Crypto Forum
Internet-Draft
Intended status: Informational
Expires: 5 September 2024

H. Davis
Seagate
D. Mouris
Nillion
C. Patton
Cloudflare
P. Sarkar
Supra Research
N. G. Tsoutsos
University of Delaware

4 March 2024

The Mastic VDAF
draft-mouris-cfrg-mastic-02

Abstract

This document describes Mastic, a two-party VDAF for the following aggregation task: each client holds a string, and the collector wishes to count how many of these strings begin with a given prefix. Such a VDAF can be used to solve the private heavy hitters problem, where the goal is to compute the subset of the strings that occur most frequently without learning which client holds which string. This document also describes different modes of operation for Mastic that support additional use cases and admit various performance and security trade-offs.

About This Document

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

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-mouris-cfrg-mastic/.

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

Source for this draft and an issue tracker can be found at https://github.com/jimouris/draft-mouris-cfrg-mastic.

Status of This Memo

This Internet-Draft will expire on 5 September 2024.

Copyright Notice

Table of Contents

1. Introduction

> TO BE REMOVED BY RFC EDITOR: The source for this draft and the reference code can be found at https://github.com/jimouris/draft-mouris-cfrg-mastic.

The "private heavy hitters" problem is to recover the most popular measurements generated by clients without learning the measurements themselves. For example, a browser vendor might want to know which websites are visited most frequently without learning which clients visited which websites.

For string measurements, this problem can be solved by combining a binary search with a subroutine solving the "private prefix histogram" subproblem. The goal of this subproblem is to compute a histogram over the fixed-length prefixes of client measurement strings without revealing the prefixes. The subproblem can be solved using a Verifiable Distributed Aggregation Function, or VDAF [VDAF]. In particular, the Poplar1 VDAF described in Section 8 of [VDAF] describes how to distribute this computation amongst a small set of aggregation servers such that, as long as one server is honest, no individual measurement is observed in the clear. At the same time, Poplar1 allows the servers to detect and remove any invalid measurements that would otherwise corrupt the computation of the histogram.

This document describes Mastic [MPDST24], a VDAF that can be used as a drop-in replacement for Poplar1, while offering improved performance and communication cost. Based on the PLASMA protocol [MST24], the scheme's design also improves communication complexity, requiring just one round for report preparation compared to Poplar1's two rounds. Mastic is specified in Section 4.

Mastic is also highly extensible. Like Poplar1, Mastic's core functionality is to compute prefix histograms. Mastic allows this basic counter data type to be generalized to support a wide variety of secure aggregation tasks. In particular, Mastic supports any data type that can be expressed as a type for the Prio3 VDAF Section 7 of [VDAF]. For example, the counter could be replaced with a bounded weight (say, representing how much time was sepnt on a website) such that the heaviest "weight" measurements are recovered. We describe this mode of operation in Section 5.1.

This generalization also allows Mastic to support another important use case. A desirable feature for a secure aggregation systems is the ability to "drill down" on the data by splitting up the aggregate based on specific properties of the clients. For example, a browser vendor may wish to partition aggregates by version (different versions of the browser may have different performance profiles) or geographic location. We will call these properties "attributes". [CP: See https://github.com/ietf-wg-ppm/draft-ietf-ppm-dap/issues/489 for the discussion that originally motivated this idea.]

This requires representing the information in such a way that that measurements submitted by clients with the same attribute are aggregated together. Prio3 can be adapted for this purpose, but the communication cost would be linear in the number of possible distinct attributes, which quickly becomes prohibitive if the number of attributes is large or subject to change over time. For example, attributes might encode the client's user agent (Section 10.1.5 of [RFC9110]), which has many possible values that tend to change over time.

Mastic encodes the attribute and measurement such that, for an arbitrary sequence of attributes, the reports can be "queried" to reveal the aggregate for each attribute without learning the attribute or measurement of any client. We describe this mode of operation in Section 5.2.

Finally, we describe two modes of operation for Mastic that admit useful performance and security trade-offs.

First, we describe an optimization for plain (i.e., non-weighted) heavy hitters that, in the best case, reduces the communication cost of preparation from linear in the number of reports to constant, leading to a dramatic improvement in performance compared to Poplar1. This best-case behavior is observed when all clients behave honestly: if a fraction of the clients submit invalid reports, then additional rounds of communication are required in order to isolate the invalid reports and remove them. We describe this idea in detail in Section 5.3.

