Vɛrity: Verifiable Local Differential Privacy

James Bell-Clark *Google*
Adrià Gascón *Google*
Baiyu Li *Google*
Mariana Raykova *Google*
Amrita Roy Chowdhury *University of Michigan*

Abstract

Local differential privacy (LDP) enables individuals to report sensitive data while preserving privacy. Unfortunately, LDP mechanisms are vulnerable to poisoning attacks, where adversaries controlling a fraction of the reporting users can significantly distort the aggregate output–much more so than in a non-private solution where the inputs are reported directly. In this paper, we present two novel solutions that prevent poisoning attacks under LDP while preserving its privacy guarantees. Our first solution, *Vɛrity-Auth*, addresses scenarios where the users report inputs with a ground truth available to a third party. The second solution, *Vɛrity*, tackles the more challenging case in which the users locally generate their input and there is no ground truth which can be used to bootstrap verifiable randomness generation.

1 Introduction

Collecting aggregate statistics while preserving individual privacy is of broad interest. Local differential privacy (LDP) provides a solution with strong privacy guarantees and minimal server requirements. Specifically, users perform all privacy-preserving computations locally, with the server only post-processing the responses to optimize utility. To achieve privacy, each user randomizes their input to introduce uncertainty about the individual values. However when aggregated across many users, the estimate still retains a weak signal that can be amplified and extracted. In other words, an LDP mechanism maintains utility by being highly sensitive to small changes in the distribution of the private inputs.

Unfortunately, the same property also leaves LDP mechanisms vulnerable to poisoning attacks, where malicious users submit false or malformed reports. In particular, poisoning the private reports can be far more harmful than poisoning the input itself. An adversary controlling a fraction of the users can disproportionately bias the server's analysis, creating a larger distortion than if the adversary only tampered with the users' inputs. Formally, under LDP, the adversary can amplify the impact of any poisoning of the input by a factor of $\frac{1}{\varepsilon}$, where ε is the privacy parameter [19].

In this work, we propose solutions that prevent poisoning attacks under LDP. In particular, we focus on randomized response (RR) [52], one of the most fundamental and widely adopted LDP mechanisms, which is used for reporting binary inputs. A key advantage of RR is its non-interactive nature – each user prepares and submits their contribution without needing to maintain state or engage in further interactions. This is especially beneficial for devices with limited connectivity that may not be online for extended periods. Additionally, it simplifies implementation and deployment. Therefore, a primary design goal for us is to preserve this valuable one-shot contribution feature of RR. We achieve this by enabling the users to generate a non-interactive cryptographic proof that their contribution results from a correct evaluation of RR.

Security Model. We consider adversaries who control a fraction of the users and can instruct them to tamper with their inputs or deviate from the RR mechanism, i.e., tamper with its randomness. Additionally, we allow malicious users to drop out and provide no response to the server, which can be modeled as submitting an invalid response (e.g., a non-binary value). Note that tampering or poisoning the input is always possible, even in the non-private case, whereas tampering with the randomizer is a characteristic of LDP. Hence, our security goal is to ensure that the output distribution matches that of an honest execution of RR directly on the users' inputs. Specifically, our solutions restrict the adversary's strategies to rely solely on the inputs—they cannot introduce additional bias by exploiting the randomness of RR.

Our Contributions

While a canonical application of RR involves reporting user-generated inputs, such as answers to survey questions, there are also scenarios where inputs have a ground truth available to a third party. For example, during the COVID-19 pandemic, exposure notification apps reported user infections across var-

ious demographics. In this case, the ground truth – whether a user tested positive – was available to the hospital conducting the tests. Another example is when a user reports some attributes about them in an online poll, such as, whether they are over 18, with the ground truth being a digital ID issued by the government. In the context of Privacy Sandbox [46], an initiative to design privacy-preserving advertising APIs, RR is used to collect information about whether an ad led to a purchase – the ground truth for this bit is available to the advertiser. In these cases, the input to RR can be *authorized* by the third party. These bits can be reported on their own by the user to the server if it's not possible to collect them from the authorizer. However, a more common scenario is when the bits are reported along with an associated payload, which is not known to the authorizer. For example, in an ad exchange, the payload contains information about the specific ad. Here, the *join* between the bit and the payload is known *only* to the user. We provide two novel constructions, *Vɛrity-Auth* and *Vɛrity*, for verifying RR for *both* input settings, i.e., with and without authorized inputs.

***Vɛrity-Auth*: With Authorized Input.** Verifiability in this setting involves two key tasks. First, the user must prove that they are using authorized inputs from the third party, hereafter referred to as the authorizer. Second, the user needs to demonstrate that the randomness used for RR was generated honestly. The high-level idea of *Vɛrity-Auth* is as follows. User obtains a signature from the authorizer on the input. The user then generates the randomness for RR by using a pseudorandom function (PRF) [31]. To ensure that the randomness is honestly generated while keeping it hidden from the server, the key for the PRF is derived jointly by both the user and the server. Finally, the user employs zero-knowledge proofs (ZKP) to demonstrate that their submitted report was generated by running RR on the signed (thereby, authorized) input, using the randomness generated by evaluating the PRF under the correct key. As mentioned above, we achieve this non-interactively.

Security. In addition to preventing tampering of randomizer, *Vɛrity-Auth* can also prevent input tampering due to the access to authorized inputs. This means that in the case of no dropouts, *Vɛrity-Auth* completely prevents poisoning attacks – i.e., the output distribution is exactly the same as if RR were executed honestly on the authorized inputs. While dropping out does give the adversary an opportunity to launch attacks, the effectiveness of such an attack now depends on the randomness of RR and the sum of the authorized bits – both of which are beyond the adversary's control.

Efficiency. *Vɛrity-Auth* is very efficient. Specifically, to authorize an input, a user takes 4.67ms while the authorizer runs in 2.05ms. To generate a verifiable RR bit from 3 pseudorandom bits, it takes the user 8.37ms to generate the required ZKP, where the proof is only 1.19KB; and it takes the server 3.54ms to verify both ZKP and signature. Compared to a concurrent work [14], that operates in a weaker threat model, *Vɛrity-Auth* is 100× more efficient.

***Vɛrity*: Without Authorized Input.** The scenario in which the user generates their own input is more challenging because merely ensuring the honest generation of randomness is not enough – a malicious user could still carry out arbitrary poisoning attacks by changing the input based on the sampled randomness (see Sec. 5). *Vɛrity* addresses this by keeping the randomness encrypted from the user. To preserve non-interactivity, we enable the user to evaluate the PRF under homomorphic encryption (HE). However, we must also enforce verifiability on top of this homomorphic computation. Achieving this through generic composition with ZKPs is extremely inefficient. Instead, *Vɛrity* adopts a different approach by leveraging the deterministic nature of homomorphically evaluated ciphertexts in the BGV scheme. Specifically, the server can verify the computation by rerunning the homomorphic operation on the encrypted inputs and checking that it obtains identical ciphertexts. To the best of our knowledge, this is first work to provide robustness by keeping the randomness encrypted.

Security. *Vɛrity* ensures the adversary cannot introduce more bias than by tampering with the inputs *alone*.

Efficiency. The user takes 1424s to homomorphically generate a verifiable RR bit from 3 pseudorandom bits, while the server verifies in 1473s. Using the alternative ZKP-based approach for our functionality is currently practically infeasible (see Sec. 9.2).

2 Background

2.1 Local Differential Privacy

We focus on the *local* model of DP which consists of a set of individual users (U) and an untrusted server.

Definition 1 (LDP [25])**.** *A randomized mechanism $\mathcal{M}: \mathcal{X} \to \mathcal{Y}$ satisfies* ε-LDP *if for any $x, x' \in \mathcal{X}$ and any $y \in \mathcal{Y}$, we have*

$$\Pr[\mathcal{M}(x) = y] \leq e^{\varepsilon} \Pr[\mathcal{M}(x') = y]$$

Randomized Response. Randomized response ($\mathsf{RR}_\rho : \{0,1\} \times [0,1] \to \{0,1\}$) is a classic LDP mechanism which releases a single bit $b \in \{0,1\}$ by flipping it with probability $\rho = \frac{1}{1+e^{\varepsilon}}$.

Theorem 1 ([25])**.** *RR_ρ satisfies* ε-LDP.

