Verifiable Distributed Aggregation Functions

Hannah Davis
University of California, San Diego
h3davis@eng.ucsd.edu

Christopher Patton
Cloudflare
cpatton@cloudflare.com

Mike Rosulek
Oregon State University
rosulekm@eecs.oregonstate.edu

Phillipp Schoppmann
Google
schoppmann@google.com

ABSTRACT

The modern Internet is built on systems that incentivize collection of information about users. In order to minimize privacy loss, it is desirable to prevent these systems from collecting more information than is required for the application. The promise of *multi-party computation* is that data can be aggregated without revealing individual measurements to the data collector. This work offers a provable security treatment for "Verifiable Distributed Aggregation Functions (VDAFs)", a class of multi-party computation protocols being considered for standardization by the IETF.

We propose a formal framework for the analysis of VDAFs and apply it to two constructions. The first is Prio3, one of the candidates for standardization. This VDAF is based on the Prio system of Corrigan-Gibbs and Boneh (NSDI 2017). We prove that Prio3 achieves our security goals with only minor changes to the draft. The second construction, called Doplar, is introduced by this paper. Doplar is a round-reduced variant of the Poplar system of Boneh et al. (IEEE S&P 2021), itself a candidate for standardization. The cost of this improvement is a modest increase in overall bandwidth and computation.

KEYWORDS

multi-party computation, cryptographic standards

1 INTRODUCTION

Operating a complex software system, such as an operating system, web browser, or web service, often requires measuring the behavior of the system's users. When used for a specific purpose, such measurements are often only consumed in some aggregated form, e.g., $F(m_1, \ldots, m_{ct})$ for some specific function F, rather than the individual measurements $m_1, \ldots, m_{ct}$. But in conventional systems, the measurements are revealed to the operator as a matter of course, resulting in an increased capability to surveil users. Consider the following motivating examples:

(1) *Identifying misbehaving or malicious origins.* To detect bugs or attack vectors, a browser vendor might want to know how often establishing a connection to a given origin or loading a given web page triggers a specific event [45]. But logging these events and aggregating them in the clear risks exposing browser history.
(2) *Measuring ad conversion rates.* Today advertising is a significant revenue source for many web service providers. In order to accurately assess the value of an ad campaign, the service provider and advertiser might want to measure how many people who clicked on a given ad made a purchase [2].
(3) *Classifying malicious client behavior.* Many operators benefit from the ability to classify (or predict) user behavior automatically, and in real-time. For example, anomaly detection systems use machine learning models, trained and validated on requests from real clients, to classify fraudulent or otherwise malicious behavior [43].

These applications require only aggregates; by collecting individual measurements, the operator learns more information than is ultimately used for the intended purpose. One way out of this predicament is *multi-party computation (MPC)*, which allows computing some function of private inputs distributed across multiple parties, without revealing these private inputs. In this paper, we consider a class of MPC protocols in which the bulk of the computation is outsourced to a small set of non-colluding servers.

Recent attention from the MPC community on problems like these has yielded solutions that are practical enough for real-world deployment [5, 9, 15, 16, 20, 30]. Notable examples include Mozilla's Origin Telemetry project [45] and the COVID-19 Exposure Notification Private Analytics system developed jointly by Apple and Google [7]. The success of these projects spurred the formation of a working group within the Internet Engineering Task Force (IETF) whose objective is to standardize MPC for "Privacy-Preserving Measurement (PPM)" [1], thereby improving interoperability and providing a deployment roadmap for new schemes.

The primary goal of this paper is to lay some of the groundwork for the provable security analysis that will be needed to support this effort. We formalize a syntax and set of security definitions for a particular class of MPC protocols from the literature [5, 15, 16, 20] of interest to the working group. Our definitions unify previous ones into an explicit, game-based framework that accounts for practical matters not attended to in prior work.

We apply our definitional framework to two constructions. The first is a candidate for standardization based on the Prio scheme designed by Corrigan-Gibbs and Boneh [20]; we show that this protocol meets our security goals with only minor changes. Another candidate for standardization is the more recent Poplar scheme due to Boneh et al. [16]; we introduce and analyze a variant of this protocol that has improved round complexity.

Proceedings on Privacy Enhancing Technologies 2023(4), 578–592

https://doi.org/10.56553/popets-2023-0126

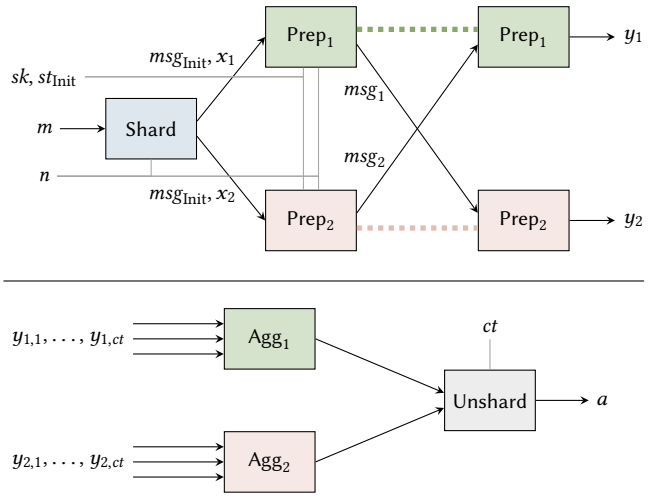

Figure 1: Illustration of (top) sharding and preparation of a single measurement and (bottom) aggregation and unsharding of a set of measurements. All parameters are defined in Section 3.

Overview. The PPM working group plans to develop multiple protocol standards, one of which is the focus of this work. The *Distributed Aggregation Protocol (DAP)* standard [28] centers around the execution of a particular class of MPC protocols, called *Verifiable Distributed Aggregation Functions (VDAFs)* [8]. A VDAF is used to securely compute some **aggregation function** F over a set of measurements generated by the **clients**. To protect their privacy, the measurements are secret-shared and the computation of the aggregate is distributed amongst multiple, non-colluding aggregation servers (called **aggregators** hereafter). Execution of a VDAF involves four basic steps (illustrated in Figure 1):

- Shard: Each client shards its measurement m_i into **input shares** and sends one share to each aggregator. In this work, we sometimes refer to this sequence of input shares as the client's **report**.
- Prepare: After receiving a report from a client, the aggregators gossip amongst themselves in order to prepare their shares for aggregation. This involves refining the shares into an aggregatable form and verifying that the outputs are "well-formed", e.g., that they correspond to an integer in a given range, or correspond to a one-hot vector (a vector that is non-zero in at most one position). We call the outputs of this process the **refined shares**.
- Aggregate: Once an aggregator has recovered the desired number of refined shares, it combines them into its share of the aggregate result, called an **aggregate share**. It then sends this to the data consumer, known as the **collector**.
- Unshard: Finally, the collector combines each of the aggregate shares into $F(m_1, \ldots, m_{ct})$.

Why standardize VDAFs? The case for standardizing this class of MPC protocols is made by the aforementioned deployments of Prio [7, 45], of which VDAFs are a natural generalization. The key feature that makes these protocols widely applicable and suited for Internet scale is that the expensive part of the computation (Shard/Prepare) is fully parallelizable across all reports being aggregated. This means that deployments can be scaled to such a degree that the time spent on executing the VDAF is primarily *network-bound* rather than *CPU-bound*. It is less clear (at least to those in the PPM working group) whether MPC techniques where the computations depend on all reports (e.g., oblivious sorting [50] or shuffling [6, 9]) would scale in the same way.

This feature also implies that VDAFs are only suitable for aggregation functions F that can be decomposed into f, g for which $F(m_1, \ldots, m_{ct}) = f(g(m_1), \ldots, g(m_{ct}))$, where g may be non-linear, but f must be affine. Indeed, the goal is not to encompass all possible MPC schemes, but a particular, useful, and highly parallelizable class of them. VDAFs can be used for a variety of aggregation tasks, including: simple statistics like sum, mean, standard deviation, quantile estimates, or linear regression [20]; a step of a gradient descent [35]; or heavy hitters (see below).

Security goals. The PPM working group's primary goal for VDAFs (cf. [28, Section 7]) is that they are **private** in the sense that the attacker learns nothing about the measurements $m_1, \ldots, m_{ct}$ beyond what it can infer from the aggregate result $F(m_1, \ldots, m_{ct})$. An active attacker who corrupts the collector and a fraction of the aggregators (typically all but one) and controls transmission of all messages in the protocol—except, of course, the input shares delivered to honest aggregators. Its corruptions are "static": the set of corrupt parties does not change over the course of the attack.

Another security consideration for VDAFs is that they are **robust** in the sense that the attacker cannot force the collector to compute anything other than the aggregate of honestly generated reports. Here the attacker is a set of malicious clients attempting to corrupt the aggregate result by sending malformed reports. For robustness we assume all of the aggregators execute the protocol correctly. Otherwise, a corrupt aggregator could trivially corrupt the result by sending the collector a malformed aggregate share.

We formalize these security notions in the game-playing paradigm [13]. First, in Section 3.2 we define privacy via an indistinguishability game $\mathsf{Exp}^{\mathrm{priv}}_{\Pi}(A)$ played by an attacker A against VDAF Π. The attacker interacts with the honest parties (i.e., the clients and uncorrupted aggregators) via a set of oracles. These oracles allow A to mount a kind of "chosen batch attack" in which the honest parties process one of two batches of measurements, and A's goal is to determine which was processed. This is analogous to the simulation-based definition of [20, Definition 1], which asks the the attacker to distinguish the protocol's execution from the view generated by a simulator.

We formalize robustness via a game $\mathsf{Exp}^{\mathrm{robust}}_{\Pi}(A)$ (Section 3.2). Here the attacker A—playing the role of a coalition of malicious clients—is given a single oracle that models the execution of the preparation step of VDAF execution on (invalid) reports. The attacker wins if an aggregator ever accepts an invalid share *or* if the aggregators compute refined shares that, when combined, do not correspond to a valid refined measurement. For natural VDAFs, robustness implies robustness in the sense of [20, Definition 6]: namely, the collector is guaranteed to correctly aggregate measurements uploaded by honest clients.

Note on the simulation paradigm. An alternative approach, and one that is more conventional for MPC, is to formulate security in the Universal Composability (UC) framework [19]. This methodology would begin by specifying the "ideal functionality" for computing an aggregation function such that, for any VDAF that securely realizes this functionality, any suitable notion of either privacy or robustness would follow from the UC composition theorem.

While this methodology is attractive, it creates the following difficulty in our setting. Many applications of VDAFs may be willing to tolerate a loose robustness bound (i.e., a non-negligible probability of accepting an invalid share) if doing so leads to better performance or communication. On the other hand, no application can accept a loose bound for privacy. In order to reason about this tradeoff, it is necessary to obtain explicit, concrete bounds for privacy and robustness *separately*. A theorem in the UC framework yields only a single bound, for the "UC-realizability" of the ideal functionality; applying this result directly would lead to parameter choices that might be more conservative than strictly necessary for the given application.

Another consideration is to make our results accessible to the target audience. Applying the UC framework, and interpreting its results, involves a number of subtleties that, based on our own observations, are often misunderstood when translated to practice.[1] One goal of our definitions is to make as explicit as possible all of the requirements an application like DAP [28] needs to meet in order to use VDAFs securely.

Previous definitions. Our definitions in Section 3 can be seen as a more precise (but not necessarily stronger) formulation of the informal definitions given in the original Prio paper [20, Appendix A]. While the authors mention the possibility of using a unified simulation-based security definition for privacy and robustness, they do not provide one.

For Poplar on the other hand, Boneh et al. [16, Appendix A] provide a simulation-based definition for the end-to-end functionality. In order to capture the fact that a malicious server can influence the output of the protocol, they define a leakage function that allows the attacker to perturb the aggregate result with an arbitrary additive offset. While we believe this captures the robustness attacks that are possible for Poplar, it does not immediately generalize to the broader class of functionalities we consider as VDAFs. Also note that Boneh et al. do not provide any proofs using their security definition. (The proofs they do provide are for definitions that are naturally captured by games, e.g., [16, Appendix D].) Finally, the simulation-based security definition of Poplar only considers a single security parameter, something that would need to be overcome to allow for separate security bounds for privacy and robustness.

Constructions. The starting point for our work is draft-irtf-cfrg-vdaf-03 [8], the current draft of the VDAF specification at the time of writing.

[1]For a recent example, consider the standards for PAKEs ("Password-Authenticated Key Exchange") developed by the CFRG. Most of these standards are based on protocols with analysis in the UC framework. For one protocol [4], one question left open by that analysis was how to securely instantiate the "session identifier", one of the artifacts of the ideal functionality. The current draft offers recommendations for choosing the session identifier, but allows applications to ignore this entirely; a game-playing argument was used to justify this (cf [3, Appendix B]).

