# POPSTAR: Lightweight Threshold Reporting with Reduced Leakage

Hanjun Li
*University of Washington*

Sela Navot
*University of Washington*

Stefano Tessaro
*University of Washington*

## Abstract

This paper proposes POPSTAR, a new lightweight protocol for the private computation of *heavy hitters*, also known as a private threshold reporting system. In such a protocol, the users provide input *measurements*, and a report server learns which measurements appear more than a pre-specified threshold. POPSTAR follows the same architecture as STAR (Davidson et al., CCS 2022) by relying on a helper randomness server in addition to a main server computing the aggregate heavy hitter statistics. While STAR is extremely lightweight, it leaks a substantial amount of information, consisting of an entire histogram of the provided measurements (but only reveals the actual measurements that appear beyond the threshold). POPSTAR shows that this leakage can be reduced at a modest cost ($\sim 7\times$ longer aggregation time). Our leakage is closer to that of Poplar (Boneh et al., S&P 2021), which relies however on distributed point functions and a different model which requires interactions of two non-colluding servers to compute the heavy hitters.

## 1 Introduction

Telemetry is essential for assessing the proper functioning of applications and operating systems. For example, a vendor would like to record which events lead to a crash to mitigate potential bugs. The desire to minimize the amount of collected information in this process has led to the emergence of *private telemetry* solutions to compute simple statistics $M(s_1, s_2, \ldots)$ of the users *measurements* $\{s_i\}$, such as their *sum* $\sum s_i$ or their *heavy hitters*, i.e., the set of measurements which appear more than a pre-defined threshold $t$ times. In these systems, we expect the provider to only learn $M(s_1, s_2, \ldots)$, whereas the users learn nothing. While this is a special case of multi-party computation (MPC), rather than using generic off-the-shelf solutions, we aim for lightweight solutions that require no interaction among the users, while tolerating some amount of leakage. This work will propose a new approach for the private computation of heavy hitters. Prior to introducing our contribution, however, we start with some background.

**Two-server aggregation.** Single-server solutions have primarily emerged in the context of Federated Machine Learning [9, 12, 26, 27]. They require multiple rounds of interaction, and consequently need to be robust to client dropouts. In contrast, a number of lightweight telemetry systems like Prio [17] (for sums) and Poplar [13] (for heavy hitters) instead rely on *two non-colluding* servers. Clients only send a single message to each server. Such systems are the subject of an IETF standardization [21], have been generalized [19], and have seen real-world deployment. Crucially, however, the recipient of the aggregation results (e.g., a browser vendor) needs to enlist an external entity trusted not to be colluding, willing to tolerate the same storage cost, and interact during the aggregation protocol. Needless to say, this can be challenging and expensive, as confirmed by the recent deployment [4] of Prio as part of Google/Apple's exposure notification platform (GAEN). Even if a third party specializes in acting as the second server for a number of services, its workload would scale with the number of services supported, and would need to *interact* every time a service recovers the output statistics.

An alternative model is offered by STAR [18], a system for the private computation of heavy hitters which in turn extends earlier telemetry systems based on anonymous tokens [22]. Here, servers operate *independently*. The first server, which we refer to as the *randomness server*, merely implements an oblivious pseudorandom function (OPRF). In contrast, the provider runs a *report server* which obtains reports (computed by the clients with help of the randomness server), and recovers the heavy hitters without interacting with the randomness server. Due to the lack of coordination, the randomness server is now more likely to be implemented by a third-party service. The problem with STAR, however, is that its leakage is substantial–in fact, an anonymized version of the entire frequency histogram for the inputs $\{s_i\}$ is revealed, but only the actual measurements appearing beyond a certain threshold are revealed.

In this paper, we ask the following question: *Can we reduce the leakage of STAR while preserving its architecture and without significantly impacting its efficiency?*

**Our contributions.** We present a new threshold reporting system, POPSTAR, to compute heavy hitters which increases privacy in the STAR system at a moderate cost. Our system, not unlike STAR and Poplar, will still leak information. For a truly passive corrupted report server, our leakage is similar to that of Poplar when reporting totally *random* measurements, without the need of the expensive interactions between two non-colluding servers. We also show the effect of active attacks to be limited.

Our randomness server remains relatively simple (although not as simple as that of STAR), and our report server is roughly seven times as slow as that of STAR under suitable parameter choices. We note that our randomness server is computationally ($\sim 70\times$) heavier than our report server, however the main benefit is its simplicity and that of being independent. We also discuss in Section 8.3 potential optimizations to speed up the randomness server.

We give a full security analysis of our system: we provide a functionality that captures its leakage precisely, and prove our protocol to implement it. We also give an empirical analysis of the leakage and propose a heuristic mechanism to provide differential privacy. Finally, we benchmark an implementation of the components of our system.

## 2 Overview of POPSTAR

**Overview of STAR.** We start with an overview of STAR [18] before introducing the key ideas behind POPSTAR. For starters, STAR's randomness server implements an *oblivious PRF* (OPRF), used to associate with every potential *measurement* $s \in \{0,1\}^*$ a randomly chosen degree $t$ *polynomial* $p_s(X) \in \mathbb{F}[X]$ over a finite field $\mathbb{F}$. In particular, each client querying $s$ to the randomness server will learn the *same* polynomial $p_s(X)$, whereas the randomness server learns nothing about either of $s$ or the polynomial obtained by the client within this interaction.

A client's report for a measurement $s$ has the form

$$\mathsf{rep} = [r, p_s(r), \mathsf{Enc}(p_s(0), s)] \ ,$$

where Enc here denotes the encryption procedure of a symmetric encryption scheme, $r \in \mathbb{F}$ is uniformly chosen, and $p_s(X)$ is the polynomial associated with $s$ previously obtained from the randomness server. If the report server then obtains $t+1$ reports for the same measurement $s$, associated with distinct $r$ values (which is the case with overwhelming probability), the value $p_s(0)$ can be reconstructed via simple interpolation, and the value $s$ can be recovered via decryption from any of the reports.[1] A client, when ready, sends the report directly to the report server (or, as we explain below, to an intermediate mixing server first to eliminate the origin and timing information of the report).

[1] It is often useful to additionally encrypt application-dependent metadata, along with $s$, but we omit this for the sake of brevity.

The problem is that the report server now accumulates reports for potentially different measurements, but cannot recognize which reports are associated with the same measurement. For this reason, STAR includes a *tag* $\mathsf{tag}(s)$ in the report as well—such a tag is also computed from an independent OPRF evaluation with the randomness server, and crucially, tags are deterministic functions of $s$ and unlikely to collide. With overwhelming probability, any $t+1$ reports with the same tag can be used to reconstruct $p_s(0)$ efficiently via interpolation, and then decrypt the associated ciphertexts.

However, tags add unwanted leakage, as the server can now build histograms *for the tags*, and while the associated measurement $s$ is only revealed for tags appearing more than $t$ times, the histogram information associated with unrevealed inputs is important information we ideally want to hide.

**POPSTAR to the rescue: Reducing leakage.** We are now ready to explain our approach to reducing leakage in POPSTAR. The main idea is that the randomness server will associate with every possible string $y \in \{0,1\}^{\leq \ell}$ of length at most $\ell$ an independent (pseudo)random polynomial $p_y(X)$ of degree $t$. Here, $\ell$ is a parameter which we will (empirically) show to be related to the privacy offered by the system–we hence often refer to it as the *privacy parameter*.

By interacting with the randomness server, the client *obliviously* obtains $\ell$ polynomials

$$p_{y_1}(X), p_{y_1 y_2}(X), \ldots, p_{y_1 y_2 \ldots y_\ell}(X)$$

associated with the $\ell$ prefixes of a (pseudo)random string $y(s) = y_1 y_2 \ldots y_\ell$ which is, in turn, derived from $s$. The client also obliviously obtains tags $\mathsf{tag}(y_1), \mathsf{tag}(y_1 y_2), \ldots, \mathsf{tag}(y_1 y_2 \ldots y_\ell)$. Crucially, querying the same $s$ multiple times (by multiple clients) will yield the same polynomials (and tags), whereas querying distinct measurements $s \neq s'$ will very likely lead to different sequences of polynomials/tags that partially overlap up to the length of the longest common substring of $y(s)$ and $y(s')$. For example, if $\ell = 4$, $y(s) = 0000$ and $y(s') = 0010$, interacting with the randomness server on input $s$ will reveal the polynomials

$$p_0(X), p_{00}(X), p_{000}(X), p_{0000}(X)$$

whereas on input $s'$ the revealed polynomials are

$$p_0(X), p_{00}(X), p_{001}(X), p_{0010}(X) \ .$$

POPSTAR's report for a measurement $s \in \{0,1\}^*$ has form

$$\mathsf{rep} = \left[ r, p_{y_1}(r), \mathsf{tag}(y_1), \mathsf{ct}^{(1)}, \ldots, \mathsf{ct}^{(\ell+1)} \right] \ ,$$

where $r \in \mathbb{F}$ is randomly chosen, and

$$\mathsf{ct}^{(i)} = \mathsf{Enc}(p_{y_1 \ldots y_i}(0), p_{y_1 \ldots y_i y_{i+1}}(r) \parallel \mathsf{tag}(y_1 \ldots y_i y_{i+1}))$$

for $i = 1, \ldots, \ell - 1$. The $\ell$-th ciphertexts encrypts $p_s(r) \parallel \mathsf{tag}(s)$, where $p_s, \mathsf{tag}(s)$ are computed from an

independent OPRF evaluation with the randomness server. The final ciphertext encrypts $s$.

$$\mathsf{ct}^{(\ell+1)} = \mathsf{Enc}(p_s(0), s) \ .$$

Now, the report server is always able to "peel off" an additional encryption layer from a report for $s$ with $y(s) = y_1 y_2 \ldots y_\ell$ whenever more than $t$ reports for a prefix $y_1, \ldots, y_i$ have been received.

While the costs of managing reports are higher in POPSTAR than in STAR, our implementation shows that they remain within feasible range. For example, aggregating 1 million reports takes 136.1 seconds, which is roughly 7× slower than STAR. The randomness server is however more complex and less efficient than in STAR–in fact, after extensive benchmarks, the best performing solution we provide is still based on garbled circuits. Still, even here the end-to-end running time of one client interaction with the randomness server is dominated by network latency ($\sim 50ms$), rather than local computation times. In POPSTAR, such an interaction takes 2 round trips whereas in STAR, 1 round trip. Hence, we estimate it to be 2× to 3× slower than STAR.

**Security Analysis.** Our approach substantially reduces leakage compared to STAR. A passive server in particular learns:

1. All strings $y' = y_1 \ldots y_i$ such that $> t$ reports are for a measurement $s$ such that $y_1 \ldots y_{i-1}$ is a prefix of $y(s)$.
2. For each such string $y'$, which reports are for a measurement $s$ such that $y'$ is a prefix of $y(s)$.

This leakage profile resembles that of Poplar, which however uses *either* the measurement itself in lieu of $y(s)$, or a deterministic hash of $s$, which makes our system stronger in this one dimension. However, in contrast to (2) above, each of the two Poplar servers cannot link a particular report to its contribution, and hence only learns *how many* reports are for a measurement $s$ such that $y'$ is a prefix of $y(s)$, but not *which reports*. As in STAR, we mitigate this by introducing an abstract *mixing server* which is used by the clients when submitting their reports. This could be an actual third-party service, or could be implemented heuristically by having all users coordinate sending their reports at pre-specified times using anonymous communication tools such as ToR.

Another attack by a malicious report server is to maliciously spawn clients and interact with the randomness server. However, this attack is rather ineffective due to the randomness of the mapping between $s$ and $y(s)$, and can intuitively only help uncover extra information about random processes up to a small depth. Also, the effect of this attack can be mitigated via rate limiting measures on the randomness server.

We give a detailed functionality capturing the security of POPSTAR in Section 7.1 (and which we use then to prove security in the full version), [2] and then interpret it empirically in Section 7.2. We also propose a heuristic mechanism to provide differential privacy in Section 7.3.

