Mastic: Private Weighted Heavy-Hitters and Attribute-Based Metrics

Dimitris Mouris[1⋆], Christopher Patton[2], Hannah Davis[3], Pratik Sarkar[4], and Nektarios G. Tsoutsos[5]

[1] Nillion & University of Delaware
dimitris@nillion.com

[2] Cloudflare
cpatton@cloudflare.com

[3] Seagate
hannahedavis@protonmail.com

[4] Supra Research
pratik93@bu.edu

[5] University of Delaware
tsoutsos@udel.edu

Abstract. Insight into user experience and behavior is critical to the success of large software systems and web services. Yet gaining such insights, while preserving user privacy, is a significant challenge. Recent advancements in multi-party computation have made it practical to compute verifiable aggregates over secret shared data. One important use case for these protocols is heavy hitters, where the servers compute the most popular inputs held by the users without learning the inputs themselves. The Poplar protocol (IEEE S&P 2021) focuses on this use case, but cannot support other aggregation tasks. Another such protocol, Prio (NSDI 2017), supports a wider variety of statistics but is unsuitable for heavy hitters.

We introduce Mastic, a flexible protocol for private and verifiable general-purpose statistics based on function secret sharing and zero-knowledge proofs on secret shared data. Mastic is the first to solve the more general problem of *weighted* heavy-hitters, enabling new use cases, not supported by Prio or Poplar. In addition, Mastic allows grouping general-purpose metrics by user attributes, such as their geographic location or browser version, without sacrificing privacy or incurring high-performance costs, which is a major improvement over Prio. We demonstrate Mastic's benefits with two real-world applications, private network error logging and browser telemetry, and compare our protocol with Prio and Poplar on a wide area network. Overall, we report over one order of magnitude performance improvement over Poplar for heavy hitters and $1.5 - 2\times$ improvement over Prio for attribute-based metrics.

Keywords: Function secret sharing, histograms, heavy hitters, privacy-enhancing technologies, secure multiparty computation

1 Introduction

Critically important to the success of today's digital technology is the ability to gain insight into user behavior. Web browsers, operating systems, and web services collect telemetry to detect performance issues, bugs, and security vulnerabilities [Hol23]; advertisers track ad impressions to make sense of which ads drive revenue [Pri, TNB+10, EPK14, MMT+24]; and AI models are trained on user data for specific purposes, like detecting landmarks in photos [App23] or classifying malicious behavior [Clob].

Enabling such use cases requires collecting privacy-sensitive measurements from users. However, often the measurements are not consumed directly, but only in some aggregated form, such as a summary statistic (e.g., mean, median, or standard deviation) [CGB17], some lossy, probabilistic data structure (Bloom filter [Blo70]

⋆ Work partially performed at the University of Delaware and continued at Nillion.

or count-min sketch [CM04]), or machine learning model (linear regression or gradient descent) [DEF+19]. In these situations, it is desirable to collect only what the application needs – the *aggregate* – and avoid computing on the plaintext measurements directly to preserve user privacy.

Many in the tech industry are investing in *multi-party computation (MPC)* to help address this problem [Hol23, Int, Cloa, App23, MPC23, Tit22] as it allows multiple parties to jointly compute a function on private inputs. Much of this work revolves around a special class of lightweight and highly parallelizable MPC schemes known as *verifiable, distributed aggregation functions (VDAFs)* [DPRS23, BCPP22]. In VDAFs, the computation of the aggregate is delegated to a small number of servers – typically two. Measurements are secret shared (i.e., cryptographically split) and uploaded to the servers such that no one server sees any measurement in the clear. In addition, the computation is *verifiable* in the sense that the servers are guaranteed to compute an aggregate of only valid measurements.[6] The validity of the measurement depends on the application: e.g., each measurement might be a bit (0 or 1) and the aggregate result would be the frequency of these binary outcomes; each measurement might fall within a predetermined integer range and the aggregate would be the sum of the integers; or each measurement might be a one-hot vector (i.e., everywhere zero except a single one) and the aggregate would be a histogram computed from the sum of the vectors.

In this work, we present *Mastic*, a new VDAF for solving a generalization of the private *heavy hitters* problem, as well as efficient private aggregation based on client attributes. First, in the heavy hitters problem, each client $\mathcal{C}_i$ holds a private input $\alpha \in \{0,1\}^n$ and the servers' goal is to compute the subset of inputs held by at least T clients for some target *threshold* T. For example, if each α represents a website visited by a user, then the T-*heavy-hitters* is the set of sites visited by at least T users. Mastic solves the *weighted* version of this problem in which each input α has a corresponding *weight* β, and the servers' goal is to compute the subset of inputs for which (some function of) the sum of the weights exceeds T. For example, if each input is a site visited by a user, then the weight might represent the time spent on the site by that user; and the T-*weighted-heavy-hitters* would represent the sites with the highest engagement (i.e., time spent on the site).

We are also interested in a special case of this functionality, which we call *attribute-based metrics*. Here the goal is to break down the aggregates (mean, median, standard deviation, and so on) based on client attributes. For example, Mastic enables collecting statistics on website loading times, broken down by browser version or client location.

Both of these tasks (*weighted heavy hitters* and *attribute-based metrics*) are aligned with the goals of IETF [IET], which is in the process of standardizing cryptographic techniques for privacy-preserving measurement. Notably, existing solutions (like [BBC+19, BBC+21, MST24]) do not provide these functionalities. Therefore, the main goal of Mastic is to fill this gap, with support for real-world applications, such as network error logging and browser telemetry.

Background & Motivation. VDAFs are being integrated into a variety of real-world applications as we speak. One of the keys to the success of these schemes is their flexibility. The canonical example is Prio, first proposed by Corrigan-Gibbs and Boneh [CGB17] and now undergoing standardization at IETF [BCPP22]. Prio can compute any aggregation function $F(m_1, \ldots, m_{\mathsf{N}})$ that can be represented as the sum of (some encoding of) the measurements $m_1, \ldots, m_{\mathsf{N}}$. Validity is then defined by an arithmetic circuit evaluated over each encoded measurement, giving Prio the flexibility of modern zero-knowledge proof systems. This flexibility is enabled by Prio's use of *zero-knowledge proofs on secret shared data* [BBC+19], which enable servers to privately validate that their shares of each m_i sum up to a valid measurement.

Of course, many private aggregation problems do not fit cleanly into this rubric, the most important of which for IETF is (weighted) heavy hitters. Poplar [BBC+21] was the first solution for plain heavy hitters with enough scalability for use cases of practical interest. Poplar is based on the *function secret sharing* [BGI15] paradigm, where each client produces secret shares of some function f such that the servers can compute secret shares of $f(x)$ for a chosen input x with minimal ($\mathcal{O}(|x|)$) communication overhead. In particular, the authors construct a secret sharing of a function $f_{\alpha,\beta}$ for which $f_{\alpha,\beta}(x) = \beta$ for each *prefix* x of bitstring α

[6] In VDAFs, verifiability refers to the ability of the servers to assert that the result was computed from valid inputs – not to be confused with the notion of publicly verifiable MPC [BDO14].

and $f_{\alpha,\beta}(x) = 0$ otherwise. Given such a scheme called an *incremental distributed point function (IDPF)*, the servers can count how many of the inputs begin with a given candidate prefix.

Given the practical importance of heavy hitters, Poplar is also being considered for standardization at IETF [BCPP22]. However, from an engineering perspective, it has some pitfalls. First, to verify that their shares of $f_{\alpha,\beta}(x)$ for each candidate prefix x add up to a valid value, the servers interactively compute an "arithmetic sketch" [BBC+23]) over their shares. The main drawback of this approach is that it requires two rounds of communication, whereas many VDAFs, like Prio, require just one. More than one round inhibits performance (especially over wide-area networks) and also requires the servers to keep state, which adds complexity to the protocol [GPP+21].

This pitfall is addressed by the *verifiable IDPF (VIDPF)* construction of PLASMA [MST24] (described in Section 2.3). The scheme relies on efficient hashing techniques to provide a much stronger and intuitive verifiability property (cf. [MST24, Section 3.4.1]), with much lower communication overhead. However, PLASMA focuses on a 3-server setting, while our focus is on the 2-server setting, like Poplar, as it is way more practical for realistic deployments.

1.1 Our Contribution

The primary goal of this paper is to devise a protocol that overcomes both Poplar's and Prio's design pitfalls while offering more flexible statistics. Our contributions are as follows:

1. We introduce a protocol for weighted heavy hitters and attribute-based metrics by adopting a natural composition of VIDPFs [MST24] and zero-knowledge proofs on secret-shared data [BBC+19]. The result is more flexible than Prio, and more versatile and easier to implement than Poplar.
2. We prove Mastic is secure in the same threat model as Poplar. Specifically, Mastic ensures 1) *privacy of the inputs* in the presence of malicious clients and one malicious server and 2) *robustness of the output* in the presence of malicious clients (i.e., servers correctly compute aggregate of valid measurements only).
3. We implement Mastic and demonstrate its suitability for two real-life applications of interest to IETF [IET], namely *Network Error Logging (NEL)* [W3C23] and *Browser Telemetry* [Hol23]. In the first, we propose a privacy-preserving version of NEL, a tool that provides telemetry crucial for detecting and diagnosing connectivity issues between clients and servers on the internet. In the second, we introduce attribute-based browser telemetry with improved usability compared to Prio without sacrificing privacy.

1.2 Related Works

We discuss related works for private heavy hitters and statistics along with their models and the applications they enable.

Private Statistics from Distributed Point Functions. Poplar [BBC+21] proposed IDPFs to address the problem of heavy hitters by computing the total number of private client inputs that begin with a given prefix. This incremental property allowed efficient evaluation of strings based on prefixes which was not possible with DPFs [BGI15]. Poplar's servers ensure that the client measurements are valid based on a 2-round MPC. PLASMA [MST24] improved on IDPFs by proposing the Verifiable IDPF (VIDPF) primitive that combined IDPFs with the verifiability property of [dCP22]. As a result, PLASMA relies solely on hashing rather than MPC operations to assert client measurement validity. Notably, PLASMA ensures robustness when one of the three servers colludes with the malicious clients. The concurrent work of Doplar [DPRS23] introduced a construction they also call a "VIDPF", but the construction is less efficient than that of [MST24]. Like Poplar and PLASMA, Doplar only solves the plain heavy hitters problem.

Private Statistics from Differential Privacy. Differential Privacy (DP)-based techniques [QYY+16, ZKM+20] have been used to solve the problem of heavy hitters. These works rely on a single data collecting server and on local DP and randomized response. Thus, they offer a data utility/privacy trade-off as to achieve higher data utility they end up leaking some information about the client measurements to the server, while other works that focus on stronger privacy guarantees require at least two not-colluding servers.

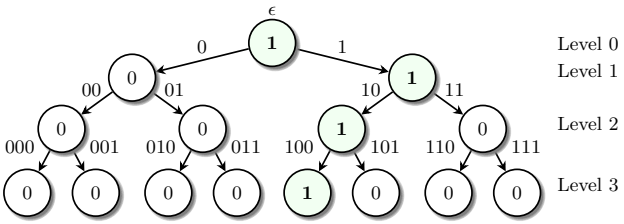

Fig. 1: IDPF tree for input $\alpha = 100$ and weight $\beta = 1$.

The work of Anderson et al. [ACD+21] relies on oblivious shuffling to unlink reports from the clients that generated them, combines them with artificial data to add DP noise, and finally computes approximate statistics over the shuffled data. Finally, the work of Bell et al. [BGG+22] computes sparse histograms by relying on two servers but reveals DP views to the aggregators. In contrast with all aforementioned works, Mastic computes the more elaborate problem of weighted heavy hitters while focusing on exact statistics and not revealing anything to the servers about individual client measurements. Additionally, as demonstrated in [BBC+21, MST24, BCPP22], DPF-based protocols like Poplar, PLASMA, and Mastic are compatible with DP and can be easily modified to report differentially private aggregations.

Statistics from (Non-DPF) MPC. The work of [BK21] utilizes general-purpose MPC [CGH+18, Kel20] for computing heavy hitters but results in prohibitively expensive solutions. Asharov et al. [AHI+22] propose an MPC sorting-based protocol for heavy hitters that improves on Poplar and PLASMA in terms of performance but requires significantly higher communication between the servers, which in many scenarios (especially over WAN) is not practical. Similarly, Vogue [JKK+22] relies on MPC sorting but inherently has high server-to-server communication. Both of these protocols specifically solve the plain heavy hitters problem and have not been extended to weighted heavy hitters. More specifically, the client votes for their input α by sharing α with the servers. The servers sort the shared inputs, and then in the aggregation phase, they consider a point to be heavy-hitting if there are more than a threshold number of votes for it. In the weighted version, the clients also have to share their weights for each point α, and the servers need to consider that during aggregation, which is non-trivial given the current protocols of [AHI+22, JKK+22]. Note that the weighted heavy hitter problem is not related to weighted MPC [GJM+23].

2 Preliminaries

2.1 Notation

Fix a prime order field $\mathbb{F}$. We use $[\![\cdot]\!]$ to denote additive secret shares: if $x \in \mathbb{F}^m$ for some m, then $[\![x]\!]_0$ and $[\![x]\!]_1$ denote shares of x for which $x = [\![x]\!]_0 + [\![x]\!]_1$. We use $[n]$ to denote the set of integers $\{1, \ldots, n\}$. We use ":=" for assignment, "$\xleftarrow{\$} D$" for sampling unifromly from a finite set D, and "$\|$" for concatenation of strings. For bitstring s, we denote by $s|^n$ the n-bit prefix of s. We use square brackets to denote vectors, e.g., $v = [v_1, \ldots, v_n]$ is the length-n vector with elements $v_1, \ldots, v_n$. For any two sets D and R, we denote by $\mathcal{AF}(D, R)$ the set of all functions with domain D and range R. Our protocol involves a large number of *clients* and a small number of *servers*, which we also call *aggregators*.

2.2 Distributed Point Functions

Mastic is based on function secret sharing [BGI15]. Secret sharing a function f allows the share holders to locally compute $[\![f(x)]\!]_0$, $[\![f(x)]\!]_1$ for a given input x without revealing anything about f. A *distributed point function (DPF)* [BGI15] is a special case in which f is defined as follows. Fix $\alpha, \beta \in \{0,1\}^n \times \mathbb{F}^m$ for some n, m: we call f the *point function* for α, β if $x = \alpha$ implies $f(x) = \beta$ and $x \neq \alpha$ implies $f(x) = 0^m$. Mastic requires two additional properties for DPFs.

Incrementality. The *incremental DPF (IDPF)* of [BBC+21] secret shares an *incremental* point function, where $f(p) = \beta$ for any *prefix* p of α.[7] This allows for richer statistics, including (weighted) heavy hitters. Each share holder has a share of the *prefix tree* for α, β. As illustrated in Fig. 1, a prefix tree is a complete binary tree whose nodes are labeled with output values, which we call *weights*. Each path in this tree corresponds to a unique prefix; when we evaluate the incremental point function f at prefix p, the output gives the weight of the node that we reach when we traverse path p. For example, in Fig. 1: $f(1) = \beta$; $f(10) = \beta$; and $f(101) = 0^m$.

Verifiability. In many applications (including Mastic), the DPF shares are generated by an untrusted client, e.g., in a web browser. Since the shares reveal nothing about the underlying function, it is trivial for a malicious client to craft malformed shares that evaluate to something other than an (incremental) point function, thus breaking the correctness of the application. To detect such attacks, Boneh et al. [BBC+21] use a two-round MPC protocol between the servers to verify that for a given level of the prefix tree, at most one of the outputs has a non-zero value. de Castro and Polychroniadou [dCP22]'s *verifiable* DPF ensures a similar property for the DPF's output. However, their approach is based on hashing, is lighter weight, and requires just one round of communication rather than two.

2.3 Verifiable IDPFs (VIDPFs)

Our starting point for Mastic is the *verifiable IDPF (VIDPF)* of Mouris et al. [MST24]. This scheme adapts the techniques of de Castro and Polychroniadou to the IDPF of Boneh et al. to allow the servers to verify the correctness of the computation without revealing the input or its weight.

For each node it traverses in its share of the prefix tree, each server computes a short bit string π^p, where p is the path to the node from the root. We refer to this string as the *VIDPF proof* (or simply *proof*) for prefix p. Each proof is computed from the proof of its parent node in such a way that, if each server computes the same proof for p, then there is at most one prefix of length $|p|$ that evaluates to a non-zero value.

A VIDPF is comprised of the following algorithms:[8]

- $\mathcal{V}.\mathsf{Gen}(\alpha, \beta) \to (\mathsf{pub}, \mathsf{key}_0, \mathsf{key}_1)$: The *key generation* takes an *input* $\alpha \in \{0,1\}^n$ and its *weight* $\beta \in \mathbb{F}^m$ and outputs the *keys* key_0 and key_1 and a *public share* pub.
- $\mathcal{V}.\mathsf{Eval}(\mathsf{key}_b, \mathsf{pub}, p, \mathsf{st}_b^{p'}, \pi_b^{p'}) \to (\llbracket y^p \rrbracket_b, \mathsf{st}_b^p, \pi_b^p)$: The *prefix evaluation algorithm* takes one of the keys and the public share and returns a share of a node of the prefix tree. It operates on a prefix $p \in \{0,1\}^{\leq n}$ and the *state* $\mathsf{st}_b^{p'}$ and *proof* $\pi_b^{p'}$ for a prefix p' of p. (Usually p' is the parent of p, i.e., $p = p' \parallel z$ for some bit z.) Its outputs are the state and proof for p and a share of the weight y^p associated with p.[9] The proof and state for the root ($p' = \epsilon$) are defined to be the ϵ.
- $\mathcal{V}.\mathsf{EvalRoot}(\mathsf{key}_b, \mathsf{pub}) \to \llbracket \beta \rrbracket_b$: The *root evaluation algorithm* takes in one of the keys and the public share and outputs the aggregator's share of β.

Since the specific VIDPF scheme we use proves its security in the random oracle model, we require it to define two sets Dom and Rng. Every security game using the scheme must begin by sampling a uniformly random function from the set of all functions with domain Dom and range Rng. It must then provide the adversary and Gen, Eval, and $\mathsf{EvalRoot}$ with oracle access to this function.

In Appendix A we define the properties of VIDPFs we need for Mastic. Briefly, $\mathcal{V}$ should be: *Correct*: when the client and aggregators are honest, the aggregators correctly evaluate shares of the prefix tree; *Verifiable:* when the aggregators are honest, malicious clients cannot construct a public share and keys $(\mathsf{pub}, \mathsf{key}_0, \mathsf{key}_1)$ for which the evaluation tree contains more than one non-zero node at any level $k \in [n]$; and *Private:* the information revealed to each aggregator reveals nothing about honest client's input.

[7] Boneh et al. [BBC+21] consider a slightly richer function parameterized by $\beta_1, \ldots, \beta_n$ where β_i is the output for the length-i prefix of α. For our purposes, it is sufficient for each output to be the same.

[8] There are a few differences between our syntax and [MST24]. The main one is the addition of the root evaluation algorithm, which Mastic uses (and is implicit in [MST24]). We also have a public share, which contains parts of the keys that are sent to both servers (namely, correction words [MST24, Fig. 14]).

[9] If p is a prefix of α, then $y^p = \beta$; otherwise $y^p = 0^m$.

Limitations. Observe that VIDPF already provides most of what we need for Mastic: using the same tree traversal and pruning strategy for PLASMA [MST24], modified slightly to use sums of weights rather than prefix counts (see Section 3 for details), we can compute the subset of inputs whose total weight exceeds the desired threshold. However, the degree of verifiability it provides is not enough for most applications, including ours (Section 4). In Section 3.2.2 we describe in detail how Mastic overcomes this gap.

2.4 Shared ZK Proofs

Zero-knowledge (ZK) proof systems enable a prover to demonstrate to a verifier that a value has some specific property in zero-knowledge, i.e., without revealing anything about the value to the verifier beyond its validity. The specific variant we use, from [BBC+19], operates on secret-shared inputs and, verification is distributed amongst the share holders. This proof system has been incorporated into the candidate standard for Prio [BCPP22]; here we devise a syntax for proofs on secret shared data that is suitable for both Prio and Mastic.

A *shared ZK proof* system defines three algorithms Prove, Query, and Decide. In practice, a client splits its input into two linear shares before providing them to Prove, which generates proofs for each aggregator. We require the existence of an algorithm $\mathsf{Extract}$ that extracts a single input from the shares; this eliminates ambiguity about the witness. The aggregators then run Query, exchange their intermediate verification strings, and call Decide to compute a final verdict.

Like VIDPFs, we define the security of shared ZK proofs in the random oracle model, so we equip the system with two sets Dom and Rng. Again, at the start of each security game involving shared ZK, we will sample a uniformly random function H from the set of all functions with domain Dom and range Rng. Then all three shared ZK algorithms and the adversary are given oracle access to H.

- $\mathcal{Z}.\mathsf{Prove}^{\mathrm{H}}(\llbracket x \rrbracket_0, \llbracket x \rrbracket_1) \to (\pi_0^{\mathsf{szk}}, \pi_1^{\mathsf{szk}}, \mathsf{nonce})$: Let x denote the private measurement, presumably in the language $\mathcal{L} \subseteq \mathbb{F}^m$ recognized by $\mathcal{Z}$. The *proof generation* algorithm takes linear shares $\mathsf{Extract}(\llbracket x \rrbracket_0, \llbracket x \rrbracket_1) = x$ of x and outputs a *partial proof* for each aggregator along with a nonce. This algorithm is run by each client.
- $\mathcal{Z}.\mathsf{Query}^{\mathrm{H}}(vk, \mathsf{nonce}, \llbracket x \rrbracket_b, \pi_b^{\mathsf{szk}}) \to (\mathsf{st}_b, \sigma_b)$: The *query generation algorithm* takes in the *verification key* ($vk \in \{0,1\}^{\mathsf{vkl}}$) shared by the aggregators, the nonce, one of the secret shares ($\llbracket x \rrbracket_b$), and the corresponding partial proof (π_b^{szk}). It returns the aggregator's state (st_b) and a *partial verifier* (σ_b). This algorithm is run by each aggregator.
- $\mathcal{Z}.\mathsf{Decide}^{\mathrm{H}}(\sigma_0, \sigma_1, \mathsf{st}) \to \mathsf{Accept}/\mathsf{Reject}$: Finally, the *decision algorithm* takes in the verifier shares and the state (st) of an aggregator. It outputs Accept if x was recognized as a member of $\mathcal{L}$ and Reject otherwise. This algorithm is run by each aggregator.

We define the security properties required for $\mathcal{Z}$ in Appendix B. Briefly, we require $\mathcal{Z}$ to be: *Complete*: when the client and aggregators are honest, the aggregators accept the measurement; *Sound:* when the aggregators are honest, invalid measurements are detected with high probability; and *Zero-knowledge:* when the client and at least one aggregator is honest, execution of the proof system reveals nothing about the measurement beyond its validity.

The shared ZK system $\mathcal{Z}$ can be instantiated from a *fully linear proof* system [DPRS23] similar to the candidate standard for Prio [BCPP22]. See Appendix B for details.

3 Weighted & Attribute-Based Metrics

This section describes the Mastic protocol, built from a VIDPF and shared ZK proof system. We focus the presentation on weighted heavy hitters and capture attribute-based metrics as a mode of operation for Mastic in Section 3.2.3. We begin by describing our threat and communication model.

3.1 Threat & Communication Model

There are many clients (thousands or millions), each of which holds a private *measurement* consisting of an *input* $\alpha \in \{0,1\}^n$ and its *weight* β. There are two aggregators who are responsible for gathering the private client measurements and aggregating them. One of these aggregators plays a special role: we call them the *leader* $\mathcal{S}_0$ and the other the *helper* $\mathcal{S}_1$. For the most part, the leader and helper perform the same computation except that the leader picks a shared *verification key* that is used for verifying the weights. No other cryptographic asset is required to execute the protocol (except those needed to establish secure channels).

Each client sends a single message to each aggregator that encodes a secret share of its measurement. Thereafter, the aggregators interact with one another to verify each pair of shares is a valid measurement and to compute the (weighted) heavy hitters. The clients do not participate in the protocol beyond sending their initial messages.

We make the following assumptions about the adversary. First, we assume that the adversary is fully malicious: it may deviate arbitrarily from the protocol. Second, the set of clients and aggregators is fixed for the adversary's attack. Third, multiple instances of the protocol may be executed simultaneously. Fourth, corruptions are static: the adversary chooses the set of parties it controls before beginning its attack. Finally, the adversary may eavesdrop on any communication channel except for those between each honest client and each honest aggregator (in practice, a secure channel will be established between them). We consider two security goals:

Privacy in the presence of malicious clients and a malicious aggregator. Our primary goal is that the attacker learns nothing more than the aggregates of the honest clients' measurements.[10] Particularly, the measurements themselves are not revealed to the adversary. We are interested in protecting the measurements from a malicious aggregator, but will further allow the attacker to corrupt some fraction of the clients.

Robustness in the presence of malicious clients. Malicious clients may attempt to disrupt the protocol by providing malformed inputs. Our goal is to prevent an attacker from forcing the aggregators to compute anything other than aggregate measurements submitted by honest clients and valid *partial* measurements submitted by malicious clients. (A valid partial measurement is of the form (α_i, β_i) where β_i is valid, i.e., $\beta_i \in \mathcal{L}$, and α_i is a string of length $\leq n$ bits instead of n bits; see Section 5 for details.)

We formalize these goals in Section 5. Note that privacy is not achievable if *both* aggregators are corrupted by the adversary. Likewise, we do not require robustness in the presence of a corrupt aggregator, as this appears to only be achievable at a significant cost: either by adding a third aggregator (where one out of three can be corrupt) as in PLASMA [MST24], or using verifiable computation in the two aggregator setting where the aggregators prove the correctness of their computation using a zero-knowledge proof [YSWW21]. General-purpose zero-knowledge proofs are prohibitively expensive for this application [YSWW21] since the ZK circuit (proven by each aggregator) would scale linearly with the number of clients.

3.2 The Mastic Protocol

Mastic is specified in Fig. 2 in terms of a VIDPF $\mathcal{V}$, a shared ZK proof system $\mathcal{Z}$, and a hash function H with domain Dom and range Rng. The input to the computation is the set $\{(\alpha_i, \beta_i)\}_{i \in [\mathsf{N}]}$ of client input/weight pairs; the output is the subset of α_i's that have the highest total weight. Its execution is associated with a *threshold* $\mathsf{T} \geq 0$ and a function $\mathsf{order} : \mathbb{F}^m \mapsto \mathbb{R}$ defining a total ordering of sums of weights.

3.2.1 Client Report Generation

Each client $\mathcal{C}_i$ holds a private measurement that comprises two parts: a bit-string α_i of length n and an associated weight β_i in some language $\mathcal{L} \subseteq \mathbb{F}^m$ of valid weights. The language $\mathcal{L}$ depends on the application. For example, we might want for β to be a bit (i.e., $\mathcal{L} = \{0,1\}$), an integer in a public range ($\mathcal{L} = [\mathsf{R}]$ for some

[10] The adversary learns not only the heavy hitters but intermediate aggregates while evaluating the prefix tree. This leakage is inherent to how IDPFs are used by Mastic and related works [BBC+21, MST24].

Π_{Mastic} for weighted heavy hitters

Client Computation:

Input: Each client $\mathcal{C}_i$ for $i \in [\mathsf{N}]$ holds measurement $(\alpha_i, \beta_i) \in (\{0,1\}^n, \mathcal{L})$ composed of an input α_i and its weight β_i.

1. $\mathcal{C}_i$ runs $(\mathsf{pub}_i, \mathsf{key}_{(i,0)}, \mathsf{key}_{(i,1)}) \coloneqq \mathcal{V}.\mathsf{Gen}(\alpha_i, \beta_i)$.
2. $\mathcal{C}_i$ runs $(\pi^{\mathsf{szk}}_{(i,0)}, \pi^{\mathsf{szk}}_{(i,1)}, \mathsf{nonce}_i) \coloneqq \mathcal{Z}.\mathsf{Prove}(\llbracket\beta_i\rrbracket_0, \llbracket\beta_i\rrbracket_1)$ where $\llbracket\beta_i\rrbracket_b \coloneqq \mathcal{V}.\mathsf{EvalRoot}(\mathsf{key}_{(i,b)}, \mathsf{pub}_i)$ for $b \in \{0,1\}$.
3. $\mathcal{C}_i$ sends report share $(\mathsf{nonce}_i, \mathsf{pub}_i, \mathsf{key}_{(i,b)}, \pi^{\mathsf{szk}}_{(i,b)})$ to $\mathcal{S}_b$ for each $b \in \{0,1\}$.

