CFRG
Internet-Draft
Intended status: Informational
Expires: 26 March 2026

R. L. Barnes
Cisco
D. Cook
ISRG
C. Patton
Cloudflare
P. Schoppmann
Google

22 September 2025

Verifiable Distributed Aggregation Functions
draft-irtf-cfrg-vdaf-16

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 invalid measurement. Two concrete VDAFs are specified, one for general-purpose aggregation (Prio3) and another for heavy hitters (Poplar1).

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/cfrg/draft-irtf-cfrg-vdaf.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

(RFC EDITOR: remove this paragraph.) The source for this draft and the reference implementation can be found at https://github.com/cfrg/draft-irtf-cfrg-vdaf.

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

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

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

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

Even when noise is added to the measurements, collecting them in the clear still reveals a significant amount of information to the collector. On the one hand, depending on the "amount" of noise a client adds to its measurement, it may be possible for a curious collector to make a reasonable guess of the measurement's true value. On the other hand, the more noise the clients add, the less reliable will be the server's estimate of the aggregate. Thus systems relying solely on a DP mechanism must strike a delicate balance between privacy and utility.

Another way of constructing a privacy-preserving measurement system is to use multi-party computation (MPC). The goal of such a system is that no participant in the protocol should learn anything about an individual measurement beyond what it can deduce from the aggregate. MPC achieves this goal by distributing the computation of the aggregate across multiple aggregation servers, one of which is presumed to be honest, i.e., not under control of the attacker. Moreover, MPC can be composed with various DP mechanisms to ensure the aggregate itself does not leak too much information about any one of the measurements [MPRV09].

This document describes two classes of MPC protocols, each aiming for a different set of goals.

In a Distributed Aggregation Function (DAF, Section 4), each client splits its measurement into multiple secret shares, one for each aggregation server. DAFs require two properties of the secret sharing scheme. First, one can reconstruct the underlying measurement by simply adding up all of the shares. (Typically the shares are vectors over some finite field.) Second, given all but

one of the shares, it is impossible to learn anything about the underlying measurement. These properties give rise to a simple strategy for privately aggregating the measurements: each aggregation server adds up its measurement shares locally before revealing their sum to the data collector; then all the data collector has to do is add up these sums to get the aggregate result.

This strategy is compatible with any aggregation function that can be represented as the sum of some encoding of the measurements. Examples include: summary statistics such as sum, mean, and standard deviation; estimation of quantiles, e.g., median; histograms; linear regression; or counting data structures, e.g., Bloom filters. However, not all functions fit into this rubric, as it is constrained to linear computations over the encoded measurements.

In fact, this document's framework admits DAFs with slightly more functionality, computing aggregation functions of the form

```
F(agg_param, meas_1, ..., meas_M) =
    G(agg_param, meas_1) + ... + G(agg_param, meas_M)
```

where meas_1, ..., meas_M are the measurements, G is a possibly non-linear function, and agg_param is a parameter of that function chosen by the data collector. This paradigm, known as function secret sharing [BGI15], allows for more sophisticated data analysis tasks, such as grouping metrics by private client attributes [MPDST25] or computing heavy hitters [BBCGGI21]. (More on the latter task below.)

The second class of protocols defined in this document are called Verifiable Distributed Aggregation Functions (VDAFs, Section 5). In addition to being private, VDAFs are verifiable in the following sense. By design, a secret sharing of a valid measurement, e.g., a number between 1 and 10, is indistinguishable from a secret sharing of an invalid measurement, e.g., a number larger than 10. This means that DAFs are vulnerable to attacks from malicious clients attempting to disrupt the computation by submitting invalid measurements. VDAFs are designed to allow the servers to interact with one another in order to detect and remove these measurements prior to aggregation. This document refers to this property as verifiability. (This is also called robustness in prior work [CGB17], [DPRS23].)

Achieving verifiability using the cryptographic techniques described in this document requires a significant amount of interaction between the servers. DAFs on the other hand are non-interactive, making them easier to deploy; but they do not on their own allow for verifying the validity of the measurements. This may be tolerable in some applications. For instance, if the client's software is executed in a trusted execution environment, it may be reasonable to assume that no client is malicious.

The DAF and VDAF abstractions encompass a variety of MPC techniques in the literature. These protocols vary in their operational and security requirements, sometimes in subtle but consequential ways. This document therefore has two important goals:

1. Provide higher-level protocols, like [DAP], with a simple, uniform interface for accessing privacy-preserving measurement schemes, document relevant operational and security requirements, and specify constraints for safe usage:

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

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

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

2. Provide cryptographers with design criteria that provide a clear deployment roadmap for new constructions of privacy-preserving measurement systems.

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

* The Prio system [CGB17] allows for the privacy-preserving computation of a variety of aggregate statistics, combining additive secret sharing as described above with a mechanism for checking the validity of each measurement. Section 7 specifies 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.

* The Poplar protocol [BBCGGI21] solves a problem known as private heavy-hitters. In this problem, each client holds a bit-string, and the goal of the aggregation servers is to compute the set of strings that occur at least T times for some threshold T. The

core primitive in their protocol is a secret sharing of a point function [GI14] (denoted G above) that allows the servers to privately count how many of the clients' strings begin with a given prefix (agg_param in the notation above). Section 8 specifies a VDAF called Poplar1 that implements this prefix counting functionality and describe how it is used in the heavy hitters protocol.

The remainder of this document is organized as follows: Section 2 lists definitions and conventions used in the remainder of the document; Section 3 gives a brief overview of DAFs and VDAFs, the parties involved in the computation, and the requirements for non-collusion; Section 4 defines the syntax for DAFs; Section 5 defines the syntax for VDAFs; Section 6 defines various functionalities that are common to the constructions defined in this document; Section 7 specifies Prio3; Section 8 specifies Poplar1; and Section 9 enumerates security considerations for DAFs and VDAFs in general and the Prio3 and Poplar1 constructions in particular.

1.1. Change Log

(RFC EDITOR: remove this section.)

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

16:

* Align document with guidelines for ASCII-safe mathematical notation from Section 3.3.1.6 of [I-D.draft-irtf-cfrg-cryptography-specification-02].

* Address feedback from Crypto Review Panel [PANEL-FEEDBACK].

15:

* Simplify the ping-pong API for 2-party preparation by merging the outbound message into the state object. This reduces the number of cases the caller has to handle.

* Update the test vector format. First, enrich the test vector schema to express negative test cases. Second, change the encoding of output shares to match the aggregate shares.

14:

* Poplar1: When decoding an aggregation parameter, require the padding bits after each prefix to be cleared.

13:

* (V)DAF: Replace the one-shot aggregation API with a streaming API. Each Aggregator initializes aggregation, then updates its aggregate share as output shares are produced. The scheme also specifies a method for merging multiple aggregate shares.

* Poplar1: Move prefix uniqueness and ordering checks from prep_init() to is_valid().

* Poplar1: Use bool to represent control bits instead of Field2.

* Prio3MultihotCountVec: Change the measurement type from list[int] to list[bool].

* Security considerations: Define our threat model for side channel attacks and enumerate the parts of the spec that are most relevant to implementers.

* Improve the specification of each Prio3 variant by listing each implementation of Valid and Gadget in full. Gadgets are listed in a new appendix section.

* Improve the specification of the FLP system by listing the proof-generation, query, and decision algorithms in full. The wrapper gadgets are listed in the new section of the appendix for gadgets.

* Add a section with a high-level overview of the IDPF construction.

* Move some sections around: move ping-pong and star topologies under communication patterns for VDAF preparation; move FLP proof generation, query, and decision algorithms up one level; move privacy considerations for aggregation parameters up one level; and move safe usage of IDPF outputs up one level.

12:

* (V)DAF: Add an application context string parameter to sharding and preparation. The motivation for this change is to harden Prio3 against offline attacks. More generally, however, it allows designing schemes for which correct execution requires agreement on the application context. Accordingly, both Prio3 and Poplar1 have been modified to include the context in the domain separation tag of each XOF invocation. (*)

* Prio3: Improve soundness of the base proof system and the circuits of some variants. Generally speaking, wherever we evaluate a univariate polynomial at a random point, we can instead evaluate a multivariate polynomial of lower degree. (*)

* Prio3: Replace the helper's measurement and proof share seeds with a single seed. (*)

* Prio3Sum: Update the circuit to support a more general range check and avoid using joint randomness. (*)

* Prio3Histogram, Prio3MultihotCountVec: Move the final reduction of the intermediate outputs out of the circuit. (*)

* IDPF: Add the application context string to key generation and evaluation and bind it to the fixed AES key. (*)

* IDPF: Use XofTurboShake128 for deriving the leaf nodes in order to ensure the construction is extractable. (*)

* IDPF: Simplify the public share encoding. (*)

* XofTurboShake128: Change SEED_SIZE from 16 bytes to 32 to mitigate offline attacks on Prio3 robustness. In addition, allow seeds of different lengths so that we can continue to use XofTurboShake128 with IDPF. (*)

* XofTurboShake128, XofFixedKeyAes128: Increase the length prefix for the domain separation tag from one by to two bytes. This is to accommodate the application context. (*)

* Reassign codepoints for all Prio3 variants and Poplar1. (*)

* Security considerations: Add a section on defense-in-depth measures taken by Prio3 and Poplar1 and more discussion about choosing FLP parameters.

11:

* Define message formats for the Poplar1 aggregation parameter and IDPF public share.

* IDPF: Require the IDPF binder must be a random nonce.

* VDAF: Replace the pseudocode description of the ping-ping topology with Python and sketch the star topology.

* DAF: Align aggregation parameter validation with VDAF.

* Replace Union[A, B] type with A | B.

* Rename FFT ("Fast Fourier Transform") with NTT ("Number Theoretic Transform").

10:

* Define Prio3MultihotCountVec, a variant of Prio3 for aggregating bit vectors with bounded weight.

* FLP: Allow the output of the circuit to be a vector. This makes it possible to skip joint randomness derivation in more cases.

* Poplar1: On the first round of preparation, handle None as an error. Previously this message was interpreted as a length-3 vector of zeros.

* Prio3: Move specification of the field from the FLP validity circuit to the VDAF itself.

* Clarify the extent to which the attacker controls the network in our threat models for privacy and robustness.

* Clean up various aspects of the code, including: Follow existing object-oriented programming patterns for Python more closely; make the type hints enforceable; and avoid shadowing variables.

* Poplar1: Align terminology with [BBCGGI23].

* IDPF: Add guidance for encoding byte strings as indices.

09:

* Poplar1: Make prefix tree traversal stricter by requiring each node to be a child of a node that was already visited. This change is intended to make it harder for a malicious Aggregator to steer traversal towards non-heavy-hitting measurements.

* Prio3: Add more explicit guidance for choosing the field size.

* IDPF: Define extractability and clarify (un)safe usage of intermediate prefix counts. Accordingly, add text ensuring public share consistency to security considerations.

08:

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

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

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

* Replace SHAKE128 with TurboSHAKE128. (*)

07:

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

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

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

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

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

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

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

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

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

06:

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

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

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

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

05:

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

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

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

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

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

04:

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

* Vdaf: Pass the nonce to the sharding algorithm.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

03:

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

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

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

* Add additional guidance for choosing FFT-friendly fields.

02:

* Complete the initial specification of Poplar1.

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

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

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

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

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

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

* Define new field Field255.

01:

* Require that VDAFs specify serialization of aggregate shares.

* Define Distributed Aggregation Functions (DAFs).

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

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

2. Conventions

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 (compatible with Python 3.12 or later). A fatal error in a program (e.g., failure to parse one of the function parameters) is usually handled by raising an exception.

In Python, array indexing starts with 0, e.g., x[0] is the first element and x[len(x)-1] is the last of x. It is also possible to index from the end of the list, e.g., x[-1] is the last element of x.

Python uses the symbols +, -, *, and / as binary operators. When the operands are integers, these have the usual meaning, except:

* Division results in a floating point number. Python includes a similar operator, x // y, which is short for floor(x / y).

* When x and y are byte strings, x + y denotes their concatenation, i.e., concat(x, y) as defined below.

Finite field arithmetic overloads these operators; see Section 6.1.

Exponentiation is denoted by x ** y in Python.

Bitwise exclusive or is denoted by x ^ y in Python.

Type hints are used to define input and output types:

* The type variable F is used in signatures to signify any type that is a subclass of Field (Section 6.1).

* bytes is a byte string.

* int is an integer.

* Generic is used in class definitions to explicitly declare type variables of generic classes.

* Any is the universal supertype, which admits values of any type.

* Optional[T] is shorthand for T | None. Its value may be None or have type T.

* Self represents the containing class of the method definition in which it appears.

* Sequence[T] is either a list or tuple of values of type T.

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

A global constant VERSION of type int is defined, which algorithms are free to use as desired. Its value SHALL be 12.

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 transmitted from one party to another, this document prescribes a particular encoding of that quantity as a byte string.

Some common functionalities:

* additive_secret_share(x: list[F], num_shares: int, field: type[F]) -> list[list[F]] takes a vector x of field elements and returns num_shares vectors of length len(x) such that they all add up to the input vector. Note that this function is not used normatively in this document.

* byte(x: int) -> bytes returns the representation of the integer x in the range [0, 256) as a single-byte byte string.

* cast(typ: type[T], x: object) -> T returns the input value unchanged. This is only present to assist with static analysis of the Python code. Type checkers will ignore the inferred type of the input value, and assume the output value has the given type.

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

* from_be_bytes(encoded: bytes) -> int decodes a big-endian byte string, i.e., returns the integer x for which to_be_bytes(x, len(encoded)) == encoded.

* from_le_bytes(encoded: bytes) -> int decodes a little-endian byte string, i.e., returns the integer x for which to_le_bytes(x, len(encoded)) == encoded.

* front(len: int, x: list[Any]) -> tuple[list[Any], list[Any]] splits x into two vectors, where the first vector is made up of the first len elements of x and the second is made up of the remaining elements. This function is equivalent to (x[:len], x[len:]).

* gen_rand(len: int) -> bytes returns a byte array of the requested length (len) generated by a cryptographically secure pseudorandom number generator (CSPRNG).

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

* range(stop: int) or range(start: int, stop: int[, step: int]) is the range function from the Python standard library. The one-argument form returns the integers from zero (inclusive) to stop (exclusive). The two- and three-argument forms allow overriding the start of the range and overriding the step between successive output values.

* to_be_bytes(x: int, len: int) -> bytes converts an integer x whose value is in the range [0, 2**(8*len)) to a big-endian byte string of length len.

* to_le_bytes(x: int, len: int) -> bytes converts an integer x whose value is in the range [0, 2**(8*len)) to a little-endian byte string of length len.

* poly_eval(field: type[F], p: list[F], x: F) -> F returns the result of evaluating the polynomial, p(x). The coefficients of polynomials are stored in lists in ascending order of degree, starting with the constant coefficient. The field parameter is the class object for F and is used by the implementation to construct field elements. (See Section 6.1.)

* poly_interp(field: type[F], inputs: list[F], outputs: list[F]) -> list[F] returns the coefficients of the lowest degree polynomial p for which p(inputs[k]) == outputs[k] for all k. Normally this will be computed using the Number Theoretic Transform (NTT) [SML24].

* poly_mul(field: type[F], p: list[F], q: list[F]) -> list[F] returns the product of two polynomials.

* poly_strip(field: type[F], p: list[F]) -> list[F] strips the zeros from the end of the input polynomial's list of coefficients. That is, it returns p[:i] where i is the index of the highest-degree non-zero coefficient of p.

* 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.

* zeros(len: int) -> bytes returns an array of bytes of the requested length (len). Each element of the array is set to zero.

2.1. Mathematical Notation

The following symbols are used in mathematical notation.

ASCII glyph(s)	Description	Comment
+, -	Addition / subtraction	Needs to be constant time. Used for addition and subtraction of field elements as well as numbers.
*	Addition / subtraction	Needs to be constant time. Used for multiplication of field elements as well as numbers.
**	Exponentiation	Needs to be constant time. Used for exponentiation of field elements as well as numbers.
/	Division	Needs to be constant time. Used for division of field elements (i.e. multiplication by inverse) as well as numbers.
||	Concatenation	Used for concatenation of byte strings and bit strings.

Table 1: Mathematical Operators and Symbols

3. Overview

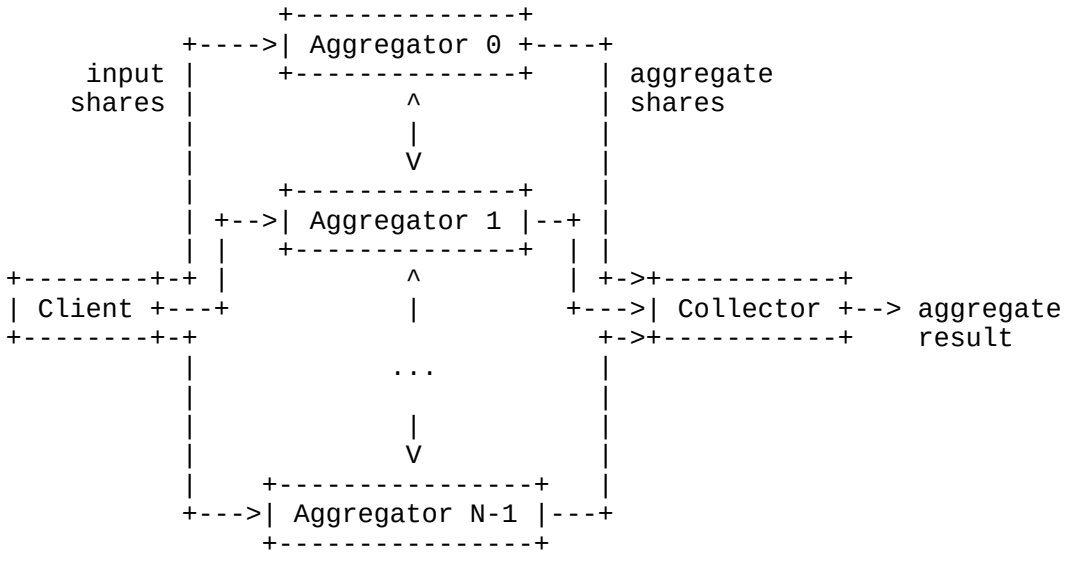

Figure 1: Overall data flow of a (V)DAF.

There are three types of actors in a DAF- or VDAF-based private measurement system: Clients, Aggregators, and the Collector. The overall flow of the measurement process is illustrated in Figure 1. The steps are as follows:

1. To submit an individual measurement, a Client shards its measurement into "input shares" and sends one input share to each Aggregator. This document sometimes refers to this sequence of input shares collectively as the Client's "report". (The report contains a few more items needed to process the measurement; these are described in Section 4.)

2. Once an Aggregator receives an input share from each Client, it processes the input shares into a value called an "aggregate share" and sends it the Collector. The aggregate share is a secret share of the aggregate representation of the measurements.

3. Once the Collector has received an aggregate share from each Aggregator, it combines them into the aggregate representation of the measurements, called the "aggregate result".

This second step involves a process called "preparation" in which the Aggregator refines each input share into an intermediate representation called an "output share". The output shares are then combined into the aggregate share as shown in Figure 2.

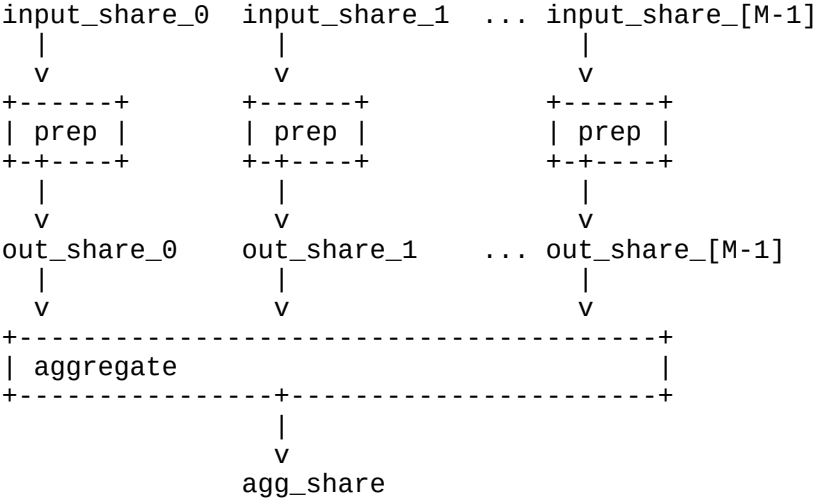

Figure 2: Preparation of input shares into output shares and aggregation of output shares into an aggregate share. Executed by each Aggregator. M denotes the number of measurements being aggregated.

In the case of VDAFs (Section 5), preparation involves interacting with the other Aggregators. This process will fail if the underlying measurement is invalid, in which case the report is rejected and not included in the aggregate result.

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 validity of the aggregate results 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; see Section 9.8), 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.

This document describes 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, it is assumed 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 disseminate 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 by the Collector on the batch of measurements. For example, the aggregation parameter is used to represent the prefixes in the prefix-counting functionality of of Poplar1 discussed in Section 1.

Execution of a DAF has four distinct stages:

* Sharding: Each Client generates input shares from its measurement and distributes them among the Aggregators. In addition to the input shares, the client generates a "public share" during this step that is disseminated to all of 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 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 the algorithm for the computation needed in each of these stages. The interface, denoted Daf, is defined in the remainder of this section. In addition, a concrete DAF defines the associated constants and types enumerated in the following table.

Parameter	Description
ID: int	Algorithm identifier for this DAF, in the range [0, 2**32).
SHARES: int	Number of input shares into which each measurement is sharded.
NONCE_SIZE: int	Size of the nonce associated with each report.
RAND_SIZE: int	Size of each random byte string consumed by the sharding algorithm.
Measurement	Type of each measurement.
PublicShare	Type of each public share.
InputShare	Type of each input share.
AggParam	Type of the aggregation parameter.
OutShare	Type of each output share.
AggShare	Type of each aggregate share.
AggResult	Type of the aggregate result.

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

The types in this table define the inputs and outputs of DAF methods at various stages of the computation. Some of these values need to be written to the network in order to carry out the computation. In particular, it is RECOMMENDED that concrete instantiations of the Daf interface specify a standard encoding for the PublicShare, InputShare, AggParam, and AggShare types.

4.1. Sharding

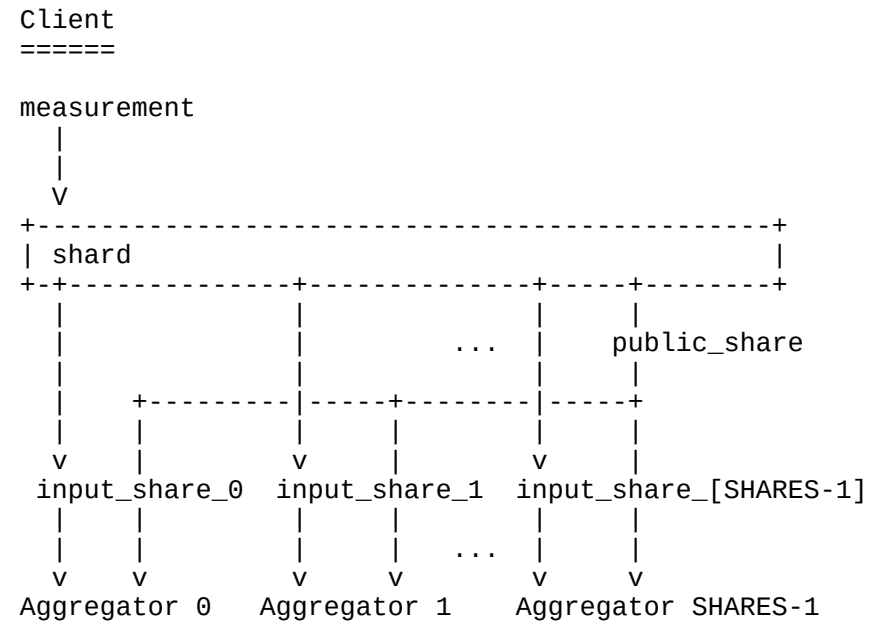

Figure 3: Illustration of the sharding algorithm.

The sharding algorithm run by each Client is defined as follows:

* daf.shard(ctx: bytes, measurement: Measurement, nonce: bytes, rand: bytes) -> tuple[PublicShare, list[InputShare]] consumes the "application context" (defined below), a measurement, and a nonce and produces the public share, distributed to each of the Aggregators, and the input shares, one for each Aggregator.

  Pre-conditions:

  - nonce MUST have length equal to daf.NONCE_SIZE and MUST be generated using a cryptographically secure random number generator (CSPRNG).