[2] https://eprint.iacr.org/2024/320

**Robustness against malicious client input.** POPSTAR as described above is not very robust. For example, a malicious client may include in its report wrong evaluations of the polynomials, causing interpolations by the report server to produce wrong decryption keys. Any honest report containing a ciphertext that is supposed to be decrypted by those keys will be affected. We note that STAR is also not robust against such malicious reports–while clients can verify that the polynomial is correct (using e.g., a *verifiable* OPRF), the report server cannot generally check that the reports contain legitimate evaluations of the polynomial.

We describe a robust variant of POPSTAR in Section 6 that prevents malicious clients' reports from affecting honest reports, assuming the randomness server behaves honestly.

## 3 Preliminaries

**General notations.** For a natural number $n \in \mathbb{N}$, we write $[n]$ to represent the set $\{1, \ldots, n\}$. We write $x \| y$ to denote the concatenation of two strings $x, y$. For a tuple $S$ and an index set $I$, we write $S_I$ to mean the subset indexed by $I$.

**Shamir's secret sharing.** Although not explicitly using Shamir's secret sharing scheme, POPSTAR relies on the same underlying idea of sharing a secret via polynomial evaluation, and reconstructing the secret via interpolation.

We briefly describe Shamir's scheme over a finite field $\mathbb{F}$, and its correctness and privacy guarantees. They directly translate to properties of polynomial interpolation.

In the following, $M \in \mathbb{N}$ is the number of share holders, $t \in [M]$ is a threshold, and $E = (\mathsf{pt}_1, \ldots, \mathsf{pt}_M)$ is a tuple of distinct points in $\mathbb{F}$.

- $Y \leftarrow \mathsf{Share}^{t,E}(k)$ outputs shares $Y = y_1, \ldots, y_M$ of the secret $k \in \mathbb{F}$, computed as $y_i = f_k(\mathsf{pt}_i)$, where $f_k(x) = k + c_1 \cdot x + \ldots + c_t \cdot x^t$, and $c_1, \ldots, c_t \leftarrow \mathbb{F}$.
- $k \leftarrow \mathsf{Recon}^{t,E}(I, Y_I)$ outputs the secret $k$, reconstructed from a subset of $> t$ shares $Y_I$ as $k = f_k(0)$ where $f_k = \mathsf{Interpolate}(E_I, Y_I, t)$.

**Lemma 1.** *Fix any number $M \in \mathbb{N}$, threshold $t \in [M]$, and distinct points $E = (\mathsf{pt}_1, \ldots, \mathsf{pt}_M) \subseteq \mathbb{F}$. Let $Y = (y_1, \ldots, y_M) \leftarrow \mathsf{Share}^{t,E}(k)$ be shares of a randomly sampled secret $k \leftarrow \mathbb{F}$.*

1. *Any subset of $> t$ shares $Y_I$ indexed by $I \subseteq [M]$ can recover the correct secret: $k = \mathsf{Recon}^{t,E}(I, Y_I)$.*
2. *Any subset of $\leq t$ shares $Y_{I'}$ indexed by $I' \subseteq [M]$ leaks no information about the secret: $(E, I', Y_{I'}) \equiv (E, I', U)$, where $U$ denotes random values over $\mathbb{F}$.*

**Symmetric key encryption with key commitment.** An encryption scheme with key commitment guarantees that a ciphertext may be only decrypted with the same key used to produce it. In particular, the decryptor either learns the correct plaintext or recognizes a decryption failure.

We describe a simple scheme for encrypting $l$-bit messages in the random oracle model, based on the simple padding idea in [2]. Let $H_E : \{0,1\}^* \to \{0,1\}^{l+\lambda}$ be a hash function modeled as a random oracle.

- $\mathsf{Enc}(k, \mathsf{msg})$ samples a random string $r \leftarrow \{0,1\}^\lambda$, and outputs $\mathsf{ct} = (r, c)$ where $c = H_E(k \| r) \oplus (0^\lambda \| m)$.
- $\mathsf{Dec}(k, \mathsf{ct})$ parses $\mathsf{ct} = (r, c)$, and computes $(v^* \| m^*) = c \oplus H_E(k \| r)$. It outputs $m^*$ if $v^* = 0^\lambda$, and $\perp$ otherwise.

In addition to key commitment, the usual correctness and IND-CPA security holds for the above scheme.

Concretely in our evaluations, we use AES-GCM with the padding fix described in [2].

**Garbled circuit [10] (GC).** We use a simplified syntax.

- $(\widehat{C}, K = \{k_0^{(i)}, k_1^{(i)}\}_{[m]}) \leftarrow \mathsf{Garb}(1^\lambda, C)$: given a Boolean circuit $C : \{0,1\}^m \to \{0,1\}^n$, outputs a garbled circuit $\widehat{C}$ and $m$ pairs of keys $K$ corresponding to the inputs to $C$.
- $C(x) = \mathsf{Eval}(\widehat{C}, K_x)$: evaluates the garbled circuit $\widehat{C}$ using $m$ keys $K_x$, which are selected from $K$ according to an input $x \in \{0,1\}^m$.

Correctness and privacy guarantees the evaluator learns $C(x)$ and nothing else. [3] POPSTAR uses GC to implement an oblivious double PRF protocol (Figure 4), between a client and the randomness server.

**Oblivious transfer (OT).** An OT protocol runs between a sender and a receiver with the following interface.

- $\mathsf{OT}^l.\mathsf{send}(\{\mathsf{msg}_0^{(i)}, \mathsf{msg}_1^{(i)}\}_{i\in[l]})$. The sender inputs $l$ pairs of messages $\mathsf{msg}_0^{(i)}, \mathsf{msg}_1^{(i)}$ for $i = 1, \ldots l$.
- $\{\mathsf{msg}_{x_i}^{(i)}\} \leftarrow \mathsf{OT}^l.\mathsf{receive}(x)$. The receiver inputs a choice vector $x \in \{0,1\}^l$, and receives one message from each pair chosen by the corresponding bit of $x$.

Security guarantees that the receiver learns only the messages chosen by $x$, while the sender learns nothing about $x$.

POPSTAR uses OT together with a GC scheme introduced above to implement the oblivious double PRF protocol (Figure 4). Concretely, we use the OT protocol of [15].

[3] More precisely, the evaluator also learns the topology of $C$.

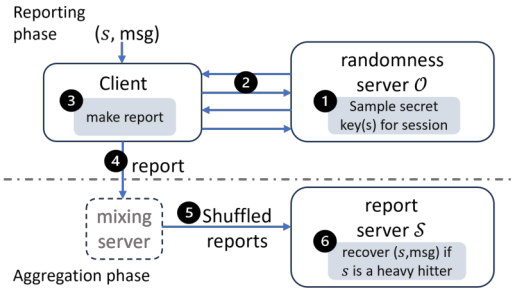



Figure 1: POPSTAR architecture. The server $\mathcal{O}$ samples fresh secret key(s) for each session. In the reporting phase, each client computes a report by interacting with the server $\mathcal{O}$, and sends it to the mixing server. In the aggregation phase, the server $\mathcal{S}$ obtains shuffled reports, and locally recovers heavy hitters and their associated messages.

# 4 System Overview and Threat Model

## 4.1 System Overview

Figure 1 illustrates the system model of POPSTAR. We explain it in more detail below.

**The basic threshold reporting system.** The basic system consists of a set of clients $P_1, P_2, \ldots$, a randomness server $\mathcal{O}$, and a report server $\mathcal{S}$. We envision the system run in recurring sessions, during which each client computes a report of a measurement $s$ and a message msg with the help of the server $\mathcal{O}$, and sends it to the server $\mathcal{S}$. The server $\mathcal{O}$ should learn nothing, and the server $\mathcal{S}$ should only learn the measurements reported more than $t$ times.

The threshold $t$ is a system parameter set appropriately depending on the application scenario and the duration of each session. We emphasize that the server $\mathcal{S}$ should not be able to aggregate reports from different sessions.

Clients do not communicate among themselves and the two servers $\mathcal{S}$ and $\mathcal{O}$ do not communicate with each other. Clients communicate with both servers through private and authenticated asynchronous channels (e.g., both servers deploy TLS for this purpose, and have each a certificate).

**Hiding client identities through a mixing server.** In many applications, it is desirable to hide client identities associated with each report from the server $\mathcal{S}$, as well as the timing of each report. For this, we will assume the availability of an abstract mixing server that collects reports from the clients during each aggregation session, shuffles them randomly, and delivers them to the server $\mathcal{S}$ in one shot (See Figure 1).

The abstract mixing server could be implemented by an actual third-party service, or heuristically by having the clients to coordinate sending their reports at pre-specified times through anonymous communication tools such as ToR.

An alternative suggested in [18] is to also rely on the randomness server $\mathcal{O}$ for this purpose. In more detail, we let the server $\mathcal{O}$ act as an oblivious HTTP proxy between the clients and the server $\mathcal{S}$, which strips away identifying information from client messages containing their reports, batches them until the end of the aggregation session, and delivers all messages in a shuffled order in one shot.

## 4.2 Threat Model and Security Goals

We consider a static malicious adversary who initially corrupts a subset of the participants, and controls them throughout the session. As in prior works [13, 18], we assume the two servers $\mathcal{O}$ and $\mathcal{S}$ *do not collude*. More specifically, we only consider three scenarios: (1) a corrupted server $\mathcal{S}$ with colluding clients; (2) corrupted clients only; (3) a corrupted server $\mathcal{O}$ with colluding clients. We explain the security goals and guarantees of POPSTAR in each scenario below. Section 7.1 will also describe a functionality $\mathcal{F}_{\mathsf{report}}$ that captures the security of POPSTAR precisely, but here we limit ourselves to an informal overview.

**Corrupted server $\mathcal{S}$ with colluding clients.** In this scenario, the goal is to protect privacy of honest clients' inputs, i.e., their measurements and associated messages.

POPSTAR guarantees that if an honest client's measurement is not a heavy hitter, and not among the ones reported by the corrupted clients, then its input is hidden from the adversary, except for a small leakage. (Each colluding client may choose to make a report of an arbitrary measurement $s^*$.)

Section 7.1 captures the leakage precisely, and Section 7.2 compares with prior works in detail. In short, POPSTAR has a leakage similar to the hashing variant of Poplar, and much smaller than STAR.

We note that a baseline attack by a malicious server $\mathcal{S}$ in POPSTAR (and also in STAR) is to spawn many colluding clients, and use each of them to statically target a different measurement $s^*$. This will cause the malicious server to identify reports by honest clients' that are on $s^*$, which will lose privacy. (Of course, this only happens if the malicious server can guess an $s^*$ for which a report is being made.) This attack is somewhat unavoidable in this model, and also affected STAR. We do not try to address this attack within POPSTAR, but argue it can be mitigated by other means in practice. (See the remark on "client rate limiting".)

**Corrupted clients only.** In this scenario, the goal is to prevent maliciously generated reports from damaging the aggregation results, e.g., causing some measurements to be unrecoverable, even if there are $> t$ honest reports of them.

We first present a very efficient construction of POPSTAR without trying to defend against such malicious reports. We then describe a robust variant (Section 6) that minimizes the effect of malicious reports.

The robust variant of POPSTAR guarantees that malicious reports get discarded from the final aggregation results, while ensuring that honest reports are still counted.

**Corrupted server $\mathcal{O}$ with colluding clients.** In this scenario the goal is to protect the privacy of honest clients' inputs.

POPSTAR completely hides honest clients' inputs from the adversary, irrespective of whether the measurements are heavy hitters or not.

POPSTAR has very limited guarantee against malicious reports from corrupted clients, and faulty reports from honest clients caused by a malicious server $\mathcal{O}$. Essentially, the adversary may cause any subset of the honest reports to be discarded from the final aggregation results.

**Remark on colluding servers $\mathcal{S}$ and $\mathcal{O}$.** POPSTAR is designed with two non-colluding servers in mind. However, we note that even when they collude, POPSTAR still provides limited privacy (see Section 5.1). In contrast, Poplar [13] has no privacy when the servers collude.

**Remark on client rate limiting.** As noted in the case of a corrupted server $\mathcal{S}$, privacy of POPSTAR relies on the assumption that the adversary cannot collude or spawn too many malicious clients, nor sending multiple queries through one colluding client to the server $\mathcal{O}$.

We envision enforcing one query per session from the clients by requiring them to register accounts with the server $\mathcal{O}$, who limits the number of interactions per account.