Aggregator Computation:

Input: The aggregators $\mathcal{S}_0$ and $\mathcal{S}_1$ start with a verification key $vk \in \{0,1\}^{\mathsf{vkl}}$ established out-of-band. Each sets $\mathsf{HH}^{\leq n} \coloneqq \{\mathsf{HH}^0, \mathsf{HH}^1, \ldots \mathsf{HH}^n\}$ as $\{\epsilon, \emptyset, \ldots, \emptyset\}$, the initial set of *candidate prefixes* for each level and sets $\mathsf{Reports} \coloneqq [\mathsf{N}]$, the initial set of *candidate reports*. Finally each initializes $(\llbracket y_i^{\epsilon}\rrbracket_b, \mathsf{st}^p_{(i,b)}, \pi^{\epsilon}_{(i,b)}) = (\perp, \epsilon, \epsilon)$ for each $i \in [\mathsf{N}]$.

1. **For each** client $i \in \mathsf{Reports}$: ▷ Weight check using $\mathcal{Z}$ at the root.
   a. Remove i from $\mathsf{Reports}$ if $(\mathsf{nonce}_i, \mathsf{pub}_i, \mathsf{key}_{(i,b)}, \pi^{\mathsf{szk}}_{(i,b)})$ does not follow the correct formatting. ▷ Input-Formatting check.
   b. $\mathcal{S}_b$ runs $(\mathsf{st}_b, \sigma_b) \coloneqq \mathcal{Z}.\mathsf{Query}(vk, \mathsf{nonce}_i, \llbracket\beta_i\rrbracket_b, \pi^{\mathsf{szk}}_{(i,b)})$, where $\llbracket\beta_i\rrbracket_b \coloneqq \mathcal{V}.\mathsf{EvalRoot}(\mathsf{key}_{(i,b)}, \mathsf{pub}_i)$.
   c. $\mathcal{S}_b$ sends σ_b to $\mathcal{S}_{1-b}$. If $\mathcal{Z}.\mathsf{Decide}(\sigma_0, \sigma_1, \mathsf{st}_b) \neq \mathsf{Accept}$, then $\mathcal{S}_b$ removes i from $\mathsf{Reports}$.
2. **For each** level $k \in [0, \ldots, n-1]$ and prefix $p \in \mathsf{HH}^k$:
   a. **For each** candidate report $i \in \mathsf{Reports}$: ▷ Path & One-hot Verifiability checks.
      i. Each $\mathcal{S}_b$ retrieves the state $(\llbracket y_i^p\rrbracket_b, \mathsf{st}^p_{(i,b)}, \pi^p_{(i,b)})$ from memory corresponding to prefix p and client $\mathcal{C}_i$.
      ii. Each $\mathcal{S}_b$ runs as $(\llbracket y_i^{\gamma}\rrbracket_b, \mathsf{st}^{\gamma}_{(i,b)}, \pi^{\gamma}_{(i,b)}) \coloneqq \mathcal{V}.\mathsf{Eval}(\mathsf{key}_{(i,b)}, \mathsf{pub}_i, \gamma, \mathsf{st}^p_{(i,b)}, \pi^p_{(i,b)})$ for each prefix $\gamma \in \{p \,\|\, 0, p \,\|\, 1\}$ and stores the results in memory.
      iii. The aggregators check that the output for prefix p is equal to the sum of the outputs on prefixes $p \,\|\, 0$ and $p \,\|\, 1$. To do so, each $\mathcal{S}_b$ computes $h^p_{(i,b)} \coloneqq (-1)^b \cdot (\llbracket y_i^p\rrbracket_b - \llbracket y_i^{p\|0}\rrbracket_b - \llbracket y_i^{p\|1}\rrbracket_b)$. ▷ Observe that $h^p_{(i,0)} = h^p_{(i,1)}$.
      iv. $\mathcal{S}_b$ accumulates its local state as $R^k_{(i,b)} \coloneqq \mathrm{H}\big(\big|\big|_{p\in\mathsf{HH}^k}(p, h^p_{(i,b)}, \pi^{p\|0}_{(i,b)}, \pi^{p\|1}_{(i,b)})\big)$. ▷ This is for all heavy-hitter prefixes.
      v. $\mathcal{S}_b$ sends $R^k_{(i,b)}$ to $\mathcal{S}_{1-b}$. If $R^k_{(i,0)} \neq R^k_{(i,1)}$, then $\mathcal{S}_b$ removes i from $\mathsf{Reports}$. ▷ One hash for each client.
   b. **For each** k-bit heavy-hitting prefix $p \in \mathsf{HH}^k$ the aggregators prune on $\gamma \in \{p \,\|\, 0, p \,\|\, 1\}$ as: ▷ Aggregation & Pruning.
      i. Each $\mathcal{S}_b$ accumulates $\llbracket\mathsf{weight}^{\gamma}\rrbracket_b \coloneqq \llbracket\mathsf{weight}^{\gamma}\rrbracket_b + \llbracket y_i^{\gamma}\rrbracket_b$. ▷ Each $\llbracket y_i^{\gamma}\rrbracket_b$ is a vector of field elements $\mathbb{F}^m$.
      ii. $\mathcal{S}_0$ and $\mathcal{S}_1$ recover $\mathsf{weight}^{\gamma} \coloneqq \llbracket\mathsf{weight}^{\gamma}\rrbracket_0 + \llbracket\mathsf{weight}^{\gamma}\rrbracket_1$. If $\mathsf{order}(\mathsf{weight}^{\gamma}) < \mathsf{T}$, then prune γ from the candidate prefix set. Otherwise, accumulate $\mathsf{HH}^{k+1} \coloneqq \mathsf{HH}^{k+1} \cup \{\gamma\}$. ▷ $\mathsf{order}(\cdot)$ is decided by the aggregators.
3. Finally, the servers output HH^n as the set of weighted T-heavy-hitters.

Fig. 2: Protocol Π_{Mastic} for $(\mathsf{T}, \mathsf{order})$-weighted-heavy-hitters built from VIDPF $\mathcal{V}$, shared ZK $\mathcal{Z}$, and hash function H.

R), or a one-hot vector ($\mathcal{L} = \{[1,0,\ldots,0], [0,1,0,\ldots,0], \ldots, [0,\ldots,0,1]\}$). The language must be agreed upon by the clients and aggregators before report generation and processing begins.

The client report generation phase is shown in Fig. 2 under "Client Computation". During this phase, each client first encodes its α_i, β_i using the VIDPF key generation algorithm that produces a pair of keys $\mathsf{key}_{(i,0)}, \mathsf{key}_{(i,1)}$ and a public share pub_i. Next, it generates a shared ZK proof that asserts to the aggregators that its private weight β_i is indeed a member of the language $\mathcal{L}$. To do this, the client passes the exact secret shares of β_i to proof generation algorithm that the aggregators will receive when evaluating the VIDPF at the root (i.e., the empty bit-string ϵ). The client computes these shares by invoking $\mathcal{V}.\mathsf{EvalRoot}$ twice, each

time with the inputs of each aggregator. The $\mathcal{Z}.\mathsf{Prove}$ algorithm returns the partial proofs $\pi^{\mathsf{szk}}_{(i,0)}, \pi^{\mathsf{szk}}_{(i,1)}$ as well as a nonce to the client. Finally, the client sends the nonce nonce_i, the public share pub_i, the VIDPF key $\mathsf{key}_{(i,b)}$, and the partial proof $\pi^{\mathsf{szk}}_{(i,b)}$ to aggregator each $\mathcal{S}_b$. After each client sends its messages, it goes offline.

3.2.2 Aggregator Computation

Next, we delve into the main phase of our protocol in Fig. 2, the "Aggregator Computation".

Initialization. The aggregators $\mathcal{S}_0$ and $\mathcal{S}_1$ must first agree on a verification key $vk \in \{0,1\}^{\mathsf{vkl}}$. For the sake of simplicity, we assume the leader $\mathcal{S}_0$ chooses this value unilaterally and sends it to the helper $\mathcal{S}_1$. This is sufficient for security, as long as the aggregators commit to this value before the protocol begins (see [DPRS23, Section 3.2]). They then initialize a set of sets $\mathsf{HH}^{\leq n} := \{\mathsf{HH}^0, \mathsf{HH}^1, \ldots \mathsf{HH}^n\}$ as $\{\epsilon, \emptyset, \ldots, \emptyset\}$, where n is the size of the clients' bit-strings. As the aggregators evaluate all n levels, they will start populating $\mathsf{HH}^{\leq n}$ with the weighted heavy hitter bit-strings. Finally, the aggregators initialize a list $\mathsf{Reports}$ that contains the *candidate reports* from the clients. Reports will be removed from this set if a validity check fails.

Verifying Client Inputs. The aggregators can now start generating secret shares of the client reports by evaluating the VIDPF keys of each client. It is crucial for the aggregators to verify that the client's input is well-formed in a privacy-preserving manner. There are three things to check: *Weight Check:* $[\![\beta_i]\!]_0 + [\![\beta_i]\!]_1 \in \mathcal{L}$, where $[\![\beta_i]\!]_0, [\![\beta_i]\!]_1$ are the aggregators' shares of β_i; *One-Hotness Check:* evaluating the keys of $\mathcal{C}_i$ on different prefixes *at the same level* should yield secret shares of a one-hot vector (i.e., only one pair of secret shares should correspond to β_i and the rest to zero); and *Path Check:* Each node along the α_i-path of the prefix tree has weight β_i. We emphasize that all the aforementioned checks need to be performed by the aggregators while keeping the measurement (α_i, β_i) private. If any of these checks are omitted, then a malicious client will be able to trick the aggregators and affect the robustness of the protocol (i.e., data poisoning attack).

We split these checks into two different categories; the checks we perform at the root level of the VIDPF tree and the checks we perform at any other level (including the leaves).

Root-Level Checks. The aggregators need to verify that the private β_i is indeed a valid weight (i.e., $\beta_i \in \mathcal{L}$, where $\mathcal{L}$ depends on the type of statistic). This is a two-fold process: First, the aggregators evaluate each client's VIDPF keys at the root level of the tree (i.e., the empty bit string ϵ) to get secret shares of β_i (i.e., $[\![\beta_i]\!]_0$ and $[\![\beta_i]\!]_1$). Each $\mathcal{S}_b$ acquires $[\![\beta_i]\!]_b$ by invoking $\mathcal{V}.\mathsf{EvalRoot}$ using the client's key $\mathsf{key}_{(i,b)}$ and the public share information pub_i. Next, each aggregator $\mathcal{S}_b$ performs a shared ZK query using their retrieved $[\![\beta_i]\!]_b$ share, the verification key vk shared by the aggregators, the nonce nonce_i associated with the client's report, and the partial proof $\pi^{\mathsf{szk}}_{(i,b)}$ generated by the client. Each $\mathcal{S}_b$ receives a state $\mathsf{st}_{i,b}$ and partial verifier $\sigma_{i,b}$. The aggregators exchange their partial verifiers and run the shared ZK decision algorithm to verify if the client's weight is valid. Finally, the aggregators remove from the candidate reports $\mathsf{Reports}$ each client $\mathcal{C}_i$ whose proof was not verified successfully.

This check is described in "Step 1." in "Aggregator Computation" in Fig. 2 and is performed for every client. Essentially, the shared ZK proof allows the aggregators to check that they have valid shares of a weight $\beta_i \in \mathcal{L}$ without reconstructing it. However, so far the aggregators have only verified that the client has submitted a report with a valid weight for the root of the VIDPF tree (i.e., the empty bit string). The next step is to verify that this weight is correctly propagated down the tree and compute the weighted heavy hitters.

Intermediate Levels and Leaves Checks. The protocol continues iteratively by processing one level at a time, starting from the two children of the root node (ϵ). The goal of the aggregators is to evaluate all the client reports on both children of the root node (namely $\epsilon \,\|\, 0$ and $\epsilon \,\|\, 1$), verify for each client that they have valid shares of these evaluations, aggregate them all together, and finally, only keep the bit strings whose aggregate weight is above the threshold T. For instance, if $0, 1$ are both heavy hitters at level one, then the aggregators will evaluate the children of both (i.e., 00, 01, 10, and 11) at level two. For each evaluation of $\mathcal{C}_i$'s report at each of these prefixes, the aggregators acquire secret shares of weight β_i if p is a prefix of α_i and secret shares of zero (0^m) otherwise. So, if $\mathcal{C}_i$'s measurement is $\alpha_i = 111\ldots$, then all the evaluations of $\mathcal{C}_i$'s VIDPF keys at level two yield shares of zero, except for the evaluation on $p = 11$ which returns shares of β_i. At the next

level, the aggregators need to verify the evaluated path, i.e., the evaluation on $p = 111$ returns shares of the same weight β_i as in the previous level, while all other evaluations at level three return secret shares of zero.

The way to do this is surprisingly *simple* and *efficient*: check that the output for p is equal to the sum of the outputs of its two children (i.e., the prefixes $p \parallel 0$ and $p \parallel 1$):

$$\mathcal{S}_0 \text{ computes } h^p_{(i,0)} \coloneqq [\![y_i^p]\!]_0 - [\![y_i^{p\|0}]\!]_0 - [\![y_i^{p\|1}]\!]_0; \text{ and}$$
$$\mathcal{S}_1 \text{ computes } h^p_{(i,1)} \coloneqq -[\![y_i^p]\!]_1 + [\![y_i^{p\|0}]\!]_1 + [\![y_i^{p\|1}]\!]_1.$$

Observe that checking that $h^p_{(i,0)}$ and $h^p_{(i,1)}$ are equal can be done by hashing both and comparing the hashes. This equality guarantees us that the evaluation of the parent node is the sum of the evaluations of the children since $h^p_{(i,0)} = h^p_{(i,1)}$ then:

$$\begin{aligned}
&[\![y_i^p]\!]_0 - [\![y_i^{p\|0}]\!]_0 - [\![y_i^{p\|1}]\!]_0 = -[\![y_i^p]\!]_1 + [\![y_i^{p\|0}]\!]_1 + [\![y_i^{p\|1}]\!]_1 \\
\Leftrightarrow &[\![y_i^p]\!]_0 + [\![y_i^p]\!]_1 = [\![y_i^{p\|0}]\!]_0 + [\![y_i^{p\|1}]\!]_0 + [\![y_i^{p\|0}]\!]_1 + [\![y_i^{p\|1}]\!]_1 \\
\Leftrightarrow &y_i^p = y_i^{p\|0} + y_i^{p\|1}.
\end{aligned}$$

However, this path check alone is not sufficient as the weight of the parent can be split between the children. This check will still pass, but this is not a valid report since it is not one-hot.

One-hotness is assured by the verifiability property of VIDPF (Section 2.3). For each prefix evaluation, the two aggregators also receive a proof. By combining multiple proofs at the same level, the aggregators can verify that the evaluations for a specific level are one-hot. This one-hot check, in conjunction with the path and weight check, is sufficient to ensure the validity of each client's report at a given level.

The aforementioned checks are described in "Step 2.a." in "Aggregator Computation" in Fig. 2 and are performed for every client. In more detail, the aggregators evaluate the VIDPF in the two children $\gamma \in \{p \parallel 0, p \parallel 1\}$ of the current path p, each receives a secret share $[\![y_i^\gamma]\!]_b$ and VIDPF proof $\pi^\gamma_{(i,b)}$. Then, they use the secret shares for the path verifiability check and generate $h^p_{(i,b)}$ for each prefix p. As there are usually multiple candidate prefixes at each, we batch all the checks for all the prefixes per client: the aggregators hash all $h^p_{(i,b)}$, $\pi^{p\|0}_{(i,b)}$, $\pi^{p\|1}_{(i,b)}$ for all $p \in \mathsf{HH}^k$ into $R^k_{(i,b)}$. Finally, the aggregators exchange $R^k_{(i,b)}$ for each client $\mathcal{C}_i$ and check that $R^k_{(i,0)} = R^k_{(i,1)}$. If they are not equal, then the aggregators remove i from the set of candidate reports $\mathsf{Reports}$ as either the path or the one-hot check has failed. The aggregators exchange as many hashes as the total clients, but this can be improved as described in Section 3.2.4.

Aggregation & Pruning. After the "Verifying Client Inputs" step is done, the two aggregators have removed all malformed reports for the root level as well as for the current level (e.g., level k). The next step is to aggregate all the client reports together and compute the heavy hitter prefixes HH^k of length k. This step is shown in "Step 2.b." in "Aggregator Computation" in Fig. 2. Each $\mathcal{S}_b$ accumulates all secret shares $[\![y_i^\gamma]\!]_b$ for prefixes $\gamma \in \{p \parallel 0, p \parallel 1\}$ into $[\![\mathsf{weight}^\gamma]\!]_b$. Then, both aggregators recover weight^γ and decide to keep (or prune) the prefix γ based on if the output of a function $\mathsf{order}(\cdot)$ is greater than a threshold T. Note that each weight β_i is a vector of field elements ($\beta_i \in \mathbb{F}^m$ as defined in Section 2.3). Finally, the aggregators prune all prefixes with order less than T and continue to the next level.

3.2.3 Modes of Operation

Mastic is designed to support a variety of secure aggregation tasks. These can be categorized into three modes of operation.

Weighted Heavy Hitters. Mastic is designed primarily to solve the weighted heavy hitters problem. In this problem, each measurement is a pair $(\alpha, \beta) \in \{0,1\}^n \times \mathcal{L}$ and the output is the subset of inputs for which $\mathsf{order}(\mathsf{weight}) \geq \mathsf{T}$, where weight is the sum of the weights and $\mathcal{L} \subseteq \mathbb{F}^m$, $\mathsf{order} : \mathbb{F}^m \mapsto \mathbb{R}$, and $\mathsf{T} \in \mathbb{R}$ are determined by the application. We provide an example of such an application in Section 4.1.

Plain Heavy Hitters. The plain heavy hitters problem is a special case of weighted heavy hitters where each $m = 1$, $\mathcal{L} = \{[0], [1]\}$, and $\mathsf{order}(\cdot)$ is the identity function.

Attribute-Based Metrics. Mastic admits an enhanced variant of general-purpose metrics (*à la* Prio [BCPP22]) in which aggregates are broken down by client attributes. Just like in weighted heavy hitters, each measurement consists of a pair $(\alpha, \beta) \in \{0,1\}^n \times \mathcal{L}$, where we call α the client's *attribute* and β the client's *value*. The aggregators evaluate the reports on a set of attributes of interest $[x_1, \ldots, x_A]$. The result is the aggregate of (i.e., total of the values for) of reports that share the same attribute. We give an example application in Section 4.2.

Mastic in this mode of operation is essentially a subset of the tree traversal algorithm in Fig. 2. The main difference is that, instead of using the weights to decide which paths to traverse, the aggregators decide in advance which leaves they want to evaluate (i.e., the attributes) and traverse the path to each of these leaves. The shared ZK proof is checked at the root. The VIDPF proof, state, and path checks are all computed as usual, except they only exchange the $R^k_{(i,b)}$-values at the last level of the tree. (The shared ZK partial proofs are exchanged in the same flow.) Note that, in order to compute the path checks, it is also necessary to evaluate the sibling of each node traversed along a path to a leaf.

3.2.4 Optimizations

Batched Path and One-Hot Checks. In the "Client Input Verification" of Section 3.2.2, during the intermediate levels (say k) and leaves verification, the aggregators need to exchange (up to)[11] N hashes (i.e., $R^k_{(i,b)}$ for $i \in [\mathsf{N}]$) to verify the inputs of N clients. We adopt a batching optimization based on Merkle trees proposed by PLASMA [MST24] to reduce this number and allow the aggregators to verify N clients in a batch. Each $\mathcal{S}_b$ creates a Merkle tree from the N hashes and sends the Merkle tree root to the other aggregator. If they match, then we know that the one-hot and path checks passed for all clients. If the roots are different, then the aggregators repeat the same process for the next level until they traverse down the tree and remove the malicious clients.

Minimizing Communication for Weight Checks. In plain heavy hitters mode, the shared ZK evaluation can be foregone completely in favor of the simpler, $\beta = 1$ check of PLASMA. This decreases the size of the reports sent from clients to aggregators (shared ZK partial proofs are omitted) as well as the size of the messages sent between the aggregators (shared ZK partial verifiers are omitted from the root checks).

The size of the shared ZK partial proofs used in our construction of $\mathcal{Z}$ is $\mathcal{O}(m)$ in general. With standard PRG-based secret sharing techniques, we can ensure the concrete communication cost of the $\mathcal{Z}$ partial proofs is small. (The helper's partial proof can be represented by a PRG seed; see [BCPP22, Section 7] for details.)

4 Applications

This section describes two key applications that motivate Mastic. Our design objectives for Mastic are fully aligned with the goals of the working group at IETF developing standards for MPC for secure aggregation [IET]. In fact, Mastic is designed to overcome limitations of existing protocols already being considered for standardization [BCPP22]: Poplar [BBC+21] and Prio [CGB17].

Poplar (and PLASMA [MST24]) are designed specifically to solve the plain (i.e., non-weighted) heavy hitters problem. There is a plethora of applications for which this limited functionality falls short, one of which we describe in detail below (Section 4.1). Our primary goal for Mastic is to provide a replacement for Poplar that is more flexible and therefore useful in a wider variety of applications.

Prio on the other hand is already fairly versatile. However, it too has at least one important limitation, which hindered progress on standardization for some use cases [IET]. In particular, it is often necessary to break down metrics by client attributes. For example, when aggregating a histogram, we may want to split the results into a separate histogram for clients grouped by some property, like their geographical location. We describe a motivating application in Section 4.2.

[11] Note that the number of hashes here may be less than N as some reports may have been removed at a previous level of the tree.

Prio can be extended to support such functionality: instead of aggregating a length-m histogram, we would aggregate a length-$(m \cdot A)$ histogram comprised of A length m histograms, one for each of the A attributes of interest. However, this would result in $\mathcal{O}(A)$ overhead in communication, which is impractical in most situations. Mastic, on the other hand, provides the same functionality, but with only $\mathcal{O}(n)$ overhead, where n is the length in bits of each attribute.

4.1 Network Error Logging

Network Error Logging (NEL) [W3C23] is a mechanism used by web browsers to report errors that occur while attempting to establish a connection to a server. 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, such as Akamai, AWS, or Cloudflare, 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., [W3C23, Section 7]). 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 [W3C23, Section 5.1]).

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 [W3C23, Section 9]. 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).

Private NEL with Mastic. 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.

We consider here a simplified version of NEL where each client reports a tuple $(\mathsf{dom}, \mathsf{err})$ 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.

Inputs: Each input α encodes the domain dom truncated to $n = 256$ bits, which is sufficient to represent most of the domains on the internet [BBC+21, MST24]. Domains that are shorter than n bits are padded with 0s.

Weights: Each weight β represents the error variant dom. To compute the distribution of errors, we encode each error variant as a distinct bucket of a histogram so that $[1, 0, 0, \ldots]$ represents `ok`, $[0, 1, 0, \ldots]$ represents `dns.unreachable`, $[0, 0, 1, \ldots]$ represents `dns.name_not_resolved`, and so on. There are 30 such variants (see [W3C23, Section 6]), so the language $\mathcal{L}$ of valid weights is exactly the set of length-30 vectors over $\mathbb{F}$ containing all 0s except for a single 1.

Ordering: Our $\mathsf{order}(\cdot)$ function computes the ratio of reports with $\mathsf{err} \neq \texttt{ok}$ to reports with $\mathsf{err} = \texttt{ok}$. The latter is simply the first bucket of the aggregated histogram; the former is the sum of the remaining 29

buckets. Note that our ordering of aggregated weights considers the error rate rather than the raw error count. This ensures that the signal for less popular domains is not swamped by the noise generated by popular sites (network issues may impact some domains but not others). Another benefit is that, under normal operating conditions, there will be a small number of heavy-hitting domains, which means Mastic will run very efficiently. During an incident, there will be more heavy hitters, which means it will take longer to compute the set of impacted domains. However, *we get the errors immediately* at the root of the prefix tree, which is the most important information needed to begin remediation. As more levels are evaluated we get more detailed errors.

4.2 Attribute-Based Browser Telemetry

Web browsers, like Chrome, Firefox, or Safari, collect telemetry generated by users as they surf the web to gain insights into trends that guide product decisions. In many cases, Prio can be used to privately aggregate this telemetry. However, this comes at the cost of flexibility.

For example, Mozilla is using Prio [Hol23] to collect page load metrics from Firefox for a list of known popular sites (e.g., `google.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. The most crucial attributes are 1) the software *version*, and 2) the information about the client's *location*.

As we mentioned above, meeting this requirement by increasing the size of the histogram leads to intolerable communication overhead. An alternative is to have each client upload this information in the clear alongside its Prio report so that the reports can be grouped by version and location. The downside of this approach is that it significantly reduces the anonymity set of each user since they are only mixed with their attribute group rather than the entire population.

Private Browser Telemetry with Mastic. Mastic provides a simple solution to this problem. For the sake of presentation, we consider a simplified version of Mozilla's use case (the same approach can be applied to any aggregation task for which Prio is suitable). Each client reports a tuple $(\mathsf{ver}, \mathsf{loc}, \mathsf{site}, \mathsf{time})$ where: ver is a string representing the client's software version (e.g., `Firefox/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., `google.com`, `wikipedia.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 4.1, 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 3.2.3):

Attributes: Two-letter country codes can easily be encoded in 2 bytes. Likewise, the number of distinct browser versions is easily less than 2^{16}, so 2 bytes are sufficient. Therefore, each α can be encoded with just $n = 32$ bits.

Values: Similar to private NEL, each weight β is a 0-vector except for a single 1 representing a bucket in a histogram. We represent $(\mathsf{site}, \mathsf{time})$ 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 $[5, \infty)$. Let $t \in [4]$ denote the time bucket for time. Next, suppose we wish to track metrics for 25 sites. Let $s \in [25]$ denote the index of site in this list. Then the index of 1 in β is simply $t \cdot s$ such that $|\beta| = m = 4 \cdot 25 = 100$.

5 Security Analysis of Mastic

In this section, we present our security analysis Π_{Mastic} (Fig. 2). Following [DPRS23, CGB17], we consider privacy and robustness separately.

5.1 Privacy: Malicious Clients and Aggregator

To define privacy, we first consider what information is protected and what is leaked. The Mastic protocol is designed to reveal the sum of the weights for every valid report whose point α is prefixed by the query p of interest to the aggregators. We ask that even if one of the two aggregators is malicious, neither aggregator learns more than this sum, even if it knows something else about the individual client measurements.

We capture this property in a simulation-based model, presented in Figs. 9 and 10 found in Appendix C. Essentially, we ask for the existence of a stateful algorithm Sim that can interact with a malicious aggregator exactly as an honest aggregator and clients would. This simulator should be indistinguishable from the real protocol operations, even if the adversary knows all honest clients' measurements, and the simulator knows only the aggregate results. When one of the aggregators is fully malicious, Mastic can be used in a variety of ways that do not conform to the weighted heavy-hitters application, so we consider privacy over all modes of operation in Section 3.2.3. This means that a malicious aggregator can ask for aggregate results across any set of client reports it desires without violating our privacy notion. Consequently, we do not protect against inference or Sybil attacks and leave the according defenses up to higher-level systems.

We define two games, $\mathcal{G}^{\mathsf{priv\text{-}real}}_{\mathsf{Mastic}}$ (c.f. Fig 9), and $\mathcal{G}^{\mathsf{priv\text{-}ideal}}_{\mathsf{Sim}}$ (c.f. Fig 10). Each of these games initially requests a verification key vk and the corrupt aggregator index b from the adversary $\mathcal{A}$, then presents an interface of six oracles capturing all the interactions of an honest aggregator: with honest clients, corrupt clients, and the malicious aggregator. The adversary may interact with these oracles at will; when it halts, it must output a bit denoting whether it believes the interface is real or simulated.

Informally, the game $\mathcal{G}^{\mathsf{priv\text{-}real}}_{\mathsf{Mastic}}$ presents the adversary with a view of the real Mastic protocol, and $\mathcal{G}^{\mathsf{priv\text{-}ideal}}_{\mathsf{Sim}}$ presents a simulated view that doesn't depend on the value of individual client measurements. Our model is actually slightly stronger, in that we give the adversary a little *more* information than it would learn from a real Mastic interaction. This is because in Mastic, aggregators process many prefixes simultaneously for each report, and they verify one-hot and path verifiability proofs in batches that are hashed together. However, since Mastic doesn't place any restrictions on the size of these batches or whether they overlap, we let the model capture only the worst-case scenario and have aggregators process prefixes individually and return all proofs without hashing. It should be clear that any information that is leaked by the batched proofs will also be leaked by the inputs to the hash function, so this strictly strengthens the security definition.