2.2 Cryptographic Primitives

Legendre PRF. We use the Legendre PRF [21], denoted as $\mathcal{F}: \mathbb{F}_p \times \{0,1\}^\kappa \to \{0,1\}$, where p is a κ-bit prime. This is a

one-bit PRF defined using the Legendre symbol $(\frac{a}{p})$ as:

$$\mathcal{F}(x,\mathsf{K}) = \left\lceil \frac{1}{2}\left(\left(\frac{\mathsf{K}+x}{p}\right)+1\right)\right\rceil. \quad (1)$$

The security of $\mathcal{F}$ as a PRF has been studied in [9, 32, 33, 47]. Let n be a quadratic nonresidue in $\mathbb{F}_p$. One can prove the evaluation of the Legendre PRF, i.e., $\mathcal{F}(x,\mathsf{K}) = b$ as follows:

- Prove that b is binary, i.e., $b(1-b) = 0$.
- If $b = 0$, set $w = \sqrt{n(K+x)}$; otherwise set $w = \sqrt{K+x}$.
- Prove that $w^2 = ((1-b)n+b)(K+x)$ with w as the witness.

Non-Interactive Zero Knowledge Argument of Knowledge. We use Schnorr's protocols [44] and Bulletproofs [16] under Fiat-Shamir transformation.

Pedersen Commitment. We use the Pedersen commitment over a cyclic group $\mathbb{G}$ of order p. Let P, Q be two distinct group generators. For $x \in \mathbb{Z}_p$, a Pedersen commitment of x is $\mathsf{com}_x = xP + rQ$, where $r \leftarrow\$ \mathbb{Z}_p$, and its opening is (x, r).

EQS Blind Signature. We use the equivalence class based blind signatures (EQS) [29], that can sign messages that are vectors of group elements $\vec{M} \in (\mathbb{G}_1^*)^l, l \in \mathbb{Z}_{>0}$:

- $(\mathsf{sk},\mathsf{pk}) \leftarrow$ **EQS.GenParam**(1^κ): On input the security parameter κ, the signer outputs a private parameter sk and a public parameter pk.
- $(\vec{M},\mathsf{st}) \leftarrow$ **EQS.SignRequest**(x,pk): On input $x \in \mathbb{Z}_p$ and a public parameter pk, the user outputs a signing request $\vec{M}$ for x and a local state st.
- $\sigma \leftarrow$ **EQS.Sign**$(\mathsf{sk},\vec{M})$: On input a signing request $\vec{M}$ and a private parameter sk, the signer outputs a signature σ.
- $\sigma' \leftarrow$ **EQS.Unblind**(st,σ): The user unblinds σ and obtains a signature σ' for their input x.

The exact construction is in the full paper [8]. Here, a signature for $\vec{M}$ can be adapted into signatures of any multiple $\mu\vec{M}, \mu \in \mathbb{Z}_p^*$. EQS signature satisfies the usual existential unforgeability under adaptive chosen-message attack (EU-CMA). For *blindness*, we consider the *malicious signer* (Def. 2 in [8]).

Threshold Homomorphic Encryption. We use a (2,2) threshold leveled homomorphic encryption scheme **HE**, where the master secret key is additively shared between two parties.

- $(\mathsf{pk},\mathsf{sk}_1,\mathsf{sk}_2) \leftarrow$ **HE.GenParam**(1^κ): On input the security parameter κ, **GenParam** outputs a public key pk and shares of the secret key $(\mathsf{sk}_1,\mathsf{sk}_2)$ for the two parties.
- $\mathsf{ct} \leftarrow$ **HE.Enc**(pk,x): The encryption algorithm outputs a ciphertext ct encrypting the plaintext input x.
- $\mathsf{ct}' \leftarrow$ **HE.Eval**$(\mathsf{pk},C,(\mathsf{ct}_i)_{i=1}^k)$: The evaluation algorithm takes a public key pk, a circuit C representing a k-ary function f, and a tuple of ciphertexts $(\mathsf{ct}_i)_{i=1}^k$, where each ct_i encrypts a plaintext input x_i, and it outputs a ciphertext ct of $f(x_1,\ldots,x_k)$.
- $d_i \leftarrow$ **HE.PartialDec**$(\mathsf{sk}_i,\mathsf{ct})$: The partial decryption algorithm takes a secret key share sk_i and a ciphertext ct, and it outputs a partial decryption result d_i.
- $x' \leftarrow$ **HE.Combine**(d_0,d_1): The recovery algorithm combines partial decryption results d_0 and d_1 from two parties, and it outputs a fully decrypted plaintext x'.

HE should satisfy IND-CPA security, and a partial decryption result d_i should hide the secret key share sk_i. We also require that the **HE** instantiation ensures **HE.Eval** is deterministic.

3 Problem Overview

3.1 Problem Setting

Without Authorized Input. Every user $\mathcal{U}_i, i \in [n]$ holds a private bit $x_i \in \{0,1\}$. The users report the corresponding noisy bits, $y_i = \mathsf{RR}_\rho(x_i)$, to an untrusted server, $\mathcal{S}$, which estimates the aggregate as $\hat{s} = \frac{1}{1-2\rho}(\sum_{i=1}^n y_i - \rho n)$[1]. This corresponds to our setup for *Verity*

With Authorized Input. Additionally, we consider the option of a third-party authorizer $\mathcal{A}$ holds the ground truth of the bits $x_i^{\mathcal{A}}$ which is the setup for *Verity-Auth*. As discussed in Sec. 1, in this case, the user holds additional information $\mathsf{payload}_i$ and reports $(y_i, \mathsf{payload}_i)$ to the server. The authorizer does not have access to the user's payload. In particular, the *join* of the private bit and the payload $(x_i, \mathsf{payload}_i)$ is known *only* to the user. The server here computes a histogram of the bits, bucketed by the payloads. In the extreme case where all bits end up in the same bucket, this reduces to aggregating the noisy y_is. Thus, the ℓ_1 error of the histogram is asymptotically bounded by that of the aggregate $\hat{s}$. For simplicity, we assume that the server computes the aggregate $\hat{s}$ instead of the histogram.

A natural example arises in COVID-19 exposure notification apps. Here, the server is the mobile app platform (such as, Google or Apple), while the authorizer is a hospital that conducts the user's COVID test. Each user holds a private bit x_i indicating infection status, known to both the user and the hospital. To enable aggregate reporting (such as, by region), users report a noisy version y_i of their status along with location or demographic metadata PL_i to the server. However, only the user knows the link between their identity in the hospital system and on the app (e.g., patient ID vs. device ID), and hence the mapping between x_i and PL_i. Verity-Auth

[1] $\hat{s}$ is the unbiased estimate of the sum of the privates bits, i.e., $\mathbb{E}[\hat{s}] = \sum_{i=1}^n x_i$.

Privacy		Authorized Input	Honest Randomness	ℓ_1 Error	Notes
No privacy		✗	N/A	m	Error only due to input tampering; randomizer tampering is not applicable.
		✓	N/A	0	Preventing input tampering completely eliminates poisoning attacks.
ε-LDP		✗	✗	$\tilde{O}(m(1+\frac{1}{\varepsilon})+\frac{\sqrt{n}}{\varepsilon})$	Corresponds to the worst-case attack. Observe that privacy increases the strength of poisoning. In particular, $\frac{1}{\varepsilon}$ factor comes from de-biasing while $\frac{\sqrt{n}}{\varepsilon}$ is the inherent error due to the noise to satisfy LDP.
		✓	✗	$\tilde{O}(m(1+\frac{1}{\varepsilon})+\frac{\sqrt{n}}{\varepsilon})$	Preventing just input tampering does not give any protection. A malicious user can still carry out the worst-case attack by tampering with the randomizer.
		✗	✓ Clear	$\tilde{O}(m(1+\frac{1}{\varepsilon})+\frac{\sqrt{n}}{\varepsilon})$	Revealing the noise in the clear provides no protection as the user can lie about their input and still carry out the worst-case attack.
			✓ Encrypted	$\tilde{O}(m+\frac{\sqrt{n}}{\varepsilon})$	This corresponds to *Vɛrity* (Fig. 6). The user is restricted to only input tampering which results in a weaker attack.
	Without drop-outs	✓	✓ Clear	$\Theta(\frac{\sqrt{n}}{\varepsilon})$	This corresponds to *Vɛrity-Auth* (Fig. 3). Poisoning attack is completely prevented; the only error term $O(\frac{\sqrt{n}}{\varepsilon})$ is inevitable due to noise.
			✓ Encrypted	$\Theta(\frac{\sqrt{n}}{\varepsilon})$	This corresponds to *Vɛrity-Auth*$^+$ (Fig. 7 in the full paper [8]). Encrypting the randomness is unnecessary and results in extra computational overhead.
	With drop-outs	✓	✓ Clear	$\tilde{O}(m(1+\frac{1}{\varepsilon})+\frac{\sqrt{n-m}}{\varepsilon})$ w. p. $\left(\frac{e^\varepsilon}{1+e^\varepsilon}\right)^q\left(\frac{1}{1+e^\varepsilon}\right)^{m-q}$	Malicious users can drop-out *based on the sampled randomness*. Consequently, they can still carry out the worst-case attack but with a much lower probability.
			✓ Encrypted	$\tilde{O}(q+\frac{\sqrt{n}}{\varepsilon})$	Drop-outs can be only on the basis of the inputs now which is inevitable.