To prevent the adversary from registering multiple accounts, one could use usual measures making the creation of multiple accounts expensive (e.g., requiring multiple e-mail addresses, distinct authentication factors, etc).

# 5 Protocol Description

## 5.1 Threshold Reporting Protocol

The POPSTAR protocol (Figure 1) consists of a reporting phase, where each client computes a report with the help of the server $\mathcal{O}$ and sends it to the mixing server, and an aggregation phase, where the report server obtains reports from the mixing server and locally recovers the heavy hitters. We focus on the more efficient non-robust variant in this section. We describe the robust variant in Section 6, and propose a heuristic mechanism to provide differential privacy in Section 7.3.

The algorithmic descriptions of each client and the report server are given in Figure 2 and 3. Below we first introduce the cryptographic tools used, and the interfaces implemented by the randomness and the mixing servers. We then describe the two phases in more detail. Finally, we analyze the correctness and privacy of the protocol. (Formal security definitions can be found in Section 7.1 and proofs in the full version.)

**Cryptographic primitives.** The protocol uses the symmetric key encryption scheme $(\mathsf{Enc}, \mathsf{Dec})$ with key commitment described in Section 3. It also uses two hash functions (modeled as random oracles): (1) $H_s : \{0,1\}^* \to \{0,1\}^\lambda$, (2) $H_p : \{0,1\}^* \to \mathbb{F}^{t+1} \times \{0,1\}^\lambda$ where $\mathbb{F}$ is a $\lambda$-bit field. The output of $H_p$ is a degree $t$ polynomial $f$ and a tag.

**The randomness and the mixing server interfaces.** The randomness server $\mathcal{O}$ implements two interfaces $\mathsf{OPRF}, \mathsf{ODPRF}$ short for oblivious (double) PRF. We give details of the implementations in Section 5.2.

- $u \leftarrow \mathsf{OPRF}(x)$. Each client can call $\mathsf{OPRF}(x)$ with an $\lambda$-bit string input, and obtain a single $\lambda$-bit string.
- $v^{(1)}, \ldots, v^{(\ell)} \leftarrow \mathsf{ODPRF}(x)$. Each client can call $\mathsf{ODPRF}(x)$ with an $\lambda$-bit string input, and obtain $\ell$ $\lambda$-bit strings.

The former result $u$ is supposed to be a PRF evaluation: $u = F(\mathsf{sk}, x)$, where $\mathsf{sk}$ is known only to the server $\mathcal{O}$. The latter results $v^{(1)}, \ldots, v^{(\ell)}$ are supposed to be PRF evaluations $v^{(d)} = F'(\mathsf{sk}', \mathsf{prefix}(u', d))$, where $u' = F'(\mathsf{sk}', x)$, and $\mathsf{prefix}(u', d)$ denotes the first $d$ bits of $u'$, padded appropriately. $\mathsf{sk}'$ is known only to the server $\mathcal{O}$.

The abstract mixing server implements two interfaces.

- $\mathsf{Mix.send}(R)$. Each client can call $\mathsf{Mix.send}(R)$ to send its report to the mixing server.
- $\{R_j\} \leftarrow \mathsf{Mix.collect}()$. The report server $\mathcal{S}$ can call $\mathsf{Mix.collect}()$ to collect reports sent by the clients, in a randomly shuffled order.

**The reporting phase.** During the reporting phase, each client who wishes to report a measurement $s$ and an associated message $\mathsf{msg}$ independently and asynchronously executes the following steps (formally described in Figure 2).

First, the client hashes its measurement to a $\lambda$-bit string $x = H_s(s)$, and calls the $\mathsf{ODPRF}$ and $\mathsf{OPRF}$ interfaces of the server $\mathcal{O}$ with the input $x$ to obtain evaluation results $v^{(1)}, \ldots, v^{(\ell)}, u$. The client hashes, using $H_p$, the evaluation results into $\ell+1$ degree $t$ polynomials $f^{(1)}, \ldots, f^{(\ell)}, f^{(\ell+1)}$, each associated with a tag.

Next, the client derives a secret key $k^{(d)} = f^{(d)}(0)$ from each polynomial, and a Shamir's secret share (Section 3) of the key $y^{(d)} = f^{(d)}(\mathsf{pt})$. The evaluation point $\mathsf{pt}$ is chosen at random for each report so that $\mathsf{pt}$ does not leak anything about the client identity, and also does not collide with other client's choices with overwhelming probability.

Finally, the client creates a chain of encryptions. The first key $k^{(1)}$ is used to encrypt the second share, together with the second tag: $\mathsf{ct}^{(1)} \leftarrow \mathsf{Enc}(k^{(1)}, \mathsf{tag}^{(2)} \,\|\, y^{(2)})$, and so on. The final key $k^{(\ell+1)}$ is used to encrypt the measurement and the message: $\mathsf{ct}^{(\ell+1)} \leftarrow \mathsf{Enc}(k^{(\ell+1)}, s \,\|\, \mathsf{msg})$. The report consists of the ciphertexts, the first share and tag, and the evaluation point: $R = (\mathsf{pt}, \mathsf{tag}^{(1)}, y^{(1)}, \mathsf{ct}^{(1)}, \ldots, \mathsf{ct}^{(\ell+1)})$. The client sends it to the mixing server using the interface $\mathsf{Mix.send}(R)$.

$\mathsf{POPSTAR\text{-}Client}^{\ell,t}(s, \mathsf{msg})$

$x = H_s(s)$
$v^{(1)}, \ldots, v^{(\ell)} \leftarrow \mathsf{ODPRF}(x)$
$u \leftarrow \mathsf{OPRF}(x)$
**for** $d = 1, \ldots, \ell$ **do**
  $(f^{(d)}, \mathsf{tag}^{(d)}) = H_p(v^{(d)})$
$f^{(\ell+1)}, \mathsf{tag}^{(\ell+1)} = H_p(u \,\|\, s)$
$\mathsf{pt} \leftarrow \mathbb{F}$
**for** $d = 1, \ldots, \ell+1$ **do**
  $k^{(d)} = f^{(d)}(0), \quad y^{(d)} = f^{(d)}(\mathsf{pt})$
**for** $d = 1, \ldots, \ell$ **do**
  $\mathsf{ct}^{(d)} \leftarrow \mathsf{Enc}(k^{(d)}, \mathsf{tag}^{(d+1)} \,\|\, y^{(d+1)})$
$\mathsf{ct}^{(\ell+1)} \leftarrow \mathsf{Enc}(k^{(\ell+1)}, s \,\|\, \mathsf{msg})$
$\mathsf{Mix.send}(R = (\mathsf{pt}, \mathsf{tag}^{(1)}, y^{(1)}, \mathsf{ct}^{(1)}, \ldots, \mathsf{ct}^{(\ell+1)}))$

Figure 2: The client pseudocode of POPSTAR.

**The aggregation phase.** During the aggregation phase, the server $\mathcal{S}$ collects the reports received by the mixing server using the interface $\{R_j\}_{j\in[m]} \leftarrow \mathsf{Mix.collect}()$, and executes the following steps (formally described in Figure 3).

First, the server $\mathcal{S}$ divides the reports into depth-1 subgroups according to the depth-1 tags included in each report, discarding the ones with size $\le t$.

Next, for each depth-1 subgroup $G^{(1)}$, the server uses the shares $\{y_j^{(1)}\}_{j\in G^{(1)}}$ and evaluation points $\{\mathsf{pt}_j\}_{j\in G^{(1)}}$ included in the reports to derive a key $k^{(1)}$ by polynomial interpolation. The server decrypts the depth-1 ciphertexts $\{\mathsf{ct}^{(1)}\}_{j\in G^{(1)}}$ in the group using $k^{(1)}$, discarding reports $R_j$ for which $\mathsf{ct}_j^{(1)}$ fails to decrypt. Note that by using an encryption scheme with key commitment, the server recognizes decryption failures and avoids proceeding with garbage results. A successful decryption of $\mathsf{ct}_j^{(1)}$ recovers the depth-2 tags and shares $\mathsf{tag}_j^{(2)}, y_j^{(2)}$ for report $R_j$. After recovering a depth-2 tag and share for each report in the group $G^{(1)}$, the server divides it further into depth-2 subgroups, discarding the ones with size $\le t$.

The server proceeds analogously for each depth-2 subgroup $G^{(2)}$, further dividing it into depth-3 subgroups, and so on. In the end, the server obtains a list of depth-$(\ell+1)$ subgroups, each with size $> t$.

Finally, for each depth-$(\ell+1)$ group $G^{(\ell+1)}$, the server decrypts the depth-$(\ell+1)$ ciphertexts in it, discarding the ones that fails. A successful decryption of $\mathsf{ct}_j^{(\ell+1)}$ recovers

a measurement $s$ and a message $\mathsf{msg}_j$ for report $R_j$. If the reports in the group contain the same measurement, then the server adds it and associated messages to the aggregation results. Otherwise, the server discards the group.

POPSTAR-Server-$\mathcal{S}^{\ell,t}$

1 : $\{R_j\}_{j\in[m]} \leftarrow \mathsf{Mix.collect}()$ // $m$ denotes the number of reports.
2 : **parse** $R_j = (\mathsf{pt}_j, \mathsf{tag}_j^{(1)}, y_j^{(1)}, \mathsf{ct}_j^{(1)}, \ldots, \mathsf{ct}_j^{(\ell+1)})$
3 : $d$-groups $\leftarrow \emptyset$, for $d \in [\ell+1]$
4 : find-subgroups$(G^{(0)} = [m], 1)$
5 : **for** $d = 1, \ldots, \ell$ **do**
6 : **for** $G^{(d)} \in d$-groups **do**
7 : $k^{(d)} \leftarrow$ derive-key$(G^{(d)}, d)$
8 : **for** $j \in G^{(d)}$ **do**
9 : **if** $\perp \leftarrow \mathsf{Dec}(k^{(d)}, \mathsf{ct}_j^{(d)})$ **then** $G^{(d)} \leftarrow G^{(d)} \setminus \{j\}$
10 : **else** $(\mathsf{tag}_j^{(d+1)}, y_j^{(d+1)}) \leftarrow \mathsf{Dec}(k^{(d)}, \mathsf{ct}_j^{(d)})$
11 : find-subgroups$(G^{(d)}, d+1)$
12 : res $\leftarrow []$ // Stores measurements and associated messages.
13 : **for** $G^{(\ell+1)} \in (\ell+1)$-groups **do**
14 : $k^{(\ell+1)} \leftarrow$ derive-key$(G^{(\ell+1)}, \ell+1)$
15 : $s \leftarrow$ **null**, msgs $\leftarrow \emptyset$
16 : **for** $j \in G^{(\ell+1)}$ **do**
17 : **if** $\perp \leftarrow \mathsf{Dec}(k^{(\ell+1)}, \mathsf{ct}_j^{(\ell+1)})$ **then** go to line 16
18 : **else** $(s_j, \mathsf{msg}_j) \leftarrow \mathsf{Dec}(k^{(\ell+1)}, \mathsf{ct}_j^{(\ell+1)})$
19 : **if** $s \neq$ **null** $\wedge\, s \neq s_j$ **then** go to line 13
20 : **else** $s \leftarrow s_j$, msgs $\leftarrow$ msgs $\cup \{\mathsf{msg}_j\}$
21 : res$[s]$ = res$[s] \cup$ msgs
22 : **return** res
23 : derive-key$(G, d)$

1 : $f = \mathsf{Interpolate}(\{\mathsf{pt}_j\}_{j\in G}, \{y_j^{(d)}\}_{j\in G}, t)$
2 : **return** $k = f(0)$

24 : find-subgroups$(G, d)$

1 : **for** distinct $\mathsf{tag}^{(d)}$ indexed by $G$ **do**
2 : $G^{(d)} = \{j : R_j$ has $\mathsf{tag}_j^{(d)} = \mathsf{tag}^{(d)}\}$
3 : **if** $|G^{(d)}| \leq t$ **then** go to line 1
4 : **else** $d$-groups = $d$-groups $\cup \{G^{(d)}\}$

Figure 3: The report server $\mathcal{S}$ pseudocode of POPSTAR.