We define the advantage $\mathbf{Adv}^{\mathsf{priv}}_{\mathsf{Mastic},\mathsf{Sim}}(\mathcal{A})$ of $\mathcal{A}$ against the privacy of Mastic with respect to simulator Sim as:

$$\left|\Pr[\mathcal{G}^{\mathsf{priv\text{-}real}}_{\mathsf{Mastic}}(\mathcal{A}) \Rightarrow 1] - \Pr[\mathcal{G}^{\mathsf{priv\text{-}ideal}}_{\mathsf{Sim}}(\mathcal{A}) \Rightarrow 1]\right|.$$

Informally, we say that Mastic is private if there is a PPT simulator for which all PPT adversaries get negligible advantage in the security parameter.

Let $\mathbf{Adv}^{\mathsf{priv}}_{\mathcal{Z},\mathsf{Sim}_{\mathrm{SZK}}}(\cdot)$ denote the advantage of an adversary in attacking the privacy of $\mathcal{Z}$ with respect to simulator $\mathsf{Sim}_{\mathrm{SZK}}$ (we define this function precisely in Appendix B). Likewise, let $\mathbf{Adv}^{\mathsf{priv}}_{\mathcal{V}}(\cdot)$ denote the advantage of an adversary in attacking the privacy of $\mathcal{V}$ (defined in Appendix A). We claim the following theorem.

Theorem 1. *For any simulator Sim_{SZK}, there exists a simulator Sim such that for any adversary $\mathcal{A}$, there exist $\mathcal{Z}$-attacker $\mathcal{B}$ and $\mathcal{V}$-attacker $\mathcal{D}$ such that*

$$\mathbf{Adv}^{\mathsf{priv}}_{\mathsf{Mastic},\mathsf{Sim}}(\mathcal{A}) \leq \mathbf{Adv}^{\mathsf{priv}}_{\mathcal{Z},\mathsf{Sim}_{SZK}}(\mathcal{B}) + q \cdot \mathbf{Adv}^{\mathsf{priv}}_{\mathcal{V}}(\mathcal{D})\,,$$

where q is the number of queries made to the "Honest Client Computation" oracle, and the runtime of $\mathcal{D}$ is about that of an honest aggregator in the Mastic protocol, and the runtime of $\mathcal{B}$ is about that of an honest aggregator plus the time to run Sim_{SZK} once per interaction with the honest aggregator.

A proof of this theorem, including the specification of the simulator, is given in Appendix C.

5.2 Robustness: Malicious Clients

Next, we focus on the robustness guarantees provided by Mastic against malicious clients. To argue robustness we assume the aggregators follow the Mastic protocol steps correctly.

We capture this property in a simulation-based model [Can00]. The adversary $\mathcal{A}$ initially corrupts a set $\mathsf{Reports}'$ of clients. In the real-world game $\mathcal{G}^{\mathsf{rob\text{-}real}}_{\mathsf{Mastic}}$ (Fig. 17 in Appendix D), the parties run the Mastic protocol using their input measurements. Both the honest and corrupt clients provide their report shares to the aggregators, who compute the output (set of heavy-hitter strings and their children, and also the weights of the heavy-hitting strings and their children) and return it to $\mathcal{G}^{\mathsf{rob\text{-}real}}_{\mathsf{Mastic}}$. The game forwards this to $\mathcal{A}$.

We also define a corresponding ideal-world game, $\mathcal{G}^{\mathsf{rob\text{-}ideal}}_{\mathsf{Sim},\mathcal{F}_{\mathsf{wHH}}}$ (Fig. 17). In this game, the corrupt clients (controlled by $\mathcal{A}$) participate in the same way as in the real world, but with a pair of simulated aggregators. This simulation is performed by a PPT algorithm called Sim (Fig. 18). The simulator also has access to an ideal functionality $\mathcal{F}_{\mathsf{wHH}}$ (Fig. 15) for weighted heavy-hitters. The simulator obtains the report shares of the corrupt clients, extracts the input measurements (α'_i, β'_i) from those shares, and invokes $\mathcal{F}_{\mathsf{wHH}}$ (on behalf of the corrupt clients) with the extracted measurements. Upon obtaining the client measurements from both honest and corrupt clients, $\mathcal{F}_{\mathsf{wHH}}$ checks the measurements and then computes the output. For consistency check, $\mathcal{F}_{\mathsf{wHH}}$ discards the measurements if they are not correctly formatted, i.e. $(\alpha_i, \beta_i) = (\bot, \bot)$, or if the weight is not valid, i.e. $\beta_i \notin \mathcal{L}$. The functionality also allows corrupt clients to submit prefix strings, i.e. $\alpha_i \in \{0,1\}^{\leq n}$. Once the measurements are validated, the functionality aggregates the validated measurements by computing the weights and heavy-hitting set similar to the aggregation phase in Mastic. $\mathcal{F}_{\mathsf{wHH}}$ computes the output and this is returned to $\mathcal{G}^{\mathsf{rob\text{-}ideal}}_{\mathsf{Sim},\mathcal{F}_{\mathsf{wHH}}}$. The game forwards this to $\mathcal{A}$.

Define the advantage $\mathbf{Adv}^{\mathsf{rob}}_{\mathsf{Mastic},\mathsf{Sim}}(\mathcal{A})$ of $\mathcal{A}$ in breaking the robustness of Π_{Mastic} with respect to simulator Sim as

$$\left| \Pr[\mathcal{G}^{\mathsf{rob\text{-}real}}_{\mathsf{Mastic}}(\mathcal{A}) \Rightarrow 1] - \Pr[\mathcal{G}^{\mathsf{rob\text{-}ideal}}_{\mathsf{Sim},\mathcal{F}_{\mathsf{wHH}}}(\mathcal{A}) \Rightarrow 1] \right| .$$

Informally, we say Π_{Mastic} is robust if there exists a PPT simulator Sim such that $\mathbf{Adv}^{\mathsf{rob}}_{\mathsf{Mastic},\mathsf{Sim}}(\mathcal{A})$ is negligible in the security parameter for all PPT adversaries $\mathcal{A}$.

We write $\mathbf{Adv}^{\mathsf{coll}}_{\mathrm{H}}(\cdot)$ to denote the probability of an attacker finding a collision against hash function H. We write $\mathbf{Adv}^{\mathsf{verif}}_{\mathcal{V},k}(\cdot)$ to denote the probability of an attacker breaking verifiability of $\mathcal{V}$ at level $k \in [n]$ (we define this function in Appendix A). We write $\mathbf{Adv}^{\mathsf{sound}}_{\mathcal{Z}}(\cdot)$ for the probability that an attacker breaks the soundness of $\mathcal{Z}$ (Appendix B).

Theorem 2. *There exists a simulator* Sim *such that for all* $k \in [n]$ *and all* Π_{Mastic}*-robustness attackers* $\mathcal{A}$*, there exist a* $\mathcal{Z}$*-soundness attacker* $\mathcal{B}$*, an algorithm* $\mathcal{C}$ *for finding* H*-collisions, and a* $\mathcal{V}$*-verifiability attacker* $\mathcal{D}$ *such that*

$$\mathbf{Adv}^{\mathsf{rob}}_{\mathsf{Mastic},\mathsf{Sim}}(\mathcal{A}) \leq \mathsf{N}' \cdot \left(\mathbf{Adv}^{\mathsf{sound}}_{\mathcal{Z}}(\mathcal{B}) + n \cdot (\mathbf{Adv}^{\mathsf{coll}}_{\mathrm{H}}(\mathcal{C}) + \mathbf{Adv}^{\mathsf{verif}}_{\mathcal{V},k}(\mathcal{D}))\right),$$

where at most N' *clients are corrupted by* $\mathcal{A}$ *and the run times of* Sim*,* $\mathcal{B}$*,* $\mathcal{C}$*, and* $\mathcal{D}$ *are upper bounded by the combined run time of honest aggregators in* Π_{Mastic} *and the run time of* $\mathcal{A}$*.*

A proof of this theorem, including the simulator Sim, the detailed hybrids, and the indistinguishability argument are given in Appendix D. The games and ideal functionality $\mathcal{F}_{\mathsf{wHH}}$ are also defined there.

6 Experimental Evaluation

Our goal in this section is to assess whether Mastic is efficient enough for the applications described in Section 4.

Setup. We performed experiments on two AWS instances (`c5.18xlarge`) each with 72 vCPUs at 3.60 GHz, 144 GB memory, and 25 Gbps of network bandwidth. All our experiments are over a wide area network (WAN), with one server in Ohio (`us-east-2`) and the other in Oregon (`us-west-2`). We only focus on WAN as this is the most common way an MPC protocol will be deployed in the real world. We measure the runtime from the moment the aggregators receive all client report shares and start running the protocol. Mastic is implemented in Rust 1.74 and uses `tarpc` for asynchronous Remote Procedure Calls (RPC) and `rayon` for

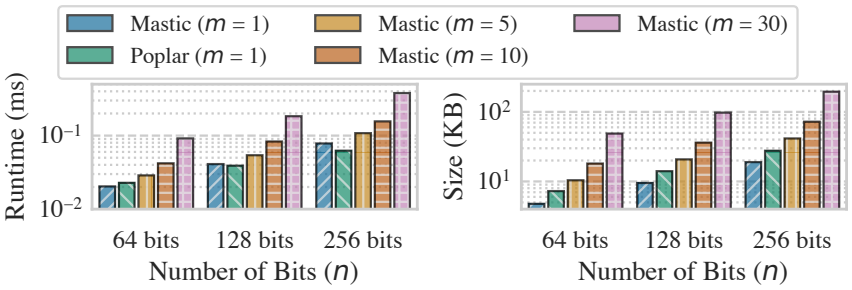

Fig. 3: Report generation times (left) and report size (right) for different values of n and m.

multi-threading. Client-side experiments are carried out using a laptop with an Intel i7-8650U CPU (1.90 GHz). For heavy hitters (plain and weighted) we set the T to be 1% of the clients' bit strings.[12]

Target experiments. The goals of our experiments are to: 1) evaluate Mastic for weighted heavy-hitters and attribute-based metrics; 2) compare Mastic with related works; and 3) demonstrate the feasibility of Mastic for the applications described in Section 4 – i.e., NEL and attribute-based telemetry.

Weighted Heavy Hitters & Attribute-Based Metrics. Mastic is the only protocol for weighted heavy hitters and we use various weight sizes m (namely 5, 10, and 30) for a fixed number of bits n=256. Note that there are multiple ways to implement pruning based on the weights. For consistency with our plain heavy-hitter examples, we increased m by one and used the identity function for $\mathsf{order}(\cdot)$; i.e., the last index of the aggregated weight counts the number of reports. This way, we can use the same threshold as in plain heavy-hitters, $\mathsf{T} = 1\%$ of N. Additionally, we explore how the presence of malicious clients affects the protocol latency. For plain and attribute-based metrics, we fixed the m to 100 and varied the number of attributes A between 1 (for plain metrics) to 128 and 1024 for attribute-based metrics. In all these cases, our field size is 128 bits.

Related Works. First, we compare Mastic to Poplar [BBC+21], the state-of-the-art for *plain* heavy hitters in the two-aggregator setting. For plain heavy hitters with Mastic, we use $m = 1$ and a field size of 64 bits. Poplar's implementation[13] uses 62-bit fields for intermediate levels and 256 bits for the leaves.

Second, we wish to benchmark Mastic's attribute-based metrics mode with Prio [CGB17]. Prio can provide a (less flexible) form of this functionality, as described in Section 4. Concretely, our goal is to aggregate a histogram of length $m = 100$ (as in Section 4.2) for each of A (fixed, in the case of Prio) attributes. We emulate this in Prio by aggregating a length $m \cdot A$ histogram, where the first m buckets correspond to the first attribute, the next m buckets correspond to the next attribute, and so on. We consider various numbers of attributes, $A = 1$, 128, and 1024. Correspondingly: for Mastic we set the bit length to $n = 1$, 7, and 10, respectively; and for Prio we set the input length to 100, 12800, and 102400, respectively. For Prio we use `libprio-rs`,[14] which implements the candidate standard [BCPP22]. Notably, we apply the same parallelization techniques across all protocols (Mastic, Poplar, and Prio) and run them in the same WAN setup.

Client Cost. First, we benchmark the client costs: the time it takes for a client to generate a report as well as the size of each report share (the message sent to each aggregator). Both of these costs vary based on the length of α and the size m of the weight β. We report the combination of three different sizes of α ($n = 64$, 128, and 256 bits) with $m = 1$ (for plain heavy hitters), $m = 5, 10$, and 30 (for weighted heavy hitters).

The client costs are shown in Fig. 3. On the left-hand side, we show the report generation time; as expected, the larger values of n and m take more time than the smaller variants, but in all cases, the report generation time is minimal. On the right-hand side of Fig. 3, we show the report sizes for the different α and

[12] Our implementation of Mastic is open-source at https://github.com/TrustworthyComputing/mastic.
[13] https://github.com/henrycg/heavyhitters
[14] https://github.com/divviup/libprio-rs

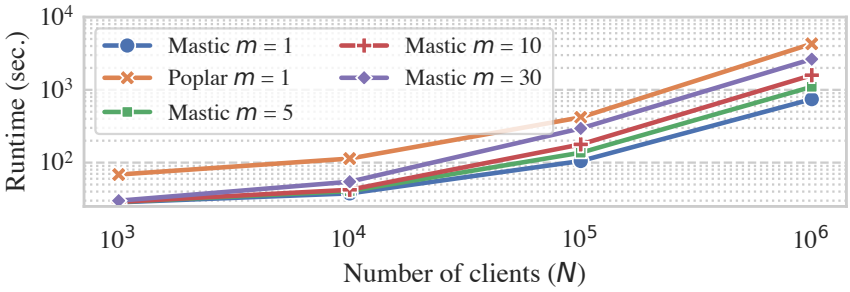

Fig. 4: *Plain* ($m = 1$) and *Weighted* ($m > 1$) *heavy hitters* for $n = 256$. Aggregator runtime (WAN) for an increasing N.

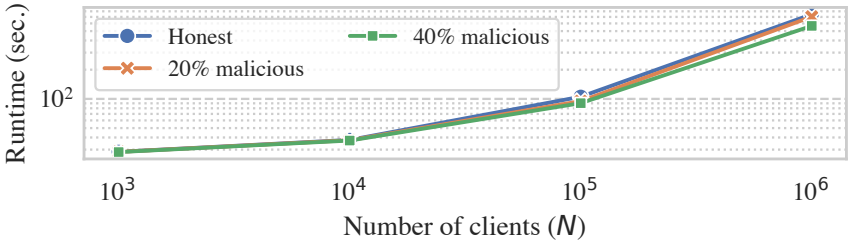

Fig. 5: Mastic plain heavy-hitters with malicious clients.

m configurations. The smallest report size (64 bits, $m = 1$) is less than 5 KB, while the biggest key size (256 bits, $m = 30$) is less than 200 KB.

Prio is not suitable for weighted heavy hitters and therefore does not appear in Fig. 3. Instead, we compare the client cost for our application from Section 4.2: the total size of Mastic report shares for $n = 32$ and $m = 100$ uploaded by each client is about 53KB, while Prio for the same purpose with $A = 1024$ attributes needs about 1.6MB (30× overhead). Prio is also less flexible since the set of attributes would need to be known to the clients in advance.

Aggregator Cost. Next, we focus on aggregator costs for plain and weighted heavy hitters. In Fig. 4, we show the aggregator runtime over WAN for an increasing number of clients, 1k, 10k, 100k, and 1 million, and for $n = 256$ bit strings. For plain heavy hitters ($m = 1$), Mastic outperforms Poplar by almost an order of magnitude. In the same figure, we show Mastic's aggregator runtime for three different weight sizes (i.e., m =5, 10, and 30). As expected, the bigger m values have an impact on the runtime of the protocol; both due to increased communication and increased computation (i.e., path verifiability now has to consider m values). For 10^6 clients, going from $m = 5$ (18 minutes) to $m = 30$ (44 minutes) more than doubles the latency (but still outperforms Poplar).

In Fig. 5, we vary the number of malicious clients for plain heavy hitters between 20% and 40% of the total number of clients. We observe a higher percentage of malicious clients results in lower latency for Mastic, which is primarily because malicious clients fail to pass the shared ZK verification so they are eliminated from the protocol. If weight check succeeds, but the path or one-hot check fails, Mastic performs similarly to PLASMA [MST24]. Note that Mastic exhibits the same scaling for weighted heavy hitters as with plain, as the only difference between the two is the size of the weight. Poplar does not show how it scales in the presence of malicious clients but we expect a similar trend.

In Fig. 6, we show the aggregator cost for attribute-based metrics with Mastic versus Prio. As expected, for a single attribute ($A = 1$) Prio is faster than Mastic after 10^5 clients as it uses $n = 0$. However, as we

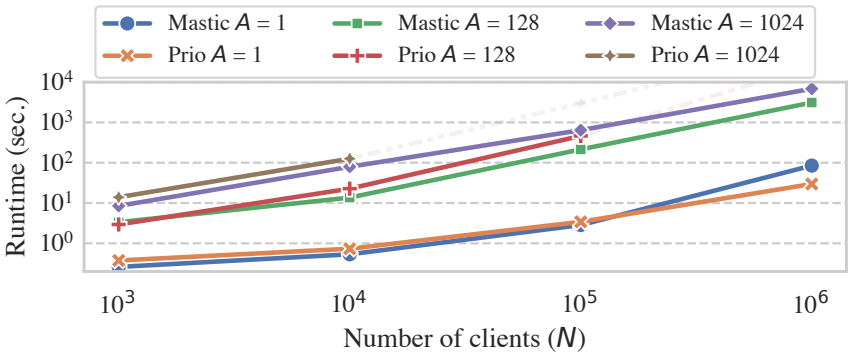

Fig. 6: *Plain* ($A = 1$) and *Attribute-based* ($A > 1$) *metrics* with $m = 100$. For Mastic, we use $n = \lfloor \log_2(A+1) \rfloor$. For Prio, the faded-out lines represent extrapolated values as the evaluation did not finish after exceeding the available server memory. Aggregator runtime over WAN for an increasing N.

increase the value of A, we observe that Mastic quickly outperforms Prio (about $1.5 - 2\times$ as fast). Another benefit of Mastic is that the attributes do not need to be fixed *a priori*; Mastic can support any number.

Suitability of Mastic for NEL. The NEL application (Section 4.1) is time-sensitive: the sooner the results are available, the sooner they can be acted upon to diagnose and resolve the issue that precipitated the errors. As shown in Fig. 4, for realistic parameters ($n = 256$; $m = 30$; $\mathsf{N} = 10^6$ clients), Mastic took 44 minutes to compute the high-error rate domains (i.e., inputs α) and their error distributions. The end-to-end latency could be further improved by issuing parallel RPCs and utilizing all the available network bandwidth. Still, waiting even 5 minutes for results might be too long, depending on the conditions. Fortunately, Mastic produces the errors themselves much faster, immediately after evaluating the first level of the prefix tree for each report (in just 10.4 seconds in our experiment). To summarize, Mastic is the first work that enables such an application; this can be used in a real NEL instantiation as long as some latency for learning impacted domains can be tolerated.

Suitability of Mastic for Browser Telemetry. Lastly, we consider the parameters described in Section 4.2, i.e., $n = 32$, $m = 100$ and 200 with a 128-bit field. For a single attribute, the latency for Mastic is 103 and 168 seconds for 1 million clients for $m = 100$ and $m = 200$, respectively. Going up to 10 attributes, Mastic takes approximately 9 minutes for $m = 100$. Based on this result, we believe it is clear that Mastic is concretely efficient enough for this application.

7 Concluding Remarks

This work presents Mastic, the first two-server MPC protocol for general-purpose metrics that supports both weighted heavy-hitters and aggregation grouped by user attributes. Mastic offers notable benefits over the previous state-of-the-art Prio and Poplar frameworks that focused on either plain aggregations or plain heavy hitters. At its core, our protocol leverages verifiable incremental distributed point functions along with shared zero-knowledge proofs to enable privacy and robustness in the presence of malicious clients and privacy in the presence of a malicious aggregator. Mastic is efficient for real-life applications, such as network error logging and attribute-based telemetry, and outperforms Poplar and Prio on heavy-hitters and attribute-based scenarios, respectively.

Acknowledgments

The applications in Section 4 were first described to us in discussions in the PPM working group at IETF. The authors would like to thank Suleman Ahmad (Cloudflare) and Simon Friedberger (Mozilla) for their help in fleshing out the details.

D. Mouris and N.G. Tsoutsos would like to acknowledge the support of the National Science Foundation (Award 2239334).

References

ACD+21. Erik Anderson, Melissa Chase, F. Betul Durak, Esha Ghosh, Kim Laine, and Chenkai Weng. Aggregate measurement via oblivious shuffling. Cryptology ePrint Archive, Report 2021/1490, 2021. https://eprint.iacr.org/2021/1490.

AHI+22. Gilad Asharov, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Ariel Nof, Benny Pinkas, Katsumi Takahashi, and Junichi Tomida. Efficient secure three-party sorting with applications to data analysis and heavy hitters. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022: 29th Conference on Computer and Communications Security*, pages 125–138. ACM Press, November 2022.

App23. Apple. Learning Iconic Scenes with Differential Privacy, 2023. https://machinelearning.apple.com/research/scenes-differential-privacy.

BBC+19. Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Zero-knowledge proofs on secret-shared data via fully linear PCPs. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019, Part III*, volume 11694 of *Lecture Notes in Computer Science*, pages 67–97. Springer, Heidelberg, August 2019.

BBC+21. Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. In *2021 IEEE Symposium on Security and Privacy*, pages 762–776. IEEE Computer Society Press, May 2021.

BBC+23. Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Arithmetic sketching. In *Advances in Cryptology – CRYPTO 2023, Part I*, Lecture Notes in Computer Science, pages 171–202. Springer, Heidelberg, August 2023.

BCPP22. Richard Barnes, David Cook, Christopher Patton, and Schoppmann Phillipp. Verifiable Distributed Aggregation Functions. Internet Engineering Task Force (IETF), Internet-Draft, 2022. https://datatracker.ietf.org/doc/draft-irtf-cfrg-vdaf.

BDO14. Carsten Baum, Ivan Damgård, and Claudio Orlandi. Publicly auditable secure multi-party computation. In Michel Abdalla and Roberto De Prisco, editors, *SCN 14: 9th International Conference on Security in Communication Networks*, volume 8642 of *Lecture Notes in Computer Science*, pages 175–196. Springer, Heidelberg, September 2014.

BGG+22. James Bell, Adrià Gascón, Badih Ghazi, Ravi Kumar, Pasin Manurangsi, Mariana Raykova, and Phillipp Schoppmann. Distributed, private, sparse histograms in the two-server model. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022: 29th Conference on Computer and Communications Security*, pages 307–321. ACM Press, November 2022.

BGI15. Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015, Part II*, volume 9057 of *Lecture Notes in Computer Science*, pages 337–367. Springer, Heidelberg, April 2015.

BK21. Jonas Böhler and Florian Kerschbaum. Secure multi-party computation of differentially private heavy hitters. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021: 28th Conference on Computer and Communications Security*, pages 2361–2377. ACM Press, November 2021.

Blo70. Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

Can00. Ran Canetti. Security and composition of multiparty cryptographic protocols. *Journal of Cryptology*, 13(1):143–202, January 2000.

CGB17. Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, page 259–282, USA, 2017. USENIX Association.

CGH+18. Koji Chida, Daniel Genkin, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Yehuda Lindell, and Ariel Nof. Fast large-scale honest-majority MPC for malicious adversaries. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 34–64. Springer, Heidelberg, August 2018.

Cloa. Cloudflare. Privacy-preserving measurement and machine learning. https://blog.cloudflare.com/deep-dive-privacy-preserving-measurement.

Clob. Cloudflare. Stop attacks before they are known: making the Cloudflare WAF smarter. https://blog.cloudflare.com/stop-attacks- before-they-are-known-making-the-cloudflare-waf-smarter.

CM04. Graham Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. In Martin Farach-Colton, editor, *LATIN 2004: Theoretical Informatics, 6th Latin American Symposium*, volume 2976 of *Lecture Notes in Computer Science*, pages 29–38. Springer, Heidelberg, April 2004.

dCP22. Leo de Castro and Antigoni Polychroniadou. Lightweight, maliciously secure verifiable function secret sharing. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022, Part I*, volume 13275 of *Lecture Notes in Computer Science*, pages 150–179. Springer, Heidelberg, May / June 2022.

DEF+19. Ivan Damgård, Daniel Escudero, Tore Kasper Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. New primitives for actively-secure MPC over rings with applications to private machine learning. In *2019 IEEE Symposium on Security and Privacy*, pages 1102–1120. IEEE Computer Society Press, May 2019.

DPRS23. Hannah Davis, Christopher Patton, Mike Rosulek, and Phillipp Schoppmann. Verifiable Distributed Aggregation Functions. *Proceedings on Privacy Enhancing Technologies*, 2023(4):578–592, July 2023.

EPK14. Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. RAPPOR: Randomized aggregatable privacy-preserving ordinal response. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014: 21st Conference on Computer and Communications Security*, pages 1054–1067. ACM Press, November 2014.

GJM+23. Sanjam Garg, Abhishek Jain, Pratyay Mukherjee, Rohit Sinha, Mingyuan Wang, and Yinuo Zhang. Cryptography with weights: MPC, encryption and signatures. In *Advances in Cryptology – CRYPTO 2023, Part I*, Lecture Notes in Computer Science, pages 295–327. Springer, Heidelberg, August 2023.

GPP+21. Tim Geoghegan, Christopher Patton, Brandon Pitman, Eric Rescorla, and Christopher Wood. Distributed Aggregation Protocol for Privacy Preserving Measurement. Internet Engineering Task Force (IETF), Internet-Draft, 2021. https://datatracker.ietf.org/doc/draft-ietf-ppm-dap.

Hol23. Bobby Holley. Built for privacy: Partnering to deploy Oblivious HTTP and Prio in Firefox, 2023. https://blog.mozilla.org/en/products/firefox/partnership-ohttp-prio.

IET. IETF. Charter for Privacy Preserving Measurement Working Group.

Int. Internet Security Research Group (ISRG). Divvi Up: A privacy-respecting system for aggregate statistics. https://divviup.org.

JKK+22. Pranav Jangir, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, and Somya Sangal. Poster: Vogue: Faster computation of private heavy hitters. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022: 29th Conference on Computer and Communications Security*, pages 3371–3373. ACM Press, November 2022.

Kel20. Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020: 27th Conference on Computer and Communications Security*, pages 1575–1590. ACM Press, November 2020.

MMT+24. Dimitris Mouris, Daniel Masny, Ni Trieu, Shubho Sengupta, Prasad Buddhavarapu, and Benjamin M Case. Delegated Private Matching for Compute. *Proceedings on Privacy Enhancing Technologies*, 2024(2):1–24, July 2024.

MPC23. MPC Deployments. The Deployment Dilemma: Merits & Challenges of Deploying MPC, 2023. https://mpc.cs.berkeley.edu/blog/deployment-dilemma.html.

MST24. Dimitris Mouris, Pratik Sarkar, and Nektarios Georgios Tsoutsos. PLASMA: Private, Lightweight Aggregated Statistics against Malicious Adversaries. *Proceedings on Privacy Enhancing Technologies*, 2024(3):1–19, July 2024.

Pri. Private Advertising Technology Community Group (PATCG). Private Advertising Technology Community Group Charter. https://patcg.github.io/charter.html.

QYY+16. Zhan Qin, Yin Yang, Ting Yu, Issa Khalil, Xiaokui Xiao, and Kui Ren. Heavy hitter estimation over set-valued data with local differential privacy. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher

Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016: 23rd Conference on Computer and Communications Security*, pages 192–203. ACM Press, October 2016.

Tit22. Trey Titone. Interoperable Private Attribution (IPA) Explained, 2022. https://adtechexplained.com/interoperable-private-attribution-ipa- explained.

TNB+10. Vincent Toubiana, Arvind Narayanan, Dan Boneh, Helen Nissenbaum, and Solon Barocas. Adnostic: Privacy preserving targeted advertising. In *ISOC Network and Distributed System Security Symposium – NDSS 2010*. The Internet Society, February / March 2010.