Table 1: n denotes the total number of users, m is the number of malicious users and q is the sum of the input bits of all malicious users. Authorized input means that input tampering is prevented while honest randomness implies randomizer tampering is prevented. For the latter, we consider two scenarios - one where the randomness (equivalently, sample of random noise) is revealed in the clear to the user and the another, where it is kept hidden via encryption. Drop-out attack means a malicious user sends no response/ invalid response to the server; drop-outs can be based on either the sampled randomness or just the input. The above results assume the single message setting and unless explicitly stated, hold with arbitrarily high probability.

(a) Input Tampering (b) Randomizer Tampering

Figure 1: Poisoning attacks on an LDP protocol.

protects the *join* between the infection status and the user payload – ensuring that no party (including a colluding server and authorizer) can link x_i and PL_i to the same individual.

Other use cases similarly arise when a user obtains information from multiple parties that know different pseudonyms for the user (e.g., a user has distinct registration identifiers across service providers), yet it is desirable to report statistics over the join of these data sources. A prominent example occurs in the Privacy Sandbox initiative [46] for privacy-preserving online advertising. RR is used to collect conversion information (a bit indicating whether an ad led to a purchase), where the ground-truth bit is known to the advertiser, and the payload consists of metadata about the specific advertiser that triggered the conversion–information known *only* to the user.

We do *not* consider either the privacy or the authenticity of the payload. In other words, the user is free to select any payload to report with their bit. Since our primary goal is verifying the correctness of the LDP mechanism, authenticating the non-private payload is currently beyond the scope of our work.

We focus on the binary RR case and discuss the extension to r-ary RR in App. C.

3.2 Security Model

We consider a *malicious adversary* threat model:

- **Malicious User.** We consider a set of m malicious user. A malicious user can deviate from the protocol in any way they chose and send a *poisoned* response. Additionally, the malicious users are allowed to collude together.
- **Malicious Server.** We consider a malicious server that may arbitrarily deviate from the protocol to recover the inputs x_is.
- **Malicious Authorizer.** In the setting with authorized inputs, the authorizer may also act maliciously. Additionally, both the the server and the authorizer are allowed to collude together.

Here, we provide additional practical motivation and context for the authorizer's role as an adversary. Although the authorizer knows the private bit x_i, it cannot "lie" about this value–this behavior is moot for two reasons. First, the user does *not* receive the bit from the authorizer. Instead, both the user and the authorizer independently know the true private bit x_i. For example, in the advertising setting, both the user and the advertiser know whether a conversion occurred. Thus, the authorizer cannot misreport x_i to the user, because the user already knows the correct value. Second, the authorizer has a vested interest in the correctness of the final statistics computed over the join of the bit and the payload. In advertising, for instance, the advertiser wants accurate campaign-measurement results. Any attempt by the authorizer to misreport x_i would harm its

own ability to obtain meaningful aggregate information and runs counter to its incentives. Rather, the authorizer's natural adversarial goal is to violate the user's privacy–namely, to learn the *join* between a user's bit and their payload.

Impact of Poisoning. A typical LDP protocol (Fig. 1) processes a user's private input x through a *randomizer*, which adds noise to produce a noisy output or response y. This separation between input x and response y creates two points of attack for an adversary:

- **Input Tampering.** First, the adversary can tamper with the input, which occurs when users cannot access the randomizer's implementation—such as in mobile apps running proprietary code. In this case, a malicious user can only falsify their input from x_i to an arbitrary x_i', and report $y_i = \mathsf{RR}_\rho(x_i')$.
- **Randomizer Tampering.** The second opportunity occurs when the adversary controls the randomizer's implementation, such as by hacking the mobile app collecting data. In this case, a malicious user can submit an arbitrary response y_i instead of generating the correct noise to randomize their input.

Input tampering is possible for *any* protocol, private or not, because a user is free to change their input anytime. However, randomizer tampering attacks are unique to LDP – the distinction between a user's input and their response is a characteristic of LDP.

Randomizer tampering enables a strong poisoning attack:

Lemma 2 (Randomizer Tampering [19])**.** *Let us assume m malicious users. They can introduce an ℓ_1 error of $\tilde{O}(m(1+\frac{1}{\varepsilon})+\frac{\sqrt{n}}{\varepsilon})$ in the server's estimate, $\hat{s}$, with arbitrarily small probability of failure.*

Intuitively, the $\frac{\sqrt{n}}{\varepsilon}$ term is due to the error from RR_ρ. The $m(1+\frac{1}{\varepsilon})$ term comes from the adversarial behavior of the malicious users – m term is inevitable and accounts for the worst case scenario where all m malicious users are colluding and perturbs their responses in the same direction (i.e., say reports a 1), while the $\frac{1}{\varepsilon}$ factor corresponds to the scaling factor required for de-biasing.

If the malicious users are restricted to input tampering only, the efficacy of the attack weakens:

Lemma 3 (Input Tampering [19])**.** *If m users execute an input poisoning attack, they can skew the server's estimate $\hat{s}$ by an ℓ_1 error of $\tilde{O}(m+\frac{\sqrt{n}}{\varepsilon})$ with arbitrarily high probability.*

Compared to Lemma 2, the input tampering attack is weaker by a factor of $\frac{m}{\varepsilon}$. This demonstrates that vanilla LDP makes the protocol more vulnerable to poisoning attacks, as the adversary can introduce additional error by exploiting the randomness.

Drop Outs. In addition to the aforementioned attacks, malicious users can drop out, which can be modeled as submitting an invalid response (e.g., a non-binary value). Ensuring that users always provide valid responses requires incentives or enforcement mechanisms, which are application-specific and are orthogonal to our goals. We analyze the robustness of our solutions in both settings – when the adversary can cause users to drop out and when users are mandated to submit a valid response.

Drop outs are interesting only in the setting with verifiability. In the absence of verifiability, drop outs do not grant the adversary any advantage since they can fully control the value that each malicious user submits. However, when a user's response can be verified, the adversary may decide to drop out based on the randomness of RR (sampled noise). Hence, we aim to restrict the adversary to making the decision to drop out based solely on the input and *not* on the randomness sampled in RR. To the best of our knowledge, this is first paper to explicitly analyze drop out attacks for LDP. Table 1 summarizes the robustness guarantees of our solutions.

4 *Vɛrity-Auth*: With Authorized Input

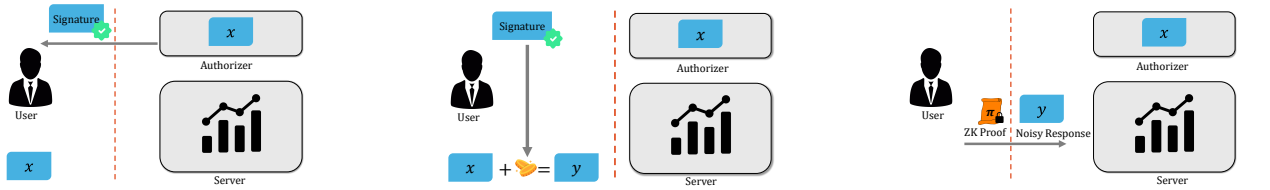
Here, we describe our solution, *Vɛrity-Auth*, that provides provable robustness against poisoning attacks when users have authorized inputs. In particular, *Vɛrity-Auth* prevents randomizer tampering by cryptographically enforcing the correct evaluation of RR. Additionally, *Vɛrity-Auth* prevents input tampering by having the authorizer perform a consistency check on the user's input against the ground truth, thereby ensuring its validity. We focus on the single-message setting, where each user holds exactly one private input to be reported.

4.1 Technical Intuition

Vɛrity-Auth works in three phases (Fig. 2). In the first phase[2], the authorizer, who has the ground truth of the input, performs a consistency check on the user's input bit. In case the consistency check passes, the authorizer issues a signature and sends it to the user. In the second phase, the user uses this signature to sample an instance of random noise, which is then used to generate the noisy bit. However, since the user could be malicious, we cannot trust them to have followed the steps so far correctly. Hence, in the last phase, the server needs to verify the correctness of the user's computation. To accomplish this, the user sends a cryptographic proof alongside the noisy bit to the server. The server then accepts the user's response only if the proof is valid. However, designing this three-phase protocol is not straightforward. In particular, our solution must satisfy the following desiderata.

- *Unlinkability.* First, the input consistency check must ensure unlinkability – the user's interaction with the autho-

[2]Note that obtaining an authorized input from $\mathcal{A}$ is inherent to our problem setting here and, therefore, is not considered a violation of the desired one-shot interaction.



(a) Authorizer checks the user's input and issues a signature if it matches the ground truth.

(b) User generates a noise sample using the signature to compute the noisy response.