  - rand consists of the random bytes consumed by the algorithm. It MUST have length equal to daf.RAND_SIZE and MUST be generated using a CSPRNG.

  Post-conditions:

  - The number of input shares MUST equal daf.SHARES.

Sharding is bound to a specific "application context". The application context is a string intended to uniquely identify an instance of the higher level protocol that uses the DAF. The goal of binding the application to DAF execution is to ensure that aggregation succeeds only if the Clients and Aggregators agree on the application context. (Preparation binds the application context, too; see Section 4.2.) Note that, unlike VDAFs (Section 5), there is no explicit signal of disagreement; it may only manifest as a garbled aggregate result.

The nonce is a public random value associated with the report. It is referred to as a nonce because normally it will also be used as a unique identifier for that report in the context of some application. The randomness requirement is especially important for VDAFs, where it may be used by the Aggregators to derive per-report randomness for verification of the computation. See Section 9.2 for details.

4.2. Preparation

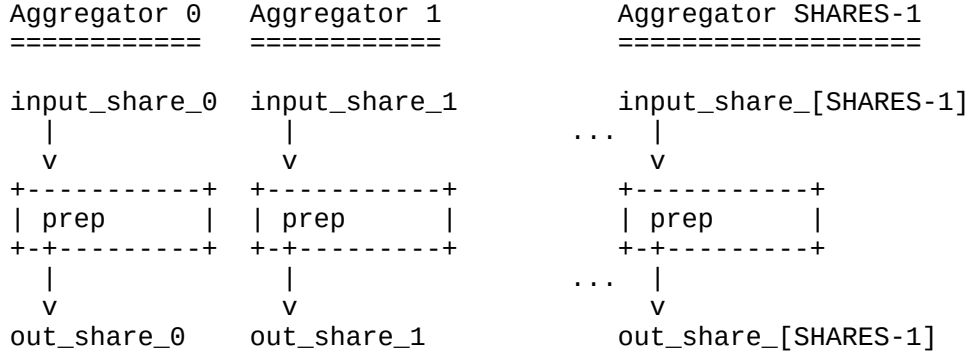

Figure 4: Illustration of preparation.

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

* daf.prep(ctx: bytes, agg_id: int, agg_param: AggParam, nonce: bytes, public_share: PublicShare, input_share: InputShare) -> OutShare consumes the public share and one of the input shares generated by the Client, the application context, the Aggregator's unique identifier, the aggregation parameter selected by the Collector, and the report nonce and returns an output share.

  Pre-conditions:

- agg_id MUST be in the range [0, daf.SHARES) and match the index of input_share in the sequence of input shares produced by the Client.

- nonce MUST have length daf.NONCE_SIZE.

The Aggregators MUST agree on the value of the aggregation parameter. Otherwise, the aggregate result may be computed incorrectly by the Collector.

4.3. Validity of Aggregation Parameters

In general, it is permissible to aggregate a batch of reports multiple times. However, to prevent privacy violations, DAFs may impose certain restrictions on the aggregation parameters selected by the Collector. Restrictions are expressed by the aggregation parameter validity function:

* daf.is_valid(agg_param: AggParam, previous_agg_params: list[AggParam]) -> bool returns True if agg_param is allowed given the sequence previous_agg_params of previously accepted aggregation parameters.

Prior to accepting an aggregation parameter from the Collector and beginning preparation, each Aggregator MUST validate it using this function.

4.4. Aggregation

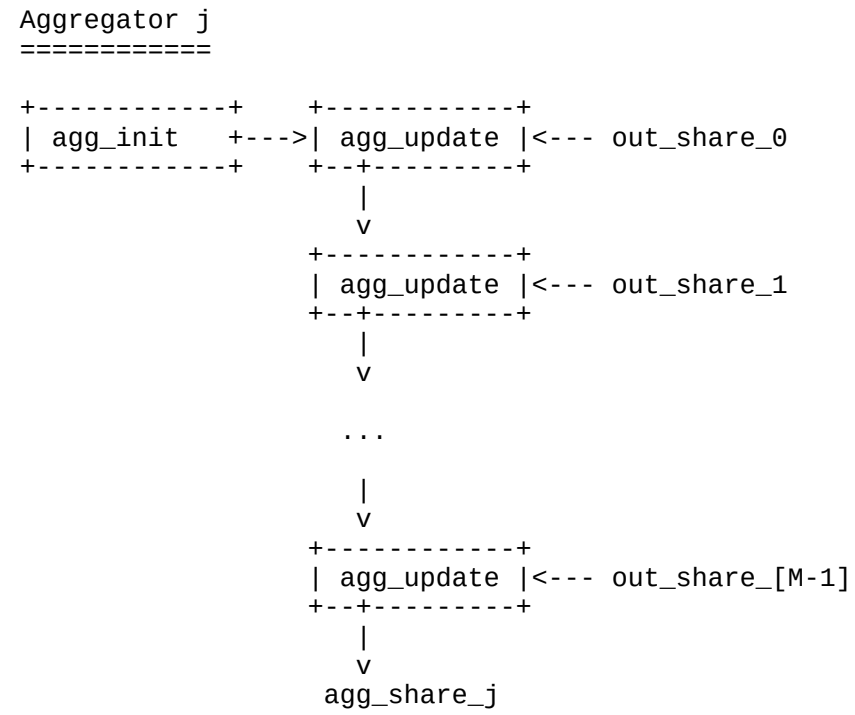

Figure 5: Illustration of aggregation. The number of measurements in the batch is denoted by M.

Once an Aggregator holds an output share, it adds it into its aggregate share for the batch. This streaming aggregation process is implemented by the following pair of algorithms:

* daf.agg_init(agg_param: AggParam) -> AggShare returns an empty aggregate share. It is called to initialize aggregation of a batch of measurements.

* daf.agg_update(agg_param: AggParam, agg_share: AggShare, out_share: OutShare) -> AggShare accumulates an output share into an aggregate share and returns the updated aggregate share.

In many situations it is desirable to split an aggregate share across multiple storage elements, then merge the aggregate shares together just before releasing the completed aggregate share to the Collector. DAFs facilitate this with the following method:

* daf.merge(agg_param: AggParam, agg_shares: list[AggShare]) -> AggShare merges a sequence of aggregate shares into a single aggregate share.

4.4.1. Aggregation Order

For most DAFs and VDAFs, the outcome of aggregation is not sensitive to the order in which output shares are aggregated. This means that aggregate shares can be updated or merged with other aggregate shares in any order. For instance, for both Prio3 (Section 7) and Poplar1 (Section 8), the aggregate shares and output shares both have the same type, a vector over some finite field (Section 6.1); and aggregation involves simply adding vectors together.

In theory, however, there may be a DAF or VDAF for which correct execution requires each Aggregator to aggregate output shares in the same order.

4.5. Unsharding

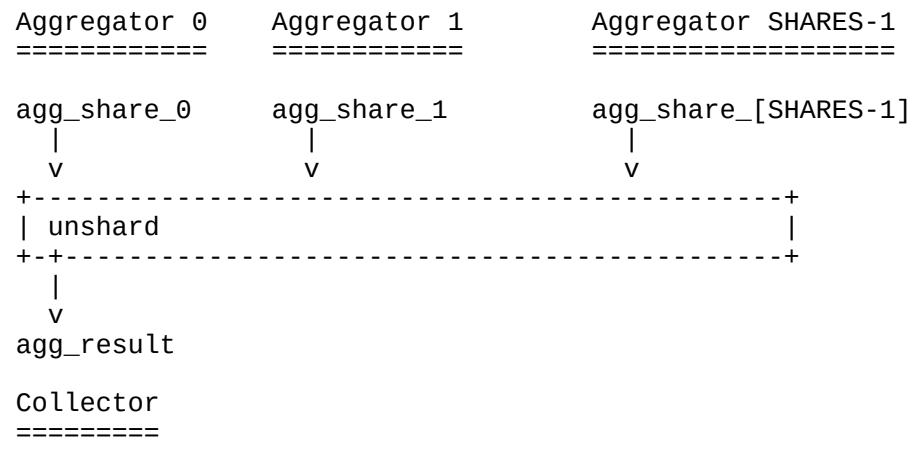

Figure 6: Illustration of unsharding.

After the Aggregators have aggregated all measurements in the batch, each sends its aggregate share to the Collector, who runs the unsharding algorithm to recover the aggregate result:

* daf.unshard(agg_param: AggParam, agg_shares: list[AggShare], num_measurements: int) -> AggResult consumes the aggregate shares and produces the aggregate result.

  Pre-conditions:

  - The length of agg_shares MUST be SHARES.

  - num_measurements MUST equal the number of measurements in the batch.

4.6. Execution of a DAF

Secure execution of a DAF involves simulating the following procedure over an insecure network.

```
def run_daf(
        daf: Daf[
            Measurement,
            AggParam,
            PublicShare,
            InputShare,
            OutShare,
            AggShare,
            AggResult,
        ],
        ctx: bytes,
        agg_param: AggParam,
        measurements: list[Measurement]) -> AggResult:
    agg_shares: list[AggShare]
    agg_shares = [daf.agg_init(agg_param)
                  for _ in range(daf.SHARES)]
    for measurement in measurements:
        # Sharding
        nonce = gen_rand(daf.NONCE_SIZE)
        rand = gen_rand(daf.RAND_SIZE)
        (public_share, input_shares) = \
            daf.shard(ctx, measurement, nonce, rand)

        # Preparation, aggregation
        for j in range(daf.SHARES):
            out_share = daf.prep(ctx, j, agg_param, nonce,
                                 public_share, input_shares[j])
            agg_shares[j] = daf.agg_update(agg_param,
                                           agg_shares[j],
                                           out_share)

    # Unsharding
    num_measurements = len(measurements)
    agg_result = daf.unshard(agg_param, agg_shares,
                             num_measurements)
    return agg_result
```

The inputs to this procedure include the parameters of the aggregation function computed by the DAF: an aggregation parameter and a sequence of measurements. They also include the application context. 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

VDAFs are identical to DAFs except that preparation is an interactive process carried out by the Aggregators. If successful, this process results in each Aggregator computing an output share. The process will fail if, for example, the underlying measurement is invalid.

Failure manifests as an exception raised by one of the algorithms defined in this section. If an exception is raised during preparation, the Aggregators MUST remove the report from the batch and not attempt to aggregate it. Otherwise, a malicious Client can cause the Collector to compute a malformed aggregate result.

The remainder of this section defines the VDAF interface, denoted by Vdaf. The attributes listed in Table 3 are defined by each concrete VDAF.

Parameter	Description
ID: int	Algorithm identifier for this VDAF, in the range [0, 2**32).
SHARES: int	Number of input shares into which each measurement is sharded.
ROUNDS: int	Number of rounds of communication during preparation.
NONCE_SIZE: int	Size of each report nonce.
RAND_SIZE: int	Size of each random byte string consumed during sharding.
VERIFY_KEY_SIZE: int	Size of the verification key used during preparation.
Measurement	Type of each measurement.
PublicShare	Type of each public share.
InputShare	Type of each input share.
AggParam	Type of the aggregation parameter.
OutShare	Type of each output share.
AggShare	Type of each aggregate share.
AggResult	Type of the aggregate result.
PrepState	Type of each prep state.
PrepShare	Type of each prep share.
PrepMessage	Type of each prep message.

Table 3: Constants and types defined by each concrete VDAF.

Some of the types in the table above need to be written to the network in order to carry out the computation. It is RECOMMENDED that concrete instantiations of the Vdaf interface specify a method of encoding the PublicShare, InputShare, AggParam, AggShare, PrepShare, and PrepMessage types.

Each VDAF is identified by a unique 32-bit integer, denoted ID. Identifiers for each VDAF specified in this document are defined in Table 19. The following method is used by both Prio3 and Poplar1:

```
def domain_separation_tag(self, usage: int, ctx: bytes) -> bytes:
    """
    Format domain separation tag for this VDAF with the given
    application context and usage.

    Pre-conditions:

        - ‘usage‘ in the range ‘[0, 2**16)‘
    """
    return format_dst(0, self.ID, usage) + ctx
```

The output, called the "domain separation tag", is used in the constructions for domain separation. Function format_dst() is defined in Section 6.2.3.

5.1. Sharding

Sharding is as described for DAFs in Section 4.1. The public share and input shares encode additional information used during preparation to validate the output shares before they are aggregated (e.g., the "proof shares" in Section 7).

Like DAFs, sharding is bound to the application context via the application context string. Again, this is intended to ensure that aggregation succeeds only if the Clients and Aggregators agree on the application context. Unlike DAFs, however, disagreement on the context should manifest as a preparation failure, causing the report to be rejected without garbling the aggregate result. The application context also provides some defense-in-depth against cross protocol attacks; see Section 9.9.

5.2. Preparation

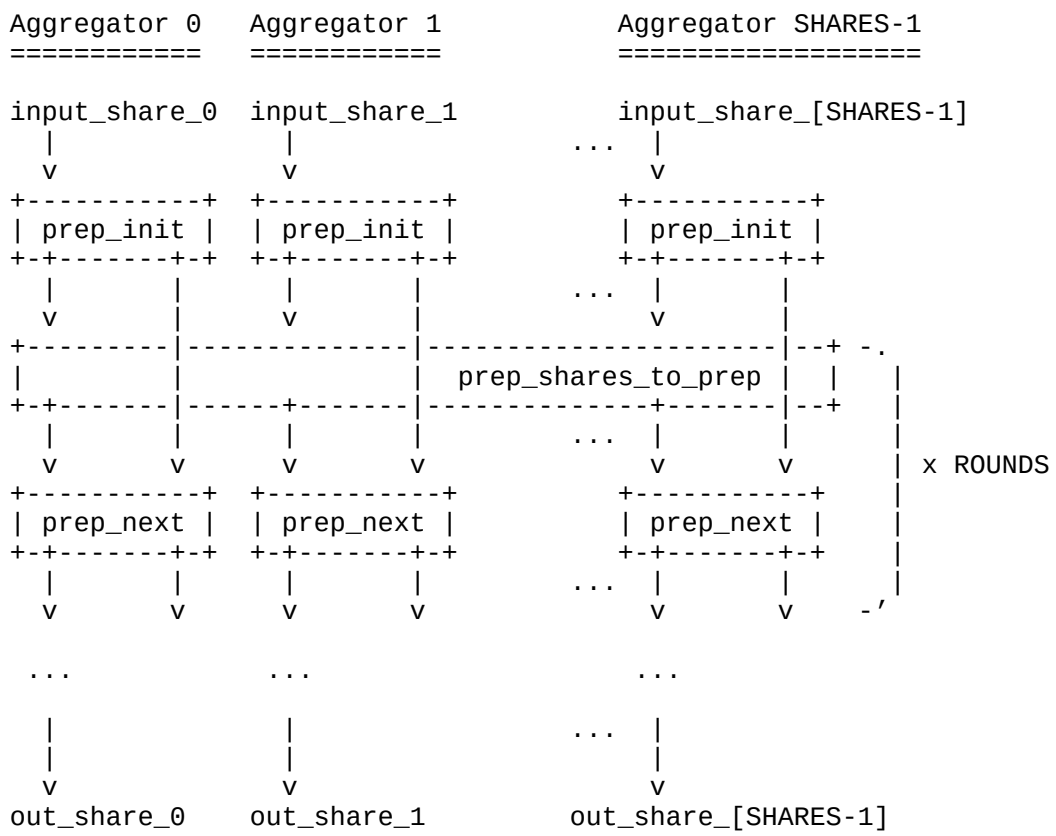

Figure 7: Illustration of interactive VDAF preparation.

Preparation is organized into a number of rounds. The number of rounds depends on the VDAF: Prio3 (Section 7) has one round and Poplar1 (Section 8) has two.

Aggregators retain some local state between successive rounds of preparation. This is referred to as "preparation state" or "prep state" for short.

During each round, each Aggregator broadcasts a message called a "preparation share", or "prep share" for short. The prep shares are then combined into a single message called the "preparation message", or "prep message". The prep message MAY be computed by any one of the Aggregators.

The prep message is disseminated to each of 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 current prep state

and the previous prep message. Its output in the last round is its output share and its output in each of the preceding rounds is a prep share.

Just as for DAFs (Section 4.2), preparation involves an aggregation parameter. The aggregation parameter is consumed by each Aggregator before the first round of communication.

Unlike DAFs, VDAF preparation involves a secret "verification key" held by each of the Aggregators. This key is used to verify validity of the output shares they compute. It is up to the high level protocol in which the VDAF is used to arrange for the distribution of the verification key prior to generating and processing reports. See Section 9 for details.

Preparation is implemented by the following set of algorithms:

* vdaf.prep_init(verify_key: bytes, ctx: bytes, agg_id: int, agg_param: AggParam, nonce: bytes, public_share: PublicShare, input_share: InputShare) -> tuple[PrepState, PrepShare] is the deterministic preparation state initialization algorithm run by each Aggregator. It consumes the shared verification key, the application context, the Aggregator’s unique identifier, the aggregation parameter chosen by the Collector, the report nonce, the public share, and one of the input shares generated by the Client. It produces the Aggregator’s initial prep state and prep share.

  Protocols MUST ensure that public share consumed by each of the Aggregators is identical. This is security critical for VDAFs such as Poplar1.

  Pre-conditions:

  - verify_key MUST have length vdaf.VERIFY_KEY_SIZE.

  - agg_id MUST be the integer in the range [0, vdaf.SHARES) that matches the index of input_share in the sequence of input shares output by the Client.

  - nonce MUST have length vdaf.NONCE_SIZE.

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

* vdaf.prep_next(ctx: bytes, prep_state: PrepState, prep_msg: PrepMessage) -> tuple[PrepState, PrepShare] | OutShare is the deterministic preparation state update algorithm run by each Aggregator. It updates the Aggregator's prep state (prep_state) and returns either its next prep state and prep share for the next round or, if this is the last round, its output share.

An exception may be raised by one of these algorithms, in which case the report MUST be deemed invalid and not processed any further.

Implementation note: The preparation process 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 the underlying data 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.

5.3. Validity of Aggregation Parameters

Aggregation parameter validation is as described for DAFs in Section 4.3. Again, each Aggregator MUST validate each aggregation parameter received from the Collector before beginning preparation with that parameter.

5.4. Aggregation

Aggregation is identical to DAF aggregation as described in Section 4.4. As with DAFs, computation of the VDAF aggregate is not usually sensitive to the order in which output shares are aggregated. See Section 4.4.1.

5.5. Unsharding

Unsharding is identical to DAF unsharding as described in Section 4.5.

5.6. Execution of a VDAF

The following function describes the sequence of computations that are carried out during VDAF execution:

```
   def run_vdaf(
           vdaf: Vdaf[
               Measurement,
               AggParam,
               PublicShare,
               InputShare,
               OutShare,
               AggShare,
               AggResult,
               PrepState,
               PrepShare,
               PrepMessage,
           ],
           verify_key: bytes,
           agg_param: AggParam,
           ctx: bytes,
           measurements: list[Measurement]) -> AggResult:
       """
       Execute the VDAF for the given measurements, aggregation
       parameter (‘agg_param‘), application context (‘ctx‘), and
       verification key (‘verify_key‘).
       """
       agg_shares = [vdaf.agg_init(agg_param)
                     for _ in range(vdaf.SHARES)]
       for measurement in measurements:
           # Sharding: The Client shards its measurement into a report
           # consisting of a public share and a sequence of input
           # shares.
           nonce = gen_rand(vdaf.NONCE_SIZE)
           rand = gen_rand(vdaf.RAND_SIZE)
           (public_share, input_shares) = \
               vdaf.shard(ctx, measurement, nonce, rand)

           # Initialize preparation: Each Aggregator receives its report
           # share (the public share and its input share) from the
           # Client and initializes preparation.
           prep_states = []
           outbound_prep_shares = []
           for j in range(vdaf.SHARES):
               (state, share) = vdaf.prep_init(verify_key, ctx, j,
                                               agg_param,
                                               nonce,
                                               public_share,
                                               input_shares[j])
               prep_states.append(state)
               outbound_prep_shares.append(share)

           # Complete preparation: The Aggregators execute each round of
```

```
        # preparation until each computes an output share. A round
        # begins by gathering the prep shares and combining them into
        # the prep message. The round ends when each uses the prep
        # message to transition to the next state.
        for i in range(vdaf.ROUNDS - 1):
            prep_msg = vdaf.prep_shares_to_prep(ctx,
                                                agg_param,
                                                outbound_prep_shares)

            outbound_prep_shares = []
            for j in range(vdaf.SHARES):
                out = vdaf.prep_next(ctx, prep_states[j], prep_msg)
                assert isinstance(out, tuple)
                (prep_states[j], prep_share) = out
                outbound_prep_shares.append(prep_share)

        prep_msg = vdaf.prep_shares_to_prep(ctx,
                                            agg_param,
                                            outbound_prep_shares)

        # Aggregation: Each Aggregator updates its aggregate share
        # with its output share.
        for j in range(vdaf.SHARES):
            out_share = vdaf.prep_next(ctx, prep_states[j], prep_msg)
            assert not isinstance(out_share, tuple)
            agg_shares[j] = vdaf.agg_update(agg_param,
                                            agg_shares[j],
                                            out_share)

    # Unsharding: The Collector receives the aggregate shares from
    # the Aggregators and combines them into the aggregate result.
    num_measurements = len(measurements)
    agg_result = vdaf.unshard(agg_param, agg_shares,
                              num_measurements)
    return agg_result
```

Depending on the VDAF, preparation and aggregation may be carried out multiple times on the same sequence of reports.

In practice, VDAF execution is distributed across Clients, Aggregators, and Collectors that exchange messages (i.e., report shares, prep shares, and aggregate shares) over an insecure network. The application must therefore take some additional steps in order to securely execute the VDAF in this environment. See Section 9 for details.

5.7. Communication Patterns for Preparation

The only stage of VDAF execution that requires interaction is preparation (Section 5.2). There are a number of ways to coordinate this interaction; the best strategy depends largely on the number of Aggregators (i.e., vdaf.SHARES). This section describes two strategies, one specialized for two Aggregators and another that is suitable for any number of Aggregators.

In each round of preparation, each Aggregator writes a prep share to some broadcast channel, which is then processed into the prep message using the public prep_shares_to_prep() algorithm and broadcast to the Aggregators to start the next round. The goal of this section is to realize this broadcast channel.

The state machine of each Aggregator is shown below.

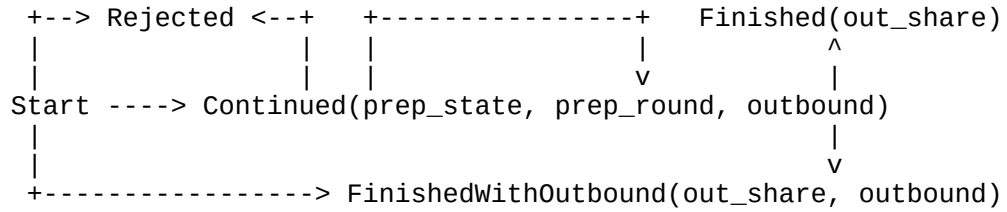

Figure 8: State machine of VDAF preparation.

State transitions are made when the state is acted upon by the Aggregator's local inputs and/or messages sent by its co-Aggregators. The initial state is Start. The terminal states are: Rejected, indicating that the report cannot be processed any further; Finished(out_share), indicating that the Aggregator has recovered an output share out_share; and FinishedWithOutbound(out_share, outbound), indicating that the Aggregator has recovered an output share, and has one more outbound message to send. For completeness, these states are defined in Appendix B.

