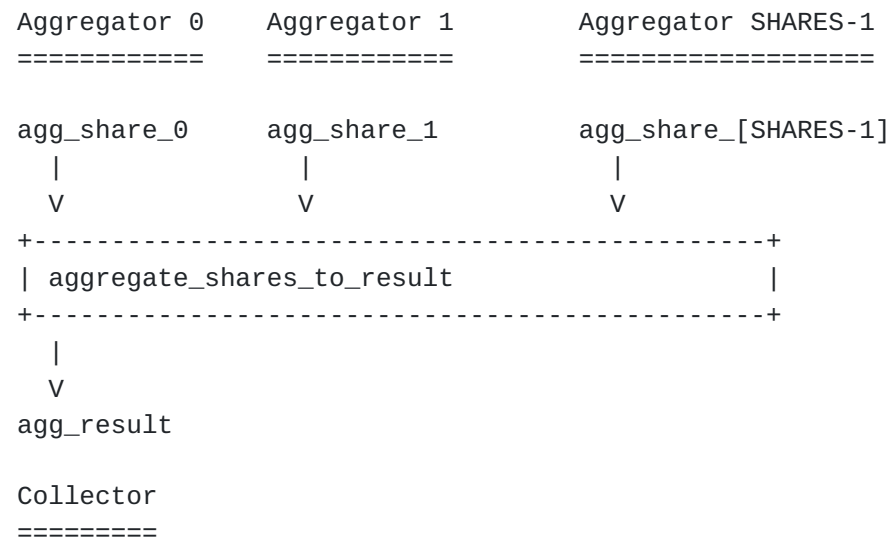

Figure 5: Computation of the final aggregate result from aggregate shares.

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 [I-D.draft-gpew-priv-ppm].) Or is this out-of-scope of this document?

4.6. Execution of a VDAF

Executing a VDAF involves the concurrent evaluation of the VDAF on individual inputs and aggregation of the recovered output shares. This is captured by the following example algorithm:

```
def run_vdaf(agg_param, nonces: Vec[Bytes], input_batch: Vec[Bytes]):
  # Distribute long-lived parameters.
  (public_param, verify_params) = vdaf_setup()

  output_shares = []
  for (nonce, input) in zip(nonces, input_batch):
    # Each client shards its input into shares.
    input_shares = measurement_to_input_shares(public_param, input)

    # Each aggregator initializes its preparation state.
    prep_states = []
    for j in range(SHARES):
      prep_states.append(validation_init(
          verify_params[j], agg_param, nonce, input_shares[j]))

    # Aggregators recover their output shares.
    inbound = []
    for i in range(ROUNDS+1):
      outbound = []
      for j in range(SHARES):
        outbound.append(prep_states[j].next(inbound))
      # This is where we would send messages over the network
      # in a distributed VDAF computation.
      inbound = outbound

    # The final outputs of validation are the output shares
    # for this input.
    output_shares.append(outbound)

  # Each aggregator aggregates its output shares into an
  # aggregate share.
  agg_shares = []
  for j in range(SHARES):
    my_output_shares = [out[j] for out in output_shares]
    my_agg_share = output_to_aggregate_shares(my_output_shares)
    agg_shares.append(my_agg_share)

  # Collector unshards the aggregate.
  return aggregate_shares_to_result(agg_shares)
```

Figure 6: Execution of a VDAF.

The inputs to this algorithm are the aggregation parameter agg_param, a list of nonces nonces, and a batch of Client inputs input_batch. The aggregation parameter is chosen by the aggregators prior to executing the VDAF. This document does not specify how the nonces are chosen, but some of our security considerations require that the nonces be unique for each VDAF evaluation. See Section 8 for details.

Another important question this document leaves out of scope is how a VDAF is to be executed by aggregators distributed over a real network. Algorithm run_vdaf prescribes the protocol's execution in a "benign" environment in which there is no adversary and 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" that implements suitable cryptographic functionalities. Moreover, some fraction of the Aggregators (or Clients) may be malicious and diverge from their prescribed behaviors. Section 8 describes the execution of the VDAF in various adversarial environments and what properties the wrapper protocol needs to provide in each.