**Correctness.** We note three facts of the aggregation phase. (1) A subgroup with size $> t$ is decrypted successfully. (2) Reports on the same measurement are put into the same depth-$d$ subgroup, for all $d \in [\ell+1]$. (3) Reports on different measurements are put into different depth-$(\ell+1)$ subgroups.

First, a depth-$d$ subgroup contains only reports with the same $\mathsf{tag}^{(d)}$. Hence, the shares $y_j^{(d)}$ in this group are evaluations on the same degree $t$ polynomial $f^{(d)}$ uniquely associated with $\mathsf{tag}^{(d)}$, and the ciphertexts $\mathsf{ct}_j^{(d)}$ are encrypted under the key $k^{(d)} = f^{(d)}(0)$. Interpolation of the $> t$ shares recovers $f^{(0)}$ and $k^{(d)}$. Hence, decryptions for the group are successful.

Second, note that the tags $\mathsf{tag}^{(1)}, \ldots, \mathsf{tag}^{(\ell+1)}$ in a report of $s$ are deterministically derived from $s$. Hence, reports of the same $s$ share the same $\mathsf{tag}^{(d)}$, and are put into the same depth-$d$ subgroup for all $d \in [\ell+1]$.

Third, note that the final tag $\mathsf{tag}^{(\ell+1)}$ in a report of $s$ is derived as $f^{(\ell+1)}, \mathsf{tag}^{(\ell+1)} = H_p(u \,\|\, s)$, where $H_p$ is modelled as a random oracle. With overwhelming probability, different $s$ leads to different $\mathsf{tag}^{(\ell+1)}$.

We can now conclude correctness. For any measurement $s$ with $> t$ reports, the subgroups containing these reports all have size $> t$ by fact (2), hence are successfully decrypted by fact (1). The final depth-$(\ell+1)$ subgroup contains only reports of $s$ by fact (3), hence $s$ and the associated messages are added to the aggregation results.

**Privacy.** We informally argue privacy for reports of non-heavy hitter measurements against the report server $\mathcal{S}$. (See Section 7.1 for formal security definitions and the full version for proofs.)

First consider the server $\mathcal{S}$ without colluding clients. A report of some measurement $s$ can be divided into two parts: (1) the final ciphertext, $\mathsf{ct}^{(\ell+1)}$, encrypting $s$ and a message under the secret key $k^{(\ell+1)}$, and (2) the rest of the report, encrypting (through a chain of ciphertexts) a share of the key $y^{(\ell+1)}$. The final ciphertext $\mathsf{ct}^{(\ell+1)}$ remains secure against the server $\mathcal{S}$ when there are $\leq t$ reports of $s$, because $\leq t$ shares of the key $k^{(\ell+1)}$ leaks no information about it.

Next, when the server $\mathcal{S}$ colludes with some clients, each such client allows it to target a certain measurement $s^*$ and directly learn the secret key $k^{*(\ell+1)}$ used for encrypting $s^*$. In more detail, the colluding client is allowed one call to the interface $u^* \leftarrow \mathsf{OPRF}(x^*)$ with $x^* = H_s(s^*)$. It then derives $k^{*(\ell+1)}$ from $u^*$. Reports of $s^*$ lose privacy, while reports for non-heavy hitters $s \neq s^*$ remain private. We emphasize the server $\mathcal{S}$ only chooses targeted measurements $s^*$ during the reporting phase, before seeing any honest client's reports.

Finally, we note that by observing how reports are grouped together during the aggregation phase, the server $\mathcal{S}$ learns a small amount of leakage of the non-heavy hitter measurements. This is because the tags in a report of some $s$ are deterministically derived from $s$. We capture the leakage precisely in our formal security definition (Figure 6), and give a detailed comparison with the leakages in prior works in Section 7.2. Briefly, our leakage is similar to that of the hashing

variant of [13] (Poplar), and much smaller than that of [18] (STAR).

**Limited privacy against colluding servers $\mathcal{S}$ and $\mathcal{O}$.** Observe that a client's report is derived deterministically from its measurement $s$ and interactions with the corrupted server $\mathcal{O}$. So a baseline attack by the adversary is to guess $s$ and verify against the client's report. We argue this is the best it can do in the random oracle model. If the measurements contain sufficient entropy, the adversary cannot efficiently guess correctly.

First note the client's input $x = H_s(s)$ to the ODPRF and OPRF protocols is a hash of $s$, which looks random to the adversary without guessing $s$. The interactions with the server $\mathcal{O}$ as well as the derived keys $k^{(1)}, \ldots k^{(\ell+1)}$ and ciphertexts $\mathsf{ct}^{(1)}, \ldots, \mathsf{ct}^{(\ell)}$ reveal nothing beyond the hash $x$.

Next note that the encryption key $k^{(\ell+1)}$ is derived as another hash of $s$, which looks random to the adversary without guessing $s$. This ensures the final ciphertext $\mathsf{ct}^{(\ell+1)}$ reveals nothing about $s$, unless the key $k^{(\ell+1)}$ is recovered through guesses of $s$ or more than $t$ reports of $s$.

**Remark on a "Lite" variant.** In [18], the authors also propose a "Lite" variant of STAR, where its randomness server is replaced with a public hash function. The privacy guarantee of this variant is weaker, which relies on the client measurements having high entropy. We note that an analogous variant of POPSTAR can be naturally derived by replacing the server $\mathcal{O}$ with public hash functions.

## 5.2 Implementing the Randomness Server

The randomness server $\mathcal{O}$ implements two interfaces, OPRF and ODPRF (see Section 5.1). The former can be implemented by any oblivious PRF protocol (OPRF), e.g., the one of [24], or by a simpler variant of the ODPRF protocol below.

To implement the latter, the server $\mathcal{O}$ samples a secret key $\mathsf{sk} \leftarrow \{0,1\}^\lambda$ for each aggregation session, and listens for client messages. Upon receiving init from a client, the server $\mathcal{O}$ executes the ODPRF (Figure 4) protocol with the client using sk as its input. At the end of the aggregation session, it deletes sk.

**The ODPRF protocol.** The protocol has three steps.

1. The server $\mathcal{O}$ defines a circuit $C$ such that $C(\mathsf{sk}, x)$ computes $\ell$ $\lambda$-bit strings $v^{(1)}, \ldots, v^{(\ell)}$ exactly as required by the interface. It computes a garbled circuit (GC) $\widehat{C}_{\mathsf{sk}}$ of $C(\mathsf{sk}, \cdot)$ together with $\lambda$ pairs of inputs keys $\{k_0^{(i)}, k_1^{(i)}\}$.
2. The server $\mathcal{O}$ and the client run an oblivious transfer (OT) protocol $\mathsf{OT}^\lambda$. The server calls $\mathsf{OT}^\lambda.\mathsf{send}(\{k_0^{(i)}, k_1^{(i)}\})$ to send the input keys, and the client calls $K_x = \{k_x^{(i)}\} \leftarrow \mathsf{OT}^\lambda(x)$ to receive input keys corresponding to $x$.

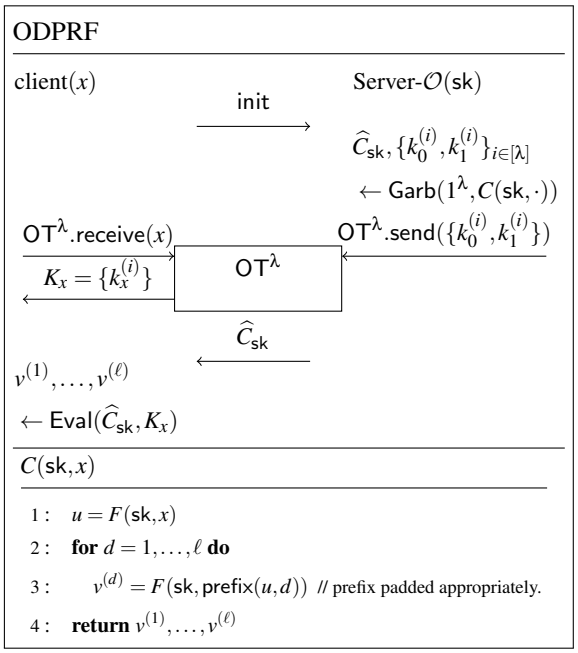



Figure 4: The oblivious double PRF protocol.

3. The server $\mathcal{O}$ sends the garbled circuit $\widehat{C}_{\mathsf{sk}}$ to the client, who locally evaluates it to obtain the results $v^{(1)}, \ldots, v^{(\ell)} = C(\mathsf{sk}, x)$.

Correctness follows directly from that of the GC scheme and the OT protocol. (See Section 3.) Privacy guarantees that the client's input $x$ and the server's secret key sk are hidden from each other. This also follows directly from the security of the GC scheme and the OT protocol.

We note that compared to a generic maliciously secure 2PC protocol computing $C$, our protocol does not enforce the server $\mathcal{O}$ to send the correct garbled circuit corresponding to $C(\mathsf{sk}, \cdot)$. We make this relaxation for better efficiency.

The observation is that the client's outputs from the ODPRF protocol only affect how its report is grouped during the aggregation phase, but not the privacy of the report.

**Concrete Choice of $F$.** We instantiate $F$ with the LowMC [3] block-cipher with 128-bit keys and blocks, and 64-bit data security. LowMC is designed to minimize the number of AND gates in its circuit, which in turn minimizes our garbled circuit (with free-XOR) size and computation.

According to the parameter calculation script, [4] our instantiation of $F$ has 1638 AND gates. Hence, the circuit $C$ computing the double PRF evaluations has $1638 \cdot (\ell+1)$ AND gates.

[4] https://github.com/LowMC/lowmc

## 6 The Robust Variant

We describe a robust variant of our protocol to minimize the effect of malicious reports from corrupted clients, assuming the randomness server $\mathcal{O}$ behaves honestly.

Recall that during the aggregation phase (Figure 3), for $d \in [\ell+1]$, the server $\mathcal{S}$ groups reports according to their revealed depth-$d$ tags. For each group $G^{(d)}$ of size $> t$, it interpolates the revealed depth-$d$ shares to obtain a secret key $k^{(d)}$, and decrypts the depth-$d$ ciphertexts in the group.

A malicious report $R_i^*$ in the group may affect the process in three ways.

1. It may contain a wrong share $y_i^{*(d)} \neq f^{(d)}(\mathsf{pt}_i)$, where $f^{(d)}$ is the polynomial uniquely associated with the $\mathsf{tag}^{(d)}$ for this group. The server $\mathcal{S}$ may derive a wrong key $k^{*(d)}$ as a result, and fail at decrypting all ciphertexts in the group.

2. It may contain a wrong ciphertext $\mathsf{ct}^{*(d)}$ that fails to decrypt under the correct key for this group. The server $\mathcal{S}$ drops the malicious report $R_i^*$ as a result.

3. When $d = \ell+1$, it may contain a $(\ell+1)$-th ciphertext decrypting to a different measurement $s^*$ from the honest reports in the group. The server $\mathcal{S}$ skips the group as a result.

We describe how to prevent (1) and (3) below. We don't prevent (2), as it only results in the malicious report $R_i^*$ being discarded.

**Preventing (3).** At high level, we prevent (3) by allowing the server $\mathcal{S}$ to verify that a decrypted measurement $s$ indeed belongs to its depth-$(\ell+1)$ group. To this end, we augment the OPRF interface implemented by the randomness server $\mathcal{O}$ with *verifiability*. (We show how to implement this interface using a verifiable oblivious PRF protocol in the end.)

- $(u, \pi_x) \leftarrow \mathsf{VOPRF}(x)$. Each client can call $\mathsf{VOPRF}(x)$ to obtain a $\lambda$-bit evaluation result $u$, and a proof $\pi_x$.

The result $u$ is supposed to be a PRF evaluation $u = F(\mathsf{sk}, x)$ as before, and the proof $\pi_x$ is supposed to be verifiable against a public key $\mathsf{pk}$ by an algorithm Verify: $b \leftarrow \mathsf{Verify}(\mathsf{pk}, u, x, \pi_x)$. We assume a PKI setup where every client and the report server $\mathcal{S}$ learns $\mathsf{pk}$.