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

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

5.7.1. The Ping-Pong Topology (Only Two Aggregators)

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

In this section, the initiating Aggregator is called the Leader and the responding Aggregator is called the Helper. The high-level idea is that the Leader and Helper will take turns running the computation locally until input from their peer is required:

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

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

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

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

Protocol messages are specified in the presentation language of TLS; see Section 3 of [RFC8446]. Each message is structured as follows:

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

struct {
  MessageType type;
  select (Message.type) {
    case initialize:
      opaque prep_share<0..4294967295>;
    case continue:
      opaque prep_msg<0..4294967295>;
      opaque prep_share<0..4294967295>;
    case finish:
      opaque prep_msg<0..4294967295>;
  };
} Message;

/* note that 4294967295 is 2 ** 32 - 1 */
```

These messages trigger all transitions in the state machine in Figure 8, except for the Leader’s initial transition. The Leader’s state is initialized using its local inputs with the following method on class Vdaf:

```
def ping_pong_leader_init(
        self,
        vdaf_verify_key: bytes,
        ctx: bytes,
        agg_param: bytes,
        nonce: bytes,
        public_share: bytes,
        input_share: bytes) -> Continued | Rejected:
    """Called by the Leader to initialize ping-ponging."""
    try:
        (prep_state, prep_share) = self.prep_init(
            vdaf_verify_key,
            ctx,
            0,
            self.decode_agg_param(agg_param),
            nonce,
            self.decode_public_share(public_share),
            self.decode_input_share(0, input_share),
        )

        encoded_prep_share = self.encode_prep_share(prep_share)
        return Continued(
            prep_state, 0,
            encode(0, encoded_prep_share),  # initialize
        )
    except Exception:
        return Rejected()
```

The output is the State to which the Leader has transitioned. If the Leader's state is Rejected, then processing halts. Otherwise, if the state is Continued, then processing continues. In this case, the state also includes the Leader's outbound message. The function encode is used to encode the outbound message, which has the message type of initialize (identified by the number 0).

To continue processing the report, the Leader sends the outbound message to the Helper. The Helper's initial transition is computed using the following procedure:

```
   def ping_pong_helper_init(
       self,
       vdaf_verify_key: bytes,
       ctx: bytes,
       agg_param: bytes,
       nonce: bytes,
       public_share: bytes,
       input_share: bytes,
       inbound: bytes,  # encoded ping pong Message
   ) -> Continued | FinishedWithOutbound | Rejected:
       """
       Called by the Helper in response to the Leader's initial
       message.
       """

       try:
           (prep_state, prep_share) = self.prep_init(
               vdaf_verify_key,
               ctx,
               1,
               self.decode_agg_param(agg_param),
               nonce,
               self.decode_public_share(public_share),
               self.decode_input_share(1, input_share),
           )

           (inbound_type, inbound_items) = decode(inbound)
           if inbound_type != 0:  # initialize
               return Rejected()

           encoded_prep_share = inbound_items[0]
           prep_shares = [
               self.decode_prep_share(prep_state, encoded_prep_share),
               prep_share,
           ]
           return self.ping_pong_transition(
               ctx,
               self.decode_agg_param(agg_param),
               prep_shares,
               prep_state,
               0,
           )
       except Exception:
           return Rejected()
```

The procedure decode() decodes the inbound message and returns the MessageType variant (initialize, continue, or finish) and the fields of the message. The procedure ping_pong_transition() takes in the prep shares, combines them into the prep message, and computes the next prep state of the caller:

```
def ping_pong_transition(
        self,
        ctx: bytes,
        agg_param: AggParam,
        prep_shares: list[PrepShare],
        prep_state: PrepState,
        prep_round: int) -> Continued | FinishedWithOutbound:
    prep_msg = self.prep_shares_to_prep(ctx,
                                        agg_param,
                                        prep_shares)
    encoded_prep_msg = self.encode_prep_msg(prep_msg)
    out = self.prep_next(ctx, prep_state, prep_msg)
    if prep_round+1 == self.ROUNDS:
        return FinishedWithOutbound(
            out,
            encode(2, encoded_prep_msg),  # finalize
        )
    (prep_state, prep_share) = cast(
        tuple[PrepState, PrepShare], out)
    encoded_prep_share = self.encode_prep_share(prep_share)
    return Continued(
        prep_state, prep_round+1,
        encode(1, encoded_prep_msg, encoded_prep_share)  # continue
    )
```

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

To continue processing, the Helper sends the outbound message to the Leader. The Leader computes its next state transition using the following method on class Vdaf:

```
def ping_pong_leader_continued(
    self,
    ctx: bytes,
    agg_param: bytes,
    state: Continued,
    inbound: bytes,  # encoded ping pong Message
) -> State:
    """
```

```
        Called by the Leader to start the next step of ping-ponging.
        """
        return self.ping_pong_continued(
            True, ctx, agg_param, state, inbound)

    def ping_pong_continued(
        self,
        is_leader: bool,
        ctx: bytes,
        agg_param: bytes,
        state: Continued,
        inbound: bytes,  # encoded ping pong Message
    ) -> State:
        try:
            prep_round = state.prep_round

            (inbound_type, inbound_items) = decode(inbound)
            if inbound_type == 0:  # initialize
                return Rejected()

            encoded_prep_msg = inbound_items[0]
            prep_msg = self.decode_prep_msg(
                state.prep_state,
                encoded_prep_msg,
            )
            out = self.prep_next(ctx, state.prep_state, prep_msg)
            if prep_round+1 < self.ROUNDS and \
                    inbound_type == 1:  # continue
                (prep_state, prep_share) = cast(
                    tuple[PrepState, PrepShare], out)
                encoded_prep_share = inbound_items[1]
                prep_shares = [
                    self.decode_prep_share(
                        prep_state,
                        encoded_prep_share,
                    ),
                    prep_share,
                ]
                if is_leader:
                    prep_shares.reverse()
                return self.ping_pong_transition(
                    ctx,
                    self.decode_agg_param(agg_param),
                    prep_shares,
                    prep_state,
                    prep_round+1,
                )
            elif prep_round+1 == self.ROUNDS and \
```

```
                    inbound_type == 2:  # finish
                return Finished(out)
            else:
                return Rejected()
        except Exception:
            return Rejected()
```

If the Leader's state is Finished or Rejected, then processing halts. Otherwise, if the Leader's state is Continued or FinishedWithOutbound, the Leader sends the outbound message to the Helper. The Helper computes its next state transition using the following method on class Vdaf:

```
def ping_pong_helper_continued(
    self,
    ctx: bytes,
    agg_param: bytes,
    state: Continued,
    inbound: bytes,  # encoded ping pong Message
) -> State:
    """Called by the Helper to continue ping-ponging."""
    return self.ping_pong_continued(
        False, ctx, agg_param, state, inbound)
```

They continue in this way until processing halts. Note that, depending on the number of rounds of preparation that are required, when one party reaches the Finished state, there may be one more message to send before the peer can also finish processing (i.e., the outbound message is not None).

5.7.2. The Star Topology (Any Number of Aggregators)

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

Again, one Aggregator initiates the computation. This Aggregator is called the Leader and all other Aggregators are called Helpers.

At the start of each round, the Leader requests from each Helper its prep share. After gathering each of the prep shares, the Leader computes the next prep message (via vdaf.prep_shares_to_prep()) and broadcasts it to the Helpers. At this point, each Aggregator runs vdaf.prep_next() locally to either recover an output share or, if more rounds of preparation are required, compute its updated state and prep share. If another round is required, then the Helper responds to the broadcast message with its next prep share.

The Aggregators proceed in this way until each recovers an output share or some step of the computation fails.

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: int is the prime modulus that defines the field.

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

Concrete fields, i.e., subclasses of Field, implement the following class methods:

* Field.zeros(length: int) -> list[Self] returns a vector of zeros of the requested length.

  Pre-conditions:

  - length MUST be greater than or equal 0.

  Post-conditions:

  - The length of the output MUST be length.

* Field.rand_vec(length: int) -> list[Self] returns a vector of random field elements and has the same pre- and post-conditions as for Field.zeros(). Note that this function is not used normatively in the specification of either Prio3 or Poplar1.

A field element is an instance of a concrete Field. Addition, subtraction, multiplication, division, negation, and inversion are denoted, respectively, x + y, x - y, x * y, x / y, -x, and x.inv().

Conversion of a field element to an int is denoted by x.int(). Likewise, each concrete Field implements a constructor for converting an integer into a field element:

* Field(integer: int) returns integer represented as a field element. The value of integer MUST be in the range (-Field.MODULUS, Field.MODULUS); negative values are treated as negations.

6.1.1. Auxiliary Functions

The following class methods on Field are used to encode and decode vectors of field elements as byte strings:

```
def encode_vec(cls, vec: list[Self]) -> bytes:
    """
    Encode a vector of field elements ‘vec‘ as a byte string.
    """
    encoded = bytes()
    for x in vec:
        encoded += to_le_bytes(x.int(), cls.ENCODED_SIZE)
    return encoded

def decode_vec(cls, encoded: bytes) -> list[Self]:
    """
    Parse a vector of field elements from ‘encoded‘.
    """
    if len(encoded) % cls.ENCODED_SIZE != 0:
        raise ValueError(
            'input length must be a multiple of the size of an '
            'encoded field element')

    vec = []
    while len(encoded) > 0:
        (encoded_x, encoded) = front(cls.ENCODED_SIZE, encoded)
        x = from_le_bytes(encoded_x)
        if x >= cls.MODULUS:
            raise ValueError('modulus overflow')
        vec.append(cls(x))
    return vec
```

Field provides the following class methods for representing an integer as a sequence of field elements, each of which represents a bit of the input. These are used to encode measurements in some variants of Prio3 (Section 7.4).

```
   def encode_into_bit_vec(
           cls,
           val: int,
           bits: int) -> list[Self]:
       """
       Encode the bit representation of ‘val‘ with at most ‘bits‘ number
       of bits, as a vector of field elements.

       Pre-conditions:

           - ‘val >= 0‘
           - ‘bits >= 0‘
       """
       if val >= 2 ** bits:
           # Sanity check we are able to represent ‘val‘ with ‘bits‘
           # number of bits.
           raise ValueError("Number of bits is not enough to represent "
                            "the input integer.")
       encoded = []
       for l in range(bits):
           encoded.append(cls((val >> l) & 1))
       return encoded

   def decode_from_bit_vec(cls, vec: list[Self]) -> Self:
       """
       Decode the field element from the bit representation, expressed
       as a vector of field elements ‘vec‘.

       This may also be used with secret shares of a bit representation,
       since it is linear.
       """
       bits = len(vec)
       if cls.MODULUS >> bits == 0:
           raise ValueError("Number of bits is too large to be "
                            "represented by field modulus.")
       decoded = cls(0)
       for (l, bit) in enumerate(vec):
           decoded += cls(1 << l) * bit
       return decoded
```

Finally, the following functions define arithmetic on vectors over a finite field. Note that an exception is raised by each function if the operands are not the same length.

```
   def vec_sub(left: list[F], right: list[F]) -> list[F]:
       """
       Subtract the right operand from the left and return the result.
       """
       if len(left) != len(right):
           raise ValueError("mismatched vector sizes")
       return list(map(lambda x: x[0] - x[1], zip(left, right)))

   def vec_add(left: list[F], right: list[F]) -> list[F]:
       """Add the right operand to the left and return the result."""
       if len(left) != len(right):
           raise ValueError("mismatched vector sizes")
       return list(map(lambda x: x[0] + x[1], zip(left, right)))

   def vec_neg(vec: list[F]) -> list[F]:
       """Negate the input vector."""
       return list(map(lambda x: -x, vec))
```

6.1.2. NTT-Friendly Fields

Some VDAFs, including Prio3, require fields that are suitable for efficient computation of the number theoretic transform (NTT) [SML24], as this allows for fast polynomial interpolation. Specifically, a field is said to be "NTT-friendly" if, in addition to the interface described in Section 6.1, it provides the following interface:

* Field.gen() -> Self is a class method that returns the generator of a large subgroup of the multiplicative group. To be NTT-friendly, the order of this subgroup MUST be a power of 2.

* GEN_ORDER: int is the order of the multiplicative subgroup generated by Field.gen(). This is the smallest positive integer for which Field.gen() ** Field.GEN_ORDER == Field(1).

The size of the subgroup dictates how large interpolated polynomials can be. It is RECOMMENDED that a generator is chosen with order at least 2**20.

6.1.3. Parameters

Table 4 defines finite fields used in the remainder of this document.

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

Table 4: Parameters for the finite fields used in this document.

6.2. Extendable Output Functions (XOFs)

VDAFs in this specification use eXtendable Output Functions (XOFs) for two purposes:

1. Extracting short, pseudorandom strings called "seeds" from high entropy inputs

2. Expanding seeds into long, pseudorandom outputs

Concrete XOFs implement a class Xof providing the following interface:

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

* Xof(seed: bytes, dst: bytes, binder: bytes) constructs an instance of the XOF from the given seed and a domain separation tag and binder string as defined in Section 6.2.3. The length of the seed will typically be SEED_SIZE, but some XOFs may support multiple seed sizes. The seed MUST be generated securely, i.e., it is either the output of a CSPRNG or a previous invocation of the XOF.

* xof.next(length: int) returns the next chunk of the output of the initialized XOF as a byte string. The length of the chunk MUST be length.

The following methods are provided for all concrete XOFs. The first is a class method used to derive a fresh seed from an existing one. The second is an instance method used to compute a sequence of field elements. The third is a class method that provides a one-shot interface for expanding a seed into a field vector.

```
   def derive_seed(cls,
                   seed: bytes,
                   dst: bytes,
                   binder: bytes) -> bytes:
       """
       Derive a new seed.

       Pre-conditions:

           - `len(seed) == cls.SEED_SIZE`
       """
       xof = cls(seed, dst, binder)
       return xof.next(cls.SEED_SIZE)

   def next_vec(self, field: type[F], length: int) -> list[F]:
       """
       Output the next `length` field elements.

       Pre-conditions:

           - `field` is sub-class of `Field`
           - `length > 0`
       """
       m = next_power_of_2(field.MODULUS) - 1
       vec: list[F] = []
       while len(vec) < length:
           x = from_le_bytes(self.next(field.ENCODED_SIZE))
           x &= m
           if x < field.MODULUS:
               vec.append(field(x))
       return vec

   def expand_into_vec(cls,
                       field: type[F],
                       seed: bytes,
                       dst: bytes,
                       binder: bytes,
                       length: int) -> list[F]:
       """
       Expand the input `seed` into a vector of `length` field elements.

       Pre-conditions:

           - `field` is sub-class of `Field`
           - `len(seed) == cls.SEED_SIZE`
           - `length > 0`
       """
       xof = cls(seed, dst, binder)
```

```
        return xof.next_vec(field, length)
```

6.2.1. XofTurboShake128

This section describes XofTurboShake128, an XOF based on the TurboSHAKE128 function specified in [TurboSHAKE]. This XOF is RECOMMENDED for all use cases for DAFs and VDAFs.

> TODO Update the [TurboSHAKE] reference to point to the RFC instead of the draft.

Pre-conditions:

* The default seed length is 32. The seed MAY have a different length, but it MUST not exceed 255. Otherwise initialization will raise an exception.

* The length of the domain separation string dst passed to XofTurboShake128 MUST NOT exceed 65535 bytes. Otherwise initialization will raise an exception.

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

    # Associated parameters
    SEED_SIZE = 32

    def __init__(self, seed: bytes, dst: bytes, binder: bytes):
        self.l = 0
        self.m = \
            to_le_bytes(len(dst), 2) + dst \
            to_le_bytes(len(seed), 1) + seed + \
            binder

    def next(self, length: int) -> bytes:
        self.l += length

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

6.2.2. XofFixedKeyAes128

The XOF in the previous section can be used safely wherever a XOF is needed in this document. However, there are some situations where TurboSHAKE128 creates a performance bottleneck and a more efficient XOF can be used safely instead.

This section describes XofFixedKeyAes128, which is used to implement the IDPF of Poplar1 (Section 8.3). It is NOT RECOMMENDED to use this XOF for any other purpose. See Section 9.6 for a more detailed discussion.

XofFixedKeyAes128 uses the AES-128 blockcipher [AES] for most of the computation, thereby taking advantage of the hardware implementations of this blockcipher that are widely available. AES-128 is used in a fixed-key mode of operation; the key is derived during initialization using TurboSHAKE128.

Pre-conditions:

* The length of the seed MUST be 16.

* The length of the domain separation string dst passed to XofFixedKeyAes128 MUST NOT exceed 65535 bytes. Otherwise initialization will raise an exception.

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

    # Associated parameters
    SEED_SIZE = 16

    def __init__(self, seed: bytes, dst: bytes, binder: bytes):
        if len(seed) != self.SEED_SIZE:
            raise ValueError("incorrect seed size")

        self.length_consumed = 0

        # Use TurboSHAKE128 to derive a key from the binder string
        # and domain separation tag. Note that the AES key does not
        # need to be kept secret from any party. However, when used
        # with an IDPF, we require the binder to be a random nonce.
        #
        # Implementation note: this step can be cached across XOF
        # evaluations with many different seeds.
```

```
            dst_length = to_le_bytes(len(dst), 2)
            self.fixed_key = TurboSHAKE128(
                dst_length + dst + binder,
                2,
                16,
            )
            self.seed = seed

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

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

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

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

6.2.3. The Domain Separation Tag and Binder String

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

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

2. The "class" of the algorithm using the output (e.g., DAF, VDAF, or IDPF as defined in Section 8.1);

3. A unique identifier for the algorithm (e.g., VDAF.ID); and

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

The following algorithm is used in the remainder of this document in order to format the domain separation tag:

```
def format_dst(algo_class: int,
               algo: int,
               usage: int) -> bytes:
    """
    Format XOF domain separation tag.

    Pre-conditions:

        - 'algo_class' in the range '[0, 2**8)'
        - 'algo' in the range '[0, 2**32)'
        - 'usage' in the range '[0, 2**16)'
    """
    return concat([
        to_be_bytes(VERSION, 1),
        to_be_bytes(algo_class, 1),
        to_be_bytes(algo, 4),
        to_be_bytes(usage, 2),
    ])
```

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

7. Prio3

This section describes Prio3, a VDAF for general-purpose aggregation. Prio3 is suitable for a wide variety of aggregation functions, including (but not limited to) sum, mean, standard deviation, histograms, and linear regression. It is compatible with any aggregation function that has the following structure:

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

* Measurement validity is determined by an "arithmetic circuit" evaluated over the encoded measurement. An arithmetic circuit is a function comprised of arithmetic operations in the field. (These are specified in full detail in Section 7.3.2.)

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

Clients protect the privacy of their measurements by secret sharing them and distributing the shares among the Aggregators. To ensure each measurement is valid, the Aggregators run a multi-party computation on their shares, the result of which is the output of the arithmetic circuit. This involves verification of a "Fully Linear Proof (FLP)" (Section 7.1) generated by the Client. FLPs are the core component of Prio3, as they specify the types of measurements and how they are encoded, verified, and aggregated. In fact Prio3 can be thought of as a transformation of an FLP into a VDAF.

Prio3 does not have an aggregation parameter. Instead, each output share is derived from each input share by applying a fixed map. See Section 8 for an example of a VDAF that makes meaningful use of the aggregation parameter.

The remainder of this section is structured as follows. The interface of 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, variants of Prio3 for various types of aggregation tasks are specified in Section 7.4. Test vectors for each variant can be found in Appendix C.

7.1. Fully Linear Proofs (FLPs)

Conceptually, an FLP is a two-party protocol executed by a prover and a verifier. The verifier is restricted to only access the messages it receives from the prover via linear queries. In actual use in Prio3, however, the prover’s computation is carried out by the Client, and the verifier’s computation is distributed among the Aggregators. The Client generates a "proof" of its measurement’s validity and distributes shares of the proof to the Aggregators. During preparation, each Aggregator performs some computation on its measurement share and proof share locally, then broadcasts the result in its prep share. The validity decision is then made by the prep_shares_to_prep() algorithm (Section 5.2).

As usual, the interface implemented by a concrete FLP is described in terms of an object flp of type 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. A concrete FLP specifies the following algorithms for generating and verifying proofs of validity (encoding is described below in Section 7.1.1):

* flp.prove(meas: list[F], prove_rand: list[F], joint_rand: list[F]) -> list[F] is the proof-generation algorithm run by the prover. Its inputs are the encoded measurement, the "prover randomness" prove_rand, and the "joint randomness" joint_rand. The prover randomness is used only by the prover, but the joint randomness is shared by both the prover and verifier.

* flp.query(meas: list[F], proof: list[F], query_rand: list[F], joint_rand: list[F], num_shares: int) -> list[F] is the linear query algorithm run by the verifier on the encoded measurement and proof. The result of the query (i.e., the output of this function) is called the "verifier message". In addition to the measurement and proof, this algorithm takes as input the query randomness query_rand and the joint randomness joint_rand. The former is used only by the verifier. num_shares specifies the number of shares (more on this below).

* flp.decide(verifier: list[F]) -> bool is the deterministic decision algorithm run by the verifier. It takes as input the verifier message and outputs a boolean indicating if the measurement from which it was generated is valid.

This application requires that the FLP is "fully linear" in the sense defined in [BBCGGI19]. As a practical matter, what this property implies is that, when run on a share of the measurement and proof, the query algorithm outputs a share of the verifier message (hereafter the "verifier share"). Furthermore, the privacy property of the FLP system ensures that the verifier message reveals nothing about the measurement other than the fact that it is valid. Therefore, to decide if a measurement is valid, the Aggregators will run the query algorithm locally, exchange verifier shares, combine them to recover the verifier message, and run the decision algorithm.

The query algorithm includes a parameter num_shares that specifies the number of shares of the measurement and proof that were generated. If these data are not secret shared, then num_shares == 1. This parameter is useful for normalizing constants in arithmetic circuits so that each Aggregator properly computes a secret share of the circuit’s output. See Section 7.3 for details.

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

```
   def run_flp(
           flp: Flp[Measurement, AggResult, F],
           meas: list[F],
           num_shares: int) -> bool:
       """Run the FLP on an encoded measurement."""

       joint_rand = flp.field.rand_vec(flp.JOINT_RAND_LEN)
       prove_rand = flp.field.rand_vec(flp.PROVE_RAND_LEN)
       query_rand = flp.field.rand_vec(flp.QUERY_RAND_LEN)

       # Prover generates the proof.
       proof = flp.prove(meas, prove_rand, joint_rand)

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

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

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

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

The proof system is designed so that, if meas is valid, then run_flp(flp, meas, num_shares) always returns True. On the other hand, if meas is invalid, then as long as joint_rand and query_rand

are generated uniform randomly, the output is False with high probability. False positives are possible: there is a small probability that a verifier accepts an invalid input as valid. An FLP is said to be "sound" if this probability is sufficiently small. The soundness of the FLP depends on a variety of parameters, like the length of the input and the size of the field. See Section 7.3 for details.

Note that soundness of an FLP system is not the same as verifiability for the VDAF that uses it. In particular, soundness of the FLP is necessary, but insufficient for verifiability of Prio3 (Section 7). See Section 9.7 for details.

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

Parameter	Description
PROVE_RAND_LEN: int	Length of the prover randomness, the number of random field elements consumed by the prover when generating a proof.
QUERY_RAND_LEN: int	Length of the query randomness, the number of random field elements consumed by the verifier.
JOINT_RAND_LEN: int	Length of the joint randomness, the number of random field elements shared by the prover and verifier.
MEAS_LEN: int	Length of the encoded measurement (Section 7.1.1).
OUTPUT_LEN: int	Length of the aggregatable output (Section 7.1.1).
PROOF_LEN: int	Length of the proof.
VERIFIER_LEN: int	Length of the verifier message generated by querying the measurement and proof.
Measurement	Type of the measurement.
AggResult	Type of the aggregate result.
field: type[F]	Class object for the field (Section 6.1).

Table 5: FLP parameters.

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) -> list[F] encodes a raw measurement as a vector of field elements.

  Post-conditions:

  - The encoded measurement MUST have length flp.MEAS_LEN.

For some FLPs, the encoded measurement also includes redundant field elements that are useful for checking the proof, but which are not needed after the proof has been checked. An example is the Sum type defined in Section 7.4.2 for which each measurement is an integer in the range [0, max_measurement]. The range check requires encoding the measurement with several field elements, though just one is needed for aggregation. Thus the FLP defines an algorithm for truncating the encoded measurement to the length of the aggregatable output:

* flp.truncate(meas: list[F]) -> list[F] maps an encoded measurement (e.g., the bit-encoding of the measurement) to an aggregatable output (e.g., the singleton vector containing the measurement).

  Pre-conditions:

  - The length of the input MUST be flp.MEAS_LEN

  Post-conditions:

  - The length of the output MUST be flp.OUTPUT_LEN.

Once the aggregate shares have been transmitted to the Collector, their sum can be converted into the aggregate result. This could be a projection from the FLP's field to the integers, or it could include additional post-processing. Either way, this functionality is implemented by the following method:

* flp.decode(output: list[F], num_measurements: int) -> AggResult maps a sum of aggregate shares to an aggregate result.

  Pre-conditions:

  - The length of the output MUST be OUTPUT_LEN.

  - num_measurements MUST equal the number of measurements that were aggregated.

Taken together, these three functionalities correspond to the notion of "Affine-aggregatable encodings (AFEs)" from [CGB17].

7.1.2. Multiple Proofs

It is sometimes desirable to generate and verify multiple independent proofs for the same input. First, this improves the soundness of the proof system without having to change any of its parameters. Second, it allows a smaller field to be used (e.g., replace Field128 with Field64) without sacrificing soundness. This is useful because it reduces the overall communication of the protocol. (This is a trade-off, of course, since generating and verifying more proofs requires more time.) Given these benefits, this feature is implemented by Prio3 (Section 7).