5. Preliminaries

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

5.1. Key Derivation

A key-derivation scheme defines a method for deriving symmetric keys and a method for expanding a symmetric into an arbitrary length key stream are required. This scheme consists of the following algorithms:

* get_key(init_key, aux_input) -> key derives a fresh key key from an initial key init_key and auxiliary input aux_input. The length of init_key and key **MUST** be equal to KEY_SIZE.

* key_stream_init(key) -> state: KeyStream returns a key stream generator that is used to generate an arbitrary length stream of pseudorandom bytes. The length of key **MUST** be KEY_SIZE.

* KeyStream.next(len: Unsigned) -> (new_state: KeyStream, output) returns the next len bytes of the key stream and updates the key stream generator state. The length of the output **MUST** be len.

  TODO This functionality closely resembles what people usually think of as an extract-then-expand KDF, but differs somewhat in its syntax and, also, its required security properties. Can we get the same functionality from something that's more commonplace? HKDF doesn't fit the bill, unfortunately, because keys can only be expanded to a fairly short length. Our application requires a rather long key stream.

Associated types:

* KeyStream represents the state of the key stream generator.

Associated constants:

- *KEY_SIZE is the size of keys for the key-derivation scheme.

5.2. Finite Fields

In this document we only consider finite fields of the form GF(p) for prime p. Finite field elements are represented by a type Field that defines binary operators for addition and multiplication in the field. The type also defines the following associated functions:

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

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

  NOTE In reality this would be achieved by generating a random key and expanding it into a sequence of field elements using a key derivation scheme. This should probably be made explicit.

- *Field.encode_vec(data: Vec[Field]) -> encoded_data: Bytes represents the input data as a byte string encoded_data.

- *Field.decode_vec(encoded_data: Bytes) -> data: Vec[Field] reverse encoded_vec, returning the vector of field elements encoded by encoded_data. Raises an exception if the input does not encode a valid vector of field elements.

5.2.1. Deriving a Pseudorandom Vector

TODO Specify the following function in terms of a key-derivation scheme. It'll use key_stream_init to create a KeyStream object and read from it multiple times.

- *expand(Field, key: Bytes, len: Unsigned) -> output: Vec[Field] expands a key into a pseudorandom sequence of field elements.

5.2.2. Inner product of Vectors

TODO Specify the following:

- *inner_product(left: Vec[Field], right: Vec[Field]) -> Field computes the inner product of left and right.

6. prio3

NOTE This construction has not undergone significant security analysis.

NOTE An implementation of this VDAF can be found here.

This section describes a VDAF suitable for the following data aggregation task. Each Client measurement is encoded as a vector over a finite field, and the aggregate is computed by summing the vectors element-wise. Validity is defined by an arithmetic circuit C that takes as input a vector of field elements x: if C(x) == 0, then we say that x is valid; otherwise, we say that x is invalid. A number of useful measurement types can be defined this way:

- Simples statistics, like sum, average, and standard deviation;
- Estimation of quantiles, via a histogram; and
- Linear regression.

This VDAF does not have an aggregation parameter. Instead, the output share is derived from an input share by applying a fixed map. See Section 7 for an example of a VDAF that makes meaningful use of the aggregation parameter.

While the construction is derived from the original Prio system [CGB17], prio3 takes advantage of optimizations described later in [BBCGGI19] that improve communication complexity significantly. The etymology of the term prio3 is that it descends from 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 by [BBCGGI19]. However, was specialized for a particular measurement type. The goal of prio3 is to provide the same level of generality as the original system.

The way prio3 ensures privacy is quite simple: the Client shards its encoded input vector x into a number of additive secret shares, one for each Aggregator. Aggregators sum up their vector shares locally, and once enough shares have been aggregated, each sends its share of the result to the Collector, who recovers the aggregate result by adding up the vectors.