Second, in Section 6 we describe an enhancement that allows Mastic to achieve robustness in the presence of a malicious Aggregator. Rather than two aggregation servers as in the previous modes, this mode of operation involves three Aggregators, where every pair of Aggregators communicate over a different channel. Using a third Aggregator, we can lift the security of Mastic from the semi-honest setting to malicious security. While more complex to implement than 2-party Mastic, this mode allows achieves "full security", where both privacy and robustness hold in the honest majority setting.

1.1. Motivating Applications

Mastic has two modes of operation, i.e., Weighted Heavy-Hitters Section 5.1 and Attribute-Based Metrics Section 5.2. We describe one applications of interest for each mode.

1.1.1. Network Error Logging

Network Error Logging (NEL) is a mechanism used by web browsers to report errors that occur while attempting to establish a connection to a server [W3C23]. Some of these errors are visible to the server, but not all: failures in DNS, TCP, TLS, and HTTP can occur without the server having any visibility into the issue. A small amount of connection errors is expected, even under normal operating conditions; but a sudden, substantial increase in errors may be an indication of an outage, or a configuration issue impacting millions of users. Without a reporting mechanism like NEL, these events would only manifest in the server's telemetry as a drop in overall traffic.

NEL is particularly important for content delivery networks that handle HTTP traffic for a large number of websites (typically millions). A content delivery network acts as a reverse proxy between clients and origin servers that provides a layer of caching and security services, such as DDoS protection.

Reports are comprised of the URL the client attempted to navigate to (e.g., "https://example.com"), the type of error that occurred, and metadata related to the attempt, such as the time that elapsed between when the connection attempt began and the error was observed (e.g., Section 7 of [W3C23]). Clients may also report successful connection attempts to give the server a sense of the error rate. The exact client behavior is determined by the reporting policy specified by the server (see Section 5.1 of [W3C23]).

NEL data is privacy-sensitive for two reasons. First, it exposes information that the server would not otherwise have access to, which can be abused to probe the client's network configuration as described in Section 9 of [W3C23]. Second, for operational reasons, the reporting endpoint may be organizationally separated from the server (i.e., run on different cloud infrastructures), leading to an increased risk of the client's browsing history being exposed (e.g., in a data breach).

MPC helps mitigate these risks by revealing to the endpoint only the information it needs to fulfill its service level objectives. This means, of course, we must be satisfied with limited functionality. Fortunately, Mastic allows us to preserve the most important functionality of NEL while minimizing privacy loss.

Mastic can be applied to a simplified version of NEL where each client reports a tuple (dom, err) consisting of a domain name dom (e.g., "example.com") and a value err that represents an error (e.g., "dns.unreachable") or an indication that no error occurred (e.g., "ok"). Notably, this can be easily extended in Mastic to represent more elaborate metrics. e.g., where each weight includes the time it took each browser to report the error (and the aggregate is the average error reporting time), user agent (browser type and version), etc. However, our main goal is to understand 1) the distribution of errors and 2) which domains are impacted.

We expect there to be a large number of distinct domain names (millions in the case of content delivery networks) and only a small number of error variants (the NEL spec [W3C23] defines 30 variants). The following Mastic parameters are suitable for this application.

Each input would encode the domain dom encoded with a number of bits sufficient to uniquely represent most of the domains; and each weight would represent the error variant dom. To compute the distribution of errors, we would encode each error variant as a distinct bucket of a histogram so that [1, 0, 0, ...] represents "ok", [0, 1, 0, ...] represents "dns.unreachable", and so on. (See ection 6 of [W3C23].), This is similar to Prio3Histogram (Section 7 of [VDAF].)

1.1.2. Attribute-Based Browser Telemetry

Web browsers collect telemetry generated by users as they navigate the web to gain insights into trends that guide product decisions. In many cases, Prio3 (Section 7 of [VDAF]) can be used to privately aggregate this telemetry. However, this comes at the cost of flexibility.

For example, Prio3 can be used to collect page load metrics from Browser for a list of known popular sites (e.g., "example.com"). The purpose of these metrics is to detect if changes to these sites cause regressions that might be correlated with an increased average load time or error rate. A subtle, but important requirement for this system is the ability to break down the metrics by client attributes. Suppose for example that we want to aggregate by 1) the software version, and 2) the information about the client’s location.