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

YSWW21. Kang Yang, Pratik Sarkar, Chenkai Weng, and Xiao Wang. QuickSilver: Efficient and affordable zero-knowledge proofs for circuits and polynomials over any field. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021: 28th Conference on Computer and Communications Security*, pages 2986–3001. ACM Press, November 2021.

ZKM+20. Wennan Zhu, Peter Kairouz, Brendan McMahan, Haicheng Sun, and Wei Li. Federated Heavy Hitters Discovery with Differential Privacy. In Silvia Chiappa and Roberto Calandra, editors, *Proceedings of the Twenty Third International Conference on Artificial Intelligence and Statistics*, volume 108 of *Proceedings of Machine Learning Research*, pages 3837–3847, Online, 26–28 Aug 2020. PMLR.

A Security Definitions for VIDPF

Let $\mathcal{V} = (\mathsf{Gen}, \mathsf{Eval}, \mathsf{EvalRoot})$ be a VIDPF as defined in Section 2.3. We recall the security definitions for $\mathcal{V}$ of [MST24], adapted to our refined syntax.

Correctness. For all inputs $\alpha \in \{0,1\}^n$, all weights $\beta \in \mathbb{F}^m$, all $k \in [n]$, and all $p \in \{0,1\}^k$, when we let $(\mathsf{pub}, \mathsf{key}_0, \mathsf{key}_1) := \mathsf{Gen}(\alpha, \beta)$ and

$$(\llbracket y^{p|^i} \rrbracket_b, \mathsf{st}_b^{p|^i}, \pi_b^{p|^i}) := \mathsf{Eval}(\mathsf{key}_b, \mathsf{pub}, p|^i, \mathsf{st}_b^{p|^{i-1}}, \pi_b^{p|^{i-1}})$$

for all $i \leq k$ and $b \in \{0,1\}$, it holds that $\pi_0^p = \pi_1^p$ and

$$\llbracket y^p \rrbracket_0 + \llbracket y^p \rrbracket_1 = \begin{cases} \beta & \text{if } p = \alpha|^k \\ 0^m & \text{otherwise.} \end{cases}$$

Privacy. The information revealed to each aggregator leaks nothing about the underlying measurement. We formalize this via an indistinguishability game. First, for adversary $\mathcal{A}$ and challenge bit c, define $\mathcal{G}^{\mathsf{priv}}_{\mathcal{V},c}(\mathcal{A})$ as the following experiment:

1. Run $\mathcal{A}$ to get $(\alpha^0, \beta^0) \in \{0,1\}^n \times \mathbb{F}^m$ and corrupt aggregator index $b \in \{0,1\}$.
2. Sample (α^1, β^1) uniformly from $\{0,1\}^n \times \mathcal{L}$.
3. Run $(\mathsf{pub}, \mathsf{key}_0, \mathsf{key}_1) := \mathsf{Gen}(\alpha^c, \beta^c)$ and give the public share and corrupt aggregator's key $(\mathsf{pub}, \mathsf{key}_b)$ to $\mathcal{A}$.
4. Run $\mathcal{A}$ to get a bit (its guess of the challenge bit) and output it.

The advantage of $\mathcal{A}$ in attacking the privacy of $\mathcal{V}$ is

$$\mathbf{Adv}^{\mathsf{priv}}_{\mathcal{V}}(\mathcal{A}) := \left| \Pr[\mathcal{G}^{\mathsf{priv}}_{\mathcal{V},1}(\mathcal{A}) \Rightarrow 1] - \Pr[\mathcal{G}^{\mathsf{priv}}_{\mathcal{V},0}(\mathcal{A}) \Rightarrow 1] \right|.$$

Informally, we say that $\mathcal{V}$ is private if no PPT adversary gets a non-negligible advantage in breaking its privacy.

Note that, if the $\mathcal{V}$ is correct, then an aggregator learns nothing from its interaction with its peer. This is because both parties compute the same proof string for each prefix.

Verifiability. No corrupt client can construct $(\mathsf{pub}, \mathsf{key}_0, \mathsf{key}_1)$ for which the portion of the prefix tree traversed by the aggregators contains more than one non-zero node at any level $k \in [n]$. Formally, we define the advantage $\mathbf{Adv}^{\mathsf{verif}}_{\mathcal{V},k}(\mathcal{A})$ of adversary $\mathcal{A}$ in breaking the verifiability of $\mathcal{V}$ at level k as the probability that the following experiment outputs true:

1. Run $\mathcal{A}(1^k)$ to get $(\mathsf{pub}, \mathsf{key}_0, \mathsf{key}_1)$ and $u, v \in \{0,1\}^k$.
2. Evaluate the prefix tree for $(\mathsf{pub}, \mathsf{key}_0, \mathsf{key}_1)$ and $u, v \in \{0,1\}^k$. That is, for all $(i, b) \in [k] \times \{0,1\}$ let

$$([\![y^{u|^i}]\!]_b, \mathsf{st}_b^{u|^i}, \pi_b^{u|^i}) \coloneqq \mathsf{Eval}(\mathsf{key}_b, \mathsf{pub}, u|^i, \mathsf{st}_b^{u|^{i-1}}, \pi_b^{u|^{i-1}})$$
$$([\![y^{v|^i}]\!]_b, \mathsf{st}_b^{v|^i}, \pi_b^{v|^i}) \coloneqq \mathsf{Eval}(\mathsf{key}_b, \mathsf{pub}, v|^i, \mathsf{st}_b^{v|^{i-1}}, \pi_b^{v|^{i-1}})$$

3. Output $([\![y^u]\!]_0 + [\![y^u]\!]_1 \neq 0^m) \wedge ([\![y^v]\!]_0 + [\![y^v]\!]_1 \neq 0^m) \wedge (\pi_0^u \oplus \pi_1^u = \pi_0^v \oplus \pi_1^v) \wedge u \neq v$.

The adversary wins if the output is true, i.e., it finds two distinct, equal length paths in the prefix tree for which 1) both nodes are non-zero and 2) the bitwise-XOR of the aggregators' proofs are equal ($\pi_0^u \oplus \pi_1^u = \pi_0^v \oplus \pi_1^v$). We say that $\mathcal{V}$ is *ϵ-verifiable* if for all PPT adversaries $\mathcal{A}$ and $k \in [n]$ it holds that $\mathbf{Adv}^{\mathsf{verif}}_{\mathcal{V},k}(\mathcal{A}) \leq \epsilon$.

PLASMA [MST24] proved their $\mathcal{V}$ construction satisfied this property based on the XOR-collision resistance of the hash function used in the construction. [dCP22] proved the verifiability property of their verifiable distributed point function based on the XOR-collision resistance property of a hash function. The hash function is modeled as a random oracle to prove XOR-collision resistance.

B Security Definitions for shared ZK

Let $\mathcal{Z} = (\mathsf{Prove}, \mathsf{Query}, \mathsf{Decide}, \mathsf{Dom}, \mathsf{Rng})$ be a shared ZK scheme as defined in Section 2.4. In this section, we define the security properties we require for $\mathcal{Z}$. These are comparable to definitions from [BBC+19, Section 6] but apply to our syntax.

Completeness. We say that $\mathcal{Z}$ is *complete* if for all $b \in \{0,1\}$, $vk \in \{0,1\}^{\mathsf{vkl}}$, $\mathrm{H} : \mathsf{Dom} \to \mathsf{Rng}$ and $[\![x]\!]_0, [\![x]\!]_1$ such that $\mathsf{Extract}([\![x]\!]_0, [\![x]\!]_1) \in \mathcal{L}$, it holds that

$$\Pr[\mathsf{Decide}^{\mathrm{H}}(\sigma_0, \sigma_1, \mathsf{st}_b) = 1] = 1\,,$$

where the state and verifier shares were generated by running $(\pi_0^{\mathsf{szk}}, \pi_1^{\mathsf{szk}}, \mathsf{nonce}) \coloneqq \mathsf{Prove}^{\mathrm{H}}([\![x]\!]_0, [\![x]\!]_1)$ then $(\mathsf{st}_b, \sigma_b) \coloneqq \mathsf{Query}^{\mathrm{H}}(vk, \mathsf{nonce}, [\![x]\!]_b, \pi_b^{\mathsf{szk}})$ for $b \in \{0,1\}$.

Zero-Knowledge. The aggregators should learn nothing about a client's input while validating it as long as one aggregator is honest. To formalize this, let $\mathcal{A}$ be an adversary (the corrupt aggregator) and let Sim be a simulator with interfaces Init, Prove, Query, and Decide. We define games $\mathcal{G}_{\mathcal{Z}}^{\text{shared ZK-real}}(\mathcal{A})$ and $\mathcal{G}_{\mathsf{Sim}}^{\text{shared ZK-ideal}}(\mathcal{A})$ as shown in Figs. 7 and 8, respectively. Define the advantage of $\mathcal{A}$ in distinguishing execution of $\mathcal{Z}$ from simulator Sim as

$$\mathbf{Adv}^{\mathsf{priv}}_{\mathcal{Z},\mathsf{Sim}}(\mathcal{A}) \coloneqq \left| \Pr[\mathcal{G}_{\mathcal{Z}}^{\text{shared ZK-real}}(\mathcal{A}) \Rightarrow 1] - \Pr[\mathcal{G}_{\mathsf{Sim}}^{\text{shared ZK-ideal}}(\mathcal{A}) \Rightarrow 1] \right|.$$

Informally, we say that $\mathcal{Z}$ is *zero-knowledge* if there exists a PPT simulator for which no PPT adversary gets a non-negligible distinguishing advantage.

Soundness. The proof system is sound if a malicious client cannot fool honest aggregators into accepting an invalid measurement. Formally, define the advantage $\mathbf{Adv}^{\mathsf{sound}}_{\mathcal{Z}}(\mathcal{A})$ of $\mathcal{A}$ in breaking the soundness of $\mathcal{Z}$ as the probability that the following experiment outputs true:

1. Run $\mathcal{A}$ to get $(\mathsf{nonce}, [\![x]\!]_0, [\![x]\!]_1, \pi_0^{\mathsf{szk}}, \pi_1^{\mathsf{szk}})$.
2. Sample $vk \xleftarrow{\$} \{0,1\}^{\mathsf{vkl}}$.
3. Run $(\mathsf{st}_b, [\![\sigma]\!]_b) \coloneqq \mathsf{Query}(vk, \mathsf{nonce}, [\![x]\!]_b, [\![\pi^{\mathsf{szk}}]\!]_b)$ for each $b \in \{0,1\}$.
4. Output $(\exists\, b\ \mathsf{Decide}([\![\sigma]\!]_0, [\![\sigma]\!]_1, \mathsf{st}_b) = 1) \wedge \mathsf{Extract}([\![x]\!]_0, [\![x]\!]_1) \notin \mathcal{L}$.

The adversary wins if the output is true, i.e., the measurement is invalid but the proof verifies. Informally, we say that $\mathcal{Z}$ is *ϵ-sound* if for all PPT $\mathcal{A}$ it holds that $\mathbf{Adv}^{\mathsf{sound}}_{\mathcal{Z}}(\mathcal{A}) \leq \epsilon$.

Similar to the candidate standard for Prio [BCPP22], the shared ZK system $\mathcal{Z}$ can be instantiated from a fully linear proof (FLP) system [DPRS23]. In the remainder, we give the high level idea of the construction.

$\mathcal{G}^{\text{shared ZK-real}}_{\mathcal{Z}}$

Init(vk, b)

1. $\mathsf{H} \xleftarrow{\$} \mathcal{AF}(\mathsf{Dom}, \mathsf{Rng})$
2. Store vk and b in memory.

Prove$([\![x]\!]_0, [\![x]\!]_1)$

We require that $\mathsf{Extract}([\![x]\!]_0, [\![x]\!]_1) \in \mathcal{L}$

1. $(\mathsf{nonce}, \pi^{\mathsf{szk}}{}_0, \pi^{\mathsf{szk}}{}_1) \xleftarrow{\$} \mathcal{Z}.\mathsf{Prove}^{\mathrm{H}}\ ([\![x]\!]_0, [\![x]\!]_1)$
2. $T_{\text{client}}[\mathsf{nonce}] \leftarrow [\![x]\!]_{1-b}, \pi^{\mathsf{szk}}{}_{1-b}$
3. return $(\mathsf{nonce}, \pi^{\mathsf{szk}}{}_b)$

Query(nonce)

1. $([\![\sigma]\!], \mathsf{st}) \leftarrow \mathcal{Z}.\mathsf{Query}^{\mathrm{H}}(vk, 1-b, \mathsf{nonce}, T_{\text{client}}[\mathsf{nonce}])$
2. $T_{\text{sZK}}[\mathsf{nonce}] \leftarrow [\![\sigma]\!], \mathsf{st}$
3. return $[\![\sigma]\!]$

Decide$(\mathsf{nonce}, [\![\sigma]\!]_b)$

1. $[\![\sigma]\!]_{1-b}, \mathsf{st} \leftarrow T_{\text{sZK}}[\mathsf{nonce}]$
2. return $\mathcal{Z}.\mathsf{Decide}^{\mathrm{H}}([\![\sigma]\!]_0, [\![\sigma]\!]_1, \mathsf{st})$

RO(d)

1. return $\mathrm{H}(d)$

Fig. 7: Real game for defining zero-knowledge of shared ZK scheme $\mathcal{Z}$.

$\mathcal{G}^{\text{shared ZK-ideal}}_{\mathsf{Sim}_{\text{SZK}}}$

Init(vk, b)

1. Store vk and b in memory.

Prove$([\![x]\!]_0, [\![x]\!]_1)$

We require that $\mathsf{Extract}([\![x]\!]_0, [\![x]\!]_1) \in \mathcal{L}$

1. $(\mathsf{nonce}, \pi^{\mathsf{szk}}{}_b, \mathsf{st}) \xleftarrow{\$} \mathsf{Sim}_{\text{SZK}}.\mathsf{Prove}(\mathsf{st}, b)$
2. return $(\mathsf{nonce}, \pi^{\mathsf{szk}}{}_b)$

Query(nonce)

1. $([\![\sigma]\!], \mathsf{st}) \leftarrow \mathsf{Sim}_{\text{SZK}}.\mathsf{Query}(vk, 1-b, \mathsf{nonce}, \mathsf{st})$
2. return $[\![\sigma]\!]$

Decide$(\mathsf{nonce}, [\![\sigma]\!]_b)$

1. return $\mathsf{Sim}_{\text{SZK}}.\mathsf{Decide}([\![\sigma]\!]_b, 1-b, \mathsf{nonce}, \mathsf{st})$

RO(i, d)

1. $\mathsf{st}, h \leftarrow \mathsf{Sim}_{\text{SZK}}.\mathsf{RO}(\mathsf{st}, i, d)$
2. return h

Fig. 8: Ideal game for defining zero-knowledge for shared ZK scheme $\mathcal{Z}$.

First, FLPs are designed to provide the same properties as conventional zero-knowledge proof systems (for membership in a finite language $\mathcal{L} \subseteq \mathbb{F}^m$); in a addition, they are "fully linear" in the sense the verifier's computations over the input and proof is linear. This allows verification to be distributed amongst multiple verifiers, each of which holds only a share of the input and proof.

FLP is the core component of the shared ZKproof system; what remains for $\mathcal{Z}$ to specify is 1) secret sharing of the FLP itself and 2) generation of shared randomness used by the prover and verifiers. The FLP used in Prio [BCPP22, Section 7.3] involves "joint randomness" used for proof generation and eevaluation and "query randomness" used by the verifiers to check the proof's correctness: in Prio, the former is derived from the input shares using an extension of the Fiat-Shamir heuristic to proofs on secret shared data; the query randomness is derived by applying a PRF to the nonce generating by client and using the key vk shared by the verifiers.

We refer to [DPRS23, Section 4] for concrete security bounds. Note that their security model for privacy and robustness does not immediately yield bounds for our setting.

C Proof of Theorem 1 (Π_{Mastic} is Private)

In this section, we prove Theorem 1, which for convenience we restate here.

$\mathcal{G}^{\text{priv-real}}_{\text{Mastic}}$

At the start of the game, the adversary outputs a key vk and the index b of the corrupt aggregator and the game samples $\text{H}_1 \xleftarrow{\$} \mathcal{AF}(\mathcal{V}.\text{Dom}, \mathcal{V}.\text{Rng})$, and $\text{H}_2 \xleftarrow{\$} \mathcal{AF}(\mathcal{Z}.\text{Dom}, \mathcal{Z}.\text{Rng})$, initializes lists T_{client} and T_{agg}, and sets $i := 0$.

Honest Client Computation:

Input: measurement $(\alpha, \beta) \in (\{0,1\}^n, \mathcal{L})$.

1. $i := i + 1$
2. Generate $(\mathsf{nonce}, \mathsf{pub}, \{\mathsf{key}_c, \pi^{\mathsf{szk}}_c\}_{c\in\{0,1\}})$ as in Fig. 2.
3. $T_{\text{client}}[i] := (\varepsilon, \mathsf{nonce}, \mathsf{pub}, \mathsf{key}_{1-b}, \pi^{\mathsf{szk}}_{1-b})$
4. $\mathsf{st}^{\varepsilon}_{(i,1-b)} := \pi^{\varepsilon}_{(i,1-b)} := \varepsilon$
5. Return $(\mathsf{nonce}, \mathsf{pub}, \mathsf{key}_b, \pi^{\mathsf{szk}}_b)$

Corrupt Client Computation:

Input: report share $(\mathsf{nonce}, \mathsf{pub}, \mathsf{key}_{1-b}, \pi^{\mathsf{szk}}_{1-b})$

1. $i := i + 1$
2. If $T_{\text{client}}[i] \neq \perp$ return $\perp$
3. $\mathsf{st}^{\varepsilon}_{(i,1-b)} := \pi^{\varepsilon}_{(i,1-b)} := \varepsilon$
4. $T_{\text{client}}[i] := (\varepsilon, \mathsf{nonce}, \mathsf{pub}, \mathsf{key}_{1-b}, \pi^{\mathsf{szk}}_{1-b})$

Aggregation

Input: set Reports of report indices and a prefix p

1. if $T_{\text{agg}}[p] \neq \perp$ then return $\perp$
2. else $T_{\text{agg}}[p] := \mathsf{Reports}$; $[\![\mathsf{weight}]\!]_{1-b} := 0$
3. for i in Reports
4. $\quad [\![\mathsf{weight}]\!]_{1-b} := [\![\mathsf{weight}]\!]_{1-b} + [\![y^p_i]\!]_{1-b}$
5. return $[\![\mathsf{weight}]\!]_{1-b}$

Random Oracle:

Input: Index j and payload d. Outputs an independent random string (or vector)

1. return $\text{H}_j(d)$

Honest Aggregator Evaluation:

Input: report index i and prefix p

1. $(\mathsf{st}, \mathsf{nonce}, \mathsf{pub}, \mathsf{key}_{1-b}, \pi^{\mathsf{szk}}_{1-b}) := T_{\text{client}}[i]$
2. if $\mathsf{st} = \varepsilon$ then
3. $\quad [\![y^{\varepsilon}_i]\!]_{1-b} := \mathcal{V}.\text{EvalRoot}^{\text{H}_1}(\mathsf{key}_{1-b}, \mathsf{pub})$
4. $\quad (\mathsf{st}^{\mathsf{szk}}, \sigma) := \mathcal{Z}.\text{Query}^{\text{H}_2}(vk, \mathsf{nonce}, [\![y^{\varepsilon}_i]\!]_{1-i}, \pi^{\mathsf{szk}})_{1-b}$
5. $\quad T_{\text{client}}[i] := ((\mathsf{st}^{\mathsf{szk}}, \sigma), \mathsf{nonce}, \mathsf{pub}, \mathsf{key}_{1-b}, \pi^{\mathsf{szk}}_{1-b})$
6. $\quad$ return σ
7. if $(\mathsf{st}^p_{(i,1-b)} = \varepsilon$ or $[\![y^p_i]\!]_{1-b} = \varepsilon)$ and $p \neq \varepsilon$
8. $\quad$ return $\perp$
9. for $\gamma \in \{p \,\|\, 0, p \,\|\, 1\}$
10. $\quad ([\![y^{\gamma}_i]\!]_{1-b}, \mathsf{st}^{\gamma}_{(i,1-b)}, \pi^{\gamma}_{(i,1-b)}) :=$
$\quad \mathcal{V}.\text{Eval}^{\text{H}_1}(\mathsf{key}_{1-b}, \mathsf{pub}, \gamma, \mathsf{st}^p_{(i,1-b)}, \pi^p_{(i,1-b)})$
11. $h^p_{(i,1-b)} := (-1)^{1-b} \cdot ([\![y^p_i]\!]_{1-b} - [\![y^{p\|0}_i]\!]_{1-b} - [\![y^{p\|1}_i]\!]_{1-b})$
12. return $\pi^p_{(i,1-b)}, \pi^{p\|0}_{(i,1-b)}, \pi^{p\|1}_{(i,1-b)}, h^p_{(i,1-b)}$

Honest Aggregator Validation:

Input: report index i and partial shared ZK verifier σ_b

1. $(\mathsf{st}, \mathsf{nonce}, \mathsf{pub}, \mathsf{key}, \pi^{\mathsf{szk}}) := T_{\text{client}}[i]$
2. if $\mathsf{st} \in \{\perp, \varepsilon\}$
3. $\quad$ return $\perp$
4. $(\mathsf{st}^{\mathsf{szk}}, \sigma_{1-i}) := \mathsf{st}$
5. $T_{\text{client}}[i] \leftarrow (\perp, \mathsf{nonce}, \mathsf{pub}, \mathsf{key}, \pi^{\mathsf{szk}})$
6. return $\mathcal{Z}.\text{Decide}(\sigma_0, \sigma_1, \mathsf{st}^{\mathsf{szk}})$

Fig. 9: Real game for defining privacy of Π_{Mastic}.

Theorem. For any simulator $\mathsf{Sim}_{\text{SZK}}$, there exists a simulator Sim such that for any adversary $\mathcal{A}$, there exist $\mathcal{Z}$-attacker $\mathcal{B}$ and $\mathcal{V}$-attacker $\mathcal{D}$ such that

$$\mathbf{Adv}^{\text{priv}}_{\text{Mastic},\mathsf{Sim}}(\mathcal{A}) \leq \mathbf{Adv}^{\text{priv}}_{\mathcal{Z},\mathsf{Sim}_{\text{SZK}}}(\mathcal{B}) + q \cdot \mathbf{Adv}^{\text{priv}}_{\mathcal{V}}(\mathcal{D}),$$

where q is the number of queries made to the "Honest Client Computation" oracle, and the runtime of $\mathcal{D}$ is about that of an honest aggregator in the Mastic protocol, and the runtime of $\mathcal{B}$ is about that of an honest aggregator plus the time to run $\mathsf{Sim}_{\text{SZK}}$ once per interaction with the honest aggregator.

The privacy advantage of $\mathcal{A}$ against the mastic protocol is its advantage in distinguishing between games $\mathcal{G}^{\text{priv-real}}_{\text{Mastic}}$ (c.f. Figure 9) and $\mathcal{G}^{\text{priv-ideal}}_{\text{Sim}}$ (c.f. Figure 10). We will establish the claim by constructing a simulator

$\mathcal{G}^{\text{priv-ideal}}_{\mathsf{Sim}}$

The simulator Sim is stateful and defines interfaces Init, $\mathsf{GenReport}$, $\mathsf{AcceptReport}$, Query, $\mathsf{ValidateSZK}$, $\mathsf{Aggregate}$, and RO. At the beginning of the game, the adversary outputs a key vk and the index b of the corrupt aggregator, and the game sets $i \coloneqq 0$, initializes list T_{client}, and runs $\mathsf{Sim.Init}(vk, b)$.

Honest Client Computation:

Input: measurement $(\alpha, \beta) \in \{0,1\}^n \times \mathcal{L}$

1. $i \coloneqq i + 1$
2. $(\mathsf{nonce}, \mathsf{pub}, \mathsf{key}_b, \pi_b^{\mathsf{szk}}) \coloneqq \mathsf{Sim.GenReport}(i)$.
3. $T_{\text{client}}[i] \coloneqq (\alpha, \beta)$
4. Return $(\mathsf{nonce}, \mathsf{pub}, \mathsf{key}_b, \pi_b^{\mathsf{szk}})$

Corrupt Client Computation:

Input: report share $(\mathsf{nonce}, \mathsf{pub}, \mathsf{key}_{1-b}, \pi_{1-b}^{\mathsf{szk}})$

1. $i \coloneqq i + 1$
2. $\mathsf{Sim.AcceptReport}(i, \mathsf{nonce}, \mathsf{pub}, \pi^{\mathsf{szk}}, \mathsf{key})$

Random Oracle:

Input: Index j and payload d

1. return $\mathsf{Sim.RO}(j, d)$

Honest Aggregator Evaluation:

Input: report index i and prefix p.

1. $(\text{pf}, \mathsf{st}_{\mathsf{Sim}_{\text{SZK}}}) \coloneqq \mathsf{Sim.Query}(i, p)$
   ▷ Depending on the state, pf may contain a verifier string σ or a tuple $(\pi^p_{(i,1-b)}, \pi^{p\|0}_{(i,1-b)}, \pi^{p\|1}_{(i,1-b)}, h^p_{(i,1-b)})$
2. return pf

Honest Aggregator Validation:

Input: Report index i and partial sharedZK verifier string σ

1. return $\mathsf{Sim.ValidateSZK}(i, \sigma)$

Aggregation

Input: set $\mathsf{Reports}$ of report indices, and a prefix p.

1. $\text{a} \coloneqq 0$
2. for $i \in \mathsf{Reports}$
3. $\quad(\alpha, \beta) \coloneqq T_{\text{client}}[i]$
4. $\quad$if p is a prefix of α then $\text{a} \coloneqq \text{a} + \beta$.
5. return $\mathsf{Sim.Aggregate}(p, N, \text{a})$

Fig. 10: Ideal game for defining the privacy of Π_{Mastic}. Let Sim be a simulator.

Sim and adversaries $\mathcal{B}_{\mathcal{Z}}$ against the zero-knowledge security of the shared ZK scheme and $\mathcal{B}_{\mathcal{V}}$ against the privacy of the VIDPF scheme. Then we transform game $\mathcal{G}^{\text{priv-ideal}}_{\mathsf{Sim}}$ into game $\mathcal{G}^{\text{priv-real}}_{\mathsf{Mastic}}$ via a series of hybrid transitions and bound the distinguishing advantage between each pair of hybrids.

First, we define simulator Sim in Figure 11. This simulator mimics the behavior of an honest aggregator in the Mastic protocol without knowledge of any client's underlying measurements. Broadly, the simulator handles two types of client reports: honest clients for whom the simulator must generate the malicious aggregator's report share, and malicious clients for whom the simulator receives only its own (maliciously generated) report share. The simulator processes these two types of reports separately in every oracle.

Let us first discuss maliciously generated client reports. The simulator receives these reports through the $\mathsf{AcceptReport}$ interface, and it simply stores them with a tag "mal" denoting that the share is malicious. In all other oracles, whenever the malicious tag is detected, the simulator honestly runs the Mastic protocol using its stored report share. Clearly, the behavior of the simulator and of an honest aggregator are identical for all malicious reports.

For honest client reports, the simulator's behavior is slightly more complex. In the $\mathsf{GenReport}$ interface, the simulator receives only an index i indicating that a new client report share should be generated. This report share must contain a nonce, a $\mathcal{V}$ public share and key, and a shared ZK partial proof, all corresponding to a measurement that the simulator does not know. Instead, the simulator picks its own measurement (α_i, β_i) at random from the set of all valid measurements, and creates a $\mathcal{V}$ public share and two keys by running the honest key generation algorithm. To hide the fact that these keys correspond to the wrong measurement, it

immediately discards its own key key_{1-b}. It then uses the zero-knowledge simulator $\mathsf{Sim}_{\mathrm{SZK}}$ to generate a simulated partial proof and nonce $(\mathsf{nonce}, \pi_b^{\mathsf{szk}})$. Before it returns all these values to the adversary, it stores them with a tag "honest" indicating that the client is honest (and, by extension, that the simulator only knows the malicious aggregator's report share).

