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 candidate protocols. The first is based on the Prio system of Corrigan-Gibbs and Boneh (NSDI 2017). Prio is fairly mature and has been deployed in real-world applications. We prove that, with only minor changes, the current draft of the standardized version achieves our security goals. The second candidate is the recently proposed Poplar system from Boneh et al. (IEEE S&P 2021). The deployability of Poplar is less certain. One difficulty is that the interactive step requires two rounds of broadcast messages, whereas Prio requires just one. This makes Poplar less suitable for many deployment scenarios. We show the round complexity can be improved, at the cost of higher bandwidth.

Change Log
• 2023/02/22: Fix a minor bug in the Doplar spec (Section 5). Previously we had included the IDPF tree level in the derivation of the joint randomness parts; but these are shared across all levels. Remove the level from the joint randomness parts binder; add the level to the joint randomness seed binder; and propagate these changes through the proofs of robustness and privacy. (No changes to the reductions or bounds were required.)
• 2023/02/03: Initial ePrint submission.

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 [47]. But logging these events and aggregating them in the clear risks exposing browser history.
2. *Measuring ad conversion rates.* Today advertising is an important 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 [45].

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 [31, 23, 17, 18, 5, 10]. Notable deployments include Mozilla’s Origin Telemetry project [47] 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 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 [23, 17, 18, 5] that conform to the operational and security considerations identified by 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 protocol due to Corrigan-Gibbs and Boneh [23]. We show that this protocol meets our security goals with only minor changes. Another candidate for standardization is the more recent Poplar protocol due to Boneh et al. [18]; we introduce and analyze a variant of this protocol that has improved round complexity.

**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 [29] centers around the execution of a particular class of MPC protocols, called *Verifiable Distributed Aggregation Functions (VDAFs)* [9]. 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 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 shares are “well-formed”, e.g., 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 $F(m_1, \ldots, m_{ct})$.

The key feature that makes VDAFs well-suited for Internet scale is that they are “streamable”, in the sense that the interactive part of the computation is fully parallelizable across all reports being aggregated. This
also means that VDAFs can only be used for aggregation functions $F$ that can be decomposed into $f, g$ for which $F(m_1, \ldots, m_	ext{ct}) = f(g(m_1), \ldots, g(m_	ext{ct}))$, where $g$ is the online computation (Shard/Prepare) and $f$ is the offline computation (Aggregate/Unshard). Indeed, the goal is not to encompass all possible MPC schemes, but a particular, useful class of them. VDAFs can be used for a variety of aggregation tasks, including: Simple summary statistics, such as sum, mean, or standard deviation, quantile estimates, linear regression [23]; a step of a gradient descent [36]; and heavy hitters (see below).

Security goals. The primary goal for VDAFs is that they are private in the sense that the attacker learns nothing about the measurements $m_1, \ldots, m_	ext{ct}$ beyond what it can infer from the aggregate result $F(m_1, \ldots, m_	ext{ct})$. We consider 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” in that 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 main consideration is 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 clearly corrupt the result by sending the collector a malformed aggregate share; there may be practical ways to prevent this, but doing so is out-of-scope for this work.)

We formalize these security notions in the game-playing paradigm [14]. First, in Section 3.2 we define privacy via an indistinguishability game $\text{Exp}_{\text{priv}}^{\Pi}(A)$ played by an attacker $A$ against VDAF $\Pi$. 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.

We formalize robustness via a game $\text{Exp}_{\text{robust}}^{\Pi}(A)$ (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.

Note on the UC paradigm. An alternative approach, and one that is more conventional for MPC, is to formulate security in the Universal Composability (UC) framework [21]. 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 can be easily misinterpreted when...
translated to practice.\footnote{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]).} One goal of our definitions is to make as explicit as possible all of the requirements an application like DAP [29] needs to meet in order to use VDAFs securely.

**Constructions.** The starting point for our work is draft-irtf-cfrg-vdaf-03 [9], the current draft of the VDAF specification at the time of writing. This document describes two VDAFs.

The first scheme, called Prio3, is based on Prio [23], but incorporates performance improvements from Boneh et al. [17] (hereafter BBCG+19). Prio3 can be used to compute a wide variety of aggregation functions due to its use of a Fully Linear Proof (FLP). Briefly, an FLP is a special class 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 [17, Theorem 4.3]) and adopted by the draft (with minor modifications; see [9, 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 1) and private (Theorem 2) 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 result in loose security bounds, which we address here.

The second scheme in the VDAF specification is called Poplar1 and is based on the recent Poplar protocol from Boneh et al. [18] (BBCG+21). Poplar is designed to solve the private “heavy hitters” problem in which each client submits an arbitrary bitstring $a$ 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) [30], 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 $a$, 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 that BBCG+21 call 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 $a$. 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 per-report latency 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 [22], filling some minor gaps in their analysis along the way. 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 [24, 26, 28, 39, 41, 43], or on anonymization networks [49, 34, 20]. 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. [16] and subsequently improved by Bell et al. [12, 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 “streamable” MPC architecture of Prio and Poplar in which the interactive portion of the computation 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 [36] is a proposal by Microsoft for training machine learning models on private data, using secret-sharing and MPC between a set of aggregators. AdScale [31] 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. [10] 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 [38] 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 [25] 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. [46] 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 resemblance 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 an exploration of how to combine our results to future work.

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 $\bot$. Let $y \leftarrow S$ denote sampling $y$ uniformly from a finite set $S$; let $y \leftarrow A(x)$ 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 $T[K_1,\ldots]$ to denote the value corresponding to key $K_1,\ldots$. We sometimes write a dot “.” in place of one of the elements of the key, e.g., “$T[K_1,]$” instead of “$T[K_1, K_2]$”. We use this notation to denote the vector of values in the table that match the key pattern. For example, we write $T[K_1,\cdot]$ for the vector $(T[K_1, K_1^1], \ldots, T[K_1, K_1^m])$ where $(K_1, K_1^1), \ldots, (K_1, K_1^m)$ 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 [9, 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:

- $\text{PRG.init}(seed \in \{0, 1\}^*, \text{context} \in \{0, 1\}^*) \rightarrow st \in Q$ takes a seed and context string to the initial PRG state. We call $\kappa$ the *seed length* and $Q$ the *state space*. 

We require the field $F$ outputs 1 whenever the input is in the input.\) work on “linear PCPs” \[8, 15, 37\] in which the verifier has linear access to the proof, but arbitrary access to performed on secret-shared data, leveraging its additive homomorphism property. (This is contrast to prior proof involves computing a strictly linear function over both the input and proof. Hence, verification can be validity condition to be expressed in terms of an arithmetic circuit evaluated over the input, similar to knowledge, that a secret-shared value has some property required by the application, e.g., the input is a formulation differs slightly, as we discuss below.) FLPs allow a prover to prove to a verifier, in zero-

Fully Linear Proof Systems. We recall the definition of FLP systems from BBCG+19 \[17\]. (Our definition of FLP with finite field $F$. We define soundness of $\text{FLP}$ in terms of experiment $\text{Err}_{\text{FLP}}(P^*)$ shown in Figure 2 associated with a language $L \subseteq F^n$. We say that $\text{FLP}$ is complete for $L$ if the proof system outputs 1 whenever the input is in $L$. That is, for all $x \in L$ it holds that

$$\Pr \left[ \text{Decide}(\sigma) : jr \leftarrow F^\ell; \pi \leftarrow \text{Prove}(x, jr); \sigma \leftarrow \text{Query}(x, \pi, jr) \right] = 1.$$ We define soundness of FLP in terms of experiment $\text{Err}_{\text{FLP}}(P^*)$ shown in Figure 2 associated with a malicious prover $P^*$. In this experiment, the prover commits to an invalid input $x \in F^n \setminus L$. Next, joint randomness $jr$ is generated and given to $P^*$, who then generates a proof $\pi$. Finally, the verifier is run on $x, \pi, jr$: the malicious prover “wins” if the verifier deems the input valid. We say FLP is $\epsilon$-sound for $L$ if for all $P^*$ it holds that $\Pr \left[ \text{Err}_{\text{FLP}}(P^*) \right] \leq \epsilon$.

Let $\text{View}_{\text{FLP}}(x)$ denote the procedure defined in Figure 2. We say FLP is $\delta$-statistical, strong, honest-verifier zero-knowledge—or, simply, $\delta$-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 $x \in L$ it holds that

$$\sum_\omega \left| \Pr \left[ \text{View}_{\text{FLP}}(x) = \omega \right] - \Pr \left[ S(\cdot) = \omega \right] \right| \leq \delta.$$
Comparison to Boneh et al. [17]. Our syntax diverges slightly from BBCG+19 in two main respects. First, we have tailored the syntax to 1.5-round, public-coin IOP systems (cf. [17, Section 3.2]), as this is the only type of system considered in the VDAF specification [9]. Following the spec, we refer to the “random challenge” as the “joint randomness”, as this allows us to more easily distinguish the challenge from the randomness consumed locally by the prover and verifier. Second, following the VDAF specification [9], we have adapted the syntax so that it describes explicitly the computations of the prover and verifier. Namely, our query-generation algorithm takes in the input and proof and outputs the verifier string consumed by the decision algorithm, whereas in BBCG+19, the query-generation algorithm outputs a description of the linear function used to compute the verifier string.

Our notion of FLP soundness differs slightly from BBCG+19 in that it explicitly requires the prover to “commit” to the invalid prior to the joint randomness being generated. This clarifies that the joint randomness needs to be independent of the input in order for soundness to be achievable.

Incremental Distributed Point Functions. A point function is a function that is zero everywhere except on a special input \( \alpha \); an incremental point function is a function that is zero everywhere except on a special input \( \alpha \) and all of its prefixes. Intuitively, 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 \( p \parallel 0 \) and \( p \parallel 1 \). Each node on the path to the leaf node \( \alpha \) is assigned the value 1, and all other nodes are assigned the value 0. (See [18, Figure 4] for an illustration.)

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

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

An Incremental Distributed Point Function (IDPF) [18] is a concise secret sharing of an incremental point function. We recall the definition of an IDPF from Boneh et al. [18] and restrict it slightly to suit the constructions of [9]. An IDPF’s domain is the set of bitstrings of length at most \( \eta \). For each input length \( \ell \), the IDPF generates outputs in the group \( G_\ell \). We present definitions only for the case of 2 parties, since leading constructions are specialized for that case. Let \( \eta \) and \( \kappa \) be positive integers, let \( M \) be a set, and let \( G_\ell \) be a group for each \( \ell \in [\eta] \). An IDPF is a pair of algorithms:

- *IDPF.Gen*(\( \alpha \in \{0, 1\}^\eta, \vec{\beta} \in G_1 \times \cdots \times G_\eta \) \( \rightarrow \) \( (\{0, 1\}^\kappa)^2 \times M \)) is the key generation algorithm that takes a bitstring \( \alpha \) and a vector \( \vec{\beta} \) of point values, each of which is an element of the group \( G_\ell \) for the corresponding input length. It outputs a pair of key shares and a “public share” (an element of \( M \)).

- *IDPF.Eval*(\( id \in \{1, 2\}, key \in \{0, 1\}^\kappa, pub \in M, pfx \in \{0, 1\}^\ell \) \( \rightarrow \) \( G_\ell \)) 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 \( \vec{\beta} \in G_1 \times \cdots \times G_\eta \), all \( (key_1, key_2, pub) \in [IDPF.Gen(\alpha, \vec{\beta})] \), and all strings \( pfx \) of length \( \ell \leq \eta \):

\[
f_{\alpha, \vec{\beta}}(pfx) = \sum_{j \in \{1, 2\}} IDPF.Eval(j, key_j, pub, pfx).
\]

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 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:

- **Shard** \((m \in I, n \in N) \to (msg_{\text{Init}} \in M, \bar{x} \in X^*)\) 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\(^2\) to be broadcasted to all aggregators and a sequence of input shares, one for each of the \( s \) aggregators.

- **Prep** \((\bar{j} \in [s], sk \in SK, st \in Q, n \in N, m\bar{sg} \in M^*, x \in X) \to (sts \in \{\text{running, finished, failed}\}, out \in (Q \times M) \cup Y \cup \{\perp\})\) is the deterministic, interactive preparation algorithm run by each aggregator during the online preparation process. Its inputs are the share index \( \bar{j} \), the verification key shared by the aggregators \( sk \), the current state \( st \), the nonce \( (n \in N) \), the most recent round of broadcast messages \( m\bar{sg} \) (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 an aggregator’s refined share \((y \in Y)\).

- **Agg** \((\bar{y} \in Y^*) \to a \in A\) is the deterministic aggregation algorithm run locally by each aggregator. It takes in a sequence of refined shares \( \bar{y} \) and outputs an aggregate share \( a \).

- **Unshard** \((ct \in N, \bar{a} \in A^*) \to r \in O\) is the deterministic unsharding algorithm used to compute the aggregate result \( r \). Its inputs are the report count \( ct \) and aggregate shares \( \bar{a} \).

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 definitions of robustness and privacy for VDAFs (defined in the next section) 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 example, in the VDAF of 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 \( \text{refine} \) and \( \text{refineFromShares} \) such that for all \( m, n \) and \( st_{\text{Init}} \in Q_{\text{Init}}\),

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

Second, we require aggregation consistency, which means, roughly, that aggregating refined shares into aggregate shares, then unsharding, is equivalent to first unsharding the 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 \( \text{finishResult} \) such that for all refined shares \( y_1^1, \ldots, y_t^1, \ldots, y_1^t, \ldots, y_t^T \in Y \), it holds that

\[
\text{Unshard}(ct, (\text{Agg}(y_1^1), \ldots, y_t^1), \ldots, \text{Agg}(y_1^T), \ldots, y_t^T)) = \\
\text{finishResult}(ct, \text{Unshard}(1, (\text{Agg}(y_1^1), \ldots, y_t^1)), \\
\ldots, \text{Unshard}(1, (\text{Agg}(y_1^T), \ldots, y_t^T))).
\]

\(^2\)This message is called the “public share” in the specification.
We will see that these notions of refinement and aggregation consistency, while fairly technical in nature, are trivial to show for natural constructions (including Prió3 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.

### 3.2 Security

Three definitions are given for VDAFs. The first, completeness, is used to specify correct evaluation of an aggregation function. The others, robustness and privacy, roughly correspond to notions of the same names from [23, Appendix A].

**Security considerations for DAP** [29]. Recall from introduction that the DAP standard is designed to securely execute a VDAF in a real world network. Aspects of our security model can be thought of 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.

**Completeness.** We require that, when executed honestly, the VDAF evaluates its aggregation function \( F \) correctly. We formalize non-adversarial execution of \( \Pi \) via procedure Run\( \Pi \) in Figure 3. Along with the VDAF \( \Pi \), this procedure is parameterized by an initial state \( stInit \) with which to configure the aggregators and a sequence of measurements and nonces to process into an aggregate result.

Algorithm Run processes the measurements as illustrated in Figure 1. First, each measurement is sharded into input shares by the submitting client (line 4), then refined into a set of refined shares by the aggregators (5–16). Next, the refined shares recovered by each aggregator are combined into an aggregate share (18). Finally, the aggregate shares are combined by the collector into the aggregate result (19).

**Definition 1** (Completeness). Let \( F : QInit \times I^* \rightarrow \mathcal{O} \) be a function. We say that VDAF \( \Pi \) is complete for \( F \) if for all \( \vec{m} \in I^* \) and \( \vec{n} \in N^* \) for which \( |\vec{m}| = |\vec{n}| \) and \( stInit \in QInit \) it holds that

\[
\Pr[\text{Run}_\Pi(stInit, \vec{m}, \vec{n}) = F(stInit, \vec{m})] = 1,
\]

where the probability is over the randomness of Run and its subroutines. We say that \( \Pi \) is complete if it is complete for some function \( F \).

**Robustness.** We say that VDAF \( \Pi \) is robust if, when all of the aggregators execute the protocol correctly, all “valid” refined measurements will be aggregated, while any “invalid” measurement will be filtered out by the aggregators (with high probability). This property is captured via the game 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 \( V_{stInit} = \{\text{refine}_{stInit}(m) : m \in I\} \) be the set of refined measurements for initial state \( stInit \). The adversary wins the robustness game if, when run on initial state \( stInit \), initial message \( msgInit \), 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 \( V_{stInit} \), see see line 15 in Figure 3); or (2) 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 2** (Robustness). Define the advantage of \( A \) in defeating the robustness of VDAF \( \Pi \) as

\[
\text{Adv}^{\Pi}_{\text{robust}}(A) = \Pr[\text{Exp}^{\Pi_{\text{robust}}}(A)].
\]

Informally, we say that \( \Pi \) is robust if for every efficient adversary \( A \), the value of \( \text{Adv}^{\Pi}_{\text{robust}}(A) \) is small.
Figure 3: Left: Procedure for defining completeness of \( \Pi \) in the sense of Definition 1. Right: Game for defining robustness of \( \Pi \). Let \( Q_{\text{init}} \subseteq Q \) denote the set of valid initial states and, for each \( \text{st}_{\text{init}} \in Q_{\text{init}} \), let \( \mathcal{V}_{\text{st}_{\text{init}}} = \{ \text{refine}_{\text{st}_{\text{init}}}(m) \mid m \in \mathcal{Z} \} \).

Remark 1. If a VDAF is robust in the sense of Definition 2 and aggregation-consistent, then the VDAF is also robust in the sense of [23, 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 \( \text{finishResult} \) function implied by aggregation consistency, namely \( F(\text{st}_{\text{init}}, m_1, \ldots, m_c) = \text{finishResult}((\Pi.\text{Agg}(y_j))_{j \in \mathcal{Z}}) \), where \( y_j \) is the refined measurement obtained from refining \( m_j \) with \( \text{st}_{\text{init}} \).

Note that ensuring all of the honest measurements are aggregated additionally requires the scheme to be complete in the sense of Definition 1.