Mastic provides a simple solution to this problem. For the sake of presentation, we consider a simplified use case (the same approach can be applied to any aggregation task for which Prio3 (Section 7 of [VDAF]) is suitable). Each client reports a tuple (ver, loc, site, time) where: ver is a string representing the client’s software version (e.g., "Browser/122.0"); loc is a string encoding its country code (e.g., "GR", "US", "IN", etc.); site is one of a fixed set of sites (e.g., "example.com", "example.org", etc.); and time is the load time of the site in seconds. The version and location are included in the Mastic input; the site and load time are encoded by the corresponding weight. Notably, this is just one example of what Mastic can do; the same idea can be applied to other types of metrics.

Compared to the private NEL application in Section 1.1.1, the number of possible inputs here is relatively small: there are less than 200 country codes and a handful of browser versions in wide use at any given time. This means the aggregators can enumerate a set of inputs of interest and evaluate them immediately. Consider the following parameters for Mastic, in its attribute-based metrics mode of operation Section 5.2:

- Attributes: Two-letter country codes can easily be encoded in 2 bytes. Likewise, the number of distinct browser versions is easily less than 216, so 2 bytes are sufficient. Therefore, each attribute can be encoded with just 32 bits.

- Values: Similar to private NEL, each weight is a 0-vector except for a single 1 representing a bucket in a histogram. We represent (site, time) as a histogram bucket as follows. First, we quantize time (in seconds) into one of four buckets: [0, 0.1), [0.1, 1), [1, 5), and [5, inf). Let 0 < t <= 4 denote the time bucket for time. Next, suppose we wish to track metrics for 25 sites. Let 0 < s <= 25 denote the index of site in this list. Then the index of 1 is simply t * s.

2. Conventions and Definitions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

This document uses the following terms as defined in [VDAF]: "Aggregator", "Client", "Collector", "aggregate result", "aggregate share", "aggregation parameter", "batch", "input share", "measurement", "order", "output share", "prep message", "prep share", and "report".

In Mastic, a Client's VDAF measurement comprises two components, which we denote alpha and beta. The function that each component serves depends on the use case: for weighted (Section 5.1) and plain (Section 5.3) heavy-hitters, we shall refer to alpha as the "payload" and beta as the payload's "weight"; for attribute-based-metrics (Section 5.2), we shall refer to alpha as the "attribute" and to beta as the "payload". When doing so is unambiguous, we may also refer to the payload as the "measurement".

The DPF tree always has as a root the "empty string", which in turn has strings "0" and "1" as the left and right children, respectively.

3. Preliminaries

Mastic makes use of three primitives described in the base VDAF specification [VDAF]: finite fields, eXtendable Output Functions (XOFs), and Fully Linear Proofs (FLPs). It also makes use of a fourth primitive, which extends the security properties of Incremental Distributed Point Functions (IDPFs), also described in the base specification. All four primitives are described below.

3.1. Finite fields

An implementation of the Field interface in Section 6.1 of [VDAF] is required. This object implements arithmetic in a prime field with a modulus suitable for use with the Number Theoretic Transform (called "FFT-friendly" in [VDAF]).

3.2. XOF

An implementation of the Xof interface in Section 6.2 of [VDAF] is required. This object implements an XOF that takes a short seed and some auxiliary data as input and outputs a string of any length required for the application.

3.3. FLP

An implementation of the Flp interface in Section 7.1 of [VDAF] is required. This object implements a zero-knowledge proof system used to verify that the measurement conforms to the data type required by the application: for weighted heavy hitters (Section 5.1), FLPs are used to check the weight; in attribute-based-metrics (Section 5.2), they are used to check the measurement itself.

The Client generates a proof and sends secret shares of this proof to each Aggregator. Verification is split into two phases. In the first phase, each Aggregator "queries" its share of the value and proof to obtain its "verifier share". In the second phase, the Aggregators sum up the verifier shares and use the sum to decide if the input is valid.

3.4. Ordering function order

The function order(list[Vidpf.Field]) -> Integer defines a total ordering of sums of weights. For plain heavy hitters, order is the identity function.