We must also simulate the validation of honest client reports, despite the lack of an honest report share. For shared ZK validation, the simulator refers again to the shared ZK zero-knowledge simulator to produce a verifier share and a decision. The remaining two portions of report validation are equality checks: the honest aggregator must produce a one-hot verifiability proof π for every prefix and level, and a path-verifiability proof h. Although we cannot compute these proofs directly, we know that for an honestly generated client report, the malicious and honest aggregators should always derive equal-valued proofs for both one-hot and path verifiability by the correctness of the VIDPF scheme. The simulator therefore uses its stored malicious report share to compute the malicious aggregator's proofs instead of the honest aggregator's, knowing they will be equal.

Formal Proof. The first change we make is to replace, one by one, the randomly sampled α_i, β_i in line 2 of $\mathsf{GenReport}$. Let q be the number of queries $\mathcal{A}$ makes to the Honest Client Computation oracle. We design q hybrids, indexed by the integers j from 0 to q. In hybrid HYB_j(c.f. Figure 12), the first j queries to the Honest Client Computation oracle will compute $\mathsf{pub}, \mathsf{key}_b$ by running $\mathsf{GenReport}$ honestly. In the remaining queries, it will instead compute $(\mathsf{pub}, \mathsf{key}_0, \mathsf{key}_1) \coloneqq \mathcal{V}.\mathsf{Gen}(\alpha, \beta)$, then discard the key key_{1-b}. HYB_0 is clearly identical to $\mathcal{G}_{\mathsf{Sim}}^{\mathsf{priv\text{-}ideal}}$, while HYB_q uses the honest client measurement to generate keys for every query. The distinction between HYB_j and HYB_{j+1} is that the $(j+1)^{\text{th}}$ Honest Client Computation query generates $(\mathsf{pub}, \mathsf{key}_b)$ from either the real measurement (α, β) or a randomly sampled measurement. Consequently, for each j we can build a reduction $\mathcal{D}$ that breaks the privacy of $\mathcal{V}$ with exactly the same probability that $\mathcal{A}$ distinguishes between HYB_j and HYB_{j+1}.

Our reduction $\mathcal{D}$ runs the hybrid HYB_j for $\mathcal{A}$. When the simulated "Honest Client Computation" oracle receives the $(j+1)^{\text{th}}$ query, it submits the client's measurement (α, β) as its own challenge along with the corrupt aggregator's index b. It uses the response $\mathsf{pub}, \mathsf{key}_b$ in the corrupt aggregator's report share. If the challenge bit b in the $\mathcal{V}$ privacy game equals 0, the simulation of the j^{th} hybrid is perfect. Otherwise, $\mathcal{D}$ perfectly simulates the $(j+1)^{\text{th}}$ hybrid. We define $\mathcal{D}$ to return 1 when $\mathcal{A}$ returns 1 and return 0 otherwise. Then

$$\mathbf{Adv}_{\mathcal{V}}^{\mathsf{priv}}(\mathcal{D}) \coloneqq \left| \Pr[\mathcal{G}_{\mathcal{V},1}^{\mathsf{priv}}(\mathcal{D}) \Rightarrow 1] - \Pr[\mathcal{G}_{\mathcal{V},0}^{\mathsf{priv}}(\mathcal{D}) \Rightarrow 1] \right| \leq \left| \Pr[\mathcal{A} \Rightarrow 1 | \mathsf{HYB}_{j+1}] - \Pr[\mathcal{A} \Rightarrow 1 | \mathsf{HYB}_j] \right|.$$

By a union bound over all $j \in [q]$, the probability of $\mathcal{A}$ distinguishing between hybrids HYB_0 and HYB_j the two oracles is at most

$$\left| \Pr[\mathcal{A} \Rightarrow 1 | \mathsf{HYB}_q] - \Pr[\mathcal{A} \Rightarrow 1 | \mathsf{HYB}_0] \right| \leq q \cdot \mathbf{Adv}_{\mathcal{V}}^{\mathsf{priv}}(\mathcal{D}).$$

In our next hybrid, HYB_{q+1}(c.f. Figure 13), we stop relying on the shared ZK simulator $\mathsf{Sim}_{\mathrm{SZK}}$. We replace all calls to $\mathsf{Sim}_{\mathrm{SZK}}$ with the corresponding $\mathcal{Z}$ scheme operations. We first stop referring random oracle queries with $j = 2$ to the simulator and instead, sample a random function H_2. Then in $\mathsf{GenReport}$ we compute shares $[\![y_i^\epsilon]\!]_0$ and $[\![y_i^\epsilon]\!]_1$ of β. We can do this by using key_b to compute $[\![y_i^\epsilon]\!]_b$ via $\mathsf{EvalRoot}$, then relying on the completeness of $\mathcal{V}$ and our knowledge of β to find $[\![y_i^\epsilon]\!]_{1-b}$. Then we call $\mathcal{Z}.\mathsf{Prove}^{Hash_2}$ on these shares to generate the nonce and proofs. We store the shares and proofs, and in the Honest Aggregator Evaluation oracle, we compute the verifier share using $\mathcal{Z}.\mathsf{Query}^{\mathrm{H}_2}(vk, \mathsf{nonce}, [\![y_i^\epsilon]\!]_{1-b}, \pi_{1-b}^{\mathsf{szk}})$.

We then design a reduction $\mathcal{B}$ whose advantage in breaking the zero-knowledge security of $\mathcal{Z}$ is exactly the probability that $\mathcal{A}$ distinguishes hybrid HYB_{q+1} from HYB_q. Our reduction runs HYB_{q+1}, with a few changes to the highlighted lines. When HYB_{q+1} would query $\mathcal{Z}.\mathsf{Prove}$ on the input shares in line 5 of the Honest Client Computation oracle, it instead calls its own **Prove** oracle to get back a nonce and proof. In line 9 of the Honest Aggregator Evaluation oracle, it queries **Query** on the honest client's nonce to get back a partial verifier string σ_{1-b}. Finally, in line 8 of the Honest Aggregator Validation oracle, the reduction queries its **Decide** oracle on the nonce and malicious verifier share instead of calling Decide. Additionally, everywhere

HYB_{q+1} would call H_2, the reduction instead forwards the query to its own random oracle. Finally, $\mathcal{B}$ returns 1 if and only if $\mathcal{A}$ returns 1 in its simulated hybrid.

If we consider the behavior of the $\mathcal{B}$ when it plays $\mathcal{G}^{\text{shared ZK-real}}_{\mathcal{Z}}$, notice that like HYB_{q+1}, the random oracle implements a randomly sampled function from the correct set, and the Prove, Query, and Decide oracles run $\mathcal{Z}$ on the proper inputs exactly as HYB_{q+1} would. Therefore the probability that $\mathcal{B}$ returns 1 is exactly $\Pr[\mathcal{A} \Rightarrow 1|\mathsf{HYB}_{q+1}]$. Conversely, when $\mathcal{B}$ is playing $\mathcal{G}^{\text{shared ZK-ideal}}_{\mathsf{Sim}_{\mathrm{SZK}}}$, the random oracle and the Prove, Query, and Decide forward their inputs to $\mathsf{Sim}_{\mathrm{SZK}}$ exactly as HYB_q would, and we have that $\Pr[\mathcal{B} \Rightarrow 1|\mathcal{G}^{\text{shared ZK-ideal}}_{\mathsf{Sim}_{\mathrm{SZK}}}] = \Pr[\mathcal{A} \Rightarrow 1|\mathsf{HYB}_q]$. Therefore,

$$\left|\Pr[\mathcal{A} \Rightarrow 1|\mathsf{HYB}_{q+1}] - \Pr[\mathcal{A} \Rightarrow 1|\mathsf{HYB}_q]\right| \leq \mathbf{Adv}^{\mathsf{priv}}_{\mathcal{Z},\mathsf{Sim}_{\mathrm{SZK}}}(\mathcal{B}).$$

In our next hybrid, HYB_{q+2}(c.f. Figure 14), we are now ready to stop discarding the honest aggregator's VIDPF key key_{1-b}. Instead of computing the honest input share as $\beta - \mathcal{V}.\mathsf{EvalRoot}^{\mathrm{H}}(\mathsf{key}_b, \mathsf{pub})$, we start to compute it directly as $\mathcal{V}.\mathsf{EvalRoot}^{\mathrm{H}}(\mathsf{key}_{1-b}, \mathsf{pub})$. These values are equivalent due to the correctness of the VIDPF, so the adversary's view of the Honest Client Computation oracle does not change between HYB_{q+1} and HYB_{q+2}.

The next change is that we also use key_{1-b} in the Honest Aggregator Evaluation oracle to generate VIDPF proofs for honest reports. To enable this change, we now store key_{1-b} instead of key_b in $\mathsf{st}[i]$ for all honest reports. This means that the output shares and proofs generated by $\mathcal{V}.\mathsf{Eval}$ are now produced from the honest aggregator's key share instead of the malicious aggregator's key share. By $\mathcal{V}$ correctness, however, the proofs π_i^p are equal for every prefix p regardless of which aggregator computes them, so the change in their derivation is undetectable.

What is left is to consider the path-verifiability check h output by Sim.Query. By $\mathcal{V}$ correctness, we have that for any $k \in [n]$ and any $p \in \{0,1\}^k$, it holds that $[\![y_i^p]\!]_b = \beta - [\![y_i^p]\!]_{1-b}$ if p is a prefix of α and $[\![y_i^p]\!]_b = -[\![y_i^p]\!]_{1-b}$ otherwise. Consequently, we have that:

$$[\![y_i^p]\!]_b - [\![y_i^{p\|0}]\!]_b - [\![y_i^{p\|1}]\!]_b = \begin{cases} \beta - [\![y_i^p]\!]_{1-b} - \beta + [\![y_i^{p\|0}]\!]_{1-b} + [\![y_i^{p\|1}]\!]_{1-b}, & \text{if } p \text{ is a prefix of } \alpha, \\ -[\![y_i^p]\!]_{1-b} + [\![y_i^{p\|0}]\!]_{1-b} + [\![]\!y_i^{p\|1}{}_{1-b}, & \text{otherwise.} \end{cases}$$

When β cancels out, we see that:

$$[\![y_i^p]\!]_b - [\![y_i^{p\|0}]\!]_b - [\![y_i^{p\|1}]\!]_b = (-1)([\![y_i^p]\!]_{1-b} - [\![y_i^{p\|0}]\!]_{1-b} - [\![y_i^{p\|1}]\!]_{1-b}).$$

Accordingly, when we switch from key_b to key_{1-b} in the derivation of h, we must also multiply it by -1. With this tweak, the behavior of the Honest Aggregator Evaluation oracle in both hybrids is identical in the view of the adversary.

Now that we are storing key_{1-b} for all oracles, we must change the Aggregation oracle to maintain the consistency of the results. For each honest client reports $i \in \mathsf{Reports}$, in HYB_{q+1} we added the stored weight β to $[\![\mathsf{weight}]\!]_{1-b}$ when p prefixed α. Then we subtracted the malicious aggregator's share $[\![y_i^p]\!]_b$ from $[\![\mathsf{weight}]\!]_{1-b}$. Since we can no longer compute the malicious aggregator's share, in HYB_{q+2} we stop adding β and subtracting $[\![y_i^p]\!]_{1-b}$. Instead we add the honest aggregator's share $[\![y_i^p]\!]_{1-b}$ to $[\![\mathsf{weight}]\!]_{1-b}$. The correctness of $\mathcal{V}$ grants that $[\![y_i^p]\!]_{1-b} = \beta - [\![y_i^p]\!]_b$ when p prefixes α and $-[\![y_i^p]\!]_b$ otherwise, so the substitution we perform gives an identical value of $[\![\mathsf{weight}]\!]_{1-b}$, and consequently

$$\Pr[\mathcal{A} \Rightarrow 1|\mathsf{HYB}_{q+2}] = \Pr[\mathcal{A} \Rightarrow 1|\mathsf{HYB}_{q+1}].$$

Notice that in HYB_{q+2}, we now treat both honest and malicious client reports identically regardless of the value of the tags ("honest" and "mal") in all oracles (except GenReport and AcceptReport, since we must generate the report ourselves in the former). Furthermore, the values computed and returned by each oracle in HYB_{q+2} are identical to those returned in $\mathcal{G}^{\mathsf{priv\text{-}real}}_{\mathsf{Mastic}}$, so

$$\Pr[\mathcal{A} \Rightarrow 1|\mathsf{HYB}_{q+2}] = \Pr[\mathcal{A} \Rightarrow 1|\mathcal{G}^{\mathsf{priv\text{-}real}}_{\mathsf{Mastic}}].$$

Collecting bounds proves the claim.

$\mathsf{Sim}_{\mathsf{Sim}_{\mathrm{SZK}}}(vk, b)$

The simulator Sim is stateful and defines interfaces Init, GenReport, AcceptReport, Query, ValidateSZK, Aggregate, and RO.

Sim.Init(vk, b)

1. $\mathsf{st}[\mathcal{Z}] \leftarrow \epsilon$
2. $\mathrm{H}_1 \xleftarrow{\$} \mathcal{AF}(\mathcal{V}.\mathsf{Dom}, \mathcal{V}.\mathsf{Rng})$
3. Store vk, b, and st as globally available state.

Sim.GenReport(i)

1. if $\mathsf{st}[i] \neq \perp$ return $\perp$
2. $\alpha_i, \beta_i \xleftarrow{\$} \{0,1\}^n \times \mathcal{L}$
3. $(\mathsf{pub}, \mathsf{key}_0, \mathsf{key}_1) := \mathcal{V}.\mathsf{Gen}(\alpha_i, \beta_i)$
4. $(\mathsf{nonce}, \pi_b^{\mathsf{szk}}, \mathsf{st}[\mathcal{Z}]) := \mathsf{Sim}_{\mathrm{SZK}}.\mathsf{Prove}(\mathsf{st}[\mathcal{Z}, b])$
5. $\mathsf{st}[\mathcal{V}_i^\epsilon] := \mathsf{st}[\pi_i^\epsilon] := \epsilon$
6. $\mathsf{st}[i] := (\text{“ honest”}, \epsilon, (\mathsf{nonce}, \mathsf{pub}, \mathsf{key}_b, \pi_b^{\mathsf{szk}}))$
7. Return $(\mathsf{nonce}, \mathsf{pub}, \mathsf{key}_b, \pi_b^{\mathsf{szk}})$

Sim.AcceptReport(i, nonce, pub, key_{1-b}, π_{1-b}^{szk})

1. if $\mathsf{st}[i] \neq \perp$ return $\perp$
2. $\mathsf{st}[\mathcal{V}_i^\epsilon] := \mathsf{st}[\pi_i^\epsilon] := \epsilon$
3. $\mathsf{st}[i] \leftarrow (\text{“mal”}, \epsilon, (\mathsf{nonce}, \mathsf{pub}, \mathsf{key}_{1-b}, \pi_{1-b}^{\mathsf{szk}}))$

Sim.ValidateSZK(i, σ_b)

1. $\mathsf{hon}, \mathsf{status}, \mathsf{params} := \mathsf{st}[i]$
2. if $\mathsf{status} \in \{\epsilon, \perp\}$ return $\perp$
3. $(\mathsf{nonce}, \mathsf{pub}, \mathsf{key}, \pi^{\mathsf{szk}}) := \mathsf{params}$
4. $\sigma_{1-b} \leftarrow \mathsf{status}$
5. $\mathsf{st}[i] \leftarrow (\mathsf{hon}, \perp, \mathsf{params})$
6. if $\mathsf{hon} = \text{“mal”}$
7. return $\mathcal{Z}.\mathsf{Decide}(\sigma_0, [\![]\!]\sigma_1, \mathsf{st}[\mathcal{Z}, i])$
8. return $\mathsf{Sim}_{\mathrm{SZK}}.\mathsf{Decide}(\sigma_b, 1-b, \mathsf{nonce}, \mathsf{st}[\mathcal{Z}])$

Sim.RO(j, d)

1. if $j = 2$ then return $\mathsf{Sim}_{\mathrm{SZK}}.\mathsf{RO}(d)$
2. else return $\mathrm{H}_1(d)$

Sim.Query(i, p)

1. $\mathsf{hon}, \mathsf{status}, \mathsf{params} := \mathsf{st}[i]$
2. if $(\mathsf{st}[\mathcal{V}_i^p] = \varepsilon$ or $\mathsf{st}[[\![y_i^p]\!]] = \epsilon)$ and $p \neq \varepsilon$ return $\perp$
3. $(\mathsf{nonce}, \mathsf{pub}, \mathsf{key}, \pi^{\mathsf{szk}}) := \mathsf{params}$
4. if $\mathsf{status} = \epsilon$
5. $\mathsf{st}[\![y_i^\epsilon]\!] := \mathcal{V}.\mathsf{EvalRoot}^{\mathrm{H}_1}(\mathsf{key}, \mathsf{pub})$
6. if $\mathsf{hon} = \text{“mal”}$
7. $(\mathsf{st}[\mathcal{Z}, i], \sigma_{1-b}) := \mathcal{Z}.\mathsf{Query}(vk, \mathsf{nonce}, [\![y_i^\epsilon]\!], \pi^{\mathsf{szk}})$
8. else
9. $(\mathsf{st}[\mathcal{Z}], \sigma_{1-b}) := \mathsf{Sim}_{\mathrm{SZK}}.\mathsf{Query}(vk, 1-b, \mathsf{nonce}, \mathsf{st}[\mathcal{Z}])$
10. $\mathsf{st}[i] \leftarrow (\mathsf{hon}, \sigma_{1-b}, \mathsf{params})$
11. return σ_{1-b}
12. for $\gamma \in \{p \,\|\, 0, p \,\|\, 1\}$
13. $(\mathsf{st}[[\![y_i^\gamma]\!]], \mathsf{st}[\mathcal{V}_i^\gamma], \mathsf{st}[\pi_i^\gamma]) := \mathcal{V}.\mathsf{Eval}^{\mathrm{H}_1}(\mathsf{pub}, \mathsf{key}, \gamma, \mathsf{st}[\mathcal{V}_i^p], \mathsf{st}[\pi_i^p])$
14. if $\mathsf{hon} = \text{“mal”}$
15. $h := (-1)^{1-b} \cdot (\mathsf{st}[[\![y_i^p]\!]] - \mathsf{st}[[\![y_i^{p\|0}]\!]] - \mathsf{st}[[\![y_i^{p\|1}]\!]])$
16. else
17. $h := (-1)^{b} \cdot (\mathsf{st}[[\![y_i^p]\!]] - \mathsf{st}[[\![y_i^{p\|0}]\!]] - \mathsf{st}[[\![y_i^{p\|1}]\!]])$
18. return $\mathsf{st}[\pi_i^p], \mathsf{st}[\pi_i^{p\|0}], \mathsf{st}[\pi_i^{p\|1}], h$

Sim.Aggregate(p, N, a)

1. $[\![\mathsf{weight}]\!]_{1-b} := \mathrm{a}$; $k := |p|$
2. for $i \in \mathsf{Reports}$
3. $\mathsf{hon}, \mathsf{status}, \mathsf{params} := \mathsf{st}[i]$
4. $(\mathsf{nonce}, \mathsf{pub}, \mathsf{key}, \pi^{\mathsf{szk}}) := \mathsf{params}$
5. if $(p^{|^{k-1}} = \varepsilon$ or $\mathsf{st}[\mathcal{V}_i^{p^{|^{k-1}}}] \neq \varepsilon)$
6. $([\![y_i^p]\!], \_, \_) := \mathcal{V}.\mathsf{Eval}^{\mathrm{H}_1}(\mathsf{key}, \mathsf{pub}, p, \mathsf{st}[\mathcal{V}_i^{p^{|^{k-1}}}], \mathsf{st}[\pi_i^{p^{|^{k-1}}}])$
7. if $\mathsf{hon} = \text{“mal”}$
8. $[\![\mathsf{weight}]\!]_{1-b} := [\![\mathsf{weight}]\!]_{1-b} + [\![y_i^p]\!]$
9. else
10. $[\![\mathsf{weight}]\!]_{1-b} := [\![\mathsf{weight}]\!]_{1-b} - [\![y_i^p]\!]$
11. return $[\![\mathsf{weight}]\!]_{1-b}$

Fig. 11: Simulator Sim for the proof of Theorem 1. It takes as a parameter a simulator $\mathsf{Sim}_{\mathrm{SZK}}$ for the privacy of $\mathcal{Z}$.

D Proof of Theorem 2 (Π_{Mastic} is Robust)

In this section, we focus on the robustness guarantees provided by Mastic against malicious clients. To argue robustness against malicious clients, we assume the aggregators follow the Mastic protocol steps correctly. An adversary that maliciously corrupts multiple clients may attempt to disrupt the protocol by providing malformed report shares in Step. 3 of Fig. 2. A report share is considered to be malformed if 1) a client double-votes using the single report share, or 2) the report share contains an invalid measurement

HYB$_j$

At the beginning of the game, the adversary outputs a key vk and the index b of the corrupt aggregator, and the game sets $i \coloneqq 0$, initializes list T_{client}, and runs $\mathsf{Sim.Init}(vk, b)$.

Honest Client Computation:

Input: measurement $(\alpha, \beta) \in \{0,1\}^n \times \mathcal{L}$

1. $i \coloneqq i + 1$
2. $\alpha_i, \beta_i \xleftarrow{\$} \{0,1\}^n \times \mathcal{L}$
3. $\boxed{\text{if } i > j \text{ then } (\mathsf{pub}, \mathsf{key}_0, \mathsf{key}_1) \coloneqq \mathcal{V}.\mathsf{Gen}(\alpha_i, \beta_i)}$
4. $\boxed{\text{else } (\mathsf{pub}, \mathsf{key}_0, \mathsf{key}_1) \coloneqq \mathcal{V}.\mathsf{Gen}(\alpha, \beta)}$
5. $(\mathsf{nonce}, \pi_b^{\mathsf{szk}}, \mathsf{st}[\mathcal{Z}]) \coloneqq \mathsf{Sim}_{\mathrm{SZK}}.\mathsf{Prove}(\mathsf{st}[\mathcal{Z}, b])$
6. $\mathsf{st}[\mathcal{V}_i^\epsilon] \coloneqq \mathsf{st}[\pi_i^\epsilon] \coloneqq \epsilon$
7. $\mathsf{st}[i] \coloneqq (\text{“ honest”}, \epsilon, (\mathsf{nonce}, \mathsf{pub}, \mathsf{key}_b, \pi_b^{\mathsf{szk}}))$
8. $T_{\text{client}}[i] \coloneqq (\alpha, \beta)$
9. Return $(\mathsf{nonce}, \mathsf{pub}, \mathsf{key}_b, \pi_b^{\mathsf{szk}})$

Corrupt Client Computation:

Input: report share $(\mathsf{nonce}, \mathsf{pub}, \mathsf{key}_{1-b}, \pi_{1-b}^{\mathsf{szk}})$

1. $i \coloneqq i + 1$
2. $\mathsf{st}[\mathcal{V}_i^\epsilon] \coloneqq \mathsf{st}[\pi_i^\epsilon] \coloneqq \epsilon$
3. $\mathsf{st}[i] \leftarrow (\text{“mal”}, \epsilon, (\mathsf{nonce}, \mathsf{pub}, \mathsf{key}_{1-b}, \pi_{1-b}^{\mathsf{szk}}))$

Aggregation

Input: set Reports of report indices, and a prefix p.

1. $\mathrm{a} \coloneqq 0$
2. for $i \in \mathsf{Reports}$
3. $\quad (\alpha, \beta) \coloneqq T_{\text{client}}[i]$
4. $\quad$ if p is a prefix of α then $\mathrm{a} \coloneqq \mathrm{a} + \beta$.
5. $[\![\mathsf{weight}]\!]_{1-b} \coloneqq \mathrm{a}$; $k \coloneqq |p|$
6. for $i \in \mathsf{Reports}$
7. $\quad \mathsf{hon}, \mathsf{status}, \mathsf{params} \coloneqq \mathsf{st}[i]$
8. $\quad (\mathsf{nonce}, \mathsf{pub}, \mathsf{key}, \pi^{\mathsf{szk}}) \coloneqq \mathsf{params}$
9. $\quad$ if $(p^{|^{k-1}} = \varepsilon$ or $\mathsf{st}[\mathcal{V}_i^{p^{|^{k-1}}}] \neq \varepsilon)$
10. $\quad\quad ([\![y_i^p]\!], \_, \_) \coloneqq \mathcal{V}.\mathsf{Eval}^{\mathrm{H}_1}(\mathsf{key}, \mathsf{pub}, p, \mathsf{st}[\mathcal{V}_i^{p^{|^{k-1}}}], \mathsf{st}[\pi_i^{p^{|^{k-1}}}])$
11. $\quad\quad$ if $\mathsf{hon} =$ “mal”
12. $\quad\quad\quad [\![\mathsf{weight}]\!]_{1-b} \coloneqq [\![\mathsf{weight}]\!]_{1-b} + [\![y_i^p]\!]$
13. $\quad\quad$ else
14. $\quad\quad\quad [\![\mathsf{weight}]\!]_{1-b} \coloneqq [\![\mathsf{weight}]\!]_{1-b} - [\![y_i^p]\!]$
15. return $[\![\mathsf{weight}]\!]_{1-b}$

Honest Aggregator Evaluation:

Input: report index i and prefix p.

1. $\mathsf{hon}, \mathsf{status}, \mathsf{params} \coloneqq \mathsf{st}[i]$
2. if $(\mathsf{st}[\mathcal{V}_i^p] = \varepsilon$ or $\mathsf{st}[[\![y_i^p]\!]] = \epsilon)$ and $p \neq \varepsilon$ return $\perp$
3. $(\mathsf{nonce}, \mathsf{pub}, \mathsf{key}, \pi^{\mathsf{szk}}) \coloneqq \mathsf{params}$
4. if $\mathsf{status} = \epsilon$
5. $\quad \mathsf{st}[[\![y_i^\epsilon]\!]] \coloneqq \mathcal{V}.\mathsf{EvalRoot}^{\mathrm{H}_1}(\mathsf{key}, \mathsf{pub})$
6. $\quad$ if $\mathsf{hon} =$ “mal”
7. $\quad\quad (\mathsf{st}[\mathcal{Z}, i], \sigma_{1-b}) \coloneqq \mathcal{Z}.\mathsf{Query}^{\mathsf{Sim}_{\mathrm{SZK}}.\mathsf{RO}}(vk, \mathsf{nonce}, [\![y_i^\epsilon]\!], \pi^{\mathsf{szk}})$
8. $\quad$ else
9. $\quad\quad (\mathsf{st}[\mathcal{Z}], \sigma_{1-b}) \coloneqq \mathsf{Sim}_{\mathrm{SZK}}.\mathsf{Query}(vk, 1-b, \mathsf{nonce}, \mathsf{st}[\mathcal{Z}])$
10. $\quad \mathsf{st}[i] \leftarrow (\mathsf{hon}, \sigma_{1-b}, \mathsf{params})$
11. $\quad$ return σ_{1-b}
12. for $\gamma \in \{p \,\|\, 0, p \,\|\, 1\}$
13. $\quad (\mathsf{st}[[\![y_i^\gamma]\!]], \mathsf{st}[\mathcal{V}_i^\gamma], \mathsf{st}[\pi_i^\gamma]) \coloneqq \mathcal{V}.\mathsf{Eval}^{\mathrm{H}_1}(\mathsf{pub}, \mathsf{key}, \gamma, \mathsf{st}[\mathcal{V}_i^p], \mathsf{st}[\pi_i^p])$
14. if $\mathsf{hon} =$ “mal”
15. $\quad h \coloneqq (-1)^{1-b} \cdot (\mathsf{st}[[\![y_i^p]\!]] - \mathsf{st}[[\![y_i^{p\|0}]\!]] - \mathsf{st}[[\![y_i^{p\|1}]\!]])$
16. else
17. $\quad h \coloneqq (-1)^{b} \cdot (\mathsf{st}[[\![y_i^p]\!]] - \mathsf{st}[[\![y_i^{p\|0}]\!]] - \mathsf{st}[[\![y_i^{p\|1}]\!]])$
18. return $\mathsf{st}[\pi_i^p], \mathsf{st}[\pi_i^{p\|0}], \mathsf{st}[\pi_i^{p\|1}], h$

Honest Aggregator Validation:

Input: Report index i and partial sharedZK verifier string σ

1. $\mathsf{hon}, \mathsf{status}, \mathsf{params} \coloneqq \mathsf{st}[i]$
2. if $\mathsf{status} \in \{\epsilon, \perp\}$ return $\perp$
3. $(\mathsf{nonce}, \mathsf{pub}, \mathsf{key}, \pi^{\mathsf{szk}}) \coloneqq \mathsf{params}$
4. $\sigma_{1-b} \leftarrow \mathsf{status}$
5. $\mathsf{st}[i] \leftarrow (\mathsf{hon}, \perp, \mathsf{params})$
6. if $\mathsf{hon} =$ “mal”
7. $\quad$ return $\mathcal{Z}.\mathsf{Decide}^{\mathsf{Sim}_{\mathrm{SZK}}.\mathsf{RO}}(\sigma_0, \sigma_1, \mathsf{st}[\mathcal{Z}, i])$
8. return $\mathsf{Sim}_{\mathrm{SZK}}.\mathsf{Decide}(\sigma_b, 1-b, \mathsf{nonce}, \mathsf{st}[\mathcal{Z}])$

Random Oracle:

Input: Index j and payload d

1. if $j = 2$ then return $\mathsf{Sim}_{\mathrm{SZK}}.\mathsf{RO}(d)$
2. else return $\mathrm{H}_1(d)$

Fig. 12: Hybrid HYB_j in the proof of Theorem 1 . Changes from $\mathcal{G}_{\mathsf{Sim}}^{\mathsf{priv\text{-}ideal}}$ have been highlighted.