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

See Section 9.7 for guidance on choosing the field size and number of proofs.

7.2. Specification

This section specifies Prio3, an implementation of the Vdaf interface defined in Section 5. The parameters 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. These methods refer to constants enumerated in Table 7.

Parameter	Value
flp	An instance of Flp (Section 7.1).
xof	XofTurboShake128 (Section 6.2.1)
PROOFS	Any int in the range [1, 256).
VERIFY_KEY_SIZE	xof.SEED_SIZE
RAND_SIZE	xof.SEED_SIZE * SHARES if flp.JOINT_RAND_LEN == 0 else 2 * xof.SEED_SIZE * SHARES
NONCE_SIZE	16
ROUNDS	1
SHARES	Any int in the range [2, 256).
Measurement	As defined by flp.
AggParam	None
PublicShare	Optional[list[bytes]]
InputShare	tuple[list[F], list[F], Optional[bytes]] | tuple[bytes, Optional[bytes]]
OutShare	list[F]
AggShare	list[F]
AggResult	As defined by flp.
PrepState	tuple[list[F], Optional[bytes]]
PrepShare	tuple[list[F], Optional[bytes]]
PrepMessage	Optional[bytes]

Table 6: Parameters for Prio3.

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

Table 7: Constants used by Prio3.

7.2.1. Sharding

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

The sharding algorithm involves the following steps:

1. Encode the Client’s measurement as specified by the FLP

2. Shard the measurement into a sequence of measurement shares

3. Derive the joint randomness from the measurement shares and nonce

4. Generate the proof using the derived joint randomness

5. Shard the proof into a sequence of proof shares

As described in Section 7.1.2, the probability of an invalid measurement being deemed valid can be decreased by generating and verifying multiple proofs. To support this:

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

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

Depending on the FLP, joint randomness may not be required. In particular, when flp.JOINT_RAND_LEN == 0, the Client does not derive the joint randomness (Step 3).

The sharding algorithm is specified below:

```
def shard(
        self,
        ctx: bytes,
        measurement: Measurement,
        nonce: bytes,
        rand: bytes) -> tuple[
            Optional[list[bytes]],
            list[Prio3InputShare]]:
    if len(nonce) != self.NONCE_SIZE:
        raise ValueError("incorrect nonce size")
    if len(rand) != self.RAND_SIZE:
        raise ValueError("incorrect size of random bytes argument")

    l = self.xof.SEED_SIZE
    seeds = [rand[i:i + l] for i in range(0, self.RAND_SIZE, l)]

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

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

7.2.1.1. FLPs Without Joint Randomness

The following method is used for FLPs that do not require joint randomness, i.e., when flp.JOINT_RAND_LEN == 0. It consists of the following steps:

1. Shard the encoded measurement into shares

2. Generate proofs and shard each into shares

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

Only one pair of measurement and proof(s) share (called the "Leader" shares) are vectors of field elements. The other shares (called the "Helper" shares) are represented instead by an XOF seed, which is expanded into vectors of field elements. The methods on Prio3 for deriving the prover randomness, measurement shares, and proof shares are defined in Section 7.2.6.

```
def shard_without_joint_rand(
        self,
        ctx: bytes,
        meas: list[F],
        seeds: list[bytes]) -> tuple[
            Optional[list[bytes]],
            list[Prio3InputShare[F]]]:
    helper_shares, seeds = front(self.SHARES - 1, seeds)
    (prove_seed,), seeds = front(1, seeds)

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

    # Generate and shard each proof into shares.
    prove_rands = self.prove_rands(ctx, prove_seed)
    leader_proofs_share = []
    for _ in range(self.PROOFS):
        prove_rand, prove_rands = front(
            self.flp.PROVE_RAND_LEN, prove_rands)
        leader_proofs_share += self.flp.prove(meas, prove_rand, [])
    for j in range(self.SHARES - 1):
        leader_proofs_share = vec_sub(
            leader_proofs_share,
            self.helper_proofs_share(
                ctx,
                j + 1,
                helper_shares[j],
            ),
        )

    # Each Aggregator's input share contains its measurement share
    # and its share of the proof(s).
    input_shares: list[Prio3InputShare[F]] = []
    input_shares.append((
        leader_meas_share,
        leader_proofs_share,
```

```
            None,
        ))
        for j in range(self.SHARES - 1):
            input_shares.append((
                helper_shares[j],
                None,
            ))
        return (None, input_shares)
```

7.2.1.2. FLPs With Joint Randomness

The following method is used for FLPs that require joint randomness, i.e., for which flp.JOINT_RAND_LEN > 0. Joint randomness derivation involves an additional XOF seed for each Aggregator called the "blind". The computation involves the following steps:

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

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

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

This three-step process is designed to ensure that the joint randomness does not leak the measurement to the Aggregators while preventing a malicious Client from tampering with the joint randomness in a way that causes the Aggregators to accept an invalid measurement. To save a round of communication between the Aggregators later, the Client encodes the joint randomness parts in the public share. (See Section 7.2.2 for details.)

All functions used in the following listing are defined in Section 7.2.6:

```
def shard_with_joint_rand(
        self,
        ctx: bytes,
        meas: list[F],
        nonce: bytes,
        seeds: list[bytes]) -> tuple[
            Optional[list[bytes]],
            list[Prio3InputShare[F]]]:
    helper_seeds, seeds = front((self.SHARES - 1) * 2, seeds)
    helper_shares = [
        helper_seeds[i]
        for i in range(0, (self.SHARES - 1) * 2, 2)
    ]
```

```
        helper_blinds = [
            helper_seeds[i]
            for i in range(1, (self.SHARES - 1) * 2, 2)
        ]
        (leader_blind, prove_seed), seeds = front(2, seeds)

        # Shard the encoded measurement into shares and compute the
        # joint randomness parts.
        leader_meas_share = meas
        joint_rand_parts = []
        for j in range(self.SHARES - 1):
            helper_meas_share = self.helper_meas_share(
                ctx, j + 1, helper_shares[j])
            leader_meas_share = vec_sub(leader_meas_share,
                                        helper_meas_share)
            joint_rand_parts.append(self.joint_rand_part(
                ctx, j + 1, helper_blinds[j],
                helper_meas_share, nonce))
        joint_rand_parts.insert(0, self.joint_rand_part(
            ctx, 0, leader_blind, leader_meas_share, nonce))

        # Generate each proof and shard it into proof shares.
        prove_rands = self.prove_rands(ctx, prove_seed)
        joint_rands = self.joint_rands(
            ctx, self.joint_rand_seed(ctx, joint_rand_parts))
        leader_proofs_share = []
        for _ in range(self.PROOFS):
            prove_rand, prove_rands = front(
                self.flp.PROVE_RAND_LEN, prove_rands)
            joint_rand, joint_rands = front(
                self.flp.JOINT_RAND_LEN, joint_rands)
            leader_proofs_share += self.flp.prove(
                meas,
                prove_rand,
                joint_rand,
            )
        for j in range(self.SHARES - 1):
            leader_proofs_share = vec_sub(
                leader_proofs_share,
                self.helper_proofs_share(
                    ctx,
                    j + 1,
                    helper_shares[j],
                ),
            )

        # Each Aggregator's input share contains its measurement share,
        # share of proof(s), and blind. The public share contains the
```

```
        # Aggregators' joint randomness parts.
        input_shares: list[Prio3InputShare[F]] = []
        input_shares.append((
            leader_meas_share,
            leader_proofs_share,
            leader_blind,
        ))
        for j in range(self.SHARES - 1):
            input_shares.append((
                helper_shares[j],
                helper_blinds[j],
            ))
        return (joint_rand_parts, input_shares)
```

7.2.2. Preparation

This section describes the process of recovering output shares from the input shares. The high-level idea is that each Aggregator first queries its measurement share and proof(s) share(s) locally, then broadcasts its verifier share(s) in its prep share. The shares of verifier(s) are then combined into the verifier message(s) used to decide whether to accept.

In addition, the Aggregators must recompute the same joint randomness used by the Client to generate the proof(s). In order to avoid an extra round of communication, the Client includes the joint randomness parts in the public share. This leaves open the possibility that the Client cheated by, say, forcing the Aggregators to use joint randomness that biases the proof check procedure some way in its favor. To mitigate this, the Aggregators also check that they have all computed the same joint randomness seed before accepting their output shares. To do so, they exchange their parts of the joint randomness along with their shares of verifier(s).

Implementation note: the prep state for Prio3 includes the output share that will be released once preparation is complete. In some situations, it may be necessary for the Aggregator to encode this state as bytes and store it for retrieval later on. For all but the first Aggregator, it is possible to save storage by storing the measurement share rather than output share itself. It is relatively inexpensive to expand this seed into the measurement share, then truncate the measurement share to get the output share.

All functions used in the following listing are defined in Section 7.2.6:

```
   def prep_init(
           self,
           verify_key: bytes,
           ctx: bytes,
           agg_id: int,
           _agg_param: None,
           nonce: bytes,
           public_share: Optional[list[bytes]],
           input_share: Prio3InputShare[F]) -> tuple[
               Prio3PrepState[F],
               Prio3PrepShare[F]]:
       joint_rand_parts = public_share
       (meas_share, proofs_share, blind) = \
           self.expand_input_share(ctx, agg_id, input_share)
       out_share = self.flp.truncate(meas_share)

       # Compute the joint randomness.
       joint_rand: list[F] = []
       corrected_joint_rand_seed, joint_rand_part = None, None
       if self.flp.JOINT_RAND_LEN > 0:
           assert blind is not None
           assert joint_rand_parts is not None
           joint_rand_part = self.joint_rand_part(
               ctx, agg_id, blind, meas_share, nonce)
           joint_rand_parts = list(joint_rand_parts)
           joint_rand_parts[agg_id] = joint_rand_part
           corrected_joint_rand_seed = self.joint_rand_seed(
               ctx, joint_rand_parts)
           joint_rands = self.joint_rands(
               ctx, corrected_joint_rand_seed)

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

```
            )

        prep_state = (out_share, corrected_joint_rand_seed)
        prep_share = (verifiers_share, joint_rand_part)
        return (prep_state, prep_share)

    def prep_shares_to_prep(
            self,
            ctx: bytes,
            _agg_param: None,
            prep_shares: list[Prio3PrepShare[F]]) -> Optional[bytes]:
        # Unshard each set of verifier shares into each verifier message.
        verifiers = self.flp.field.zeros(
            self.flp.VERIFIER_LEN * self.PROOFS)
        joint_rand_parts = []
        for (verifiers_share, joint_rand_part) in prep_shares:
            verifiers = vec_add(verifiers, verifiers_share)
            if self.flp.JOINT_RAND_LEN > 0:
                assert joint_rand_part is not None
                joint_rand_parts.append(joint_rand_part)

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

        # Combine the joint randomness parts computed by the
        # Aggregators into the true joint randomness seed. This is
        # used in the last step.
        joint_rand_seed = None
        if self.flp.JOINT_RAND_LEN > 0:
            joint_rand_seed = self.joint_rand_seed(ctx, joint_rand_parts)
        return joint_rand_seed

    def prep_next(
        self,
        _ctx: bytes,
        prep_state: Prio3PrepState[F],
        prep_msg: Optional[bytes]
    ) -> tuple[Prio3PrepState[F], Prio3PrepShare[F]] | list[F]:
        joint_rand_seed = prep_msg
        (out_share, corrected_joint_rand_seed) = prep_state

        # If joint randomness was used, check that the value computed by
        # the Aggregators matches the value indicated by the Client.
        if joint_rand_seed != corrected_joint_rand_seed:
            raise ValueError('joint randomness check failed')
```

```
        return out_share
```

7.2.3. Validity of Aggregation Parameters

Prio3 only permits a report to be aggregated once.

```
    def is_valid(
            self,
            _agg_param: None,
            previous_agg_params: list[None]) -> bool:
        return len(previous_agg_params) == 0
```

7.2.4. Aggregation

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

```
    def agg_init(self, _agg_param: None) -> list[F]:
        return self.flp.field.zeros(self.flp.OUTPUT_LEN)

    def agg_update(self,
                   _agg_param: None,
                   agg_share: list[F],
                   out_share: list[F]) -> list[F]:
        return vec_add(agg_share, out_share)

    def merge(self,
              _agg_param: None,
              agg_shares: list[list[F]]) -> list[F]:
        agg = self.agg_init(None)
        for agg_share in agg_shares:
            agg = vec_add(agg, agg_share)
        return agg
```

7.2.5. Unsharding

To unshard a set of aggregate shares, the Collector first adds up the vectors element-wise, then decodes the aggregate result from the sum according to the FLP (Section 7.1.1).

```
    def unshard(
            self,
            _agg_param: None,
            agg_shares: list[list[F]],
            num_measurements: int) -> AggResult:
        agg = self.merge(None, agg_shares)
        return self.flp.decode(agg, num_measurements)
```

7.2.6. Auxiliary Functions

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

```
def helper_meas_share(
        self,
        ctx: bytes,
        agg_id: int,
        share: bytes) -> list[F]:
    return self.xof.expand_into_vec(
        self.flp.field,
        share,
        self.domain_separation_tag(USAGE_MEAS_SHARE, ctx),
        byte(agg_id),
        self.flp.MEAS_LEN,
    )

def helper_proofs_share(
        self,
        ctx: bytes,
        agg_id: int,
        share: bytes) -> list[F]:
    return self.xof.expand_into_vec(
        self.flp.field,
        share,
        self.domain_separation_tag(USAGE_PROOF_SHARE, ctx),
        byte(self.PROOFS) + byte(agg_id),
        self.flp.PROOF_LEN * self.PROOFS,
    )

def expand_input_share(
        self,
        ctx: bytes,
        agg_id: int,
        input_share: Prio3InputShare[F]) -> tuple[
            list[F],
            list[F],
            Optional[bytes]]:
    if agg_id > 0:
        assert len(input_share) == 2
        (share, blind) = input_share
        meas_share = self.helper_meas_share(ctx, agg_id, share)
        proofs_share = self.helper_proofs_share(ctx, agg_id, share)
    else:
        assert len(input_share) == 3
        (meas_share, proofs_share, blind) = input_share
    return (meas_share, proofs_share, blind)
```

```
   def prove_rands(self, ctx: bytes, prove_seed: bytes) -> list[F]:
       return self.xof.expand_into_vec(
           self.flp.field,
           prove_seed,
           self.domain_separation_tag(USAGE_PROVE_RANDOMNESS, ctx),
           byte(self.PROOFS),
           self.flp.PROVE_RAND_LEN * self.PROOFS,
       )

   def query_rands(
           self,
           verify_key: bytes,
           ctx: bytes,
           nonce: bytes) -> list[F]:
       return self.xof.expand_into_vec(
           self.flp.field,
           verify_key,
           self.domain_separation_tag(USAGE_QUERY_RANDOMNESS, ctx),
           byte(self.PROOFS) + nonce,
           self.flp.QUERY_RAND_LEN * self.PROOFS,
       )

   def joint_rand_part(
           self,
           ctx: bytes,
           agg_id: int,
           blind: bytes,
           meas_share: list[F],
           nonce: bytes) -> bytes:
       return self.xof.derive_seed(
           blind,
           self.domain_separation_tag(USAGE_JOINT_RAND_PART, ctx),
           byte(agg_id) + nonce + self.flp.field.encode_vec(meas_share),
       )

   def joint_rand_seed(self,
                       ctx: bytes,
                       joint_rand_parts: list[bytes]) -> bytes:
       """Derive the joint randomness seed from its parts."""
       return self.xof.derive_seed(
           zeros(self.xof.SEED_SIZE),
           self.domain_separation_tag(USAGE_JOINT_RAND_SEED, ctx),
           concat(joint_rand_parts),
       )

   def joint_rands(self,
                   ctx: bytes,
                   joint_rand_seed: bytes) -> list[F]:
```

```
        """Derive the joint randomness from its seed."""
        return self.xof.expand_into_vec(
            self.flp.field,
            joint_rand_seed,
            self.domain_separation_tag(USAGE_JOINT_RANDOMNESS, ctx),
            byte(self.PROOFS),
            self.flp.JOINT_RAND_LEN * self.PROOFS,
        )
```

7.2.7. Message Serialization

This section defines serialization formats for messages exchanged over the network while executing Prio3. Messages are defined in the presentation language of TLS as defined in Section 3 of [RFC8446].

Let prio3 denote an instance of Prio3. In the remainder, let S be an alias for prio3.xof.SEED_SIZE and F as an alias for prio3.field.ENCODED_SIZE. XOF seeds are represented as follows:

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

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

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

7.2.7.1. Public Share

The contents of the public share depend on whether joint randomness is required for the underlying FLP (i.e., prio3.flp.JOINT_RAND_LEN > 0). If joint randomness is not used, then the public share is the empty string. Otherwise, if joint randomness is used, then the public share encodes the joint randomness parts as follows:

```
struct {
    Prio3Seed joint_rand_parts[S * prio3.SHARES];
} Prio3PublicShareWithJointRand;
```

7.2.7.2. Input Share

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

In addition, the encoding of the input shares depends on which aggregator is receiving the message. If the aggregator ID is 0, then the input share includes the full measurement share and proofs(s)

share(s). Otherwise, if the aggregator ID is greater than 0, then the measurement and shares of proof(s) are represented by an XOF seed. Just as in Section 5.7.2, the former is called the Leader and the latter the Helpers.

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

```
struct {
    Prio3Field meas_share[F * prio3.flp.MEAS_LEN];
    Prio3Field proofs_share[F * prio3.flp.PROOF_LEN * prio3.PROOFS];
} Prio3LeaderShare;
```

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

```
struct {
    Prio3Seed share;
} Prio3HelperShare;
```

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

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

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

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

7.2.7.3. Prep Share

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

```
struct {
    Prio3Field verifiers_share[F * V];
} Prio3PrepShare;
```

where V = prio3.flp.VERIFIER_LEN * prio3.PROOFS. When joint randomness is used, the prep share includes the Aggregator's joint randomness part and is structured as follows:

```
struct {
    Prio3Field verifiers_share[F * V];
    Prio3Seed joint_rand_part;
} Prio3PrepShareWithJointRand;
```

7.2.7.4. Prep Message

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

```
struct {
    Prio3Seed joint_rand;
} Prio3PrepMessageWithJointRand;
```

7.2.7.5. Aggregation

Aggregate shares are structured as follows:

```
struct {
    Prio3Field agg_share[F * prio3.flp.OUTPUT_LEN];
} Prio3AggShare;
```

7.3. FLP Specification

This section specifies an implementation of the Flp interface (Section 7.1) based on the construction from [BBCGGI19], Section 4.2. The types and parameters required by this interface are listed in the table below.

Section 7.3.1 provides an overview of the proof system and some extensions to it. Section 7.3.2 defines validity circuits, the core component of the proof system that determines measurement validity and how measurements are aggregated. The proof-generation algorithm, query algorithm, and decision algorithm are defined in Section 7.3.3, Section 7.3.4, and Section 7.3.5 respectively.

Parameter	Value
valid	An instance of Valid (Section 7.3.2).
field	valid.field
PROVE_RAND_LEN	valid.prove_rand_len()
QUERY_RAND_LEN	valid.query_rand_len()
JOINT_RAND_LEN	valid.JOINT_RAND_LEN
MEAS_LEN	valid.MEAS_LEN
OUTPUT_LEN	valid.OUTPUT_LEN
PROOF_LEN	valid.proof_len()
VERIFIER_LEN	valid.verifier_len()
Measurement	As defined by valid.
AggResult	As defined by valid.

Table 8: FLP parameters for a validity circuit.

7.3.1. Overview

An FLP is a type of "zero-knowledge proof". A conventional zero-knowledge proof system involves two parties:

* The prover, who holds a measurement and generates a proof of the measurement’s validity

* The verifier who holds an encryption of, or commitment to, the measurement and checks the proof

The proof system here is much the same, except the verifier is split across multiple Aggregators, each of which has a secret share of the measurement rather than a commitment to it.

Validity is defined in terms of an arithmetic circuit evaluated over the measurement. The inputs to this circuit are elements of a finite field that comprise the encoded measurement; the gates of the circuit are multiplication, addition, and subtraction operations; and the output of the circuit is a single field element. If the value is zero, then the measurement is deemed valid; otherwise, if the output is non-zero, then the measurement is deemed invalid.

For example, the simplest circuit specified in this document is the following (Section 7.4.1):

```
C(x) = x * (x-1)
```

This circuit contains one subtraction gate (x-1) and one multiplication gate (x * (x-1)). Observe that C(x) = 0 if and only if x is in the range [0, 2).

The goal of the proof system is to allow each Aggregator to privately and correctly compute a share of C(x) from its share of x. Then all they need to do to determine validity is to broadcast their shares of C(x).