Privacy. We formalize privacy via the indistinguishability game \( \text{Exp}^{\text{priv}}_{\Pi,t}(A) \) in the right panel of Figure 4. The game is associated with VDAF \( \Pi \), 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 \( \text{Prep} \) 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 \( \text{Setup} \) 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 [29], 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 \( \text{Shard} \) (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
For clients to choose their nonces at random.)

\[ \Pi \] denote the aggregation function for which \( \Pi \) is complete and let \( Q_{\text{init}} \) its set of initial states. Let \( T = [s] \setminus V \).

\begin{align*}
\text{Game } \text{Exp}_{\Pi,N}^{\text{priv}}(A): \\
1 & (st_A, V_i = (sk_j)_{j \in V} \leftarrow A() \\
2 & \text{if } |V| + t \neq s \text{ return } \bot \\
3 & b \leftarrow \{0, 1\} \\
4 & b' \leftarrow \text{Shard.Setup}(A, st_A) \\
5 & \text{ret } b = b' \\
\end{align*}

\begin{align*}
\text{Shard}(\hat{k} \in \mathbb{N}, m_0, m_1 \in T): \\
0 & \text{if } \text{Used}[\hat{k}] \neq \bot \text{ ret } \bot \\
1 & n \leftarrow \mathbb{N} \\
2 & (\text{Pub}[\hat{k}], \text{In} \hat{k}) \leftarrow \Pi.\text{Shard}(m_0, n) \\
3 & \text{Used}[\hat{k}] \leftarrow (n, m_0, m_1) \\
4 & (n, \text{Pub}[\hat{k}], (\text{In} \hat{k}, j))_{j \in T} \\
\end{align*}

\begin{align*}
\text{Setup}(\hat{i} \in \mathbb{N}, j \in V, st_{\text{init}} \in Q_{\text{init}}): \\
10 & \text{if } \text{Status} \hat{i}, j \neq \bot \\
11 & \text{or not } \Pi.\text{validSt}([\hat{i}, j], st_{\text{init}}): \\
12 & \text{ret } \bot \\
13 & \text{Setup}(\hat{i}, j) \leftarrow st_{\text{init}} \\
14 & \text{Status} \hat{i}, j \leftarrow \text{running} \\
\end{align*}

\begin{align*}
\text{Prep}(\hat{i} \in \mathbb{N}, j \in V, \hat{k} \in \mathbb{N}, m_{\text{msg}} \in \mathcal{M}^*): \\
14 & \text{if } \text{Status} \hat{i}, j \neq \text{running} \text{ or } \text{In} \hat{k}, j = \bot \text{ ret } \bot \\
15 & \text{if } \text{St}[\hat{i}, j, \hat{k}] = \bot: \\
16 & \text{St}[\hat{i}, j, \hat{k}] \leftarrow \text{Setup}(\hat{i}, j); m_{\text{msg}} \leftarrow (\text{Pub}[\hat{k},]) \\
17 & (n, m_0, m_1) \leftarrow \text{Used}[\hat{k}] \\
18 & (st, \text{out}) \leftarrow \\
21 & \Pi.\text{Prep}(\hat{j}, sk_j, \text{St}[\hat{i}, j, \hat{k}], n, m_{\text{msg}}, \text{In}[\hat{k}, j]) \\
22 & \text{if } st = \text{running}: \\
23 & (st, \text{msg}) \leftarrow \text{out}; \text{St}[\hat{i}, j, \hat{k}] \leftarrow st \\
24 & \text{else if } st = \text{finished}: \\
25 & \text{St}[\hat{i}, j, \hat{k}] \leftarrow \bot; \text{Out}[\hat{i}, j, \hat{k}] \leftarrow \text{out} \\
26 & \text{Batch}[\hat{i}, j, \hat{k}] \leftarrow m_0; \text{Batch}[\hat{i}, j, \hat{k}] \leftarrow m_1 \\
27 & \text{else if } st = \text{failed}: \text{St}[\hat{i}, j, \hat{k}] \leftarrow \bot \\
28 & \text{ret } (st, \text{msg}) \\
\end{align*}

\begin{align*}
\text{Agg}(\hat{i} \in \mathbb{N}, j \in V): \\
29 & \text{if } \text{Status} \hat{i}, j \neq \text{running} \text{ ret } \bot \\
30 & (st_1, \ldots, st_s) \leftarrow \text{Setup}(\hat{i}, j): \\
31 & \text{if } F(st_j, \text{Batch}[\hat{i}, j, .]) \neq F(st_{j'}, \text{Batch}[\hat{i}, j, .]) \\
32 & \text{and } (\forall j, j' \in V) \text{ st}_j = st_{j'} \land sk_j = sk_{j'}: \\
33 & \text{ret } \bot \\
34 & \text{Status} \hat{i}, j \leftarrow \text{finished} \\
35 & \text{ret } \Pi.\text{Agg}([\hat{i}, j, .]) \\
\end{align*}

Figure 4: Game for defining privacy of a complete, \( s \)-party VDAF \( \Pi \) for corruption threshold \( \geq 0 \). Let \( F \) denote the aggregation function for which \( \Pi \) is complete and let \( Q_{\text{init}} \) its set of initial states. Let \( T = [s] \setminus V \).

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 \text{Agg} oracle (lines 29–35). Queries to this oracle are subject to the restriction that the aggregate share does not trivially leak challenge bit: Namely, the aggregate of both batch of measurements specified by the adversary must be equal (31). (Tables \text{Batch}_0, \text{Batch}_1 keep track of the pairs of measurements \( m_0, m_1 \) passed to the \text{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 \cite[Appendix A]{} and \cite[Appendix A]{18}. 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** (Privacy). Let \( \Pi \) be an \( s \)-party VDAF and let \( t < s \) be a positive integer. Define the \( t \)-advantage of \( A \) in attacking the privacy of \( \Pi \) as

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

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

\section{Prio3}

In this section we present our security analysis for Prio3 \cite{9}. The starting point for this construction is an FLP system (Section 2) that defines the set of valid measurements; drawing on techniques from Boneh et al. \cite{17}, 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 \( F : \mathcal{I} \rightarrow \mathcal{O} \), we need the proof system to define how measurements \( (\mathcal{I}) \) are encoded as inputs to the prover and how refined shares are processed into the aggregate results \( (\mathcal{O}) \).
Figure 5: Definition of 1-round, s-party VDAF Prio3[FLP, PRG]. Let $\text{label}_1, \ldots, \text{label}_7$ be arbitrary, distinct bitstrings.

**Definition 4** (Affine, aggregatable encodings [23, Sec. 5.]). Let $F: \mathcal{I} \to \mathcal{O}$ be a function. An FLP system FLP admits an affine, aggregatable encoding for $F$ if it defines the following algorithms:

- **FLP.Encode** $(m \in \mathcal{I}) \to inp \in \mathbb{F}^n$ is an injective map from the domain of $F$ to the input space $\mathbb{F}^n$ of FLP.
- **FLP.Truncate** $(inp \in \mathbb{F}^n) \to out \in \mathbb{F}^{al}$ refines an FLP input into a format more suitable for aggregation. We call $al$ the output length.
- **FLP.Decode** $(ct \in \mathbb{N}, out \in \mathbb{F}^{al}) \to a \in \mathcal{O}$ 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 $\tilde{m} \in \mathcal{I}^{ct}$ it holds that

$$F(\tilde{m}) = \text{Decode}(ct, \sum_{i \in [ct]} \text{Truncate}(\text{Encode}(m[i])))$$

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 5. (This version includes some minor changes to idraft-irtf-cfrg-vdaf-03 [9], as we discuss below.) 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 [17, 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 used run on the combined verifier shares to decide validity.

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 $rSeed$ of “joint randomness hints” consisting of the intermediate values computed by the aggregators. This allows $jr$ to be computed immediately on receipt of the input shares. However, since a malicious client might try to break robustness by transmitting malformed hints, 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 $Q_{init} = \{\varepsilon\}$. In our security analysis, we assume honest aggregators to process a batch at most once. Accordingly, the allowed-state algorithm $\text{Prio3}[\text{FLP}, PRG].\text{validSt}$ 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. That is, on input of $st_{init} \in \{\varepsilon\}$ and $m \in I$, the refinement algorithm $\text{Prio3}[\text{FLP}, PRG].\text{refine}$ first encodes $m$, then truncates and decodes it as prescribed by FLP. The refine-from-shares algorithm, $\text{Prio3}[\text{FLP}, PRG].\text{refineFromShares}$, unpacks each input share (see Unpack in Figure 5), 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 $\text{finishResult}$ such that for all truncated outputs $out_1, \ldots, out_{ct} \in \mathbb{F}^\|l\|$ it holds that $\text{Decode}(ct, \sum_{k \in \{ct\}} out_k) = \text{finishResult}(ct, \text{Decode}(1, out_1), \ldots, \text{Decode}(1, out_{ct}))$.

**Changes to the specification [9].** Figure 5 differs from draft-03 of the VDAF spec in three ways. The most important change is to incorporate the nonce provided by the environment into the joint randomness computation. This turns out to be crucial for a tight robustness bound; without this change, we must contend with cases in which joint randomness is reused across reports.

Second, we have revised the domain separation tags for the PRG invocations so that each $RG_i$ in Figure 5 can be treated as an independent random oracle.

Lastly, we have moved the joint randomness hints from the input shares into the client’s initial broadcast message. This change allowed us to simplify our proofs somewhat, but we do not believe it is essential for security. It also has the added benefit of reducing overall communication overhead for $s > 2$.

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

**Theorem 1.** Modeling each $RG_i$ in Figure 5 as a random oracle, if FLP is $\epsilon$-sound (Section 2), then for every adversary $A$ against the robustness of $\Pi$ it holds that

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

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

For reasonable choices of the seed size for PRG, the loosest term in this bound is $(q_{RG} + q_{\text{Prep}}) \cdot \epsilon$. The multiplicative loss of $q_{RG} + q_{\text{Prep}}$ 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 $\epsilon$-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 $\text{Prep}$ queries to $\text{Prep}$, and it can use its queries to $RG_1$ to bias these interactions’ joint randomness.

**Proof sketch.** We sketch the security reduction here and defer the detailed proof to Appendix C.1. Our goal is to construct from $A$ a malicious prover $P^*$ for the soundness of FLP. The overall idea is to run $A$ in a simulation of the robustness game for $\Pi$ in which $P^*$’s instance of the soundness experiment (Figure 2) is embedded in a random $\text{Prep}$ query so that $P^*$ wins its game precisely when $A$ sets $w \leftarrow \text{true}$ for the first time in that query. The main difficulty is that $P^*$ must arrange to use the joint randomness it received as input in its own game. To provide a consistent simulation of $RG_1$, we need to arrange to extract the input to commit to from $A$’s queries. This results in a union bound over all queries to $RG_1$, either by the simulation of $\text{Prep}$ or by $A$ directly. \qed
Remark 2. For FLPs that do not make use of joint randomness (i.e., those for which $j_l = 0$), queries to $RG_1$ can be disregarded, as this oracle is not used by $\Pi$. In particular, a similar reduction can be shown that results in a multiplicative loss of just $q_{\text{prep}}$.

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 3.)

Theorem 2. Modeling each $RG_i$ in Figure 5 as a random oracle, if FLP is $\delta$-private, then for all $0 < t < s$ and attackers $A$ it holds that

$$\text{Adv}_{\Pi,t}^{\text{priv}}(A) \leq 2q_{\text{Shard}} + \frac{q_{\text{RG}} + q_{\text{Shard}}}{|N|} + \frac{s \cdot q_{\text{RG}}}{2^{\kappa-1}},$$

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

Proof sketch. The full proof is given in Section C.2. The main idea is to arrange for $A$’s queries to its oracles to be independent of the challenge bit. We do so via a game-playing argument in which we incrementally revise the game until the outcome of each oracle is independent of the current state of the game. The last step involves a hybrid argument, where in each hybrid world we replace one invocation of the proof- and query-generation algorithms of FLP (see Figure 2) with invocation of the simulator hypothesized by the $\delta$-privacy of FLP. This accounts for the multiplicative loss of $q_{\text{Shard}}$ in the bound.

Remark 4. Instead of using separate seeds for the input share, proof share, and blind, it may be safe to reuse the same seed for all three purposes, similar to the seed in Doplar (Section 5). This may only result in a slightly looser bound: This change would the attacker to test guesses of the input share because the joint randomness part would be derived from the same seed.

5 Doplar

Poplar1 [9] solves 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 aggregate result is a vector representing the number of measurements prefixed by each candidate.

The “secure sketch” MPC of Boneh et al. [18] 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 Appendix A we show how to instantiate this primitives using a simple technique from de Castro and Polychroniadou [22].

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 in Section 5.2 how to transform an FLP into one that is delayed input [42]. 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 Appendix B.

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, at least for the current instantiations of the VIDPF and delayed-input FLP. We compare the communication 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 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 [19, 22], and a construction specifically for VIDPF was given in [22].

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

- **VIDPF**.\(\text{VEval}(id \in \{1, 2\}, key \in \{0, 1\}^n, pub \in \mathcal{M}, \vec{x} \in \{0, 1\}^l) \rightarrow \{0, 1\}^n \times (\mathbb{G}_k)^n\) 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.
- **VIDPF**.\(\text{Verify}(h_1, h_2) \rightarrow \{0, 1\}\) 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

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

We say VIDPF is correct if, for all \(\alpha \in \{0, 1\}^n\), all \(\vec{\beta} \in \mathbb{G}_1 \times \cdots \times \mathbb{G}_n\), all \(\vec{x} \in \{0, 1\}^l\), all \((key_1, key_2, pub) \in \text{Gen}(\alpha, \vec{\beta})\), all \((h_1, \vec{y}_1) \in \text{VEval}(1, key_1, pub, \vec{x})\), and all \((h_2, \vec{y}_2) \in \text{VEval}(2, key_2, pub, \vec{x})\):

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

Theorem 3 requires VIDPF to be extractable. Intuitively, there should be an algorithm that can extract \(\alpha, \vec{\beta}\) from adversarially generated VIDPF key shares. Then \(\text{VEval}\) must produce shares consistent with the incremental point function \(f_{\alpha, \vec{\beta}}\), whenever \(\text{Verify}\) succeeds. (A similar property is formalized for IDPFs by BBCG+21.) This property implies, among other things, that if \(\text{Verify}\) succeeds, then shareholders are guaranteed to hold shares of a one-hot vector. We formalize this property below.

**Definition 5** (Extractable VIDPF (cf. [18, Definition 7])). Suppose that VIDPF is defined in terms of a random oracle with co-domain \(\mathcal{V}\). Refer to the game in Figure 6 associated to VIDPF, extractor \(E\), and adversary \(\mathcal{A}\). Define \(\mathcal{A}\)'s advantage in fooling \(E\) as \(\text{Adv}_{\text{VIDPF, } E}(\mathcal{A}) = 2 \cdot \Pr[\text{Exp}_{\text{VIDPF, } E}^{\text{extract}}(\mathcal{A}) = 1] - 1\).

Finally, our privacy reduction for Doplar (Theorem 4) requires the underlying VIDPF to be private, in the sense that one shareholder’s view—consisting of its share \(key_j\), the public share \(pub\), and the other shareholder’s verification value \(h\)—leaks nothing about the secrets \(\alpha\) and \(\vec{\beta}\). Prior definitions of verifiable FSS—e.g., the one in de Castro and Polychroniadou [22]—only define privacy with respect to a single vector of evaluation points and single 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 for privacy than in prior works.\(^3\)

**Definition 6.** Let \(\text{Exp}_{\text{VIDPF, } S}^{\text{priv}}(\mathcal{A})\) be the privacy game for VIDPF, simulator \(S = (S_1, S_2)\), and adversary \(\mathcal{A}\) defined in Figure 6. Define the advantage of \(\mathcal{A}\) in distinguishing \(S\)’s simulation from its view of VIDPF’s execution as \(\text{Adv}_{\text{VIDPF, } S}^{\text{priv}}(\mathcal{A}) = 2 \cdot \Pr[\text{Exp}_{\text{VIDPF, } S}^{\text{extract}}(\mathcal{A}) = 1] - 1\).

If this privacy game withholds the Sketch oracle from the adversary (shaded in Figure 6) then we obtain the privacy game for plain IDPFs, with the adversary’s advantage defined analogously.

In Section A we formally describe a VIDPF construction that satisfies all the necessary security properties. The construction is heavily based on the verifiable DPF technique from [22].

\(^3\)The game does not need to provide an oracle for VIDPF.\(\text{Verify}\) since it is a deterministic algorithm whose inputs are known to the adversary.
We adopt the terminology of delayed-input 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 (Appendix B).

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 [42] is often regarded as the first ZK protocol with delayed input, while Katz and Ostrovsky [40] were the first to explicitly rely on the delayed input property while using a ZK proof in an application.

**Definition 7.** A delayed-input FLP DFLP consists of the following algorithms:

- **DFLP.Encode**($\Delta \in \mathbb{F}^{el}, x \in \mathbb{F}^{n} \rightarrow e \in \mathbb{F}^{n'}$: Takes as input encoding randomness $\Delta$, and an input instance $x$. Returns an encoding of $x$; we let $n'$ denote the length of the encoding. The function $\text{Encode}(\Delta, \cdot)$ must be a linear function and invertible. We denote the inverse by $\text{Decode}$.

- **DFLP.Prove**($\mathcal{X} \subseteq \mathbb{F}^{n} \Delta \in \mathbb{F}^{el}, jr \in \mathbb{F}^{jl} \rightarrow \pi \in \mathbb{F}^{m}$: Takes as input a set of possible instances, encoding randomness $\Delta$, and joint randomness $jr$. Produces output proof $\pi$.