(c) User sends the response and a ZKP of correct computation to the server for verification.

Figure 2: High-level overview of *Vɛrity-Auth*.

rizer in the first round and their interaction with the server in the final round should not be traceable to a *single* user through the messages communicated. More precisely, the user's report $(y_i, \mathsf{payload}_i)$ to the server should not be linked with the authorizer's ground truth $x_i^{\mathcal{A}}$. Otherwise they could collude to recover the exact join $(x_i, \mathsf{payload}_i)$, which is a violation of the user's privacy.

- *Non-Interactivity.* The second phase of the protocol involves generating a sample of noise securely. While previous work has addressed this issue [2, 10, 17, 24, 35, 36, 40], all existing solutions require interaction. However, as motivated in Sec. 1, we aim to securely sample noise non-interactively.
- *Efficiency.* The final task of verifying the user's computation is done with the help of ZKPs. However, ZKPs are typically extremely computationally intensive, so the challenge is to design an efficient solution that performs well in practice.

4.2 Workflow

The protocol can be broken into three phases with an initial one-time setup phase. The full protocol is outlined in Fig. 3.

Setup Phase. The setup phase is used to initialize the parameters for various cryptographic primitives. Specifically, the authorizer $\mathcal{A}$ generates the parameters for the EQS based blind signature scheme:

$$(\mathsf{pprm}_{\mathcal{A}}, \mathsf{privprm}_{\mathcal{A}}) \leftarrow \textbf{EQS.GenParam}(1^{\kappa}).$$

Additionally, each user $\mathcal{U}_i$ samples a secret key $\mathsf{sk}_i \leftarrow\$ \mathbb{F}_p$ for the Legendre PRF and sends a commitment to it, $\mathsf{com}(\mathsf{sk}_i)$, to the server. sk_i will be used to sample noise, which we will discuss later.

Input Authorization Phase. In a nutshell, the input consistency check is performed using a *blind signature* issued by the authorizer $\mathcal{A}$. Let $x_i^{\mathcal{A}}$ be the ground truth of $\mathcal{U}_i$'s input bit, which is known to the authorizer $\mathcal{A}$. The user $\mathcal{U}_i$ submits a blinded message containing a commitment com_{x_i} to their input bit x_i, along with a ZK proof of the opening of com_{x_i}. If the user acts truthfully, com_{x_i} should commit to the ground truth bit $x_i^{\mathcal{A}}$ – the authorizer validates the proof to ensure this consistency. If the proof is valid, the authorizer $\mathcal{A}$ issues a signature σ_i on the blinded message. The user then unblinds this signature to construct a new, valid signature σ_i', which is used later in the verification phase with the server. Specifically, σ_i' is obtained by *re-randomizing* σ_i which ensures the unlinkability of the user's interactions with the authorizer and the server. In what follows, we elaborate on the details of this construction.

As depicted in Fig. 2, the signature σ_i obtained from the authorizer is used in both of the two phases following the input authorization:

- A token τ_i is derived from the signature σ_i and is used for sampling noise.
- The re-randomized signature σ_i' is used in the verification phase to prove that the user performed their computation (of running RR) on an authorized input.

Additionally, it is important to *bind* the user's payload and secret key to their input x_i, *prior* to sampling noise. Recall that our overarching security goal is to prevent the adversary from exploiting the sampled noise of RR in any way. Hence, binding the payload beforehand prevents a malicious user from choosing the payload *after* observing the sampled noise. The rationale for binding the secret key is more involved. The high-level intuition is that, without such a binding, malicious users could swap inputs based on the token τ_i and carry out an attack. We explain this with an example in App. B.

This necessitates the ability to issue a blind signature on a vector of values, where the input bit, token τ_i, and the fingerprints of sk_i and $\mathsf{payload}_i$ are all components of the vector. We accomplish this by using an EQS-based blind signature with some modifications, which we describe below. Recall, that in the standard EQS scheme, the message is a vector of length two given by $\overrightarrow{M_i} = (h\mathsf{com}_{x_i}, hP)$ where $\mathsf{com}_{x_i} = x_i P + rQ$, with $r \leftarrow\$ \mathbb{Z}_p^*$ is a commitment to the input bit x_i, and P and Q are public parameters of the signature scheme (generators of the underlying bilinear group). The value $h \leftarrow\$ \mathbb{Z}_p^*$ is the blinding factor.

In contrast, *Vɛrity-Auth* employs messages of length five. Specifically, the third component is a (blinded) random value $\overrightarrow{M_i}[3] = hv_iP$ for $v_i \leftarrow\$ \mathbb{Z}_p^*$. Additionally, the two other components are the hashes of the secret key and the payload. The final message is given by:

$$\overrightarrow{M_i} = (h\mathsf{com}_{x_i}, hP, hv_iP, h\mathsf{H}_{\mathbb{G}_1^*}(\mathsf{com}_{\mathsf{sk}_i}), h\mathsf{H}_{\mathbb{G}_1^*}(\mathsf{payload}_i||\gamma_i))$$

Note that we concatenate the payload with a nonce γ_i, which is a random bit string of length determined by the security parameter κ. This is because of a technical detail in our security proof (see Sec. 7) which requires the fingerprint of the payload to form an equivocable commitment.

In addition, the user generates ZK proofs that the message is of the correct form. To be precise, this entails generating:

- $\pi_1 = \mathsf{ZKPoK}\{r \in \mathbb{Z}_p \mid \mathsf{com}_{x_i} = x_iP + rQ\}$
- $\pi_2 = \mathsf{ZK}\{\exists h \in \mathbb{Z}_p^* \mid \overrightarrow{M_i}[2] = hP \text{ and } \overrightarrow{M_i}[1] = h\mathsf{com}_{x_i}\}$

The vector components corresponding to sk_i and $\mathsf{payload}_i$ need not be checked, as these will be validated in the verification phase.

On receiving the message, the authorizer first verifies it. For an honest user, we must have $x_i = x_i^{\mathcal{A}}$, so the authorizer checks whether $\pi_{\overrightarrow{M_i}}$ shows a commitment to the ground truth $x_i^{\mathcal{A}}$. If the check passes, the authorizer randomly samples $c_i \leftarrow\$ \mathbb{Z}_p^*$ and issues a signature σ_i on a *modified* message, $\overrightarrow{M_i^{\mathcal{A}}}$, where the third component is set to $\overrightarrow{M_i^{\mathcal{A}}}[3] = c_i\overrightarrow{M_i}[3]$. The authorizer then sends c_i and σ_i to the user, who derives an unblinded and re-randomized signature σ_i' . Note that σ_i' is a signature on the message $\overrightarrow{M_i'} = (\mathsf{com}_{x_i}, P, c_iv_iP)$. The user sets the token $\tau_i = \mathsf{H}_{\mathbb{F}_p}(c_iv_iP)$, which will be used in the next phase. Here $\mathsf{H}_{\mathbb{F}_p} : \mathbb{G}_1^* \mapsto \mathbb{F}_p$ is a hash function. c_i ensures the token remains uniformly random, while v_i masks the token and provides unlinkability – we will elaborate on this later.

Note. We emphasize that blindness of the authorizer's signatures is *essential* to our construction. Each user must obtain two pieces of information from the authorizer without allowing the authorizer to link them back to the user. First, the user needs a signed commitment to the input bit, which is later used in a ZK proof to convince the server that the RR output y_i was computed correctly on an authorized input. Second, the user needs a token that will be used in the PRF for sampling the noise (as described in the following section). If the authorizer could see either the commitment or the token before signing, then a colluding authorizer and server could later match them to the user's submission, thereby violating unlinkability. Blind signatures prevent this by keeping the message of the signature hidden from the authorizer.

Vɛrity-Auth ensures input authorization with unlinkability via a new construction for blind signatures.

Noise Sampling Phase. The next step is to ensure that the user generates the correct noise sample. Previous works achieved this through interactive ZK protocols. While one might hope for a straightforward conversion of these protocols to a non-interactive one using the Fiat-Shamir transform, this approach is not applicable here. This limitation arises because here randomness is used not only for the probabilistic verification of deterministic statements, but also to generate probabilistic outputs [6]. As a result, a cheating prover (user) can carry out a poisoning attack without detection by simply using rejection sampling on the protocol. For instance, if a malicious user intends to report 1 regardless of their private input, they can resample protocol messages until they obtain an accepting transcript for an output of 1 (in expectation, the user needs to retry only $\lceil \frac{1}{\rho} \rceil$ times[3]).

We address this challenge by disentangling the noise sampling from the verification of its correctness. Specifically, we make the user sample noise *locally* in this phase. The task of verifying whether the user followed the protocol is delegated to the final verification phase – we design the sampling process in a way such that this verification is efficient.