3.5. Verifiable IDPF (VIDPF)

Function secret sharing [GI14] allows secret sharing of the output of a function f() into additive shares, where each function share is represented by a separate key. These keys enable the Aggregators to efficiently generate an additive share of the functions output f(x) for a given input x. Distributed Point Functions (DPF) are a particular case of function secret sharing where f() is a "point function" for which f(x) = beta if x equals alpha and 0 otherwise for some alpha, beta. The computation is distributed in such a way that no one party knows either the point or what it evaluates to.

An IDPF (Section 8.1 of [VDAF]) generalizes DPF by secret-sharing an "incremental point function", i.e., the "point" in DPF is now a path on a full binary tree from the root to one of the leaves. Here we take alpha to be a bit string of fixed length, and we have that f(x) = beta if x is a prefix of alpha and 0 otherwise.

An IDPF has two main operations. The first is the key-generation algorithm, which is run by the Client. It takes as input alpha and beta and returns two values: a list of "key shares", one for each Aggregator; and the "public share", to be distributed to both Aggregators. The second is the key-evaluation algorithm, run by each Aggregator. It takes as input a candidate prefix string x, the public share, and the Aggregator’s key share and returns the Aggregator’s share of f(x).

Shares of the IDPF outputs can be aggregated together across multiple reports. This is used in Poplar1 (Section 8 of [VDAF]) to solve the private prefix histogram problem. IDPFs are private in the sense that each Aggregator learns nothing about the underlying inputs beyond the value of this sum. However, IDPFs on their own do not provide robustness. For example, it is possible for a malicious Client to fool the Aggregators into accepting malformed counter (i.e., a value other than 0 or 1). It is also possible for a Client to "vote twice" by constructing key shares for which f(x) = f(x') = beta, where x and x' are distinct, equal-length candidate prefixes.

To mitigate these issues, IDPF must be composed with some interactive mechanism for ensuring the IDPF outputs are well-formed. Mastic uses the VIDPF of [MST24] for this purpose, which endows IDPF with the following properties:

1. One-hot Verifiability: There is at most one prefix of each length whose value under f is non-zero. In particular, the output shares at each level are additive shares of a one-hot vector.

2. Path Verifiability: The One-hot Verifiability property alone is not sufficient to guarantee that the keys are well-formed. The Aggregators still need to verify that: a) the non-zero output values are across a single path in the tree, and b) the value of the root node is consistently propagated down the VIDPF tree. For example, if the root value is beta, then there is only a single path from root to the leaves with non-zero values, and all such values equal beta.

Below we describe the syntax of VIDPF; in Section 7 we specify the concrete construction of [MST24].

A concrete Vidpf defines the types and constants enumerated in Table 1. In addition, it implements the following methods:

* Vidpf.gen(alpha: Unsigned, beta: list[Vidpf.Field], binder: bytes, rand: bytes) -> tuple[PublicShare, list[bytes]] is the randomized key generation algorithm. (rand denotes the random bytes consumed by the algorithm.) Its inputs are the VIDPF index alpha (defined the same way as "IDPF index" in Section 8 of [VDAF]), the output value beta, and a binder string. The value of alpha MUST be in range [0, 2^Vidpf.BITS); and len(rand) MUST be Vidpf.RAND_SIZE. The outputs are the public share and the list of key shares, one for each Aggregator. The length of each key share MUST be Vidpf.KEY_SIZE.

- Vidpf.eval(agg_id: Unsigned, public_share: Vidpf.PublicShare, key_share: bytes, level: Unsigned, prefixes: tuple[Unsigned, ...], binder: bytes) -> tuple[list[Vidpf.Field], bytes] is the deterministic key evaluation algorithm. It takes as input the Aggregator ID (which MUST be in range [0, Vidpf.SHARES), the public share, the Aggregator’s key share, the VIDPF level (defined the same way as "IDPF level" in Section 8 of [VDAF]), the list of prefixes to evaluate, and a binder string. Its outputs are the VIDPF output share and the VIDPF proof.

The verifiability properties are guaranteed as long as each Aggregator computes the same VIDPF proof. Note that One-hot Verifiability and Path Verifiability are not sufficient to ensure robustness of Mastic; we will also need to ensure that the beta chosen by the Client is "in range". We will rely on FLPs (Section 3.3) for this purpose. ([MST24] describe a simple range(2) check, but we would like more sophisticated range checks for Mastic.)