- **DFLP.Query**($e \in \mathbb{F}^{n}, \Delta \in \mathbb{F}^{el}, \pi \in \mathbb{F}^{m}, jr \in \mathbb{F}^{jl}, qr \in \mathbb{F}^{ql} \rightarrow \sigma \in \mathbb{F}^{v}$: Takes as input an encoded instance $e$, encoding randomness $\Delta$, proof $\pi$, joint randomness $jr$, and query randomness $qr$. Returns a verifier $\sigma$. The function $\text{Query}(\cdot, \cdot, \cdot, \cdot, qr)$ must be linear.
If arbitrary, distinct bitstrings.

\[
\Delta[\ell] \leftarrow \text{RG}_0(\text{seed}_1, n \parallel \ell \parallel 1)
\]
\[
+ \text{RG}_2(\text{seed}_2, n \parallel \ell \parallel 2)
\]
\[
\tilde{\pi}[\ell] \leftarrow \text{DFLP}.\text{Encode}(\Delta[\ell], 1)
\]

(\text{key}, \text{key}, \text{pub}) \leftarrow \text{VIDPF}.\text{Gen}(\alpha, \beta)

// Prepare the joint randomness parts.
\[
\text{seed}[1] \leftarrow \text{RG}_0(\text{seed}_1, n \parallel 1 \parallel \text{pub} \parallel \text{key}_1)
\]
\[
\text{seed}[2] \leftarrow \text{RG}_2(\text{seed}_2, n \parallel 2 \parallel \text{pub} \parallel \text{key}_2)
\]

// Generate the level proofs.
\[
\text{jq}[\ell] \leftarrow \text{DFLP}.\text{Prove}(\{0, 1\}, \tilde{\Delta}[\ell], jq)
\]
\[
\text{pf}[\ell] \leftarrow \pi \leftarrow \text{RG}_3(\text{seed}_3, n \parallel \ell)
\]

// Process initial message from client
\[
\text{x}[1] \leftarrow (\text{key}, \text{seed}_1, \text{pf})
\]
\[
\text{x}[2] \leftarrow (\text{seed}_2, \text{pf})
\]

\[
\text{msg} \leftarrow (\text{pub}, \text{seed})
\]

Algorithm Unpack\((\ell, x, n)\):

\[
\text{if } \ell = 1: (\text{key}, \text{seed}, \text{pf}) \leftarrow x; \pi \leftarrow \text{pf}[\ell]
\]
\[
\text{else: (key, seed)} \leftarrow x; \pi \leftarrow \text{RG}_3(\text{seed}, n \parallel \ell)
\]

\[
\text{ret (key, seed, } \pi)\]

Figure 7: Definition of 1-round, 2-party VDAF Doplar[VIDPF, DFLP, PRG]. Let \(\text{label}_1, \ldots, \text{label}_6\) be arbitrary, distinct bitstrings.

- DFLP.\text{Decide}(\sigma \in \mathbb{F}^n) \rightarrow \text{acc} \in \{0, 1\}: Takes as input query responses \(\sigma\) and returns a boolean.

If \text{Prove} is restricted to sets \(\mathcal{X}\) with \(|\mathcal{X}| = k\) 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 \(\mathcal{X} \subseteq \mathcal{L}\), all \(x \in \mathcal{X}\), and all \(\Delta\):

  \[
  \Pr[\text{Decide}(\sigma) : jq \leftarrow \mathbb{F}^k; \pi \leftarrow \text{Prove}(\mathcal{X}, \Delta, jq); \\
  \sigma \leftarrow \text{Query}(\text{Encode}(\Delta, x), \Delta, \pi, jq) = 1.
  \]

- **Soundness** (with respect to \(\mathcal{L}\)): The scheme should be sound in the usual sense of FLPs, with respect to the language \(\mathcal{L}' = \{\text{Encode}(\Delta, x) \mid x \in \mathcal{L}\}\). In other words, it is hard for a malicious prover to generate a proof that verifies with respect to \((c, \Delta) \notin \mathcal{L}'\).

- **Privacy**: In Figure 6 we define a game for delayed-input FLPs, in which the proof is generated using some set \(\mathcal{X}\) of candidates, and later verified with respect to a particular \(x \in \mathcal{X}\). A delayed-input FLP is \(\delta\)-private if there exists a simulator \(S\) such that every \(A\)'s advantage is \(\text{Adv}_{\text{DFLP,S}}^\text{priv}(A) \leq \delta\), where

  \[
  \text{Adv}_{\text{DFLP}}^\text{priv}(A) = 2 \cdot \Pr[\text{Exp}_{\text{DFLP,S}}^\text{priv}(A)] - 1.
  \]
5.3 Construction

We specify our construction Doplar[VIDPF,DFLP,PRG] in Figure 7. Its three components are: A verifiable IDPF VIDPF with input length $\eta$; a delayed-2-input FLP DFLP with input set $\{0, 1\}$, proof length $m$, encoded input length $\alpha$, encoding randomness length $el$, joint randomness length $jl$, and query randomness length $ql$; and a pseudorandom generator PRG (Section 2) with seed length $\kappa$. To be suitable for our construction, we must choose VIDPF and DFLP so that $\text{VIDPF,DFLP,PRG}\{\ell}\eta$ for each $\ell \in [\eta]$.

To shard its measurement $\alpha \in \{0, 1\}^7$, the client begins by running the VIDPF key generator on $\alpha$. The initial state for Doplar encodes the “level” $\ell$ at which the VIDPF shares are to be evaluated; each candidate prefix must have length $\ell$. (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.) Thus, 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 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 $st = (\ell, pfx)$ if $\ell$ is distinct for all previous states and each of the prefixes $pfx$ is distinct.

**Remark 5.** Although not addressed in Boneh et al. [18] 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 $F$ for which the non-zero element is equal to 0 or 1. For a given initial state $(\ell, pfx)$, this can be computed from the VIDPF public share and key shares by evaluating the shares on each of the prefixes $pfx$. Aggregation consistency for Doplar is similarly straightforward: Since the refined share space and aggregate share space are the same.

**Security.** Let $\Pi = \text{Doplar[VIDPF,DFLP,PRG]}$ as specified above. Let $\mathcal{N}$ be the nonce space and let $\kappa$ be the seed length for PRG.

**Theorem 3.** Modeling each RG$_i$ in Figure 7 as a random oracle, if DFLP is $\epsilon$-sound, then for all $t_A$-time adversaries $A$ and $t_E$-time extractors $E$ there exists a $O(t_A + q_{\text{prep}}t_E)$-time adversary $B$ for which

$$\text{Adv}_{\Pi}^{\text{extract}}(A) \leq 2(q_{\text{RG}} + q_{\text{prep}}) \cdot \epsilon + \frac{(q_{\text{RG}} + 3q_{\text{prep}})^2}{2\kappa} + q_{\text{prep}} \cdot \text{Adv}_{\text{VIDPF,DFLP}}(B),$$

where $A$ makes $q_{\text{prep}}$ queries to Prep and a total of $q_{\text{RG}}$ queries to its random oracles.

**Proof sketch.** The proof has a similar structure to Theorem 1 in that the last step is a reduction to the soundness of DFLP. However in order to use this, we must first revise the game so that the challenge input issued by the malicious prover $P^*$ was constructed from the sum of refined shares that are otherwise valid (i.e., one-hot). Using the extractability property of VIDPF, we can simplify the winning condition by extracting the input measurement from the adversary’s random oracle queries and use it to compute the refined measurement whenever the one-hotness check succeeds. Refer to Appendix C.3 for the full proof.

**Theorem 4.** For all $t_A$-time adversaries $A$ and $t'$-time simulators $S,T$ there exist $O(t_A + q_{\text{shard}}t')$-time adversaries $B,C$ for which

$$\text{Adv}_{\Pi,1}^{\text{priv}}(A) \leq 2q_{\text{shard}} \left( \text{Adv}_{\text{VIDPF,DFLP}}^{\text{priv}}(B) + \eta \cdot \text{Adv}_{\text{DFLP}}^{\text{priv}}(C) \right) + \frac{\eta q_{\text{RG}} + q_{\text{shard}}}{|\mathcal{N}|} + \frac{3q_{\text{RG}}}{2\kappa - 1},$$
where each $RG_i$ in Figure 7 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 $Shard$.

Proof sketch. The reduction to DFLP privacy follows the same lines as Theorem 2 except there are $\eta \cdot q_{Shard}$ different hybrid worlds in the last step. Privacy of VIDPF is used to ensure that the simulation of the boundary world can be carried out without access to the input measurement. Refer to Appendix C.4 for the proof.

Comparison to Poplar1. To finish this section, we evaluate the bandwidth cost of our round-reduced variant of Poplar1. The parameters chosen for Poplar1 by the specification [9] match those in the performance evaluation conducted by Boneh et al. [17]. We therefore take these parameters as our basis for comparison. In the following, we have instantiated VIDPF and DFLP as described in Appendix A and Appendix B respectively.

Boneh et al. [17] claim a per-report robustness bound of roughly $2/|F|$, where $F$ is the field chosen for the inner nodes. 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 choose the PRG seed length to be $\kappa = 128$.

In Figure 8 we plot the communication cost of Doplar and Poplar1 for various choices of input lengths $\eta$. We plot the total number of kilobytes sent by each client and the total number of kilobytes sent by each aggregator, per report, over all $\eta$ 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.

Level skipping. One way to improve bandwidth for both schemes is to “skip” IDPF evaluation at certain levels. For example, if we descend the IDPF tree in $\tau$-bit increments instead of 1-bit increments, then (1) our VIDPF construction requires one-hot check material only in every $\tau$-th level, and (2) the Doplar construction requires DFLPs only at every $\tau$-th level. As a result, these major contributors to communication cost are reduced by a factor of $\tau$. Additionally, the process of aggregating (traversing the tree of prefixes to find heavy hitters) requires fewer rounds by a factor of $\tau$. The trade-off is that we consider more candidate prefixes at each level—i.e., at each step we consider the $2^\tau$ descendants at depth $\tau$ from each candidate—but this cost is amortized over the batch.

Notably, the impact of this optimization is more significant for Doplar than for Poplar1. (For example, a “skip factor” of $\tau = 2$, i.e., skipping every other level, reduces the client’s overhead from 6 to 5 times that of Poplar1 with the same optimization.) This is primarily due to the reduction in the number of delayed-input proofs, which make up the bulk of the first input share. (The second input share compresses its shares of the proofs into a single PRG seed.)

---

Footnotes:

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

5. The underlying (non-verifiable) IDPF is still organized as a binary tree, so its cost is not affected.
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 the standardization effort rests on firm formal foundations. However, it leaves 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 [29]. 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. That is, we 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* [48]: 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 provide for differential privacy (DP) [27] in addition to secure computation. There are definitions of DP that extend to the multi-party setting [44, 51], and a number of works have considered MPC protocols for aggregate functionalities that also guarantee differential privacy of the outputs [50, 35, 10]. 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 in the communication cost of Doplar. 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 (Appendix B), while simple, effectively doubles the proof size of the base FLP.

References


In this section we present our proposed VIDPF construction.
De Castro-Polychroniadou technique. De Castro & Polychroniadou [22] (hereafter DP22) proposed the following simple and elegant technique to verify that a vector is one-hot. Consider a vector $\vec{v}$ that is additively secret-shared $\vec{v} = \vec{v}_1 \oplus \vec{v}_2$. For simplicity, we describe the technique assuming that the sharing is with respect to XOR, since in that case the shares of zero are identical strings. The technique adapts readily to the more general case of additive shares over any group. Assume also that the parties have additive shares of a binary indicator vector $\vec{b} = \vec{b}_1 \oplus \vec{b}_2$, which is nonzero exactly in the same positions that $\vec{v}$ is.

First, observe that the parties can easily verify whether they hold shares of an all-zeroes vector, since this happens if and only if their shares (as strings) are identical. They can simply exchange and compare hashes of their share-vectors (although see our remark below for a disclaimer about this idea). The technique of DP22 is to adjust a one-hot vector into an all-zeroes vector, with the help of the dealer.

Define

$$\text{adjust}(\vec{v}_i, \vec{b}_i, C) = \left(H(1, \vec{v}_i[1]) \oplus \vec{b}_i[1] \cdot C, \ H(2, \vec{v}_i[2]) \oplus \vec{b}_i[2] \cdot C, \ldots \right)$$

If $\vec{v}$ and $\vec{b}$ are nonzero in (only) position $i^*$, then set $C^* = H(i^*, \vec{v}_1[i^*]) \oplus H(i^*, \vec{v}_2[i^*])$. Now consider the result of both shareholders applying $\text{adjust}(\cdot, \cdot, C^*)$ to their shares:

- In positions $i \neq i^*$ where they share zero, we have $\vec{v}_1[i] = \vec{v}_2[i]$ and $\vec{b}_1[i] = \vec{b}_2[i]$. For these positions in the output of $\text{adjust}$, both parties will compute identical strings.

- In position $i^*$, the parties have $\vec{b}_1[i^*] \neq \vec{b}_2[i^*]$. By symmetry, suppose $\vec{b}_1[i^*] = 1$ and $\vec{b}_2[i^*] = 0$. Then the first party will compute

$$H(i^*, \vec{v}_1[i^*]) \oplus C^* = H(i^*, \vec{v}_1[i^*]) \oplus (H(i^*, \vec{v}_1[i^*]) \oplus H(i^*, \vec{v}_2[i^*]))$$

$$= H(i^*, \vec{v}_1[i^*])$$

and the second party will compute $H(i^*, \vec{v}_2[i^*])$ as well.

In all cases, both parties will compute the same output of $\text{adjust}$, which they can check for equality by exchanging and comparing hashes. Hence, the dealer will compute and the $C^*$ value and include it in the parties’ DPF keys. They can use $C^*$ to perform their verification.

To see why the DP22 approach is sound, suppose the parties hold shares of a non-one-hot vector — i.e., it is nonzero at positions $i \neq i^*$. Do both parties compute the same output of $\text{adjust}$? This can only happen if $C^*$ value somehow corrects both positions $i$ and $i'$, and this happens only when

$$H(i, \vec{v}_1[i]) \oplus H(i, \vec{v}_2[i]) = C^* = H(i', \vec{v}_1[i']) \oplus H(i', \vec{v}_2[i'])$$

$$\iff H(i, \vec{v}_1[i]) \oplus H(i, \vec{v}_2[i]) \oplus H(i', \vec{v}_1[i']) \oplus H(i', \vec{v}_2[i']) = 0$$

The construction is therefore sound if it is hard to find “multi-collisions” of this form in $H$. In particular, if $H$ is a random oracle with output length $4\kappa$ then an adversary making $q < 2^\kappa$ queries to $H$ can find such a collision with probability bounded by $q^4/2^{4\kappa} \ll q/2^\kappa$.

Issues with DP22. We point out two issues with the technique of DP22, which we remedy in our subsequent presentation. First, the construction described in DP22 does not use the index $i$ as a domain-separating first argument to $H$. If somehow $\vec{v}_1[i] = \vec{v}_1[i']$ and $\vec{v}_2[i] = \vec{v}_2[i']$, and the indices $i, i'$ are not used as input to $H$, then a single $C^*$ value $H(\vec{v}_1[i]) \oplus H(\vec{v}_2[i])$ would cause the parties to compute identical outputs of $\text{adjust}$ in both positions $i$ and $i'$. In this situation, it is possible that a non-one-hot vector can pass the verification, breaking soundness. Adding these domain separations, as we have in our presentation above, fixes the issue.

Second, party #1 holds its share $\vec{v}_1$ and the correction value $C^* = H(i^*, \vec{v}_1[i^*]) \oplus H(i^*, \vec{v}_2[i^*])$. Suppose this party has a guess for $i^*$ and a guess for the nonzero value $v = \vec{v}_1[i^*] \oplus \vec{v}_2[i^*]$. Then she can verify this guess by checking whether $C^* = H(i^*, \vec{v}_1[i^*]) \oplus H(i^*, \vec{v}_1[i^*] \oplus v)$ — all values she knows. Hence, $C^*$ exposes an offline dictionary attack on the secret values $i^*$ and $\vec{v}[i^*]$. If $\vec{v}[i^*]$ is high entropy, then this is no vulnerability at all. But if $\vec{v}[i^*]$ is known to be a small value like 1 (as is the case in many applications), then this issue allows a corrupt shareholder to unilaterally learn $i^*$, violating privacy.\footnote{The [22] always hashes a high-entropy PRG seed, so we do not claim a concrete attack against their construction. However, their privacy proof does not seem to use the fact that this argument to the hash is high entropy.} We resolve this by simply ensuring...
VIDPF.Gen(α ∈ {0, 1}^n, β ∈ G_1 × · · · × G_n):
1. for ℓ ∈ [η]:
2. \( \tilde{R}[ℓ] \leftarrow \{0, 1\}^\kappa \)
3. \( \tilde{β}^* [\ell] \leftarrow (1, \tilde{β}[\ell], \tilde{R}[\ell]) \)
4. \( (key_1, key_2, pub) \leftarrow \text{IDPF.Gen}(α, \tilde{β}^*) \)
5. for ℓ ∈ [η]:
6. \( pfx \leftarrow α[1 : ℓ] \)
7. \( \langle \cdot, data_1, R_1 \rangle \leftarrow \text{IDPF.Eval}(1, key_1, pub, pfx) \)
8. \( \langle \cdot, data_2, R_2 \rangle \leftarrow \text{IDPF.Eval}(2, key_2, pub, pfx) \)
9. \( C[\ell] \leftarrow \text{RG}(pfx || data_1, \| R_1) \)
10. \( \oplus \text{RG}(pfx || data_2 \| R_2) \)
11. \( pub^* \leftarrow (pub, C) \)
12. ret (key_1, key_2, pub^* )

VIDPF.VEval(id, key, pub*, x):
13. \( (pub, C) \leftarrow pub^* \)
14. for i ∈ [ |x| ]
15. \( y[i] \leftarrow \text{IDPF.Eval}(id, key, pub, x[i]) \)
16. \( (b, data[i], R) \leftarrow y[i] \)
17. \( h \leftarrow h \| \text{adjust}(id, key, pub^*, b, x[i]) \)
18. ret (h, data)

VIDPF.adjust(id, key, pub*, b, x):
19. \( (pub, C) \leftarrow pub^* \)
20. if |x| = 0: ret x // length of x as a bit string
21. \( \langle \cdot, d, R \rangle \leftarrow \text{IDPF.Eval}(id, key, pub, b, x) \)
22. \( \text{prefix} \leftarrow \text{adjust}(id, key, pub^*, b, x[1 : |x| - 1]) \)
23. ret prefix \( \| (\text{RG}(x \| (-1)^{id}d \| (-1)^{id}R) \oplus b \| C[x]) \)

VIDPF.Verify(h_1, h_2):
24. ret \( h_1 =\equiv h_2 \)

Figure 9: VIDPF construction VIDPF[IDPF], based on any IDPF. If the VIDPF is to be instantiated with groups \( G_1, \ldots, G_η \) then the underlying IPDF is instantiated with groups \( \mathbb{G}_1, \ldots, \mathbb{G}_η \), where \( \mathbb{G}_ℓ = \{0, 1\} \times \mathbb{G}_ℓ \times \{0, 1\}^κ \).

that the dealer encodes a random element at the one-hot position (in addition to a potentially low-entropy desired value).