During the reporting phase (Figure 2), each client calls $(u, \pi_x) \leftarrow \mathsf{VOPRF}$ in place of $u \leftarrow \mathsf{OPRF}(x)$ (line 3), and encrypts $u, \pi_x$ in addition to its measurement and message in the depth-$(\ell+1)$ ciphertext: (line 12)

$$\mathsf{ct}_i^{(\ell+1)} \leftarrow \mathsf{Enc}(k^{(\ell+1)}, (s, \mathsf{msg}_i, \boxed{u \,\|\, \pi_x} \,\|\, s \,\|\, \mathsf{msg})).$$

During the aggregation phase (Figure 3), the server $\mathcal{S}$ decrypts $(u, \pi_x, s, \mathsf{msg}) \leftarrow \mathsf{Dec}(k^{(\ell+1)}, \mathsf{ct}^{(\ell+1)})$ (line 18), and checks $s$ against the attached proof $\pi_x$ as follows.

- Compute $x = H_s(s)$, and run $b = \mathsf{Verify}(\mathsf{pk}, u_s, x, \pi_x)$.
- If $b = 1$, then derive $f^{(\ell+1)}, \mathsf{tag}^{(\ell+1)} = H_p(u \,\|\, s)$.
- If the derived $\mathsf{tag}^{\ell+1}$ equals the tag for the current depth-$(\ell+1)$ group, then $s$ is a correct measurement. Otherwise, discard $s$.

**Preventing (1).** Our goal is to prevent the server $\mathcal{S}$ from deriving a wrong key $k^{*(d)}$ for some depth-$d$ group of $> t$ reports that contains wrong shares.

We first show a lightweight modification that allows the server $\mathcal{S}$ to efficiently recover the correct key, assuming the group contains much more than $t$ honest shares, and much fewer than $t$ wrong shares. When the assumption does not hold, the server $\mathcal{S}$ might time out trying to recover the correct key, and skips the group. We believe this suffices for many real world use cases, where the number of corrupted clients in the system is much smaller than the threshold $t$, and the majority of the heavy hitter measurements are reported much more than $t$ times.

We modify how the server $\mathcal{S}$ derives a key for the group.

1. Interpolate a random subset $R$ of $t+1$ shares to a polynomial $f_R^{(d)}$, and derive a *candidate* key $k_R^{(d)} = f^{(d)}(0)$.

2. Try decrypting the depth-$d$ ciphertexts in $R$. If all decryption succeeds, then use $k_R^{(d)}$ for decrypting the rest of the group. Otherwise, repeat the above with a fresh random subset $R'$.

This procedure succeeds whenever the random subset $R$ contains no malicious reports, which happens with a good probability when the above assumption holds. As an example, consider a threshold $t = 1000 - 1$, a group with $50,000$ reports, where 50 are malicious. The probability of sampling a subset $R$ with no malicious reports is $\binom{49,950}{1000} / \binom{50,000}{1000} > 0.36$. Hence, in expectation it takes less than 3 tries to recover the correct secret key.

It's also possible to let the server $\mathcal{S}$ unconditionally recognize and discard all wrong shares during the aggregation process using a polynomial commitment scheme (PC). This allows us to fully prevent (1). We describe this heavier-weight method very briefly.

At high level, a PC scheme allows each client to compute a commitment $C_f$ to a polynomial $f$, and a proof $\pi_{\mathsf{pt},f,y}$ that an evaluation $y$ is computed correctly from the committed polynomial as $y = f(\mathsf{pt})$. We use a PC scheme, e.g. that of [25], where the commitment $C_f$ is deterministically derived from $f$, so that $C_f$ also functions as a unique tag for $f$.

In the reporting phase, each client replaces $\mathsf{tag}^{(d)}$ with a commitment $C_f^{(d)}$ to the polynomial $f^{(d)}$ and a proof $\pi^{(d)}_{\mathsf{pt}_i,f,y_i}$, for $d \in [\ell+1]$. During the aggregation phase, the server first proceeds optimistically without checking the commitments and proofs. (Our use of key-committing encryption ensures

successful decryption results are correct.) Only in case of a decryption failure, indicating a wrong key $k^{*(d)}$ is derived, does the server verify the shares used for interpolation against their commitments and proofs to discard the wrong ones.

Thanks to the optimistic server strategy, we do not expect significant changes in server loads, even though polynomial commitment checks could be expensive.

**Implementing the interface** VOPRF**.** We can implement the interface using any existing verifiable oblivious PRF (VOPRF) protocol, e.g. the one of [24] with an extra step.

At high level, a VOPRF protocol allows a client and the randomness server $\mathcal{O}$ securely evaluate a PRF based on the client's input $x$ and the server's secret key $\mathsf{sk}$. Additionally, the client can verify the evaluation result against a public key $\mathsf{pk}$, using a proof $\pi$ from the server.

This almost matches the desired interface, except the proof is only intended to be verified by the client. To output a proof that can be verified by anyone holding $\mathsf{pk}$, we simply let the client attach its view during the protocol, including its internal randomness, to the proof.

## 7 Security Analysis

### 7.1 Ideal Functionality and Security Proofs

We formalize the security properties of POPSTAR into an ideal functionality $\mathcal{F}_{\mathsf{report}}$ (Figure 5, 6, 7) in the universal composability (UC) framework. (See the full version for an overview of UC.) The functionality has the following parameters.

- $M \in \mathbb{N}$ is an upper bound on the number of clients, and $t \in [M]$ is the threshold for heavy hitters.
- $\ell \in \mathbb{N}$ is the depth of the prefix tree $T$ internally maintained by $\mathcal{F}_{\mathsf{report}}$. As we will explain, the leakage to a corrupted server $\mathcal{S}$ is captured by a set of leaked nodes on $T$. Asymptotically, setting $\ell = O(\lambda)$, where $\lambda$ is the security parameter, bounds the number of leaked nodes to be $O(\lambda \cdot M/t)$ with overwhelming probability.

**The honest interface.** Figure 5 describes the interface with honest clients and the two servers $\mathcal{O}$ and $\mathcal{S}$, which captures the following correctness guarantee. *If all parties behave honestly, then* $\mathcal{F}_{\mathsf{report}}$ *reveals exactly the measurements (with associated messages) reported by* $> t$ *clients to the server* $\mathcal{S}$. We can verify every distinct measurement $s$ is mapped to a distinct leaf node on the prefix tree $T$, and that in the end only the leaf nodes with count $> t$ are revealed to the server $\mathcal{S}$.

**Functionality** $\mathcal{F}_{\mathsf{report}}^{M,t,\ell}$

The functionality runs with up to $M$ clients $P_1, P_2, \ldots$, a randomness server $\mathcal{O}$ and an report server $\mathcal{S}$.

**Honest Interface:**

Ignore all messages associated with $\mathsf{sid}$ before receiving $(\mathsf{init}, \mathsf{sid})$ from the server $\mathcal{O}$ and after receiving $(\mathsf{collect}, \mathsf{sid})$ from the server $\mathcal{S}$.

1. Upon receiving $(\mathsf{init}, \mathsf{sid})$ from the server $\mathcal{O}$, initialize two data structures (associated with $\mathsf{sid}$):
   - $\mathsf{tb}$: a random table mapping received measurements to $\ell$-bit strings.
   - $T$: an $\ell$-level binary tree where each node stores a count (initially 0) and a state (initially $\mathsf{hidden}$).

   We call the initial nodes of $T$ *inner nodes*. The functionality will add a *leaf node* for each distinct received measurement.
2. Upon receiving $(i, s, \mathsf{msg}, \mathsf{sid})$ from a client $P_i$, interpret $\mathsf{path}_s = \mathsf{tb}[s] \in \{0,1\}^\ell$ as a path on the tree $T$.
   - Add a leaf node labelled by $s$ at the end of $\mathsf{path}_s$ if there is not one already, and store $\mathsf{msg}$ on it. (Let $\overline{\mathsf{path}_s}$ denote the path including the leaf node.)
   - Increase the count of each node on $\overline{\mathsf{path}_s}$.
   - If a node has count $> t$, set its state to $\mathsf{revealed}$.
3. Upon receiving $(\mathsf{collect}, \mathsf{sid})$ from the server $\mathcal{S}$, collect the measurements and messages stored on each revealed *leaf* node, and send them to the server $\mathcal{S}$.

Figure 5: The interface with honest parties. It captures the correctness of POPSTAR.

**The leakage to a corrupted server** $\mathcal{S}$**.** Figure 6 describes the interface with an adversary who statically corrupts the server $\mathcal{S}$ and a subset of the clients. It captures two ways the adversary learns additional information (i.e., the leakage) about the honest reports besides the legitimate aggregation results.

First, without any colluding clients, the server $\mathcal{S}$ learns the count of every $\mathsf{revealed}$ node and its children on the tree $T$. (See Figure 8 for an illustration.) This leakage does not reveal the clients' reports directly, but only how they are partially grouped together on the tree $T$.

To understand this leakage, we focus on the deepest leaked nodes, which we call *end* nodes. (The counts on the remaining leaked nodes can be inferred from those on the end nodes.) In the extreme case where all nodes on $T$ are leaked, the end nodes are all the leaves. The counts on them essentially leaks an anonymized histogram of all received measurements.

**Functionality $\mathcal{F}_{\mathsf{report}}^{M,t,\ell}$ Continued**

**Corrupted report server $\mathcal{S}$ and colluding clients:**
For each corrupted client $P_i$, the adversary may send $(i,s,\mathsf{inner},\mathsf{sid})$ and $(i,s',\mathsf{leaf},\mathsf{sid})$ in any order.

- Upon receiving $(i,s,\mathsf{inner},\mathsf{sid})$, set all nodes on $\mathsf{path}_s$ to revealed, and leak $\mathsf{path}_s$ to the adversary;
- Upon receiving $(i,s',\mathsf{leaf},\mathsf{sid})$, set the leaf for $s'$ to revealed, and leak its location on $T$ to the adversary.

In the end, leak all revealed nodes and their children (i.e., their locations on $T$ and stored counts) to the adversary.

Figure 6: The interface with the adversary corrupting the report server $\mathcal{S}$ and a subset of clients. It captures the potential leakages to a corrupted report server $\mathcal{S}$.

We argue the leakage in POPSTAR is much smaller than the extreme case by showing the number of end nodes on $T$ is much less than the number of all leaves.

- If a revealed path has length $l < \ell$, then it creates $\leq l+1$ end nodes: two children by the last node on the path, and 1 child by each of the rest.
- The case of $l = \ell$ is similar, except the last node on the path causes all its leaves to be end nodes. When $\ell = O(\lambda)$, there are $O(1)$ leaves with overwhelming probability. Hence, $\ell + O(1)$ end nodes are created.

There are at most $M/(t+1)$ revealed paths, which creates at most $O(\ell \cdot M/(t+1))$ end nodes. If the threshold $t$ is a constant fraction of $M$, there are $O(\ell) = O(\lambda)$ end nodes.

Second, with each corrupted client $P_i$, the adversary may cause $\mathcal{F}_{\mathsf{report}}$ to reveal a length-$\ell$ path and a leaf node, through the messages $(i,s,\mathsf{inner},\mathsf{sid})$ and $(i,s',\mathsf{leaf},\mathsf{sid})$. A revealed path adds at most $\ell + O(1)$ leaked *end* nodes in the leakage above. A revealed leaf (for $s'$) lets the adversary learn the count of reports for $s'$ and their associated messages.

**The effect of malicious reports.** Figure 7 describes the interfaces with an adversary who statically corrupts a subset of clients, and with one who additionally corrupts the server $\mathcal{O}$. It captures the effects of malicious reports.

When only clients are corrupted, for each corrupted client $P_i$, the adversary may first send $(i,s,\mathsf{sid})$ to submit a measurement, but then instruct $\mathcal{F}_{\mathsf{report}}$ to update the prefix tree $T$ based on an arbitrary $\mathsf{path}^*$ of length $\ell^* \leq \ell+1$.

In more detail, we assume every node in $T$ is labelled with some tag $\tau$, and $\mathsf{path}^*$ contains $\ell^*$ tags $\tau_1,\ldots,\tau_{\ell^*}$. The first tag $\tau_1$ indicates the child of the root labelled with $\tau_1$. Add such a child if it does not exist in $T$. Inductively, the $d$-th tag $\tau_d$ indicates the child of $\tau_{d-1}$ labelled with $\tau_d$.