Note that Vidpf is less general than Idpf as defined Section 8 of [VDAF] in that beta value is the same for each level of the tree. This constraint is necessary for Path Verifiability.

Parameter	Description
SHARES	Number of VIDPF keys output by VIDPF-key generator
BITS	Length in bits of each input string
VALUE_LEN	Number of field elements of each output value
RAND_SIZE	Size of the random string consumed by the VIDPF-key generator. Equal to twice the XOF’s seed size.
KEY_SIZE	Size in bytes of each VIDPF key
Field	Implementation of Field (Section 3.1) used for each value
PublicShare	Type of the VIDPF public share

Table 1: Constants and types defined by a concrete VIDPF.

4. Definition of Mastic

> NOTE We are pretty confident about the overall structure of the VDAF, but there are some details to work out and security analysis to do. In the meantime, check out the current reference implementation at https://github.com/jimouris/draft-mouris-cfrg-mastic/tree/main/poc.

This section describes Mastic, a VDAF suitable for a plethora of aggregation functions including sum, mean, histograms, heavy hitters, weighted heavy-hitters (see Section 5.1), attribute-based metrics (see Section 5.2), linear regression and more. Mastic allows computing functions _à la_ Prio3 VDAF Section 7 of [VDAF].

The core component of Mastic is a VIDPF as defined in Section 3.5. VIDPFs inherently have the "one-hot verifiability" property, meaning that in each level of the tree there exists at most one non-zero value. To guarantee that the Client's input is well-formed, Mastic first verifies that the Client measurement is valid at the root level using an FLP, and then, it ensures that this valid measurement is propagated correctly down the tree using the one-hot verifiability and the path verifiability properties. Note that Mastic allows the measurement to be of any type that can be verified by an arithmetic circuit, not just a counter. For instance, the measurement can be a tuple of values, a string, a secret number within a public range, etc.

As described in Section 2, each Client input consists of two components, which we denote alpha and beta. At a high level, the Client generates VIDPF keys that encodes alpha and beta and an FLP for the validity of beta. Then the Client sends one VIDPF key to each Aggregator and also publishes the VIDPF public share. FLPs for certain validity functions, including most range proofs, rely on the establishment of shared random coins (joint randomness) between the Client and all Aggregators. When it is necessary for the Client to generate joint randomness, it includes generator seeds in its shares for each Aggregator, and the Aggregators confirm that they have derived the same joint randomness during the FLP verification process.

The Aggregators agree on an initial set of level-bit strings, where level < BITS. We refer to these strings as "candidate prefixes". They evaluate their VIDPF key shares at each prefix in this set, to obtain an additive share of the VIDPF output.

Mastic uses a combination of techniques to certify the validity of this output.

1. First, the Aggregators exchange VIDPF proofs. If they are equal, then this implies One-hot Verifiability and Path Verifiability as described in Section 3.5. One-hotness ensures that the VIDPF output contains beta at most once (and every other output is 0). Path Verifiability implies that, if the previous level contained a non-zero value, then it is the same value as the current level.

2. Second, the Aggregators interactively verify the FLP (Section 3.3) to assert that beta is valid. We instantiate the FLP with FlpGeneric from Section 7.3 of [VDAF], which defines validity via an arithmetic circuit (Section 7.3.2 of [VDAF]) evaluated over (shares of) beta: if the output of the circuit is 0, then the value is said to be "valid"; otherwise it is "invalid".

   If none of the candidate prefixes are a prefix of alpha, then the VIDPF output shares will not contain any shares of beta. Moreover, VIDPF as specified in Section 7 does not as specified permit evaluation at the root of the VIDPF tree. Instead, each Aggregator computes a share of beta by evaluating the VIDPF tree at prefixes 0 and 1 and level == 0 and adding them up. One-hot Verifiability and Path Verifiability imply that the sum is equal to the Aggregator’s share of beta.

   > CP: An alternative way to spell this is to say that VIDPF evaluation outputs a share of beta, which is what our current API does in the reference code.

The aggregate result is obtained by summing up the encoded measurement shares for each prefix and computing some function of the sum. The aggregation parameter contains the level and the set of candidate prefixes.