Extending to incremental DP. The technique of DP22 is well-suited for DPFs. In an incremental DPF (IDPF), we can apply their technique to each prefix-length. However, this guarantees only that each prefix-length corresponds to some point function. It does not necessarily guarantee that the point functions of the different prefix-lengths satisfy the prefix condition that is needed in an IDPF.

In our construction, we extend the DP22 technique to IDPFs. For each evaluation of the IDPF — say, at point \( x \) — we compute the adjustment strings using the DP22 technique, for \( x \) and all of its prefixes. This alone is not enough to guarantee the prefix property. To “tie different prefix lengths together,” we ask the shareholders to compute the adjustment strings with respect to the same sharings of the indicator bit, for all the prefixes of \( x \). We show that this forces the point functions at every prefix-length to be prefix-consistent.

Immediate Optimizations in an Implementation. Our construction evaluates the underlying IDPF on all prefixes of the given strings. Doing this naively would increase the computational costs by a factor of \( ℓ \) when evaluating on strings of length \( ℓ \). However, these extra evaluations are essentially free in existing IDPFs — while evaluating at string \( x \), these constructions already evaluate all prefixes of \( x \) along the way. A reasonable implementation of our VIDPF will take advantage of this fact.

The verification value \( h \) produced by \text{VEval} is a very long string, consisting of \( ℓ \cdot 4κ \) bits for each query point of length \( ℓ \). If parties are to exchange these \( h \) values in an application of our VIDPF, it would account for a significant fraction of the total communication. However, the Verify algorithm that uses these \( h \) values merely checks them for equality. Therefore, it suffices for each party to send only a collision-resistant hash of their \( h \) value, which can have fixed length only \( 2κ \). This optimization changes the concrete security bound for VIDPF soundness, by adding a term for the probability of finding a collision under the hash function.

Lemma 1. Let IDPF be an IDPF and RG be a random oracle with outputs of length \( 4κ \). Let \( A \) be an adversary making \( q \) queries to RG. There is a \( O(t_A) \)-time adversary \( A' \) such that the construction VIDPF[IDPF] in

---

7 We have chosen to describe our VIDPF to use an underlying IDPF as a black-box. When this is the case, we must ensure that the IDPF outputs have sufficient entropy for the one-hotness check. If we were to instead analyze our VIDPF (instantiated with a natural IDPF construction) as a monolithic construction, it is likely that the underlying IDPF would already have internal entropy available that could be used for the one-hotness check. I.e., we may be able to obtain smaller share sizes by exploiting internal properties of the underlying IDPF.
Figure 9 satisfies the following:
\[
\text{Adv}^\text{extract}_{\text{VIDPF}[\text{IDPF}], E}(A) \leq (q^4 + q^2)/2^{4k}
\]
\[
\text{Adv}^\text{priv}_{\text{VIDPF}[\text{IDPF}]}(A) \leq \text{Adv}^\text{priv}_{\text{IDPF}}(A') + q/2^k
\]

Proof. Correctness of our construction follows from the discussion above, and is the same as in DP22.

Extractability: We begin with a few observations, which hold for all VIDPF keys, even adversarially generated ones:

**Observation:** If \(\text{adjust}(1, \text{key}_1, pub^*, b_1, x) = \text{adjust}(2, \text{key}_2, pub^*, b_2, x)\), then \(\text{adjust}(1, \text{key}_1, pub^*, b_1, x') = \text{adjust}(2, \text{key}_2, pub^*, b_2, x')\) as well, for every prefix \(x'\) of \(x\). This follows trivially by inspection and the recursive nature of \(\text{adjust}\). Note that the same \(b_1, b_2\) are used for both \(x\) and \(x'\).

**Observation:** Let \(pub^* = (pub, \bar{C})\). Suppose \(\text{adjust}(1, \text{key}_1, pub^*, b_1, x) = \text{adjust}(2, \text{key}_2, pub^*, b_2, x)\), and \(\text{IDPF.Eval}(1, \text{key}_1, pub, x) = (\rho, y_1, R_1)\), and \(\text{IDPF.Eval}(2, \text{key}_2, pub, x) = (\rho, y_2, R_2)\). Then:

1. If \(b_1 = b_2\) then \(\text{RG}(x || -y_1 || -R_1) = \text{RG}(x || y_2 || R_2)\). This includes the case where \((y_1, R_1) + (y_2, R_2) = (0, 0)\), making the two calls to \(\text{RG}\) identical. It also includes the case where these two calls to \(\text{RG}\) are a collision.
2. If \(b_1 \neq b_2\) then \(\bar{C}[||x||] = \text{RG}(x || -y_1 || -R_1) \oplus \text{RG}(x || y_2 || R_2)\).

This observation can be verified by inspection.

Let \(E_1\) denote the bad event that the adversary queries \(\text{RG}\) and observes a collision. If \(\text{RG}\) has outputs of length \(4k\), and the adversary makes \(q\) oracle queries, then the probability of this bad event is bounded by \(q^2/2^{4k}\). When \(E_1\) does not happen, then in condition (1) above, only the case that \((y_1, R_1) + (y_2, R_2) = (0, 0)\) is possible.

Let \(E_2\) denote the bad event that the adversary makes any four queries to \(\text{RG}\) that satisfy:

\[
\text{RG}(\text{pfx} || -d_1 || -R_1) \oplus \text{RG}(\text{pfx} || d_2 || R_2) = \\
\text{RG}(\text{pfx'} || -d'_1 || -R'_1) \oplus \text{RG}(\text{pfx'} || d'_2 || R'_2)
\]

for \(\text{pfx} \neq \text{pfx'}\) and \(d_1 + d_2 \neq 0\) and \(d'_1 + d'_2 \neq 0\). (These conditions ensure that the four calls to \(\text{RG}\) must be on distinct inputs.) If \(\text{RG}\) has outputs of length \(4k\), and the adversary makes \(q\) oracle queries, then the probability of this bad event is bounded by \(q^4/2^{4k}\). When \(E_2\) does not happen, then any value \(C \in \{0, 1\}^{4k}\) uniquely determines at most one pair of queries satisfying \(C = \text{RG}(\text{pfx} || -d_1 || -R_1) \oplus \text{RG}(\text{pfx} || d_2 || R_2)\).

We can apply the two observations inductively and obtain the following. If \(\text{adjust}(1, \text{key}_1, pub^*, b_1, x) = \text{adjust}(2, \text{key}_2, pub^*, b_2, x)\) for \(b_1 \neq b_2\), then every correction word \(\bar{C}[||\ell||] \in \{0, 1\}^k\) must be of the form \(\text{RG}(x[1: \ell] || \cdots) \oplus \text{RG}(x[1: \ell] || \cdots)\), for \(\ell \leq |x|\). Then, provided that \(E_2\) does not happen, there is at most one \(x\) of length \(\ell\) for which \(\bar{C}\) can be written in this way.

Combining all of these observations, we can define the extractor as shown in Figure 10.

Conditioned on the event that \(E\) doesn’t abort (which happens only with probability \((q^4 + q^2)/2^{4k}\)), we claim that the adversary has no advantage in the extractability game.

Consider a query to \(\text{Eval}(\bar{x})\) in the game, and assume the call to \(\text{Verify}\) succeeds. Then for every \(\bar{x}[i]\), the corresponding calls to \(\text{adjust}\) produce identical output. If these calls to \(\text{adjust}\) have \(b_1 = b_2\), and \(\bar{E}_1\) has not happened, then the corresponding output \(y[i]\) must be 0. If these calls to \(\text{adjust}\) have \(b_1 \neq b_2\), then \(\bar{C}[||\ell||] \in \{0, 1\}^k\) must have the form \(\text{RG}(\bar{x}[i] || \cdots) \oplus \text{RG}(\bar{x}[i] || \cdots)\). If \(E_2\) has not happened, then \(\bar{x}[i]\) is in fact unique with this property, and therefore \(\bar{x}[i]\) is a prefix of \(\alpha\) computed by the extractor \(E\). One can easily check that \(E\) extracts \(\beta[||\ell||] \in \{0, 1\}^\ell\) that is equal to the VIDPF output \(y[i]\). In other words, \(\bar{y}\) matches the output of \(f_\alpha, \beta\). Hence, the adversary’s advantage is zero.

Privacy: Let \(S^{\text{IDPF}}\) be the simulator for privacy for the underlying IDPF. The simulator for our construction is given in Figure 11. We prove privacy in a series of hybrids, also illustrated in Figure 11. Game \(G_0\) refers to the original experiment \(\text{Exp}^\text{priv}_{\text{IDPF}}\), where we have inlined the definition of \(S_2\) for convenience. The \(b = 0\) and \(b = 1\) branches of the Sketch oracle differ only in whose shares are given as input to VIDPF.Eval. By the correctness of the scheme, the distinction doesn’t matter, so the Sketch oracle is independent of \(b\). Eliminating the conditional in the Sketch oracle, we obtain \(G_1\), which is distributed identically to \(G_0\).
\[ E(key_1, key_2, pub^*, \text{Rand}); \]

1. \((pub, C) \leftarrow pub^*\)
2. if \(E_1\) or \(E_2\): // defined in the text, here with respect to oracle queries listed in Rand
   3. abort
   4. \(\alpha \leftarrow \text{empty string}\)
   5. for \(\ell \in [n]\):
      6. if \(\exists a \in \{0, 1\}, y_1, R_1, y_2, R_2\) such that
         7. \(C[\ell] = \text{Rand}[(\alpha \parallel a) \parallel y_1 \parallel R_1] \oplus \text{Rand}[(\alpha \parallel a) \parallel y_1 \parallel R_2]\)
      8. \(\alpha \leftarrow \alpha \parallel a\)
      9. \(\tilde{\beta}[\ell] \leftarrow y_1 + y_2\)
     10. else: \(\alpha \leftarrow \alpha \parallel 0; \tilde{\beta}[\ell] \leftarrow 0\)
   11. \(\text{ret} (\alpha, \tilde{\beta})\)

Figure 10: Extractor for the proof of Lemma 1.

\(G_3\) is identical to \(G_1\), but we have inlined the definition of VIDPF.Gen for convenience. By the correctness of the underlying IDPF, outputs of IDPF.Eval(1, \cdot) and IDPF.Eval(2, \cdot) are secret-shares of the appropriate plaintext values. So it has no effect on the adversary's view to solve for the output of IDPF.Eval(3 − \(\hat{j}\), \cdot) using the plaintext values and the output of IDPF.Eval(\(\hat{j}\), \cdot), instead of using key\(_{3−\hat{j}}\). In doing so, we obtain \(G_3\) which is distributed identically to \(G_2\).

Now notice that in \(G_3\), the value key\(_{3−\hat{j}}\) is never used. As such, we can replace line 26 (the call to IDPF.Gen) with a corresponding call to the simulator \(\hat{s}^{\text{IDPF}}\), which generates a simulated key\(_j\) and pub. Call the result \(\hat{G}_3\) (not pictured); this advantage in distinguishing \(G_3\) from \(\hat{G}_3\) is at most \(\epsilon_{\text{priv}}\).

In \(\hat{G}_3\), the random values \(\hat{R}[\ell]\) are used only to solve for \(R_{3−\hat{j}}\), which is in turn used only as an argument to RG. Define a bad event that the adversary ever queries \(\hat{R}[\ell]\) at an input of this form — i.e., of the form \(\hat{R}[\cdot \parallel \cdot \parallel \hat{R}[\ell] \ominus R_{\hat{j}}]\). The probability of the bad event is bounded by \(q/2^k\) since \(\hat{R}[\ell]\) is uniformly random. Conditioned on this bad event not happening, the results of these queries to RG are freshly random, and the value that is assigned to \(C[\ell]\) is uniform. In that case, the behavior of the game is independent of the challenge bit because \(S_1\) also assigns uniform values to \(\hat{C}[\ell]\). The advantage in guessing the challenge bit is therefore bounded by the probability of the bad event.

\[\square\]

B Instantiating Delayed-Input FLP

Our main result is to construct a delayed-2-input FLP for use in Doplar.

**Lemma 2.** The construction in Figure 12 (when suitably instantiated) is a delayed-2-input FLP with perfect completeness, soundness \(4(n+2)/(|F|−n−2)\), and privacy \(1/|F|\), for \(F\)-arithmetic circuits with \(n\) multiplication gates.

The main idea of the construction is simple. The prover wishes to generate a proof that will work with either of two instances \(x_1\) and \(x_2\). She simply generates a separate FLP proof for both instances \(x_1\) and \(x_2\), and randomly permutes the two proofs. To verify the combined proof against some \(x\), the verifier accepts iff either of the component proofs verifies against that \(x\).

Completeness and soundness of this construction are relatively clear. However, the construction is not necessarily zero-knowledge. While verifying the combined proof, we expect to verify a component proof against a proof that was generated for some other instance — e.g., verify a proof generated for \(x_1\) against \(x_2\). The standard zero-knowledge property of the underlying FLP does not apply to this situation. Indeed, since the Query function is linear, the result of querying a “mismatched” instance-proof pair will reveal “how far away” the instance is from the correct one.

We show that, when the underlying FLP is that of Boneh et al. [17], and extra randomness is introduced into the statement by means of the \text{Encode}(\Delta, \cdot) function, even the queries to the “mismatched"
Figure 11: Simulator and hybrids used in the proof of privacy for the VIDPF construction.

instance+proof can be simulated. Intuitively, the extra uncertainty of \( \Delta \) blinds the results of the problematic queries.

**Proof of Lemma 2.** Figure 12 describes a delayed-input FLP that uses a basic FLP as a building block. Our claims in this proof rely on that FLP being instantiated using the construction of [17], also used in the VDAF draft specification (see [9, Section 7.3]). We recall the relevant aspects of that construction below, as needed.

In Doplar, we will use our DFLP construction for the language \( \mathcal{L} = \{0, 1\} \) — i.e., we use it to prove that a value is zero or one. In this case, we instantiate the underlying FLP with the circuit:

\[
C((s, t, \Delta), r) = (r \cdot s(s - 1) + r^2 \cdot (s \cdot \Delta - t))^2,
\]

where \( r \) denotes the joint randomness. This circuit recognizes the set of inputs \( (s, t, \Delta) \in \{(0, 0, \Delta), (1, \Delta, \Delta)\} \). Note that FLP has input length \( n = 3 \) and joint-randomness length \( jl = 1 \); its circuit has 3 multiplication gates \((s(s - 1), s\Delta, \) and the outer square\). In the more general case, FLP will be instantiated for the language \( \{ (s, t, \Delta) | s \in \mathcal{L} \wedge s\Delta = t \} \). If the circuit for membership in \( \mathcal{L} \) has \( n \) multiplication gates, then FLP will be instantiated with a circuit with \( n + 2 \) multiplication gates.

Completeness follows immediately from the perfect completeness of the underlying FLP. The FLP of [17] has soundness \( 2n'/(|\mathcal{F}| - n') \) when its circuit has \( n' \) multiplication gates. We instantiate that FLP with \( n' = n + 2 \), and we also incur a factor 2 loss in soundness since our construction verifies two proofs in the underlying FLP. Hence, we obtain the soundness bound stated in the lemma.
so, we obtain the hybrid

\[ \Delta \]

Hence, we can apply the zero-knowledge property of the underlying FLP to each such expression. In doing this, \[ \Delta \] is random, and \[ \sigma \] simulator samples \[ \sigma \] adversary’s choice. Note that proof \[ \pi \] DF LP is the FLP for arithmetic circuits from [17].

Making this change of notation in the game yields hybrid \[ G_1 \] in Figure 13, gray boxes excluded and outlined box included). \[ G_1 \] is distributed identically to the original privacy game.

In each call to \[ \text{Query} \] in \[ G_1 \] that involves a value \[ \pi_i \], we use the same input that was used to generate \[ \pi_i \]. Hence, we can apply the zero-knowledge property of the underlying FLP to each such expression. In doing so, we obtain the hybrid \[ G_2 \] on the right of Figure 13. The underlying FLP of [17] has perfect zero-knowledge, so \[ G_2 \] is distributed identically to the original game.

To complete the proof, it suffices to show that \[ \sigma_u \] is distributed pseudorandomly in \[ \mathbb{F}^3 \times \{0\} \], since the simulator samples \[ \sigma_u \] uniformly from that set. In particular, when \( \sigma \) is distributed as in a simulated proof, \( \Delta \) is random, and \( d \neq 0 \), what is the distribution on \( \sigma + \text{Query}(\text{Encode}(\Delta, d), \tilde{0}, \tilde{0}, \cdots) \)?

To answer this question, we must use specific properties of the FLP from [17]. We first briefly review the main idea behind their proof. The prover defines two polynomials \( L \) and \( R \) such that, for each multiplication gate \( i \) in the verification circuit, the value on its left wire is \( L(i) \) and its right wire \( R(i) \). Additionally, \( L(0) \) and \( R(0) \) are chosen uniformly. Define the “gadget” polynomial \( G = L \times R \) — then \( G(i) \) is the value of the output wire of the \( i \)th gate.