The first scheme described in draft-03, called Prio3, is based on Prio [20], but incorporates performance improvements from Boneh et al. [15] (hereafter BBCG+19). Prio3 can be used to compute a wide variety of aggregation functions due to its use of *Fully Linear Proofs (FLPs)*. Briefly, an FLP is a special type of zero-knowledge proof that allows the client's input measurement to be validated by the aggregators (e.g., ensure that it is a number in some pre-determined range) who have only secret shares of the input and proof. The FLP designed by BBCG+19 (see [15, Theorem 4.3]) and adopted by the draft (with minor modifications; see [8, Section 7.3]) is expressed in terms of some arithmetic circuit C that takes in the prover's input x and a random string jr computed jointly by the prover and verifier. Computing this joint randomness, verifying the proof, and evaluating $C(x, jr)$ requires just one round of communication among the aggregators.

In Section 4, we prove Prio3 is both robust (Theorem 4.2) and private (Theorem 4.3) under the assumption that the underlying FLP is, respectively, *sound* and *honest-verifier zero-knowledge* as defined by BBCG+19. Our analysis unveiled a few subtle design issues in draft-03 that we address here.

The second scheme in draft-03 is called Poplar1 and is based on the recent Poplar protocol from Boneh et al. [16] (BBCG+21). Poplar is designed to solve the private "heavy hitters" problem in which each client submits an arbitrary bitstring α and the collector wants to compute the set of unique strings that occurred at least T times. The key idea of BBCG+21 is an extension of *distributed point functions (DPFs)* [29], where two aggregators hold a share of a "DPF key" that concisely represents a *point function*. A point function evaluates to 0 on every input, except for the distinguished point α, where the function evaluates to some $\beta \neq 0$. By secret sharing the DPF keys generated by the clients, the aggregators can count *how many* clients submitted a particular candidate string without revealing *which* clients submitted it.

Poplar1 makes use of an enriched primitive called an *incremental DPF (IDPF)*. IDPF keys can be queried not only at a given point, but a given *prefix*. That is, an *incremental point function* is one that evaluates to 0 on every input except for the set of strings that are a prefix of α. This new primitive gives rise to an efficient solution to the heavy hitters problem that involves running Poplar1 multiple times over the same set of IDPF keys, where each run begins with a set of candidate prefixes computed from the previous run.

To achieve robustness, Poplar1 uses a two-round multi-party computation in which the aggregators verify that the IDPF outputs are well-formed. That means that, compared to Prio3, the Poplar1 VDAF costs one additional round of communication, per report, during the preparation phase. The additional roundtrip is significant from an operational perspective.

In Section 5 we introduce *Doplar*, our modification to Poplar which achieves a one-round preparation. To achieve this, we combine FLPs and methods from distributed point functions in a novel way. We adopt a point-function verification method from De Castro and Polychroniadou [24]. We also introduce a new flavor of *delayed-input* FLPs, which may be of independent interest.

Related Work. Several works have considered private aggregate statistics, relying either on secret-sharing between non-colluding servers [21, 25, 27, 37, 39, 41], or on anonymization networks [18,

33, 47]. However, these works either do not provide privacy against malicious clients or rely on expensive zero-knowledge proofs.

A protocol for Secure Aggregation (SecAgg) in the single-server setting was presented by Bonawitz et al. [14] and subsequently improved by Bell et al. [10, 11]. While SecAgg can provide security against malicious parties, it relies on multiple rounds of interaction between clients and server.

The VDAF abstraction was designed to encompass the architecture of Prio and Poplar in which the expensive portion of the MPC is fully parallelizable. Another example of a VDAF from the literature is the protocol of Addanki et al. [5], which uses boolean (bit-wise) secret sharing instead of arithmetic circuit to improve communication cost from client to aggregator. However, this comes at a cost of weaker privacy, since their protocol does not protect against malicious servers.

There are also protocols that do not fit neatly into the VDAF framework as specified, but which might be adapted into VDAFs in the future. Masked LARK [35] is a proposal by Microsoft for training machine learning models on private data, using secret-sharing and MPC between a set of aggregators. AdScale [30] presents an aggregation system focused on private ads measurement. While designed for a single aggregation server, their construction appears to be amenable to our multi-server setting.

Other protocols in the literature share the same security goals of VDAFs, but do not have the same streaming architecture. One example is the recent "Oblivious Shuffling" protocol due to Anderson et al. [6], which involves an MPC, assisted by a third-party, for unlinking each report from the client that sent it. The online processing for this procedure intrinsically involves all of the reports being shuffled; for VDAFs, all of the online processing is per-report. Similarly, Bell et al. [9] present a protocol for computing sparse histograms with two aggregators that is more efficient than DPFs for large domains, but reveals differentially private views to the aggregators. Again, the protocol crucially relies on shuffling contributions from multiple users. Vogue [36] is a protocol for computing private heavy hitters using three non-colluding servers. The protocol is secure against malicious servers and clients, but again relies on shuffling. Finally, the STAR protocol [22] uses an anonymizing proxy to ensure the collector only learns "popular" measurements, while any measurement that occurs less than a pre-determined threshold is not revealed to any party.

In recent concurrent work, Mouris et al. [44] present another three-party, honest-majority protocol for computing heavy hitters. Their full protocol relies on a secure comparison protocol that is run after the aggregation phase, and thus doesn't immediately fit our setting. However, we believe their input validation protocol can be adapted to obtain a VDAF for heavy hitters that has similar characteristics as our protocol in Section 5. (Indeed their core primitive, which they also call "Verifiable IDPF", bears a striking resemblence to our own VIDPF abstraction.) Likewise, one could get robustness against malicious aggregators in the honest-majority setting by applying their "duplicate aggregator" technique to our protocols. We leave exploration of how to combine our results to future work.

Full version. This is the proceedings version of our paper. The full version [23] includes proofs of all theorems, a notion of "completeness" for VDAFs, and additional remarks and commentary.

2 PRELIMINARIES

This section describes cryptographic primitives on which our constructions are based. We begin with a bit of non-standard notation.

Notation. Let $[i..j]$ denote the set of integers $\{i, \ldots, j\}$ and write $[i]$ as shorthand for the set $[1..i]$. If $\vec{v}$ is a vector, let $\vec{v}[i]$ denote the i-th element of $\vec{v}$. Let $(x,)$ denote the singleton vector with value x and $()$ the empty vector.

In our pseudocode, all variables that are undeclared implicitly have the value $\perp$. Let $y \xleftarrow{\$} S$ denote sampling y uniformly from a finite set S; let $y \xleftarrow{\$} A(x)$ denote execution of randomized algorithm A; and let $y \leftarrow A(x; r)$ denote execution of randomized algorithm A with coins r. If X is a random variable with support $\{0, 1\}$ we let $\Pr[X]$ denote the probability that $X = 1$.

A table T is a map from unique keys to values; we write $\mathrm{T}[K_1, \ldots]$ to denote the value corresponding to key $K_1, \ldots$. We sometimes write a dot "$\cdot$" in place of one of the elements of the key, e.g., "$\mathrm{T}[K_1, \cdot]$" instead of "$\mathrm{T}[K_1, K_2]$". We use this notation to denote the vector of values in the table that match the key pattern. For example, we write $\mathrm{T}[K_1, \cdot]$ for the vector $(\mathrm{T}[K_1, K_2^1], \ldots, \mathrm{T}[K_1, K_2^n])$ where $(K_1, K_2^1), \ldots, (K_1, K_2^n)$ are all of the keys in the table prefixed by K_1, in lexicographic order.

We measure an adversary's runtime by the time it takes to run its experiment to completion, including evaluating its queries.

Pseudorandom Generators. The VDAF spec [8, Section 6.2] calls for a particular type of object they call "pseudorandom generator (PRG)". Unlike the conventional PRGs, these objects are stateful. A PRG is comprised of the following algorithms:

- $\mathsf{PRG.Init}(seed \in \{0,1\}^\kappa, cntxt \in \{0,1\}^*) \to st \in Q$ takes a seed and context string to the initial PRG state. We call κ the **seed length**.
- $\mathsf{PRG.Next}(st \in Q, \ell \in \mathbb{N}) \to (st' \in Q, out \in \{0,1\}^\ell)$ takes in the current PRG state and outputs a string of the desired length.

We also make use of an algorithm $\mathsf{Expand}[\mathsf{PRG}]$ that uses the given PRG to map a seed and context string to a vector of integers over the modular ring $\mathbb{Z}_p$ for the desired modulus p. We defer to [8, Section 6.2] for the full definition of $\mathsf{Expand}[\mathsf{PRG}]$.

In our security proofs, we model PRGs as random oracles [12]. In some cases, such as the distributed point functions (DPFs) in Section 5.1, constructions based on computational assumptions are known to be sufficient. We refer to Guo et al. [31, 32] for an overview of the state-of-the-art PRGs for DPFs and similar constructions.

Fully Linear Proof Systems. We recall the definition of FLP systems from BBCG+19 [15]. (Our formulation differs slightly, as we discuss in the full version.) FLPs allow a prover to prove to a verifier, in zero-knowledge, that a secret-shared value has some property required by the application, e.g., the input is a number in the desired range, is a one-hot vector, etc. (The main construction of BBCG+19 allows the validity condition to be expressed in terms of an arithmetic circuit evaluated over the input, similar to more conventional zero-knowledge proof systems.) They are "fully linear" in the sense that verifying the proof involves computing a strictly linear function over both the input and proof. This allows verification to be performed on secret-shared data, leveraging its additive homomorphism property.

Algorithm $\mathsf{View}_{\mathsf{FLP}}(x)$:

1 $jr \leftarrow\!\!\$\ \mathbb{F}^{jl}$; $qr \leftarrow\!\!\$\ \mathbb{F}^{ql}$
2 $\pi \leftarrow\!\!\$\ \mathsf{Prove}(x, jr)$
3 $\sigma \leftarrow \mathsf{Query}(x, \pi, jr;\ qr)$
4 ret $jr \parallel qr \parallel \sigma$

Algorithm $\mathsf{Err}_{\mathsf{FLP}}(P^*)$:

5 $(st_{P^*}, x) \leftarrow\!\!\$\ P^*(\,)$; $jr \leftarrow\!\!\$\ \mathbb{F}^{jl}$
6 $\pi \leftarrow\!\!\$\ P^*(st_{P^*}, jr)$
7 $\sigma \leftarrow\!\!\$\ \mathsf{Query}(x, \pi, jl)$
8 ret $x \notin \mathcal{L} \wedge \mathsf{Decide}(\sigma)$

Figure 2: Procedures for defining security of FLPs.

An FLP with finite field $\mathbb{F}$, proof length m, verifier length v, prover randomness length pl, joint randomness length jl, and query randomness length ql is a triple of algorithms FLP defined as follows:

- $\mathsf{FLP.Prove}(x \in \mathbb{F}^n, jr \in \mathbb{F}^{jl}) \rightarrow \pi \in \mathbb{F}^m$ is the randomized **proof-generation** algorithm that takes in an input x and joint randomness jr and outputs a proof string $\pi \in \mathbb{F}^m$. We shall assume this algorithm generates random coins by sampling uniformly from $\mathbb{F}^{pl}$.
- $\mathsf{FLP.Query}(x \in \mathbb{F}^n, \pi \in \mathbb{F}^m, jr \in \mathbb{F}^{jl}) \rightarrow \sigma \in \mathbb{F}^v$ is the randomized **query-generation** algorithm that takes in an input x, proof string π, and joint randomness jr and outputs a verifier string σ. We shall assume the random coins are sampled uniformly from $\mathbb{F}^{ql}$.
- $\mathsf{FLP.Decide}(\sigma \in \mathbb{F}^v) \rightarrow acc \in \{0,1\}$ is the deterministic **decision predicate** that takes in a verifier string σ and outputs a bit acc indicating whether the input is valid.

We require the field $\mathbb{F}$ to have prime order; we occasionally denote its order by $\mathbb{F}.p$. We say that FLP is *fully linear* if the query-generation algorithm computes a linear function of the input and proof. That is, there exists a function Q whose output is a matrix in $\mathbb{F}^{v \times (n+m)}$ and, for all inputs x, proofs π, joint randomnesses jr, and query randomnesses qr, it holds that $\mathsf{Query}(x, \pi, jr;\ qr) = Q(jr;\ qr) \cdot (x \parallel \pi) \in \mathbb{F}^v$.