The Aggregators send their aggregate shares to the Collector, who unshards them to recover the results for each candidate prefix.

4.1. Sharding

NOTE to be specified in full detail.

4.2. Preparation

NOTE to be specified in full detail.

4.3. Validity of Aggregation Parameters

NOTE to be specified in full detail.

4.4. Aggregation

NOTE to be specified in full detail.

4.5. Unsharding

NOTE to be specified in full detail.

5. Modes of Operation for Mastic

5.1. Weighted Heavy-Hitters

See Section 1.1.1 for a motivating application and example_weighted_heavy_hitters_mode() in the reference implementation for an end-to-end example.

The primary use case for Mastic is a variant of the heavy-hitters problem, in which the prefix counts are replaced with a notion of weight that is specific to some application. For example, when measuring the performance of an ad campaign, it is useful to learn not only which ads led to purchases, but how much money was spent.

To support this use case, we view the Client's alpha value as its measurement and the beta value as the measurement's "weight". The range of valid values for beta are therefore determined by the FLP with which Mastic is instantiated. Concretely, validity of beta is expressed by a validity circuit (Section 7.3.2 of [VDAF]).

To compute the weighted heavy-hitters, the Collector and Aggregators proceed as described in Section 8 of [VDAF], except that the threshold represents a minimum weight rather than a minimum count. In addition:

1. The Aggregators MUST perform the range check (i.e., verify the FLP) at the first round of aggregation and remove any invalid reports before proceeding.

2. The level at which the reports are Aggregated MUST be strictly increasing.

5.1.1. Different Thresholds

For an end-to-end example, see example_weighted_heavy_hitters_mode_with_different_thresholds() in the reference implementation.

So far, we have assumed that there is a single threshold for determining which prefixes are "heavy". However, we can easily extend this to have different thresholds for different prefixes. There exist use-cases where prefixes starting with "000" may be significantly more popular than prefixes starting with "111". Setting a low threshold may result in an overwhelmingly big set of heavy hitters starting with "000", while setting a high threshold might prune anything starting with "111". Consider the following examples:

1. Popular URLs: a.example.com receives a massive amount of traffic whereas b.example.com may have lower traffic. To identify heavy-hitting search queries on a.example.com, the Aggregators should set a high threshold, while queries with different domain prefixes may require lower thresholds to be considered popular.

2. E-commerce: Grocery items are essential and have a high volume of sales. In contrast, electronics, though popular, usually come with a higher price compared to groceries. Meanwhile, luxury items command significantly higher prices but generally experience lower sales volumes. To identify heavy-hitting grocery items on an e-commerce website, Aggregators could use different threshold for each of these categories. These thresholds are set to ensure that only the top-selling grocery items qualify as heavy hitters while electronics and luxury items are also considered heavy hitters on their own categories.

To tackle this, Mastic can allow different prefixes having different thresholds. When a specific prefix does not have an associated threshold, we first search if any of its prefixes has a specified threshold, otherwise we use a default threshold. For example, if the Aggregators have set the thresholds to be {"000": 10, "111": 2, "default": 5} and the search for prefix "01", then threshold 5 should be used. However, if the Aggregators search for prefix "11101", then threshold 2 should be used.

5.2. Attribute-based Metrics

See Section 1.1.2 for a motivating application and example_attribute_based_metrics_mode() in the reference implementation for an end-to-end example.

In this mode of operation, we take the beta value to be the Client's measurement and alpha to be an arbitrary "attribute". For a given sequence of attributes, the goal of the Collector is to aggregate the measurements that share the same attribute. This provides functionality similar to Prio3 [VDAF], except that the aggregate is partitioned by Clients who share some property. For example, the attribute might encode the Client's user agent [RFC9110].

Mastic requires each alpha to have the same length (Vidpf.BITS). Thus, it is necessary for each application to choose a scheme for encoding attributes as fixed-length strings. The following scheme is RECOMMENDED. Choose a cryptographically secure hash function, such as SHA256 [SHS], compute the hash of the Client's input string, and interpret each bit of the hash as a bit of the VIDPF index. [CP: Are we comfortable recommending truncating the hash? Collisions aren't so bad since the Client can just lie about alpha anyway. The main thing is to pick a value for BITS that is large enough to avoid accidental collisions.]