The proof vector \( \pi \) then consists of \( L(0), R(0) \), and the coefficients of the \( G \) polynomial. With that in mind, the \( \text{Query} \) algorithm makes 4 linear queries to the input + proof vector:

1. Obtain evaluations of the polynomial \( L \) as follows:
   - \( L(0) \) is part of the proof vector.
   - For \( i > 0 \), if the left input to gate \( i \) is an input to the circuit, then \( L(i) \) is given as part of the proof input-instance, to which \( \text{Query} \) has access.
   - Otherwise, the left input to gate \( i \) is the output of some other multiplication gate \( j \). This value can be obtained as \( G(j) \), since the coefficients of \( G \) are included in the proof vector.

\[
\text{DFLP}^*.\text{Prove}(\vec{e}_1, \vec{e}_2, \Delta, jr): \begin{align*}
1. \quad (jr_1, jr_2) & \leftarrow jr \\
2. \quad \vec{e}_1 & \leftarrow \text{FLP}.\text{Encode}(\Delta, \vec{x}_1) \\
3. \quad \vec{e}_2 & \leftarrow \text{FLP}.\text{Encode}(\Delta, \vec{x}_2) \\
4. \quad b & \leftarrow \{1, 2\} \\
5. \quad \sigma_b & \leftarrow \text{Prove}(\vec{e}_1, \Delta, jr_b) \\
6. \quad \sigma_{3-b} & \leftarrow \text{Prove}(\vec{e}_2, \Delta, jr_{3-b}) \\
7. \quad \text{ret } (\sigma_1, \sigma_2) \\
\end{align*}
\]

\[
\text{DFLP}^*.\text{Decide}(\sigma): \begin{align*}
12. \quad (\sigma_1, \sigma_2) & \leftarrow \sigma \\
13. \quad \text{ret } \text{FLP}.\text{Decide}(\sigma_1) \\
14. \quad \lor \text{FLP}.\text{Decide}(\sigma_2) \\
\end{align*}
\]

\[
\text{DFLP}^*.\text{Encode}(\Delta \in \mathbb{F}, \vec{x} \in \mathbb{F}^n): \begin{align*}
15. \quad \text{for } i \in [n]: \\
16. \quad \vec{e}[i] & \leftarrow \vec{x}[i] \\
17. \quad \vec{e}[i + n] & \leftarrow \Delta \cdot \vec{x}[i] \\
18. \quad \text{ret } \vec{e} \\
\end{align*}
\]

\[
\text{DFLP}^*.\text{Decode}(\vec{e} \in \mathbb{F}^{2n}): \begin{align*}
19. \quad \text{ret } \vec{e}[1 : n] \\
\end{align*}
\]
Reconstruct $L$ as the result of Lagrange interpolation over the points \{(i, L(i))\}. Evaluate this polynomial $L$ at point $qr$ (the query randomness).

2. Similarly, reconstruct $R$ and evaluate it at point $qr$.

3. Evaluate the polynomial $G$ at point $qr$.

4. Evaluate $G$ at point $m$, where the output of verification circuit is the output wire of the $m$’th multiplication gate.

Suppose the results of these queries are $(r, s, t, u)$; the Decide algorithm checks that $t = rs$ and $u = 0$. The zero-knowledge property is that the result of the queries is distributed as $(r, s, rs, 0)$ for uniform $r, s \leftarrow \mathbb{F}$.

With Query as above, we now consider the distribution of $(r, s, rs, 0) + \text{Query}(\text{Encode}(\Delta, \mathbf{d}), \mathbf{0}, \mathbf{0}, \cdots)$, where $\Delta, r, s$ are uniform in $\mathbb{F}$.

- The first component of this expression is uniform due to $r$.
- With overwhelming probability $1 - 1/|\mathbb{F}|$ we have $r \neq 0$. Conditioned on $r \neq 0$, the third component of the expression is uniform, since it is masked with $rs$, and $s$ is uniform (even conditioned on the first component).
- Let $q_4$ be the 4th component of Query’s output in the above expression. By definition of Query, $q_4$ is the result of evaluating polynomial $G$ at point $qr$. But in this expression, the “proof vector” argument to Query is all zeroes, hence Query evaluates the all-zeroes polynomial and outputs $q_4 = 0$. Hence the 4th component of the overall expression is zero.
- Let $q_2$ be the second output of Query in the above expression. We see that $q_2$ is the result of evaluating polynomial $R$ at point $qr$, after reconstructing $R$ as described above. Fix a position $i$ in which $\mathbf{d}[i] \neq 0$. Then the $(n + i)$th position of $\mathbf{e} = \text{Encode}(\Delta, \mathbf{d})$ is $\mathbf{e}[i] = \mathbf{d}[i] \Delta$, and therefore is uniformly distributed when $\Delta$ is uniformly distributed.

The final multiplication gate in the verification circuit is the outermost square in (1). The input to this squaring operation is a linear combination that includes $\mathbf{e}[i]$. So as $\mathbf{e}[i]$ is uniformly distributed, the input to this multiplication gate is also uniformly distributed. Then the result of interpolating polynomial $R$ (based on $\mathbf{e}[i]$ among other values) and evaluating $R$ at $qr$ is also uniformly distributed. In other words, $q_2$ is uniformly distributed over uniform choice of $\Delta$, so the second component of the above expression is uniform.

Overall, we have shown that the distribution of $\sigma_u$ in $\mathbb{G}_2$ of Figure 13 is statistical distance $1/|\mathbb{F}|$ from the simulator’s distribution: uniform over $\mathbb{F}^3 \times \{0\}$. Hence in $\mathbb{G}_2$ the adversary has advantage bounded by $1/|\mathbb{F}|$ in $\mathbb{G}_2$. □

C Proofs of Theorems

C.1 Prio3 Robustness (Theorem 1)

We begin by instantiating the robustness game for $II$ in Figure 14. We expand the Prep algorithm and make a few simplifications to the game’s internal notation and bookkeeping. First, the game Exp$^{\text{robust}}$ calls for a VDAF with an arbitrary number of rounds, but Prio3 constructions has just one round. Second, we know that the Prep algorithm will be called exactly twice for each aggregator, and that the initial broadcast message and state are empty. We Therefore unroll the loop of lines 5–14 of Figure 3 and evaluate those if-statements whose values are pre-determined. Third, we replace table $St$ with a vector $\mathbf{s}t$ and remove table $Msg$ altogether. (The transcript output by the oracle is now constructed at the end on line 21 on the left-hand panel of Figure 14.) Fourth, we evaluate Prep in parallel for all aggregators instead of in sequence; the order of these operations does not affect their results because aggregators do not share state. Fifth, we
perform the deterministic Decide operation only once since its result is the same for all aggregators. Finally, we replace each call to $\text{RO}_i$ with a call to the corresponding random oracle $\text{RO}_i$. Let $q_i$ denote the number of queries $A$ makes to $\text{RO}_i$; note that $q_{RG} = q_1 + \cdots + q_7$.

We have also dropped winning condition on line 16 of Figure 3. By definition, $\Pi.\text{refineFromShares}(\hat{z}, \hat{x}) = \Pi.\text{Unshard}(1, \Pi.\text{Unshard}(1, (\Pi.\text{Agg}(\text{inp}[1]), \ldots, \Pi.\text{Agg}(\text{inp}[8])))$, where $\text{inp}[j]$ is the unpacked inner measurement share of input share $\hat{z}[j]$ for each $j$. Thus $w$ can never be set by forcing the refined shares to mismatch the expected refined measurement.

Now we express the proof with a series of incrementally changed games, beginning with $\text{G}_1$ (c.f. Figure 14). The joint randomness for each aggregator $j$ is derived in $\text{Exp}_I^{\text{robust}}$ from the seed $\text{jseed}_j$ of that aggregator, which is also the state $\hat{s}[j]$. In $\text{G}_1$, we instead derive joint randomness from $\hat{s}[1]$ for all aggregators, thus ensuring that the joint randomness is the same for everyone.

We build a wrapper adversary $B$ for which

$$\text{Adv}_I^{\text{robust}}(A) \leq \Pr[\text{G}_1(B)] + \frac{q_5}{2^k}. \quad (2)$$

Adversary $B$ only makes queries to $\text{Prep}$ that set $\hat{s}[j] = \hat{s}[1]$ for all $j$. It accomplishes this by calculating $\hat{s}[j]$ for every aggregator and $\text{Prep}$ query made by $A$. If it finds that $\hat{s}[j] = \hat{s}[1]$ for all aggregators, it forwards the query to its own $\text{Prep}$ oracle. Otherwise, it runs $\text{Prep}$ itself. $B$ can perfectly simulate $\text{Prep}$ except for line 9, because it does not know $sk$. Instead, $B$ picks its own verification key $sk'$ and uses $sk'$ in line 9 where $\text{Prep}$ would use $sk$. Adversary $A$ can detect the substitution of $sk'$ for $sk$ in two cases: If $A$ queries $\text{RO}_9$ on seed $sk'$; or if the $\text{Prep}$ oracle and $B$ query $\text{RO}_9$ on the same context string. The latter event does not occur because each query to $\text{RO}_9$ contains a unique nonce. The former occurs with probability at most $\frac{q_5}{2^k}$, because $sk'$ is a uniformly random $k$-bit string. The queries that $B$ simulates would always set

![Figure 13: Hybrids for zero-knowledge property of the delayed-2-input FLP construction.](image)
Figure 14: Left: Definition of game $G_1$ for the proof of Theorem 1. Also shown is the robustness game for $II$ and adversary $A$ with some simplifications applied. Right: Adversary $B$.

acc; $\leftarrow 0$ in line 17. Thus any query that would set $w \leftarrow \text{true}$ is forwarded to the $\text{Prep}$ oracle by $B$, and $B$ wins whenever $A$ does. The claim follows.

Next, we use the full linearity of $\text{FLP}$ to decompose $\text{FLP}.\text{Query}$ into algorithm $Q$ and a matrix multiplication operation, as shown in the left-hand panel of Figure 15. $Q$ is a randomized algorithm, but it is executed deterministically with fixed input $jr$ and coins $qr$. We may therefore call $Q$ only once to eliminate redundancy. Finally, we sum the vectors $\vec{x}_j || \vec{\pi}_j$ before the multiplication instead of multiplying then summing the products. This preserves the output thanks to the associativity of matrix multiplication.

Full linearity is an information theoretic property that holds unconditionally for all inputs, proofs, and coins, so the new game computes the same verifier string $vf$. Thus

$$\Pr[G_3(B)] = \Pr[G_2(B)].$$  

We produce the next modified game, in the right-hand panel of Figure 15, to define variables $\text{inp} = \sum_{j=1}^s \text{inp}[j]$ and $\vec{\pi} = \sum_{j=1}^s \vec{\pi}[j]$ and invoke $\text{PRG}.\text{Query}$ on $\text{inp}, \vec{\pi}$ directly. In addition, from Figure 5, we can see that $\text{inp} = II.\text{refineFromShares}(\varepsilon, \bar{x})$, so we substitute $\text{inp}$ into line 21. By the full linearity of $\text{FLP}$, we have that $Q(jr; qr) \cdot (\text{inp} \parallel \vec{\pi}) = \text{FLP}.\text{Query}((\text{inp}, \vec{\pi}, jr; qr)$. Again, these operations do not affect the adversary’s view of $\text{Prep}$, and

$$\Pr[G_2(B)] = \Pr[G_3(B)].$$  

In the next game, we replace the pseudorandom query randomness $qr$ with a fresh random string that is implicitly sampled by $\text{Query}$. We bound the difference in advantage between games $G_3$ and $G_1$ via a reduction $B'$ to the pseudorandomness of $\text{RG}_5$. The reduction honestly simulates $G_3$, except in line 11, where it queries its challenge oracle on $n$ and sets $qr$ to the response. Because every nonce is unique, these queries are all distinct. When the challenge oracle is a random function, this is a perfect simulation of $G_4$; otherwise it is a perfect simulation of $G_3$. Adversary $B'$ makes $q_{\text{prep}}$ queries to its challenge oracle; when $\text{RG}_5$ is modeled as a random oracle, there is a maximum of $q_s$ random oracle queries.

The generic PRF advantage for a $(q_s, q_{\text{prep}})$-query attacker against a random oracle with domain $\{0,1\}^n$ is bounded by the probability $\frac{q_s}{2^n}$ that the attacker makes a random oracle query containing $sk$. Thus
\[
\Pr[\mathcal{G}_2(B)] \leq \Pr[\mathcal{G}_4(B)] + \frac{q_5}{2^\kappa}
\]

Our next game (\(\mathcal{G}_5\) defined in the right-hand panel of Figure 16) differs from \(\mathcal{G}_4\) as follows. We set a \texttt{bad} flag and force the adversary to lose if it makes two queries to \texttt{Prep} which derive their joint randomness from the same seed. Each query to \texttt{Prep} derives its joint randomness seed from a unique nonce, so duplicate seeds require a collision between two queries to \(\text{RO}_0\) or between two vectors of hints. Both seeds and hints are randomly sampled by random oracles \(\text{RO}_0\) and \(\text{RO}_7\) respectively, so we limit the probability of both types of collision with a birthday bound over the \(q_{\text{Prep}}\) queries to \texttt{Prep}:

\[
\frac{q_{\text{Prep}}^2}{2^{2\kappa+1}} + \frac{q_{\text{Prep}}^2}{2^{2\kappa+1}} < \frac{q_{\text{Prep}}^2}{2^\kappa}.
\]

Since the games are identical until \texttt{bad} gets set, we have

\[
\Pr[\mathcal{G}_4(B)] \leq \Pr[\mathcal{G}_5(B)] + \frac{q_{\text{Prep}}^2}{2^\kappa}.
\]

We are now ready to reduce to FLP soundness. To do so, we construct a malicious prover \(P^*\) in Figure 17 from \(B\) whose advantage in the FLP soundness experiment is related to \(B\)'s advantage in winning game \(\mathcal{G}_5\). Recall from Figure 2 that the prover is called twice, first to choose an input and a second time to generate a proof. The prover is given joint randomness \(jr\) in this second call, after committing to the input. Thus, in our reduction we must extract this input from \(B\) random oracle queries, then program the random oracle with \(jr\) before proceeding.
The malicious prover $P^*$ runs $B$ in a simulation of $G_5$. Its random oracle queries are answered by lazily-evaluating a table Rand; all oracle queries are handled the same way except for a distinguished query, which will be programmed using the $jr$ string generated as part of the malicious prover’s experiment. At the start of the simulation, the prover $P^*$ samples $i^*\leftarrow [q_1 + q_{\text{rep}}]$. On the $i^*$ unique invocation of $\text{RO}_1$ (see ROExt$_1$ in Figure 17), the prover checks the table Rand for a nonce $n$ and input shares $i_{\text{inp}}$, which give rise to the seed $j\text{seed}$ provided as input. If successful, the prover’s output $i_{\text{inp}}$ as its challenge input, and sets Rand$[1,j\text{seed},\varepsilon] \leftarrow jr$ (line 11). It also records $n^* \leftarrow n$ for use later on.

The simulation of Prep queries is identical except after two events. First, the prover $P^*$ halts and concedes if two Prep queries generate the same joint randomness seed $st[1]$. (Adversary $B$ loses in this case.) Second, if $n = n^*$, then $P^*$ immediately halts and outputs the proof $\pi$ computed on line 30. If the simulation has reached this point, then the probability that $P^*$ wins its game is at least the probability that the game sets $w \leftarrow \text{true}$ on line 36. Conditioning on the probability that $P^*$ guesses the winning query to $\text{RO}_1$, we have that

$$\Pr \left[ G_5(B) \right] \leq (q_1 + q_{\text{rep}}) \cdot \epsilon. \quad (7)$$

The claimed bound follows by gathering up all of the bounds across the games and simplifying.

### C.2 Prio3 Privacy (Theorem 2)

We begin by instantiating the privacy game $\text{Exp}^{\text{priv}}_{P,\Pi}$ for Prio3 VDAF $\Pi$. Game $G_6$ in Figure 18 was constructed by inlining $\Pi$’s constituent algorithms and cleaning up the control flow. In addition, calls to $\text{RG}_1$ have been substituted with calls to a random oracle $\text{RO}_1$. Let $q_i$ denote the number of queries $A$ makes to $\text{RO}_1$; note that $q_{\text{rep}} = q_1 + \cdots + q_7$.

In our first game hop, we modify $\text{Shard}$ oracle’s behavior after setting flag $\text{bad}_1$ on line 7. In the new game, $G_1$ (Figure 18), the nonce $n$ is sampled without replacement, ensuring that each nonce is used is unique.
Adversary $P^*[B](\cdot)$:
1. $w \leftarrow \text{false}; \ sk \leftarrow \{0,1\}^n$; bad $\leftarrow \text{false}; \ J \leftarrow \emptyset$
2. $ctr \leftarrow 0; \ n^* \leftarrow \downarrow; \ i^* \leftarrow [q_1 + q_{\text{mp}}]$
3. $B^\text{ROExt}; \ RO_2; \ldots; \ RO_{\text{PrepSim}}()$; ret $w$

$\text{ROExt}_i(\text{seed},\ cntxt)$:
4. if Rand[$1$, seed, cntxt] $\neq \downarrow$: ret $\text{RO}_i(\text{seed},\ cntxt)$
5. $ctr \leftarrow ctr + 1$
6. if $ctr = i^* \land (\exists\ n, (\text{blind}_j, \text{inp}_j, \rho_j) \in [n])$
7. $(\forall j)$ Rand[$7, \text{blind}_j, j||n||\text{inp}_j = \rho_j$
8. $\land$ Rand[$6, 0^\circ, (\rho_1, \ldots, \rho_s)] = \text{seed}$
9. output $\text{inp}_1 + \cdots + \text{inp}_s$ and wait for $jr$.
10. $n^* \leftarrow n$; Rand[$1, \text{seed},\ cntxt$] $\leftarrow jr$
11. ret $\text{RO}_i(\text{seed},\ cntxt)$

$\text{RO}_i(\text{seed},\ cntxt)$:
12. $l \leftarrow (j_i, n, m, pl, \text{fl})$
13. if Rand[$i, \text{seed},\ cntxt$] $= \downarrow$
14. if $i \leq 5$: Rand[$i, \text{seed},\ cntxt$] $\leftarrow P[i][]$
15. else: Rand[$i, \text{seed},\ cntxt$] $\leftarrow \{0,1\}^n$
16. ret Rand[$i, \text{seed},\ cntxt$]

PrepSim($n, \vec{x}, msg_{\text{init}}, st_{\text{init}}$):
17. if Used[$n$] $\neq \downarrow$: ret $\downarrow$
18. Used[$n$] $\leftarrow T$
19. for $j \in [n]$
20. $(\text{inp}_j, \vec{\pi}[j], \text{blind}) \leftarrow \text{Unpack}(j, \vec{x}[j])$
21. $(\vec{\rho}_j) \leftarrow \\text{msg}; \, \vec{\rho}[j] \leftarrow \text{RO}_j(\text{blind}, j||n||\text{inp}_j)$
22. $\text{rseed}[j] \leftarrow \vec{\rho}[j]$
23. $\vec{s}[j] \leftarrow \text{RO}_n(\text{t}, \vec{\rho})$ // joint rand seed
24. if $\vec{s}[1] \in J$: bad $\leftarrow \text{true};$ halt.
25. $J \leftarrow J \cup \{ \vec{s}[1] \}$
26. $jr \leftarrow \text{ROExt}_i(\vec{s}[1], \varepsilon)$
27. $\text{inp} \leftarrow \bigoplus_{j=1}^{\infty} \text{inp}_j[\pi] \leftarrow \bigoplus_{j=1}^{\infty} \vec{\pi}[j]$.
28. if $n = n^*$: output $\pi$ and halt.
29. $vf \leftarrow \text{FLP}.\text{Query}(\text{inp}, \pi, jr)$
30. $d \leftarrow \text{FLP}.\text{Decide}(vf)$
31. for $j \in [s]$
32. $\text{jseed}_j \leftarrow \vec{s}[j]; \text{jseed}_j \leftarrow \text{RO}_n(\text{t}, \text{rseed})$
33. $\text{acc}_j \leftarrow d \land [\text{[jseed}_j = \text{jseed}_j]]$
34. $w \leftarrow (w \lor \lnot \text{bad} \land \text{acc}_j \land \text{inp} \notin L)$
35. ret $(w, (msg_{\text{init}}, (vf[s][j], \text{rseed}[j]))_{j \in [s]}$)

Figure 17: Malicious prover $P^*$ for the proof of Theorem 1. The lookup in the random oracle table Rand on lines 6–8 can performed efficiently by creating a reverse-lookup table; we omit the details for brevity.