Suppose for a moment that C is an affine arithmetic circuit, meaning its only operations are addition, subtraction, and multiplication-by-constant. (The circuit above is non-affine because it contains a multiplication gate with two non-constant inputs.) Then each Aggregator can compute its share locally, since

```
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 is necessary to scale any addition of a constant in the circuit by 1/SHARES.) However, this is not the case if C contains multiplication gates with two non-constant inputs. Thus the goal is to transform these multiplication gates into computations on secret shared data that each Aggregator can perform locally.

The key idea is to have the prover construct a polynomial p such that p(j) is equal to the output of the j-th multiplication gate. Polynomial evaluation is fully linear, which means the coefficients of the polynomial can be secret shared in a way that allows each Aggregator to compute a share of p(j) for any j. These intermediate results can then be combined with the affine arithmetic operations of the validity circuit to produce the final output.

Applying this idea to the example circuit C above:

1. The Client, given its measurement x, constructs the lowest degree polynomial p for which p(0) = s and p(1) = x * (x-1), where s is a random blinding value generated by the Client. (The blinding value is to protect the privacy of the measurement.) It then sends shares of x and shares of the coefficients of p to each of the Aggregators.

2. Each Aggregator locally computes and broadcasts its share of p(1), which is equal to its share of C(x).

In fact, the FLP is slightly more general than this. One can replace the multiplication gate with any non-affine sub-circuit and apply the same idea. For example, in Section 7.4.2, the validity circuit uses the following sub-circuit multiple times:

Range2(x) = x * (x-1) = x**2 - x

(This is the same functionality computed by the example circuit C above.) Here again one can interpolate the lowest degree polynomial p for which p(j) is the value of the j-th call to Range2 in the validity circuit. Each validity circuit defines a sub-circuit that encapsulates its non-affine arithmetic operations. This sub-circuit is called the "gadget".

Finally, the proof system has one more important component. It is possible for a malicious Client to produce a gadget polynomial p that would result in C(x) being computed incorrectly, potentially resulting in an invalid measurement being accepted. To prevent this, the Aggregators perform a probabilistic test to check that the gadget polynomial was constructed properly. This "gadget test", and the procedure for constructing the polynomial, are described in detail in Section 7.3.3.

7.3.1.1. Extensions

The FLP described in Section 7.3 extends the proof system of [BBCGGI19], Section 4.2 in a few ways.

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

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

(Note that this is a special case of [BBCGGI19], Theorem 5.2.) Here x 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 x is valid, i.e., each x[j] is in the range [0, 2), then the circuit will evaluate to zero regardless of the value of r; but if some x[j] is not in the range [0, 2), then the output will be non-zero with high probability.

The second extension implemented by the FLP allows the validity circuit to contain multiple gadget types. (This generalization was suggested in [BBCGGI19], Remark 4.5.) This provides additional flexibility for designing circuits by allowing multiple, non-affine sub-circuits. For example, the following circuit is allowed:

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

where Range3(x) = x**3 - 3x**2 + 2x. This circuit checks that the first L inputs are in the range [0, 2) and the last N-L inputs are in the range [0, 3). The same circuit can be expressed using a simpler gadget, namely multiplication, but the resulting proof would be longer.

Third, rather than interpolate polynomials at inputs 1, 2, ..., j, where j is the j-th invocation of the gadget, roots of unity for the field are used. This allows constructing each polynomial via the number theoretic transform [SML24], which is far more efficient than generic formulas. Note that the roots of unity are powers of the generator for the NTT-friendly field (see Section 6.1.2).

Finally, the validity circuit in the FLP may have any number of outputs (at least one). The input is said to be valid if each of the outputs is zero. To save bandwidth, the FLP takes a random linear combination of the outputs. If each of the outputs is zero, then the reduced output will be zero; but if one of the outputs is non-zero, then the reduced output will be non-zero with high probability.

7.3.2. Validity Circuits

An instance of the proof system is defined in terms of a validity circuit that implements the Valid interface specified in this section. The parameters are listed in the table below.

Parameter	Description
GADGETS: list[Gadget]	A list of gadgets.
GADGET_CALLS: list[int]	Number of times each gadget is called.
MEAS_LEN: int	Length of the measurement.
JOINT_RAND_LEN: int	Length of the joint randomness.
EVAL_OUTPUT_LEN: int	Length of the circuit output.
OUTPUT_LEN: int	Length of the aggregatable output.
Measurement	Type of the measurement.
AggResult	Type of the aggregate result.
field: type[F]	Class object for the field (Section 6.1.2)

Table 9: Validity circuit parameters.

The circuit is invoked with the following method:

* valid.eval(meas: list[F], joint_rand: list[F], num_shares: int) -> list[F] evaluates the arithmetic circuit on a measurement and joint randomness. The output is a list of field elements: if every element is equal to valid.field(0), then the circuit is said to "accept" the measurement; otherwise, if any element is not equal to valid.field(0), then the circuit is said to "reject" the measurement.

This method can also be called on a secret share of the measurement, in which case it produces a secret share of the output.

The circuit must be composed of affine gates and gadget calls, so that the verifier may check the prover's proof and circuit evaluation using linear queries. This means that all non-affine multiplications in the circuit must be encapsulated in gadget calls. Additions of constants must be rescaled by the inverse of num_shares.

Pre-conditions:

- The length of meas MUST be valid.MEAS_LEN.
- The length of joint_rand MUST be valid.JOINT_RAND_LEN.
- num_shares MUST be the number of secret shares of meas, or 1 if meas is not secret shared.

Post-conditions:

- The length of the output MUST be valid.EVAL_OUTPUT_LEN.

Each circuit has a list of gadgets, denoted GADGETS, that are invoked by valid.eval(). The circuit evaluated by the gadget should be non-affine, and MUST be arithmetic, i.e., composed only of multiplication, addition, and subtraction gates. An instance of class Gadget has the following interface:

* ARITY: int is the number of input wires. For example, the multiplication gadget Mul(x,y) = x*y has arity of 2.
* DEGREE: int is the arithmetic degree of the gadget circuit. This is defined to be the degree of the polynomial that computes it. This exists because the circuit is arithmetic. For example, Mul has degree 2.
* gadget.eval(field: type[F], inp: list[F]) -> F evaluates the gadget over the given inputs and field.
* gadget.eval_poly(field: type[F], inp_poly: list[list[F]]) -> list[F] is the same as gadget.eval() except it evaluates the circuit over the polynomial ring of the field. This is well defined because the circuit is arithmetic.

In addition to the list of gadgets, the validity circuit specifies how many times each gadget is called (GADGET_CALLS). The circuit needs to define an ordering of the calls it makes to each gadget, so that all parties agree on how to identify recorded wire values. It also specifies the length of the circuit's input (MEAS_LEN), the length of the joint randomness (JOINT_RAND_LEN), and the length of the circuit's output (EVAL_OUTPUT_LEN).

A validity circuit also specifies parameters and methods needed for Prio3 aggregation. These are used to implement the interface in Section 7.1.1:

* valid.encode(measurement: Measurement) -> list[F] returns a vector of length MEAS_LEN representing a measurement of type Measurement.

* valid.truncate(meas: list[F]) -> list[F] returns a vector of length OUTPUT_LEN representing (a share of) an aggregatable output.

* valid.decode(agg: list[F], num_measurements: int) -> AggResult returns an aggregate result of type AggResult. This computation may depend on the number of outputs aggregated.

Finally, the following are helper methods used to instantiate parameters of the Flp interface (Section 7.1):

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

   def query_rand_len(self) -> int:
       """Length of the query randomness."""
       query_rand_len = len(self.GADGETS)
       if self.EVAL_OUTPUT_LEN > 1:
           query_rand_len += self.EVAL_OUTPUT_LEN
       return query_rand_len

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

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

7.3.3. Generating the Proof

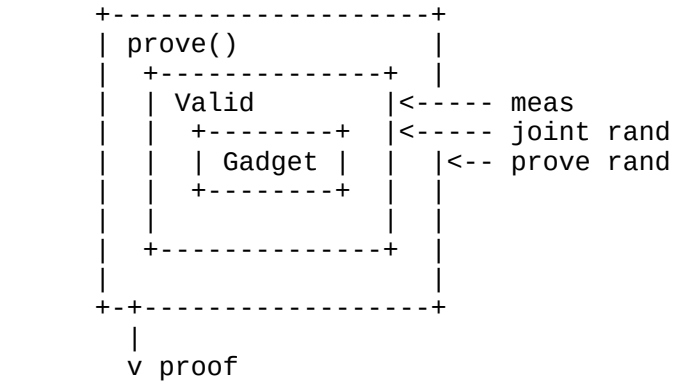

Figure 9: Components of the proof generation algorithm.

The proof generation algorithm invokes the validity circuit on the encoded measurement and joint randomness. The validity circuit in turn invokes the gadgets defined by the circuit. The prover records the values on input wires of gadget instances during circuit

evaluation, and constructs gadget polynomials that the verifier will use to compute the outputs of each gadget. Additionally, the prove randomness is used as a blinding factor when constructing gadget polynomials.

To generate the gadget polynomials, the prover evaluates the validity circuit, and records the values on each input wire of each call to each gadget. This is accomplished by "wrapping" each gadget in a class ProveGadget that records the wire inputs. This class is listed in Appendix A.4. Denote the value of the j-th wire for the k-th invocation of gadget g as g.wires[j][k].

Next, the prover computes each of the "wire polynomials" for each gadget. For each wire polynomial, take one prove randomness value and designate it the "wire seed" for that polynomial. The j-th wire polynomial is the lowest degree polynomial that evaluates to the "wire seed" at one point and g.wires[j][k] at a sequence of other points. The gadget polynomial is obtained by evaluating the gadget on the wire polynomials.

```
   def prove(self,
             meas: list[F],
             prove_rand: list[F],
             joint_rand: list[F]) -> list[F]:
        # Evaluate the validity circuit, recording the value of each
        # input wire for each evaluation of each gadget.
        valid = ProveGadget.wrap(self.valid, prove_rand)
        valid.eval(meas, joint_rand, 1)

        # Construct the proof, which consists of the wire seeds and
        # gadget polynomial for each gadget.
        proof = []
        for g in cast(list[ProveGadget[F]], valid.GADGETS):
            p = len(g.wires[0])

            # Compute the wire polynomials for this gadget. For each ‘j‘,
            # find the lowest degree polynomial ‘wire_poly‘ for which
            # ‘wire_poly(alpha**k) = g.wires[j][k]‘ for all ‘k‘. Note that
            # each ‘g.wires[j][0]‘ is set to the seed of wire ‘j‘, which
            # is included in the prove randomness.
            #
            # Implementation note: ‘alpha‘ is a root of unity, which
            # means ‘poly_interp()‘ can be evaluated using the NTT. Note
            # that ‘g.wires[j]‘ is padded with 0s to a power of 2.
            alpha = self.field.gen() ** (self.field.GEN_ORDER // p)
            wire_inp = [alpha ** k for k in range(p)]
            wire_polys = []
            for j in range(g.ARITY):
                wire_poly = poly_interp(self.field, wire_inp, g.wires[j])
                wire_polys.append(wire_poly)

            # Compute the gadget polynomial by evaluating the gadget on
            # the wire polynomials. By construction we have that
            # ‘gadget_poly(alpha**k)‘ is the ‘k‘-th output.
            gadget_poly = g.eval_poly(self.field, wire_polys)

            for j in range(g.ARITY):
                proof.append(g.wires[j][0])
            proof += gadget_poly

        return proof
```

7.3.4. Querying the Proof

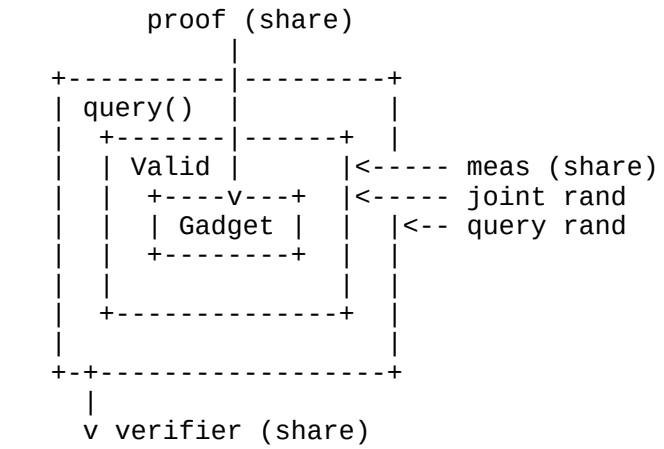

Figure 10: Components of the query algorithm.

The query algorithm invokes the validity circuit on the encoded measurement and joint randomness. It evaluates the gadget polynomials encoded by the proof (share) to produce (a share of) each gadget output. The verifier (share) consists of (a share of) the validity circuit's output and (a share of) each gadget test. The gadget tests consume the query randomness.

The goal of each gadget test is to ensure the inputs used by the prover to generate the gadget polynomial match the inputs used to evaluate it. This is done by partially reconstructing the gadget polynomial and evaluating it at a random point: when the gadget polynomial is evaluated at the same point, the result should be the same.

To start a gadget test, first construct the (shares of the) wire polynomials just as the prover did. Then record the input (share) of the j-th wire of the k-th invocation of the gadget as g.wires[j][k]. Again, this is accomplished by a wrapper gadget, QueryGadget, listed in Appendix A.4. This gadget also evaluates the gadget polynomial for each gadget invocation in order to produce the gadget's output. Then compute the wire polynomials from the recorded values.

Next, choose a random point t (parsed from the query randomness), evaluate each wire polynomial at t, and evaluate the gadget polynomial at t. The results are recorded in the verifier message passed to the decision algorithm, where the test is finished.

The random point t MUST NOT be one of the fixed evaluation points used to interpolate the wire polynomials. Otherwise, the verifier message may partially leak the encoded measurement.

```
   def query(self,
             meas: list[F],
             proof: list[F],
             query_rand: list[F],
             joint_rand: list[F],
             num_shares: int) -> list[F]:
        # Evaluate the validity circuit, recording the value of each
        # input wire for each evaluation of each gadget. Use the gadget
        # polynomials encoded by ‘proof‘ to compute the gadget outputs.
        valid = QueryGadget.wrap(self.valid, proof)
        out = valid.eval(meas, joint_rand, num_shares)

        # Reduce the output.
        if self.valid.EVAL_OUTPUT_LEN > 1:
            (rand, query_rand) = front(
                self.valid.EVAL_OUTPUT_LEN,
                query_rand,
            )
            v = self.field(0)
            for (r, out_elem) in zip(rand, out):
                v += r * out_elem
        else:
            [v] = out

        # Construct the verifier message, which consists of the reduced
        # circuit output and each gadget test.
        verifier = [v]
        for (g, t) in zip(cast(list[QueryGadget[F]], valid.GADGETS),
                          query_rand):
            p = len(g.wires[0])

            # Abort if ‘t‘ is one of the inputs used to compute the wire
            # polynomials so that the verifier message doesn’t leak the
            # gadget output. It suffices to check if ‘t‘ is a root of
            # unity, which implies it is a power of ‘alpha‘.
            if t ** p == self.field(1):
                raise ValueError(’test point is a root of unity’)

            # To test the gadget, we re-compute the wire polynomials and
            # check for consistency with the gadget polynomial provided
            # by the prover. To start, evaluate the gadget polynomial and
            # each of the wire polynomials at the random point ‘t‘.
            wire_checks = []
            wire_inp = [g.alpha ** k for k in range(p)]
            for j in range(g.ARITY):
                wire_poly = poly_interp(self.field, wire_inp, g.wires[j])
                wire_checks.append(poly_eval(self.field, wire_poly, t))
```

```
            gadget_check = poly_eval(self.field, g.poly, t)

            verifier += wire_checks
            verifier.append(gadget_check)

        return verifier
```

7.3.5. Deciding Validity

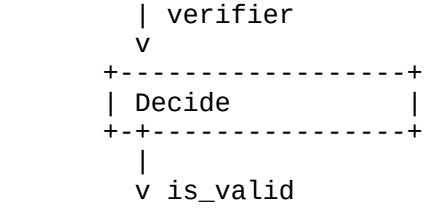

Figure 11: Components of the decision algorithm.

The decision algorithm consumes the verifier message. (Each of the Aggregators computes an additive share of the verifier message after the previous step.) The verifier message consists of the reduced circuit output and the gadget tests.

To finish each gadget test, evaluate the gadget on the wire checks: if the encoded measurement and joint randomness used to generate the proof are the same as the measurement (share) and joint randomness used to verify the proof, then the output of the gadget will be equal to the gadget check; otherwise, the output will not equal the gadget check with high probability.

```
def decide(self, verifier: list[F]) -> bool:
    # Check the output of the validity circuit.
    ([v], verifier) = front(1, verifier)
    if v != self.field(0):
        return False

    # Complete each gadget test.
    for g in self.valid.GADGETS:
        (wire_checks, verifier) = front(g.ARITY, verifier)
        ([gadget_check], verifier) = front(1, verifier)
        if g.eval(self.field, wire_checks) != gadget_check:
            return False

    return True
```

7.4. Variants

This section specifies instantiations of Prio3 for various aggregation tasks. Each variant is determined by a field (Section 6.1), a validity circuit (Section 7.3.2),and the number of proofs to generate and verify. All gadgets are listed in Appendix A. Test vectors for each can be found in Appendix C.

7.4.1. Prio3Count

Parameter	Value
field	Field64 (Table 4)
Valid	Count(field) (this section)
PROOFS	1

Table 10: Parameters for Prio3Count.

The first variant of Prio3 is for a simple counter: each measurement is either one or zero and the aggregate result is the sum of the measurements. Its validity circuit uses the multiplication gadget Mul specified in Appendix A.1, which takes two inputs and multiplies them. The circuit is specified below:

```
   class Count(Valid[int, int, F]):
       GADGETS: list[Gadget[F]] = [Mul()]
       GADGET_CALLS = [1]
       MEAS_LEN = 1
       JOINT_RAND_LEN = 0
       OUTPUT_LEN = 1
       EVAL_OUTPUT_LEN = 1

       # Class object for the field.
       field: type[F]

       def __init__(self, field: type[F]):
           self.field = field

       def encode(self, measurement: int) -> list[F]:
           return [self.field(measurement)]

       def eval(
               self,
               meas: list[F],
               joint_rand: list[F],
               _num_shares: int) -> list[F]:
           squared = self.GADGETS[0].eval(self.field,
                                          [meas[0], meas[0]])
           return [squared - meas[0]]

       def truncate(self, meas: list[F]) -> list[F]:
           return meas

       def decode(self, output: list[F], _num_measurements: int) -> int:
           return output[0].int()
```

7.4.2. Prio3Sum

Parameter	Value
field	Field64 (Table 4)
Valid	Sum(field, max_measurement) (this section)
PROOFS	1

Table 11: Parameters for Prio3Sum.

The next variant of Prio3 supports summing of integers in a pre-determined range. Each measurement is an integer in the range [0, max_measurement], where max_measurement defines the largest valid measurement.

The range check is accomplished by encoding the measurement as a bit vector, encoding the measurement plus an offset as a bit vector, then checking that the two encoded integers are consistent. Let

* bits = max_measurement.bit_length(), the number of bits needed to encode the largest valid measurement

* offset = 2 ** bits - 1 - max_measurement

The first bit-encoded integer is the measurement itself. Note that only measurements between 0 and 2**bits - 1 can be encoded this way with as many bits. The second bit-encoded integer is the sum of the measurement and offset. Observe that this sum can only be encoded this way if it is between 0 and 2**bits - 1, which implies that the measurement is between -offset and max_measurement.

The circuit first checks that each entry of both bit vectors is a one or a zero. It then decodes both the measurement and the offset measurement, and subtracts the offset from the latter. It then checks if these two values are equal. Since both the measurement and the measurement plus offset are in the same range of [0, 2**bits), this means that the measurement itself is between 0 and max_measurement.

The circuit uses the polynomial-evaluation gadget PolyEval specified in Appendix A.2. The polynomial is p(x) = x**2 - x, which is equal to 0 if and only if x is in the range [0, 2). The complete circuit is specified below:

Note that decoding a sequence of bits into an integer is a linear operation, specifically, a linear combination with a sequence of powers of two, so it can be done within a validity circuit using "free" affine gates. Furthermore, decoding secret shares of a bit-encoded integer will produce secret shares of the original integer.

```
class Sum(Valid[int, int, F]):
    GADGETS: list[Gadget[F]] = [PolyEval([0, -1, 1])]
    JOINT_RAND_LEN = 0
    OUTPUT_LEN = 1
    field: type[F]

    def __init__(self, field: type[F], max_measurement: int):
        self.field = field
```

```
            self.bits = max_measurement.bit_length()
            self.offset = self.field(2**self.bits - 1 - max_measurement)
            self.max_measurement = max_measurement
            self.GADGET_CALLS = [2 * self.bits]
            self.MEAS_LEN = 2 * self.bits
            self.EVAL_OUTPUT_LEN = 2 * self.bits + 1

        def encode(self, measurement: int) -> list[F]:
            encoded = []
            encoded += self.field.encode_into_bit_vec(
                measurement,
                self.bits
            )
            encoded += self.field.encode_into_bit_vec(
                measurement + self.offset.int(),
                self.bits
            )
            return encoded

        def eval(
                self,
                meas: list[F],
                joint_rand: list[F],
                num_shares: int) -> list[F]:
            shares_inv = self.field(num_shares).inv()

            out = []
            for b in meas:
                out.append(self.GADGETS[0].eval(self.field, [b]))

            range_check = self.offset * shares_inv + \
                self.field.decode_from_bit_vec(meas[:self.bits]) - \
                self.field.decode_from_bit_vec(meas[self.bits:])
            out.append(range_check)
            return out

        def truncate(self, meas: list[F]) -> list[F]:
            return [self.field.decode_from_bit_vec(meas[:self.bits])]

        def decode(self, output: list[F], _num_measurements: int) -> int:
            return output[0].int()
```

7.4.3. Prio3SumVec

Parameter	Value
field	Field128 (Table 4)
Valid	SumVec(field, length, bits, chunk_length) (this section)
PROOFS	1

Table 12: Parameters for Prio3SumVec.

This instance of Prio3 supports summing vectors of integers. It has three parameters: length, bits, and chunk_length. Each measurement is a vector of positive integers with length equal to the length parameter. Each element of the measurement is an integer in the range [0, 2**bits). It is RECOMMENDED to set chunk_length to an integer near the square root of length * bits (see Section 7.4.3.1).

The circuit is denoted SumVec. Each measurement is encoded as a vector of field elements with a length of length * bits. The field elements in the encoded vector represent all the bits of the measurement vector's elements, consecutively, in LSB to MSB order.

The validity circuit uses the ParallelSum gadget in Appendix A.3. This gadget applies an arithmetic subcircuit to multiple inputs in parallel, then sums the results. Along with the subcircuit, the parallel-sum gadget is parameterized by an integer, denoted count, specifying how many times to call the subcircuit. It takes in a list of inputs and passes them through to instances of the subcircuit in the same order. It returns the sum of the subcircuit outputs.

Note that only the ParallelSum gadget itself, and not its subcircuit, participates in the FLP's wire recording during evaluation, gadget consistency proofs, and proof validation, even though the subcircuit is provided to ParallelSum gadget as an implementation of the Gadget interface.

The SumVec validity circuit checks that the encoded measurement consists of ones and zeros. Rather than use the PolyEval gadget on each element, as in the Sum validity circuit, it instead uses Mul subcircuits (Appendix A.1) and "free" constant multiplication and addition gates to simultaneously evaluate the same range check polynomial on each element, and multiply by a constant. One of the two Mul subcircuit inputs is equal to a measurement element

multiplied by a power of one of the elements of the joint randomness vector, and the other is equal to the same measurement element minus one. These Mul subcircuits are evaluated by a ParallelSum gadget, and the results are added up both within the ParallelSum gadget and after it.

The complete circuit is specified below:

```
class SumVec(Valid[list[int], list[int], F]):
    EVAL_OUTPUT_LEN = 1
    length: int
    bits: int
    chunk_length: int
    field: type[F]

    def __init__(self,
                 field: type[F],
                 length: int,
                 bits: int,
                 chunk_length: int):
        """
        Instantiate the ‘SumVec‘ circuit for measurements with
        ‘length‘ elements, each in the range ‘[0, 2**bits)‘.
        """
        self.field = field
        self.length = length
        self.bits = bits
        self.chunk_length = chunk_length
        self.GADGETS = [ParallelSum(Mul(), chunk_length)]
        self.GADGET_CALLS = [
            (length * bits + chunk_length - 1) // chunk_length
        ]
        self.MEAS_LEN = length * bits
        self.OUTPUT_LEN = length
        self.JOINT_RAND_LEN = self.GADGET_CALLS[0]

    def encode(self, measurement: list[int]) -> list[F]:
        encoded = []
        for val in measurement:
            encoded += self.field.encode_into_bit_vec(
                val, self.bits)
        return encoded

    def eval(
            self,
            meas: list[F],
            joint_rand: list[F],
            num_shares: int) -> list[F]:
```

```
            out = self.field(0)
            shares_inv = self.field(num_shares).inv()
            for i in range(self.GADGET_CALLS[0]):
                r = joint_rand[i]
                r_power = r
                inputs: list[Optional[F]]
                inputs = [None] * (2 * self.chunk_length)
                for j in range(self.chunk_length):
                    index = i * self.chunk_length + j
                    if index < len(meas):
                        meas_elem = meas[index]
                    else:
                        meas_elem = self.field(0)

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

                    r_power *= r

                out += self.GADGETS[0].eval(
                    self.field,
                    cast(list[F], inputs),
                )

            return [out]

        def truncate(self, meas: list[F]) -> list[F]:
            truncated = []
            for i in range(self.length):
                truncated.append(self.field.decode_from_bit_vec(
                    meas[i * self.bits: (i + 1) * self.bits]
                ))
            return truncated

        def decode(
                self,
                output: list[F],
                _num_measurements: int) -> list[int]:
            return [x.int() for x in output]
```

7.4.3.1. Selection of ParallelSum Chunk Length

The chunk_length parameter provides a trade-off between the arity of the ParallelSum gadget (Appendix A.3) and the number of times the gadget is called. The proof length is asymptotically minimized when the chunk length is near the square root of the length of the encoded measurement. However, the relationship between VDAF parameters and proof length is complicated, involving two forms of rounding:

* The circuit pads the inputs to its last ParallelSum gadget call, up to the chunk length.

* The proof system rounds the degree of wire polynomials, determined by the number of times a gadget is called, up to the next power of two.

Therefore, the optimal choice of chunk_length for a concrete measurement size will vary, and must be found through trial and error. Setting chunk_length equal to the square root of the appropriate measurement length will result in proofs up to 50% larger than the optimal proof size.

7.4.4. Prio3Histogram

Parameter	Value
field	Field128 (Table 4)
Valid	Histogram(field, length, chunk_length) (this section)
PROOFS	1

Table 13: Parameters for Prio3Histogram.

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

The validity circuit is denoted Histogram. It has two parameters: length, the number of histogram buckets; and chunk_length, which is used by a circuit optimization described below. It is RECOMMENDED to set chunk_length to an integer near the square root of length (see Section 7.4.3.1).

The measurement is encoded as a one-hot vector representing the bucket into which the measurement falls. The circuit checks for one-hotness in two steps, by checking that the encoded measurement consists of ones and zeros, and by checking that the sum of all elements in the encoded measurement is equal to one. The individual checks constitute the output of the circuit.

As in the SumVec validity circuit (Section 7.4.3), the first part of the validity circuit uses the ParallelSum (Appendix A.3) gadget to perform range checks while achieving a smaller proof size. The ParallelSum gadget uses Mul subcircuits (Appendix A.1) to evaluate a range check polynomial on each element, and includes an additional constant multiplication. One of the two Mul subcircuit inputs is equal to a measurement element multiplied by a power of an element of the joint randomness vector, and the other is equal to the same measurement element minus one. The results are added up both within the ParallelSum gadget and after it.