The main problem that needs to be solved is to verify that the additive shares generated by the Client add up to a valid input, i.e., that C(x)=0. The solution introduced by [BBCGGI19] is what they call a zero-knowledge proof system on distributed data. Viewing the Client as the prover and the Aggregators as the (distributed) verifier, the goal is to devise a protocol by which the Client

convinces the Aggregators that they hold secret shares of a valid input, without revealing the input itself.

The core tool for accomplishing this task is a refinement of Probabilistically Checkable Proof (PCP) systems called a Fully Linear Proof (FLP) system. We describe FLPs in detail below. Briefly, the Client generates a "proof" of its input's validity and distributes additive shares of the proof among the Aggregators. Each Aggregator then performs a computation on its input share and proof share locally and sends the result to the other Aggregators. Combining the results yields allows each Aggregator to decide if the input shares are valid.

prio3 can be viewed as a transformation of a particular class of FLP systems into a VDAF. The next section describes FLPs in detail, and the construction is given in Section 6.2.

6.1. Fully Linear Proof (FLP) Systems

Conceptually, an FLP system 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. (More on this in Section 6.2.) An FLP specifies the following core algorithms (encoding of inputs is described in Section 6.1.1:

- flp_prove(input: Vec[Field], prove_rand: Vec[Field], joint_rand: Vec[Field]) -> proof: 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. Type Field is a finite field as defined in Section 5.2.

- flp_query(input: Vec[Field], proof: Vec[Field], query_rand: Vec[Field], joint_rand: Vec[Field]) -> verifier: Vec[Field] is the query-generation algorithm run by the verifier. This is 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, but the latter is the same randomness used by the prover.

- flp_decide(verifier: Vec[Field]) -> decision: Boolean 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 whence it was generated is valid.

A concrete FLP defines the following constants:

- *JOINT_RAND_LEN: Unsigned is the length of the joint randomness in number of field elements.

- *PROVE_RAND_LEN: Unsigned is the length of the prover randomness.

- *QUERY_RAND_LEN: Unsigned is the length of the query randomness.

- *INPUT_LEN: Unsigned is the length of the input.

- *OUTPUT_LEN: Unsigned is the length of the aggregable output.

- *PROOF_LEN: Unsigned is the length of the proof.

- *VERIFIER_LEN: Unsigned is the length of the verifier message.

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 the query-generation algorithm can be run by each aggregator locally on its share of the input and proof, and the results can be combined to recover the verifier message. In the remainder, the result generated by an aggregator will be referred to as its "verifier share".

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

```
def run_flp(input: Vec[Field]):
  joint_rand = Field.rand_vec(JOINT_RAND_LEN)
  prove_rand = Field.rand_vec(PROVE_RAND_LEN)
  query_rand = Field.rand_vec(QUERY_RAND_LEN)

  # Prover generates the proof.
  proof = flp_prove(input, prove_rand, joint_rand)

  # Verifier queries the input and proof.
  verifier = flp_query(input, proof, query_rand, joint_rand)

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

Figure 7: Execution of an FLP.

The proof system is constructed so that, if input is a valid input, then run_flp(input) 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. In addition, the proof system is designed so that the verifier

message leaks nothing about the input (in an information theoretic sense). See Definition 3.9 from [BBCGGI19] for details.

An FLP is typically constructed from an arithmetic circuit that in turn defines validity. However, in the remainder we do not explicitly mention this circuit and allow validity to be defined by the set of inputs recognized by the FLP when run as described above.

Note that [BBCGGI19] defines a much larger class of fully linear proof systems; what is called a FLP here is called a 1.5-round, public-coin, interactive oracle proof system in their paper.

6.1.1. 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: Bytes) -> input: Vec[Field] encodes a raw measurement as a vector of field elements. The returned input **MUST** be of length INPUT_LEN. An error is raised if the measurement cannot be represented as a valid input.

In addition, for some FLPs, the encoded input includes redundant field elements that are useful for checking the proof, but which are not needed after the proof has been checked. Thus the FLP defines an algorithm for truncating the input to the length of the aggregated output:

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

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

6.2. Construction

This VDAF involves a single round of communication (ROUNDS == 1). It is defined for at least two Aggregators, but no more than 255. (2 <= SHARES <= 255).

6.2.1. Setup

The setup algorithm generates a symmetric key shared by all of the aggregators. The key is used to derive unique joint randomness for the FLP query-generation algorithm run by the aggregators during preparation.