The Aggregators MAY aggregate a report any number times, but:

1. They MUST perform the range check (i.e., verify the FLP) the first time the reports are aggregated and remove any invalid reports before aggregating again.

2. The aggregation parameter MUST specify the last level of the VIDPF tree (i.e., level MUST be Vidpf.BITS-1).

> OPEN ISSUE Figure out if these requirements are strict enough. We may need to tighten aggregation parameter validity if we find out that aggregating at the same level more than once is not safe.

5.3. Plain Heavy-Hitters with VIDPF-Proof Aggregation

> NOTE to be specified in full detail. Proof aggregation is not yet implemented by the reference code.

The total communication cost of using Mastic (or Poplar1 [VDAF]) for heavy hitters is O(num_measurements * Vidpf.BITS) bits exchanged between the Aggregators, where num_measurements is the number of reports being aggregated. For plain heavy-hitters, this can be reduced to O(Vidpf.BITS) in the best case.

The idea is to take advantage of the feature of VIDPF evaluation whereby the Aggregators compute identical VIDPF proofs if and only if the report is valid. This allows the proofs themselves to be aggregated: if each report in a batch of reports is valid, then the hash of their proofs will be equal as well; on the other hand, if one report is invalid, then the hash of the proofs will not be equal.

To facilitate isolation of the invalid report(s), the proof strings are arranged into a Merkle tree. During aggregation, the Aggregators interactively traverse the tree to detect the subtree(s) containing invalid reports and remove them from the batch.

> OPEN ISSUE Decide if we should spell this out in greater detail. This feature is not compatible with [DAP]; if we wanted to extend DAP to support this, then we'd need to specify the wire format of the messages exchanged between the Aggregators.

In the worst case, isolating invalid reports requires O(num_measurements * Vidpf.BITS) bits of communication and many Vidpf.BITS rounds of communication between the Aggregators. However, this behavior would only be observed under attack conditions in which the vast majority of Clients are malicious.

In the simple case where the beta value is a constant (e.g., 1) we can replace the FLP check with a simpler check. FLPs are not compatible with proof aggregation the way VIDPFs are. In order to perform the range check without FLPs, we use an extension of VIDPF described by [MST24]. The high-level idea here is that the Aggregators can evaluate the empty string and verify that they have shares of the constant beta. Next, as described in Section 4, we use the "one-hot verifiability" and "path verifiability" checks to verify that each level is non-zero at only a single point and that the same constant beta is propagated down the tree correctly. Note that this trick is not suitable for weighted heavy-hitters, since it expects that each beta value is constant (e.g., 1).

> OPEN ISSUE Proof aggregation could work with plain Mastic, but we would need to check the FLPs at the first round of aggregation, leading to best-case communication cost would be O(num_measurements + Vidpf.BITS). This would be OK, but we would still want to support a mode for plain heavy-hitters that is as good as we can get.
>
> One idea is to always do the PLASMA 0/1 check alongside the FLP. This would be useful for another reason: Usually FLP decoding requires num_measurements as a parameter. We currently don't support this because we currently don't have a pure counter as part of the VIDPF output.

6. Robustness Against a Malicious Aggregator

Next, we describe an enhancement that allows Mastic to achieve robustness in the presence of a malicious Aggregator. The two-party Mastic (as well as Poplar1) is susceptible to additive attacks by a malicious Aggregator. In more detail, if one of the Aggregators starts acting maliciously, they can arbitrarily add to the aggregation result (simply by adding to their own aggregation shares) without the honest Aggregator noticing.

We can solve this problem in Mastic by using a technique from [MST24] that lifts the two-party semi-honest secure PLASMA to the three-party maliciously secure setting. Rather than having two Aggregators as in the previous setting, this flavor involves three Aggregators, where every pair of Aggregators communicate over a different channel. In essence, each pair of Aggregators will run one session of the VDAF with unique randomness but on the same Client measurement. The following changes are necessary:

1. The Client needs to generate three pairs of VIDPF keys all corresponding to the same alpha and beta values. We represent the keys based on the session as follows:

    1. Session 0 (between Aggregators 0 and 1): key_01, key_10

    2. Session 1 (between Aggregators 1 and 2): key_12, key_21

    3. Session 2 (between Aggregators 2 and 0): key_20, key_02

    Each pair of Aggregators cannot check that the Client input is consistent across two sessions without the involvement of the third Aggregator. To address this, we let two Aggregators (i.e., Aggregators 0 and 1) to run all three sessions so that they can check that the Client input is consistent across three sessions. The third Aggregator (i.e., Aggregator 2) is involved as an attestator in two of the sessions. The check involves field addition and subtraction and then hash comparisons.