In what follows, we elaborate on the local noise sampling process. Recall that RR_ρ involves flipping the private bit x_i with some probability ρ. This implies that the required noise is essentially a random sample drawn from $Ber(\rho)$. To generate a sample $b_{\rho,i} \sim Ber(\rho)$, i.e., a biased bit, proceed as follows: sample a uniform number $r \in [0,1]$ and then output heads if $r \le \rho$ and tails otherwise. This approach can be implemented with statistical error at most 2^{-k} by discretizing the sampling of r. Specifically, one can sample k bits uniformly at random, interpret these bits as the binary expansion of r, and then perform the comparison. Based on this observation, a user can locally sample the biased bit by generating uniform random bits with the aid of a PRF. In particular, we use the Legendre PRF for this purpose – $\mathcal{F}(j, K_i)$ gives us the j-th bit b_{ij} where K_i is the key of the PRF for user $\mathcal{U}_i$. This choice offers a two-fold advantage. First, the output domain of the Legendre PRF is binary, providing uniform random bits directly. Second, the evaluation of Legendre PRFs can be verified efficiently, as explained later.

The remaining detail is to discuss the key K_i of the PRF. Specifically, we need to ensure the following properties:

- K_i has to be kept secret from the server. This is to ensure that the noise $(b_{\rho,i})$ is hidden, else it would violate DP.
- K_i cannot be generated completely by the user. This is because a malicious user can select K_i strategically via rejection sampling to force the noisy bit to any value of their choosing.

[3] Even if the malicious user's true input is 0 and they follow the protocol honestly, the final response is 1 with probability ρ, implying the expected number of retries is $\lceil \frac{1}{\rho} \rceil$.

We resolve this by splitting the key into two parts as follows:

$$\mathsf{K}_i = \underbrace{\mathsf{sk}_i}_{\text{User's secret}} + \underbrace{\tau_i}_{\text{Authorizer's token}} \tag{2}$$

The first component is a secret sk_i chosen by the user which ensures the privacy of the noise added. The second component is the token $\tau_i = \mathsf{H}_{\mathbb{F}_p}(c_i v_i)$ derived in the previous phase. Recall that for the token, c_i is a random value chosen by the authorizer which prevents the user from influencing the process of key generation. Once $b_{\rho,i}$ is generated, the input bit can be flipped as:

$$y_i = (1-x_i)b_{\rho,i} + x_i(1-b_{\rho,i}) \tag{3}$$

Vɛrity-Auth samples noise non-interactively by having the users locally evaluate a PRF on a *well-formed* key.

Verification of Computation Phase. In the final phase, the user communicates the noisy response y_i and the payload $\mathsf{payload}_i$ to the server. However, since *Vɛrity-Auth* must account for malicious users, the server needs to verify the correctness of the user's computation. To this end, the user $\mathcal{U}_i$ also sends a set of ZK proofs, along with the unblinded signature σ'_i on the message $\overrightarrow{M'_i}$ and the nonce γ_i. Specifically, the server needs to verify two pieces of computation:

1. *Evaluation of RR_ρ.* To prevent poisoning attacks that could involve tampering with the randomizer, the server must verify that RR_ρ was evaluated correctly. This involves two steps: (1) verifying the correct generation of uniform random bits b_{ij}'s using the PRF with the appropriate key K_i, and (2) ensuring the input bit was flipped correctly based on the sampled noise.

- *Evaluation of the PRF.* The typical way of expressing a computation for ZKPs is via an arithmetic circuit[4]. In other words, computations that can be expressed as small-depth arithmetic circuits lead to better performance for ZKPs. Our choice of the Legendre PRF is ZKP-friendly. Specifically, for a public quadratic non-residue n_{ij}, the PRF evaluation $b_{ij} = \mathcal{F}(j, K_i)$ in our protocol can be efficiently proven in ZK using the following relation with a simple arithmetic circuit

$$\mathcal{R}_{b_{ij}} = \left\{ ((\mathsf{com}_{\mathsf{sk}_i}, b_{ij}), (\mathsf{sk}_i, w_{ij})) \;\middle|\; \begin{array}{ll} \mathsf{sk}_i = \mathsf{Open}(\mathsf{com}_{\mathsf{sk}_i}), & (4)\\ b_{ij}(1-b_{ij}) = 0, & (5)\\ w_{ij}^2 - ((1-b_{ij})n_{ij} + b_{ij})(\mathsf{K}_i + j) = 0 & (6)\\ \text{where} & \\ \mathsf{K}_i = \mathsf{sk}_i + \mathsf{H}_{\mathbb{F}_p}(\overrightarrow{M'_i}[3]) & (7)\\ w_{ij} = (1-b_{ij}) \cdot \sqrt{n_{ij}(\mathsf{K}_i + j)} + b_{ij} \cdot \sqrt{\mathsf{K}_i + j} & (8) \end{array} \right\}$$

In the above relation, Eqs. (4) and (7) ensure that the correct key is used, while Eqs. (5), (6) and (8) demonstrate the correctness of the Legendre symbol computation. It is easy to see that multiple proofs of PRF evaluations on the same key K_i can be efficiently merged into a single ZK proof $\pi_{i,1}$ by concatenating their circuits.

- *Input bit flipping.* Recall, that the LDP randomizer, RR_ρ, flips the input bit based on the Bernoulli sample, $b_{\rho,i}$. The standard process of sampling $b_{\rho,i}$ involves a comparison operation, which is costly under ZKPs because it is a non-arithmetic operation. To address this challenge, we choose a ZKP-friendly noise distribution. Specifically, we set the bias of the Bernoulli distribution to be a power of $\frac{1}{2}$, i.e., $\rho = \frac{1}{2^k}$ where $k = \lfloor \log_2(1+e^\varepsilon) \rfloor$ (see App. A for a discussion on the resulting privacy parameter). In this way, the Bernoulli sample can be generated by a simple product of all the k uniform random bits $b_{\rho,i} = \prod_{j=1}^{k} b_{ij}$ and thereby, avoid the costly non-arithmetic comparison operation. The correct flipping of the input bit can then be proven in a ZKP π_{y_i} for the following relation:

$$\mathcal{R}_{y_i} = \left\{ ((\mathsf{com}_{x_i}, y_i), (x_i, b_{\rho,i})) \;\middle|\; \begin{array}{ll} b_{\rho,i} & = \prod_{j=1}^{k} b_{ij}, \\ y_i & = (1-x_i)b_{\rho,i} \\ & \quad + x_i(1-b_{\rho,i}) \end{array} \right\} \tag{9}$$

The above proof, π_{y_i}, together with the proof of correct evaluation of the PRF, $\pi_{i,1}$, guarantees the correct evaluation of RR_ρ, thereby preventing randomizer tampering.

2. *Use of authorized input.* The final task is to verify that the user $\mathcal{U}_i$ computed on an authorized input, i.e., the input bit used for evaluating RR_ρ was checked by the authorizer $\mathcal{A}$. This check prevents input tampering attack. To accomplish this, the user produces a signature and message pair $(\sigma'_i, \overrightarrow{M'_i})$ and a ZKP π_{x_i} such that:

- A ZKP showing that the message $\overrightarrow{M'_i}$ corresponds to a commitment to the flipped bit which is given by:

$$\pi_{x_i} = \mathsf{ZK}\{\mathsf{Open}(\mathsf{com}_{x_i}) = \mathsf{Open}(\overrightarrow{M'_i}[1])\} \tag{10}$$

where com_{x_i} is as defined in Eq. 9.

- $(\sigma'_i, \overrightarrow{M'_i})$ is a valid signature issued by the authorizer, i.e., σ'_i verifies under the authorizer's public key $\mathsf{pprm}_{\mathcal{A}}$.

Thus, the ZK proof π_{x_i} ties the authorizer's signature to the user's computation. Due to the unforgeability of the signature, the user can only pass the above check if they used an authorized input $(\overrightarrow{M'_i}[1])$ in their computation.

To recall, the server obtains $y_i, \mathsf{payload}_i$, along side signature σ'_i, ZK proofs $\pi_i = (\pi_{i,1}, \pi_{y_i}, \pi_{x_i})$, the message $\overrightarrow{M'_i}$ and the nonce γ_i from user $\mathcal{U}_i$. The server accepts $\mathcal{U}_i$'s response only if (1) all the proofs are valid, (2) σ'_i is a valid signature on

[4] Arithmetic circuits use addition and multiplication gates over a finite field.

$\overrightarrow{M_i'}$, and (3) the message contains the correct payload and the secret key registered during the setup phase, i.e.,

$$\overrightarrow{M_i'}[4] = \mathsf{H}_{\mathbb{G}_1^*}(\mathsf{com}_{\mathsf{sk}_i}) \text{ and } \overrightarrow{M_i'}[5] = \mathsf{H}_{\mathbb{G}_1^*}(\mathsf{payload}_i || \gamma_i)$$

Note that $\mathcal{U}_i$'s interactions with the authorizer and the server remain unlinkable because:

- σ_i' is re-randomized which is unlinkable by the property of the EQS blind signature.
- The mask v_i is unknown to the server/authorizer and thereby prevents linking of the signatures via the token τ_i.