Associated with FLP is a language $\mathcal{L} \subseteq \mathbb{F}^n$. We say that FLP is **complete for** $\mathcal{L}$ if the proof system outputs 1 whenever the input is in $\mathcal{L}$. That is, for all $x \in \mathcal{L}$ it holds that

$$\Pr[\mathsf{Decide}(\sigma) : jr \leftarrow\!\!\$\ \mathbb{F}^{jl}; \pi \leftarrow\!\!\$\ \mathsf{Prove}(x, jr); \sigma \leftarrow\!\!\$\ \mathsf{Query}(x, \pi, jr)] = 1\,.$$

We define soundness of FLP in terms of experiment $\mathsf{Err}_{\mathsf{FLP}}(P^*)$ shown in Figure 2 associated with a malicious prover P^*. In this experiment, the prover commits to an invalid input $x \in \mathbb{F}^n \setminus \mathcal{L}$. Next, joint randomness jr is generated and given to P^*, who then generates a proof π. Finally, the verifier is run on x, π, jr; the malicious prover "wins" if the verifier deems the input valid. We say FLP is ϵ-**sound for** $\mathcal{L}$ if for all P^* it holds that $\Pr\left[\ \mathsf{Err}_{\mathsf{FLP}}(P^*)\ \right] \leq \epsilon$.

Let $\mathsf{View}_{\mathsf{FLP}}(x)$ denote the procedure defined in Figure 2. We say FLP is δ-**statistical, strong, honest-verifier zero-knowledge**—or, simply, δ-**private**—if the verifier's view can be simulated without knowledge of the input. That is, there exists a randomized algorithm S such that for all $x \in \mathcal{L}$ it holds that

$$\sum_{\omega} \left|\Pr\left[\ \mathsf{View}_{\mathsf{FLP}}(x) = \omega\ \right] - \Pr\left[\ S(\,) = \omega\ \right]\right| \leq \delta\,.$$

Incremental Distributed Point Functions. A point function is a function that is 0 everywhere except on a special input α; an incremental point function is a function that is 0 everywhere except on *any prefix of* α. One can imagine arranging the co-domain of this function into a complete, binary tree in which the nodes are labeled with prefixes; and for each node labeled p, its children are labeled with $p \parallel 0$ and $p \parallel 1$. Each node on the path to the leaf node α is assigned a non-0 value, and all other nodes are assigned 0. (See [16, Figure 4] for an illustration.)

An incremental point function that gives output $\vec{\beta}[\ell]$ on the length-ℓ prefix of α is defined formally as:

$$f_{\alpha,\vec{\beta}}\left(pfx \in \{0,1\}^{\leq \eta}\right) = \begin{cases} \vec{\beta}\left[\ |pfx|\ \right] & \text{if } pfx \text{ is a prefix of } \alpha \\ \mathbf{0} & \text{otherwise.} \end{cases}$$

An *Incremental Distributed Point Function (IDPF)* [16] is a concise secret sharing of an incremental point function. We recall the definition of an IDPF from Boneh et al. [16] and restrict it slightly to suit the constructions of [8]. An IDPF's domain is the set of bitstrings of length at most η. For each input length ℓ, the IDPF generates outputs in the group $\mathbb{G}_\ell$. We present definitions only for the case of 2 parties, since leading constructions are specialized for that case. Let η, and κ be positive integers, let M be a set, and let $\mathbb{G}_\ell$ be a group for each $\ell \in [\eta]$. An IDPF is a pair of algorithms:

- $\mathsf{IDPF.Gen}(\alpha \in \{0,1\}^\eta, \vec{\beta} \in \mathbb{G}_1 \times \cdots \times \mathbb{G}_\eta) \rightarrow (\{0,1\}^\kappa)^2 \times M$ is the **key generation** algorithm that takes a bitstring α and a vector $\vec{\beta}$ of point values, each of which is an element of the group $\mathbb{G}_\ell$ for the corresponding input length. It outputs a pair of key shares and a "public share" (an element of M).
- $\mathsf{IDPF.Eval}(id \in \{1,2\}, key \in \{0,1\}^\kappa, pub \in M, pfx \in \{0,1\}^\ell) \rightarrow \mathbb{G}_\ell$ is the **point-function evaluation** algorithm that takes in a shareholder index, an IDPF key share, a public share pub, and a prefix string of $\ell \leq \eta$ bits, then outputs a share of the IDPF output.

An IDPF is *correct* if for all $\alpha \in \{0,1\}^\eta$, all $\vec{\beta} \in \mathbb{G}_1 \times \cdots \times \mathbb{G}_\eta$, all $(key_1, key_2, pub) \in [\mathsf{IDPF.Gen}(\alpha, \vec{\beta})]$, and all strings pfx of length $\ell \leq \eta$:

$$f_{\alpha,\vec{\beta}}(pfx) = \sum_{\hat{j} \in \{1,2\}} \mathsf{IDPF.Eval}(\hat{j}, key_{\hat{j}}, pub, pfx)\,.$$

We define *privacy* for an IDPF later in Section 5.1.

3 SECURITY MODEL

3.1 Syntax

As discussed in Section 1, a VDAF can be thought of as a protocol for evaluating an aggregation function F that takes as input the vector of measurements generated by the clients and outputs an aggregate result. In addition, the function may include an auxiliary "aggregation parameter" that allows the measurements to be "refined" to contain only the information of interest to the collector. Accordingly, prior to executing the VDAF, each aggregator's state is initialized with this aggregation parameter.

Recall that execution of a VDAF proceeds in four distinct phases. (See Figure 1 for an illustration.) We formalize the computation of the parties in each phase as the component algorithms of a VDAF:

- $\mathsf{Shard}(m \in I, n \in N) \rightarrow (msg_{\text{Init}} \in M, \vec{x} \in X^s)$ is the randomized **sharding** algorithm run by the client. It takes in the client's input measurement m and a nonce n and returns

an **initial message** to be broadcasted to all aggregators and a sequence of **input shares**, one for each of the s aggregators.

- $\mathsf{Prep}(\hat{j} \in [s], sk \in SK, st \in Q, n \in N, \vec{msg} \in M^*, x \in X) \to (sts \in \{\texttt{running}, \texttt{finished}, \texttt{failed}\}, out \in (Q \times M) \cup Y \cup \{\perp\}$ is the deterministic, interactive **preparation** algorithm run by each aggregator during the online preparation process. Its inputs are the share index $\hat{j}$, the **verification key** shared by the aggregators sk, the current state st, the nonce n, the most recent round of **broadcast messages** $\vec{msg}$ (or $(msg_{\text{Init}},)$ if this is the first round), and the aggregator's input share x. The preparation algorithm returns an indication sts of whether the process is `running`, `finished`, or `failed`. When the status is `running`, the output includes the aggregator's next state and broadcast message ($(st, msg) \in Q \times M$); and when the status is `finished`, the output includes the aggregator's **refined share** ($y \in Y$).
- $\mathsf{Agg}(\vec{y} \in Y^*) \to a \in A$ is the deterministic *aggregation* algorithm run locally by each aggregator. It takes in a sequence of refined shares $\vec{y}$ and outputs an **aggregate share** a.
- $\mathsf{Unshard}(ct \in \mathbb{N}, \vec{a} \in A^s) \to r \in O$ is the deterministic *unsharding* algorithm used to compute the aggregate result r. Its inputs are the report count ct and aggregate shares $\vec{a}$.

The sets I, N, M, X, SK, Q, Y, A, and O must also be defined by the VDAF. (We typically do so only implicitly.) In addition to these sets, the VDAF specifies a set $Q_{\text{Init}} \subseteq Q$ of possible **initial states**.

Our security definitions for VDAFs require three additional syntactic properties. The first is a property we call **refinement consistency**. Intuitively, this property insists that, for a given initial state, the VDAF defines the set of refined measurements with respect to which the validity of the refined shares is to be verified. For Doplar for example (Section 5), the set of measurements are fixed-length bitstrings, while the refined measurements are one-hot vectors over a finite field. Formally, refinement consistency requires the existence of functions refine and refineFromShares such that for all m, n and $st_{\text{Init}} \in Q_{\text{Init}}$,

$$\Pr[\mathsf{refine}(st_{\text{Init}}, m) = \mathsf{refineFromShares}(st_{\text{Init}}, msg, \vec{x}) : (msg, \vec{x}) \leftarrow\!\!\$\ \mathsf{Shard}(m, n)] = 1\,.$$

Second, we require **aggregation consistency**, which means, roughly, that aggregating refined shares into aggregate shares, then unsharding, is equivalent to first unsharding the individual refined shares, then aggregating. To illustrate this idea, imagine arranging the refined shares into a matrix, where the rows correspond to aggregators and the columns to measurements. Aggregation consistency means that one can either add up the columns, then the rows, or add up the rows, then the columns. Formally, we require the existence of a function finishResult such that for all refined shares $y_1^1, \ldots, y_{ct}^1, \ldots, y_1^s, \ldots, y_{ct}^s \in Y$, it holds that

$$\begin{aligned}&\mathsf{Unshard}(ct, (\mathsf{Agg}(y_1^1, \ldots, y_{ct}^1), \ldots, \mathsf{Agg}(y_1^s, \ldots, y_{ct}^s))) = \\ &\quad \mathsf{finishResult}(ct, \mathsf{Unshard}(1, (\mathsf{Agg}(y_1^1), \ldots, \mathsf{Agg}(y_1^s))), \\ &\qquad \ldots, \mathsf{Unshard}(1, (\mathsf{Agg}(y_{ct}^1), \ldots, \mathsf{Agg}(y_{ct}^s))))\,.\end{aligned}$$

We will see that these notions of refinement and aggregation consistency, while fairly technical in nature, are trivial to show for natural constructions (including Prio3 and Doplar).

Lastly, our privacy definition allows the VDAF to be executed multiple times over the same batch of measurements, each time beginning with a new initial state. (This accounts for the iterative nature of IDPFs.) Depending on the VDAF, it may be necessary for aggregators to restrict the sequence of initial states to prevent trivial leakage. Accordingly, we require each VDAF to specify an **allowed-state** algorithm validSt that takes in the sequence of previous initial states and the next initial state and returns a bit indicating whether the next initial state is allowed.

Remark 1. *A notable feature of the VDAF syntax is the "verification key" shared by the aggregators. Looking ahead, this key is used to derive, from the nonce supplied by the client, shared randomness used for verifying refined shares. This is how the authors of the VDAF spec [8] chose to instantiate the "ideal coin-flipping functionality" used in the descriptions of protocols in the papers on which the spec is based [15, 16, 20]. As we will see in the next section, the details to how this functionality is instantiated are crucial to the privacy and robustness of VDAFs.*

3.2 Security

Two definitions are given that roughly correspond[2] to the notions of robustness and privacy from [20, Appendix A]. (A notion of *completeness* is given in the full version [23].)

Security considerations for DAP [28]. Recall from the introduction that the DAP standard being developed by the PPM working group is designed to securely execute a VDAF in a real world network. Aspects of our security model can be thought of as abstracting away the functionality provided by DAP. As such, many of our modeling decisions here amount to requirements that the DAP protocol must fulfill. We will highlight some of these considerations throughout this section.

Robustness. We say that VDAF Π is robust if, when all of the aggregators execute the protocol correctly, "valid" refined measurements are correctly aggregated, while any "invalid" measurements are filtered out by the aggregators (with high probability). This property is captured via the game $\mathsf{Exp}_{\Pi}^{\text{robust}}(A)$ defined in Figure 3. In this game the adversary, acting as a coalition of malicious clients, submits reports to the aggregators, eavesdrops on their communication, and observes the result of their computation. This functionality is modeled by the Prep oracle, which the adversary may query any number of times. It controls the nonce and initial state for each trial, but its oracle queries are subject to the restriction that, for each distinct nonce, the sequence of initial states must be valid (according to the allowed-state algorithm validSt).

Validity is defined in terms of the refinement-consistency algorithms (see Section 3.1). Let $V_{st_{\text{Init}}} = \{\mathsf{refine}_{st_{\text{Init}}}(m) : m \in I\}$ be the set of refined measurements for initial state st_{Init}. The adversary wins the robustness game if, when run on initial state st_{Init}, initial message msg_{Init}, and input shares $\vec{x}$, either: (1) an aggregator accepts a share of an invalid refined measurement, i.e., one of the aggregators ends in state `finished`, but the refined share y is not valid (i.e., not in the set $V_{st_{\text{Init}}}$, see line 15 in Figure 3); or (2)

[2] We have not attempted to work out formal relationships between our definitions and those of Corrigan-Gibbs and Boneh [20]; whether our definitions, when restricted to the same class of protocols, are stronger, weaker, or equivalent is an open question.