Applying the Fundamental Lemma of Game Playing [14], and using a birthday bound for the probability of bad$_1$ getting set,

$$\Pr[\mathcal{G}_0(A)] \leq \Pr[\mathcal{G}_1(A)] + \frac{q_{\text{hard}}^2}{|N|}. \quad (8)$$

Next we replace the adversary $A$ with one that controls all but one aggregator. We construct such an adversary $B$ as a wrapper around $A$, and show that $B$ wins with at least the probability of $A$. The adversary $B$, defined in Figure 19, presents four oracles ShardSim, SetupSim, PrepSim, and AggSim to adversary $A$, each emulating an oracle in game $\mathcal{G}_1$. Algorithms SetupSim, PrepSim, AggSim are computed by $B$ just as the respective oracles in game $\mathcal{G}_1$, except that queries pertaining to aggregator $z$ are forwarded to $B$’s own oracles. Algorithm ShardSim forwards $A$’s query to Shard in the natural way, but returns the shares of the aggregators deemed honest by $A$.

We claim that $B$ perfectly simulates $\mathcal{G}_1(A)$. This is obvious for ShardSim and for queries for which $\hat{z} = z$; in these cases, $B$ simply forwards its queries to the appropriate oracles without changing their inputs. The only difference is that Shard returns more input shares than $A$ requests; $B$ stores these extra input shares for its own use and does not reveal them. By construction, this is a subset of the input shares returned by the query.

When $A$ makes queries to SetupSim, PrepSim, or AggSim with $\hat{z} \in V \setminus \{z\}$, our wrapper adversary performs the operations of Setup, Prep, or Agg respectively. Effectively, adversary $B$ uses its stored input shares to fill in entries of tables In, Batch, Setup, Status, St, and Out exactly as the real privacy game would. Since each entry is disambiguated by its $\hat{z}$, there is no overlap with the tables maintained by the game; every table entry read by $B$ must first have been written by $B$ and thus all the information it needs to simulate the game perfectly is accessible. It follows that

$$\Pr[\mathcal{G}_1(A)] = \Pr[\mathcal{G}_1(B)]. \quad (9)$$

In the next game hop (Figure 20) we make some simplifying changes, including cleaning up the bad$_1$ flag and substituting $\{z\}$ for $V$ and simplifying accordingly. (We do not highlight this change in Figure 20, as it is fairly straightforward.) We also make the following breaking change: In game $\mathcal{G}_2$, we program the
<table>
<thead>
<tr>
<th>Game (_{\mathcal{G}_0}(A))</th>
<th>Game (_{\mathcal{G}_1}(A))</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\mathcal{S}<em>{\mathcal{A}}, \mathcal{V}, (sk_j)</em>{j \in \mathcal{V}}) \leftarrow \text{A}^{\text{RO}_0}; T \leftarrow [s] \setminus \mathcal{V})</td>
<td>((\mathcal{S}<em>{\mathcal{A}}, \mathcal{V}, (sk_j)</em>{j \in \mathcal{V}}) \leftarrow \text{A}^{\text{RO}_0}; T \leftarrow [s] \setminus \mathcal{V})</td>
</tr>
<tr>
<td>if (</td>
<td>\mathcal{V}</td>
</tr>
<tr>
<td>(b \leftarrow {0,1}; \ b' \leftarrow A^{\text{RO}_0} \text{Shard} \text{Setup} \text{Prep} \text{Agg}(st_A))</td>
<td>(\mathcal{S}_{\mathcal{A}} \land \mathcal{V} = \emptyset)</td>
</tr>
<tr>
<td>ret (\hat{b}')</td>
<td>(\mathcal{S}_{\mathcal{A}} \land \mathcal{V} = \emptyset)</td>
</tr>
</tbody>
</table>

Shard \((\hat{k} \in \mathbb{N}, m_0, m_1 \in \mathcal{T})\):

- if \(\text{Used}[\hat{k}] \neq \bot\) ret ⊥
- \(\hat{n} \leftarrow \mathcal{N}\)
- if \(n \in \mathcal{N}'\) bad \(_1\) true: \(\hat{n} \leftarrow \mathcal{N} \setminus \mathcal{N}'\)
- \(\mathcal{N}' \leftarrow \mathcal{N}' \cup \{n\}\)
- \(\text{inp} \leftarrow \text{Encode}(m_0)\)
- for \(j \in [2..s]\):
  - \(\text{blind}_j, \text{xseed}_j, \text{psseed}_j \leftarrow \{0,1\}^n\)
  - \(\vec{x}[j] \leftarrow \text{RO}_2(\text{xseed}_j, j)\)
  - \(\text{rseed}[j] \leftarrow \text{RO}_2(\text{blind}_j, j, n \parallel \vec{x}[j])\)
  - \(\vec{x}[1] \leftarrow \text{inp} - \sum_{j=2}^{s} \vec{x}[j]\)
  - \(\text{blind}_1 \leftarrow \{0,1\}^k; \text{ps} \leftarrow \{0,1\}^s\)
  - \(\text{rseed}[1] \leftarrow \text{RO}_2(\text{blind}_1, 1, n \parallel \vec{x}[1])\)
  - \(\text{jseed} \leftarrow \text{RO}_1(0^n, \text{rseed}[1])\)
- \(\text{ps} \leftarrow \text{RO}_1(\text{ps}, \epsilon)\)
- \(\vec{\pi}[1] \leftarrow \text{Prove(inp, pr,}\)\( \vec{x}\)\)
- \(\vec{\pi}[1] \leftarrow \vec{\pi}[1] - \sum_{j=2}^{s} \text{RO}_2(\text{psseed}_j, j)\)
- \(\vec{x}[1] \leftarrow (\vec{x}[1], \vec{\pi}[1], \text{blind}_1)\)
- for \(j \in [2..s]\):
  - \(\vec{x}[j] \leftarrow (\text{xseed}_j, \text{psseed}_j, \text{blind}_j)\)
  - \(\text{Pub}[\hat{k}] \leftarrow \text{rseed}; \text{In}[\hat{k}, \cdot] \leftarrow \vec{x}\)
  - \(\text{Used}[\hat{k}] \leftarrow (n, m_0, m_1)\)
  - ret \((n, \text{Pub}[\hat{k}], [\text{In}[\hat{k}, \cdot]]) \in \mathcal{E}\) |

Setup \((\hat{i} \in \mathbb{N}, \hat{j} \in \mathcal{V}, \hat{\text{stat}}_{\text{Init}} \in \{\epsilon\})\):

- if \(\text{Status}[\hat{i}, \hat{j}] \neq \bot\) or \(|\text{Setup}[\hat{i}, \hat{j}]| > 0\): ret ⊥
- \(\text{Setup}[\hat{i}, \hat{j}] \leftarrow \text{stat}_{\text{Init}}\)
- \(\text{Status}[\hat{i}, \hat{j}] \leftarrow \text{running}\)

Preg \((\hat{i} \in \mathbb{N}, \hat{j} \in \mathcal{V}, \hat{k} \in \mathbb{N}, m\hat{sg} \in \mathcal{M}^*)\):

- if \(\text{Status}[\hat{i}, \hat{j}] \neq \text{running}\) or \(\text{In}[\hat{k}, \hat{j}] = \bot\):
  - ret ⊥
- if \(\text{Status}[\hat{i}, \hat{j}] = \bot\):
  - \(\text{St}[\hat{i}, \hat{j}, \hat{k}] \leftarrow \text{Setup}[\hat{i}, \hat{j}]\)
  - \(\text{msg} \leftarrow (\text{Pub}[\hat{k}], \cdot)\)
  - \((n, m_0, m_1) \leftarrow \text{Used}[\hat{k}]\)
- if \(\text{Status}[\hat{i}, \hat{j}] = \epsilon\):
  - \(\text{inp} \leftarrow \text{Unpack}(\hat{j}, \text{In}[\hat{k}, \hat{j}]\)
  - \(\text{rseed}, \epsilon \leftarrow \text{msg}\)
  - \(\text{rseed}[\hat{j}] \leftarrow \text{RO}_2(\text{blind}, \hat{j} \parallel n \parallel \text{inp})\)
  - \(\text{jseed} \leftarrow \text{RO}_3(0^n, \text{jseed}); \text{jr} \leftarrow \text{RO}_3(\text{jseed}, \epsilon)\)
  - \(\text{qr} \leftarrow \text{RO}_3(\text{sk}_j, n)\)
  - \(\text{msg} \leftarrow (\text{Query}(\text{inp}, \pi, \text{jr}, \text{qr}), \text{rseed}[\hat{j}]\)
  - \(\text{St}[\hat{i}, \hat{j}, \hat{k}] \leftarrow (\text{jseed}, \text{Truncate}(\text{in}))\)
  - ret (\text{running}, \text{msg})\)

// Process broadcast messages from aggregators

- \((\text{jseed}, \text{y}) \leftarrow \text{St}[\hat{i}, \hat{j}, \hat{k}]\)
- \((\text{vfs}[\hat{j}], \text{rseed}[\hat{j}])_{\hat{j} \in [\hat{i}]} \leftarrow \text{msg}\)
- \(\text{acc} \leftarrow \text{Decide}(\sum_{\hat{j}=1}^{\hat{i}} \text{vfs}[\hat{j}]\)
- \(\text{St}[\hat{i}, \hat{j}, \hat{k}] \leftarrow \bot\)

if \(\text{acc} = 0\) or \(\text{jseed} \neq \text{RO}_3(0^n, \text{rseed})\):

- ret (\text{failed}, \bot)
- \(\text{Out}[\hat{i}, \hat{j}, \hat{k}] \leftarrow \text{y}\)
- \(\text{Batch}_0[\hat{i}, \hat{j}, \hat{k}] \leftarrow m_0\)
- \(\text{Batch}_1[\hat{i}, \hat{j}, \hat{k}] \leftarrow m_1\)
- ret (\text{finished}, \bot)

Agg \((\hat{i} \in \mathbb{N}, \hat{j} \in \mathcal{V})\):

- if \(\text{Status}[\hat{i}, \hat{j}] \neq \text{running}\): ret ⊥
- if \(F(\text{Batch}_0[\hat{i}, \hat{j}, \hat{k}]) \neq F(\text{Batch}_0[\hat{i}, \hat{j}, \hat{k}])\) or \((\forall j', j' \in \mathcal{V}) sk_j = sk_{j'}\): ret ⊥
- \(\text{Status}[\hat{i}, \hat{j}] \leftarrow \text{finished}\)
- \(\text{y} \leftarrow \text{Out}[\hat{i}, \hat{j}, \cdot]\)
- ret \(\sum_{i=1}^{\hat{i}} y[i]\)

\(\text{RO}_3(\text{seed}, \text{ctxt})\):

- \(l \leftarrow (\{l, m, m, pl, ql\})\)
- if \(\text{Rand}[\hat{i}, \text{seed}, \text{ctxt}] = \bot\):
  - if \(i \leq 5\): \(\text{Rand}[\hat{i}, \text{seed}, \text{ctxt}] \leftarrow \{0,1\}^s\)
  - else: \(\text{Rand}[\hat{i}, \text{seed}, \text{ctxt}] \leftarrow \{0,1\}^s\)
- ret \(\text{Rand}[\hat{i}, \text{seed}, \text{ctxt}]\)

Figure 18: Games \(_{\mathcal{G}_0}\) and \(_{\mathcal{G}_1}\) for the proof of Theorem 2. This game is identical to the privacy game for \(\Pi\), except the Shard, Prep, and Agg algorithms have been inlined. Algorithm Unpack is as defined in Figure 5. The random oracles \(_{\mathcal{RO}_0}\) are lazy-evaluated in a table Rand.
In particular, if ever Game \((14)\) to show that overwritten. Game evaluated by \(\text{table Rand with values chosen by the Shard oracle for the joint randomness, prover randomness, and query randomness. Accordingly, we pass these joint randomness and query randomness to the honest aggregator via its input share \((\text{In}[k, z]; \text{see line 29). This is to simplify bookkeeping in the next step. Game } \mathcal{G}_3 \text{ is identical to game } \mathcal{G}_1 \text{ until programming Rand overwrites an already existing value on line 18, 19, 20, or 21.}

- Line 18: Adversary \(B\) either has to guess \(jseed\) or guess the input to \(\text{RO}_6\) used to derive it. For the latter it must must guess \(rseed\) all of the corresponding inputs to \(\text{RO}_2\), which include the blinds generated by oracle \text{Shard}. Taking union bound over all the queries to \text{Shard}, the game overwrites Rand at this point with probability at most \(q_4 q_{\text{Shard}}/2^\kappa + (q_6 + q_7)q_{\text{Shard}}/(2^{s_\kappa}).\)

- Line 19: \(B\) must guess the \(ps\) generated by oracle \text{Shard}, so the game overwrites the table with probability at most \(q_4 q_{\text{Shard}}/2^\kappa\).

- Line 20: \(B\) must guess the nonce \(n\) generated by the oracle. The game overwrites the table here with probability at most \(q_5 q_{\text{Shard}}/|\mathcal{V}|\).

- Line 21: \(B\) must guess \(rseed\) or all of the corresponding inputs to \(\text{RO}_7\), so the game overwrites the table with probability at most \((q_6 + q_7)q_{\text{Shard}}/(2^{s_\kappa}).\)

We bound the probability of \(B\) distinguishing between these games by the probability that any one of these events occurs. Gathering up the terms yields

\[
\Pr[\mathcal{G}_1(B)] \leq \Pr[\mathcal{G}_2(B)] + \frac{(q_1 + q_4)q_{\text{Shard}}}{2^\kappa} + \frac{(q_6 + q_7)q_{\text{Shard}}}{2^{s_\kappa}} + \frac{q_5 q_{\text{Shard}}}{|\mathcal{V}|}. \tag{11}
\]

In the next game hop (see \(\mathcal{G}_3\) in the left panel of Figure 21) we prepare to ensure that all of the input shares \(\vec{x}\), proof shares \(\vec{\pi}\), and the public share \(rseed\) sampled by the \text{Shard} oracle are uniform random. We do so by sampling these values prior to processing \(m_b\) and programming the random oracle with the sample values (lines 3–12) so long as doing so does overwrite existing values (see procedure PO on lines 37–40). The game sets a flag \(\text{bad}_4\) if \text{Shard} would have overwritten an existing value. This does not change the adversary’s view of the experiment, so

\[
\Pr[\mathcal{G}_3(B)] = \Pr[\mathcal{G}_3(B)]. \tag{12}
\]

Next, in game \(\mathcal{G}_4\) (top-right panel of Figure 21) we change oracle \text{Shard’s} behavior after \(\text{bad}_4\) gets set. In particular, if ever PO is called on an input \((i, seed, cntxt, out)\) for which Rand\([i, seed, cntxt]\), the value is overwritten. Game \(\mathcal{G}_4\) is identical to game \(\mathcal{G}_3\) until \(\text{bad}_4\) gets set. Then we apply the Fundamental Lemma of Game Playing [14] to show that

\[
\Pr[\mathcal{G}_4(B)] \leq \Pr[\mathcal{G}_4(B)] + \Pr[\mathcal{G}_4(B) \text{ sets } \text{bad}_4] \leq \Pr[\mathcal{G}_4(B)] + \frac{(s - 1)(q_2 + q_4) + s(q_7))q_{\text{Shard}}}{2^\kappa}. \tag{14}
\]
Figure 20: Game $G_2$ for the proof of Theorem 2.
The probability that $B$ sets the bad flag in Game $G_4$ is the probability that $B$ makes a random oracle query that gets overwritten on line 9, 10, 11, or 12. On each line, the random oracle is programmed with a uniform random string sampled by the oracle prior to being revealed to the adversary. Rolling out the for-loop on line 8 and taking a union bound over all Shard queries yields the claimed bound.