```
def vdaf_setup():
  k_query_init = gen_rand(KEY_SIZE)
  verify_param = [ (j, k_query_init) for j in range(SHARES) ]
  return (None, verify_param)
```

Figure 8: The setup algorithm for prio3.

6.2.2. Sharding

Recall from Section 6.1 that the syntax for FLP systems 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. (Note that this idea comes from 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 prover randomness generated locally.

5. Shard the proof into a sequence of input shares.

The input and proof shares of one Aggregator (below we call it the "leader") are vectors of field elements. For shares of the other aggregators (below we call them the "helpers") as constant-sized symmetric keys. This is accomplished by mapping the key to a key stream and expanding the key stream into a pseudorandom vector of field elements. This involves a key-derivation scheme described in Section 5 and the helper function expand described in the same section.

This algorithm also makes use of a pair of helper functions for encoding the leader share and helper share. These are called encode_leader_share and encode_helper_share respectively and they are described in Section 6.2.6.

```
def measurement_to_input_shares(_, measurement):
  input = flp_encode(measurement)
  k_joint_rand = zeros(SEED_SIZE)

  # Generate input shares.
  leader_input_share = input
  k_helper_input_shares = []
  k_helper_blinds = []
  k_helper_hints = []
  for j in range(SHARES-1):
    k_blind = gen_rand(KEY_SIZE)
    k_share = gen_rand(KEY_SIZE)
    helper_input_share = expand(Field, k_share, INPUT_LEN)
    leader_input_share -= helper_input_share
    k_hint = get_key(k_blind,
        byte(j+1) + Field.encode_vec(helper_input_share))
    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(KEY_SIZE)
  k_leader_hint = get_key(k_leader_blind,
      byte(0) + Field.encode_vec(leader_input_share))
  k_joint_rand ^= k_leader_hint

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

  # Generate the proof shares.
  joint_rand = expand(Field, k_joint_rand, JOINT_RAND_LEN)
  prove_rand = expand(Field, gen_rand(KEY_SIZE), PROVE_RAND_LEN)
  proof = flp_prove(input, prove_rand, joint_rand)
  leader_proof_share = proof
  k_helper_proof_shares = []
  for j in range(SHARES-1):
    k_share = gen_rand(KEY_SIZE)
    k_helper_proof_shares.append(k_share)
    helper_proof_share = expand(Field, k_share, PROOF_LEN)
    leader_proof_share -= helper_proof_share

  input_shares = []
  input_shares.append(encode_leader_share(
    leader_input_share,
    leader_proof_share,
    k_leader_blind,
    k_leader_hint,
  ))
```

```
  for j in range(SHARES-1):
    input_shares.append(encode_helper_share(
      k_helper_input_share[j],
      k_helper_proof_share[j],
      k_helper_blinds[j],
      k_helper_hints[j],
    ))
  return input_shares
```

Figure 9: Input-distribution algorithm for prio3.

6.2.3. Preparation

This section describes the process of recovering output shares from the input shares. The high-level idea is that each of the Aggregators runs the FLP query-generation algorithm on its share of the input and proof and exchange shares of the verifier message. Once they've done that, each runs the FLP decision algorithm on the verifier message locally 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 symmetric key. Each Aggregator derives an XOR secret share of this key from its input share and the "blind" generated by the client. Before it can run the query-generation algorithm, it must first gather the XOR secret shares derived by the other Aggregators.

So that the Aggregators can avoid an extra round of communication, the client sends each Aggregator a "hint" equal to the XOR of the other Aggregators' shares of the joint randomness key. However, this leaves open the possibility that the client cheated by, say, forcing the Aggregators to use joint randomness the 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 key before accepting their output shares.

> NOTE This optimization somewhat diverges from Section 6.2.3 of [BBCGGI19]. We'll need to understand better how this impacts security.

The preparation state is defined as follows. It involves two additional helper functions, encode_verifer_share and decode_verifier_share, both of which are defined in Section 6.2.6.