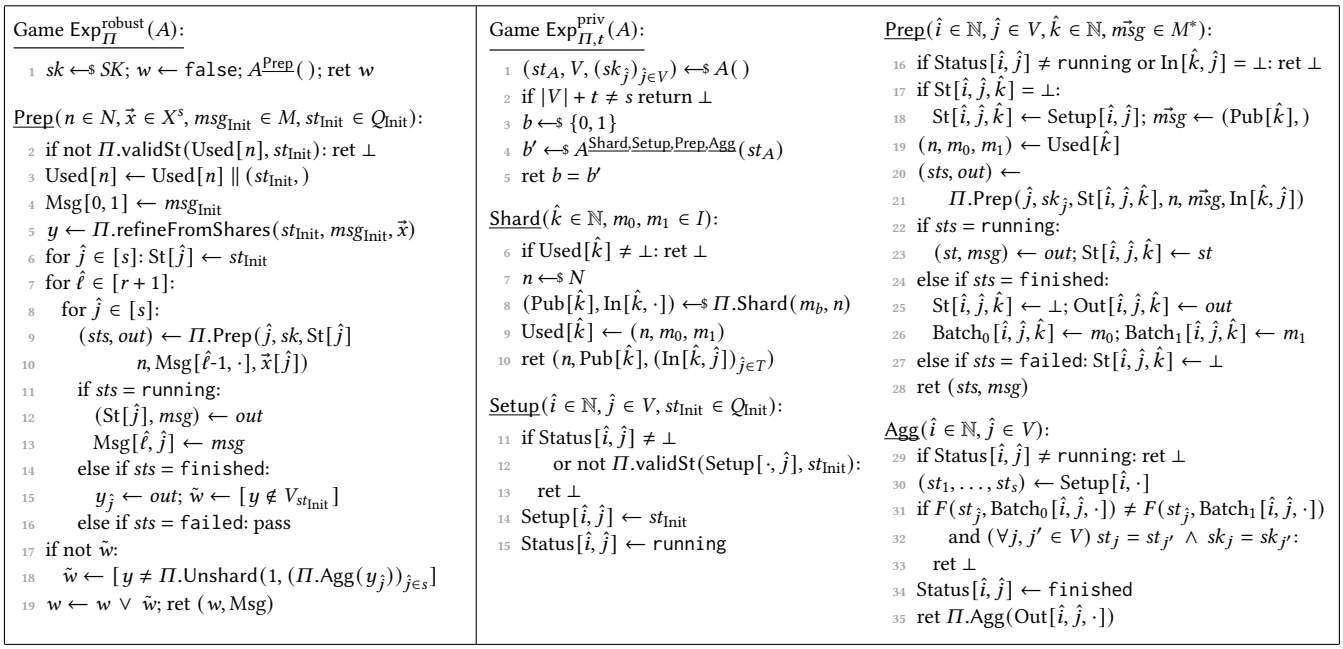

Figure 3: Games for defining robustness (left) and privacy (right) of r-round, s-party VDAF Π. Let F denote the aggregation function computed by Π. (Refer to the full version for a formal definition.) For each $st_{\mathrm{Init}} \in Q_{\mathrm{Init}}$, let $V_{st_{\mathrm{Init}}} = \{\mathrm{refine}_{st_{\mathrm{Init}}}(m) : m \in I\}$. Let $T = [s] \setminus V$. The "execution index" $\hat{i}$, "share index" $\hat{j}$, "measurement index" $\hat{k}$, and "round index" $\hat{\ell}$ correspond to, respectively, a unique initial state, an aggregator, a measurement, and a preparation round.

the refined shares computed by the aggregators do not match the expected refined measurement, i.e., unsharding the refined shares does not result in y (line 18).

Definition 3.1 (Robustness). Define the advantage of A in defeating the robustness of VDAF Π as

$$\mathrm{Adv}^{\mathrm{robust}}_{\Pi}(A) = \Pr\left[\,\mathrm{Exp}^{\mathrm{robust}}_{\Pi}(A)\,\right].$$

Informally, we say that Π is **robust** if for every efficient adversary A, the value of $\mathrm{Adv}^{\mathrm{robust}}_{\Pi}(A)$ is small.

Remark 2. *If a VDAF is robust in the sense of Definition 3.1 and aggregation-consistent, then the VDAF is also robust in the sense of [20, Definition 6]. Namely, as long as the aggregators execute the VDAF correctly, the collector is guaranteed to correctly aggregate measurements from honest clients (and reject the measurements from dishonest clients). The aggregation function that is computed is determined by the* finishResult *function implied by aggregation consistency, namely* $F(st_{Init}, m_1, \ldots, m_{ct}) = \mathsf{finishResult}(ct, (y_1, \ldots, y_{ct}))$*, where* $y_{\hat{k}}$ *is the refined measurement obtained from refining* $m_{\hat{k}}$ *with* st_{Init}*.*

Privacy. We formalize privacy via the indistinguishability game $\mathrm{Exp}^{\mathrm{priv}}_{\Pi,t}(A)$ in the right panel of Figure 3. The game is associated with VDAF Π, adversary A, and **corruption threshold** t. We consider an attacker that controls the collector and statically corrupts at most t aggregators (lines 1–2). Using its Prep oracle (lines 16–28), the adversary controls transmission of all messages in the protocol, *except* for the honestly generated input shares sent to honest (uncorrupted) aggregators. We assume that the adversary also controls setup (see the Setup oracle on lines 11–15), meaning that it can pick the verification keys for honest aggregators (1) and the initial state of each run of the preparation phase (14). This captures the real-world setting of the DAP protocol [28], where one of the aggregators (the "leader") effectively picks these values on behalf of the others (the "helpers"). Note that our game requires the secret key to be committed to prior to generating measurements: this is a deliberate restriction that was necessary to prove security of our constructions. (It is necessary for DAP to enforce this restriction.)

The initial state for each run is subject to the restriction imposed by the allowed-state algorithm defined by the VDAF (lines 11–13). (Accordingly, it is necessary for honest aggregators to enforce this restriction in the DAP protocol.)

The game asks A to distinguish execution of the protocol on two sets of measurements of its choosing. To capture this, the attacker is given an oracle Shard (lines 6–10) that models execution of the honest clients. This oracle takes in two measurements m_0, m_1 and shards m_b, where b is the challenge bit chosen at the start of the game, and returns the initial message and the input shares of the corrupted aggregators. The oracle chooses a nonce n from the nonce space N at random. (Accordingly, the DAP protocol must arrange for clients to choose their nonces at random.)

To model an attacker that controls the collector, the game allows the adversary to learn the aggregate shares computed by honest aggregators. This is captured by the Agg oracle (lines 29–35). Queries to this oracle are subject to the restriction that the aggregate share does not trivially leak the challenge bit: namely, the aggregate of

both batches of measurements specified by the adversary must be equal (31). (Tables Batch_0, Batch_1 keep track of the pairs of measurements m_0, m_1 passed to the Shard for which a given aggregator has recovered a refined share for a given initial state.) This restriction is analogous to the "leakage function" provided to the simulator in previous simulation-style definitions. See [20, Appendix A] and [16, Appendix A]. We consider something slightly stronger: if the honest aggregators disagree either on the initial state or the verification key, then we do not impose the restriction (32). This amounts to demanding that the aggregate shares leak nothing in this case.

Definition 3.2 (Privacy). Let Π be an s-party VDAF and let $t < s$ be a positive integer. Define the t-advantage of A in attacking the privacy of Π as

$$\mathsf{Adv}^{\text{priv}}_{\Pi,t}(A) = 2 \cdot \Pr\left[\, \mathsf{Exp}^{\text{priv}}_{\Pi,t}(A) \,\right] - 1\,.$$

Informally, we say that Π is t-**private** if for every efficient A the value of $\mathsf{Adv}^{\text{priv}}_{\Pi,t}(A)$ is small.

4 PRIO3

In this section we present our security analysis for Prio3, one of the candidates for standardization specified in draft-irtf-cfrg-vdaf-03 [8]. The starting point for this VDAF is an FLP system (Section 2) that defines the set of valid measurements. Drawing on techniques from Boneh et al. [15], Prio3 exploits the full-linearity property to allow the aggregators to validate the secret shared input. However, in order for the resulting VDAF to be suitable for a particular aggregation function $F : I \to O$, we need the proof system to define how measurements (I) are encoded as inputs to the prover and how refined shares are processed into the aggregate results (O).

Definition 4.1 (Affine, aggregatable encodings [20, Sec. 5.]). Let $F : I \to O$ be a function. An FLP system FLP admits an *affine, aggregatable encoding for* F if it defines the following algorithms:

- $\mathsf{FLP.Encode}(m \in I) \to inp \in \mathbb{F}^n$ is an injective map from the domain of F to the input space $\mathbb{F}^n$ of FLP.
- $\mathsf{FLP.Truncate}(inp \in \mathbb{F}^n) \to out \in \mathbb{F}^{ol}$ refines an FLP input into a format suitable for aggregation. We call ol the *output length*.
- $\mathsf{FLP.Decode}(ct \in \mathbb{N}, out \in \mathbb{F}^{ol}) \to a \in O$ converts a refined, aggregated output out to its final form a. This computation may depend on the number of measurements ct.

Correctness requires that for all $ct \geq 0$ and $\vec{m} \in I^{ct}$ it holds that

$$F(\vec{m}) = \mathsf{Decode}\Big(ct, \sum_{i \in [ct]} \mathsf{Truncate}\left(\mathsf{Encode}\left(\vec{m}[i]\right)\right)\Big)\,.$$

Let FLP be an FLP system that admits an affine, aggregatable encoding for F and let PRG be a PRG. We specify the core algorithms of Prio3[FLP, PRG] in Figure 4. (This version includes changes to draft-irtf-cfrg-vdaf-03 [8], as we discuss in the full version.) The sharding algorithm begins by encoding the measurement as prescribed by the FLP. It then splits the encoded measurement inp into shares, generates a proof of inp's validity, and splits the proof into shares as well. The joint randomness jr passed to the proof generation algorithm is derived from the input shares following the Fiat-Shamir-style transform described—but not formally analyzed—in [15, Section 6.2.3]. During preparation, the aggregators collectively re-compute jr from their input shares. Each aggregator broadcasts a share of the verifier by running the FLP query-generation algorithm on its share of the input and proof. (The query randomness qr is derived from the shared verification key sk and the nonce n provided by the environment.) The FLP decision algorithm is run on the combined verifier shares.

The aggregators must derive the joint randomness prior to computing their verifier shares. In order to allow them to perform both computations in parallel in a single round, the client sends in its initial message the sequence $\vec{rseed}$ of "joint randomness parts" consisting of the intermediate values computed by the aggregators. This allows jr to be computed immediately on receipt of the input shares. To detect if a malicious client transmitted malformed parts, the aggregators also verify the joint randomness was computed properly in the same flow.

Allowed initial states. The set of initial states for Prio3 is simply $Q_{\text{Init}} = \{\varepsilon\}$. In our security analysis, we assume honest aggregators process a batch at most once. Accordingly, the allowed-state algorithm Prio3[FLP, PRG].validSt accepts only if the batch was not aggregated previously.

Consistency. The set of refined measurements includes any output of the affine, aggregatable encoding for FLP. On input of $st_{\text{Init}} \in \{\varepsilon\}$ and $m \in I$, the refinement algorithm Prio3[FLP, PRG].refine first encodes m, then truncates and decodes it as prescribed by FLP. The refine-from-shares algorithm, Prio3[FLP, PRG].refineFromShares, unpacks each input share (see Unpack in Figure 4), extracts the shares of the FLP input, truncates them, adds them together, and decodes the result.

For aggregation consistency, we require the encoding scheme for FLP to be aggregation-consistent in a similar sense. Specifically, there must exist a function finishResult such that for all outputs $out_1, \ldots, out_{ct} \in \mathbb{F}^{ol}$ it holds that $\mathsf{Decode}(ct, \sum_{\hat{k} \in [ct]} out_{\hat{k}}) = \mathsf{finishResult}(ct, \mathsf{Decode}(1, out_1), \ldots, \mathsf{Decode}(1, out_{ct}))$.

Security. Fix $s > 2$ and let $\Pi = \text{Prio3[FLP, PRG]}$ be as specified above. Let N denote the nonce space for Π and let κ denote the seed length of PRG.

Theorem 4.2. *Modeling each* RG_i *in Figure 4 as a random oracle, if* FLP *is* ϵ*-sound (Section 2), then for every adversary* A *against the robustness of* Π *it holds that*

$$\mathsf{Adv}^{\text{robust}}_{\Pi}(A) \leq (q_{\mathsf{RG}} + q_{\mathsf{Prep}}) \cdot \epsilon + \frac{{q_{\mathsf{RG}} + q_{\mathsf{Prep}}}^2}{2^{\kappa-1}}\,,$$

where A *makes* q_{Prep} *queries to* Prep *and a total of* q_{RG} *queries to its random oracles.*

For reasonable choices of the PRG seed size, the loosest term in this bound is $(q_{\mathsf{RG}} + q_{\mathsf{Prep}}) \cdot \epsilon$. The multiplicative loss of $q_{\mathsf{RG}} + q_{\mathsf{Prep}}$ reflects the adversary's ability to partially control the randomness of the FLP insofar as it is able to use rejection sampling to obtain query and joint randomness with any property. The ϵ-soundness of FLP bounds the probability of violating soundness in a single interaction, but in a VDAF the attacker may interact with the underlying FLP once in each of its q_{Prep} queries to Prep, and it can use its queries to RG_1 to bias these interactions' joint randomness.