Vɛrity-Auth provides unlinkability, assuming there are no side-channel attacks, such as timing attacks. This can be enforced by anonymizing user messages via shuffling before sending them to the server [50, 51].

Vɛrity-Auth ensures efficient verification through a ZKP-friendly implementation.

Practical Considerations. We reiterate that *Vɛrity-Auth* is designed for a single-message setting in which each user should, under normal operation, obtain exactly one signature from the authorizer. In practice, however, two situations require special handling. First, a malicious user may attempt to submit multiple distinct blinded messages in order to obtain multiple signatures, hoping to bias the resulting randomness. Second, honest users may legitimately need a reissued signature due to transient infrastructure failures, such as, network interruptions or device-side transmission errors. To prevent abuse while still accommodating genuine retries, the authorizer can maintain some per-user state: it can record the user identifier together with the signature previously issued to that user. If the same user requests another signature, the authorizer simply returns the earlier one. This ensures that malicious users cannot obtain multiple signatures, while honest users can recover gracefully from short-lived failures. Typically, these data collection happen over a bounded, rolling time window (e.g., one week), making the required state retention lightweight and operationally manageable [11, 22, 26].

A subtlety arises when an honest user loses access to the underlying message itself (more precisely, the blinding factors for the message h and for the embedded commitment), for example due to a device crash that wipes local storage. Because this information is known only to the user, they can no longer produce the downstream ZK proofs and therefore must drop out of the protocol. Large-scale telemetry deployments routinely tolerate small rates of user dropout due to device failures, and such events do not meaningfully affect aggregate statistics.

We now outline two strategies that can further reduce the authorizer's space requirements.

(1) Limited reissue counter. Recall that, in expectation, a malicious user would require $\lceil 1/\rho \rceil$ attempts to obtain favorable randomness, where ρ is the bit-flip probability. Instead of enforcing a strict "no new signatures" policy, the authorizer may allow $k < \lceil 1/\rho \rceil$ retries per user tracked via a small per-user counter (far smaller than storing a full signature). This accommodates honest users who have lost access to their signed message due to device failure, while still restricting attackers to a bounded window for a poisoning attempt–succeeding with probability $1-(1-\rho)^k$.

(2) Space-efficient membership filters. To further reduce state overhead, the authorizer may maintain only a compact membership data structure–such as, a Bloom filter or Cuckoo filter [12, 27]–representing the set of users who have already received a signature. This eliminates the need to store full `(user ID, signature)` pairs. While it prevents malicious users from obtaining multiple signatures, it also inherently disallows all legitimate retries. Moreover, false positives may cause a small fraction of first-time users to be incorrectly denied; however, with appropriate parameterization, such false positives can be made negligible.

5 *Vɛrity*: Without Authorized Input

In this section, we explore the scenario where access to an authorized input is *unavailable*. Hence, a malicious user is free to lie and tamper with their input. Consequently, the only preventive measure against poisoning is ensuring the honest generation of randomness. However, a challenge arises – if the user can observe the noise sample in the clear, even if it is honestly generated, they could still mount arbitrary attacks by maliciously setting their input *based on* the generated sample. For an illustration, consider a malicious user who wishes to report the value 1. If the generated noise sample is $b_{i,\rho} = 1$ (i.e., the input bit will be flipped), the user set their input $x_i = 1$, and they set $x_i = 0$ otherwise. We tackle this challenge with *Vɛrity*. The key idea is to restrict the users to generating only an *encrypted* noise sample. This ensures that any tampering with the inputs remain *independent* of the noise, providing the strongest protection in the absence of authorized inputs.

5.1 Technical Intuition

In *Vɛrity*, users generate encrypted noise sample via a HE scheme. Specifically, users homomorphically evaluate the PRF to generate a sequence of uniform random bits, which are then converted to a Bernoulli sample. In particular, *Vɛrity* uses a (2,2)-threshold HE scheme where the secret key is shared between both the user and the server. This restricts the information that can be decrypted by both parties – user $\mathcal{U}_i$ should not be able to decrypt the response y_i while the server should not be able to decrypt anything beyond the final response. Now the main task is to efficiently verify this

Parameters. ε - Privacy parameter; κ - Security parameter

- **Setup Phase.**
  Authorizer:
  - Generates the parameters from the blind signature scheme $(\mathsf{pprm}_{\mathcal{A}}, \mathsf{privprm}_{\mathcal{A}}) \leftarrow$ **EQS.GenParam**(1^{κ})

  User: Each user $\mathcal{U}_i$
  - Samples $\mathsf{sk}_i \leftarrow\!\!\$\, \mathbb{F}_p$, sends a commitment $\mathsf{com}_{\mathsf{sk}_i} \leftarrow \mathsf{Com}(\mathsf{sk}_i)$ and proof $\pi_{\mathsf{sk}_i} = \mathsf{ZKPoK}\{\mathsf{sk}_i \mid \mathsf{sk}_i = \mathsf{Open}(\mathsf{com}_{\mathsf{sk}_i})\}$ to the server

  Server: For each user $\mathcal{U}_i$
  - Rejects $\mathcal{U}_i$ if π_{sk_i} does not verify
- **Input Authorization Phase.**
  User: Each user $\mathcal{U}_i$
  - Prepares the signature request using x_i, sk_i and payload: $(\overrightarrow{M_i}, \pi_{\overrightarrow{M_i}}, \gamma_i, \mathsf{st}_i) \leftarrow$ **EQS.SignRequest**$(x_i, \mathsf{payload}_i, \mathsf{com}_{sk_i}, \mathsf{pprm}_{\mathcal{A}})$
  - Stores the state information st_i with itself
  - Sends the message $\overrightarrow{M_i}$ and the associated proof $\pi_{\overrightarrow{M_i}}$ to the authorizer: $\mathcal{U}_i \xrightarrow{\overrightarrow{M_i}, \pi_{\overrightarrow{M_i}}} \mathcal{A}$

  Authorizer: For each user $\mathcal{U}_i$
  - Verifies the proof: $b =$ **EQS.VerifyMsg**$(\overrightarrow{M_i}, x_i^{\mathcal{A}}, \pi_{\overrightarrow{M_i}})$
  - If $b = 1$, issues a blind signature on the message and sends it to the user: $\sigma_i, c_i \leftarrow$ **EQS.Sign**$(\overrightarrow{M_i}, \mathsf{sk}_{\mathcal{A}}), \mathcal{A} \xrightarrow{\sigma_i, c_i} \mathcal{U}_i$

  User: Each user $\mathcal{U}_i$
  - Unblinds the signature to obtain a new signature and token τ_i: $(\overrightarrow{M_i'}, \sigma_i') \leftarrow$ **EQS.Unblind**$(\sigma_i, \mathsf{st}_i, c_i), \tau_i = H_{\mathbb{F}_p}(\overrightarrow{M_i'}[3])$
- **Noise Sampling Phase.**
  User: Each user $\mathcal{U}_i$
  - Sets $k = \left\lfloor \log_2(1 + e^{\varepsilon}) \right\rfloor$ and the Legendre PRF key $\mathsf{K}_i = \mathsf{sk}_i + \tau_i$
  - For all $j \in [k]$, computes $b_{ij} = \mathcal{F}(\mathsf{K}_i, j)$
  - Computes the biased bit (sample of noise) $b_{\rho,i} = \prod_{j=1}^{k} b_{ij}$ and the noisy response $y_i = b_{\rho,i} \cdot (1 - x_i) + (1 - b_{\rho,i}) \cdot x_i$
- **Verification of Computation Phase.**
  User: Each user $\mathcal{U}_i$
  - Selects quadratic non-residues n_{ij} for $j \in [k]$, and generates a zero knowledge proof for showing
    - Correct PRF evaluation $\pi_{i,1} = \left(\{n_{ij}\}, \mathsf{ZK} \left\{ \begin{array}{c} \mathsf{sk}_i = \mathsf{Open}(\mathsf{com}_{sk_i}), \\ \forall j \in [k], b_{ij}(1 - b_{ij}) = 0 \text{ AND } w_{ij}^2 - ((1 - b_{ij})n_{ij} + b_{ij})(\mathsf{K}_i + j) = 0 \text{ where} \\ \mathsf{K}_i = \mathsf{sk}_i + \mathsf{H}_{\mathbb{F}_p}(\overrightarrow{M_i'}[3]), w_{ij} = \sqrt{n_{ij}(\mathsf{K}_i + j)} \text{ if } b_{ij} = 0, \text{ or } w_{ij} = \sqrt{\mathsf{K}_i + j} \end{array} \right\} \right)$
    - Correct flipping of the authorized input bit: $\pi_{i,2} = \mathsf{ZK}\{y_i = b_{\rho,i} \cdot (1 - x_i) + (1 - b_{\rho,i}) x_i \text{ where } x_i = \mathsf{Open}(\overrightarrow{M_i'}[1])\}$
  - Sends the noisy response y_i, $\mathsf{payload}_i$, $\pi_i = (\pi_{i,1}, \pi_{i,2})$, message, signature, nonce to the server: $\mathcal{U}_i \xrightarrow{y_i, \mathsf{payload}_i, \pi_i, \overrightarrow{M_i'}, \sigma_i', \gamma_i} \mathcal{S}$