**Functionality $\mathcal{F}_{\mathsf{report}}^{M,t,\ell}$ Continued**

**Corrupted clients only:**
For each corrupted client $P_i$, the adversary sends $(i,s,\mathsf{inner},\mathsf{sid})$ to $\mathcal{F}_{\mathsf{report}}$, who replies with $\mathsf{path}_s$. The adversary then sends one of the following, where $\mathsf{path}^*$ is an arbitrary path of length $\ell^* \leq \ell+1$.

- $(\mathsf{path}^*,\mathsf{msg},\mathsf{sid})$: update the count and state for each node on $\mathsf{path}^*$ as in the honest interface. If $\mathsf{path}^*$ includes a leaf node, then store msg on it.
- $(\mathsf{path}^*,\mathsf{damage},\mathsf{sid})$: after updating the counts and states for nodes on $\mathsf{path}^*$, set the *sub-tree* rooted at its last node to damaged, which can no longer be revealed.

**Corrupted randomness server $\mathcal{O}$ and colluding clients:**
Denote the set of corrupted clients $C$, and the rest $H$. Notify the adversary upon receiving an honest input message. Upon receiving $(\mathsf{collect},\mathsf{sid})$ from the server $\mathcal{S}$, notify the adversary, who replies $(\{i,s_i^*,\mathsf{msg}_i^*\}_C,\mathsf{Discard}^*,\mathsf{sid})$, where $\mathsf{Discard}^*$ is a circuit that takes $\{s_i\}_{i\in H}$ as inputs, and outputs a subset $D \subseteq [H]$.
Run $D = \mathsf{Discard}^*(\{s_i\}_H)$, and discard the inputs indicated by $D$. Output the aggregation results over the remaining inputs to the server $\mathcal{S}$.

Figure 7: The interface with the adversary corrupting a subset of clients and possibly also the randomness server $\mathcal{O}$. It captures the potential damages caused by malicious reports.

In addition to updating the tree $T$ according to $\mathsf{path}^*$, the adversary specifies one of the two further actions for $\mathcal{F}_{\mathsf{report}}$. (1) $(\mathsf{path}^*,\mathsf{msg},\mathsf{sid})$ instructs $\mathcal{F}_{\mathsf{report}}$ to store msg on the leaf node, if any, specified by $\mathsf{path}^*$. (2) $(\mathsf{path}^*,\mathsf{damage},\mathsf{sid})$ instructs $\mathcal{F}_{\mathsf{report}}$ to mark the sub-tree rooted at the last node on $\mathsf{path}^*$ as damaged and never revealed.

When the server $\mathcal{O}$ plus a subset of clients are corrupted, the adversary is allowed to specify an arbitrary function $\mathsf{Discard}^*$ that decides a subset $D \subseteq [H]$ of client inputs to discard. More specifically, the adversary first commits a measurement and message $(s_i^*,\mathsf{msg}_i^*)$ for every corrupted client, and specifies the function $\mathsf{Discard}^*$. Then $\mathcal{F}_{\mathsf{report}}$ runs it over received honest measurements $D = \mathsf{Discard}^*(\{s_i\}_{i\in H})$, and computes the aggregation results over the remaining inputs.

**The robust variant.** A weakness of $\mathcal{F}_{\mathsf{report}}$, is that even when only one client is corrupted, its malicious report could cause many valid measurements to become un-recoverable. This is modeled as the $(\mathsf{path}^*,\mathsf{damage},\mathsf{sid})$ command in Figure 7, which damages the entire sub-tree rooted at the last node of $\mathsf{path}^*$. We describe a robust variant of POPSTAR in Section 6 that prevents this attack. Formally, the robust

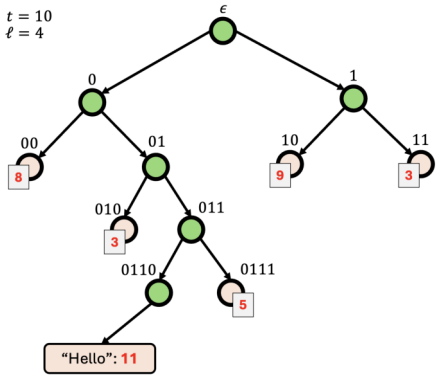



Figure 8: Example of a prefix-tree visible to the report server for a threshold $t = 4$ and depth $\ell = 4$. The numbers in the boxes correspond to the number of reports associated with the end nodes. Here, only a single measurement "Hello" exceeds the threshold.

functionality is identical to $\mathcal{F}_{\text{report}}$, except it does not allow the $(\mathsf{path}^*, \mathsf{damage}, \mathsf{sid})$ command.

**Theorem statement.** We state the security of POPSTAR below. See the full version for the formally stated theorem, the analogous theorem for the robust variant, and the proofs.

**Theorem 1** (Informal)**.** *The protocol described in Section 5 UC-realizes the functionality $\mathcal{F}_{\text{report}}$ in the random oracle model, against a malicious adversary that statically corrupts at most one of the servers $\mathcal{S}, \mathcal{O}$ and any number of colluding clients.*

## 7.2 Leakage Comparisons with Prior Work

This section compares our leakage profile, formally captured by our ideal functionality, to that of prior works, and offers a heuristic evaluation of the profile. It is helpful to refer to Figure 8, which illustrates an example of the prefix tree visible to the server.

**Leakage comparison with [18] (STAR).** We first consider the case where only the report server $\mathcal{S}$ is corrupted, without any colluding clients. Every report in STAR contains a tag deterministically computed from the measurement $s$. The server learns an unlabeled histogram of the reported measurements by grouping them according to their tags.

While POPSTAR's leakage profile is complex, and could lead to more refined leakage abuse attacks, here we discuss the simplest type of inference attack which exploits the report counts for the end nodes in the prefix tree, and attempts to reconstruct the frequency histogram leaked by STAR. In general, because measurements are assigned to uniformly random paths, the deeper an end node in the tree, the more likely its count is attributed to reports for a single measurement. However, we argue that in POPSTAR many end nodes represent counts coming from reports for different measurements, and this therefore strictly reduces leakage.

To verify this experimentally, we sample $1,000,000$ reports from a Zipf power-law distribution with a support $N = 10,000$ and parameter $s := 1.03$, matching the evaluation settings in Section 8 and in [18]. In Table 1, we report the number of end nodes in the prefix tree of POPSTAR, (excluding the leaf nodes for actual heavy hitters), compared against the number of non-heavy hitter report groups formed in STAR. We also report the number of end nodes whose count is attributed to a single measurement (denoted "exact counts") in Table 1.

| | $t = 10,000$ | $t = 1000$ | $t = 100$ |
|---|---|---|---|
| Groups (STAR) | 9990 | 9899 | 9059 |
| End nodes (POPSTAR) | 175 | 1071 | 4568 |
| Exact counts (POPSTAR) | 19 | 167 | 2120 |

Table 1: Leakage comparison between STAR and POPSTAR (with $\ell = 16$). A group in STAR leaks the exact count of a non-heavy hitter. An end node in POPSTAR leaks the combined count of $\geq 1$ non-heavy hitters. The end nodes leaking 1 non-heavy hitter are called exact counts.

In the above experiment setting, increasing $\ell$ from 1 to 16 leads to a decrease in the number of end nodes. Increasing $\ell$ beyond 16 does not change the numbers anymore.

From Table 1, we observe that POPSTAR is most effective at leakage reduction at high thresholds. At a 0.1% threshold ($t = 1000$), which is the main setting considered by STAR and Poplar, STAR leaks the exact counts of 9899 non-heavy hitters, while POPSTAR leaks only 1071 combined counts. Among them, only 167 are exact counts.

Finally, we briefly note that in POPSTAR, a corrupted server $\mathcal{S}$, with each colluding client, may actively cause a path on the prefix tree to leak. In the worst case, i.e., when the leaked path corresponds to a non-heavy hitter reported by honest clients, this causes $\leq \ell$ leaked end nodes. Otherwise, the leaked path is likely to only overlap with the prefix tree at top levels, causing few leaked end nodes. In practice, the attack can be further mitigated by other means, such as rate limiting measures in the server $\mathcal{O}$.

**Leakage comparison with [13] (Poplar).** We compare with Poplar in the setting where one of its report servers is corrupted. Similar to POPSTAR, the leakage in Poplar can be captured by the report counts for the end nodes in a prefix tree. The difference, however, lies in how each measurement $s$ maps to a path of the prefix tree.

In the main variant of Poplar, $s$ maps to a path corresponding to itself in the clear. (In contrast, in POPSTAR, $s$ maps to a path corresponding to $\mathsf{ODPRF}(H(s))$.) The leakage to the corrupted server reveals the counts of heavy hitter pre-

fixes of reported measurements, which can sometimes be dangerous. Consider an example borrowed from [18], where the measurements are country names, and a heavy hitter is `'united states'` with count 4. A leaked prefix `'united'` with a count 5 indicates the existence of a non-heavy hitter among only a few possibilities (e.g., `'united kingdom'`).

In the (slower) hashing variant ([13], Appendix B) of Poplar, $s$ maps to a path corresponding to a public hash $H(s)$. The leakage only reveals the counts of heavy hitter prefixes of *hashes* of the measurements, which is much safer. Still, a possible attack from the corrupted server is to locally evaluate $H(\cdot)$ and try matching possible measurements to the leaked prefixes. Such an attack is impossible in POPSTAR because the corrupted server $\mathcal{S}$ cannot evaluate ODPRF locally.

Finally, we briefly note that in Poplar, a corrupted report server may actively, and adaptively, cause nodes on the prefix tree to leak. This is because the prefix tree is interactively reconstructed from the root by the two servers. For each node with count $> t$, the servers continue to reconstruct its two children. A corrupted server may arbitrarily inflate the count of any node, causing its two children to be leaked. The only restriction is that the total (inflated) counts of each level cannot exceed $M$.

### 7.3 Adding Differential Privacy Heuristically

In the setting where each client reports only its measurement without any associated messages, we propose a heuristic mechanism that we conjecture satisfies a meaningful notion of differential privacy [5] for sufficiently large $M$ and when the threshold $t = O(M)$ is a constant fraction of $M$. We leave it as an important open question to analyze this method, and/or to provide attacks.

The idea is to let each client, in addition to the report of its actual measurement, send up to two fake reports:

1. with probability $p = O(1/M)$, send a fake report of its measurement (i.e., reporting the same measurement twice);
2. send a fake report of a random $\lambda$-bit measurement.

For a measurement with an actual count $c$, the fake reports may inflate the count to $c' = c \cdot (1+p) + O(\lambda)$. To counteract the inflation, we increase the original threshold $t$ to $t' = t \cdot (1+p) + O(\lambda)$.

We provide some intuitions for the conjectured privacy. First consider a report of some measurement $s$ with an (inflated) count $c' > t'$. The leakage with respect to this report is exactly the count $c'$, which contains at least a noise of $\mathsf{Bin}(t,p) = \mathsf{Bin}(O(M), O(1/M))$ contributed by the type-(1) fake reports, where $\mathsf{Bin}(t,p)$ denotes the binomial distribution with $t$ trials and probability $p$.

[5] The mechanism may still reduce the leakage to a corrupted server $\mathcal{S}$ even if we can not prove it provides differential privacy.

Next consider a report of some $s$ with a count $c' \le t'$. The leakage with respect to this report is the count on a leaked inner node at some depth $d$. If $c'$ is still relatively large, $c' > t'/2$, then the leakage still contains at least a noise of $\mathsf{Bin}(t/2, p) = \mathsf{Bin}(O(M), O(1/M))$.

If the count $c'$ of $s$ is small, $c' < t'/2$, we will argue that with $1 - O(\lambda/M)$ probability, the leaked inner node is at a low depth $d < \log(4M)$. In this case, each type-(2) fake report of a random measurement has a chance $2^{-d} > 1/(4M)$ of being mapped also to this leaked node. They contribute to at least a noise of $\mathsf{Bin}(M, 1/(4M)) = \mathsf{Bin}(O(M), O(1/M))$ to the leaked count of the inner node.