Algorithm $\mathsf{Shard}(m, n)$:

$inp \leftarrow \mathsf{Encode}(m)$
for $\hat{j} \in [2..s]$:
  $blind_{\hat{j}}, xseed_{\hat{j}}, pseed_{\hat{j}} \leftarrow\$ \{0,1\}^\kappa$; $\vec{x}[\hat{j}] \leftarrow \mathrm{RG}_2(xseed_{\hat{j}}, \hat{j})$
  $\vec{rseed}[\hat{j}] \leftarrow \mathrm{RG}_7(blind_{\hat{j}}, \hat{j} \parallel n \parallel \vec{x}[\hat{j}])$
$\vec{x}[1] \leftarrow inp - \sum_{\hat{j}=2}^{s} \vec{x}[\hat{j}]$
$blind_1 \leftarrow\$ \{0,1\}^\kappa$; $\vec{rseed}[1] \leftarrow \mathrm{RG}_7(blind_1, 1 \parallel n \parallel \vec{x}[1])$
$jseed \leftarrow \mathrm{RG}_6(0^\kappa, \vec{rseed})$; $jr \leftarrow \mathrm{RG}_1(jseed, \varepsilon)$
$ps \leftarrow\$ \{0,1\}^\kappa$; $pr \leftarrow \mathrm{RG}_4(ps, \varepsilon)$
$\vec{\pi}[1] \leftarrow \mathsf{Prove}(inp, jr; pr) - \sum_{\hat{j}=2}^{s} \mathrm{RG}_3(pseed_{\hat{j}}, \hat{j})$
$\vec{x}[1] \leftarrow (\vec{x}[1], \vec{\pi}[1], blind_1)$
for $\hat{j} \in [2..s]$: $\vec{x}[\hat{j}] \leftarrow (xseed_{\hat{j}}, pseed_{\hat{j}}, blind_{\hat{j}})$
ret $(\vec{rseed}, \vec{x})$

Algorithm $\mathsf{Unpack}(\hat{j}, x)$:

if $\hat{j} = 1$: $(inp, \pi, blind) \leftarrow x$
else:
  $(xseed, pseed, blind) \leftarrow x$
  $inp \leftarrow \mathrm{RG}_2(xseed, \hat{j})$; $\pi \leftarrow \mathrm{RG}_3(pseed, \hat{j})$
ret $(inp, \pi, blind)$

Algorithm $\mathsf{Prep}(\hat{j}, sk, st, n, \vec{msg}, x)$:

if $st = \varepsilon$: // Process initial message from client
  $(inp, \pi, blind) \leftarrow \mathsf{Unpack}(\hat{j}, x)$
  $(\vec{rseed},) \leftarrow \vec{msg}$; $\vec{rseed}[\hat{j}] \leftarrow \mathrm{RG}_7(blind, \hat{j} \parallel n \parallel inp)$
  $jseed \leftarrow \mathrm{RG}_6(0^\kappa, \vec{rseed})$; $jr \leftarrow \mathrm{RG}_1(jseed, \varepsilon)$; $qr \leftarrow \mathrm{RG}_5(sk, n)$
  $msg \leftarrow (\mathsf{Query}(inp, \pi, jr; qr), \vec{rseed}[\hat{j}])$
  $st \leftarrow (jseed, \mathsf{Truncate}(inp))$
  ret $(\mathtt{running}, st, msg)$
// Process broadcast messages from aggregators
$(jseed, y) \leftarrow st$; $(\vec{vfs}[\hat{j}], \vec{rseed}[\hat{j}])_{\hat{j} \in [s]} \leftarrow \vec{msg}$
if $\mathsf{Decide}(\sum_{\hat{j}=1}^{s} \vec{vfs}[\hat{j}])$ and $jseed = \mathrm{RG}_6(0^\kappa, \vec{rseed})$: ret $(\mathtt{finished}, y)$
else ret $(\mathtt{failed}, \perp)$

Algorithm $\mathsf{Agg}(\vec{y})$: ret $\sum_{i=1}^{|\vec{y}|} \vec{y}[i]$

Algorithm $\mathsf{Unshard}(ct, \vec{a})$: ret $\mathsf{Decode}(ct, \sum_{i=1}^{|\vec{a}|} \vec{a}[i])$

Algorithm $\mathrm{RG}_i(seed, cntxt)$:

$l \leftarrow (jl, n, m, pl, ql)$
if $i \le 5$: ret $\mathsf{Expand}[\mathrm{PRG}](seed, \mathtt{label}_i \parallel cntxt, \mathbb{F}.p, l[i])$
else: ret $\mathrm{PRG.Next}(\mathrm{PRG.Init}(seed, \mathtt{label}_i \parallel cntxt), \kappa)$

Figure 4: Definition of 1-round, s-party VDAF Prio3[FLP, PRG]. Let $\mathtt{label}_1, \ldots, \mathtt{label}_7$ be arbitrary, distinct bitstrings.

Remark 3. *Although we have not addressed this explicitly in our specification, the extraction step of our security reduction relies on the encoding of the context string passed to each* RG_i *being invertible. (Similarly for Theorem 5.4.)*

Theorem 4.3. *Modeling each* RG_i *in Figure 4 as a random oracle, if* FLP *is* δ*-private, then for all* $0 < t < s$ *and attackers* A *it holds that*

$$\mathrm{Adv}^{\mathrm{priv}}_{\Pi,t}(A) \le 2q_{\mathsf{Shard}} \left(\delta + \frac{q_{\mathrm{RG}} + q_{\mathsf{Shard}}}{|N|} + \frac{s \cdot q_{\mathrm{RG}}}{2^{\kappa-1}} \right),$$

where A *makes* q_{Shard} *queries to* Shard *and a total of* q_{RG} *queries to the random oracles.*

5 DOPLAR

In this section we describe and analyze Doplar, our round-reduced variant of Poplar1 [8]. Poplar1 is a candidate for standardization in draft-irtf-cfrg-vdaf-03; Doplar is introduced by our paper.

Poplar1 is designed to solve the "heavy hitters" problem (as described in Section 1) using an IDPF (Section 2) in the following way. Two aggregators hold shares of an IDPF key generated by the clients. Each evaluates its IDPF key at a number of equal-length candidate prefixes. They expect that the output is non-zero for at most one of these candidates; to verify this, they execute an MPC to determine if they hold shares of a one-hot vector, and that the non-zero value is in the desired range (i.e., equal to one or zero). If verification succeeds, then each adds its share of the vector together with the other verified shares. The result is a vector representing the number of measurements prefixed by each candidate.

The "secure sketch" MPC of Boneh et al. [16] requires two rounds of communication between the aggregators. (Computing and verifying this sketch occurs during the preparation phase of VDAF evaluation.) In this section we propose an alternative strategy that, leveraging techniques in Section 4, requires just one.

Our first step is to factor the validity check into two, parallelizable computations. The first computation is solely responsible for checking that the vector of IDPF outputs is one-hot. In Section 5.1 we extend IDPFs (Section 2) into *verifiable* IDPFs (VIDPFs), which preserve the same privacy properties as IDPFs, but additionally verify the one-hotness of the refined shares. In the full version [23] we show how to instantiate this primitive using a simple technique from de Castro and Polychroniadou [24].

The second computation checks that the *sum* of the elements of the vector is in the desired range. Our first idea is to perform this range check using an FLP (Section 2). This does not work, however, since a standard FLP requires the prover to know the statement it is proving; in our case, it does not know the value of the sum computed by the aggregators, since it does not know the candidate prefixes. To overcome this, we show how to transform an FLP into one that is *delayed input* [40]. Such a proof system allows a proof to be generated for a *set* of potential inputs such that the honest verifier accepts the proof for any input in this set, but rejects otherwise (with high probability). We define delayed-input FLP in Section 5.2 and defer the construction to the full version [23].

The result is the 1-round, 2-party VDAF presented in Section 5.3. The cost of this round reduction is a modest increase in overall communication cost and CPU time, at least for the current instantiations of the VIDPF and delayed-input FLP. We compare the cost of Doplar and Poplar1 at the end of this section.

5.1 Verifiable IDPF

A **verifiable** IDPF (VIDPF) allows the dealer to prove to the shareholders that their shares represent a one-hot vector. For our purposes, we define a **one-hot vector** as a vector that is nonzero in *at most* one component (i.e., the all-zeroes vector is also one-hot). Verifiable function secret sharings (of which VIDPF is a special

case) were previously considered in [17, 24], and a construction specifically for VIDPF was given in [24].

A VIDPF has two algorithms in addition to the usual Gen, Eval:

- $\mathsf{VIDPF.VEval}(id \in \{1,2\}, key \in \{0,1\}^\kappa, pub \in M, \vec{x} \in (\{0,1\}^\ell)^u) \to \{0,1\}^* \times (\mathbb{G}_\ell)^u$ takes as input an IDPF share (private and public parts), and a sequence of IDPF inputs. It outputs a **verification value** and a sequence of output shares.
- $\mathsf{VIDPF.Verify}(h_1, h_2) \to \{0,1\}$ takes as input two verification values and returns a boolean.

We also overload the syntax of the plaintext evaluation function to take a vector of inputs, i.e., we let

$$f_{\alpha,\vec{\beta}}(\vec{x}) = \left(f_{\alpha,\vec{\beta}}(\vec{x}[1]), f_{\alpha,\vec{\beta}}(\vec{x}[2]), \dots\right).$$

We say VIDPF is *correct* if, for all $\alpha \in \{0,1\}^\eta$, all $\vec{\beta} \in \mathbb{G}_1 \times \cdots \times \mathbb{G}_\eta$, all $\vec{x} \in (\{0,1\}^\ell)^*$, all $(key_1, key_2, pub) \in [\mathsf{Gen}(\alpha, \vec{\beta})]$, all $(h_1, \vec{y}_1) \in [\mathsf{VEval}(1, key_1, pub, \vec{x})]$, and all $(h_2, \vec{y}_2) \in [\mathsf{VEval}(2, key_2, pub, \vec{x})]$:

- $\vec{y}_1 + \vec{y}_2 = f_{\alpha,\vec{\beta}}(\vec{x})$
- If $(\vec{y}_1 + \vec{y}_2)$ is a one-hot vector then $V.\mathsf{Verify}(h_1, h_2) = 1$

Theorem 5.4 requires VIDPF to be *extractable*. Intuitively, there should be an algorithm that can extract $\alpha, \vec{\beta}$ from adversarially generated VIDPF key shares. Then VEval must produce shares consistent with the incremental point function $f_{\alpha,\vec{\beta}}$, whenever Verify succeeds. (A similar property is formalized for IDPFs by BBCG+21.) This property implies, among other things, that if Verify succeeds, then shareholders are guaranteed to hold shares of a one-hot vector. We formalize this property below.

Definition 5.1 (Extractable VIDPF (cf. [16, Definition 7])). Suppose that VIDPF is defined in terms of a random oracle with co-domain Y. Refer to the game in Figure 5 associated to VIDPF, **extractor** E, and adversary A. Define A's advantage in **fooling** E as $\mathsf{Adv}^{\mathsf{extract}}_{\mathsf{VIDPF},E}(A) = 2 \cdot \Pr\left[\, \mathsf{Exp}^{\mathsf{extract}}_{\mathsf{VIDPF},E}(A) \,\right] - 1$.

Finally, our privacy reduction for Doplar (Theorem 5.5) requires the underlying VIDPF to be *private*, in the sense that one shareholder's view—consisting of its share $key_{\hat{j}}$, the public share pub, and the other shareholder's verification value h—leaks nothing about the secrets α and β. Prior definitions of verifiable FSS—e.g., the one in de Castro and Polychroniadou [24]—only define privacy with respect to a single vector of evaluation points and verification predicate, both of which are assumed to be known at the time of share generation. In our setting, shares are generated and only later is there a choice of evaluation points and verification predicates. The same shares may be evaluated many times, on different input vectors and with different verification predicates. This leads to a more interactive, and stronger, definition than in prior works.[3]

Definition 5.2. Let $\mathsf{Exp}^{\mathsf{priv}}_{\mathsf{VIDPF},S}(A)$ be the privacy game for VIDPF, **simulator** $S = (S_1, S_2)$, and adversary A defined in Figure 5. Define the advantage of A in distinguishing S's simulation from its view of VIDPF's execution as $\mathsf{Adv}^{\mathsf{priv}}_{\mathsf{VIDPF},S}(A) = 2 \cdot \Pr[\mathsf{Exp}^{\mathsf{priv}}_{\mathsf{VIDPF},S}(A)] - 1$.