HYB_{q+1}

At the beginning of the game, the adversary outputs a key vk and the index b of the corrupt aggregator, and the game sets $i \coloneqq 0$, initializes list T_{client}, and runs $\mathsf{Sim.Init}(vk, b)$. We also sample $\mathrm{H}_2 \xleftarrow{\$} \mathcal{AF}(\mathcal{Z}.\mathsf{Dom}, \mathcal{Z}.\mathsf{Rng},)$.

Honest Client Computation:

Input: measurement $(\alpha, \beta) \in \{0,1\}^n \times \mathcal{L}$

1. $i \coloneqq i + 1$
2. $(\mathsf{pub}, \mathsf{key}_0, \mathsf{key}_1) \coloneqq \mathcal{V}.\mathsf{Gen}(\alpha, \beta)$
3. $[\![y_i^\epsilon]\!]_b \coloneqq \mathsf{EvalRoot}(\mathsf{pub}, \mathsf{key}_b)$
4. $[\![y_i^\epsilon]\!]_{1-b} \coloneqq \beta - [\![y_i^\epsilon]\!]_b$
5. $(\mathsf{nonce}, \pi_0^{\mathsf{szk}}, \pi_1^{\mathsf{szk}}) \coloneqq \mathcal{Z}.\mathsf{Prove}^{\mathrm{H}_2}([\![y_i^\epsilon]\!]_0, [\![y_i^\epsilon]\!]_1)$
6. $\mathsf{st}[\mathcal{V}_i^\epsilon] \coloneqq \mathsf{st}[\pi_i^\epsilon] \coloneqq \epsilon$
7. $\mathsf{st}[i] \coloneqq (\text{“ honest”}, \epsilon, (\mathsf{nonce}, \mathsf{pub}, \mathsf{key}_b, \pi_b^{\mathsf{szk}}))$
8. $T_{\text{client}}[i] \coloneqq (\alpha, \beta)$
9. Return $(\mathsf{nonce}, \mathsf{pub}, \mathsf{key}_b, \pi_b^{\mathsf{szk}})$

Corrupt Client Computation:

Input: report share $(\mathsf{nonce}, \mathsf{pub}, \mathsf{key}_{1-b}, \pi_{1-b}^{\mathsf{szk}})$

1. $i \coloneqq i + 1$
2. $\mathsf{st}[\mathcal{V}_i^\epsilon] \coloneqq \mathsf{st}[\pi_i^\epsilon] \coloneqq \epsilon$
3. $\mathsf{st}[i] \leftarrow (\text{“mal”}, \epsilon, (\mathsf{nonce}, \mathsf{pub}, \mathsf{key}_{1-b}, \pi_{1-b}^{\mathsf{szk}}))$

Aggregation

Input: set $\mathsf{Reports}$ of report indices, and a prefix p.

1. $\mathrm{a} \coloneqq 0$
2. for $i \in \mathsf{Reports}$
3. $\quad (\alpha, \beta) \coloneqq T_{\text{client}}[i]$
4. $\quad$ if p is a prefix of α then $\mathrm{a} \coloneqq \mathrm{a} + \beta$.
5. $[\![\mathsf{weight}]\!]_{1-b} \coloneqq \mathrm{a}$; $k \coloneqq |p|$
6. for $i \in \mathsf{Reports}$
7. $\quad \mathsf{hon}, \mathsf{status}, \mathsf{params} \coloneqq \mathsf{st}[i]$
8. $\quad (\mathsf{nonce}, \mathsf{pub}, \mathsf{key}, \pi^{\mathsf{szk}}) \coloneqq \mathsf{params}$
9. if $(p^{\lfloor k-1} = \varepsilon$ or $\mathsf{st}[\mathcal{V}_i^{p^{\lfloor k-1}}] \neq \varepsilon)$
10. $\quad ([\![y_i^p]\!], \_, \_) \coloneqq \mathcal{V}.\mathsf{Eval}^{\mathrm{H}_1}(\mathsf{key}, \mathsf{pub}, p, \mathsf{st}[\mathcal{V}_i^{p^{\lfloor k-1}}], \mathsf{st}[\pi_i^{p^{\lfloor k-1}}])$
11. $\quad$ if $\mathsf{hon} =$ “mal”
12. $\quad\quad [\![\mathsf{weight}]\!]_{1-b} \coloneqq [\![\mathsf{weight}]\!]_{1-b} + [\![y_i^p]\!]$
13. $\quad$ else
14. $\quad\quad [\![\mathsf{weight}]\!]_{1-b} \coloneqq [\![\mathsf{weight}]\!]_{1-b} - [\![y_i^p]\!]$
15. return $[\![\mathsf{weight}]\!]_{1-b}$

Honest Aggregator Evaluation:

Input: report index i and prefix p.

1. $\mathsf{hon}, \mathsf{status}, \mathsf{params} \coloneqq \mathsf{st}[i]$
2. if $(\mathsf{st}[\mathcal{V}_i^p] = \varepsilon$ or $\mathsf{st}[[\![y_i^p]\!]] = \epsilon)$ and $p \neq \varepsilon$ return $\perp$
3. $(\mathsf{nonce}, \mathsf{pub}, \mathsf{key}, \pi^{\mathsf{szk}}) \coloneqq \mathsf{params}$
4. if $\mathsf{status} = \epsilon$
5. $\quad \mathsf{st}[[\![y_i^\epsilon]\!]] \coloneqq \mathcal{V}.\mathsf{EvalRoot}^{\mathrm{H}_1}(\mathsf{key}, \mathsf{pub})$
6. $\quad$ if $\mathsf{hon} =$ “mal”
7. $\quad\quad (\mathsf{st}[\mathcal{Z}, i], \sigma_{1-b}) \coloneqq \mathcal{Z}.\mathsf{Query}^{\mathrm{H}_2}(vk, \mathsf{nonce}, [\![y_i^\epsilon]\!], \pi^{\mathsf{szk}})$
8. $\quad$ else
9. $\quad\quad (\mathsf{st}[\mathcal{Z}, i], \sigma_{1-b}) \coloneqq \mathcal{Z}.\mathsf{Query}(vk, \mathsf{nonce}, [\![y_i^\epsilon]\!], \pi^{\mathsf{szk}})$
10. $\quad \mathsf{st}[i] \leftarrow (\mathsf{hon}, \sigma_{1-b}, \mathsf{params})$
11. $\quad$ return σ_{1-b}
12. for $\gamma \in \{p \,\|\, 0, p \,\|\, 1\}$
13. $\quad (\mathsf{st}[[\![y_i^\gamma]\!]], \mathsf{st}[\mathcal{V}_i^\gamma], \mathsf{st}[\pi_i^\gamma]) \coloneqq \mathcal{V}.\mathsf{Eval}^{\mathrm{H}_1}(\mathsf{pub}, \mathsf{key}, \gamma, \mathsf{st}[\mathcal{V}_i^p], \mathsf{st}[\pi_i^p])$
14. if $\mathsf{hon} =$ “mal”
15. $\quad h \coloneqq (-1)^{1-b} \cdot (\mathsf{st}[[\![y_i^p]\!]] - \mathsf{st}[[\![y_i^{p\|0}]\!]] - \mathsf{st}[[\![y_i^{p\|1}]\!]])$
16. else
17. $\quad h \coloneqq (-1)^{b} \cdot (\mathsf{st}[[\![y_i^p]\!]] - \mathsf{st}[[\![y_i^{p\|0}]\!]] - \mathsf{st}[[\![y_i^{p\|1}]\!]])$
18. return $\mathsf{st}[\pi_i^p], \mathsf{st}[\pi_i^{p\|0}], \mathsf{st}[\pi_i^{p\|1}], h$

Honest Aggregator Validation:

Input: Report index i and partial sharedZK verifier string σ

1. $\mathsf{hon}, \mathsf{status}, \mathsf{params} \coloneqq \mathsf{st}[i]$
2. if $\mathsf{status} \in \{\epsilon, \perp\}$ return $\perp$
3. $(\mathsf{nonce}, \mathsf{pub}, \mathsf{key}, \pi^{\mathsf{szk}}) \coloneqq \mathsf{params}$
4. $\sigma_{1-b} \leftarrow \mathsf{status}$
5. $\mathsf{st}[i] \leftarrow (\mathsf{hon}, \perp, \mathsf{params})$
6. if $\mathsf{hon} =$ “mal”
7. $\quad$ return $\mathcal{Z}.\mathsf{Decide}^{\mathrm{H}_2}(\sigma_0, \sigma_1, \mathsf{st}[\mathcal{Z}, i])$
8. return $\mathcal{Z}.\mathsf{Decide}^{\mathrm{H}_2}(\sigma_0, \sigma_1, \mathsf{st}[\mathcal{Z}, i])$

Random Oracle:

Input: Index j and payload d

1. if $j = 2$ then return $\mathrm{H}_2(d)$
2. else return $\mathrm{H}_1(d)$

Fig. 13: Hybrid HYB_{q+1} in the proof of Theorem 1 . Changes from HYB_q have been highlighted.

HYB_{q+2}

At the beginning of the game, the adversary outputs a key vk and the index b of the corrupt aggregator, and the game sets $i \coloneqq 0$, initializes list T_{client}, and runs $\mathsf{Sim.Init}(vk, b)$. We also sample $\mathrm{H}_2 \xleftarrow{\$} \mathcal{AF}(\mathcal{Z}.\mathsf{Dom}, \mathcal{Z}.\mathsf{Rng},)$.

Honest Client Computation:

Input: measurement $(\alpha, \beta) \in \{0,1\}^n \times \mathcal{L}$

1. $i \coloneqq i + 1$
2. $(\mathsf{pub}, \mathsf{key}_0, \mathsf{key}_1) \coloneqq \mathcal{V}.\mathsf{Gen}(\alpha, \beta)$
3. $[\![y_i^\epsilon]\!]_b \coloneqq \mathsf{EvalRoot}(\mathsf{pub}, \mathsf{key}_b)$
4. $\boxed{[\![y_i^\epsilon]\!]_{1-b} \coloneqq \mathsf{EvalRoot}(\mathsf{pub}, \mathsf{key}_{1-b})}$
5. $(\mathsf{nonce}, \pi_0^{\mathsf{szk}}, \pi_1^{\mathsf{szk}}) \coloneqq \mathcal{Z}.\mathsf{Prove}^{\mathrm{H}_2}([\![y_i^\epsilon]\!]_0, [\![y_i^\epsilon]\!]_1)$
6. $\mathsf{st}[\mathcal{V}_i^\epsilon] \coloneqq \mathsf{st}[\pi_i^\epsilon] \coloneqq \epsilon$
7. $\boxed{\mathsf{st}[i] \coloneqq (\text{“ honest”}, \epsilon, (\mathsf{nonce}, \mathsf{pub}, \mathsf{key}_{1-b}, \pi_{1-b}^{\mathsf{szk}}))}$
8. $T_{\text{client}}[i] \coloneqq (\alpha, \beta)$
9. Return $(\mathsf{nonce}, \mathsf{pub}, \mathsf{key}_b, \pi_b^{\mathsf{szk}})$

Corrupt Client Computation:

Input: report share $(\mathsf{nonce}, \mathsf{pub}, \mathsf{key}_{1-b}, \pi_{1-b}^{\mathsf{szk}})$

1. $i \coloneqq i + 1$
2. $\mathsf{st}[\mathcal{V}_i^\epsilon] \coloneqq \mathsf{st}[\pi_i^\epsilon] \coloneqq \epsilon$
3. $\mathsf{st}[i] \leftarrow (\text{“mal”}, \epsilon, (\mathsf{nonce}, \mathsf{pub}, \mathsf{key}_{1-b}, \pi_{1-b}^{\mathsf{szk}}))$

Aggregation

Input: set $\mathsf{Reports}$ of report indices, and a prefix p.

1. $\boxed{[\![\mathsf{weight}]\!]_{1-b} \coloneqq 0}$; $k \coloneqq |p|$
2. for $i \in \mathsf{Reports}$
3. $\quad \mathsf{hon}, \mathsf{status}, \mathsf{params} \coloneqq \mathsf{st}[i]$
4. $\quad (\mathsf{nonce}, \mathsf{pub}, \mathsf{key}, \pi^{\mathsf{szk}}) \coloneqq \mathsf{params}$
5. if $(p^{|^{k-1}} = \varepsilon$ or $\mathsf{st}[\mathcal{V}_i^{p^{|^{k-1}}}] \neq \varepsilon)$
6. $\quad ([\![y_i^p]\!], \_, \_) \coloneqq \mathcal{V}.\mathsf{Eval}^{\mathrm{H}_1}(\mathsf{key}, \mathsf{pub}, p, \mathsf{st}[\mathcal{V}_i^{p^{|^{k-1}}}], \mathsf{st}[\pi_i^{p^{|^{k-1}}}])$
7. $\quad$ if $\mathsf{hon} =$ “mal”
8. $\quad\quad [\![\mathsf{weight}]\!]_{1-b} \coloneqq [\![\mathsf{weight}]\!]_{1-b} + [\![y_i^p]\!]$
9. $\quad$ else
10. $\quad\quad \boxed{[\![\mathsf{weight}]\!]_{1-b} \coloneqq [\![\mathsf{weight}]\!]_{1-b} + [\![y_i^p]\!]}$
11. return $[\![\mathsf{weight}]\!]_{1-b}$

Honest Aggregator Evaluation:

Input: report index i and prefix p.

1. $\mathsf{hon}, \mathsf{status}, \mathsf{params} \coloneqq \mathsf{st}[i]$
2. if $(\mathsf{st}[\mathcal{V}_i^p] = \varepsilon$ or $\mathsf{st}[[\![y_i^p]\!]] = \epsilon)$ and $p \neq \varepsilon$ return $\perp$
3. $(\mathsf{nonce}, \mathsf{pub}, \mathsf{key}, \pi^{\mathsf{szk}}) \coloneqq \mathsf{params}$
4. if $\mathsf{status} = \epsilon$
5. $\quad \mathsf{st}[\![y_i^\epsilon]\!] \coloneqq \mathcal{V}.\mathsf{EvalRoot}^{\mathrm{H}_1}(\mathsf{key}, \mathsf{pub})$
6. $\quad$ if $\mathsf{hon} =$ “mal”
7. $\quad\quad (\mathsf{st}[\mathcal{Z}, i], \sigma_{1-b}) \coloneqq \mathcal{Z}.\mathsf{Query}^{\mathrm{H}_2}(vk, \mathsf{nonce}, [\![y_i^\epsilon]\!], \pi^{\mathsf{szk}})$
8. $\quad$ else
9. $\quad\quad (\mathsf{st}[\mathcal{Z}, i], \sigma_{1-b}) \coloneqq \mathcal{Z}.\mathsf{Query}(vk, \mathsf{nonce}, [\![y_i^\epsilon]\!], \pi^{\mathsf{szk}})$
10. $\quad \mathsf{st}[i] \leftarrow (\mathsf{hon}, \sigma_{1-b}, \mathsf{params})$
11. $\quad$ return σ_{1-b}
12. for $\gamma \in \{p \,\|\, 0, p \,\|\, 1\}$
13. $\quad (\mathsf{st}[[\![y_i^\gamma]\!]], \mathsf{st}[\mathcal{V}_i^\gamma], \mathsf{st}[\pi_i^\gamma]) \coloneqq \mathcal{V}.\mathsf{Eval}^{\mathrm{H}_1}(\mathsf{pub}, \mathsf{key}, \gamma, \mathsf{st}[\mathcal{V}_i^p], \mathsf{st}[\pi_i^p])$
14. if $\mathsf{hon} =$ “mal”
15. $\quad h \coloneqq (-1)^{1-b} \cdot (\mathsf{st}[[\![y_i^p]\!]] - \mathsf{st}[[\![y_i^{p\|0}]\!]] - \mathsf{st}[[\![y_i^{p\|1}]\!]])$
16. else
17. $\quad \boxed{h \coloneqq (-1)^{b+1} \cdot (\mathsf{st}[[\![y_i^p]\!]] - \mathsf{st}[[\![y_i^{p\|0}]\!]] - \mathsf{st}[[\![y_i^{p\|1}]\!]])}$
18. return $\mathsf{st}[\pi_i^p], \mathsf{st}[\pi_i^{p\|0}], \mathsf{st}[\pi_i^{p\|1}], h$

Honest Aggregator Validation:

Input: Report index i and partial sharedZK verifier string σ

1. $\mathsf{hon}, \mathsf{status}, \mathsf{params} \coloneqq \mathsf{st}[i]$
2. if $\mathsf{status} \in \{\epsilon, \perp\}$ return $\perp$
3. $(\mathsf{nonce}, \mathsf{pub}, \mathsf{key}, \pi^{\mathsf{szk}}) \coloneqq \mathsf{params}$
4. $\sigma_{1-b} \leftarrow \mathsf{status}$
5. $\mathsf{st}[i] \leftarrow (\mathsf{hon}, \perp, \mathsf{params})$
6. if $\mathsf{hon} =$ “mal”
7. $\quad$ return $\boxed{\mathcal{Z}.\mathsf{Decide}^{\mathrm{H}_2}(\sigma_0, \sigma_1, \mathsf{st}[\mathcal{Z}, i])}$
8. return $\boxed{\mathcal{Z}.\mathsf{Decide}^{\mathrm{H}_2}(\sigma_0, \sigma_1, \mathsf{st}[\mathcal{Z}, i])}$

Random Oracle:

Input: Index j and payload d

1. if $j = 2$ then return $\boxed{\mathrm{H}_2(d)}$
2. else return $\mathrm{H}_1(d)$

Fig. 14: Hybrid HYB_{q+2} in the proof of Theorem 1 . Changes from HYB_q have been $\boxed{\text{highlighted}}$.

$\mathcal{F}_{\mathsf{wHH}}$

Parameters: Aggregators $\mathcal{S}_0$ and $\mathcal{S}_1$. N clients $\mathcal{C}_i$ for $i \in [\mathsf{N}]$. $\mathcal{S}_0, \mathcal{S}_1$ agree on:

- A bound N on the number of client submissions.
- A bound T on the threshold for heavy hitters.
- Function order that defines a total ordering over sums of weights.

Inputs:

Aggregators: $\mathcal{S}_0, \mathcal{S}_1$ do not have any input.

Clients: Each client $\mathcal{C}_i$ for $i \in [\mathsf{N}]$ holds "partial measurement" $(\alpha_i, \beta_i) \in (\{0,1\}^{\leq n} \cup \{\perp\}) \times (\mathbb{F}^m \cup \{\perp\})$ composed of an input α_i and its weight β_i. Each honest client holds a valid measurement, i.e. $(\alpha_i, \beta_i) \in \{0,1\}^n \times \mathcal{L}$. Let $\alpha_{i,k}$ represent the kth bit of α_i.

Algorithm:

1. *Init:* $\mathsf{HH}^{\leq n} := \{\mathsf{HH}^0, \mathsf{HH}^1, \ldots \mathsf{HH}^n\} := \{\{\epsilon\}, \emptyset, \ldots, \emptyset\}$. Set $\mathsf{Reports} := [\mathsf{N}]$.
2. *Check:* For each $i \in [\mathsf{N}]$: If $(\alpha_i, \beta_i) \notin \{0,1\}^{\leq n} \times \mathcal{L}$ then discard it from computation by updating $\mathsf{Reports} := \mathsf{Reports} \setminus \{i\}$.
3. *Aggregation:* For $k \in [0, \ldots, n-1]$ and for each prefix $p \in \mathsf{HH}^k$, consider $\gamma \in \{p \,\|\, 0, p \,\|\, 1\}$ and update $\mathsf{HH}^{k+1} := \mathsf{HH}^{k+1} \cup \gamma$ if $\mathsf{order}(\mathsf{weight}^\gamma) > \mathsf{T}$, where weight^γ is the sum of weights β_i of each input α_i with prefix γ. More formally:
$$\mathsf{weight}^\gamma := \sum_i \beta_i \text{ for } i \in \mathsf{Reports} \wedge (\alpha_i|^{k+1} = \gamma)\,.$$

$\mathcal{F}_{\mathsf{wHH}}$ outputs the following:

- *Aggregators* $\mathcal{S}_0, \mathcal{S}_1$: Set of T-heavy hitters $\mathsf{HH}^{\leq n}$. For each heavy-hitting string p the aggregators also obtain weight^p, $\mathsf{weight}^{p||0}$ and $\mathsf{weight}^{p||1}$.
- *Clients* $\mathcal{C}_i$ for $i \in [\mathsf{N}]$: No output.

Corruption: Adversary $\mathcal{A}_{\mathsf{HH}}$ corrupts multiple clients together. $\mathcal{A}_{\mathsf{HH}}$ sets the input of each corrupt client as a partial measurement (α_i, β_i).

Note: $\mathcal{F}_{\mathsf{wHH}}$ models the problem of "*weighted*" heavy-hitters. It can be weakened to capture "*plain*" heavy-hitters by letting *Check* enforce that $\beta_i = 1$ for every measurement.

Fig. 15: Ideal Functionality for Weighted Heavy-Hitters

$(\alpha_i, \beta_i) \notin \{0,1\}^n \times \mathcal{L}$. We argue that if the malformed report share passes the consistency checks and gets incorporated into the aggregation process by the honest aggregators then the malicious client breaks the verifiability of $\mathcal{V}$ in the first case, and soundness of $\mathcal{Z}$ in the second case.

To argue robustness formally, for each pair of report shares submitted by a malicious client, the protocol needs to either "*extract*" a valid measurement from or detect that it is invalid. Once this distinction is performed the protocol needs to "*compute*" the aggregation function over the honest client *inputs* $(\alpha_i, \beta_i) \in \{0,1\}^n \times \mathcal{L}$ (represented via their report shares) and the valid measurements submitted by the malicious clients.

We capture this property in a simulation-based model [Can00], presented in Fig. 17 (Appendix D). The adversary $\mathcal{A}$ initially corrupts a set $\mathsf{Reports}'$ of clients. In the real-world game $\mathcal{G}^{\mathsf{rob\text{-}real}}_{\mathsf{Mastic}}$ (Fig. 16), the parties run the Mastic protocol using their input measurements. Both the honest and corrupt clients provide their report shares to the aggregators, who compute the output (set of heavy-hitter strings and their children, and also the weights of the heavy-hitting strings and their children) and return it to $\mathcal{G}^{\mathsf{rob\text{-}real}}_{\mathsf{Mastic}}$. The game forwards this to $\mathcal{A}$.

We also define a corresponding ideal-world game, $\mathcal{G}^{\mathsf{rob\text{-}ideal}}_{\mathsf{Sim},\mathcal{F}_{\mathsf{wHH}}}$ (Fig. 17). In this game, the corrupt clients (controlled by $\mathcal{A}$) participate in the same way as in the real world, but with a pair of simulated aggregators. This simulation is performed by a PPT algorithm called Sim (Fig. 18). This simulator obtains the report

$\mathcal{G}^{\mathsf{rob\text{-}real}}_{\mathsf{Mastic}}$

Input: Each client $\mathcal{C}_i$ has input (α_i, β_i) for $i \in [\mathsf{N}]$.

1. Adversary $\mathcal{A}$ initially corrupts a set $\mathsf{Reports}'$ of clients and sends $\left(\mathsf{nonce}_i, \mathsf{pub}_i, \mathsf{key}_{(i,b)}, \pi^{\mathsf{szk}}_{(i,b)}\right)$ to aggregator $\mathcal{S}_b$ for each $b \in \{0,1\}$ and $i \in \mathsf{Reports}'$.
2. Each aggregator $\mathcal{S}_b$ obtains $\left(\mathsf{nonce}_i, \mathsf{pub}_i, \mathsf{key}_{(i,b)}, \pi^{\mathsf{szk}}_{(i,b)}\right)$ from ith honest client for $i \in \mathsf{Reports} \setminus \mathsf{Reports}'$, where $\mathsf{Reports} := [\mathsf{N}]$. Client i computes this by running the "Client Computation" protocol on $(\alpha_i, \beta_i) \in (\{0,1\}^n, \mathcal{L})$.
3. Each aggregator $\mathcal{S}_b$ runs the "Aggregation Computation" protocol (Fig. 2) on $\left(\mathsf{nonce}_i, \mathsf{pub}_i, \mathsf{key}_{(i,b)}, \pi^{\mathsf{szk}}_{(i,b)}\right)$ for $b \in \{0,1\}$ to obtain $\mathsf{HH}^{\leq n}$, and for each heavy-hitting string $p \in \mathsf{HH}^{\leq n}$ obtain weight^p, $\mathsf{weight}^{p||0}$ and $\mathsf{weight}^{p||1}$.
4. Send the following to $\mathcal{A}$: Output $\mathsf{HH}^{\leq n}$, and weights weight^p, $\mathsf{weight}^{p||0}$ and $\mathsf{weight}^{p||1}$ for each $p \in \mathsf{HH}^{\leq n}$.

Fig. 16: Real Game for defining robustness of Π_{Mastic}.

shares of the corrupt clients, extracts the input measurements (α'_i, β'_i) from those shares, and invokes $\mathcal{F}_{\mathsf{wHH}}$ (on behalf of the corrupt clients) with the extracted measurements.

Upon obtaining the client measurements from both honest and corrupt clients, $\mathcal{F}_{\mathsf{wHH}}$ checks the measurements and then computes the output. For consistency check, $\mathcal{F}_{\mathsf{wHH}}$ discards the measurements if they are not correctly formatted, i.e. $(\alpha_i, \beta_i) \neq (\perp, \perp)$, or if the weight is not valid, i.e. $(\alpha_i, \beta_i) \notin \mathcal{L}$. The functionality also allows corrupt clients to submit prefix strings, i.e. $\alpha_i \in \{0,1\}^{\leq n}$. Once the measurements are validated, the functionality aggregates the validated measurements by computing the weights and heavy-hitting set similar to the aggregation phase in Mastic. This is the Aggregation step in Fig. 15. $\mathcal{F}_{\mathsf{wHH}}$ computes the output and this is returned to $\mathcal{G}^{\mathsf{rob\text{-}ideal}}_{\mathsf{Sim},\mathcal{F}_{\mathsf{wHH}}}$. The game forwards this to $\mathcal{A}$.

Robustness guarantees that an adversary (who has corrupted an arbitrary number of clients) cannot distinguish between the real and ideal world outputs. We define the advantage of an adversary $\mathcal{A}$ against the robustness of Mastic with respect to simulator Sim by $\mathbf{Adv}^{\mathsf{rob}}_{\mathsf{Mastic},\mathsf{Sim}}(\mathcal{A})$ as:

$$\left|\Pr[\mathcal{G}^{\mathsf{rob\text{-}real}}_{\mathsf{Mastic}}(\mathcal{A}) \Rightarrow 1] - \Pr[\mathcal{G}^{\mathsf{rob\text{-}ideal}}_{\mathsf{Sim},\mathcal{F}_{\mathsf{wHH}}}(\mathcal{A}) \Rightarrow 1]\right|.$$

Mastic (also PLASMA and Poplar) permits a malicious client to submit report shares (in Mastic protocol) which are only valid until level $k \leq n$, after which robustness requires the report shares to be discarded.

Assume the adversary $\mathcal{A}$ corrupts $\mathsf{N}' = |\mathsf{Reports}'|$ clients. We formally introduce the games $\mathcal{G}^{\mathsf{rob\text{-}real}}_{\mathsf{Mastic}}$ and $\mathcal{G}^{\mathsf{rob\text{-}ideal}}_{\mathsf{Sim},\mathcal{F}_{\mathsf{wHH}}}$ for robustness in Figs. 16 and 17. We prove the robustness of Mastic in this section by proving that $\mathcal{G}^{\mathsf{rob\text{-}real}}_{\mathsf{Mastic}}$ is indistinguishable from $\mathcal{G}^{\mathsf{rob\text{-}ideal}}_{\mathsf{Sim},\mathcal{F}_{\mathsf{wHH}}}$ by providing the formal simulator algorithm Sim in Fig. 18.