  Server: For each user $\mathcal{U}_i$
  - Verifies the proof π_i using the commitment $\mathsf{com}_{\mathsf{sk}_i}$ received in the setup phase, reject $\mathcal{U}_i$ otherwise
  - Verifies the signature σ_i' as $b \leftarrow$ **EQS.VerifyToken**$(\sigma', M_i', \mathsf{pprm}_{\mathcal{A}}, \mathsf{com}_{\mathsf{sk}_i}, \mathsf{payload}_i, \gamma_i)$, reject $\mathcal{U}_i$ if invalid

Figure 3: *Vɛrity-Auth*: Verifiable randomized response where the users have authorized inputs.

computation. A natural solution is to rely on ZK proofs [38] or homomorphic MACs [13, 28]. In this approach, generating the ZK proof (or MAC) is computationally expensive but verifying the proof is efficient, making it suitable for settings where the verifier is resource-constrained. A typical example is verifiable outsourced computation, where the resource-constrained user outsources a computation-heavy task to an untrusted server, which is one of the most common use cases for HE schemes. However, the situation is reversed in our setting – the user is tasked with performing the homomorphic computation, which makes the above solution ill-suited. *Vɛrity* solve this by taking a completely different approach to verification by leveraging the deterministic nature of the HE ciphertexts. The server replicates the entire computation on its side and checks that the generated ciphertexts match. If the user behaves honestly, the server's computation should yield exactly the same ciphertext as the user's. By shifting the computational load to the resource-heavy server, *Vɛrity*, thus, takes advantage of the architectural constraints of our setting.

5.2 Workflow

Vɛrity consists of two rounds, preceded by a one-time setup phase. In particular, it does not include the input authorization round as in *Vɛrity-Auth*. Due to space constraint, the full protocol is outlined in Fig. 6.

Setup Phase. The one-time setup phase is used to initialize the keys for the threshold HE scheme and the PRFs. First, the server and each user $\mathcal{U}_i$ jointly sample the parameters of the HE scheme $(\mathsf{pk}_i^{\mathsf{HE}}, \mathsf{sk}_{i,\mathcal{U}}^{\mathsf{HE}}, \mathsf{sk}_{i,\mathcal{S}}^{\mathsf{HE}}) \leftarrow \textbf{HE.GenParam}(1^\kappa, 2, 2)$ where $\mathsf{sk}_{i,\mathcal{S}}^{\mathsf{HE}}$ and $\mathsf{sk}_{i,\mathcal{U}}^{\mathsf{HE}}$ are the secret keys of the server and $\mathcal{U}_i$, respectively. As elaborated in the following paragraph, the secret key for the PRF associated with $\mathcal{U}_i$ is also split between the two parties. Specifically, the server and the user $\mathcal{U}_i$ each sample their respective shares of the PRF secret key – $\mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}} \in \mathbb{F}_p$ and $\mathsf{sk}_{i,\mathcal{U}}^{\mathcal{F}} \in \mathbb{F}_p$, respectively, for $\mathcal{U}_i$'s PRF. Since both parties need to homomorphically evaluate the PRF, the key shares must be encrypted and exchanged between them. Additionally, both parties generate a ZK proof of the correctness of this encryption and send it to each other for verification. To be precise, the server sends the following to $\mathcal{U}_i$:

$$\mathsf{ct}_{\mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}}} \leftarrow \textbf{HE.Enc}(\mathsf{pk}_i^{\mathsf{HE}}, \mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}}),$$
$$\pi_{\mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}}} \leftarrow \mathsf{ZKPoK}\{\mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}} \in \mathbb{F}_p \mid \mathsf{ct}_{\mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}}} = \textbf{HE.Enc}(\mathsf{pk}_i^{\mathsf{HE}}, \mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}})\}.$$

As we will instantiate **HE** using a RLWE-based HE scheme, the proof $\pi_{\mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}}}$ is implemented to prove in ZK the knowledge of secret $\mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}}$ and $\mathsf{sk}_i^{\mathsf{HE}}$, encryption randomness r, and encryption noises e_0, e_1 such that

- $\mathsf{ct}_{\mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}}} = \mathsf{pk}_i^{\mathsf{HE}} \cdot r + (e_0, e_1) + (\Delta \cdot \mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}}, 0)$ over the polynomial ring defined by HE; and
- r, e_0, e_1 all have bounded norms according to their distributions.

Similarly, the user $\mathcal{U}_i$ sends $(\mathsf{ct}_{\mathsf{sk}_{i,\mathcal{U}}^{\mathcal{F}}}, \pi_{\mathsf{sk}_{i,\mathcal{U}}^{\mathcal{F}}})$ to the server. Finally, both the parties initialize a counter $\mathsf{cnt}_i = 0$ which tracks the number of messages exchanged between them.

Noise Sampling Phase. The sampling process is the same as *Vɛrity-Auth* with some modifications to accommodate homomorphic evaluation. In particular, $\mathcal{U}_i$ uses a PRF key $\mathsf{K}_i = \mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}} + \mathsf{sk}_{i,\mathcal{U}}^{\mathcal{F}} + \mathsf{cnt}_i$. Here $\mathsf{sk}_{i,\mathcal{U}}^{\mathcal{F}}$ ensures that the sampled noise is kept private from the server, while $\mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}}$ prevents the user $\mathcal{U}_i$ from generating the PRF key on their own maliciously. These two components together ensure that the key is well-formed, as discussed in Sec. 4. Additionally, the counter cnt_i ensures that a different key is used for each message, providing a fresh noise sample every time. [5] Re-arranging the equations, the noisy response is then given by:

$$\begin{aligned} y_i &= f_{\mathsf{cnt}_i}(\mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}}, \mathsf{sk}_{i,\mathcal{U}}^{\mathcal{F}}, x_i) \\ &= x_i + (1 - 2x_i) \prod_{j=1}^{k} \mathcal{F}(\mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}} + \mathsf{sk}_{i,\mathcal{U}}^{\mathcal{F}} + \mathsf{cnt}_i, j). \end{aligned}$$

The user $\mathcal{U}_i$ encrypts their input x_i as ct_{x_i} and generates a ZK proof of correctness of the encryption, π_{x_i}, similar to the ZK proofs $\pi_{\mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}}}$ and $\pi_{\mathsf{sk}_{i,\mathcal{U}}^{\mathcal{F}}}$ above. $\mathcal{U}_i$ is now ready to compute the ciphertext of the response as:

$$\mathsf{ct}_{y_i} \leftarrow \textbf{HE.Eval}(\mathsf{pk}_i^{\mathsf{HE}}, \mathsf{ct}_{\mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}}}, \mathsf{ct}_{\mathsf{sk}_{i,\mathcal{U}}^{\mathcal{F}}}, \mathsf{ct}_{x_i}, f_{\mathsf{cnt}_i})$$

At this point, the user can only *partially* decrypt the response (equivalently the noise sample $b_{\rho,i}$) because of *Vɛrity*'s usage of threshold HE scheme. This ensures that the noise sample remains hidden from the user in *Vɛrity*. The user $\mathcal{U}_i$ proceeds with the partial decryption of the response, and additionally creates a ZK proof of correctness of this decryption:

$$d_{i,\mathcal{U}} \leftarrow \textbf{HE.PartialDec}(\mathsf{sk}_{i,\mathcal{U}}^{\mathsf{HE}}, \mathsf{ct}_{y_i}),$$
$$\pi_{y_i} \leftarrow \mathsf{ZKPoK}\{\mathsf{sk}_{i,\mathcal{U}}^{\mathsf{HE}} \mid d_{i,\mathcal{U}} = \textbf{HE.PartialDec}(\mathsf{sk}_{i,\mathcal{U}}^{\mathsf{HE}}, \mathsf{ct}_{y_i})\}.$$

In our RLWE-based implementation, the proof π_{y_i} is to prove in ZK the knowledge of secret key $\mathsf{sk}_{i,\mathcal{U}}^{\mathsf{HE}}$ and a flooding noise term $e'_{i,\mathcal{U}}$ such that

- $d_{i,\mathcal{U}} = \mathsf{ct}_{y_i,1} \cdot \mathsf{sk}_{i,\mathcal{U}}^{\mathsf{HE}} + e'_{i,\mathcal{U}}$ over the ring defined by HE; and
- $\mathsf{sk}_{i,\mathcal{U}}^{\mathsf{HE}}$ and $e'_{i,\mathcal{U}}$ have bounded norms according to their distributions.