[3]The game does not need to provide an oracle for VIDPF.Verify since it is a deterministic algorithm whose inputs are known to the adversary.

Game $\mathsf{Exp}^{\mathsf{extract}}_{\mathsf{VIDPF},E}(A)$:

$b \leftarrow\$ \{0,1\}$; $(key_1, key_2, pub, st_A) \leftarrow\$ A^{\underline{\mathsf{RO}}}()$
if $b = 0$: $(\alpha, \vec{\beta}) \leftarrow\$ E(key_1, key_2, pub, \mathrm{Rand})$
$b' \leftarrow\$ A^{\underline{\mathsf{RO}},\underline{\mathsf{Eval}}}(st_A)$; ret $b = b'$

$\underline{\mathsf{Eval}}(\vec{x})$:

$(h_1, \vec{y}_1) \leftarrow\$ \mathsf{VIDPF.VEval}^{\underline{\mathsf{RO}}}(1, key_1, pub, \vec{x})$
$(h_2, \vec{y}_2) \leftarrow\$ \mathsf{VIDPF.VEval}^{\underline{\mathsf{RO}}}(2, key_2, pub, \vec{x})$
if $b = 0$ and $\mathsf{VIDPF.Verify}^{\underline{\mathsf{RO}}}(h_1, h_2) = 1$: ret $f_{\alpha,\vec{\beta}}(\vec{x})$
else: ret $\vec{y}_1 + \vec{y}_2$

$\underline{\mathsf{RO}}(inp)$:

if $\mathrm{Rand}[inp] = \perp$: $\mathrm{Rand}[inp] \leftarrow\$ Y$
ret $\mathrm{Rand}[inp]$

Game $\mathsf{Exp}^{\mathsf{priv}}_{\mathsf{VIDPF},S}(A)$:

$b \leftarrow\$ \{0,1\}$; $(st_A, \alpha, \vec{\beta}, \hat{j}) \leftarrow A()$
if $b = 0$: $(key_{\hat{j}}, pub) \leftarrow\$ S_1(\hat{j})$
else: $(key_1, key_2, pub) \leftarrow\$ \mathsf{VIDPF.Gen}(\alpha, \vec{\beta})$
$b' \leftarrow A^{\underline{\mathsf{Sketch}}}(st_A, key_{\hat{j}}, pub)$; ret $b = b'$

$\underline{\mathsf{Sketch}}(\vec{x})$:

if $b = 0$: $h \leftarrow S_2(\hat{j}, key_{\hat{j}}, pub, \vec{x})$
else: $(h, \_) \leftarrow \mathsf{VIDPF.VEval}(3 - \hat{j}, key_{3-\hat{j}}, pub, \vec{x})$
ret h

Game $\mathsf{Exp}^{\mathsf{priv}}_{\mathsf{DFLP},S}(A)$:

$b \leftarrow\$ \{0,1\}$; $(X, st_A) \leftarrow A()$
if $b = 0$: $(st_S, jr, qr) \leftarrow S_1(|X|)$
else:
  $jr \leftarrow\$ \mathbb{F}^{jl}$; $qr \leftarrow\$ \mathbb{F}^{ql}$; $\Delta \leftarrow\$ \mathbb{F}^{el}$
  $\pi \leftarrow\$ \mathsf{DFLP.Prove}(X, \Delta, jr)$
$(x, st_A) \leftarrow A(st_A, jr, qr)$; assert $x \in X$
if $b = 0$: $\sigma \leftarrow S(st_S)$
else: $\sigma \leftarrow \mathsf{DFLP.Query}(\mathsf{DFLP.Encode}(\Delta, x), \Delta, \pi, jr;\ qr)$
$b' \leftarrow A(st_A, \sigma)$; ret $b = b'$

Figure 5: Games for defining extractability and privacy for VIDPFs and privacy of delayed-input FLP.

If this privacy game withholds the Sketch oracle from the adversary (shaded in Figure 5) then we obtain the privacy game for plain IDPFs, with the adversary's advantage defined analogously.

In the full version [23] we describe a VIDPF construction that satisfies all the necessary security properties. The construction is heavily based on the verifiable DPF technique from [24].

5.2 Delayed-Input FLPs

We introduce a new variant of fully linear proofs (FLPs), in which the prover does not know in advance which instance (i.e., input) will be used during verification. Instead, the proof is generated only knowing a set of possible instances; later, the proof is verified using one of those instances. For technical reasons, the proof and verification steps operate not on the instance, but on a *randomized encoding* of the instance. This extra randomness is useful in our eventual construction (described in the full version [23]).

We adopt the terminology of **delayed-input**, which is standard in the study of (interactive) zero-knowledge protocols. In an interactive protocol with delayed input, the instance and witness need not be known/chosen until some intermediate round (often the prover's final round). In our setting, the actual choice of instance/witness is not chosen until after the prover finishes "speaking". The protocol of Lapidot and Shamir [40] is often regarded as the first ZK protocol with delayed input, while Katz and Ostrovsky [38] were the first to explicitly rely on the delayed input property while using a ZK proof in an application.

Definition 5.3. A **delayed-input FLP** DFLP consists of the following algorithms:

- $\mathsf{DFLP.Encode}(\Delta \in \mathbb{F}^{el}, x \in \mathbb{F}^{n}) \to e \in \mathbb{F}^{n'}$ takes as input encoding randomness Δ, and an input instance x. Returns an encoding of x; we let n' denote the length of the encoding. The function $\mathsf{Encode}(\Delta, \cdot)$ must be a linear function and invertible. We denote the inverse by Decode.
- $\mathsf{DFLP.Prove}(X \subseteq \mathbb{F}^{n}, \Delta \in \mathbb{F}^{el}, jr \in \mathbb{F}^{jl}) \to \pi \in \mathbb{F}^{m}$ takes as input a **set** of possible instances, encoding randomness Δ, and joint randomness jr. Produces output proof π.
- $\mathsf{DFLP.Query}(e \in \mathbb{F}^{n'}, \Delta \in \mathbb{F}^{el}, \pi \in \mathbb{F}^{m}, jr \in \mathbb{F}^{jl}; qr \in \mathbb{F}^{ql}) \to \sigma \in \mathbb{F}^{v}$ takes as input an encoded instance e, encoding randomness Δ, proof π, joint randomness jr, and query randomness qr. Returns a verifier σ. The function $\mathsf{Query}(\cdot, \cdot, \cdot, jr; qr)$ must be linear.
- $\mathsf{DFLP.Decide}(\sigma \in \mathbb{F}^{v}) \to acc \in \{0, 1\}$: Takes as input query responses σ and returns a boolean.

If Prove is restricted to sets X with $|X| = k$ then we call the construction a **delayed-k-input FLP**.

A delayed-input FLP should satisfy the following properties:

- **Completeness** (with respect to language $\mathcal{L}$): For all $X \subseteq \mathcal{L}$, all $x \in X$, and all Δ:

$$\Pr[\mathsf{Decide}(\sigma) : jr \leftarrow\!\!\$\ \mathbb{F}^{jl}; \pi \leftarrow\!\!\$\ \mathsf{Prove}(X, \Delta, jr); \sigma \leftarrow\!\!\$\ \mathsf{Query}(\mathsf{Encode}(\Delta, x), \Delta, \pi, jr)] = 1\,.$$

- **Soundness** (with respect to $\mathcal{L}$): The scheme should be sound in the usual sense of FLPs, with respect to the language $\mathcal{L}^* = \{(\mathsf{Encode}(\Delta, x), \Delta) \mid x \in \mathcal{L}\}$. In other words, it is hard for a malicious prover to generate a proof that verifies with respect to $(e, \Delta) \notin \mathcal{L}^*$.
- **Privacy:** In Figure 5 we define a game for delayed-input FLPs, in which the proof is generated using some set X of candidates, and later verified with respect to a particular $x \in X$. A delayed-input FLP is δ-private if there exists a simulator S such that every A's advantage is $\mathsf{Adv}^{\mathrm{priv}}_{\mathsf{DFLP},S}(A) \le \delta$, where

$$\mathsf{Adv}^{\mathrm{priv}}_{\mathsf{DFLP}}(A) = 2 \cdot \Pr[\mathsf{Exp}^{\mathrm{priv}}_{\mathsf{DFLP},S}(A)] - 1\,.$$

5.3 Construction

We specify our construction Doplar[VIDPF, DFLP, PRG] in Figure 6. Its three components are: a verifiable IDPF VIDPF with input length η; a delayed-2-input FLP DFLP with input set $\{0, 1\}$, proof length m, encoded input length n, encoding randomness length el, joint randomness length jl, and query randomness length ql; and a pseudorandom generator PRG (Section 2) with seed length κ. To be suitable for our construction, we must choose VIDPF and DFLP so that $\mathsf{VIDPF}.\mathbb{G}_\ell = \mathsf{DFLP}.\mathbb{F}^n$ for each $\ell \in [\eta]$.

To shard its measurement $\alpha \in \{0, 1\}^\eta$, the client begins by running the VIDPF key generator on α. The initial state for Doplar encodes the "level" ℓ at which the VIDPF shares are to be evaluated; each candidate prefix must have length ℓ. (Recall from Section 2 that (V)IDPFs can be thought of as shares of values arranged in a binary tree with nodes labeled by prefixes.) For each level of the VIDPF tree, the client generates a delayed-input proof of the refined shares' validity; just as for Prio3 (Section 4), the joint randomness used at each level is derived from the aggregator's input shares. The VIDPF output is programmed so that the sum of the output shares corresponds to an encoded input for the delayed-input FLP.

To prepare a report for aggregation, the aggregators evaluate their VIDPF key shares at the desired candidate prefixes, then interact in order to check that (1) the joint randomness was computed correctly, (2) their refined shares are one-hot, and (3) the sum of their refined shares is either one or zero.

Allowed initial states. An initial state is valid if it consists of a sequence of candidate prefixes all having the same length. Moreover, each of the prefixes must be distinct. An initial state is allowed for Doplar[VIDPF, DFLP, PRG] if the prefix length is distinct from all previous states for the same report. That is, the allowed-state algorithm validSt only permits a new state $st = (\ell, \vec{pfx})$ if ℓ is distinct for all previous states and each of the prefixes $\vec{pfx}$ is distinct.

Remark 4. *Although not addressed in Boneh et al. [16] explicitly, this restriction on the candidate prefixes is necessary for Poplar as well, as re-using the correlated randomness shared by the client would reveal information about the secret-shared vector.*

Consistency. The set of refined measurements for Doplar are one-hot vectors over the field $\mathbb{F}$ for which the non-zero element is equal to 0 or 1. For a given initial state $(\ell, \vec{pfx})$, this can be computed from the VIDPF public share and key shares by evaluating the shares on each of the prefixes $\vec{pfx}$. Since the VIDPF is a point function and the prefixes are distinct, the vector of VIDPF outputs will contain at most one nonzero entry. Aggregation consistency for Doplar is similarly straight-forward, since the refined share space and aggregate share space are the same and both aggregation and unsharding are vector summation. When we let finishResult be vector summation as well, the desired property is trivially true.

Security. Let Π = Doplar[VIDPF, DFLP, PRG] as specified above. Let N be the nonce space and let κ be the seed length for PRG.