```
class Histogram(Valid[int, list[int], F]):
    EVAL_OUTPUT_LEN = 2
    field: type[F]
    length: int
    chunk_length: int

    def __init__(self,
                 field: type[F],
                 length: int,
                 chunk_length: int):
        """
        Instantiate an instance of the ‘Histogram‘ circuit with the
        given ‘length‘ and ‘chunk_length‘.
        """
        self.field = field
        self.length = length
        self.chunk_length = chunk_length
        self.GADGETS = [ParallelSum(Mul(), chunk_length)]
        self.GADGET_CALLS = [
            (length + chunk_length - 1) // chunk_length]
        self.MEAS_LEN = self.length
        self.OUTPUT_LEN = self.length
        self.JOINT_RAND_LEN = self.GADGET_CALLS[0]

    def encode(self, measurement: int) -> list[F]:
        encoded = [self.field(0)] * self.length
        encoded[measurement] = self.field(1)
        return encoded

    def eval(
```

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

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

                    r_power *= r

                range_check += self.GADGETS[0].eval(
                    self.field,
                    cast(list[F], inputs),
                )

            # Check that the buckets sum to 1.
            sum_check = -shares_inv
            for b in meas:
                sum_check += b

            return [range_check, sum_check]

        def truncate(self, meas: list[F]) -> list[F]:
            return meas

        def decode(
                self,
                output: list[F],
                _num_measurements: int) -> list[int]:
            return [bucket_count.int()
                    for bucket_count in output]
```

7.4.5. Prio3MultihotCountVec

Parameter	Value
field	Field128 (Table 4)
Valid	MultihotCountVec(field, length, max_weight, chunk_length) (this section)
PROOFS	1

Table 14: Parameters for Prio3MultihotCountVec.

For this instance of Prio3, each measurement is a vector of Boolean values, where the number of True values is bounded. This provides a functionality similar to Prio3Histogram except that more than one entry (or none at all) may be non-zero. This allows Prio3MultihotCountVec to be composed with a randomized response mechanism, like [EPK14], for providing differential privacy. (For example, each Client would set each entry with some small probability.)

The validity circuit is denoted MultihotCountVec and has three parameters: length, the number of entries in the count vector; max_weight, the maximum number of True entries (i.e., the weight must be at most max_weight); and chunk_length, used the same way as in Section 7.4.3 and Section 7.4.4.

Validation works as follows. Let

* bits_for_weight = max_weight.bit_length()

* offset = 2 ** bits_for_weight - 1 - max_weight

The Client reports the weight of the count vector by adding offset to it and bit-encoding the result. Observe that only a weight of at most max_weight can be encoded with bits_for_weight bits.

The verifier checks that each entry of the encoded measurement is a bit (i.e., either one or zero). It then decodes the reported weight and subtracts it from offset + sum(count_vec), where count_vec is the count vector. The result is zero if and only if the reported weight is equal to the true weight. The two checks constitute the output of the circuit. The complete circuit is defined below.

```
   class MultihotCountVec(Valid[list[bool], list[int], F]):
       EVAL_OUTPUT_LEN = 2
       field: type[F]

       def __init__(self,
                    field: type[F],
                    length: int,
                    max_weight: int,
                    chunk_length: int):
           """
           Instantiate an instance of the this circuit with the given
           ‘length‘, ‘max_weight‘, and ‘chunk_length‘.

           Pre-conditions:

               - ‘length > 0‘
               - ‘0 < max_weight‘ and ‘max_weight <= length‘
               - ‘chunk_length > 0‘
           """
           self.field = field

           # Compute the number of bits to represent ‘max_weight‘.
           self.bits_for_weight = max_weight.bit_length()
           self.offset = self.field(
               2**self.bits_for_weight - 1 - max_weight)

           # Make sure ‘offset + length‘ doesn’t overflow the field
           # modulus. Otherwise we may not correctly compute the sum
           # measurement vector entries during circuit evaluation.
           if self.field.MODULUS - self.offset.int() <= length:
               raise ValueError(’length and max_weight are too large ’
                                ’for the current field size’)

           self.length = length
           self.max_weight = max_weight
           self.chunk_length = chunk_length
           self.GADGETS: list[Gadget[F]] = [
               ParallelSum(Mul(), chunk_length),
           ]
           self.GADGET_CALLS = [
               (length + self.bits_for_weight + chunk_length - 1)
               // chunk_length
           ]
           self.MEAS_LEN = self.length + self.bits_for_weight
           self.OUTPUT_LEN = self.length
           self.JOINT_RAND_LEN = self.GADGET_CALLS[0]

       def encode(self, measurement: list[bool]) -> list[F]:
```

```
            if len(measurement) != self.length:
                raise ValueError('invalid Client measurement length')

            # The first part is the vector of counters.
            count_vec = [self.field(int(x)) for x in measurement]

            # The second part is the reported weight.
            weight_reported = sum(count_vec, self.field(0))

            encoded = []
            encoded += count_vec
            encoded += self.field.encode_into_bit_vec(
                (self.offset + weight_reported).int(),
                self.bits_for_weight)
            return encoded

        def eval(
                self,
                meas: list[F],
                joint_rand: list[F],
                num_shares: int) -> list[F]:
            # Check that each entry in the input vector is one or zero.
            range_check = self.field(0)
            shares_inv = self.field(num_shares).inv()
            for i in range(self.GADGET_CALLS[0]):
                r = joint_rand[i]
                r_power = r
                inputs: list[Optional[F]]
                inputs = [None] * (2 * self.chunk_length)
                for j in range(self.chunk_length):
                    index = i * self.chunk_length + j
                    if index < len(meas):
                        meas_elem = meas[index]
                    else:
                        meas_elem = self.field(0)

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

                    r_power *= r

                range_check += self.GADGETS[0].eval(
                    self.field,
                    cast(list[F], inputs),
                )

            # Check that the weight 'offset' plus the sum of the counters
            # is equal to the value claimed by the Client.
```

```
            count_vec = meas[:self.length]
            weight = sum(count_vec, self.field(0))
            weight_reported = \
                self.field.decode_from_bit_vec(meas[self.length:])
            weight_check = self.offset*shares_inv + weight - \
                weight_reported

            return [range_check, weight_check]

        def truncate(self, meas: list[F]) -> list[F]:
            return meas[:self.length]

        def decode(
                self,
                output: list[F],
                _num_measurements: int) -> list[int]:
            return [bucket_count.int() for
                    bucket_count in output]
```

8. Poplar1

This section specifies Poplar1, a VDAF for the following task. Each Client holds a bit-string of length BITS and the Collector chooses a sequence of L-bit strings, where L <= BITS. The latter is referred to as the "candidate prefixes". The goal is to count how many of the Clients' inputs begin with each candidate prefix.

This functionality is the core component of the heavy hitters protocol of [BBCGGI21]. The goal of this protocol is to compute the subset of inputs held by at least T Clients for some threshold T. It invokes Poplar1 as follows:

1. Each Client shards its string into secret shares and uploads one share to each of the Aggregators.

2. The Collector picks an initial set of candidate prefixes, say 0 and 1, and sends them to the Aggregators.

3. The Aggregators run Poplar1 preparation and aggregation on each of the reports and send their aggregate shares to the Collector.

4. The Collector unshards the aggregate result, which consists of the hit count for each candidate prefix. For each prefix p with hit count at least T, the Collector adds p || 0 and p || 1 to the next generation of candidate prefixes and repeats Step 2.

While Poplar1 is intended to be used to compute heavy hitters in the above protocol, it may be possible to use it for other applications as well. However, care must be taken to ensure such usage is secure. See Section 9.4 and Section 9.5 for details.

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

An IDPF generalizes this "point" to a path on a full binary tree from the root to one of the leaves. It is evaluated on an "index" representing a unique node of the tree. If the node is on the programmed path, then the function evaluates to a non-zero value; otherwise it evaluates to zero. This structure allows an IDPF to provide the functionality required for the above protocol: to compute the hit count for an index, just evaluate each set of IDPF shares at that index and add up the results.

Consider the sub-tree constructed from a set of input strings and a target threshold T by including all indices with hit count at least T. This structure is called the "prefix tree" of the batch of measurements and target threshold. To compute the T-heavy-hitters for the batch, the Aggregators and Collector first compute the prefix tree, then extract the heavy hitters from the leaves of this tree. Note that the prefix tree leaks more information about the set than the heavy hitters themselves; see Section 9.4 for more discussion.

Poplar1 composes an IDPF with the arithmetic sketch of [BBCGGI21], Section 4.2. (The paper calls this a "secure sketch", but the underlying technique was later generalized in [BBCGGI23], where it is called "arithmetic sketching".) The sketch ensures that evaluating a set of input shares on a set of unique candidate prefixes results in shares of a zero vector or a "one-hot" vector, i.e., a vector that is zero everywhere except for in at most one position. Moreover, the value at that position should be one.

The remainder of this section is structured as follows. The syntax of IDPFs is defined in Section 8.1. The Poplar1 VDAF is defined in Section 8.2 in terms of a generic IDPF. A specification of the IDPF of [BBCGGI21] is given in Section 8.3. Test vectors for Poplar1 can be found in Appendix C.

8.1. Incremental Distributed Point Functions (IDPFs)

An IDPF is defined over a domain of size 2**BITS, where BITS is a constant. Indices into the IDPF tree are bit strings. (In Poplar1, each Client's bit string is an index; see Section 8.1.1 for details.) The Client specifies an index alpha and a vector of values beta, one for each "level" L in the range [0, BITS). The key generation algorithm generates one IDPF "key" for each Aggregator. When evaluated at level L and index prefix, each IDPF key returns an additive share of beta[L] if prefix is the L-bit prefix of alpha and shares of zero otherwise.

Each of the programmed points beta is a vector of elements of some finite field. There are two types of fields: one for inner nodes (denoted FieldInner), and one for leaf nodes (FieldLeaf). (The instantiation of Poplar1 (Section 8.2) will use a much larger field for leaf nodes than for inner nodes. This is to ensure the IDPF is "extractable" as defined in [BBCGGI21], Definition 1. See Section 9.5 for details.)

A concrete IDPF defines the types and parameters enumerated in Table 15. In the remainder, Output is used as shorthand for the type list[list[FieldInner]] | list[list[FieldLeaf]]. (This type denotes either a vector of inner node field elements or leaf node field elements.) The scheme is comprised of the following algorithms:

* idpf.gen(alpha: tuple[bool, ...], beta_inner: list[list[FieldInner]], beta_leaf: list[FieldLeaf], ctx: bytes, nonce: bytes, rand: bytes) -> tuple[PublicShare, list[bytes]] is the IDPF-key generation algorithm. Its inputs are the index alpha, the values beta, the application context, and the report nonce.

  The output is a public part (of type PublicShare) that is sent to each Aggregator and a vector of private IDPF keys, one for each Aggregator. The nonce and application context are used to derive the fixed AES key for XofFixedKeyAes128 (Section 6.2.2). Looking ahead, this key is used for extending a node's seed into the seeds for the child nodes at each level of the tree; see Section 8.3.

  Pre-conditions:

  - alpha MUST have length BITS.

  - beta_inner MUST have length BITS - 1.

  - beta_inner[level] MUST have length VALUE_LEN for each level in the range [0, BITS - 1).

- beta_leaf MUST have length VALUE_LEN.

- rand MUST be generated by a CSPRNG and have length RAND_SIZE.

- nonce MUST be generated by a CSPRNG (see Section 9.2 for details) and have length idpf.NONCE_SIZE.

Post-conditions:

- The number of IDPF keys MUST be idpf.SHARES.

* idpf.eval(agg_id: int, public_share: PublicShare, key: bytes, level: int, prefixes: Sequence[tuple[bool, ...]], ctx: bytes, nonce: bytes) -> Output is the IDPF-key evaluation algorithm run by each Aggregator. Its inputs are the Aggregator's unique identifier, the public share distributed to all of the Aggregators, the Aggregator's IDPF key, the "level" at which to evaluate the IDPF, the sequence of candidate prefixes, the application context, and the report nonce. It returns the share of the value corresponding to each candidate prefix.

  Pre-conditions:

  - agg_id MUST be in the range [0, idpf.SHARES) and match the index of key in the sequence of IDPF keys output by the Client.

  - level MUST be in the range [0, BITS).

  - Each prefix MUST be distinct and have length level + 1.

  - The length of the nonce MUST be idpf.NONCE_SIZE.

  Post-conditions:

  - The length of the output MUST be len(prefixes)

  - The length of each element of the output MUST be idpf.VALUE_LEN

  - If level == idpf.BITS - 1, then the output field MUST be FieldLeaf and FieldInner otherwise

In addition, the IDPF provides the following method:

```
def current_field(
        self,
        level: int) -> type[FieldInner] | type[FieldLeaf]:
    if level < self.BITS - 1:
        return self.field_inner
    return self.field_leaf
```

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 VDAF evaluations. In practice, of course, it will often be beneficial to expose a stateful API for IDPFs and carry the state across evaluations. See Section 8.3 for details.

Parameter	Description
SHARES: int	Number of IDPF keys output by IDPF-key generator.
BITS: int	Length in bits of each input string.
VALUE_LEN: int	Number of field elements of each output value.
RAND_SIZE: int	Size of the random string consumed by the IDPF-key generator.
NONCE_SIZE: int	Size of the random nonce generated by the Client.
KEY_SIZE: int	Size in bytes of each IDPF key.
FieldInner	Implementation of Field (Section 6.1) used for values of inner nodes.
FieldLeaf	Implementation of Field used for values of leaf nodes.
PublicShare	Type of public share for this IDPF.
Output	Alias of list[list[FieldInner]] | list[list[FieldLeaf]].
FieldVec	Alias of list[FieldInner] | list[FieldLeaf].

Table 15: Constants and types defined by a concrete IDPF.

8.1.1. Encoding Inputs as Indices

How data are represented as IDPF indices is up to the application. When the inputs are fixed-length byte strings, the most natural choice of representation is as a bit string formed from all the bits of the byte string, first ordered by byte position, then ordered from most significant bit to least significant bit within each byte. This ensures that, when a byte string is a prefix of another, so too is its corresponding index. (Index prefixes are defined in Section 8.1.) For example,

Byte string: 01 02
Bit string: 00000001 00000010

is a prefix of

Byte string: 01 02 03
Bit string: 00000001 00000010 00000011

Additionally, lexicographic ordering is preserved by this mapping from a byte string to a bit string.

When the inputs are variable length, it is necessary to pad each input to some fixed length. Further, the padding scheme must be non-ambiguous. For example, each input could be padded with b"\x01" followed by as many b"\x00" bytes as needed.

8.2. Specification

This section specifies Poplar1, an implementation of the Vdaf interface (Section 5). It is defined in terms of the Idpf implementation of Section 8.3 with SHARES == 2 and VALUE_LEN == 2 and XofTurboShake128 as specified in Section 6.2.1. The associated constants and types required by the Vdaf interface are defined in Table 16. The methods required for sharding, preparation, aggregation, and unsharding are described in the remaining subsections. These methods make use of constants defined in Table 17.

Parameter	Value
idpf	As specified in Section 8.3.
xof	XofTurboShake128 (Section 6.2.1)
VERIFY_KEY_SIZE	xof.SEED_SIZE
RAND_SIZE	xof.SEED_SIZE * 3 + idpf.RAND_SIZE
NONCE_SIZE	16
ROUNDS	2
SHARES	2
Measurement	tuple[bool, ...]
AggParam	tuple[int, Sequence[tuple[bool, ...]]]
PublicShare	As defined by idpf.
InputShare	tuple[bytes, bytes, list[FieldInner], list[FieldLeaf]]
OutShare	FieldVec
AggShare	FieldVec
AggResult	list[int]
PrepState	tuple[bytes, int, FieldVec]
PrepShare	FieldVec
PrepMessage	Optional[FieldVec]

Table 16: VDAF parameters for Poplar1.

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

Table 17: Constants used by Poplar1.

8.2.1. Sharding

The Client's measurement is an IDPF index, denoted alpha, whose type is a sequence of bits tuple[bool, ...] (See Section 8.1.1 for guidelines on index encoding.)

The programmed IDPF values are pairs of field elements (1, k) where each k is chosen at random. This random value is used as part of the arithmetic sketching protocol of [BBCGGI21], Appendix C.4. After evaluating their IDPF key shares on a given sequence of candidate prefixes, the Aggregators use the sketching protocol to verify that they hold shares of a zero vector or a one-hot vector at a given level of the IDPF tree.

In addition to programming k into the IDPF output, for each level of the tree, the Client generates random elements a, b, and c and computes

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

and sends additive shares of a, b, c, A and B to each of the Aggregators. These help the Aggregators evaluate the sketch during preparation.

Putting everything together, the sharding algorithm is defined as follows.

```
   def shard(
       self,
       ctx: bytes,
       measurement: tuple[bool, ...],
       nonce: bytes,
       rand: bytes,
   ) -> tuple[Poplar1PublicShare, list[Poplar1InputShare]]:
       if len(nonce) != self.NONCE_SIZE:
           raise ValueError("incorrect nonce size")
       if len(rand) != self.RAND_SIZE:
           raise ValueError("incorrect size of random bytes argument")

       l = self.xof.SEED_SIZE

       # Split the random input into the random input for IDPF key
       # generation, correlated randomness, and sharding.
       if len(rand) != self.RAND_SIZE:
           raise ValueError('incorrect rand size')
       idpf_rand, rand = front(self.idpf.RAND_SIZE, rand)
       seeds = [rand[i:i + l] for i in range(0, 3 * l, l)]
       corr_seed, seeds = front(2, seeds)
       (shard_seed,), seeds = front(1, seeds)

       xof = self.xof(
           shard_seed,
           self.domain_separation_tag(USAGE_SHARD_RAND, ctx),
           nonce,
       )

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

       # Generate the IDPF keys.
       (public_share, keys) = self.idpf.gen(
           measurement,
           beta_inner,
           beta_leaf,
           ctx,
```

```
            nonce,
            idpf_rand,
        )

        # Generate correlated randomness used by the Aggregators to
        # evaluate the sketch over their output shares. Seeds are used
        # to encode shares of the ‘(a, b, c)‘ triples. (See [BBCGGI21,
        # Appendix C.4].)
        corr_offsets: list[Field] = vec_add(
            self.xof.expand_into_vec(
                self.idpf.field_inner,
                corr_seed[0],
                self.domain_separation_tag(USAGE_CORR_INNER, ctx),
                byte(0) + nonce,
                3 * (self.idpf.BITS - 1),
            ),
            self.xof.expand_into_vec(
                self.idpf.field_inner,
                corr_seed[1],
                self.domain_separation_tag(USAGE_CORR_INNER, ctx),
                byte(1) + nonce,
                3 * (self.idpf.BITS - 1),
            ),
        )
        corr_offsets += vec_add(
            self.xof.expand_into_vec(
                self.idpf.field_leaf,
                corr_seed[0],
                self.domain_separation_tag(USAGE_CORR_LEAF, ctx),
                byte(0) + nonce,
                3,
            ),
            self.xof.expand_into_vec(
                self.idpf.field_leaf,
                corr_seed[1],
                self.domain_separation_tag(USAGE_CORR_LEAF, ctx),
                byte(1) + nonce,
                3,
            ),
        )

        # For each level of the IDPF tree, shares of the ‘(A, B)‘
        # pairs are computed from the corresponding ‘(a, b, c)‘
        # triple and authenticator value ‘k‘.
        corr_inner: list[list[Field64]] = [[], []]
        for level in range(self.idpf.BITS):
            field = cast(type[Field], self.idpf.current_field(level))
            k = beta_inner[level][1] if level < self.idpf.BITS - 1 \
```

```
            else beta_leaf[1]
        (a, b, c), corr_offsets = corr_offsets[:3], corr_offsets[3:]
        A = -field(2) * a + k
        B = a ** 2 + b - a * k + c
        corr1 = xof.next_vec(field, 2)
        corr0 = vec_sub([A, B], corr1)
        if level < self.idpf.BITS - 1:
            corr_inner[0] += cast(list[Field64], corr0)
            corr_inner[1] += cast(list[Field64], corr1)
        else:
            corr_leaf = [
                cast(list[Field255], corr0),
                cast(list[Field255], corr1),
            ]

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

8.2.2. Preparation

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

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

```
def prep_init(
        self,
        verify_key: bytes,
        ctx: bytes,
        agg_id: int,
        agg_param: Poplar1AggParam,
        nonce: bytes,
        public_share: Poplar1PublicShare,
        input_share: Poplar1InputShare) -> tuple[
```

```
            Poplar1PrepState,
            FieldVec]:
        (level, prefixes) = agg_param
        (key, corr_seed, corr_inner, corr_leaf) = input_share
        field = self.idpf.current_field(level)

        # Evaluate the IDPF key at the given set of prefixes.
        value = self.idpf.eval(
            agg_id, public_share, key, level, prefixes, ctx, nonce)

        # Get shares of the correlated randomness for evaluating the
        # Aggregator's share of the sketch.
        if level < self.idpf.BITS - 1:
            corr_xof = self.xof(
                corr_seed,
                self.domain_separation_tag(USAGE_CORR_INNER, ctx),
                byte(agg_id) + nonce,
            )
            # Fast-forward the XOF state to the current level.
            corr_xof.next_vec(field, 3 * level)
        else:
            corr_xof = self.xof(
                corr_seed,
                self.domain_separation_tag(USAGE_CORR_LEAF, ctx),
                byte(agg_id) + nonce,
            )
        (a_share, b_share, c_share) = corr_xof.next_vec(field, 3)
        if level < self.idpf.BITS - 1:
            (A_share, B_share) = cast(
                list[Field],
                corr_inner[2 * level:2 * (level + 1)],
            )
        else:
            (A_share, B_share) = cast(list[Field], corr_leaf)

        # Evaluate the Aggregator's share of the sketch. These are
        # called the "masked input values" [BBCGGI21, Appendix C.4].
        verify_rand_xof = self.xof(
            verify_key,
            self.domain_separation_tag(USAGE_VERIFY_RAND, ctx),
            nonce + to_be_bytes(level, 2),
        )
        verify_rand = cast(
            list[Field],
            verify_rand_xof.next_vec(field, len(prefixes)),
        )
        sketch_share = cast(
            list[Field],
```

```
            [a_share, b_share, c_share],
        )
        out_share = []
        for (i, r) in enumerate(verify_rand):
            data_share = cast(Field, value[i][0])
            auth_share = cast(Field, value[i][1])
            sketch_share[0] += data_share * r
            sketch_share[1] += data_share * r ** 2
            sketch_share[2] += auth_share * r
            out_share.append(data_share)

        prep_mem = [A_share, B_share, field(agg_id)] + out_share
        return (
            (
                b'evaluate sketch',
                level,
                cast(FieldVec, prep_mem),
            ),
            cast(FieldVec, sketch_share),
        )

    def prep_next(
        self,
        _ctx: bytes,
        prep_state: Poplar1PrepState,
        prep_msg: Optional[FieldVec]
    ) -> tuple[Poplar1PrepState, FieldVec] | FieldVec:
        prev_sketch = cast(list[Field], prep_msg)
        (step, level, prep_mem) = prep_state

        if step == b'evaluate sketch':
            if prev_sketch is None:
                raise ValueError('expected value, got none')
            elif len(prev_sketch) != 3:
                raise ValueError('incorrect sketch length')
            A_share = cast(Field, prep_mem[0])
            B_share = cast(Field, prep_mem[1])
            agg_id = cast(Field, prep_mem[2])
            prep_mem = prep_mem[3:]
            sketch_share = [
                agg_id * (prev_sketch[0] ** 2
                          - prev_sketch[1]
                          - prev_sketch[2])
                + A_share * prev_sketch[0]
                + B_share
            ]
            return cast(
                tuple[Poplar1PrepState, FieldVec],
```

```
                (
                    (
                        b'reveal sketch',
                        level,
                        prep_mem,
                    ),
                    sketch_share,
                )
            )

        elif step == b'reveal sketch':
            if prev_sketch is None:
                return prep_mem  # Output shares
            else:
                raise ValueError('invalid prep message')

        raise ValueError('invalid prep state')

    def prep_shares_to_prep(
            self,
            _ctx: bytes,
            agg_param: Poplar1AggParam,
            prep_shares: list[FieldVec]) -> Optional[FieldVec]:
        if len(prep_shares) != 2:
            raise ValueError('incorrect number of prep shares')
        (level, _) = agg_param
        field = self.idpf.current_field(level)
        sketch = vec_add(
            cast(list[Field], prep_shares[0]),
            cast(list[Field], prep_shares[1]),
        )
        if len(sketch) == 3:
            return cast(FieldVec, sketch)
        elif len(sketch) == 1:
            if sketch == field.zeros(1):
                # In order to reduce communication overhead, let 'None'
                # denote a successful sketch verification.
                return None
            else:
                raise ValueError('sketch verification failed')
        else:
            raise ValueError('incorrect sketch length')
```

8.2.3. Validity of Aggregation Parameters

Aggregation parameters are valid for a given input share if no aggregation parameter with the same level has been used with the same input share before. The whole preparation phase MUST NOT be run more than once for a given combination of input share and level. This function checks that candidate prefixes are unique and lexicographically sorted, checks that levels are increasing between calls, and also enforces that the prefixes at each level are suffixes of the previous level’s prefixes.