Proof Sketch. To argue robustness, the aggregators need to ensure that the report shares provided by a corrupt client encode a valid measurement (α_i, β_i). As described in Section. 3.2.2, this involves checking three things.

The weight β_i encoded inside the report shares is valid, i.e. $\beta_i \in \mathcal{L}$. This is ensured by the aggregators by evaluating the $\mathcal{V}$ keys at the root layer to obtain β_i, and then running the $\mathcal{Z}$ to validate β_i (without reconstructing β_i). An adversarial client who provides an invalid β_i, i.e. $\beta_i \notin \mathcal{L}$, but passes the checks of $\mathcal{Z}$ breaks soundness of $\mathcal{Z}$.

Next, the aggregators must ensure that the same β_i value is propagated across a single path in the evaluation tree (encoded inside the $\mathcal{V}$ keys). This reduces to verifying that each level $k(\in [n])$ in the evaluation tree contains only a single non-zero node. This is ensured by verifying the $\mathcal{V}$ proofs for every node considered (as part of an evaluation path) during the computation of the heavy-hitter set. An adversarial client whose – report shares encode more than one non-zero node (as part of or two evaluation paths), and those nodes are also encountered during the heavy-hitter evaluation, and still the client passes the checks, can be used to

$\mathcal{G}^{\mathsf{rob\text{-}ideal}}_{\mathsf{Sim},\mathcal{F}_{\mathsf{wHH}}}$

Input: Each client $\mathcal{C}_i$ has input (α_i, β_i) for $i \in [\mathsf{N}]$.

1. Adversary $\mathcal{A}$ initially corrupts a set $\mathsf{Reports}'$ of clients and sends $\left(\mathsf{nonce}_i, \mathsf{pub}_i, \mathsf{key}_{(i,b)}, \pi^{\mathsf{szk}}_{(i,b)}\right)$ to the simulated aggregators $\mathcal{S}_b$ (controlled by Sim) for each $b \in \{0,1\}$ and $i \in \mathsf{Reports}'$.
2. Sim extracts the ith malicious client inputs as follows for $i \in \mathsf{Reports}'$:

$$(i, \alpha'_i, \beta'_i) \leftarrow \mathsf{Sim}\Big(\left(\mathsf{nonce}_i, \mathsf{pub}_i, \mathsf{key}_{(i,b)}, \pi^{\mathsf{szk}}_{(i,b)}\right)_{b\in\{0,1\}}\Big),$$

   (α'_i, β'_i) is a *partial measurement* that is not necessarily valid. (E.g., $(\perp, \perp)$.) Let $R' \coloneqq \{(i, \alpha'_i, \beta'_i) : i \in \mathsf{Reports}'\}$.
3. Invoke $\mathcal{F}_{\mathsf{wHH}}$ on each $(i, \alpha'_i, \beta'_i) \in R'$ (on behalf of each corrupt client $\mathcal{C}_i \in \mathsf{Reports}'$ chosen by $\mathcal{A}_{\mathsf{HH}}$). Each honest client provides its input $(\alpha_i, \beta_i) \in (\{0,1\}^n, \mathcal{L})$ to $\mathcal{F}_{\mathsf{wHH}}$. $\mathcal{F}_{\mathsf{wHH}}$ returns the following outputs:
   - T-heavy hitters $\mathsf{HH}^{\leq n}$, and
   - For each heavy-hitting string $p \in \mathsf{HH}^{\leq n}$ $\mathcal{F}_{\mathsf{wHH}}$ also sends weight^p, $\mathsf{weight}^{p||0}$ and $\mathsf{weight}^{p||1}$.
4. Send the following to $\mathcal{A}$: Output $\mathsf{HH}^{\leq n}$, and weights weight^p, $\mathsf{weight}^{p||0}$ and $\mathsf{weight}^{p||1}$ for each $p \in \mathsf{HH}^{\leq n}$.

Fig. 17: Ideal Game for defining robustness of Π_{Mastic}. Let $\mathcal{F}_{\mathsf{wHH}}$ be as defined in 15.

break the verifiability of $\mathcal{V}$. Finally, the aggregator nodes must ensure that the non-zero nodes at each level are along the same path, say p. This is performed by checking that the output for prefix p is equal to the sum of the output of its children - $(p \,\|\, 0, p \,\|\, 1)$. This check is information-theoretic. By combining the one-hot verifiability and path-verifiability guarantees, we provide stronger guarantees where the adversarial input $\alpha_i \in \{0,1\}^{\leq n}$ can be uniquely extracted from the report shares.

Finally, we optimized the communication by allowing the aggregator nodes to hash the results of the consistency checks for each client and then match the hash values. So, here we also need to rely on the collision-resistance of the hash function to ensure that if the hashes match then the underlying preimages are also equal.

Formal Proof. We prove Theorem 2 by showing that $\mathcal{G}^{\mathsf{rob\text{-}real}}_{\mathsf{Mastic}}$ and $\mathcal{G}^{\mathsf{rob\text{-}ideal}}_{\mathsf{Sim},\mathcal{F}_{\mathsf{wHH}}}$ are computationally indistinguishable in the presence of our Sim against all PPT adversaries. We argue this via a sequence of hybrids.

- HYB_0 : This is game $\mathcal{G}^{\mathsf{rob\text{-}real}}_{\mathsf{Mastic}}$ the clients compute their report shares based on their inputs, provide these shares to the honest aggregators, the honest aggregators compute the Mastic protocol and then provide the output to $\mathcal{G}^{\mathsf{rob\text{-}real}}_{\mathsf{Mastic}}$. This output is returned to the adversary.
- HYB_1 : This is same as HYB_0, except the simulated aggregators reconstruct β_i and removes i from $\mathsf{Reports}$ if $\beta_i \notin \mathcal{L}$. We present it in Fig. 19.

  A distinguisher distinguishes between the two hybrids if a malformed report containing $\beta_i \notin \mathcal{L}$ passes the consistency check for $\mathcal{Z}$ in HYB_0 and gets included in $\mathsf{Reports}$, whereas in HYB_1 it gets removed from $\mathsf{Reports}$. This alters the output distribution. Assuming a distinguisher $\mathcal{A}_{0,1}$ that distinguishes between the two hybrids then we build an adversary $\mathcal{B}$ that breaks the soundness of $\mathcal{Z}$ as follows.

  When $\mathcal{A}_{0,1}$ (on behalf of a malicious client i) returns a malformed report share $\left(\mathsf{nonce}_i, \mathsf{pub}_i, \mathsf{key}_{(i,b)}, \pi^{\mathsf{szk}}_{(i,b)}\right)$ to $\mathcal{S}_b$ for each $b \in \{0,1\}$, $\mathcal{B}$ extracts $[\![\beta_i]\!]_b \coloneqq \mathcal{V}.\mathsf{EvalRoot}^{\mathrm{H}_1}(\mathsf{key}_{(i,b)}, \mathsf{pub}_i)$ for $b \in \{0,1\}$. $\mathcal{B}$ computes $\beta_i \coloneqq [\![\beta_i]\!]_0 + [\![\beta_i]\!]_1$ and $(\mathsf{st}_b, \sigma_b) \coloneqq \mathcal{Z}.\mathsf{Query}^{\mathrm{H}_2}(vk, \mathsf{nonce}_i, [\![\beta_i]\!]_b, \pi^{\mathsf{szk}}_{(i,b)})$. If $\beta_i \notin \mathcal{L}$ and $\mathsf{Accept} \coloneqq \mathcal{Z}.\mathsf{Decide}^{\mathrm{H}_2}(\sigma_0, \sigma_1, \mathsf{st}_b)$, then $\mathcal{B}$ returns $(\mathsf{nonce}_i, [\![\beta_i]\!]_0, [\![\beta_i]\!]_1, \pi^{\mathsf{szk}}_{(i,0)}, \pi^{\mathsf{szk}}_{(i,1)})$ to the challenger of $\mathcal{Z}$. It can be observed that $\mathcal{B}$ wins the game only when $\mathcal{A}_{0,1}$ constructs a malformed report such that the $\mathcal{Z}$ verifies but $\beta_i \notin \mathcal{L}$, and this directly translates into a win for $\mathcal{B}$. Assume the $\mathcal{A}_{0,1}$ has an advantage $\mathbf{Adv}_{0,1}$ of distinguishing between the two hybrids. The same attack should be considered for all N' clients. By

Sim for $\mathcal{G}^{\mathsf{rob\text{-}ideal}}_{\mathsf{Sim},\mathcal{F}_{\mathsf{wHH}}}$

Sim simulates the role of the honest aggregators in this protocol.

Malicious Client Computation: Each corrupt client $\mathcal{C}_i$ for $i \in \mathsf{Reports}'$ sends report share $(\mathsf{nonce}_i, \mathsf{pub}_i, \mathsf{key}_{(i,b)}, \pi^{\mathsf{szk}}_{(i,b)})$ to the simulated aggregators $\mathcal{S}_b$ for each $b \in \{0,1\}$.

Simulated Aggregator Computation:

Input: The aggregators $\mathcal{S}_0$ and $\mathcal{S}_1$ are run by Sim. They start with a verification key $vk \in \{0,1\}^{\mathsf{vkl}}$ established out-of-band. They receive the set of corrupt clients as $\mathsf{Reports}'$. Each aggregator $\mathcal{S}_b$ obtains $(\mathsf{nonce}_i, \mathsf{pub}_i, \mathsf{key}_{(i,b)}, \pi^{\mathsf{szk}}_{(i,b)})$ for $i \in \mathsf{Reports}'$.

1. **For each** client $i \in \mathsf{Reports}$:
   a. If $(\mathsf{nonce}_i, \mathsf{pub}_i, \mathsf{key}_{(i,b)}, \pi^{\mathsf{szk}}_{(i,b)})$ does not follow the correct input-formatting of Mastic then Sim sets $(\alpha'_i, \beta'_i) \coloneqq (\perp, \perp)$ as the ith client's measurement and skips rest of this loop for this particular value of i. ▷ Input-Formatting check.
   b. Otherwise, Sim computes $\beta'_i \coloneqq [\![\beta_i]\!]_0 + [\![\beta_i]\!]_1$, where $[\![\beta_i]\!]_b \coloneqq \mathcal{V}.\mathsf{EvalRoot}^{\mathrm{H}_1}(\mathsf{key}_{(i,b)}, \mathsf{pub}_i)$ for $b \in \{0,1\}$. ▷ Weight-computation. $\mathcal{F}_{\mathsf{wHH}}$ performs weight-check on this β'_i in Step 2.
   c. Sim extracts the α'_i as follows. Initialize $r \coloneqq \epsilon$ and $\alpha'_i \coloneqq \epsilon$. For $b \in \{0,1\}$, Sim sets $[\![y^r_i]\!]_b \coloneqq \epsilon, \mathsf{st}^r_{(i,b)} \coloneqq \epsilon$, $\pi^r_{(i,b)} \coloneqq \epsilon$ and store them in memory. For $k \in [0, 1, \ldots, n-1]$ run the following:
      i. Each $\mathcal{S}_b$ retrieves the state $([\![y^r_i]\!]_b, \mathsf{st}^r_{(i,b)}, \pi^r_{(i,b)})$ from memory corresponding to prefix r.
      ii. Each $\mathcal{S}_b$ runs $([\![y^\gamma_i]\!]_b, \mathsf{st}^\gamma_{(i,b)}, \pi^\gamma_{(i,b)}) \coloneqq \mathcal{V}.\mathsf{Eval}^{\mathrm{H}_1}(\mathsf{key}_{(i,b)}, \mathsf{pub}_i, \gamma, \mathsf{st}^\gamma_{(i,b)}, \pi^\gamma_{(i,b)})$ for each prefix $\gamma \in \{r \,\|\, 0, r \,\|\, 1\}$ and stores the results in memory.
      iii. For $\gamma \in \{r \,\|\, 0, r \,\|\, 1\}$, Sim computes $y^\gamma_i \coloneqq [\![y^\gamma_i]\!]_0 + [\![y^\gamma_i]\!]_1$.
      iv. If any of the three conditions hold then Sim considers (α'_i, β'_i) as the ith client's measurement and skips this inner and the outer loop for this particular value of i:
         – Both $\pi^{r\|0}_{(i,0)} \neq \pi^{r\|0}_{(i,1)} \wedge \pi^{r\|1}_{(i,0)} \neq \pi^{r\|1}_{(i,1)}$,
         – Both $y^{r\|0}_i \neq 0$ and $y^{r\|1}_i \neq 0$, or ▷ One-hot verifiability.
         – $y^{r\|0}_i + y^{r\|1}_i \neq \beta'_i$. ▷ Path verifiability.
      v. If $(y^{r\|0}_i = \beta'_i)$ then update $r \coloneqq r \,\|\, 0$ and $\alpha'_i \coloneqq \alpha'_i \,\|\, 0$. Otherwise, if $(y^{r\|1}_i = \beta'_i)$ then update $r \coloneqq r \,\|\, 1$ and $\alpha'_i \coloneqq \alpha'_i \,\|\, 1$
   d. After the above loop, Sim stores the ith client's measurement as (i, α'_i, β'_i).
2. Sim returns all the corrupt client measurements as (i, α'_i, β'_i) for $i \in \mathsf{Reports}'$.

Fig. 18: Simulator for ideal robustness game $\mathcal{G}^{\mathsf{rob\text{-}ideal}}_{\mathsf{Sim},\mathcal{F}_{\mathsf{wHH}}}$.

applying the union bound over all N' and by using triangle inequality we bound $\mathbf{Adv}_{0,1}$ as follows:

$$\mathbf{Adv}_{0,1} \leq \sum_{i \in [\mathsf{N}']} \mathbf{Adv}^i_{0,1} \leq \mathsf{N}' \cdot \mathbf{Adv}^{\mathsf{sound}}_{\mathcal{Z}}(\mathcal{B}).$$

Note: Next, we consider a pair of $2n$ hybrids, each pair for a level. We first replace the hash functions with plain communication and then we rely on the verifiability of $\mathcal{V}$ to argue that each client provides a single non-zero path in the evaluation tree.

Run the following for $j \in [1, \ldots, n-1]$:

– HYB_{2j} : This is same as HYB_{2j-1}, except $\mathcal{S}_b$ accumulates its local state as $R^j_{(i,b)} \coloneqq \big(\big|\big|_{p \in \mathsf{HH}^j}(p, h^p_{(i,b)}, \pi^{p\|0}_{(i,b)}, \pi^{p\|1}_{(i,b)})\big)$ instead of $R^j_{(i,b)} \coloneqq \mathrm{H}\big(\big|\big|_{p \in \mathsf{HH}^j}(p, h^p_{(i,b)}, \pi^{p\|0}_{(i,b)}, \pi^{p\|1}_{(i,b)})\big)$. The checks are performed over $R^j_{(i,b)}$. We present it in Figures. 20, 21.
An adversary $\mathcal{A}_{2j-1,2j}$ distinguishes between the two hybrids if it finds a collision in H s.t. the verification in HYB_{2j} fails due to $R^j_{(i,0)} \neq R^j_{(i,1)}$ whereas they match in HYB_{2j-1} since to a collision in the hash. Using this adversary, we construct an adversary $\mathcal{C}$ that finds a collision in the hash by returning $\big(\big|\big|_{p \in \mathsf{HH}^j}(p, h^p_{(i,0)}, \pi^{p\|0}_{(i,0)}, \pi^{p\|1}_{(i,0)})\big)$ and $\big(\big|\big|_{p \in \mathsf{HH}^j}(p, h^p_{(i,1)}, \pi^{p\|0}_{(i,b)}, \pi^{p\|1}_{(i,1)})\big)$. These two values are unequal but

their hash values match. Assume the advantage of $\mathcal{A}_{2j-1,2j}$ is $\mathbf{Adv}_{2j-1,2j}$. The same attack should be considered for all N' clients. By applying the union bound over all N' and by using triangle inequality we bound $\mathbf{Adv}_{2j-1,2j}$ as follows:

$$\mathbf{Adv}_{2j-1,2j} \leq \sum_{i\in[\mathsf{N}']} \mathbf{Adv}^{i}_{2j-1,2j} \leq \mathsf{N}' \cdot \mathbf{Adv}^{\mathsf{coll}}_{\mathrm{H}}(\mathcal{C}).$$

– HYB_{2j+1} : This is same as HYB_{2j}, except the servers extract $\alpha'_{i,j+1}$ by running the extraction algorithm till level j and then the heavy-hitter set $\mathsf{HH}^{\leq j+1}$ is computed based on the extracted values, whereas $\widetilde{\mathsf{HH}^{j+1}}$ is computed following the Mastic protocol; and if $\mathsf{HH}^{j+1} \neq \widetilde{\mathsf{HH}^{j+1}}$, then Sim return $(\perp,\perp)$ to $\mathcal{F}_{\mathsf{wHH}}$ on behalf of the corrupt clients. We present it in Figures. 22, 23.
An adversary $\mathcal{A}_{2j,2j+1}$ distinguishes between the two hybrids if it finds two non-zero evaluation paths in the $\mathcal{V}$ evaluation at level j s.t. they lead to two different non-zero values on prefixes $r \parallel 0$ and $r \parallel 1$. In such a case, HYB_{2j} fails to detect it and HYB_{2j+1} detects it. HYB_{2j} will consider $\widetilde{\mathsf{HH}^{j}}$ as the heavy-hitting set and HYB_{2j+1} will return $(\perp,\perp)$ to $\mathcal{F}_{\mathsf{wHH}}$. Using this adversary, we construct an adversary $\mathcal{D}$ that breaks the verifiability of the $\mathcal{V}$. When client i returns the report shares simulate as per HYB_{2j} and if it encounters two such non-zero evaluation paths - $(r \parallel 0, r \parallel 1)$, on which the proofs verify, then return the $\mathcal{V}$ keys in the report shares and $(r \parallel 0, r \parallel 1)$ to the $\mathcal{V}$ challenger. Assume the advantage of $\mathcal{A}_{2j,2j+1}$ is $\mathbf{Adv}_{2j,2j+1}$. The same attack should be considered for all N' clients. By applying the union bound over all N' and by using triangle inequality we bound $\mathbf{Adv}_{2j,2j+1}$ as follows:

$$\mathbf{Adv}_{2j,2j+1} \leq \sum_{i\in[\mathsf{N}']} \mathbf{Adv}^{i}_{2j,2j+1} \leq \mathsf{N}' \cdot \mathbf{Adv}^{\mathsf{verif}}_{\mathcal{V},k}(\mathcal{D}).$$

It can be observed that HYB_{2n} corresponds to $\mathcal{G}^{\mathsf{rob\text{-}ideal}}_{\mathsf{Sim},\mathcal{F}_{\mathsf{wHH}}}$. Thus, we bound the advantage of $\mathcal{A}$ distinguishing between $\mathcal{G}^{\mathsf{rob\text{-}real}}_{\mathsf{Mastic}}$ and $\mathcal{G}^{\mathsf{rob\text{-}ideal}}_{\mathsf{Sim},\mathcal{F}_{\mathsf{wHH}}}$ as follows:

$$\begin{aligned}
&\mathbf{Adv}^{\mathsf{rob}}_{\mathsf{Mastic},\mathsf{Sim},\mathcal{F}_{\mathsf{wHH}}}(\mathcal{A}) \\
=\ & \mathbf{Adv}_{0,1} + \textstyle\sum_{j\in[1,\ldots n-1]}(\mathbf{Adv}_{2j-1,2j} + \mathbf{Adv}_{2j,2j+1}) \\
\leq\ & \mathsf{N}' \cdot \mathbf{Adv}^{\mathsf{sound}}_{\mathcal{Z}}(\mathcal{B}) \\
+\ & \textstyle\sum_{j\in[1,\ldots n-1]} \left(\mathsf{N} \cdot (\mathbf{Adv}^{\mathsf{coll}}_{\mathrm{H}}(\mathcal{C}) + \mathbf{Adv}^{\mathsf{verif}}_{\mathcal{V},k}(\mathcal{D}))\right) \\
=\ & \mathsf{N}' \cdot \left(\mathbf{Adv}^{\mathsf{sound}}_{\mathcal{Z}}(\mathcal{B}) + n \cdot (\mathbf{Adv}^{\mathsf{coll}}_{\mathrm{H}}(\mathcal{C}) + \mathbf{Adv}^{\mathsf{verif}}_{\mathcal{V},k}(\mathcal{D}))\right).
\end{aligned}$$

This concludes the robustness proof.

Hybrid HYB_1

Sim simulates the role of the honest aggregators in this protocol.

Primitives:

1. A VIDPF $(\mathcal{V}.\mathsf{Gen}, \mathcal{V}.\mathsf{Eval}, \mathcal{V}.\mathsf{EvalRoot})$ as defined in Section 2.3.
2. A shared ZK $(\mathcal{Z}.\mathsf{Prove}, \mathcal{Z}.\mathsf{Query}, \mathcal{Z}.\mathsf{Decide})$ for a language $\mathcal{L} \subseteq \mathbb{F}^m$ as defined in Section 2.4.
3. Functions H, H_1, H_2 modeled in our analysis as random oracles.

Client Computation:

Input: Each client $\mathcal{C}_i$ for $i \in [\mathsf{N}]$ holds measurement $(\alpha_i, \beta_i) \in (\{0,1\}^n, \mathcal{L})$ composed of an input α_i and its weight β_i.

1. $\mathcal{C}_i$ runs $(\mathsf{pub}_i, \mathsf{key}_{(i,0)}, \mathsf{key}_{(i,1)}) \xleftarrow{\$} \mathcal{V}.\mathsf{Gen}^{\mathrm{H}_1}(\alpha_i, \beta_i)$.
2. $\mathcal{C}_i$ runs $(\pi^{\mathsf{szk}}_{(i,0)}, \pi^{\mathsf{szk}}_{(i,1)}, \mathsf{nonce}_i) \xleftarrow{\$} \mathcal{Z}.\mathsf{Prove}^{\mathrm{H}_2}([\![\beta_i]\!]_0, [\![\beta_i]\!]_1)$ where $[\![\beta_i]\!]_b := \mathcal{V}.\mathsf{EvalRoot}^{\mathrm{H}_1}(\mathsf{key}_{(i,b)}, \mathsf{pub}_i)$ for $b \in \{0,1\}$.
3. $\mathcal{C}_i$ sends report share $(\mathsf{nonce}_i, \mathsf{pub}_i, \mathsf{key}_{(i,b)}, \pi^{\mathsf{szk}}_{(i,b)})$ to $\mathcal{S}_b$ for each $b \in \{0,1\}$.

Aggregator Computation:

Input: The simulated aggregators $\mathcal{S}_0$ and $\mathcal{S}_1$ start with a verification key $vk \in \{0,1\}^{\mathsf{vkl}}$ established out-of-band. Each sets $\mathsf{HH}^{\leq n} := \{\mathsf{HH}^0, \mathsf{HH}^1, \ldots \mathsf{HH}^n\}$ as $\{\epsilon, \emptyset, \ldots, \emptyset\}$, the initial set of *candidate prefixes* for each level and sets $\mathsf{Reports} := [\mathsf{N}]$, the initial set of *candidate reports*.

1. **For each** client $i \in \mathsf{Reports}$: ▷ Weight check using $\mathcal{Z}$ at the root.
   a. Remove i from $\mathsf{Reports}$ if $(\mathsf{nonce}_i, \mathsf{pub}_i, \mathsf{key}_{(i,b)}, \pi^{\mathsf{szk}}_{(i,b)})$ does not follow the correct formatting ▷ Input-Formatting check.
   b. Otherwise, Sim computes $\beta'_i := [\![\beta_i]\!]_0 + [\![\beta_i]\!]_1$, where $[\![\beta_i]\!]_b := \mathcal{V}.\mathsf{EvalRoot}^{\mathrm{H}_1}(\mathsf{key}_{(i,b)}, \mathsf{pub}_i)$ for $b \in \{0,1\}$.
   Sim removes i from $\mathsf{Reports}$ if $\beta'_i \notin \mathcal{L}$. ▷ Weight-check.
2. **For each** level $k \in [0, \ldots, n-1]$ and prefix $p \in \mathsf{HH}^k$:
   a. **For each** candidate report $i \in \mathsf{Reports}$: ▷ Path & One-hot Verifiability checks.
      i. Each $\mathcal{S}_b$ retrieves the state $([\![y^p_i]\!]_b, \mathsf{st}^p_{(i,b)}, \pi^p_{(i,b)})$ from memory corresponding to prefix p and client $\mathcal{C}_i$.
      ii. Each $\mathcal{S}_b$ runs as $([\![y^\gamma_i]\!]_b, \mathsf{st}^\gamma_{(i,b)}, \pi^\gamma_{(i,b)}) := \mathcal{V}.\mathsf{Eval}^{\mathrm{H}_1}(\mathsf{key}_{(i,b)}, \mathsf{pub}_i, \gamma, \mathsf{st}^p_{(i,b)}, \pi^p_{(i,b)})$ for each prefix $\gamma \in \{p \,\|\, 0, p \,\|\, 1\}$ and stores the results in memory.
      iii. The aggregators check that the output for prefix p is equal to the sum of the outputs on prefixes $p \,\|\, 0$ and $p \,\|\, 1$. To do so, each $\mathcal{S}_b$ computes $h^p_{(i,b)} := (-1)^b \cdot ([\![y^p_i]\!]_b - [\![y^{p\|0}_i]\!]_b - [\![y^{p\|1}_i]\!]_b)$. ▷ Observe that $h^p_{(i,0)} = h^p_{(i,1)}$.
      iv. $\mathcal{S}_b$ accumulates its local state as $R^k_{(i,b)} := \mathrm{H}\big(||_{p \in \mathsf{HH}^k}(p, h^p_{(i,b)}, \pi^{p\|0}_{(i,b)}, \pi^{p\|1}_{(i,b)})\big)$. ▷ This is for all heavy-hitter prefixes.
      v. $\mathcal{S}_b$ sends $R^k_{(i,b)}$ to $\mathcal{S}_{1-b}$. If $R^k_{(i,0)} \neq R^k_{(i,1)}$, then $\mathcal{S}_b$ removes i from $\mathsf{Reports}$. ▷ One hash for each client.
   b. **For each** k-bit heavy-hitting prefix $p \in \mathsf{HH}^k$ the aggregators prune on $\gamma \in \{p \,\|\, 0, p \,\|\, 1\}$ as: ▷ Aggregation & Pruning.
      i. Each $\mathcal{S}_b$ accumulates $[\![\mathsf{weight}^\gamma]\!]_b := [\![\mathsf{weight}^\gamma]\!]_b + [\![y^\gamma_i]\!]_b$. ▷ Each $[\![y^\gamma_i]\!]_b$ is a vector of field elements $\mathbb{F}^m$.
      ii. $\mathcal{S}_0$ and $\mathcal{S}_1$ recover $\mathsf{weight}^\gamma := [\![\mathsf{weight}^\gamma]\!]_0 + [\![\mathsf{weight}^\gamma]\!]_1$. If $\mathsf{order}(\mathsf{weight}^\gamma) < \mathsf{T}$, then prune γ from the candidate prefix set. Otherwise, accumulate $\mathsf{HH}^{k+1} := \mathsf{HH}^{k+1} \cup \{\gamma\}$. ▷ $\mathsf{order}(\cdot)$ is decided by the aggregators.
3. Finally, the servers output HH^n as the set of weighted T-heavy-hitters.

Fig. 19: Hybrid HYB_1 for the Robustness Proof. Changes from HYB_0 are highlighted.

Hybrid HYB_{2j}

Sim simulates the role of the honest aggregators in this protocol.

Primitives:

1. A VIDPF $(\mathcal{V}.\mathsf{Gen}, \mathcal{V}.\mathsf{Eval}, \mathcal{V}.\mathsf{EvalRoot})$ as defined in Section 2.3.
2. A shared ZK $(\mathcal{Z}.\mathsf{Prove}, \mathcal{Z}.\mathsf{Query}, \mathcal{Z}.\mathsf{Decide})$ for a language $\mathcal{L} \subseteq \mathbb{F}^m$ as defined in Section 2.4.
3. Functions H, H_1, H_2 modeled in our analysis as random oracles.