Theorem 5.4. *Modeling each RG_i in Figure 6 as a random oracle, if* DFLP *is ϵ-sound, then for all t_A-time adversaries A and t_E-time extractors E there exists a $O(t_A + q_{\mathsf{Prep}} t_E)$-time adversary B for which*

$$\mathsf{Adv}^{\mathrm{robust}}_{\Pi}(A) \le 2(q_{\mathsf{RG}} + q_{\mathsf{Prep}}) \cdot \epsilon + \frac{(q_{\mathsf{RG}} + 3q_{\mathsf{Prep}})^2}{2^\kappa} + q_{\mathsf{Prep}} \cdot \mathsf{Adv}^{\mathrm{extract}}_{\mathsf{VIDPF},E}(B)\,,$$

where A makes q_{Prep} queries to Prep *and a total of q_{RG} queries to its random oracles.*

Algorithm Shard(α, n):

// Construct the VIDPF key shares.
$seed_1, seed_2 \leftarrow\$ \{0,1\}^\kappa$
for $\ell \in [\eta]$:
  $\vec{\Delta}[\ell] \leftarrow \mathsf{RG}_2(seed_1, n \,\|\, \ell \,\|\, 1)$
    $+ \mathsf{RG}_2(seed_2, n \,\|\, \ell \,\|\, 2)$
  $\vec{\beta}[\ell] \leftarrow \mathsf{DFLP.Encode}(\vec{\Delta}[\ell], 1)$
$(key_1, key_2, pub) \leftarrow\$ \mathsf{VIDPF.Gen}(\alpha, \vec{\beta})$
// Prepare the joint randomness parts.
$\vec{rseed}[1] \leftarrow \mathsf{RG}_5(seed_1, n \,\|\, 1 \,\|\, pub \,\|\, key_1)$
$\vec{rseed}[2] \leftarrow \mathsf{RG}_5(seed_2, n \,\|\, 2 \,\|\, pub \,\|\, key_2)$
// Generate the level proofs.
for $\ell \in [\eta]$:
  $jseed \leftarrow \mathsf{RG}_6(0^\kappa, \ell \,\|\, \vec{rseed})$
  $jr \leftarrow \mathsf{RG}_1(jseed, n \,\|\, \ell)$
  $\pi \leftarrow\$ \mathsf{DFLP.Prove}(\{0,1\}, \vec{\Delta}[\ell], jr)$
  $\vec{pf}[\ell] \leftarrow \pi - \mathsf{RG}_3(seed_2, n \,\|\, \ell)$
// Prepare the initial message and input shares.
$x_1 \leftarrow (key_1, seed_1, \vec{pf})$
$x_2 \leftarrow (key_2, seed_2)$
$msg \leftarrow (pub, \vec{rseed})$
ret (msg, x_1, x_2)

Algorithm Unpack$(\hat{j}, x, n, \ell)$:

if $\hat{j} = 1$: $(key, seed, \vec{pf}) \leftarrow x$; $\pi \leftarrow \vec{pf}[\ell]$
else: $(key, seed) \leftarrow x$; $\pi \leftarrow \mathsf{RG}_3(seed, n \,\|\, \ell)$
ret $(key, seed, \pi)$

Algorithm Prep$(\hat{j}, sk, st, n, msg, x)$:

if $st \in Q_{\mathsf{Init}}$: // Process initial message from client
  $(\ell, \vec{pfx}) \leftarrow st$; $u \leftarrow |\vec{pfx}|$
  $(pub, \vec{rseed}) \leftarrow msg$; $(key, seed, \pi) \leftarrow \mathsf{Unpack}(\hat{j}, x, n, \ell)$
  $\Delta \leftarrow \mathsf{RG}_2(seed, n \,\|\, \ell \,\|\, \hat{j})$
  $\vec{rseed}[\hat{j}] \leftarrow \mathsf{RG}_5(seed, n \,\|\, \ell \,\|\, \hat{j} \,\|\, pub \,\|\, key)$
  $jseed \leftarrow \mathsf{RG}_6(0^\kappa, \vec{rseed})$
  $jr \leftarrow \mathsf{RG}_1(jseed, n \,\|\, \ell)$; $qr \leftarrow \mathsf{RG}_4(sk, n \,\|\, \ell)$
  $(h, \vec{y}) \leftarrow \mathsf{VIDPF.VEval}(\hat{j}, pub, key, \vec{pfx})$
  $inp \leftarrow \sum_{i \in [u]} \vec{y}[i]$
  $\sigma \leftarrow \mathsf{DFLP.Query}(inp, \Delta, \pi, jr;\ qr)$
  $msg \leftarrow (\sigma, \vec{rseed}[\hat{j}], h)$; $st \leftarrow (jseed, (\mathsf{DFLP.Decode}(\vec{y}[i]))_{i \in [u]})$
  ret $(\mathtt{running}, st, msg)$
// Process broadcast messages from aggregators
$(jseed, \vec{y}) \leftarrow st$; $\left((\sigma_1, rseed_1, h_1), (\sigma_2, rseed_2, h_2)\right) \leftarrow msg$
$acc \leftarrow \mathsf{DFLP.Decide}(\sigma_1 + \sigma_2)$
if acc and $jseed = \mathsf{RG}_6(0^\kappa, (rseed_1, rseed_2))$
    and $\mathsf{VIDPF.Verify}(h_1, h_2)$: ret $(\mathtt{finished}, \vec{y})$
else: ret $(\mathtt{failed}, \perp)$

Algorithm Agg$(\vec{y})$: ret $\sum_{i=1}^{|\vec{y}|} \vec{y}[i]$

Algorithm Unshard$(_, \vec{a})$: ret $\sum_{i=1}^{|\vec{a}|} \vec{a}[i]$

Algorithm $\mathsf{RG}_i(seed, cntxt)$:

$l \leftarrow (jl, el, m, ql)$
if $i \le 4$: ret $\mathsf{Expand[PRG]}(seed, \mathtt{label}_i \,\|\, cntxt, \mathbb{F}.p, l[i])$
else: ret $\mathsf{PRG.Next}(\mathsf{PRG.Init}(seed, \mathtt{label}_i \,\|\, cntxt), \kappa)$

Figure 6: Definition of 1-round, 2-party VDAF Doplar[VIDPF, DFLP, PRG]. Let $\mathtt{label}_1, \ldots, \mathtt{label}_6$ be arbitrary, distinct bitstrings.

Theorem 5.5. *For all t_A-time adversaries A and t'-time simulators S, T there exist $O(t_A + q_{\mathsf{Shard}} t')$-time adversaries B, C for which*

$$\mathsf{Adv}^{\mathrm{priv}}_{\Pi,1}(A) \le 2q_{\mathsf{Shard}}\Big(\mathsf{Adv}^{\mathrm{priv}}_{\mathsf{VIDPF},S}(B) + \eta \cdot \mathsf{Adv}^{\mathrm{priv}}_{\mathsf{DFLP},T}(C) + \frac{\eta q_{\mathsf{RG}} + q_{\mathsf{Shard}}}{|N|} + \frac{3q_{\mathsf{RG}}}{2^{\kappa-1}}\Big),$$

where each RG_i in Figure 6 is modeled as a random oracle, adversary A makes a total of q_{RG} queries to all of its random oracles and q_{Shard} queries to Shard.

5.4 Performance Evaluation

In this section we compare the cost of Doplar to Poplar1 in terms of communication (total bits written to the wire) and computation. The parameters chosen for Poplar1 by the specification [8] match those in the performance evaluation conducted by Boneh et al. [15]. We therefore take these parameters as our basis for comparison. In the following, we have instantiated VIDPF and DFLP as described in the full version.

Boneh et al. [15] claim a per-report robustness bound of roughly $2/|\mathbb{F}|$, where $\mathbb{F}$ is the field chosen for the inner nodes.[4] They choose a 62-bit field. In order to obtain the same robustness bound, while permitting the adversary at most 2^{64} queries to its random oracles, we need to use a 128-bit field for Doplar. For both constructions, we instantiate the PRG with AES-128 as described in [8, Section 6.2] (hence the seed length is $\kappa = 128$).

Communication overhead. In Figure 7 we plot the communication cost of Doplar and Poplar1 for various choices of the input length η. We plot the total number of kilobytes sent by each client. We also plot the total number of kilobytes sent by each aggregator, per report, over all η rounds of aggregation. As one would expect, the communication cost for Doplar scales linearly with the input length. However, the client's bandwidth is about 6 times that of Poplar1; and the Aggregator's bandwidth is about 5 times.

Computational overhead. To evaluate Doplar's computational overhead, we implement a prototype[5] and benchmark it against an existing implementation of Poplar1. The ISRG (Internet Security Research Group) maintains Rust implementations of the current crop of VDAF standard candidates.[6] The code includes a work-in-progress version of Poplar1 (on a development branch, as of this writing) as well as the FLP and IDPF primitives we use in our own implementation of Doplar.

We use the Criterion framework for Rust.[7] All benchmarks reported below were run on a 2019 MacBook Pro (2.6 GHz 6-Core Intel Core i7) running rustc version 1.67.1 and cargo-criterion version

[4] Poplar1 uses a smaller field for the inner nodes of the IDPF tree than the leaf nodes.

[5] https://github.com/cloudflareresearch/doplar/tree/cjpatton/PoPETS-2023.4-Artifact
[6] Source code for the `prio` crate: https://github.com/divviup/libprio-rs
[7] Criterion: https://docs.rs/criterion/latest/criterion/

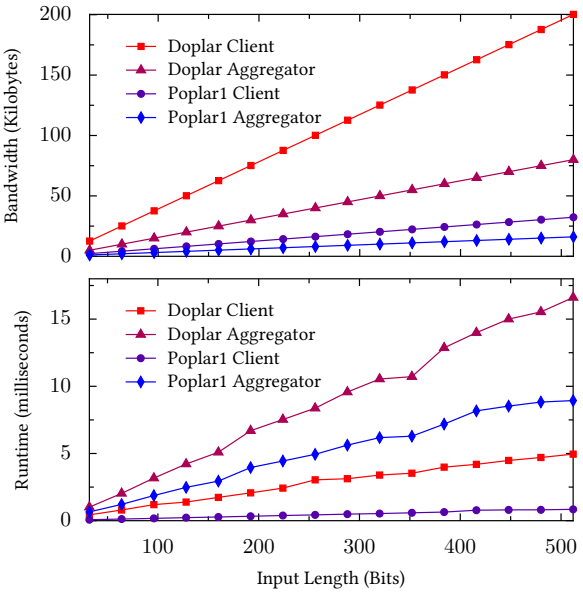

Figure 7: Bandwidth (top) and runtime (bottom) for Doplar and Poplar1.

1.1.0. The default parameters were used, except the measurement time was set to 30 seconds for all benchmarks.

Microbenchmarks for sharding. To benchmark the client, we chose a random input string of the desired length, then measured the runtime of the sharding algorithm on that input. Figure 7 shows the runtimes for lengths ranging from 32 to 512 bits. From these data we see that sharding is about 6 times as expensive for Doplar as for Poplar1. However, sharding a 512-bit input takes only 5 milliseconds, which is still quite practical. (Moreover, there is more room for optimization of our prototype.)

Microbenchmarks for preparation. Due to the highly parallelizable nature of VDAFs, much of the time the aggregators spend on executing the protocol is network-bound. However, it is useful to assess the amount of CPU time spent on processing a single report. To do so, we report microbenchmarks for per-report preparation, specifically how much time it takes an aggregator to compute its (first) broadcast message from the initial state provided by the collector and the input share provided by the client. Let us call this "preparation initialization".

One complicating factor is that the runtime of IDPF evaluation depends intrinsically on the distribution of the batch of measurements and the heavy-hitters threshold used. (We refer the reader to Algorithm 3 in Boneh et al. [16] for details.) To address this, we generated a synthetic batch of measurements and computed the prefix tree (cf. [16, Section 5.1]) for the desired threshold, then ran preparation initialization on the longest paths of this tree.

The following experiment was run 10 times. Following Boneh et al. [16], we sample random input strings from a Zipf distribution (with parameter 1.03 and support 128), then compute the prefix tree with a heavy-hitters threshold of 10. We chose a batch size of 1000. For both Doplar and Poplar, run Criterion to measure the runtime of preparation for the longest paths of the tree.

Figure 7 shows the runtime averaged over all trials for lengths ranging from 32 to 512 bits. From these data we see that preparation is only about 1.75 times as expensive for Doplar as for Poplar1. This is not surprising, given that the runtime is dominated by IDPF evaluation, which in turn depends on the number of candidates.

6 CONCLUSION AND FUTURE WORK

The PPM working group's ambition is to preserve user privacy even as software systems rely increasingly on gaining insights into user behavior. Our work aims to help ensure that this effort rests on firm formal foundations. However, we leave open a number of directions for future work. We discuss two in the remainder.

Security analysis of DAP. The definitions in this paper apply to VDAFs, which are only a component of the DAP specification [28]. Thus, our work necessarily leaves open the security of the end-to-end protocol. There are two important questions. First, DAP is designed to inherit the security properties of VDAF, i.e., one would hope that whatever can be proven about the VDAF also holds when the VDAF is instantiated in the real-world environment in which DAP runs. One way to address this is to formulate the problem in terms of *indifferentiability* [46]: if DAP's execution can be shown to be indifferentiable from the execution of the VDAF in the idealized environment described here, then any attack against DAP can be translated into an attack against the underlying VDAF.

The other important question is whether DAP meets its own security goals, which, depending on the application, might go beyond what can be achieved with a VDAF alone. Consider that whether MPC-style definitions like ours are enough for privacy depends intrinsically on the nature of the measurements being collected and how they are aggregated. It is one thing to ensure that we securely compute the aggregate; it is another to ensure that the aggregate itself does not leak "too much" information about the measurements. In particular, in many applications it will be useful to achieve differential privacy (DP) [26] in addition to secure computation. There are definitions of DP that extend to the multi-party setting [42, 49], and a number of works have considered MPC protocols for aggregation functionalities that also guarantee differential privacy of the outputs [9, 34, 48]. We hope to see future work extend this investigation to specific VDAFs.