Next, in game $G_5$ (bottom-right panel of Figure 21) we simplify the Shard oracle by inlining calls to PO and replacing invocations of RO with corresponding value generated by the oracle. These changes do not change the view of the adversary, so

$$
\Pr[\text{G}_5(B)] = \Pr[\text{G}_6(B)].
$$

(15)

Up to this point, we have constructed the “leader” input and proof shares differently than all of the other shares: we pick all other shares randomly, then set $\vec{x}[1] = \text{inp} - \sum_{j=2}^s \vec{x}[j]$ and $\pi[1] = \pi - \sum_{j=2}^s \pi[j]$. In our next game, we instead sample the “leader” shares randomly and compute the shares of the honest aggregator $z$ in a distinguished manner: $\vec{x}[z] = \text{inp} - \sum_{j \in T} \vec{x}[j]$ and $\pi[z] = \pi - \sum_{j \in T} \pi[j]$, where $T = [s]\setminus\{z\}$. If $z = 1$, this changes nothing. Otherwise, consider that in $G_6$, we have

$$
\vec{x}[1] = \text{inp} - \sum_{j=2}^s \vec{x}[j] = \text{inp} - \left(\sum_{j \in T} \vec{x}[j] - \vec{x}[z] + \vec{x}[1]\right).
$$

If we add $\vec{x}[z] - \vec{x}[1]$ to both sides of this equation, we can see that in $G_4$, it was already true that $\vec{x}[z] = \text{inp} - \sum_{j \in T} \vec{x}[j]$. The same holds true for $\pi[z]$ by an analogous argument. Therefore the distributions of aggregators’ 1 and $z$’s input and proof shares are unchanged between $G_4$ and $G_5$, and we have

$$
\Pr[\text{G}_5(B)] = \Pr[\text{G}_6(B)].
$$

(16)

In our next game ($G_7$, defined in the right panel of Figure 22) we run the query algorithm for aggregator $z$ in the Shard oracle and only send the result to Prep. The adversary cannot detect the timing of when this algorithm is run, so we have

$$
\Pr[\text{G}_6(B)] = \Pr[\text{G}_7(B)].
$$

(17)

In the next game ($G_8$, defined in the left panel of Figure 23) we run ViewFLP (as defined in Section 2) on input $\text{inp}$ to get $jr$, $qr$, and a verifier $\sigma$ and use these to compute Shard’s output. We have defined $\vec{x}[z] = \text{inp} - \sum_{j \in T} \vec{x}[j]$ and $\vec{x}[z] = \pi - \sum_{j \in T} \pi[j]$. Using the full linearity of FLP, we can the honest aggregator $z$’s verifier share $vfs$ in terms of $jr$, $qr$, $\sigma$ and the corrupt aggregators’ shares, since:

$$
\text{Query}(\text{inp}, \pi, jr; qr) = \text{Query}(\vec{x}[z], \pi[z], jr; qr)
$$

(18)

$$
+ \sum_{j \in T} \text{Query}(\vec{x}[j], \pi[j], jr; qr)
$$

(19)

This revision to the game does not change the outcome of the experiment. However, since we do not have access to the prover randomness generated by ViewFLP (inp), we can no longer consistently program the random oracle (see 19). Fortunately, to trigger this inconsistency, the adversary would have to guess the seed $ps$ used to to program it prior to calling Shard. It follows that

$$
\Pr[\text{G}_7(B)] \leq \Pr[\text{G}_8(B)] + \frac{q_4/\text{Shard}}{2^\kappa}.
$$

(20)

Let $S$ be the simulator hypothesized by $\delta$-privacy of FLP. In the right panel of Figure 23 we define a series of hybrid games that replace ViewFLP with a simulator $S$ for the privacy of FLP. Recall from Section 2 that $S$ outputs a string $jr \parallel qr \parallel \sigma$. In $G_9(B)$, the first $i-1$ queries to Shard generate $jr$, $qr$, $\sigma$ by calling $S()$; the remaining queries call ViewFLP instead. This means that $G_9$ is identical to $G_8$, so

$$
\Pr[\text{G}_8(B)] = \Pr[\text{G}_9(B)].
$$

(21)

For every $v \in \mathbb{F}_2^{l_2 \times q^3 \times n}$, we let $p_{i,v}$ denote the probability that $B$ wins hybrid $G_9$, conditioned on the event that the $i$th query to Shard sets $v = jr \parallel qr \parallel \sigma$. A union bound over all $v$ shows that

$$
\Pr[\text{G}_9(B)] = \sum_{v \in \mathbb{F}_2^{l_2 \times q^3 \times n}} p_{i,v}.
$$

(22)
Shard($k \in \mathbb{N}, m_0, m_1 \in T$):

1. if $\text{Used}[\hat{k}] \neq \bot$: ret $\bot$
2. $n \leftarrow N \setminus N^*; N^* \leftarrow N^* \cup \{n\}$
3. $\tilde{x} \leftarrow (F^n)^s; \tilde{\pi} \leftarrow (F^n)^s$
4. $(\text{blind}_1, \ldots, \text{blind}_s) \leftarrow \{(0, 1)^n\}^s$
5. $(\text{xseed}_2, \ldots, \text{xseed}_s) \leftarrow \{(0, 1)^n\}^{s-1}$
6. $(\text{pseed}_2, \ldots, \text{pseed}_s) \leftarrow \{(0, 1)^n\}^{s-1}$
7. $\text{rseed} \leftarrow \{(0, 1)^n\}^s$
8. for $j \in [s]$:
9. $\text{PO}_2(\text{xseed}_j, j, \tilde{x}[j])$
10. $\text{PO}_3(\text{pseed}_j, j, \tilde{\pi}[j])$
11. $\text{PO}_4(\text{blind}_j, j \parallel n \parallel \tilde{x}[j], \text{rseed}[j])$
12. $\text{PO}_5(\text{blind}_1, 1 \parallel n \parallel \tilde{x}[1], \text{rseed}[1])$
13. $\text{inp} \leftarrow \text{Encode}(m_0)$
14. for $j \in [2..s]$:
15. $\text{blind}_j, \text{xseed}_j, \text{pseed}_j \leftarrow \{(0, 1)^n\}^s$
16. $\tilde{x}[j] \leftarrow \text{RO}_2(\text{xseed}_j, j)$
17. $\text{rseed}[j] \leftarrow \text{RO}_2(\text{blind}_j, j \parallel n \parallel \tilde{x}[j])$
18. $\tilde{x}[1] \leftarrow \text{inp} \leftarrow \sum_{j=2}^{s} \tilde{x}[j]$
19. $\text{blind}_1 \leftarrow \{(0, 1)^n\}; \text{ps} \leftarrow \{(0, 1)^n\}^s$
20. $\text{rseed}[1] \leftarrow \text{RO}_2(\text{blind}_1, 1 \parallel n \parallel \tilde{x}[1])$
21. $\text{jseed} \leftarrow \{(0, 1)^n\}^s$
22. $\text{jr} \leftarrow F^d; \text{pr} \leftarrow F^d; \text{qr} \leftarrow F^d$
23. Rand[1, $\text{seed}, \varepsilon] \leftarrow \text{jr}$
24. Rand[4, $\text{ps}, \varepsilon] \leftarrow \text{pr}$
25. Rand[5, $\text{sk}_z, n] \leftarrow \text{qr}$
26. Rand[6, 0, $\text{rseed}] \leftarrow \text{jseed}$
27. $\tilde{\pi}[1] \leftarrow \text{Prove}(\text{inp}, \text{jr}; \text{pr})$
28. $\tilde{\pi}[1] \leftarrow \sum_{j=2}^{s} \text{RO}_2(\text{pseed}_j, j)$
29. $\tilde{x}[1] \leftarrow (\tilde{x}[1], \tilde{\pi}[1], \text{blind}_1)$
30. for $j \in [2..s]$:
31. $\tilde{x}[j] \leftarrow (\text{rseed}_j, \text{pseed}_j, \text{blind}_j)$
32. Pub[$\hat{k}] \leftarrow \text{rseed}$
33. In[$\hat{k}, \tilde{x}]
34. In[\hat{k}, \tilde{z}] \leftarrow (\tilde{x}[z], \text{seed}, \text{jr}, \text{qr})$
35. Used[\hat{k}] \leftarrow (n, m_0, m_1)$
36. ret $(n, \text{Pub}[\hat{k}], (\text{In}[\hat{k}, \tilde{z}] \downarrow r)_{r \in T})$

Algorithm PO$_i$(seed, cntxt, out):
37. if Rand[i, $\text{seed}, \text{cntxt}] = \bot$
38. Rand[i, $\text{seed}, \text{cntxt}] \leftarrow \text{out}$
39. else:
40. $\text{bad}_i \leftarrow \text{true}$

Algorithm PO$_i$(seed, cntxt, out):
1. if Rand[i, $\text{seed}, \text{cntxt}] = \bot$
2. Rand[i, $\text{seed}, \text{cntxt}] \leftarrow \text{out}$
3. else:
4. $\text{bad}_i \leftarrow \text{true}$

Figure 21: Games G3 (left), G4 (top-right), and G5 (bottom-right) for the proof of Theorem 2. Only the Shard is shown, as this is the only object that changes in each game hop.
Figure 22: Game $G_6$ (left) and game $G_7$ (right) for the proof of Theorem 2.
We are now ready to bound the quantity $\Pr[\mathcal{G}_0^{i+1}(B)] - \Pr[\mathcal{G}_0^i(B)]$. The two games $\mathcal{G}_0^{i+1}$ and $\mathcal{G}_0^i$ differ only in the tuple $v$ chosen by of the $(i + 1)$th query to $\text{Shard}$: the former calls $\text{ViewFLP}$ and the latter calls $S$. We therefore decompose both probabilities over the possible choices of $v$, and substitute in the statement

$$\sum_{v \in \mathcal{F}^i \times \mathcal{F}^i \times \mathcal{F}^i} |\Pr[\text{ViewFLP}(\text{inp}) = v] - \Pr[S() = v]| \leq \delta$$

that follows from the $\delta$-privacy of FLP for all $\text{inp}$. Since $p_{i,v} \leq 1$ for all $i$ and $v$ and $\Pr[\text{ViewFLP}(\text{inp}) =...
\[ \Pr[G_{\Pi}^{+1}(B)] - \Pr[G_{\Pi}(B)] = \sum_v p_{i,v} \cdot \Pr[View_{\Pi}(inp) = v] - \sum_v p_{i,v} \cdot \Pr[S() = v] \]
\[ = \sum_v p_{i,v} \cdot (\Pr[View_{\Pi}(inp) = v] - \Pr[S() = v]) \]
\[ \leq \sum_v |\Pr[View_{\Pi}(inp) = v] - \Pr[S() = v]| \leq \delta. \]

A union bound over all \( i \in [q_{\text{Shard}}] \) produces the final inequality:
\[ \Pr[G_{\Pi}^{\text{Shard}}(B)] - \Pr[G_{\Pi}(B)] \leq \delta \cdot q_{\text{Shard}}. \tag{23} \]

Finally, we observe that game \( G_{\Pi}^{\text{Shard}} \) can now be rewritten so that the outcome is independent of the challenge bit \( b \). Hence
\[ \Pr[G_{\Pi}^{\text{Shard}}(B)] = \frac{1}{2}. \tag{24} \]
Collecting bounds across all games and simplifying yields the theorem.

\[ \square \]

### C.3 Doplar Robustness (Theorem 3)

The proof is by a game-playing argument. We begin with the game \( G_0 \) defined in Figure 24 played by the given adversary \( A \). This game was constructed from \( \text{Exp}_{\Pi}^{\text{robust}}(A) \) by applying the following revisions. First, we have replaced \( \text{Prep} \) with its implementation, rolled out the loops in the \( \text{Prep} \) oracle, and simplified some of the control flow. Second, we have removed the call to \( \text{refineFromShares} \) and set the purported refined measurement with the sum of the refined shares output by the calls to \( \text{VIDPF.VEval} \). (This is equivalent by refinement consistency of \( \Pi \).) Third, we use the fact that the allowed-state validSt algorithm for \( \Pi \) only permits \( \text{Prep} \) queries with unique \((n, l)\) pairs to make the contents of table \text{Used} more explicit. Finally, we lazy-evaluate each random oracle, denoted \( \text{RO}_i \), with a table \text{Rand}. We use \( \text{RO}^\prime \) to denote the random oracle for \( \text{VIDPF} \). By construction we have that
\[ \text{Adv}_{\Pi}^{\text{robust}}(A) = \Pr[ G_0(A) ]. \tag{25} \]
In the remainder, we let \( q_i \) denote the number queries \( A \) makes to \( \text{RO}_i \) and \( q_i \) denote the number of queries \( A \) makes to \( \text{RO}^\prime \); note that \( q_{\text{RO}} = q_1 + \cdots + q_6 + q' \).

Similar to the proof of Theorem 1, note that we have dropped the winning condition on line 16 of the robustness game (Figure 3). The refined measurement computed from the input shares is equal to \( \Pi.\text{Unshard}(1, (\Pi.\text{Agg}(y_1), \Pi.\text{Agg}(y_2))) = y_1 + y_2 \), so this condition is never met by definition.

Next, in game \( G_1 \) (left panel of Figure 24) we revise the definition of the \( \text{RO} \) oracle so that for each \( i \in \{5, 6\} \), the values of \( \text{Rand}[i, \text{seed}, \text{ctxt}] \) are sampled without replacement. The new game is identical to \( G_0 \) up to a collision in the output for either \( \text{Rand}[5, \cdot, \cdot] \) or \( \text{Rand}[6, \cdot, \cdot] \). Applying a birthday bound over all queries by \( A \) or by the \( \text{Prep} \) oracle yields
\[ \Pr[ G_0(A) ] \leq \Pr[ G_1(A) ] + \frac{(q_5 + 2q_{\text{rep}})^2}{2^{k+1}} + \frac{(q_6 + 3q_{\text{rep}})^2}{2^{k+1}}. \tag{26} \]

Next, in game \( G_2 \) (right panel of Figure 25) we simplify the \( \text{Prep} \) oracle by substituting aggregator \( j \)'s local computation of the joint randomness seed \( jseed_j \) with a direct computation of the seed \( jseed_j \) from the parts \( \rho_1, \rho_2 \) computed on lines 9–10. Accordingly, We simplify the joint local randomness checks (lines 26–27) to just check if the purported hint \( rseed[j] \) matches the computed part \( \rho_j \) (28–29). This change is only detectable to the adversary if it can find a joint randomness seed and hints such that the check succeeds, but the aggregators compute distinct \( jseed_1 \neq jseed_2 \). This is impossible by construction (transition from \( G_1 \) to \( G_2 \), so
Figure 24: Games $G_0$, $G_1$, and $G_2$ for the proof of Theorem 3. Let $\mathcal{Y}$ denote the co-domain of the random oracle used by VIDPF.
Next, in game $G_3$ (Figure 25), we make the following changes. First, we modify oracle $RO_4$ so that, for any query that coincides with the secret verification key $sk$ sampled at the beginning of the game, the oracle immediately returns $\perp$ without programming the RO table. Second, we modify $Prep$ by replacing the call to $qr \leftarrow RO_4(sk, n \parallel \ell)$ with

$$qr \leftarrow \text{Rand}[4, sk, n \parallel \ell] \leftarrow \mathbb{F}^{q_4}.$$

That way each call to $Prep$ samples fresh query randomness. The second change does not overwrite any value in Rand due to the first change. Thus the new game is identical to $G_3$ until the adversary makes a query to $RO_4$ with the seed equal to $sk$. Taking a union bound over all of $A$'s queries, we have that

$$\Pr[\mathcal{G}_2(A)] \leq \Pr[\mathcal{G}_4(A)] + \frac{q_4q_{\text{prep}}}{2^c}.$$  

(28)

In the last game, $G_4$ (right-hand panel of Figure 25), we use the extractability of VIDPF to simplify the winning condition. First, we change how the IDPF output vector $\vec{y}$ is computed by $Prep$: If the one-hot check succeeds, i.e., $v$ is set to 1 on line 24, then we use the extractor $E$ to extract $(\alpha, \vec{y})$ from the transcript of the random oracle (7) and set $\vec{y}$ to $f_{\alpha, \vec{y}}(pfx)$. Second, we revise the winning condition (28) by requiring only that the sum of the elements of $\vec{y}$ is not in the delayed-input set $\mathcal{X} = \{0, 1\}$ for DFLP. In particular, we no longer require $\vec{y}$ to be one-hot for the adversary to win. (Recall that $V_{\text{ddtcur}}$ is the set of one-hot vectors where the non-zero element is in $\mathcal{X}$.) These conditions are equivalent in the revised game, since (1) $A$ cannot set $w$ if $v = 0$, and if $v = 1$, vector $\vec{y}$ is one-hot by definition.

We claim that there exists an $O(t_A + q_{\text{prep}}t_E)$-time adversary $B$ for which

$$\Pr[\mathcal{G}_3(A)] \leq \Pr[\mathcal{G}_4(A)] + q_{\text{prep}} \cdot \text{Adv}_{\text{VIDPF,E}}^{\text{extract}}(B).$$  

(29)

The proof is by a hybrid argument. For each $i \in [q_{\text{prep}}]$ let $G_i'$ be the game $G_3$ except that only the first $i$ queries to $Prep$ are answered in the usual way; the remaining queries are answered as they are in game $G_4$. Adversary $B$ first samples $i \leftarrow [q_{\text{prep}}]$ then runs $G_i'(A)$ as usual, except that it simulates $Prep$ queries for one of the reports using its own game. Specifically, after unpacking IDPF public share $pub$ and key shares $key_1, key_2$ on lines 4–6, it pauses the simulation, outputs $(pub, key_1, key_2)$, and waits to be invoked again. On its second invocation, it resumes the simulation of the $Prep$ query until it reaches the computation of $\vec{y}$ on lines 23–26: At this point it queries its own $\text{Eval}$ oracle on the candidate prefixes $pfx$ and sets $\vec{y}$ to the return value. Thereafter, it simulates the remainder of the game faithfully. If $A$ sets $w \leftarrow \text{true}$ in its game, then $B$ guesses 1; otherwise it guesses 0.