It remains to analyze the probability that the leaked inner node for $s$ has depth $d > \log(4M)$. Recall that such a leaked inner node is a child of a revealed parent node at depth $d - 1 > \log(2M)$. Multiple measurements, including $s$, with a total count $> t'$ are mapped to this parent node. That is, $s$ is mapped onto the same length-$(d-1)$ path together with other measurements with a total count of $> t'/2$. We further distinguish two cases, at least one of which must happen.

1. More than $\lambda$ other measurements are mapped to the length-$(d-1)$ path together with $s$. Let this number be $k > \lambda$. Then this case happens with probability $\binom{M}{k}/2^{(d-1)\cdot k} < (M/2^{(d-1)})^k < 2^{-k} < 2^{-\lambda}$.
2. At least 1 other measurement with count $> t'/(2\lambda)$ is mapped to the length-$(d-1)$ path together with $s$. As there are at most $M/(t'/2\lambda) = O(\lambda)$ such measurements, this case happens with probability $O(\lambda)/2^{d-1} = O(\lambda/M)$.

In summary, for a report of some measurement $s$, the relevant leakage in our system is the number of actual report of $s$ plus at least a binomial noise of parameter $\mathsf{Bin}(O(M), O(1/M))$ contributed by the fake reports. We conjecture that such noises are enough to hide the contribution of any single report.

## 8 Evaluation

We implement (in C++) the clients and the report server $\mathcal{S}$ following the (non-robust) protocol in Figure 2 and 3.

Below we report performances of the components in POPSTAR and compare them with prior threshold reporting systems STAR [18] and Poplar [13].

### 8.1 Implementation Details

Concretely, we choose $\mathbb{F} = \mathrm{GF}(2^{128})$, and use the NTL [6] library for polynomial evaluation and interpolation over $\mathbb{F}$.

We use SHA-256 to implement the two hash functions $H_s, H_p$ in the protocol, and use AES-GCM with

[6] https://libntl.org/

128-bit keys to implement the symmetric key encryption scheme $(\mathsf{Enc}, \mathsf{Dec})$. To ensure the encryption scheme is key-committing [2], we always append a 96-bit vector $\mathbf{0}$ to the encrypted messages. We use the CryptoPP [7] library for the above cryptographic primitives.

We use the half-gate GC scheme of [34] and the actively secure OT scheme of [15] to implement our ODPRF and OPRF protocols. We use existing implementations from the emp-toolkit library. [8]

All benchmarks are run using a desktop machine with 32 Gigabyte of memory and a Ryzen 7 3800x CPU. Our prototype implementations run only on a single thread. For computation time measurements, we report the average over 10 experiment runs.

**Client measurement sampling.** We follow the same sampling process as in [18] (STAR) to sample measurements from a Zipf power-law distribution with a support of $N = 10,000$ and parameter $s = 1.03$. Each client's report contains a sampled 256-bit measurement and a 256-byte message.

**Choosing the leakage parameter $\ell$.** Concretely, we benchmark three choices. As discussed in Section 7.2, choosing $\ell = 16$ is optimal for the setting of $1,000,000$ reports sampled from a Zipf power-law distribution with a support of $N = 10,000$ and parameter $s = 1.03$. We also benchmark a more aggressive choice $\ell = 8$, and a more conservative choice $\ell = 32$, which will be suitable when the reports are sampled from much smaller and larger supports respectively.

## 8.2 Communication Costs

Each client's communication with the server $\mathcal{O}$ consists of two parts: (1) receiving garbled circuits (GC) for double-PRF and PRF evaluations $v^{(1)}, \ldots, v^{(d)}$, and $u$; (2) running $\lambda$ 1-out-of-2 OT with $\lambda$-bit strings as the receiver to obtain input keys $K_x$ for evaluating the GCs.

Part (1) has size $1638 \cdot (\ell + 2) \cdot 2 \cdot \lambda$ bits using LowMC [3] as our choice of PRF, and [34], our choice of GC. Part (2) is much smaller compared to the former. We report concrete GC sizes in Table 2.

Each client's communication with the server $\mathcal{S}$ consists only of its report, which contains two field elements in the clear, $\ell$ encryptions of field element and tag pairs, and a final encryption of the client's measurement and message. We report concrete report sizes in Table 2.

## 8.3 Computational Costs

**Client computation time.** Client computation has two parts (Figure 2, and 4): (1) evaluating the GCs received from the server $\mathcal{O}$; (2) computing its report using the evaluations.

[7] https://www.cryptopp.com/

[8] https://github.com/emp-toolkit

| | $\ell = 8$ | $\ell = 16$ | $\ell = 32$ |
|---|---|---|---|
| GC size (MB) | 0.50 | 0.90 | 1.70 |
| Report Size (KB) | 0.93 | 1.49 | 2.62 |

Table 2: Communication sizes of a client with the randomness server $\mathcal{O}$ (first row), and with the report server $\mathcal{S}$ (second row).

| | Client interaction w/ $\mathcal{O}$ | Client com. w/ $\mathcal{S}$ | Client local comp. | Server $\mathcal{S}$ comp. |
|---|---|---|---|---|
| STAR | 1 round | 0.45 (KB) | 0.33* (ms) | 20 (s) |
| POPSTAR | 2 rounds | 1.49 (KB) | 4.82 (ms) | 136.1 (s) |

Table 3: Comparison between STAR (with 129-bit field) and POPSTAR (with $\ell = 16$) for 1 million reports and a threshold $t = 1,000$ (0.1%). The client computation time (*) for STAR excludes the VOPRF verification time.

The cost of (1) are $1.7/3.4/7.1$ ms when $\ell = 8/16/32$ respectively. We benchmark the cost of (2) for thresholds $t = 200$ to $t = 1,000$, i.e. 0.1% of $200,000$ to $1,000,000$ reports, and plot the combined computation cost of both steps in Figure 9a. We observe the computation cost of (2) increases linearly with $t$, which comes from expanding the double PRF and PRF evaluations $v^{(1)}, \ldots, v^{(\ell)}, u$ into degree $t+1$ polynomials, and computing their evaluations at a random point.

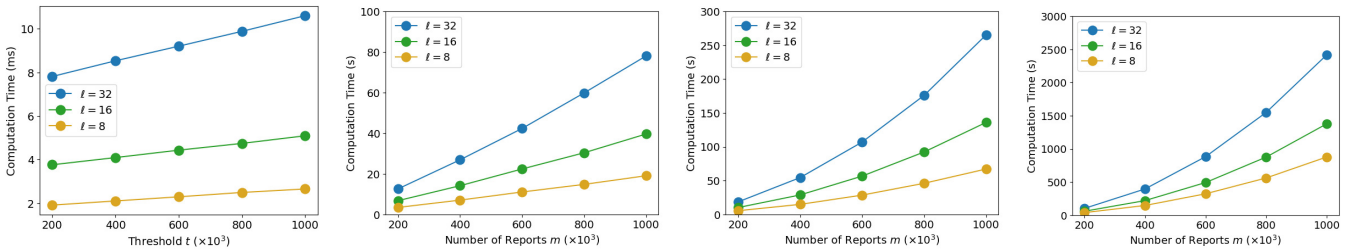
**Server computation times.** The computation cost of the server $\mathcal{O}$ consists mainly of preparing garbled circuits for each client. The costs per client are $1.9/4.1/8.4$ ms when $\ell = 8/16/32$ respectively, and don't depend on the total number of reports or the threshold. We benchmark the computation cost of the server $\mathcal{S}$ for aggregating $m = 200,000$ to $m = 1,000,000$ reports, and with thresholds at 0.01%, 0.1%, and 1% respectively. The results are plotted in Figure 9.

**Computation costs of OT.** The above computation times for each client and the server $\mathcal{O}$ exclude the costs of running $\lambda = 128$ OTs where the client is the receiver and the server $\mathcal{O}$ is the sender. Assuming the state-of-art protocol of [15], the main computation costs are $2\lambda$ and $2 + \lambda$ group exponentiations for the receiver and the sender respectively.

While the computation costs of group exponentiations are significant (6.5 ms for the client and 5.8 ms for the server $\mathcal{O}$), we argue that they are categorically different from the rest of the costs that we benchmark. The OT protocol consists of two round trips, [9] and the group exponentiations happen in between. In the end-to-end running time of $\lambda$ parallel OT protocols, the computation times are insignificant compared to the network latency (e.g., $\sim 50ms$ between AWS datacenters). [10] In contrast, the rest of the computations we

[9] The first sender OT message in the protocol of [15] can be reused across different OT instances. Assuming a PKI setup where every client learns this message from the server $\mathcal{O}$, we only need 1 round trip for each OT protocol.

[10] According to https://www.cloudping.co/grid.




(a) Client report generation time. (b) Server comp. time, $t = 100$. (c) Server comp. time, $t = 1000$. (d) Server comp. time, $t = 10000$.

Figure 9: The computation times for each client and the report server $\mathcal{S}$.

benchmark above happen entirely locally. Finally, we note that the group exponentiations during $\lambda$ OT protocols can be completely parallelized via multi-threading to increase throughput.

**Computation costs of GC.** We also note the cost of GC can be further reduced as our ODPRF protocol repeatedly garbles a fixed circuit, and some of the costs in managing the XOR gates are due to the generic nature of emp-toolkit. Instead of reading from a data structure in memory to determine the next gate to garble, a more optimized implementation can hard-code the circuit to avoid memory access costs.

**Heavier server $\mathcal{O}$ than server $\mathcal{S}$.** Throughout an aggregation session, the computation of the randomness server $\mathcal{O}$ is significantly ($\sim 70\times$) heavier than the aggregation server $\mathcal{S}$: under our main setting with 1 million reports at a 0.1% threshold and $\ell = 16$, the computation per client of the server $\mathcal{O}$ is 10 ms (4.1 ms for GC generation and 5.8 ms for OT), while that of the server $\mathcal{S}$ is 0.14 ms. (In fact, this is also true for STAR: as reported, its server $\mathcal{O}$ is $\sim 40\times$ heavier than its server $\mathcal{S}$.)

However, we believe the heavier server $\mathcal{O}$ does *not* create a bottleneck because its computation are distributed throughout an entire session as the clients asynchronously decide to make reports.

## 8.4 Comparing with STAR

In Table 3 we show the comparison in the setting of 1 million reports with a threshold $t = 1,000$ (0.1%) and $\ell = 16$. Overall, POPSTAR significantly reduces the leakage of STAR, at the cost of moderately increasing the computation times of each client ($\sim 15\times$) and the report server $\mathcal{S}$ ($\sim 7\times$). Each report in POPSTAR is roughly $3\times$ larger than in STAR.

Admittedly, each client's interaction with the randomness server $\mathcal{O}$ is significantly heavier in POPSTAR, both communication wise and computation wise due to our oblivious double PRF protocol based on GC and OT. However, we argue that the end-to-end running time of this interaction is bottlenecked by network latency rather than communication size or computation time. In POPSTAR, the oblivious double PRF protocol requires *two* round trips, assuming the OT protocol of [15], while in STAR the oblivious PRF protocol requires only *one* round trip. Therefore, we estimate each client's interaction with the server $\mathcal{O}$ to be $2\times$ to $3\times$ slower than in STAR. Finding a more efficient oblivious double PRF protocol is an intriguing direction for future work.

We note that in [18], the authors implemented STAR using the *partially* oblivious verifiable PRF protocol of [33]. The added verifiability in STAR lets a client detect whenever the randomness server deviates from the protocol and abort early, while the partially oblivious feature is not used. Our non-robust system uses an oblivious PRF without verifiability, hence giving up clients' ability to detect a malicious randomness server. To make the comparison fair, we exclude the verification time (0.301 ms) from the reported client computation time of STAR.

## 8.5 Comparing with Poplar

POPSTAR achieves a similar leakage profile to the hashing variant of Poplar ([13], Appendix B), while reducing the aggregation time dramatically. Note that the authors of [13] only benchmarked the more efficient variant without hashing. We use those reported numbers as an optimistic estimate for their hashing variant to compare with our system.

According to the benchmarking results in [13], the throughput for aggregating 1 million reports with a 0.1% threshold is $\sim 120/s$. Our randomness server $\mathcal{O}$ achieves a throughput of $100/s$, while our aggregation server $\mathcal{S}$ achieves more than $\sim 7300/s$. Comparing only the end-to-end aggregation time after all reports are collected, POPSTAR takes roughly 2 minutes while Poplar roughly 2 hours, i.e., a $\sim 60\times$ reduction. Communication wise, each client in POPSTAR needs to communicate $\sim 0.9$ MB while in Poplar, only 70 KB, i.e. a $\sim 13\times$ increase.