```
class PrepState:
  def __init__(verify_param, _, nonce, r_input_share):
    (j, k_query_init) = verify_param

    if j == 0: # leader
      (self.input_share, self.proof_share,
       k_blind, k_hint) = decode_leader_share(r_input_share)
    else:
      (k_input_share, k_proof_share,
       k_blind, k_hint) = decode_helper_share(r_input_share)
      self.input_share = expand(Field, k_input_share, INPUT_LEN)
      self.proof_share = expand(Field, k_proof_share, PROOF_LEN)

    self.k_joint_rand_share = get_key(
      k_blind, byte(j) + self.input_share)
    self.k_joint_rand = k_hint ^ self.k_joint_rand_share
    self.k_query_rand = get_key(k_query_init, byte(255) + nonce)
    self.step = "ready"

  def next(self, inbound: Vec[Bytes]):
    if self.step == "ready" and len(inbound) == 0:
      joint_rand = expand(Field, self.k_joint_rand, JOINT_RAND_LEN)
      query_rand = expand(Field, self.k_query_rand, QUERY_RAND_LEN)
      verifier_share = flp_query(
        self.input_share, self.proof_share, query_rand, joint_rand)

      self.output_share = flp_truncate(input_share)
      self.step = "waiting"
      return encode_verifier_share(
        self.k_joint_rand_share,
        self.verifier_share,
      )

    elif self.step == "waiting" and len(inbound) == SHARES:
      k_joint_rand = zeros(KEY_SIZE)
      verifier = vec_zeros(VERIFIER_LEN)
      for r_share in inbound:
        (k_joint_rand_share,
         verifier_share) = decode_verifier_share(r_share)

        k_joint_rand ^= k_joint_rand_share
        verifer += verifier_share

      if k_joint_rand != self.k_joint_rand: raise ERR_INVALID
      if not flp_decide(verifier): raise ERR_INVALID
      return Field.encode_vec(self.output_share)

    else: raise ERR_INVALID_STATE
```

Figure 10: Preparation state for prio3.

NOTE JOINT_RAND_LEN may be 0, in which case the joint randomness computation is not necessary. Should we bake this option into the spec?

6.2.4. Aggregation

```
def output_to_aggregate_shares(_, output_shares: Vec[Bytes]):
  agg_share = vec_zeros(OUTPUT_LEN)
  for output_share in output_shares:
    agg_share += Field.decode_vec(output_share)
  return Field.encode_vec(agg_share)
```

Figure 11: Aggregation algorithm for prio3.

6.2.5. Unsharding