Let $\delta_1^i$ (resp. $\delta_0^i$) denote the probability that $B$ samples $i$ and guesses 1 in the VIDPF extractability experiment, conditioned on the outcome of the coin toss being 1 (resp. 0). Then for all $i$,

$$\text{Adv}_{\text{VIDPF,E}}^{\text{extract}}(A) \geq \frac{1}{q_{\text{prep}}} (\delta_1^i - \delta_0^i).$$  

(30)

Moreover, by construction we have that

$$\delta_1^i - \delta_0^i = \Pr[\mathcal{G}_i'(A)] - \Pr[\mathcal{G}_{i+1}(A)].$$  

(31)

for all $i$. The claim follows from the observation that $\Pr[\mathcal{G}_3(A)] = \Pr[\mathcal{G}_4(A)]$ and $\Pr[\mathcal{G}_4(A)] = \Pr[\mathcal{G}_{q_{\text{prep}}}(A)]$.

Consider what $A$ must do to set $w \leftarrow \text{true}$ in game $G_4$. For some $Prep$ query, the delayed-input proof check must succeed when in fact the sum $\sum_{i \in [n]} y[i]$ is not a valid encoded input. We bound $A$’s advantage in game $G_4$ by a reduction to the soundness of DFLP. Recall from the definition of soundness in Section 5.2 that the malicious prover $P^*$ first commits to an encoded input $(e, \Delta)$, then gets a fresh joint randomness $jr$, then picks a proof forgery $\pi$. It wins if DFLP.Decode$(e) \notin \mathcal{L}$ but the verifier deems the input valid (i.e., DFLP.Decide(DFLP.Query$(e, \Delta, \pi, jr; qr)) = 1$, where $qr$ is a fresh query randomness sampled by the game).
Consider the malicious prover $P^*$ in Figure 26. The basic idea is that $P^*$ simulates $G_4(A)$ and extracts its commitment from queries to the random oracle. Specifically, the prover samples $i^* \leftarrow [q_1 + q_{\text{prep}}]$ at the beginning of the game, and for the $i^*$-th query to $RO_1$, it attempts to compute $(\epsilon, \Delta)$ as follows (see lines 15–19).

The prover maintains a reverse look-up table for random oracle queries for computing the query randomness (i.e., $RO_1$), the joint randomness seed parts ($RO_5$), and the joint randomness seed ($RO_3$). On the $i^*$-th query, it looks for values $n$, $\ell$, $pub$, $key_1$, $key_2$, $seed_1$, and $seed_2$ that would be used by a query to $\text{Prep}$. If successful, it uses these to construct its encoded input $(\text{DFLP. Encode}(\Delta, inp^*), \Delta)$ to output in its game (20). It computes $\Delta$ as the sum of the $\Delta_j$’s corresponding to that query (16–17). So how does it compute $inp^*$?

Well, in $G_4$, the $\text{Prep}$ query corresponding to $i^*$ evaluates IDPF keys shares at a set of candidate prefixes $pfx$ chosen by the adversary. But because $pfx$ is not known at this point, the best it can do is guess. It therefore chooses $inp^*$ by sampling uniform randomly from the set $X = \{0, 1\}$ of delayed-input values.

If extraction of the commitment is successful, then the prover outputs it, awaits the response from its game, and programs the table with the response $jr$ (21). Thereafter, prover $P^*$ runs $G_5(A)$ as usual until a $\text{Prep}$ query is made for the session $(n^*, \ell^*)$ that coincides with the distinguished $RO_1$ query $i^*$. At this point, the prover cannot compute the decision bit $d$ and the verifier shares $\sigma_1, \sigma_2$ consistently, as it does not have access to the query randomness sampled by its game. Instead, it simply halts and outputs $\pi_1 + \pi_2$ as its proof forgery (35–37).

Observe that $P^*$’s simulation of $G_5(A)$ is perfect up until the point it halts and outputs its forgery. This is due to the full linearity of DFLP, which allows us to substitute the computation of the query-generation algorithm secret-shared data in $G_3$ with the computation of the query-generation algorithm on plaintext inputs in the prover’s soundness game. It follows that $P^*$ wins precisely when $A$ sets $w \leftarrow \text{true}$ in the call to $\text{Prep}$ that coincides with the distinguished session. Conditioning on the probability that $P^*$ guesses the correct call to $RO_1$, and that we guessed the value of $inp^*$ correctly, we conclude that

$$\Pr[G_4(A)] \leq 2(q_1 + q_{\text{prep}}) \cdot \epsilon.$$  \hspace{1cm} (32)

The bound follows from gathering up each of the equations in simplifying.

### C.4 Doplar Privacy (Theorem 4)

We begin with a game $G_6$ (Figure 27) in which we instantiate $\text{Exp}^\text{priv}_H(A)$ in the random oracle model, in-line the sub-routines of $H$, and simplify the code. Calls to $\text{RG}$ have been replaced with a random oracle $RO$; as usual, $RO$ is implemented by lazy-evaluating a table Rand. In the remainder, we let $q_1$ denote the number of queries $A$ makes to $RO_i$. Another simplifying change we have made is to hard-code the index of the corrupt aggregator, which we denote by $\hat{j}$. (We denote the honest aggregator by $j$.) Accordingly, we have removed the share index $j$ from the oracle parameters and tables, as there is only one valid choice for these. (This is without loss of generality.) None of these changes impact the outcome of the experiment, so

$$\Pr[\text{Exp}^\text{priv}_H(A)] = \Pr[G_6(A)].$$  \hspace{1cm} (33)

In game $G_7$ (Figure 27) we revise the $\text{Shard}$ oracle by sampling the nonce without replacement (line 5). This ensures each report has a unique nonce, which will be useful in subsequent steps. By a birthday bound, we have that

$$\Pr[G_6(A)] \leq \Pr[G_7(A)] + \frac{q_{\text{Shard}}^2}{|N|}.$$  \hspace{1cm} (34)

In our next step, $G_8$ (Figure 28), we modify the $\text{Shard}$ oracle such that, instead of querying the random oracle $RO$, it programs the random oracle using a new sub-routine, $PO$ (31–34). This ensures that the output of $\text{Shard}$ is not correlated with the game’s current state, allowing us to treat the sampled values as fresh. This has a cost, however, since if any of the values programmed by the oracle overwrite existing values in table Rand, then the adversary will end up with an inconsistent view. We can bound this by considering the probability of any one of the following events occurring:
• Seed seed$_1$ or seed$_2$ sampled on line 4 coincides with a query to RO$_2$ made by A (see lines 6–7). We write this as Rand$_2$ for short in the remainder.
• Seed seed$_1$ or seed$_2$ coincides with an element of Rand$_5$ (11–12).
• Vector rseed sampled on lines 11–12 coincides with an element of Rand$_6$ (13).
• Seed jseed sampled on line 15 coincides with an element of Rand$_1$ (16).
• Seed seed$_2$ coincides with an element of Rand$_3$ (18).

Because the nonces sampled by Shard are unique, and because each of this oracle queries encodes the nonce, we can be certain that points programmed into the table by each Shard query do not collide with one another. Indeed, it is only possible for these values to coincide with random oracle queries made by A. Apply a union bound over all qShard queries, we conclude that

$$\Pr[G_1(A)] \leq \Pr[G_2(A)] + \frac{q_2 q_{\text{Shard}}}{2^{k-1}} + \frac{q_5 q_{\text{Shard}}}{2^{k-1}} + \frac{q_6 q_{\text{Shard}}}{2^{2k}} + \frac{q_1 q_{\text{Shard}}}{2^{2k}}.$$  \hspace{1cm}(35)$$

In the next step, $G_3$ (Figure 29), we substitute calls to VIDPF.Gen and VIDPF.VEval with calls to the simulator $S = (S^1_{\text{VIDPF}}, S^2_{\text{VIDPF}})$. The first part, $S^1_{\text{VIDPF}}$, is used to simulate the public share corrupt aggregator’s key share (10); the second part, $S^2_{\text{VIDPF}}$, is used to simulate the honest aggregators one-hot check, based on the output of the first (41). After this second point, we no longer compute the honest aggregator’s refined share $y'$ consistently. Instead, we compute the corrupt aggregator’s refined share $y'$ and compute the challenge input $\eta$ by subtracting the sum from the true sum for the input $a_b$ (43–44).

There exists an adversary $B$ for which

$$\Pr[G_2(A)] \leq \Pr[G_3(A)] + q_{\text{Shard}} \cdot \text{Adv}_{\text{VIDPF} \cdot S}^b(B).$$  \hspace{1cm}(36)$$

The proof is by a standard argument. In each hybrid game, we answer one more Shard query (and the corresponding Prep query) using $S$. Adversary $B$ simply runs $A$ in one of these hybrid games, chosen at random, and outputs whatever $A$ outputs.

In game $G_4$ (Figure 30), we prepare for the Shard oracle for the reduction to DFLP privacy. The primary change is that we have Shard sample the query randomness $q_r$ that will be used to query the proof at each level (see line 18 in the left panel). This ensures that the query randomness is “committed” even before the query is made. We use the unpredictability of the nonce to bound the probability that this change leads to an inconsistent view of the experiment. In particular,

$$\Pr[G_3(A)] \leq \Pr[G_4(A)] + \frac{\eta q_{\text{Shard}}}{|N|}.$$  \hspace{1cm}(37)$$

In this step, we also make a couple of non-breaking changes. First, we in-line programming of the random oracle with the joint randomness and encoding randomness (16–17, 19). Second, we store each proof and encoding randomness in tables $P$ and $D$ respectively. These changes are made to clarify the next step.

In game $G_5$ (Figure 30) we prepare the Prep oracle by re-arranging the proof query. In particular, we run the query-generation algorithm on the plaintext encoded input and proof, and generate the verifier share that is output by subtracting from the verifier (denoted $V[k, \ell]$; see line 19 of the right panel) the verifier share generated from the corrupt aggregator’s share. The adversary’s view is consistent with the previous game by the full linearity of DFLP.

Lastly, in game $G_6$ (not pictured) we modify the Prep oracle by replacing computation of the verifier from $a_b$ with the DFLP-privacy simulator $T$. There exists an adversary $C$ for which

$$\Pr[G_5(A)] \leq \Pr[G_6(A)] + \eta q_{\text{Shard}} \cdot \text{Adv}_{\text{DFLP} \cdot T}^b(C).$$  \hspace{1cm}(38)$$

The proof is by a hybrid argument, where each hybrid game $G'_{u, v}$ is defined as follows. For the first $u$ reports and for the first $v$ levels of the VIDPF tree, the verifier $V[u, v]$ is generated as specified in game $G_5$ (line 19 in the right panel of Figure 30); all other verifiers are generated by $T$ as specified in game $G_6$. By construction,

$$\Pr[G_5(A)] - \Pr[G_6(A)] = \Pr[G'_{0, 0}(A)] - \Pr[G'_{\text{Shard}, 0}(A)].$$  \hspace{1cm}(39)$$
Define DFLP-privacy attacker $C$ as follows. (Refer to Figure 6.) On its first invocation, it simply outputs $X = \{0, 1\}$ as the input set, as this is what is required by the game. On its next invocation, it is given joint randomness $jr^*$ and query randomness $qr^*$. It proceeds by simulating $A$ in a random hybrid game. It first samples $u^* \leftarrow [q_{\text{Shard}}]$ and $v^* \leftarrow [\eta]$. It then runs $G'_{u^*, v^*}(A)$ except:

- On the $u^*$-th query to Shard, for the $v^*$-th level, it uses $jr^*$ and $qr^*$ to program the random oracles for the joint and query randomness respectively.

- When $A$ makes a Prep query corresponding to report $u^*$ and level $v^*$, it halts and outputs $x_b$ and awaits a response from its game. Upon being invoked once more on input $\sigma$, it sets $V[u^*, v^*] \leftarrow \sigma$ and continues the simulation.

Finally, when $A$ halts, $C$ halts and returns whatever $A$ output. Then $C$ perfectly simulates $G'_{u^*, v^*}(A)$ when the value of its challenge bit is 1, and it perfectly simulates $G'_{u^*, v^*+1}(A)$ when its challenge bit is equal to 0. The claimed bound follows from a standard conditioning argument.

To complete the proof, we note that

$$\Pr[G_6(A)] = \frac{1}{2}.$$  \hspace{1cm} (40)

Gathering up all of the terms and simplifying yields the desired bound.
Figure 25: Games $G_3$ and $G_4$ for the proof of Theorem 3. Let $\mathcal{X} = \{0,1\}$ denote the delayed-input set for DFLP.
Figure 26: Malicious prover $P^*$ against the soundness of DFLP for the proof of Theorem 3.
Game $G_0(A)$ $G_1(A)$
1. $(st_A, \{\cdot\}, (sk,\cdot)) \leftarrow \mathsf{Setup}(\cdot); z \leftarrow 3 \cdot z$
2. $b \leftarrow \{0, 1\}; b' \leftarrow \mathsf{Setup \cdot Setup \cdot Setup}(st_A)$
3. ret $b = b'$

$\mathsf{Share}(k \in N, \alpha_0, \alpha_1 \in I)$:
4. if Used[$k$] $\neq \bot$: ret $ot$
5. $n \leftarrow N \setminus N^*; N^* \leftarrow N^* \cup \{n\}$
6. // Construct the VIDPF key shares.
7. seed$_1, seed_2 \leftarrow \{0, 1\}^n$
8. for $\ell \in [n]$:
9. $D[k, \ell] \leftarrow \mathsf{RO}_1(\text{seed}_1, n \parallel \ell \parallel 1)$
10. $+ \mathsf{RO}_2(\text{seed}_2, n \parallel \ell \parallel 2)$
11. $\tilde{\beta}[\ell] \leftarrow \mathsf{Encode}(D[k, \ell], 1)$
12. $(\text{key}_1, \text{key}_2, pub) \leftarrow \mathsf{VIDPF.Gen}(\alpha_1, \tilde{\beta})$ 
13. // Prepare the joint randomness.
14. $\text{rseed}[1] \leftarrow \mathsf{RO}_3(\text{seed}_1, n \parallel pub \parallel \text{key}_1)$
15. $\text{rseed}[2] \leftarrow \mathsf{RO}_3(\text{seed}_2, n \parallel pub \parallel \text{key}_2)$
16. // Prepare the level proofs.
17. for $\ell \in [n]$:
18. $\text{jseed} \leftarrow \mathsf{RO}_3(0^n, \ell \parallel \text{rseed})$
19. $\tilde{\jmath} \leftarrow \mathsf{RO}_3(\text{jseed}, n \parallel \ell)$
20. $\pi \leftarrow \mathsf{DFLP.Prove}(0, 1, D[k, \ell], \tilde{\jmath})$
21. $\text{pff}[\ell] \leftarrow \pi \mathsf{- RO}_3(\text{seed}_2, n \parallel \ell)$
22. // Prepare the initial message and input shares.
23. $x_0 \leftarrow (\text{key}_1, \text{seed}_1, \text{pff})$
24. $x_2 \leftarrow (\text{key}_2, \text{seed}_2)$
25. $\text{In}[k] \leftarrow x_2$
26. Pub$[k] \leftarrow (\text{pub}, \text{rseed})$
27. Used$[k] \leftarrow (n, \alpha_0, \alpha_1)$
28. ret $(n, \text{Pub}[k], (x_2,))$

$\mathsf{Setup}(i \in N, st_{\text{init}} \in Q_{\text{init}})$:
29. $(\ell, p\text{ff}) \leftarrow st_{\text{init}}$
30. if Status[$i$] $\neq \bot$ or $\ell \in U$ or $p\text{ff}$ not distinct: ret $\bot$
31. $U \leftarrow U \cup \{\ell\}$
32. Setup$[i] \leftarrow st_{\text{init}}$; Status$[i] \leftarrow \text{running}$

Figure 27: Games $G_0$ and $G_1$ for the proof of Theorem 4.
\textbf{Shard}(k \in \mathbb{N}, \alpha_0, \alpha_1 \in I):

1. if Used[k] \neq \perp: ret \perp
2. n \leftarrow N \setminus N^*; N^* \leftarrow N^* \cup \{n\}
3. // Construct the VIDPF key shares.
4. seed_1, seed_2 \leftarrow \{0, 1\}^\kappa
5. for \ell \in [\eta]:
6. D[k, \ell] \leftarrow \text{RO}_2 \text{PO}_2(\text{seed}_1, n \parallel \ell \parallel 1)
7. + \text{RO}_2 \text{PO}_2(\text{seed}_2, n \parallel \ell \parallel 2)
8. \vec{\beta}[\ell] \leftarrow \text{Encode}(D[k, \ell], 1)
9. (key_1, key_2, pub) \leftarrow \text{VIDPF.Gen}(\alpha_1, \vec{\beta})
10. // Prepare the joint randomness.
11. rseed[1] \leftarrow \text{RO}_5 \text{PO}_5(\text{seed}_1, n \parallel 1 \parallel pub \parallel key_1)
12. rseed[2] \leftarrow \text{RO}_5 \text{PO}_5(\text{seed}_2, n \parallel 2 \parallel pub \parallel key_2)
13. // Generate the level proofs.
14. for \ell \in [\eta]:
15. jseed \leftarrow \text{RO}_6 \text{PO}_6(0^\kappa, \ell \parallel rseed)
16. jr \leftarrow \text{RO}_1 \text{PO}_1(jseed, n \parallel \ell)
17. \pi \leftarrow \text{DFLP.Prove}(\{0, 1\}, D[k, \ell], jr)
18. pf[\ell] \leftarrow \pi - \text{RO}_3 \text{PO}_3(\text{seed}_2, n \parallel \ell)
19. // Prepare the initial message and input shares.
20. x_1 \leftarrow (key_1, seed_1, pf)
21. x_2 \leftarrow (key_2, seed_2)
22. In[k] \leftarrow x_2
23. Pub[k] \leftarrow (pub, rseed)
24. Used[k] \leftarrow (n, \alpha_0, \alpha_1)
25. ret (n, Pub[k], (x_2,))

\textbf{Figure 28: Game G_2 for the proof of Theorem 4.}
Figure 29: Game $\mathcal{G}_3$ for the proof of Theorem 4.
Figure 30: Games $G_4$ and $G_5$ for the proof of Theorem 4.