Doplar improvements. For some applications, it would be useful for Doplar (or Poplar1) if the leaf output could be "weighted", i.e., a number in range $\{a, \ldots, b\}$ rather than $\{0, 1\}$. (Consider the ad-conversion use case from Section 1: it might be useful to know not only how many purchases were made per ad impression, but the total amount of money that was spent.) The delayed-k-input FLP paradigm may allow for this generalization, if schemes can be constructed for $k > 2$. (In this work, we only construct the delayed-2-input FLP needed for plain heavy hitters.)

There is also room for improvement of the communication cost. Despite the round reduction, the higher bandwidth may be prohibitive for some applications. However, we are optimistic that the bandwidth can be improved. Future work should focus on the delayed-2-input FLP. The current instantiation (in the full verison), while simple, effectively doubles the proof size of the base FLP.

ACKNOWLEDGMENTS

Thank you to the anonymous reviewers from the PETS 2023 program committee whose feedback helped us improve a number of technical aspects of our paper. Thank you as well to Christopher Wood who helped us position this work in the context of the ongoing standardization effort at IETF. Finally, thanks to Nikita Borisov, Sofía Celi, Tanya Verma, Tara Whalen, and Avani Wildani for editorial improvements.

Hannah and Mike carried out their work on this paper while visiting Cloudflare Research. This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

REFERENCES

[1] 2022. Privacy Preserving Measurement. https://datatracker.ietf.org/wg/ppm/about/

[2] 2022. Private Advertising Tecnology Community Group. https://www.w3.org/community/patcg/

[3] Michel Abdalla, Björn Haase, and Julia Hesse. 2021. Security Analysis of CPace. Cryptology ePrint Archive, Paper 2021/114. https://eprint.iacr.org/2021/114

[4] Michel Abdalla, Björn Haase, and Julia Hesse. 2022. *CPace, a balanced composable PAKE.* Internet-Draft draft-irtf-cfrg-cpace-06. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-irtf-cfrg-cpace/06/ Work in Progress.

[5] Surya Addanki, Kevin Garbe, Eli Jaffe, Rafail Ostrovsky, and Antigoni Polychroniadou. 2021. Prio+: Privacy Preserving Aggregate Statistics via Boolean Shares. Cryptology ePrint Archive, Report 2021/576. https://ia.cr/2021/576.

[6] Erik Anderson, Melissa Chase, F. Betul Durak, Esha Ghosh, Kim Laine, and Chenkai Weng. 2021. Aggregate Measurement via Oblivious Shuffling. Cryptology ePrint Archive, Report 2021/1490. https://ia.cr/2021/1490.

[7] Apple and Google. 2021. Exposure Notification Privacy-preserving Analytics (ENPA). White paper. https://covid19-static.cdn-apple.com/applications/covid19/current/static/contact-tracing/pdf/ENPA_White_Paper.pdf.

[8] Richard Barnes, Christopher Patton, and Phillipp Schoppmann. 2022. *Verifiable Distributed Aggregation Functions.* Internet-Draft draft-irtf-cfrg-vdaf-03. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-irtf-cfrg-vdaf/03/ Work in Progress.

[9] James Bell, Adrià Gascón, Badih Ghazi, Ravi Kumar, Pasin Manurangsi, Mariana Raykova, and Phillipp Schoppmann. 2022. Distributed, Private, Sparse Histograms in the Two-Server Model. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 307–321.

[10] James Bell, Adrià Gascón, Tancrède Lepoint, Baiyu Li, Sarah Meiklejohn, Mariana Raykova, and Cathie Yun. 2022. ACORN: Input Validation for Secure Aggregation. *Cryptology ePrint Archive* (2022). https://eprint.iacr.org/2022/1461

[11] James Henry Bell, Kallista A Bonawitz, Adrià Gascón, Tancrède Lepoint, and Mariana Raykova. 2020. Secure single-server aggregation with (poly) logarithmic overhead. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1253–1269.

[12] Mihir Bellare and Phillip Rogaway. 1993. Random Oracles Are Practical: A Paradigm for Designing Efficient Protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security* (Fairfax, Virginia, USA) *(CCS '93)*. ACM, New York, NY, USA, 62–73.

[13] Mihir Bellare and Phillip Rogaway. 2006. The Security of Triple Encryption and a Framework for Code-Based Game-Playing Proofs. In *Advances in Cryptology - EUROCRYPT 2006*, Serge Vaudenay (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 409–426.

[14] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. 2017. Practical secure aggregation for privacy-preserving machine learning. In *proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1175–1191.

[15] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. 2019. Zero-Knowledge Proofs on Secret-Shared Data via Fully Linear PCPs. In *Advances in Cryptology – CRYPTO 2019*, Alexandra Boldyreva and Daniele Micciancio (Eds.). Springer International Publishing, Cham, 67–97.

[16] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. 2021. Lightweight Techniques for Private Heavy Hitters. In *IEEE Symposium on Security and Privacy*. IEEE, 762–776.

[17] Elette Boyle, Niv Gilboa, and Yuval Ishai. 2016. Function secret sharing: Improvements and extensions. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. 1292–1303.

[18] Justin Brickell and Vitaly Shmatikov. 2006. Efficient anonymity-preserving data collection. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. 76–85.

[19] Ran Canetti. 2020. Universally Composable Security. *J. ACM* 67, 5, Article 28 (sep 2020), 94 pages. https://doi.org/10.1145/3402457

[20] Henry Corrigan-Gibbs and Dan Boneh. 2017. Prio: Private, Robust, and Scalable Computation of Aggregate Statistics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 259–282. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/corrigan-gibbs

[21] George Danezis, Cédric Fournet, Markulf Kohlweiss, and Santiago Zanella-Béguelin. 2013. Smart meter aggregation via secret-sharing. In *Proceedings of the first ACM workshop on Smart energy grid security*. 75–80.

[22] Alex Davidson, Peter Snyder, EB Quirk, Joseph Genereux, Benjamin Livshits, and Hamed Haddadi. 2022. STAR: Secret Sharing for Private Threshold Aggregation Reporting. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 697–710.

[23] Hannah Davis, Christopher Patton, Mike Rosulek, and Phillipp Schoppmann. 2023. Verifiable Distributed Aggregation Functions. Cryptology ePrint Archive, Paper 2023/130. https://eprint.iacr.org/2023/130

[24] Leo de Castro and Antigoni Polychroniadou. 2022. Lightweight, Maliciously Secure Verifiable Function Secret Sharing. In *Advances in Cryptology - EUROCRYPT 2022 - 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30 - June 3, 2022, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 13275)*, Orr Dunkelman and Stefan Dziembowski (Eds.). Springer, 150–179. https://doi.org/10.1007/978-3-031-06944-4_6

[25] Yitao Duan, John Canny, and Justin Zhan. 2010. {P4P}: Practical {Large-Scale}{Privacy-Preserving} Distributed Computation Robust against Malicious Users. In *19th USENIX Security Symposium (USENIX Security 10)*.

[26] Cynthia Dwork. 2006. Differential Privacy. In *Automata, Languages and Programming*, Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–12.

[27] Tariq Elahi, George Danezis, and Ian Goldberg. 2014. Privex: Private collection of traffic statistics for anonymous communication networks. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 1068–1079.

[28] Tim Geoghegan, Christopher Patton, Eric Rescorla, and Christopher A. Wood. 2022. *Distributed Aggregation Protocol for Privacy Preserving Measurement.* Internet-Draft draft-ietf-ppm-dap-02. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-ietf-ppm-dap/02/ Work in Progress.

[29] Niv Gilboa and Yuval Ishai. 2014. Distributed Point Functions and Their Applications. In *Advances in Cryptology – EUROCRYPT 2014*, Phong Q. Nguyen and Elisabeth Oswald (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 640–658.

[30] Matthew Green, Watson Ladd, and Ian Miers. 2016. A Protocol for Privately Reporting Ad Impressions at Scale. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria). Association for Computing Machinery, New York, NY, USA, 1591–1601. https://doi.org/10.1145/2976749.2978407

[31] Chun Guo, Jonathan Katz, Xiao Wang, and Yu Yu. 2020. Efficient and secure multiparty computation from fixed-key block ciphers. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 825–841.

[32] Xiaojie Guo, Kang Yang, Xiao Wang, Wenhao Zhang, Xiang Xie, Jiang Zhang, and Zheli Liu. 2022. Half-Tree: Halving the Cost of Tree Expansion in COT and DPF. Cryptology ePrint Archive, Paper 2022/1431. https://eprint.iacr.org/2022/1431

[33] Susan Hohenberger, Steven Myers, Rafael Pass, et al. 2014. ANONIZE: A large-scale anonymous survey system. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 375–389.

[34] Thomas Humphries, Rasoul Akhavan Mahdavi, Shannon Veitch, and Florian Kerschbaum. 2022. Selective MPC: Distributed Computation of Differentially Private Key-Value Statistics. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1459–1472.

[35] Joseph J. Pfeiffer III, Denis Charles, Davis Gilton, Young Hun Jung, Mehul Parsana, and Erik Anderson. 2021. Masked LARk: Masked Learning, Aggregation and Reporting worKflow. arXiv:2110.14794 [cs.CR]

[36] Pranav Jangir, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, and Somya Sangal. 2022. Vogue: Faster Computation of Private Heavy Hitters. Cryptology ePrint Archive, Paper 2022/1561. https://eprint.iacr.org/2022/1561

[37] Marek Jawurek and Florian Kerschbaum. 2012. Fault-tolerant privacy-preserving statistics. In *International Symposium on Privacy Enhancing Technologies Symposium*. Springer, 221–238.

[38] Jonathan Katz and Rafail Ostrovsky. 2004. Round-optimal secure two-party computation. In *Annual International Cryptology Conference*. Springer, 335–354.

[39] Klaus Kursawe, George Danezis, and Markulf Kohlweiss. 2011. Privacy-friendly aggregation for the smart-grid. In *International Symposium on Privacy Enhancing Technologies Symposium*. Springer, 175–191.

[40] Dror Lapidot and Adi Shamir. 1990. Publicly Verifiable Non-Interactive Zero-Knowledge Proofs. In *Advances in Cryptology - CRYPTO '90, 10th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1990, Proceedings (Lecture Notes in Computer Science, Vol. 537)*, Alfred Menezes and Scott A. Vanstone (Eds.). Springer, 353–365. https://doi.org/10.1007/3-540-38424-

3_26
[41] Luca Melis, George Danezis, and Emiliano De Cristofaro. 2015. Efficient private statistics with succinct sketches. *arXiv preprint arXiv:1508.06110* (2015).
[42] Ilya Mironov, Omkant Pandey, Omer Reingold, and Salil Vadhan. 2009. Computational differential privacy. In *Annual International Cryptology Conference*. Springer, 126–142.
[43] Daniele Molteni. 2022. Improving the WAF with Machine Learning. Cloudflare blog. https://blog.cloudflare.com/waf-ml/
[44] Dimitris Mouris, Pratik Sarkar, and Nektarios Georgios Tsoutsos. 2023. PLASMA: Private, Lightweight Aggregated Statistics against Malicious Adversaries with Full Security. Cryptology ePrint Archive, Paper 2023/080. https://eprint.iacr.org/2023/080 https://eprint.iacr.org/2023/080.
[45] Mozilla. 2022. Origin Telemetry. https://firefox-source-docs.mozilla.org/toolkit/components/telemetry/collection/origin.html
[46] Christopher Patton and Thomas Shrimpton. 2020. Quantifying the Security Cost of Migrating Protocols to Practice. In *Advances in Cryptology – CRYPTO 2020*, Daniele Micciancio and Thomas Ristenpart (Eds.). Springer International Publishing, Cham, 94–124.
[47] Raluca Ada Popa, Andrew J Blumberg, Hari Balakrishnan, and Frank H Li. 2011. Privacy and accountability for location-based aggregate statistics. In *Proceedings of the 18th ACM conference on Computer and communications security*. 653–666.
[48] Edo Roth, Daniel Noble, Brett Hemenway Falk, and Andreas Haeberlen. 2019. Honeycrisp: Large-Scale Differentially Private Aggregation without a Trusted Core. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 196–210. https://doi.org/10.1145/3341301.3359660
[49] Phillipp Schoppmann, Lennart Vogelsang, Adrià Gascón, and Borja Balle. 2020. Secure and Scalable Document Similarity on Distributed Databases: Differential Privacy to the Rescue. *Proceedings on Privacy Enhancing Technologies* 2 (2020), 209–229.
[50] Erik Taubeneck, Martin Thomson, Benjamin Savage, Benjamin Case, Daniel Masny, and Richa Jain. 2022. IPA End to End Protocol. Proposal submitted to the PATCG working group of the W3. https://github.com/patcg-individual-drafts/ipa/blob/main/IPA-End-to-End.md