```
   def is_valid(
           self,
           agg_param: Poplar1AggParam,
           previous_agg_params: list[Poplar1AggParam]) -> bool:
       """
       Checks that candidate prefixes are unique and lexicographically
       sorted, checks that levels are increasing between calls, and also
       enforces that the prefixes at each level are suffixes of the
       previous level's prefixes.
       """
       (level, prefixes) = agg_param

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

       if len(previous_agg_params) < 1:
           return True

       (last_level, last_prefixes) = previous_agg_params[-1]
       last_prefixes_set = set(last_prefixes)

       # Check that level increased.
       if level <= last_level:
           return False

       # Check that prefixes are suffixes of the last level's prefixes.
       for prefix in prefixes:
           last_prefix = get_ancestor(prefix, last_level)
           if last_prefix not in last_prefixes_set:
               # Current prefix not a suffix of last level's prefixes.
               return False
       return True

   def get_ancestor(
           index: tuple[bool, ...],
           level: int) -> tuple[bool, ...]:
       """
       Helper function to determine the prefix of 'index' at
       'level'.
       """
       return index[:level + 1]
```

8.2.4. Aggregation

Aggregation involves simply adding up the output shares.

```
def agg_init(self, agg_param: Poplar1AggParam) -> FieldVec:
    (level, prefixes) = agg_param
    field = self.idpf.current_field(level)
    return field.zeros(len(prefixes))

def agg_update(self,
               agg_param: Poplar1AggParam,
               agg_share: FieldVec,
               out_share: FieldVec) -> FieldVec:
    a = cast(list[Field], agg_share)
    o = cast(list[Field], out_share)
    return cast(FieldVec, vec_add(a, o))

def merge(self,
          agg_param: Poplar1AggParam,
          agg_shares: list[FieldVec]) -> FieldVec:
    (level, prefixes) = agg_param
    field = self.idpf.current_field(level)
    agg = cast(list[Field], field.zeros(len(prefixes)))
    for agg_share in agg_shares:
        agg = vec_add(agg, cast(list[Field], agg_share))
    return cast(FieldVec, agg)
```

8.2.5. Unsharding

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

```
def unshard(
        self,
        agg_param: Poplar1AggParam,
        agg_shares: list[FieldVec],
        _num_measurements: int) -> list[int]:
    agg = self.merge(agg_param, agg_shares)
    return [x.int() for x in agg]
```

8.2.6. Message Serialization

This section defines serialization formats for messages exchanged over the network while executing Poplar1. Messages are defined in the presentation language of TLS as defined in Section 3 of [RFC8446].

Let poplar1 be an instance of Poplar1. In the remainder let Fi be an alias for poplar1.idpf.field_inner.ENCODED_SIZE, Fl as an alias for poplar1.idpf.field_leaf.ENCODED_SIZE, and B as an alias for poplar1.idpf.BITS.

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

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

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

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

8.2.6.1. Public Share

The public share of the IDPF scheme in Section 8.3 consists of a sequence of "correction words". A correction word has three components:

1. the XOF seed of type bytes;

2. the control bits of type tuple[bool, bool]; and

3. the payload of type list[Field64] for the first BITS-1 words and list[Field255] for the last word.

The encoding is a straightforward structure of arrays, except that the control bits are packed as tightly as possible. The encoded public share is structured as follows:

```
struct {
    opaque packed_control_bits[packed_len];
    opaque seed[poplar1.idpf.KEY_SIZE*B];
    Poplar1FieldInner payload_inner[Fi*poplar1.idpf.VALUE_LEN*(B-1)];
    Poplar1FieldLeaf payload_leaf[Fl*poplar1.idpf.VALUE_LEN];
} Poplar1PublicShare;
```

Here packed_len = (2*B + 7) // 8 is the length of the packed control bits. Field packed_control_bits is encoded with the following function:

```
packed_control_buf = [int(0)] * packed_len
for i, bit in enumerate(control_bits):
    packed_control_buf[i // 8] |= bit << (i % 8)
packed_control_bits = bytes(packed_control_buf)
```

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

```
control_bits = []
for i in range(length):
    control_bits.append(bool(
        (packed_control_bits[i // 8] >> (i % 8)) & 1
    ))
leftover_bits = packed_control_bits[-1] >> (
    (length + 7) % 8 + 1
)
if (length + 7) // 8 != len(packed_control_bits) or \
        leftover_bits != 0:
    raise ValueError('trailing bits')
```

8.2.6.2. Input Share

Each input share is structured as follows:

```
struct {
    opaque idpf_key[poplar1.idpf.KEY_SIZE];
    opaque corr_seed[poplar1.xof.SEED_SIZE];
    Poplar1FieldInner corr_inner[Fi * 2 * (B- 1)];
    Poplar1FieldLeaf corr_leaf[Fl * 2];
} Poplar1InputShare;
```

8.2.6.3. Prep Share

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

For the first round and inner field:

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

For the first round and leaf field:

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

For the second round and inner field:

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

For the second round and leaf field:

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

8.2.6.4. Prep Message

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

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

For the first round and leaf field:

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

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

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

8.2.6.5. Aggregate Share

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

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

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

For the leaf field:

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

8.2.6.6. Aggregation Parameter

The aggregation parameter is encoded as follows:

```
struct {
    uint16_t level;
    uint32_t num_prefixes;
    opaque encoded_prefixes[prefixes_len];
} Poplar1AggParam;
```

The fields in this struct are: level, the level of the IDPF tree of each prefixes; num_prefixes, the number of prefixes to evaluate; and encoded_prefixes, the sequence of prefixes encoded into a byte string of length prefixes_len. Each prefix is packed into a byte string, with the bits assigned in MSB-to-LSB order, and then the byte strings for each prefix are concatenated together. The prefixes are encoded with the following procedure:

```
prefixes_len = ((level + 1) + 7) // 8 * len(prefixes)
encoded_prefixes = bytearray()
for prefix in prefixes:
    for chunk in itertools.batched(prefix, 8):
        byte_out = 0
        for (bit_position, bit) in enumerate(chunk):
            byte_out |= bit << (7 - bit_position)
        encoded_prefixes.append(byte_out)
```

Decoding involves the following procedure:

```
   prefixes = []

   last_byte_mask = 0
   leftover_bits = (level + 1) % 8
   if leftover_bits > 0:
       for bit_index in range(8 - leftover_bits, 8):
           last_byte_mask |= 1 << bit_index
       last_byte_mask ^= 255

   bytes_per_prefix = ((level + 1) + 7) // 8
   for chunk in itertools.batched(encoded_prefixes, bytes_per_prefix):
       if chunk[-1] & last_byte_mask > 0:
           raise ValueError('trailing bits in prefix')

       prefix = []
       for i in range(level + 1):
           byte_index = i // 8
           bit_offset = 7 - (i % 8)
           bit = (chunk[byte_index] >> bit_offset) & 1 != 0
           prefix.append(bit)
       prefixes.append(tuple(prefix))
```

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

8.3. IDPF Specification

This section specifies a concrete IDPF suitable for instantiating Poplar1. The constant and type definitions required by the Idpf interface are given in Table 18.

The IDPF requires an XOF for deriving the output shares, as well as a variety of other artifacts used internally. For performance reasons, this object is instantiated using XofFixedKeyAes128 (Section 6.2.2) wherever possible. See Section 9.6 for security considerations.

Parameter	Value
xof	XofFixedKeyAes128 (Section 6.2.2)
SHARES	2
BITS	Any positive integer.
VALUE_LEN	Any positive integer.
KEY_SIZE	xof.SEED_SIZE
FieldInner	Field64 (Table 4)
FieldLeaf	Field255 (Table 4)

Table 18: Constants and type definitions for the concrete IDPF.

8.3.1. Overview

At a high level, the IDPF maps a key generator's input (alpha, beta_1, ..., beta_BITS) onto a binary tree with BITS+1 levels, where each edge going from a parent node to a left child is labeled 0, and each right edge is labeled 1. Then each leaf node corresponds to a bit string of length BITS, where the labels on the path from the root to x contain the individual bits. Finally, all nodes in the tree have an assigned value, with the nodes on the path from the root to alpha having values beta_1, ..., beta_BITS, and all other nodes having value 0.

The IDPF construction now boils down to secret-sharing the values at each node of that tree in an efficient way. Note that explicitly representing the tree requires O(2**BITS) space, so the generator cannot just compute additive shares of it and send them to the two evaluators. Instead, the evaluators will re-generate shares of values at selected nodes of the tree using a XOF (Section 6.2).

The basic observation is that if both evaluators have the same seed s of length KEY_SIZE, then expanding s using a XOF will also result in the same expansion. If the length of the XOF expansion is set to 2*KEY_SIZE, it can then be split again into two seeds s_l, s_r, that can again serve as XOF seeds. Now, viewing the seeds as XOR-shares of integers, if evaluators have the same seed at the root of the tree, then their expanded trees will form a secret-shared tree of zeros. The actual construction will additionally use a convert()

function before each expansion, which maps seeds into the appropriate output domain (see Section 8.3.4), generating a new seed for the next level in the process.

The open task now is to ensure that evaluators have different seeds at nodes that lie on the path to alpha, while having the same seeds on all other nodes. This is done using so-called "correction words" included in the public share. The correction words are conditionally added to the XOF output by both evaluators. The condition here is a secret-shared bit, called a "control bit", which indicates whether the current node is on the path to alpha or not. On the path, the control bits add up to 1, meaning only one evaluator will add the correction word to its XOF output. Off the path, either none or both evaluators add the correction word, and so the seeds at the next level stay the same.

What remains is to turn the (now pseudorandom) values on the path to alpha into the desired beta values. This is done by including "value correction words" in the public share, which are chosen such that when added with the pseudorandom shares at the ith node on the path to alpha, they add up to shares of beta_i.

The following two sections describe the algorithms for key generation in full detail.

8.3.2. Key Generation

The description of the IDPF-key generation algorithm makes use of auxiliary functions extend() and convert() defined in Section 8.3.4.

```
def gen(
        self,
        alpha: tuple[bool, ...],
        beta_inner: list[list[Field64]],
        beta_leaf: list[Field255],
        ctx: bytes,
        nonce: bytes,
        rand: bytes) -> tuple[list[CorrectionWord], list[bytes]]:
    if len(alpha) != self.BITS:
        raise ValueError("incorrect alpha length")
    if len(beta_inner) != self.BITS - 1:
        raise ValueError("incorrect beta_inner length")
    if len(rand) != self.RAND_SIZE:
        raise ValueError("incorrect rand size")
    if len(nonce) != self.NONCE_SIZE:
        raise ValueError("incorrect nonce size")

    key = [
```

```
            rand[:XofFixedKeyAes128.SEED_SIZE],
            rand[XofFixedKeyAes128.SEED_SIZE:],
        ]

        seed = key.copy()
        ctrl = [False, True]
        public_share = []
        for level in range(self.BITS):
            bit = alpha[level]
            keep = int(bit)
            lose = 1 - keep

            (s0, t0) = self.extend(level, seed[0], ctx, nonce)
            (s1, t1) = self.extend(level, seed[1], ctx, nonce)
            seed_cw = xor(s0[lose], s1[lose])
            ctrl_cw = (
                t0[0] ^ t1[0] ^ (not bit),
                t0[1] ^ t1[1] ^ bit,
            )

            # Implementation note: these conditional XORs and
            # input-dependent array indices should be replaced with
            # constant-time selects in practice in order to reduce
            # leakage via timing side channels.
            if ctrl[0]:
                x0 = xor(s0[keep], seed_cw)
                ctrl[0] = t0[keep] ^ ctrl_cw[keep]
            else:
                x0 = s0[keep]
                ctrl[0] = t0[keep]
            if ctrl[1]:
                x1 = xor(s1[keep], seed_cw)
                ctrl[1] = t1[keep] ^ ctrl_cw[keep]
            else:
                x1 = s1[keep]
                ctrl[1] = t1[keep]
            (seed[0], w0) = self.convert(level, x0, ctx, nonce)
            (seed[1], w1) = self.convert(level, x1, ctx, nonce)

            if level < self.BITS - 1:
                b = cast(list[Field], beta_inner[level])
            else:
                b = cast(list[Field], beta_leaf)
            if len(b) != self.VALUE_LEN:
                raise ValueError(
                    "length of beta must match the value length"
                )
```

```
            w_cw = vec_add(vec_sub(b, w0), w1)
            # Implementation note: this conditional negation should be
            # replaced with a constant time select or a constant time
            # multiplication in practice in order to reduce leakage via
            # timing side channels.
            if ctrl[1]:
                for i in range(len(w_cw)):
                    w_cw[i] = -w_cw[i]

            public_share.append((seed_cw, ctrl_cw, w_cw))
        return (public_share, key)
```

8.3.3. Key Evaluation

The description of the IDPF-evaluation algorithm makes use of auxiliary functions extend() and convert() defined in Section 8.3.4.

```
   def eval(
            self,
            agg_id: int,
            public_share: list[CorrectionWord],
            key: bytes,
            level: int,
            prefixes: Sequence[tuple[bool, ...]],
            ctx: bytes,
            nonce: bytes) -> list[list[Field64]] | list[list[Field255]]:
        if agg_id not in range(self.SHARES):
            raise ValueError('aggregator id out of range')
        if level not in range(self.BITS):
            raise ValueError('level out of range')
        if len(set(prefixes)) != len(prefixes):
            raise ValueError('prefixes must be unique')

        out_share = []
        for prefix in prefixes:
            if len(prefix) != level + 1:
                raise ValueError('incorrect prefix length')

            # The Aggregator's output share is the value of a node of
            # the IDPF tree at the given 'level'. The node's value is
            # computed by traversing the path defined by the candidate
            # 'prefix'. Each node in the tree is represented by a seed
            # ('seed') and a control bit ('ctrl').
            seed = key
            ctrl = bool(agg_id)
            y: FieldVec
            for current_level in range(level + 1):
                bit = int(prefix[current_level])
```

```
                # Implementation note: typically the current round of
                # candidate prefixes would have been derived from
                # aggregate results computed during previous rounds.
                # For example, when using the IDPF to compute heavy
                # hitters, a string whose hit count exceeded the
                # given threshold in the last round would be the
                # prefix of each ‘prefix‘ in the current round. (See
                # [BBCGGI21, Section 5.1].) In this case, part of the
                # path would have already been traversed.
                #
                # Re-computing nodes along previously traversed paths is
                # wasteful. Implementations can eliminate this added
                # complexity by caching nodes (i.e., ‘(seed, ctrl)‘
                # pairs) output by previous calls to ‘eval_next()‘.
                (seed, ctrl, y) = self.eval_next(
                    seed,
                    ctrl,
                    public_share[current_level],
                    current_level,
                    bit,
                    ctx,
                    nonce,
                )
            if agg_id == 0:
                out_share.append(cast(list[Field], y))
            else:
                out_share.append(vec_neg(cast(list[Field], y)))
        return cast(
            list[list[Field64]] | list[list[Field255]],
            out_share,
        )

    def eval_next(
            self,
            prev_seed: bytes,
            prev_ctrl: bool,
            correction_word: CorrectionWord,
            level: int,
            bit: int,
            ctx: bytes,
            nonce: bytes) -> tuple[bytes, bool, FieldVec]:
        """
        Compute the next node in the IDPF tree along the path determined
        by a candidate prefix. The next node is determined by ‘bit‘, the
        bit of the prefix corresponding to the next level of the tree.
        """

        seed_cw = correction_word[0]
```

```
        ctrl_cw = correction_word[1]
        w_cw = cast(list[Field], correction_word[2])
        (s, t) = self.extend(level, prev_seed, ctx, nonce)

        # Implementation note: these conditional operations and
        # input-dependent array indices should be replaced with
        # constant-time selects in practice in order to reduce leakage
        # via timing side channels.
        if prev_ctrl:
            s[0] = xor(s[0], seed_cw)
            s[1] = xor(s[1], seed_cw)
            t[0] ^= ctrl_cw[0]
            t[1] ^= ctrl_cw[1]

        next_ctrl = t[bit]
        convert_output = self.convert(level, s[bit], ctx, nonce)
        next_seed = convert_output[0]
        y = cast(list[Field], convert_output[1])
        # Implementation note: this conditional addition should be
        # replaced with a constant-time select in practice in order to
        # reduce leakage via timing side channels.
        if next_ctrl:
            for i in range(len(y)):
                y[i] += w_cw[i]

        return (next_seed, next_ctrl, cast(FieldVec, y))
```

8.3.4. Auxiliary Functions

```
    def extend(
            self,
            level: int,
            seed: bytes,
            ctx: bytes,
            nonce: bytes) -> tuple[list[bytes], list[bool]]:
        xof = self.current_xof(
            level,
            seed,
            format_dst(1, 0, 0) + ctx,
            nonce,
        )
        s = [
            bytearray(xof.next(self.KEY_SIZE)),
            bytearray(xof.next(self.KEY_SIZE)),
        ]
        # Use the least significant bits as the control bit correction,
        # and then zero it out. This gives effectively 127 bits of
        # security, but reduces the number of AES calls needed by 1/3.
```

```
        t = [bool(s[0][0] & 1), bool(s[1][0] & 1)]
        s[0][0] &= 0xFE
        s[1][0] &= 0xFE
        return ([bytes(s[0]), bytes(s[1])], t)

    def convert(
            self,
            level: int,
            seed: bytes,
            ctx: bytes,
            nonce: bytes) -> tuple[bytes, FieldVec]:
        xof = self.current_xof(
            level,
            seed,
            format_dst(1, 0, 1) + ctx,
            nonce,
        )
        next_seed = xof.next(self.KEY_SIZE)
        field = self.current_field(level)
        w = xof.next_vec(field, self.VALUE_LEN)
        return (next_seed, cast(FieldVec, w))

    def current_xof(self,
                    level: int,
                    seed: bytes,
                    dst: bytes,
                    nonce: bytes) -> Xof:
        if level < self.BITS-1:
            return XofFixedKeyAes128(seed, dst, nonce)
        return XofTurboShake128(seed, dst, nonce)
```

9. Security Considerations

VDAFs (Section 5) have two essential security goals:

1. Privacy: an attacker that controls the Collector and a subset of Clients and a subset of Aggregators learns nothing about the measurements of honest Clients beyond what it can deduce from the aggregate result. It is assumed that the attacker controls the entire network except for channels between honest Clients and honest Aggregators. In particular, it cannot forge or prevent transmission of messages on these channels.

2. Verifiability: an attacker that controls a subset of Clients cannot cause the Collector to compute anything other than the aggregate of the measurements of honest Clients, plus valid measurements from some of the attacker-controlled Clients. It is assumed that the attacker eavesdrops on the network but does not control transmission of messages between honest parties.

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

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

* Establishing secure channels:

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

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

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

In such an environment, a VDAF provides the high-level privacy property described above: the Collector learns only the aggregate result, 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 provide:

1. Mitigation of Sybil attacks [Dou02]. In this attack, the adversary observes a subset of input shares transmitted by a Client it is interested in. It allows the input shares to be processed, but corrupts and picks bogus measurements for the remaining Clients. Applications can guard against these risks by adding additional controls on report submission, such as Client authentication and rate limits.

2. Differential privacy [Dwo06]. Depending on the distribution of the measurements, the aggregate result itself can still leak a significant amount of information about an individual measurement or the person that generated it.

3. Verifiability in the presence of a malicious Aggregator. An Aggregator can, without detection, manipulate the aggregate result by modifying its own aggregate share.

4. Guaranteed output delivery [GSZ20]. An attacker that controls transmission of messages between honest parties can prevent computation of the aggregate result by dropping messages.

9.1. The Verification Key

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

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

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

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

9.2. The Nonce

The sharding and preparation steps of VDAF execution depend on a nonce associated with the Client's report. To ensure privacy of the underlying measurement, the Client MUST generate this nonce using a CSPRNG. This is required in order to leverage security analysis for the privacy definition of [DPRS23], which assumes the nonce is chosen at random prior to generating the report. Uniqueness of the nonce is not sufficient because the verification key is controlled by the attacker.

Applications will need to protect against replay attacks to prevent disallowed re-use of reports (see Section 9.4). Furthermore, in applications that aim to achieve differential privacy, it is necessary to limit how much each party contributes to a single batch or multiple batches. It is RECOMMENDED that the nonce generated by the Client be used by the Aggregators for replay protection.

9.3. The Public Share

The Aggregators MUST ensure they have both received the same public share from the Client. It is sufficient, for example, to exchange a hash of the public share over a secure channel.

9.4. The Aggregation Parameter

As described in Section 4.3 and Section 5.3 respectively, DAFs and VDAFs may impose restrictions on the re-use of reports. For Prio3, reports should only be aggregated once; for Poplar1, reports may be aggregated multiple times, but never twice at the same level of the tree. Otherwise, one risks re-using correlated randomness, which might compromise confidentiality of the Client's measurement.

Higher level applications that use DAFs or VDAFs MUST enforce aggregation parameter validity. In particular, prior to beginning preparation with an aggregation parameter provided by the Collector, they MUST invoke is_valid() to decide if the parameter is valid given the sequence of previously accepted parameters.

Note that aggregating a batch of reports multiple times, even with a valid sequence of aggregation parameters, can result in information leakage beyond what is used by the application.

For example, when Poplar1 is used for heavy hitters, the Aggregators learn not only the heavy hitters themselves, but also the prefix tree (as defined in Section 8) computed along the way. Indeed, this leakage is inherent to any construction that uses an IDPF (Section 8.1) in the same way. Depending on the distribution of the

measurements, the prefix tree can leak a significant amount of information about unpopular inputs. For instance, it is possible (though perhaps unlikely) for a large set of non-heavy-hitter values to share a common prefix, which would be leaked by a prefix tree with a sufficiently small threshold.

A malicious adversary controlling the Collector and one of the Aggregators can further turn arbitrary non-heavy prefixes into heavy ones by tampering with the IDPF output at any position. While the construction ensures that the nodes evaluated at one level are children of the nodes evaluated at the previous level, this still may allow an adversary to discover individual non-heavy strings. This is called a "steering attack".

The only practical, general-purpose defense against steering attacks is to compose Poplar1 with some mechanism for differential privacy. It is therefore RECOMMENDED to use differential privacy for any heavy-hitter type application.

9.5. Safe Usage of IDPF Outputs

The arithmetic sketch described in Section 8 is used by the Aggregators to check that the shares of the vector obtained by evaluating a Client's IDPF at a sequence of candidate prefixes has at most one non-zero value, and that the non-zero value is 1. Depending on how the values are used, the arithmetic sketch on its own may not be sufficient to verify the correctness of the computation. In particular, a malicious Client may attempt to influence the computation by choosing an IDPF that evaluates to 1 at more than one node at a given level of the tree.

This issue can be mitigated by using an IDPF that is extractable as defined in Appendix D of [BBCGGI21]. Extractability ensures that, for a particular level of the tree, it is infeasible for an attacker to control values of the IDPF such that it takes on chosen non-zero values at more than one node. (It can practically only achieve the zero function, a point function, or a pseudorandom function.)

The IDPF specified in Section 8.1 only guarantees extractability at the last level of the tree. (This is by virtue of using a larger field for the leaves than for inner nodes and using an XOF to derive leaves that is safe to model as a random oracle. See Section 9.6.) For intermediate levels, it is feasible for a client to produce IDPF shares with two controlled non-zero nodes.

This is not an issue for running heavy hitters, since (1) each node in the prefix tree is a child of a previously traversed node, (2) the arithmetic sketch would detect double voting at every level of the

prefix tree, and (3) the IDPF is extractable at the last level of the tree. However, the lack of extractability at intermediate levels may result in attacks on the correctness of the computation in certain applications.

Thus applications SHOULD NOT use prefix counts for intermediate levels for any purpose beyond computing the prefix tree for heavy hitters.

9.6. Safe Usage of XOFs

In the security analyses of these protocols, XOFs (Section 6.2) are usually modeled as random oracles. XofTurboShake128 is designed to be indifferentiable from a random oracle [MRH04], making it a suitable choice for most situations.

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

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

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

XofFixedKeyAes128 in Section 6.2.2 implements a correlation-robust hash function using fixed-key AES. For security, it assumes that AES with a fixed key can be modeled as a random permutation [GKWY20]. Additionally, a different AES key is used for every report, which in the ideal cipher model leads to better concrete security [GKWWY20].

Note that for verifiability, the analysis of [BBCGGI21] still assumes a random oracle to make the IDPF extractable. Thus XofTurboShake128 is used instead for the last level of the tree. It is important that XofTurboShake128 supports many seed lengths, in particular 16 bytes, as this is the seed size for the inner levels.

While XofFixedKeyAes128 has been shown to be differentiable from a random oracle [GKWWY20], there are no known attacks exploiting this difference. And even if the IDPF is not extractable, Poplar1 guarantees that every client can contribute to at most one prefix among the ones being evaluated by the helpers.

9.7. Choosing FLP Parameters

Prio3 and other systems built from the FLP of Section 7.3 may benefit from choosing a field size that is as small as possible. Generally speaking, a smaller field results in lower communication and storage costs. Care must be taken, however, since a smaller field also results in degraded (or even vacuous) verifiability.

Different variants of Prio3 (Section 7) use different field sizes: Prio3Count and Prio3Sum use Field64; but Prio3SumVec, Prio3Histogram, and Prio3MultihotCountVec all use Field128, a field that is twice as large as Field64. This is due to the use of joint randomness (Section 7.1) in the latter variants. Joint randomness allows for more flexible circuit design (see Section 7.3.1.1), but opens up Prio3 to offline attacks in which the attacker searches for input shares for an invalid measurement that derive joint randomness that causes the circuit to accept. Choosing a large enough field ensures this computation is too expensive to be feasible. (See [DPRS23], Theorem 1.) Note that privacy is not susceptible to such attacks.

Another way to mitigate this issue (or improve verifiability in general) is to generate and verify multiple, independent proofs. (See Section 7.1.2.) For Prio3, the PROOFS parameter controls the number of proofs (at least one) that are generated and verified. In general the soundness error of the FLP is given by the following formula:

```
(circuit_soundness + flp_soundness)**PROOFS
```

where:

* circuit_soundness is the soundness of the validity circuit (Section 7.3.2)

* flp_soundness is the base soundness of the proof system ([BBCGGI19], Theorem 4.3)

For circuits involving joint randomness, one should aim for the soundness error to be close to 2**-128 in order to mitigate offline attacks. Such circuits MUST use Field128 with at least one proof or Field64 with at least three proofs. Depending on the circuit, Field64 with two proofs might have significantly lower soundness than Field128 with one proof.

Weak parameters (too small a field, too few proofs, or both) can be exploited to attack any aggregation task using those parameters. To mitigate offline attacks, it is necessary to disable all tasks that use the weak parameters.

9.8. Choosing the Number of Aggregators

Two Aggregators are required for privacy in the threat model, but some (V)DAFs, including Prio3 (Section 7), allow for any number of Aggregators, only one of which needs to be trusted in order for the computation to be private. To hedge against corruptions that happen during the course of the attack, deployments may consider involving more than two Aggregators as described for example in Section 5.7.2. Note however that some schemes are not compatible with this mode of operation, such as Poplar1.

9.9. Defense-in-Depth Measures

Prio3 and Poplar1 are designed to resist some attacks that fall outside the main threat model for VDAFs.

Broadly speaking, domain separation is used to prevent cross protocol attacks, in which data from evaluation of one VDAF translates to an attack against another. For example:

1. Weak entropy sources: the VDAF algorithm ID is bound to each XOF invocation, thereby ensuring the outputs are different between VDAF invocations, even if the underlying randomness is the same. For example, two different instances of Prio3 would compute different measurement shares.

2. Weak parameters: Prio3 variants that require joint randomness are subject to offline attacks against verifiability. These attacks are feasible if the field size or number of proofs is sufficiently small. (See Section 9.7.) The joint randomness derivation is bound to both the field (via the algorithm ID) and the number of proofs, thereby ensuring that joint randomness derived for weak parameters is not reused for stronger parameters. In addition, the joint randomness is bound to the application context, meaning any work the attacker does to attack some application is not useful for other applications that use the same parameters.

There are also some important limitations to be aware of. For example, Prio3 provides domain separation between families of circuits, but does not provide domain separation between instances of a circuit. Concretely, it is possible for Aggregators to accept a report for Prio3SumVec from a Client who disagrees with them on the value of bits and length (so long as the encoded measurement is the same size). This is because there is no binding of the circuit parameters to the computation.

9.10. Side-Channel Resistance

Implementations of VDAFs should incorporate defenses against side-channel attacks. For side-channel attacks against the privacy security goal, the relevant threat model includes an attacker that may control the Collector, a subset of Clients, and a subset of Aggregators, and monitor side-channel signals from the honest Clients and Aggregators. Side-channel attacks by third parties may indirectly target verifiability by trying to leak the Aggregators' verification key. Thus, implementations of Clients and Aggregators should treat measurements, input shares, output shares, and the verification key as secret, and avoid leaking those secret values or any intermediate computations that depend on them.

For example, the following routines should all be implemented in a side-channel resistant manner.

* Finite field arithmetic

* XOFs

* IDPF generation and evaluation, including handling of control bits

10. IANA Considerations

IANA is requested to make one new registry:

* DAF and VDAF Identifiers

This registry should be created under the heading "Verifiable Distributed Aggregation Functions (VDAF)", and administered under the Specification Required policy [RFC8126].

The "VDAF Identifiers" registry lists identifiers for Distributed Aggregation Functions (DAFs) and Verifiable Distributed Aggregation Functions (VDAFs). These identifiers are four-byte values, so the minimum possible value is 0x00000000 and the maximum possible value is 0xffffffff.

Template:

* Value: The four-byte identifier for the DAF or VDAF

* Scheme: The name of the DAF or VDAF

* Type: Either "DAF" for a Distributed Aggregation Function or "VDAF" for a Verifiable Distributed Aggregation Function

* Reference: Where the algorithm is defined

The initial contents of the registry are as follows:

Value	Scheme	Type	Reference
0x00000000	Reserved	n/a	RFC XXXX
0x00000001	Prio3Count	VDAF	Section 7.4.1 of RFC XXXX
0x00000002	Prio3Sum	VDAF	Section 7.4.2 of RFC XXXX
0x00000003	Prio3SumVec	VDAF	Section 7.4.3 of RFC XXXX
0x00000004	Prio3Histogram	VDAF	Section 7.4.4 of RFC XXXX
0x00000005	Prio3MultihotCountVec	VDAF	Section 7.4.5 of RFC XXXX
0x00000006	Poplar1	VDAF	Section 8.2 of RFC XXXX
0xFFFF0000 to 0xFFFFFFFF	Reserved for Private Use	n/a	n/a

Table 19: Verifiable Distributed Aggregation Function Identifiers Registry

(RFC EDITOR: Please replace "RFC XXXX" above with the RFC number assigned to this document.)

VDAF identifiers are used for domain separation, as described in Section 6.2.3. Domain separation guards against cross protocol attacks and certain failures of entropy sources. See Section 9.9.

The benefits of domain separation are undermined if different VDAFs are used with the same VDAF Identifier. The "Reserved for Private Use" code points should thus be used judiciously, because they provide no defense against such collisions. Applications SHOULD prefer the use of registered code points.

11. References

11.1. Normative References

[AES] Dworkin, M. J., Barker, E., Nechvatal, J. R., Foti, J., Bassham, L. E., Roback, E., and J. Dray Jr, "Advanced Encryption Standard (AES)", 2001, <https://www.nist.gov/publications/advanced-encryption-standard-aes>.

[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>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/rfc/rfc8126>.

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

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

[TurboSHAKE] Viguier, B., Wong, D., Van Assche, G., Dang, Q., and J. Daemen, "KangarooTwelve and TurboSHAKE", Work in Progress, Internet-Draft, draft-irtf-cfrg-kangarootwelve-17, 21 February 2025, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-kangarootwelve-17>.

11.2. Informative References

[AGJOP21] Addanki, S., Garbe, K., Jaffe, E., Ostrovsky, R., and A. Polychroniadou, "Prio+: Privacy Preserving Aggregate Statistics via Boolean Shares", Security and Cryptography for Networks (SCN), 2022, <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, <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 Security & Privacy (S&P), 2021, <https://ia.cr/2021/017>.

[BBCGGI23] Boneh, D., Boyle, E., Corrigan-Gibbs, H., Gilboa, N., and Y. Ishai, "Arithmetic Sketching", Crypto, 2023, <https://ia.cr/2023/1012>.

[BGI15] Boyle, E., Gilboa, N., and Y. Ishai, "Function Secret Sharing", Eurocrypt, 2015, <https://www.iacr.org/archive/eurocrypt2015/90560300/90560300.pdf>.

[CGB17] Boneh, D. and H. Corrigan-Gibbs, "Prio: Private, Robust, and Scalable Computation of Aggregate Statistics", USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2017, <https://dl.acm.org/doi/10.5555/3154630.3154652>.

[DAP] Geoghegan, T., Patton, C., Pitman, B., Rescorla, E., and C. A. Wood, "Distributed Aggregation Protocol for Privacy Preserving Measurement", Work in Progress, Internet-Draft, draft-ietf-ppm-dap-16, 2 September 2025, <https://datatracker.ietf.org/doc/html/draft-ietf-ppm-dap-16>.

[Dou02] Douceur, J. R., "The Sybil Attack", International Workshop on Peer-to-Peer Systems (IPTPS), 2002, <https://doi.org/10.1007/3-540-45748-8_24>.

[DPRS23] Hannah Davis, Christopher Patton, Mike Rosulek, and Phillipp Schoppmann, "Verifiable Distributed Aggregation Functions", Privacy Enhancing Technologies Symposium (PETS), 2023, <https://ia.cr/2023/130>.

[Dwo06] Cynthia Dwork, "Differential Privacy", International Colloquium on Automata, Languages, and Programming (ICALP), 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, ., Pihur, V., and A. Korolova, "RAPPOR: Randomized Aggregatable Privacy-Preserving Ordinal Response", ACM Conference on Computer and Communications Security (CCS), 2014, <https://dl.acm.org/doi/10.1145/2660267.2660348>.

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

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

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

[GSZ20] Goyal, V., Song, Y., and C. Zhu, "Guaranteed Output Delivery Comes Free in Honest Majority MPC", Crypto, 2020, <https://link.springer.com/chapter/10.1007/978-3-030-56880-1_22>.

[I-D.draft-irtf-cfrg-cryptography-specification-02] Sullivan, N. and C. A. Wood, "Guidelines for Writing Cryptography Specifications", Work in Progress, Internet-Draft, draft-irtf-cfrg-cryptography-specification-02, 7 July 2025, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-cryptography-specification-02>.

[MPDST25] Mouris, D., Patton, C., Davis, H., Sarkar, P., and N. G. Tsoutsos, "Mastic: Private Weighted Heavy-Hitters and Attribute-Based Metrics", Privacy Enhancing Technologies Symposium (PETS), 2025, <https://eprint.iacr.org/2024/221>.

[MPRV09] Mironov, I., Pandey, O., Reingold, O., and S. Vadhan, "Computational Differential Privacy", Crypto, 2009, <https://link.springer.com/chapter/10.1007/978-3-642-03356-8_8>.

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

[OriginTelemetry] "Origin Telemetry", 2020, <https://web.archive.org/web/20221025174046/https://firefox-source-docs.mozilla.org/toolkit/components/telemetry/collection/origin.html>.

[PANEL-FEEDBACK] Hesse, J., "Review of draft-irtf-cfrg-vdaf-15", 2025, <https://mailarchive.ietf.org/arch/msg/cfrg/Omdhr4rO1pla_nlju2l7OJEGWPM/>.

[SML24] Satriawan, A., Mareta, R., and H. Lee, "A Complete Beginner Guide to the Number Theoretic Transform (NTT)", IEEE Access, vol. 11, 2024, <https://eprint.iacr.org/2024/585>.

[TestVectors] "Test vectors for Prio3 and Poplar1", commit hash 5b7df1d, December 2024, <https://github.com/cfrg/draft-irtf-cfrg-vdaf>.

Appendix A. FLP Gadgets

This section defines gadgets used to specify variants of Prio3 defined in Section 7.4 and to construct FLPs as defined in Section 7.3.

A.1. Multiplication

The multiplication gadget takes in a pair of inputs and multiplies them together. The degree of this circuit is 2.

```
class Mul(Gadget[F]):
    ARITY = 2
    DEGREE = 2

    def eval(self, _field: type[F], inp: list[F]) -> F:
        return inp[0] * inp[1]

    def eval_poly(self,
                  field: type[F],
                  inp_poly: list[list[F]]) -> list[F]:
        return poly_mul(field, inp_poly[0], inp_poly[1])
```

A.2. Polynomial Evaluation

The polynomial-evaluation gadget takes in a single input x and returns p(x), where p is a polynomial specified by the user. Its degree is the same as p.

```
   class PolyEval(Gadget[F]):
       ARITY = 1
       p: list[int]  # polynomial coefficients

       def __init__(self, p: list[int]):
           """
           Instantiate this gadget with the given polynomial.
           """
           # Strip leading zeros.
           for i in reversed(range(len(p))):
               if p[i] != 0:
                   p = p[:i+1]
                   break
           if len(p) < 1:
               raise ValueError('invalid polynomial: zero length')

           self.p = p
           self.DEGREE = len(p) - 1

       def eval(self, field: type[F], inp: list[F]) -> F:
           p = [field(coeff) for coeff in self.p]
           return poly_eval(field, p, inp[0])

       def eval_poly(self,
                     field: type[F],
                     inp_poly: list[list[F]]) -> list[F]:
           p = [field(coeff) for coeff in self.p]
           out = [field(0)] * (self.DEGREE * len(inp_poly[0]))
           out[0] = p[0]
           x = inp_poly[0]
           for i in range(1, len(p)):
               for j in range(len(x)):
                   out[j] += p[i] * x[j]
               x = poly_mul(field, x, inp_poly[0])
           return poly_strip(field, out)
```

A.3. Parallel Sum

The parallel-sum gadget applies a subcircuit to multiple inputs in parallel, then returns the sum of the results. The arity of the gadget is equal to count times the arity of the subcircuit, where count denotes the number of times the subcircuit is called. The degree of the gadget is equal to the degree of the subcircuit.

```
   class ParallelSum(Gadget[F]):
       subcircuit: Gadget[F]
       count: int

       def __init__(self, subcircuit: Gadget[F], count: int):
           self.subcircuit = subcircuit
           self.count = count
           self.ARITY = subcircuit.ARITY * count
           self.DEGREE = subcircuit.DEGREE

       def eval(self, field: type[F], inp: list[F]) -> F:
           out = field(0)
           for i in range(self.count):
               start_index = i * self.subcircuit.ARITY
               end_index = (i + 1) * self.subcircuit.ARITY
               out += self.subcircuit.eval(
                   field,
                   inp[start_index:end_index],
               )
           return out

       def eval_poly(self,
                     field: type[F],
                     inp_poly: list[list[F]]) -> list[F]:
           output_poly_length = self.DEGREE * (len(inp_poly[0]) - 1) + 1
           out_sum = [field(0) for _ in range(output_poly_length)]
           for i in range(self.count):
               start_index = i * self.subcircuit.ARITY
               end_index = (i + 1) * self.subcircuit.ARITY
               out_current = self.subcircuit.eval_poly(
                   field,
                   inp_poly[start_index:end_index]
               )
               for j in range(output_poly_length):
                   out_sum[j] += out_current[j]
           return poly_strip(field, out_sum)
```

A.4. Shims for Generating and Querying Proofs

This section specifies two "shim" gadgets, one for generating an FLP as specified in Section 7.3.3 and another for querying an FLP as specified in Section 7.3.4.

```
   class ProveGadget(Gadget[F]):
       """
       Gadget wrapper that records the input wires for each evaluation.
       """

       def __init__(self,
                    field: type[F],
                    wire_seeds: list[F],
                    g: Gadget[F],
                    g_calls: int):
           p = next_power_of_2(1 + g_calls)
           self.inner = g
           self.ARITY = g.ARITY
           self.DEGREE = g.DEGREE
           self.wires = []
           self.k = 0  # evaluation counter
           for s in wire_seeds:
               wire = field.zeros(p)
               wire[0] = s  # set the wire seed
               self.wires.append(wire)

       def eval(self, field: type[F], inp: list[F]) -> F:
           self.k += 1
           for j in range(len(inp)):
               self.wires[j][self.k] = inp[j]
           return self.inner.eval(field, inp)

       def eval_poly(self,
                     field: type[F],
                     inp_poly: list[list[F]]) -> list[F]:
           return self.inner.eval_poly(field, inp_poly)

       @classmethod
       def wrap(cls,
                valid: Valid[Measurement, AggResult, F],
                prove_rand: list[F],
                ) -> Valid[Measurement, AggResult, F]:
           """
           Make a copy of ‘valid‘ with each gadget wrapped for recording
           the wire inputs. ‘prove_rand‘ is used to produce the wire
           seeds for each gadget.
           """
           wrapped_gadgets: list[Gadget[F]] = []
           for (g, g_calls) in zip(valid.GADGETS, valid.GADGET_CALLS):
               (wire_seeds, prove_rand) = front(g.ARITY, prove_rand)
               wrapped = cls(valid.field, wire_seeds, g, g_calls)
               wrapped_gadgets.append(wrapped)
           wrapped_valid = deepcopy(valid)
```

```
            wrapped_valid.GADGETS = wrapped_gadgets
            return wrapped_valid

    class QueryGadget(Gadget[F]):
        """
        Gadget wrapper that records the input wires for each evaluation.
        Rather than evaluate the circuit, use the provided gadget
        polynomial to produce the output.
        """

        def __init__(
                self,
                field: type[F],
                wire_seeds: list[F],
                gadget_poly: list[F],
                g: Gadget[F],
                g_calls: int):
            p = next_power_of_2(1 + g_calls)
            self.alpha = field.gen() ** (field.GEN_ORDER // p)
            self.poly = gadget_poly
            self.ARITY = g.ARITY
            self.DEGREE = g.DEGREE
            self.wires = []
            self.k = 0
            for s in wire_seeds:
                wire = field.zeros(p)
                wire[0] = s  # set the wire seed
                self.wires.append(wire)

        def eval(self, field: type[F], inp: list[F]) -> F:
            self.k += 1
            for j in range(len(inp)):
                self.wires[j][self.k] = inp[j]
            return poly_eval(field, self.poly, self.alpha ** self.k)

        @classmethod
        def wrap(cls,
                 valid: Valid[Measurement, AggResult, F],
                 proof: list[F]) -> Valid[Measurement, AggResult, F]:
            wrapped_gadgets: list[Gadget[F]] = []
            for (g, g_calls) in zip(valid.GADGETS, valid.GADGET_CALLS):
                p = next_power_of_2(1 + g_calls)
                gadget_poly_len = g.DEGREE * (p - 1) + 1
                (wire_seeds, proof) = front(g.ARITY, proof)
                (gadget_poly, proof) = front(gadget_poly_len, proof)
                wrapped = cls(valid.field,
                              wire_seeds,
                              gadget_poly,
```

```
                            g,
                            g_calls)
                wrapped_gadgets.append(wrapped)
            wrapped_valid = deepcopy(valid)
            wrapped_valid.GADGETS = wrapped_gadgets
            return wrapped_valid
```

Appendix B. VDAF Preparation State

This section lists the classes used to define each Aggregator’s state during VDAF preparation (Section 5.7).

```
   class State:
       pass

   class Start(State):
       pass

   class Continued(State, Generic[PrepState]):
       def __init__(self,
                    prep_state: PrepState,
                    prep_round: int,
                    outbound: bytes):
           self.prep_state = prep_state
           self.prep_round = prep_round
           self.outbound = outbound

       def __eq__(self, other: object) -> bool:
           return isinstance(other, Continued) and \
               self.prep_state == other.prep_state and \
               self.prep_round == other.prep_round and \
               self.outbound == other.outbound

   class Finished(State, Generic[OutShare]):
       def __init__(self, out_share: OutShare):
           self.out_share = out_share

       def __eq__(self, other: object) -> bool:
           return isinstance(other, Finished) and \
               self.out_share == other.out_share

   class FinishedWithOutbound(State, Generic[OutShare]):
       def __init__(self, out_share: OutShare, outbound: bytes):
           self.out_share = out_share
           self.outbound = outbound

       def __eq__(self, other: object) -> bool:
           return isinstance(other, FinishedWithOutbound) and \
               self.out_share == other.out_share and \
               self.outbound == other.outbound

   class Rejected(State):
       pass
```

Appendix C. Test Vectors

Test vectors for Prio3 (Section 7) and Poplar1 (Section 8) are available at [TestVectors]. The test vector directory, test_vec/vdaf, contains a set of JSON files. Each file contains a test vector for an instance of class Vdaf as defined in Section 5. A test vector covers sharding, preparation, aggregation, and unsharding of a batch of several measurements. The test vector schema is defined below.

C.1. Schema

ctx: The application context string encoded in hexadecimal.

verify_key: The verification key encoded in hexadecimal.

agg_param: The aggregation parameter encoded in hexadecimal.

prep: A list of objects with the following schema:

- measurement: The measurement of type Measurement defined by the VDAF.
- nonce: The nonce encoded in hexadecimal.
- rand: The sharding randomness encoded in hexadecimal.
- public_share: The expected public share encoded in hexadecimal.
- input_shares: The expected list of input shares, each encoded in hexadecimal.
- prep_shares: The expected list of prep shares generated by each Aggregator at each round of preparation, encoded in hexadecimal.
- prep_messages: The expected list of prep messages for each round of preparation, encoded in hexadecimal.
- out_shares: The expected list of output shares, encoded in hexadecimal.

agg_shares: The expected aggregate shares encoded in hexadecimal.

agg_result: The expected aggregate result of type AggResult defined by the VDAF.

operations: This lists the VDAF operations that should be executed

as part of known answer tests, using messages from this test vector as input. Operations should be executed in the order they appear, to ensure that prepare state values are computed before they are consumed. Prepare state values are not included in test vectors because this document does not specify their representation or encoding.

Each operation in the operations list has the following schema:

operation: The type of operation to be performed. This is one of "shard", "prep_init", "prep_shares_to_prep", "prep_next", "aggregate", or "unshard".

round: For any preparation operation, the round number of the operation to be performed. This determines which prepare share, prepare state, and/or prepare message to use.

aggregator_id: The aggregator ID to use when performing this operation. This determines which messages and which prepare state to use, in addition to the aggregator ID argument itself.

report_index: The index of the report on which to perform this operation. This is an index into the prep array.

success: If this is True, the operation should succeed, and its output should match the corresponding values in the test vector. If this is False, the operation should fail, terminating preparation of this report.

The test vector schema also includes whatever parameters are required to instantiate the VDAF. These are listed in the subsections below.

C.1.1. Prio3Count

shares: The number of shares, an integer.

C.1.2. Prio3Sum

shares: The number of shares, an integer.

max_measurement: The largest valid measurement, an integer. Each measurement is in the range [0, max_measurement].

C.1.3. Prio3SumVec

shares: The number of shares, an integer.

length: The length of the vector, an integer.

chunk_length: a parameter of the ParallelSum gadget, an integer.

bits: the bit length of each element of the vector, an integer. Each element is in the range [0, 2**bits).

C.1.4. Prio3Histogram

shares: The number of shares, an integer.

length: The length of the vector, an integer.

chunk_length: a parameter of the ParallelSum gadget, an integer.

C.1.5. Prio3MultihotCountVec

shares: The number of shares, an integer.

length: The length of the vector, an integer.

chunk_length: a parameter of the ParallelSum gadget, an integer.

max_weight: The largest vector weight, an integer. The sum of the elements of the measurement must be in the range [0, max_weight].

C.1.6. Poplar1

bits: The length of each input in bits, an integer.

Acknowledgments

The impetus of this work is the success of recent deployments of predecessors of Prio3. These include the Mozilla Origin Telemetry project [OriginTelemetry] and the Exposure Notification Private Analytics system [ENPA] developed jointly by ISRG, Google, Apple, and others. Together these systems have aggregated data from hundreds of millions of users.

As the name implies, Prio3 is a descendant of the original Prio construction [CGB17]. 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 security considerations in Section 9 are based largely on the security analysis of [DPRS23]. Thanks to Hannah Davis and Mike Rosulek, who lent their time to developing definitions and security proofs.

Thanks to Julia Hesse who provided feedback on behalf of the Crypto Review Panel.

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

Authors' Addresses

Richard L. Barnes
Cisco
Email: rlb@ipv.sx

David Cook
ISRG
Email: divergentdave@gmail.com

Christopher Patton
Cloudflare
Email: chrispatton+ietf@gmail.com

Phillipp Schoppmann
Google
Email: schoppmann@google.com