2. The Client sends the following keys to the Aggregators:

    1. Aggregator 0 receives: key_01, key_02, and key_21

    2. Aggregator 1 receives: key_10, key_12, and key_20

    3. Aggregator 2 receives: key_21 and key_20

3. The Aggregators need to verify that the Client's input is consistent across the different sessions (i.e., that all the keys correspond to the same alpha and beta values). Aggregators 0 and 1 check that:

    1. Their output shares of Session 0 minus their output shares of Session 1 are shares of zero

    2. Their output shares of Session 1 minus their output shares of Session 2 are shares of zero.

    The subtraction is a local operation and verifying that two Aggregators possess a sharing of zero requires exchanging one hash.

Using a third Aggregator, we can lift the security of Mastic from the semi-honest setting to malicious security. While more complex to implement than 2-party Mastic, this mode allows achieves both privacy and robustness against a malicious Aggregator.

> NOTE to be specified in full detail.

7. Definition of Vidpf

The construction of [MST24] builds on techniques from [CP22] to lift an IDPF to a VIDPF with the properties described in Section 3.5. Instead of a 2-round "secure sketch" MPC like that of Poplar1, the scheme relies on hashing.

TODO(jimouris) Add an overview.

> NOTE To be specified. The design is based on VIDPF from [MST24]. https://github.com/jimouris/draft-mouris-cfrg-mastic/tree/main/poc for the reference implementation.

8. Security Considerations

A security analysis of Mastic is provided in [MPDST24].

9. IANA Considerations

> NOTE to be specified.

10. References

10.1. Normative References

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

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

[VDAF] Barnes, R., Cook, D., Patton, C., and P. Schoppmann, "Verifiable Distributed Aggregation Functions", Work in Progress, Internet-Draft, draft-irtf-cfrg-vdaf-08, 20 November 2023, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-vdaf-08>.

10.2. Informative References

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

[CP22] Leo de Castro and Antigoni Polychroniadou, "Lightweight, Maliciously Secure Verifiable Function Secret Sharing", EUROCRYPT 2022 , 2022, <https://iacr.org/cryptodb/data/paper.php?pubkey=31935>.

[DAP] Geoghegan, T., Patton, C., Rescorla, E., and C. A. Wood, "Distributed Aggregation Protocol for Privacy Preserving Measurement", Work in Progress, Internet-Draft, draft-ietf-ppm-dap-07, 14 September 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-ppm-dap-07>.

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

[MPDST24] Dimitris Mouris, Christopher Patton, Hannah Davis, Pratik Sarkar, and Nektarios Georgios Tsoutsos, "Mastic: Private Weighted Heavy-Hitters and Attribute-Based Metrics", 2024, <https://ia.cr/2024/221>.

[MST24] Dimitris Mouris, Pratik Sarkar, and Nektarios Georgios Tsoutsos, "PLASMA: Private, Lightweight Aggregated Statistics against Malicious Adversaries", PETS 2024 , 2024, <https://ia.cr/2023/080>.

[RFC9110] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <https://www.rfc-editor.org/rfc/rfc9110>.

[SHS] Dang, Q., "Secure Hash Standard", National Institute of Standards and Technology, DOI 10.6028/nist.fips.180-4, July 2015, <https://doi.org/10.6028/nist.fips.180-4>.

[W3C23] W3C Working Group, "Network Error Logging", 2023, <https://www.w3.org/TR/network-error-logging>.

Acknowledgments

NOTE to be specified.

Authors' Addresses

Hannah Davis
Seagate
Email: hannah.e.davis@seagate.com

Dimitris Mouris
Nillion
Email: dimitris@nillion.com

Christopher Patton
Cloudflare
Email: chrispatton+ietf@gmail.com

Pratik Sarkar
Supra Research
Email: pratik93@bu.edu

Nektarios G. Tsoutsos
University of Delaware
Email: tsoutsos@udel.edu