Again, we stress that we view the stateless nature of the server $\mathcal{O}$ and the decoupling between the two servers a much bigger benefit of POPSTAR.

## 9 Related Work

We briefly discuss alternative approaches (that do not rely on generic MPC) to privately compute heavy hitters.

**Single server aggregation.** Single server aggregation systems [8, 9, 12, 26, 27] allow the server to securely compute the sum of the clients' inputs. Melis et al. [28] shows that these systems can compute approximate heavy hitters using the count-min sketch data structure [16]. A drawback of these systems is that aggregation requires multiple rounds of interaction, hence need to tolerate client dropouts.

**Out-sourced MPC.** A common paradigm is to let each client secret share its input to multiple ($\geq 2$) servers, who then run a secure protocol to compute the heavy hitters. A subclass of such systems [1, 13, 17] (with two non-colluding servers) is formulated in [19] as verifiable distributed aggregation functions (VDAF). Their server protocols involve (1) a parallelizable phase where the shares are verified, and (2) a final phase where heavy hitters are computed. Other systems that do not fit the VDAF model include [7, 31, 32] (with two servers), and [11, 20, 23, 29] (with three servers). A challenge in deploying these systems is enlisting external entity(s) trusted not to be colluding, and willing to interact during the aggregation protocols.

**Two non-communicating servers.** STAR [18], as well as POPSTAR, involve two non-colluding servers that do not communicate with each other. One server, upon requests, provides randomness for clients to compute their reports. The other receives reports from the clients and computes the heavy hitters locally. To break the link between reports and clients, an abstract mixing server may be implemented as a buffer between the clients and the report server.

**Randomized response.** The systems for private analytics based on randomized response [5,6,14,30,35,36] involve each client just sending a message to a single server. A downside of these systems is that the clients' messages leak a non-negligible amount of information about their private inputs.

## 10 Conclusions

In this work, we have introduced POPSTAR, a threshold reporting system in the two server model, following the same architecture as STAR [18], and reducing its leakage at a moderate cost. Our prototype implementation is able to aggregate 1 million reports in $\sim 2$ minutes (roughly $7\times$ longer than STAR, but still within feasible range).

We provide an ideal functionality definition that captures the leakage in POPSTAR precisely, and a heuristic evaluation of this leakage profile. However, we believe further leakage analysis (both for POPSTAR and for STAR/Poplar) is needed to better understand leakage-abuse attacks. We pose this as an important open direction for future work which goes beyond the scope of this work.

## Acknowledgements

Hanjun Li was supported by a NSF grant CNS-2026774 and a Cisco Research Award.

Stefano Tessaro was supported in part by NSF grants CNS-2026774, CNS-2154174, a JP Morgan Faculty Award, a CISCO Faculty Award, and a gift from Microsoft.

## References

[1] Surya Addanki, Kevin Garbe, Eli Jaffe, Rafail Ostrovsky, and Antigoni Polychroniadou. Prio+: Privacy preserving aggregate statistics via boolean shares. In Clemente Galdi and Stanislaw Jarecki, editors, *Security and Cryptography for Networks - 13th International Conference, SCN 2022, Amalfi, Italy, September 12-14, 2022, Proceedings*, volume 13409 of *Lecture Notes in Computer Science*, pages 516–539. Springer, 2022.

[2] Ange Albertini, Thai Duong, Shay Gueron, Stefan Kölbl, Atul Luykx, and Sophie Schmieg. How to abuse and fix authenticated encryption without key commitment. In Kevin R. B. Butler and Kurt Thomas, editors, *USENIX Security 2022*, pages 3291–3308. USENIX Association, August 2022.

[3] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 430–454. Springer, Heidelberg, April 2015.

[4] Apple and Google. Exposure notification privacy-preserving analytics (enpa) white paper, 2021. Available at https://covid19-static.cdn-apple.com/applications/covid19/current/static/contact-tracing/pdf/ENPA_White_Paper.pdf.

[5] Raef Bassily, Kobbi Nissim, Uri Stemmer, and Abhradeep Thakurta. Practical locally private heavy hitters. *J. Mach. Learn. Res.*, 21:16:1–16:42, 2020.

[6] Raef Bassily and Adam D. Smith. Local, private, efficient protocols for succinct histograms. In Rocco A. Servedio and Ronitt Rubinfeld, editors, *47th ACM STOC*, pages 127–135. ACM Press, June 2015.

[7] James Bell, Adrià Gascón, Badih Ghazi, Ravi Kumar, Pasin Manurangsi, Mariana Raykova, and Phillipp

Schoppmann. Distributed, private, sparse histograms in the two-server model. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 307–321. ACM Press, November 2022.

[8] James Bell, Adrià Gascón, Tancrède Lepoint, Baiyu Li, Sarah Meiklejohn, Mariana Raykova, and Cathie Yun. ACORN: input validation for secure aggregation. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 4805–4822. USENIX Association, 2023.

[9] James Henry Bell, Kallista A. Bonawitz, Adrià Gascón, Tancrède Lepoint, and Mariana Raykova. Secure single-server aggregation with (poly)logarithmic overhead. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1253–1269. ACM Press, November 2020.

[10] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 784–796. ACM Press, October 2012.

[11] Jonas Böhler and Florian Kerschbaum. Secure multi-party computation of differentially private heavy hitters. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2361–2377. ACM Press, November 2021.

[12] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H. Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1175–1191. ACM Press, October / November 2017.

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

[14] Mark Bun, Jelani Nelson, and Uri Stemmer. Heavy hitters and the structure of local privacy. *ACM Trans. Algorithms*, 15(4):51:1–51:40, 2019.

[15] Tung Chou and Claudio Orlandi. The simplest protocol for oblivious transfer. In Kristin E. Lauter and Francisco Rodríguez-Henríquez, editors, *LATINCRYPT 2015*, volume 9230 of *LNCS*, pages 40–58. Springer, Heidelberg, August 2015.

[16] Graham Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. In Martin Farach-Colton, editor, *LATIN 2004*, volume 2976 of *LNCS*, pages 29–38. Springer, Heidelberg, April 2004.

[17] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In Aditya Akella and Jon Howell, editors, *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 259–282. USENIX Association, 2017.

[18] Alex Davidson, Peter Snyder, E. B. Quirk, Joseph Genereux, Benjamin Livshits, and Hamed Haddadi. STAR: Secret sharing for private threshold aggregation reporting. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 697–710. ACM Press, November 2022.

[19] Hannah Davis, Christopher Patton, Mike Rosulek, and Phillipp Schoppmann. Verifiable distributed aggregation functions. *Proc. Priv. Enhancing Technol.*, 2023(4):578–592, 2023.

[20] F. Betül Durak, Chenkai Weng, Erik Anderson, Kim Laine, and Melissa Chase. Precio: Private aggregate measurement via oblivious shuffling. Cryptology ePrint Archive, Paper 2021/1490, 2021. https://eprint.iacr.org/2021/1490.

[21] Tim Geoghegan, Christopher Patton, Brandon Pitman, Eric Rescorla, and Christopher A. Wood. Distributed Aggregation Protocol for Privacy Preserving Measurement. Internet-Draft draft-ietf-ppm-dap-09, Internet Engineering Task Force, December 2023. Work in Progress.

[22] Sharon Huang, Subodh Iyengar, Sundar Jeyaraman, Shiv Kushwah, Chen-Kuei Lee, Zutian Luo, Payman Mohassel, Ananth Raghunathan, Shaahid Shaikh, Yen-Chieh Sung, and Albert Zhang. DIT: Deidentified authenticated telemetry at scale, 2021. Technical report, Facebook Inc., https://research.fb.com/wp-content/uploads/2021/04/DIT-DeIdentified-Authenticated-Telemetry-at-Scale_final.pdf.

[23] Pranav Jangir, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, and Somya Sangal. Vogue: Faster computation of private heavy hitters. Cryptology ePrint Archive, Report 2022/1561, 2022. https://eprint.iacr.org/2022/1561.

[24] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. Round-optimal password-protected secret sharing and T-PAKE in the password-only model. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 233–253. Springer, Heidelberg, December 2014.

[25] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 177–194. Springer, Heidelberg, December 2010.

[26] Hanjun Li, Huijia Lin, Antigoni Polychroniadou, and Stefano Tessaro. LERNA: secure single-server aggregation via key-homomorphic masking. In Jian Guo and Ron Steinfeld, editors, *Advances in Cryptology - ASIACRYPT 2023 - 29th International Conference on the Theory and Application of Cryptology and Information Security, Guangzhou, China, December 4-8, 2023, Proceedings, Part I*, volume 14438 of *Lecture Notes in Computer Science*, pages 302–334. Springer, 2023.

[27] Yiping Ma, Jess Woods, Sebastian Angel, Antigoni Polychroniadou, and Tal Rabin. Flamingo: Multi-round single-server secure aggregation with applications to private federated learning. In *2023 IEEE Symposium on Security and Privacy*, pages 477–496. IEEE Computer Society Press, May 2023.

[28] Luca Melis, George Danezis, and Emiliano De Cristofaro. Efficient private statistics with succinct sketches. In *NDSS 2016*. The Internet Society, February 2016.

[29] Dimitris Mouris, Pratik Sarkar, and Nektarios Georgios Tsoutsos. PLASMA: Private, lightweight aggregated statistics against malicious adversaries with full security. Cryptology ePrint Archive, Report 2023/080, 2023. https://eprint.iacr.org/2023/080.

[30] Zhan Qin, Yin Yang, Ting Yu, Issa Khalil, Xiaokui Xiao, and Kui Ren. Heavy hitter estimation over set-valued data with local differential privacy. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 192–203. ACM Press, October 2016.

[31] Mayank Rathee, Conghao Shen, Sameer Wagh, and Raluca Ada Popa. ELSA: Secure aggregation for federated learning with malicious actors. In *2023 IEEE Symposium on Security and Privacy*, pages 1961–1979. IEEE Computer Society Press, May 2023.

[32] Kunal Talwar, Shan Wang, Audra McMillan, Vojta Jina, Vitaly Feldman, Bailey Basile, Áine Cahill, Yi Sheng Chan, Mike Chatzidakis, Junye Chen, Oliver Chick, Mona Chitnis, Suman Ganta, Yusuf Goren, Filip Granqvist, Kristine Guo, Frederic Jacobs, Omid Javidbakht, Albert Liu, Richard Low, Dan Mascenik, Steve Myers, David Park, Wonhee Park, Gianni Parsa, Tommy Pauly, Christian Priebe, Rehan Rishi, Guy N. Rothblum, Michael Scaria, Linmao Song, Congzheng Song, Karl Tarbe, Sebastian Vogt, Luke Winstrom, and Shundong Zhou. Samplable anonymous aggregation for private federated data analysis. *CoRR*, abs/2307.15017, 2023.

[33] Nirvan Tyagi, Sofía Celi, Thomas Ristenpart, Nick Sullivan, Stefano Tessaro, and Christopher A. Wood. A fast and simple partially oblivious PRF, with applications. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 674–705. Springer, Heidelberg, May / June 2022.

[34] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole - reducing data transfer in garbled circuits using half gates. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part II*, volume 9057 of *LNCS*, pages 220–250. Springer, Heidelberg, April 2015.

[35] Mingxun Zhou, Tianhao Wang, T.-H. Hubert Chan, Giulia Fanti, and Elaine Shi. Locally differentially private sparse vector aggregation. In *2022 IEEE Symposium on Security and Privacy*, pages 422–439. IEEE Computer Society Press, May 2022.

[36] Wennan Zhu, Peter Kairouz, Brendan McMahan, Haicheng Sun, and Wei Li. Federated heavy hitters discovery with differential privacy. In Silvia Chiappa and Roberto Calandra, editors, *The 23rd International Conference on Artificial Intelligence and Statistics, AISTATS 2020, 26-28 August 2020, Online [Palermo, Sicily, Italy]*, volume 108 of *Proceedings of Machine Learning Research*, pages 3837–3847. PMLR, 2020.