```
def aggregate_shares_to_result(_, agg_shares: Vec[Bytes]):
  agg = vec_zeros(OUTPUT_LEN)
  for agg_share in agg_shares:
    agg += Field.decode_vec(agg_share)
  return Field.encode_vec(agg)
```

Figure 12: Computation of the aggregate result for prio3.

6.2.6. Helper Functions

TODO Specify the following functionalities.

*encode_leader_share(input_share: Vec[Field], proof_share: Vec[Field], blind, hint) -> encoded: Bytes encodes a leader share as a byte string.

*encode_helper_share(input_share, proof_sahre, blind, hint) -> encoded: Bytes encodes a helper share as a byte string.

*decode_leader_share and decode_helper_share decode a leader and helper share respectively.

*encode_verifier_share(hint, verifier_share: Vec[Field]) -> encoded encodes a hint and verifier share as a byte string.

*decode_verifier_share decodes a verifier share.

7. hits

NOTE An implementation of this VDAF can be found here.

This section specifies hits, 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 the privacy-preserving t-heavy-hitters protocol of [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.

hits 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.

The name hits is an anagram of "hist", which is short for "histogram". It is a nod toward the "subset histogram" problem formulated by [BBCGGI21] and for which the hits is a solution.

7.1. Incremental Distributed Point Functions (IDPFs)

NOTE An implementation of IDPFs can be found here.

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 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 6.

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 IPDFs specified here is stateless, in the sense that there is no state carried between IPDF 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.

7.2. Construction

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

7.2.1. 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(KEY_SIZE)
  return (None, [(0, k_verify_init), (1, k_verify_init)])
```

Figure 13: The setup algorithm for hits.

7.2.1.1. 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 7.2.5.

```
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(input, beta)

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

  return input_shares
```

Figure 14: The input-distribution algorithm for hits.

TODO It would be more efficient to represent the correlation shares using PRG seeds as suggested in [BBCGGI21].

7.2.2. 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 prefixe, 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.

```
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: Vec[Bytes]):
    l = self.l
    (a_share, b_share, c_share,
     A_share, B_share) = correlation_shares[l-1]

    if self.step == "ready" and len(inbound) == 0:
      # Evaluation IPPF on candidate prefixes.
      data_share, auth_share = [], []
      for x in self.candiate_prefixes:
        value = kdpf_key.eval(l, x)
        data_share.append(value[0])
        auth_share.append(value[1])

      # Prepare first sketch verification message.
      r = expand(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 len(inbound) == 2:
      verifier_1 = Field[l].deocde_vec(inbound[0]) + \
                   Field[l].deocde_vec(inbound[1])

      verifier_share_2 = [
        (verifier_1[0] * verifier_1[0] \
         - verifier_1[1] \
         - verifier_1[2]) * self.party_id \
        + A_share * verifer_1[0] \
        + B_share
      ]

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

    elif self.step == "sketch round 2" and len(inbound) == 2:
```

```
      verifier_2 = Field[l].decode_vec(inbound[0]) + \
                   Field[l].decode_vec(inbound[1])

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

    else: raise ERR_INVALID_STATE
```

Figure 15: Preparation state for hits.

7.2.3. Aggregation

```
def output_to_aggregate_shares(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].vec_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 16: Aggregation algorithm for hits.

7.2.4. Unsharding

```
def aggregate_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 17: Computation of the aggregate result for hits.

7.2.5. 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 IDFP 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.

8. 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.

Multi-party protocols for privacy-preserving measurement have two essential security gaols:

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 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.) A VDAF is the core cryptographic primitive of a protocol that achieves these 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.

9. IANA Considerations

This document makes no request of IANA.

10. References

10.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/rfc2119>.

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

10.2. 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>.

[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>.

[I-D.draft-gpew-priv-ppm] Geoghegan, T., Patton, C., Rescorla, E., and C. A. Wood, "Privacy Preserving Measurement", Work in Progress, Internet-Draft, draft-gpew-priv-ppm-00, 25 October 2021, <https://datatracker.ietf.org/doc/html/draft-gpew-priv-ppm-00>.

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

[Vad16] Vadhan, S., "The Complexity of Differential Privacy", 2016, <https://link.springer.com/chapter/10.1007/978-3-319-57048-8_7>.

Acknowledgments

Thanks to Henry Corrigan-Gibbs, Mariana Raykova, and Christopher Wood for useful feedback on the syntax of VDAF schemes.

Authors' Addresses

Christopher Patton
Cloudflare, Inc.

Email: cpatton@cloudflare.com

Richard L. Barnes
Cisco

Email: rlb@ipv.sx

Phillipp Schoppmann
Google

Email: schoppmann@google.com