Client Computation:

Input: Each client $\mathcal{C}_i$ for $i \in [\mathsf{N}]$ holds measurement $(\alpha_i, \beta_i) \in (\{0,1\}^n, \mathcal{L})$ composed of an input α_i and its weight β_i.

1. $\mathcal{C}_i$ runs $(\mathsf{pub}_i, \mathsf{key}_{(i,0)}, \mathsf{key}_{(i,1)}) \xleftarrow{\$} \mathcal{V}.\mathsf{Gen}^{\mathrm{H}_1}(\alpha_i, \beta_i)$.
2. $\mathcal{C}_i$ runs $(\pi^{\mathsf{szk}}_{(i,0)}, \pi^{\mathsf{szk}}_{(i,1)}, \mathsf{nonce}_i) \xleftarrow{\$} \mathcal{Z}.\mathsf{Prove}^{\mathrm{H}_2}([\![\beta_i]\!]_0, [\![\beta_i]\!]_1)$ where $[\![\beta_i]\!]_b \coloneqq \mathcal{V}.\mathsf{EvalRoot}^{\mathrm{H}_1}(\mathsf{key}_{(i,b)}, \mathsf{pub}_i)$ for $b \in \{0,1\}$.
3. $\mathcal{C}_i$ sends report share $(\mathsf{nonce}_i, \mathsf{pub}_i, \mathsf{key}_{(i,b)}, \pi^{\mathsf{szk}}_{(i,b)})$ to $\mathcal{S}_b$ for each $b \in \{0,1\}$.

Aggregator Computation:

Input: The simulated aggregators $\mathcal{S}_0$ and $\mathcal{S}_1$ start with a verification key $vk \in \{0,1\}^{\mathsf{vkl}}$ established out-of-band. Each sets $\mathsf{HH}^{\leq n} \coloneqq \{\mathsf{HH}^0, \mathsf{HH}^1, \ldots \mathsf{HH}^n\}$ as $\{\epsilon, \emptyset, \ldots, \emptyset\}$, the initial set of *candidate prefixes* for each level and sets $\mathsf{Reports} \coloneqq [\mathsf{N}]$, the initial set of *candidate reports*.

1. **For each** client $i \in \mathsf{Reports}$: ▷ Weight check using $\mathcal{Z}$ at the root.
   a. Remove i from $\mathsf{Reports}$ if $(\mathsf{nonce}_i, \mathsf{pub}_i, \mathsf{key}_{(i,b)}, \pi^{\mathsf{szk}}_{(i,b)})$ does not follow the correct formatting. ▷ Input-Formatting check.
   b. Otherwise, Sim computes $\beta_i \coloneqq [\![\beta_i]\!]_0 + [\![\beta_i]\!]_1$, where $[\![\beta_i]\!]_b \coloneqq \mathcal{V}.\mathsf{EvalRoot}^{\mathrm{H}_1}(\mathsf{key}_{(i,b)}, \mathsf{pub}_i)$ for $b \in \{0,1\}$. Sim removes i from $\mathsf{Reports}$ if $\beta_i \notin \mathcal{L}$
   ▷ Weight-check.
2. **For each** client $i \in \mathsf{Reports}$: Sim extracts the $\alpha_{i,\leq j-1}$ as follows. Initialize $r \coloneqq \epsilon$ and $\alpha_i \coloneqq \epsilon$. For $b \in \{0,1\}$, Sim sets $[\![y_i^r]\!]_b \coloneqq \epsilon, \mathsf{st}^r_{(i,b)} \coloneqq \epsilon$, $\pi^r_{(i,b)} \coloneqq \epsilon$ and store them in memory. **For** $k \in [0, 1, \ldots, j-1]$ run the following:
   a. Each $\mathcal{S}_b$ retrieves the state $([\![y_i^r]\!]_b, \mathsf{st}^r_{(i,b)}, \pi^r_{(i,b)})$ from memory corresponding to prefix r.
   b. Each $\mathcal{S}_b$ runs $([\![y_i^\gamma]\!]_b, \mathsf{st}^\gamma_{(i,b)}, \pi^\gamma_{(i,b)}) \coloneqq \mathcal{V}.\mathsf{Eval}^{\mathrm{H}_1}(\mathsf{key}_{(i,b)}, \mathsf{pub}_i, \gamma, \mathsf{st}^\gamma_{(i,b)}, \pi^\gamma_{(i,b)})$ for each prefix $\gamma \in \{r \,\|\, 0, r \,\|\, 1\}$ and stores the results in memory.
   c. For $\gamma \in \{r \,\|\, 0, r \,\|\, 1\}$, Sim computes $y_i^\gamma \coloneqq [\![y_i^\gamma]\!]_0 + [\![y_i^\gamma]\!]_1$.
   d. If any of the three conditions hold then Sim considers (α_i', β_i') as the ith client's measurement and skips this inner and the outer loop for this particular value of i:
      i. Both $\pi^{r\|0}_{(i,0)} \neq \pi^{r\|0}_{(i,1)} \wedge \pi^{r\|1}_{(i,0)} \neq \pi^{r\|1}_{(i,1)}$,
      ii. Both $y_i^{r\|0} \neq 0$ and $y_i^{r\|1} \neq 0$, or ▷ One-hot verifiability.
      iii. $y_i^{r\|0} + y_i^{r\|1} \neq \beta_i'$. ▷ Path verifiability.
   e. If $(y_i^{r\|0} = \beta_i)$ then update $r \coloneqq r \,\|\, 0$ and $\alpha_i \coloneqq \alpha_i \,\|\, 0$. Otherwise, if $(y_i^{r\|1} = \beta_i)$ then update $r \coloneqq r \,\|\, 1$ and $\alpha_i' \coloneqq \alpha_i \,\|\, 1$
3. Compute $\mathsf{HH}^{\leq j}$ as follows: For $k \in [0, \ldots, j-1]$ and for each prefix $p \in \mathsf{HH}^k$, consider $\gamma \in \{p \,\|\, 0, p \,\|\, 1\}$ and update $\mathsf{HH}^{k+1} \coloneqq \mathsf{HH}^{k+1} \cup \gamma$ if $\mathsf{order}(\mathsf{weight}^\gamma) > \mathsf{T}$, where weight^γ is sum of weights β_i of each input α_i with prefix γ. More formally:

$$\mathsf{weight}^\gamma \coloneqq \sum_i \beta_i \text{ for } i \in [\mathsf{Reports}] \wedge (\alpha_{i,\leq k+1} = \gamma).$$

Fig. 20: Hybrid HYB_{2j} for the Robustness Proof (Cont. in Fig. 21). Changes from HYB_{2j-1} are highlighted.

Hybrid HYB_{2j}

4. At the jth level - For each prefix $p \in \mathsf{HH}^j$:
   a. **For each** candidate report $i \in \mathsf{Reports}$: ▷ Path & One-hot Verifiability checks.
      i. Each $\mathcal{S}_b$ retrieves the state $(\llbracket y_i^p \rrbracket_b, \mathsf{st}^p_{(i,b)}, \pi^p_{(i,b)})$ from memory corresponding to prefix p and client $\mathcal{C}_i$.
      ii. Each $\mathcal{S}_b$ runs as $(\llbracket y_i^\gamma \rrbracket_b, \mathsf{st}^\gamma_{(i,b)}, \pi^\gamma_{(i,b)}) \coloneqq \mathcal{V}.\mathsf{Eval}^{\mathrm{H}_1}(\mathsf{key}_{(i,b)}, \mathsf{pub}_i, \gamma, \mathsf{st}^p_{(i,b)}, \pi^p_{(i,b)})$ for each prefix $\gamma \in \{p \,\|\, 0, p \,\|\, 1\}$ and stores the results in memory.
      iii. The aggregators check that the output for prefix p is equal to the sum of the outputs on prefixes $p \,\|\, 0$ and $p \,\|\, 1$. To do so, each $\mathcal{S}_b$ computes $h^p_{(i,b)} \coloneqq (-1)^b \cdot (\llbracket y_i^p \rrbracket_b - \llbracket y_i^{p\|0} \rrbracket_b - \llbracket y_i^{p\|1} \rrbracket_b)$. ▷ Observe that $h^p_{(i,0)} = h^p_{(i,1)}$.
      iv. $\boxed{\mathcal{S}_b \text{ accumulates its local state as } R^j_{(i,b)} \coloneqq \big(\big|\big|_{p \in \mathsf{HH}^j}(p, h^p_{(i,b)}, \pi^{p\|0}_{(i,b)}, \pi^{p\|1}_{(i,b)})\big).}$ ▷ This is for all heavy-hitter prefixes.
      v. $\mathcal{S}_b$ sends $R^j_{(i,b)}$ to $\mathcal{S}_{1-b}$. If $R^k_{(i,0)} \neq R^k_{(i,1)}$, then $\mathcal{S}_b$ removes i from $\mathsf{Reports}$. ▷ One hash for each client.
   b. **For each** j-bit heavy-hitting prefix $p \in \mathsf{HH}^j$ the aggregators prune on $\gamma \in \{p \,\|\, 0, p \,\|\, 1\}$ as: ▷ Aggregation & Pruning.
      i. Each $\mathcal{S}_b$ accumulates $\llbracket \mathsf{weight}^\gamma \rrbracket_b \coloneqq \llbracket \mathsf{weight}^\gamma \rrbracket_b + \llbracket y_i^\gamma \rrbracket_b$. ▷ Each $\llbracket y_i^\gamma \rrbracket_b$ is a vector of field elements $\mathbb{F}^m$.
      ii. $\mathcal{S}_0$ and $\mathcal{S}_1$ recover $\mathsf{weight}^\gamma \coloneqq \llbracket \mathsf{weight}^\gamma \rrbracket_0 + \llbracket \mathsf{weight}^\gamma \rrbracket_1$. If $\mathsf{order}(\mathsf{weight}^\gamma) < \mathsf{T}$, then prune γ from the candidate prefix set. Otherwise, accumulate $\mathsf{HH}^{j+1} \coloneqq \mathsf{HH}^{j+1} \cup \{\gamma\}$.
5. **For each** level $k \in [j+1, \ldots, n-1]$ and prefix $p \in \mathsf{HH}^k$:
   a. **For each** candidate report $i \in \mathsf{Reports}$: ▷ Path & One-hot Verifiability checks.
      i. Each $\mathcal{S}_b$ retrieves the state $(\llbracket y_i^p \rrbracket_b, \mathsf{st}^p_{(i,b)}, \pi^p_{(i,b)})$ from memory corresponding to prefix p and client $\mathcal{C}_i$.
      ii. Each $\mathcal{S}_b$ runs as $(\llbracket y_i^\gamma \rrbracket_b, \mathsf{st}^\gamma_{(i,b)}, \pi^\gamma_{(i,b)}) \coloneqq \mathcal{V}.\mathsf{Eval}^{\mathrm{H}_1}(\mathsf{key}_{(i,b)}, \mathsf{pub}_i, \gamma, \mathsf{st}^p_{(i,b)}, \pi^p_{(i,b)})$ for each prefix $\gamma \in \{p \,\|\, 0, p \,\|\, 1\}$ and stores the results in memory.
      iii. The aggregators check that the output for prefix p is equal to the sum of the outputs on prefixes $p \,\|\, 0$ and $p \,\|\, 1$. To do so, each $\mathcal{S}_b$ computes $h^p_{(i,b)} \coloneqq (-1)^b \cdot (\llbracket y_i^p \rrbracket_b - \llbracket y_i^{p\|0} \rrbracket_b - \llbracket y_i^{p\|1} \rrbracket_b)$. ▷ Observe that $h^p_{(i,0)} = h^p_{(i,1)}$.
      iv. $\mathcal{S}_b$ accumulates its local state as $R^k_{(i,b)} \coloneqq \mathrm{H}\big(\big|\big|_{p \in \mathsf{HH}^k}(p, h^p_{(i,b)}, \pi^{p\|0}_{(i,b)}, \pi^{p\|1}_{(i,b)})\big)$. ▷ This is for all heavy-hitter prefixes.
      v. $\mathcal{S}_b$ sends $R^k_{(i,b)}$ to $\mathcal{S}_{1-b}$.
      vi. $\mathcal{S}_b$ computes $d' \coloneqq R^k_{(i,0)} = R^k_{(i,1)}$. If $d' = False$, then remove i from $\mathsf{Reports}$.
   b. **For each** k-bit heavy-hitting prefix $p \in \mathsf{HH}^k$ the aggregators prune on $\gamma \in \{p \,\|\, 0, p \,\|\, 1\}$ as: ▷ Aggregation & Pruning.
      i. Each $\mathcal{S}_b$ accumulates $\llbracket \mathsf{weight}^\gamma \rrbracket_b \coloneqq \llbracket \mathsf{weight}^\gamma \rrbracket_b + \llbracket y_i^\gamma \rrbracket_b$. ▷ Each $\llbracket y_i^\gamma \rrbracket_b$ is a vector of field elements $\mathbb{F}^m$.
      ii. $\mathcal{S}_0$ and $\mathcal{S}_1$ recover $\mathsf{weight}^\gamma \coloneqq \llbracket \mathsf{weight}^\gamma \rrbracket_0 + \llbracket \mathsf{weight}^\gamma \rrbracket_1$. If $\mathsf{order}(\mathsf{weight}^\gamma) < \mathsf{T}$, then prune γ from the candidate prefix set. Otherwise, accumulate $\mathsf{HH}^{k+1} \coloneqq \mathsf{HH}^{k+1} \cup \{\gamma\}$. ▷ $\mathsf{order}(\cdot)$ is decided by the aggregators.
6. Finally, the servers output HH^n as the set of weighted T-heavy-hitters.

Fig. 21: Hybrid HYB_{2j} for the Robustness Proof (Cont. from Fig. 20).

Hybrid HYB_{2j+1}

Sim simulates the role of the honest aggregators in this protocol.

Primitives:

1. A VIDPF $(\mathcal{V}.\mathsf{Gen}, \mathcal{V}.\mathsf{Eval}, \mathcal{V}.\mathsf{EvalRoot})$ as defined in Section 2.3.
2. A shared ZK $(\mathcal{Z}.\mathsf{Prove}, \mathcal{Z}.\mathsf{Query}, \mathcal{Z}.\mathsf{Decide})$ for a language $\mathcal{L} \subseteq \mathbb{F}^m$ as defined in Section 2.4.
3. Functions H, H_1, H_2 modeled in our analysis as random oracles.

Client Computation:

Input: Each client $\mathcal{C}_i$ for $i \in [\mathsf{N}]$ holds measurement $(\alpha_i, \beta_i) \in (\{0,1\}^n, \mathcal{L})$ composed of an input α_i and its weight β_i.

1. $\mathcal{C}_i$ runs $(\mathsf{pub}_i, \mathsf{key}_{(i,0)}, \mathsf{key}_{(i,1)}) \xleftarrow{\$} \mathcal{V}.\mathsf{Gen}^{\mathrm{H}_1}(\alpha_i, \beta_i)$.
2. $\mathcal{C}_i$ runs $(\pi^{\mathsf{szk}}_{(i,0)}, \pi^{\mathsf{szk}}_{(i,1)}, \mathsf{nonce}_i) \xleftarrow{\$} \mathcal{Z}.\mathsf{Prove}^{\mathrm{H}_2}([\![\beta_i]\!]_0, [\![\beta_i]\!]_1)$ where $[\![\beta_i]\!]_b \coloneqq \mathcal{V}.\mathsf{EvalRoot}^{\mathrm{H}_1}(\mathsf{key}_{(i,b)}, \mathsf{pub}_i)$ for $b \in \{0,1\}$.
3. $\mathcal{C}_i$ sends report share $(\mathsf{nonce}_i, \mathsf{pub}_i, \mathsf{key}_{(i,b)}, \pi^{\mathsf{szk}}_{(i,b)})$ to $\mathcal{S}_b$ for each $b \in \{0,1\}$.

Aggregator Computation:

Input: The simulated aggregators $\mathcal{S}_0$ and $\mathcal{S}_1$ start with a verification key $vk \in \{0,1\}^{\mathsf{vkl}}$ established out-of-band. Each sets $\mathsf{HH}^{\leq n} \coloneqq \{\mathsf{HH}^0, \mathsf{HH}^1, \ldots \mathsf{HH}^n\}$ as $\{\epsilon, \emptyset, \ldots, \emptyset\}$, the initial set of *candidate prefixes* for each level and sets $\mathsf{Reports} \coloneqq [\mathsf{N}]$, the initial set of *candidate reports*.

1. **For each** client $i \in \mathsf{Reports}$: ▷ Weight check using $\mathcal{Z}$ at the root.
   a. Remove i from $\mathsf{Reports}$ if $(\mathsf{nonce}_i, \mathsf{pub}_i, \mathsf{key}_{(i,b)}, \pi^{\mathsf{szk}}_{(i,b)})$ does not follow the correct formatting. ▷ Input-Formatting check.
   b. Otherwise, Sim computes $\beta_i \coloneqq [\![\beta_i]\!]_0 + [\![\beta_i]\!]_1$, where $[\![\beta_i]\!]_b \coloneqq \mathcal{V}.\mathsf{EvalRoot}^{\mathrm{H}_1}(\mathsf{key}_{(i,b)}, \mathsf{pub}_i)$ for $b \in \{0,1\}$. Sim removes i from $\mathsf{Reports}$ if $\beta_i \notin \mathcal{L}$
   ▷ Weight-check.
2. **For each** client $i \in \mathsf{Reports}$: Sim extracts the $\boxed{\alpha_{i,\leq j}}$ as follows. Initialize $r \coloneqq \epsilon$ and $\alpha_i \coloneqq \epsilon$. For $b \in \{0,1\}$, Sim sets $[\![y_i^r]\!]_b \coloneqq \epsilon, \mathsf{st}^r_{(i,b)} \coloneqq \epsilon$, $\pi^r_{(i,b)} \coloneqq \epsilon$ and store them in memory. **For** $\boxed{k \in [0, 1, \ldots, j]}$ run the following:
   a. Each $\mathcal{S}_b$ retrieves the state $([\![y_i^r]\!]_b, \mathsf{st}^r_{(i,b)}, \pi^r_{(i,b)})$ from memory corresponding to prefix r.
   b. Each $\mathcal{S}_b$ runs $([\![y_i^\gamma]\!]_b, \mathsf{st}^\gamma_{(i,b)}, \pi^\gamma_{(i,b)}) \coloneqq \mathcal{V}.\mathsf{Eval}^{\mathrm{H}_1}(\mathsf{key}_{(i,b)}, \mathsf{pub}_i, \gamma, \mathsf{st}^\gamma_{(i,b)}, \pi^\gamma_{(i,b)})$ for each prefix $\gamma \in \{r \,\|\, 0, r \,\|\, 1\}$ and stores the results in memory.
   c. For $\gamma \in \{r \,\|\, 0, r \,\|\, 1\}$, Sim computes $y_i^\gamma \coloneqq [\![y_i^\gamma]\!]_0 + [\![y_i^\gamma]\!]_1$.
   d. If any of the three conditions hold then Sim considers (α_i', β_i') as the ith client's measurement and skips this inner and the outer loop for this particular value of i:
      i. Both $\pi^{r\|0}_{(i,0)} \neq \pi^{r\|0}_{(i,1)} \wedge \pi^{r\|1}_{(i,0)} \neq \pi^{r\|1}_{(i,1)}$,
      ii. Both $y_i^{r\|0} \neq 0$ and $y_i^{r\|1} \neq 0$, or ▷ One-hot verifiability.
      iii. $y_i^{r\|0} + y_i^{r\|1} \neq \beta_i'$. ▷ Path verifiability.
   e. If $(y_i^{r\|0} = \beta_i)$ then update $r \coloneqq r \,\|\, 0$ and $\alpha_i \coloneqq \alpha_i \,\|\, 0$. Otherwise, if $(y_i^{r\|1} = \beta_i)$ then update $r \coloneqq r \,\|\, 1$ and $\alpha_i' \coloneqq \alpha_i \,\|\, 1$
3. $\boxed{\text{Compute } \mathsf{HH}^{\leq j+1} \text{ as follows: For } k \in [0, \ldots, j]}$ and for each prefix $p \in \mathsf{HH}^k$, consider $\gamma \in \{p \,\|\, 0, p \,\|\, 1\}$ and update $\mathsf{HH}^{k+1} \coloneqq \mathsf{HH}^{k+1} \cup \gamma$ if $\mathsf{order}(\mathsf{weight}^\gamma) > \mathsf{T}$, where weight^γ is sum of weights β_i of each input α_i with prefix γ. More formally:

$$\mathsf{weight}^\gamma \coloneqq \sum_i \beta_i \text{ for } i \in [\mathsf{Reports}] \wedge (\alpha_{i,\leq k+1} = \gamma).$$

Fig. 22: Hybrid HYB_{2j+1} for the Robustness Proof (Cont. in Fig. 23). Changes from HYB_{2j} are $\boxed{\text{highlighted}}$.

Hybrid HYB_{2j+1}

4. At the jth level - For each prefix $p \in \mathsf{HH}^j$:
   a. **For each** candidate report $i \in \mathsf{Reports}$: ▷ Path & One-hot Verifiability checks.
      i. Each $\mathcal{S}_b$ retrieves the state $(\llbracket y_i^p \rrbracket_b, \mathsf{st}^p_{(i,b)}, \pi^p_{(i,b)})$ from memory corresponding to prefix p and client $\mathcal{C}_i$.
      ii. Each $\mathcal{S}_b$ runs as $(\llbracket y_i^\gamma \rrbracket_b, \mathsf{st}^\gamma_{(i,b)}, \pi^\gamma_{(i,b)}) \coloneqq \mathcal{V}.\mathsf{Eval}^{\mathrm{H}_1}(\mathsf{key}_{(i,b)}, \mathsf{pub}_i, \gamma, \mathsf{st}^p_{(i,b)}, \pi^p_{(i,b)})$ for each prefix $\gamma \in \{p \,\|\, 0, p \,\|\, 1\}$ and stores the results in memory.
      iii. The aggregators check that the output for prefix p is equal to the sum of the outputs on prefixes $p \,\|\, 0$ and $p \,\|\, 1$. To do so, each $\mathcal{S}_b$ computes $h^p_{(i,b)} \coloneqq (-1)^b \cdot (\llbracket y_i^p \rrbracket_b - \llbracket y_i^{p\|0} \rrbracket_b - \llbracket y_i^{p\|1} \rrbracket_b)$. ▷ Observe that $h^p_{(i,0)} = h^p_{(i,1)}$.
      iv. $\mathcal{S}_b$ accumulates its local state as $R^j_{(i,b)} \coloneqq \big(\big|\big|_{p \in \mathsf{HH}^j} \big(p, h^p_{(i,b)}, \pi^{p\|0}_{(i,b)}, \pi^{p\|1}_{(i,b)}\big)\big)$. ▷ This is for all heavy-hitter prefixes.
      v. $\mathcal{S}_b$ sends $R^j_{(i,b)}$ to $\mathcal{S}_{1-b}$. If $R^j_{(i,0)} \neq R^j_{(i,1)}$, then $\mathcal{S}_b$ removes i from $\mathsf{Reports}$.
      vi. For $\gamma \in \{p \,\|\, 0, p \,\|\, 1\}$: simulated $\mathcal{S}_0$ and $\mathcal{S}_1$ reconstruct $y_i^\gamma \coloneqq \llbracket y_i^\gamma \rrbracket_0 + \llbracket y_i^\gamma \rrbracket_1$.
   b. **For each** j-bit heavy-hitting prefix $p \in \mathsf{HH}^j$ the aggregators prune on $\gamma \in \{p \,\|\, 0, p \,\|\, 1\}$ as: ▷ Aggregation & Pruning.
      i. Each $\mathcal{S}_b$ accumulates $\llbracket \mathsf{weight}^\gamma \rrbracket_b \coloneqq \llbracket \mathsf{weight}^\gamma \rrbracket_b + \llbracket y_i^\gamma \rrbracket_b$. ▷ Each $\llbracket y_i^\gamma \rrbracket_b$ is a vector of field elements $\mathbb{F}^m$.
      ii. $\mathcal{S}_0$ and $\mathcal{S}_1$ recover $\mathsf{weight}^\gamma \coloneqq \llbracket \mathsf{weight}^\gamma \rrbracket_0 + \llbracket \mathsf{weight}^\gamma \rrbracket_1$. If $\mathsf{order}(\mathsf{weight}^\gamma) < \mathsf{T}$, then prune γ from the candidate prefix set. Otherwise, accumulate $\widetilde{\mathsf{HH}^{j+1}} \coloneqq \widetilde{\mathsf{HH}^{j+1}} \cup \{\gamma\}$.

   If $\mathsf{HH}^{j+1} \neq \widetilde{\mathsf{HH}^{j+1}}$ then return $(\perp, \perp)$ to $\mathcal{F}_{\mathsf{wHH}}$ for all corrupt clients.
5. **For each** level $k \in [j+1, \ldots, n-1]$ and prefix $p \in \mathsf{HH}^k$:
   a. **For each** candidate report $i \in \mathsf{Reports}$: ▷ Path & One-hot Verifiability checks.
      i. Each $\mathcal{S}_b$ retrieves the state $(\llbracket y_i^p \rrbracket_b, \mathsf{st}^p_{(i,b)}, \pi^p_{(i,b)})$ from memory corresponding to prefix p and client $\mathcal{C}_i$.
      ii. Each $\mathcal{S}_b$ runs as $(\llbracket y_i^\gamma \rrbracket_b, \mathsf{st}^\gamma_{(i,b)}, \pi^\gamma_{(i,b)}) \coloneqq \mathcal{V}.\mathsf{Eval}^{\mathrm{H}_1}(\mathsf{key}_{(i,b)}, \mathsf{pub}_i, \gamma, \mathsf{st}^p_{(i,b)}, \pi^p_{(i,b)})$ for each prefix $\gamma \in \{p \,\|\, 0, p \,\|\, 1\}$ and stores the results in memory.
      iii. The aggregators check that the output for prefix p is equal to the sum of the outputs on prefixes $p \,\|\, 0$ and $p \,\|\, 1$. To do so, each $\mathcal{S}_b$ computes $h^p_{(i,b)} \coloneqq (-1)^b \cdot (\llbracket y_i^p \rrbracket_b - \llbracket y_i^{p\|0} \rrbracket_b - \llbracket y_i^{p\|1} \rrbracket_b)$. ▷ Observe that $h^p_{(i,0)} = h^p_{(i,1)}$.
      iv. $\mathcal{S}_b$ accumulates its local state as $R^k_{(i,b)} \coloneqq \mathrm{H}\big(\big|\big|_{p \in \mathsf{HH}^k} \big(p, h^p_{(i,b)}, \pi^{p\|0}_{(i,b)}, \pi^{p\|1}_{(i,b)}\big)\big)$. ▷ This is for all heavy-hitter prefixes.
      v. $\mathcal{S}_b$ sends $R^k_{(i,b)}$ to $\mathcal{S}_{1-b}$. If $R^k_{(i,0)} \neq R^k_{(i,1)}$, then $\mathcal{S}_b$ removes i from $\mathsf{Reports}$. ▷ One hash for each client.
   b. **For each** k-bit heavy-hitting prefix $p \in \mathsf{HH}^k$ the aggregators prune on $\gamma \in \{p \,\|\, 0, p \,\|\, 1\}$ as: ▷ Aggregation & Pruning.
      i. Each $\mathcal{S}_b$ accumulates $\llbracket \mathsf{weight}^\gamma \rrbracket_b \coloneqq \llbracket \mathsf{weight}^\gamma \rrbracket_b + \llbracket y_i^\gamma \rrbracket_b$. ▷ Each $\llbracket y_i^\gamma \rrbracket_b$ is a vector of field elements $\mathbb{F}^m$.
      ii. $\mathcal{S}_0$ and $\mathcal{S}_1$ recover $\mathsf{weight}^\gamma \coloneqq \llbracket \mathsf{weight}^\gamma \rrbracket_0 + \llbracket \mathsf{weight}^\gamma \rrbracket_1$. If $\mathsf{order}(\mathsf{weight}^\gamma) < \mathsf{T}$, then prune γ from the candidate prefix set. Otherwise, accumulate $\mathsf{HH}^{k+1} \coloneqq \mathsf{HH}^{k+1} \cup \{\gamma\}$. ▷ $\mathsf{order}(\cdot)$ is decided by the aggregators.
6. Finally, the servers output HH^n as the set of weighted T-heavy-hitters.

Fig. 23: Hybrid HYB_{2j+1} for the Robustness Proof (Cont. from Fig. 22).