Note that the ZK proofs $\pi_{\mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}}}, \pi_{\mathsf{sk}_{i,\mathcal{U}}^{\mathcal{F}}}, \pi_{x_i}, \pi_{y_i}$ together ensure that the final decryption noise has a bounded norm according to noise estimation using the HE parameters, preventing an adversary from exploiting the noises in encryption and partial decryption to tamper the underlying plaintext value y_i.

Finally, the user updates the counter $\mathsf{cnt}_i = \mathsf{cnt}_i + 1$ and sends $(\mathsf{ct}_{x_i}, \mathsf{ct}_{y_i}, d_{i,\mathcal{U}}, \pi_{x_i}, \pi_{y_i})$ to the server for verification.

Verification of Computation Phase. The server first verifies the proof π_{x_i} on ct_{x_i} and aborts in case of an invalid proof. Next, the server re-computes the ciphertext:

$$\mathsf{ct}'_{y_i} = \textbf{HE.Eval}(\mathsf{pk}_i^{\mathsf{HE}}, \mathsf{ct}_{\mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}}}, \mathsf{ct}_{\mathsf{sk}_{i,\mathcal{U}}^{\mathcal{F}}}, \mathsf{ct}_{x_i}, f_{\mathsf{cnt}_i}).$$

Since **HE.Eval** is deterministic, we must have $\mathsf{ct}_{y_i} = \mathsf{ct}'_{y_i}$ for an honest user. Additionally, the proof of partial decryption π_{y_i} must be valid. If both conditions are satisfied, the server proceeds to decrypt ct_{y_i}. To do this, $\mathcal{S}$ generates its decryption share $d_{i,\mathcal{S}}$ and combines $(d_{i,\mathcal{U}}, d_{i,\mathcal{S}})$ for full decryption as:

$$d_{i,\mathcal{S}} \leftarrow \textbf{HE.PartialDec}(\mathsf{sk}_{i,\mathcal{S}}^{\mathsf{HE}}, \mathsf{ct}_{y_i}),$$
$$y_i \leftarrow \textbf{HE.Combine}(\mathsf{pk}^{\mathsf{ct}_{y_i}}, d_{i,\mathcal{U}}, d_{i,\mathcal{S}}).$$

Vɛrity efficiently verifies HE computation by leveraging the deterministic nature of the HE ciphertexts.

[5] This is not required for *Vɛrity-Auth* since the key includes a fresh token τ_i for each message. Also, a public nonce, such as the timestamp, can be used in place of a counter.

6 Drop Out Attacks

Here, we analyze the impact of drop out attacks on the two solutions presented so far. For *Vɛrity-Auth*, although the input itself cannot be tampered with, a malicious user could selectively drop out *based on the sampled noise*. This attack is possible even when the noise is sampled honestly, as users can observe the noise in the clear before reporting. For instance, consider a malicious user $\mathcal{U}_i$ with true input $x_i^{\mathcal{A}} = 1$. If $\mathcal{U}_i$ wants to report a 0, they could drop out if the sampled noise is $b_{i,\rho} = 1$ (i.e., the input bit would be flipped).

This attack can be prevented by combining both of our proposed protocols to ensure both authorized input as well as honestly generated, *encrypted* noise. In particular, we start with *Vɛrity* and introduce the following changes. First, the protocol is augmented with an additional phase of input authorization which is identical to that of *Vɛrity-Auth*. Second, the key of the Legendre PRF is modified to match that of *Vɛrity-Auth*. In particular, we no longer require $\mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}}$ and cnt_i; instead we have $\mathsf{K}_i = \mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}} + \tau_i$ as in *Vɛrity-Auth*. Finally, in the verification phase, each user provides additional information–the signature and message pair $(\sigma_i', \overrightarrow{M_i'})$, along with an additional ZKP: $\pi_{x_i} \leftarrow \mathsf{ZKPoK}\Big\{z \in \{0,1\} \mid z = \mathsf{Open}(\overrightarrow{M_i'}[1])$ and $\mathsf{ct}_{x_i} = \mathbf{HE.Enc}(\mathsf{pk}_i^{\mathsf{HE}}, z)\Big\}$. The goal of this information is to prove that RR was evaluated under HE on the authorized input. Thus, a malicious user can drop out based *only* on its input (which cannot be tampered with) and public parameters (ε determines the probability of flipping), providing the strongest guarantee that can be achieved in this context. We call this protocol as *Vɛrity-Auth*$^+$(Fig. 7 in the full paper [8]). For *Vɛrity*, the sampled noise is encrypted, preventing the adversary from exploiting it. Since a malicious user can change their input arbitrarily, they gain no advantage by dropping out.

7 Security Analysis

We analyze security in the real-ideal world security model [44], assuming a malicious adversary. Fig. 4 and Fig. 5 show the ideal functionalities for *Vɛrity-Auth* and *Vɛrity*, respectively. The functionality in Fig. 5 evaluates RR honestly on the inputs that are *chosen* by the users, and outputs all the responses. In contrast, in Fig. 4, the input is authorized but the adversary has the ability to drop out based on the randomness.

We analyze *Vɛrity-Auth* in two cases: when the authorizer is honest or when it is controlled by a malicious adversary.

***Vɛrity-Auth* – Honest Authorizer.** In this setting, ZKP ensures that malicious users can only produce valid proofs for outputs generated honestly using RR on the authorized input. Since the PRF key is jointly generated by the user and the authorizer, the user cannot control or predict the randomness. Their only option is to drop out if dissatisfied with the output. This is formalized as:

Theorem 4. *Model $\mathsf{H}_{\mathbb{G}_1^*}$ as a random oracle. Assume $\mathcal{F}$ is a PRF, Pedersen scheme is a commitment and our proofs are ZKPs. Let a malicious adversary control some parties but not the authorizer. Then Vɛrity-Auth achieves the ideal functionality of Fig. 4.*

***Vɛrity-Auth* – Corrupted Authorizer.** We do not prevent an authorizer colluding with a user from providing them with multiple signed bits – the whole premise of *Vɛrity-Auth* is to leverage an authorizer to prevent malicious user behavior. In such cases, the adversary cannot access honest users' payloads or alter their submissions, but can only drop them. However, even if the authorizer is malicious, it cannot compromise the privacy of honest users. This holds true even if both the authorizer and server are colluding, as stated in as Thm 5. The private information we want to protect is the join of $\mathsf{payload}_i$ and x_i since the authorizer knows all x_i. Intuitively what we want to prove is that the input authorization that our protocol relies on, does not enable the authorizer to fingerprint the user contribution in a way that enables the server to link it to a payload. While the authorizer and server might have side channels, we argue that our construction introduces no *additional* linking. To model this formally, we assume that the responses are submitted to the server in a shuffled order.

Theorem 5. *Assume the ZKPs are Perfect Special Honest-Verifier Zero-Knowledge i.e., they can be faked by a simulator, $\mathsf{H}_{\mathbb{G}_1^*}$ is a random oracle, $\mathcal{F}$ is a PRF and EQS is a blind signature scheme. Suppose a malicious adversary controls the authorizer, the server and a subset of users. Then Vɛrity-Auth achieves the ideal functionality of Fig. 4.*
Furthermore, if the collection of messages to the server in each round are shuffled (by an honest third party or other equivalent means) before being given to the server. Then the protocol implements that functionality with the servers outputs being shuffled.

The blind signatures ensure unlinkability between signing requests and signatures, preventing any additional communication between the authorizer and the server

Vɛrity. *Vɛrity* prevents randomness-dependent tampering with the input for two reasons. First, the randomness is encrypted. Second, the adversary cannot attack homomorphically, as the output ciphertext results from a homomorphic evaluation of the expected fixed functionality.

Theorem 6. *Assume our proofs are ZKPs, $\mathcal{F}$ is a PRF and the HE scheme is semantically secure. Vɛrity achieves the ideal functionality of Fig. 5 with malicious security.*

8 Robustness Analysis

With Authorized Input. We start by analyzing *Vɛrity-Auth* in the case where there are no drop-outs, i.e., all n users are

Parameters. ε - Privacy parameter
User's Input. Each user $\mathcal{U}_i$ inputs a bit x_i and chooses $\mathsf{payload}_i$
Authorizer's Input. Authorizer inputs the ground truth $x_i^{\mathcal{A}}$ for every user $\mathcal{U}_i$
Functionality.

- Set $k \leftarrow \lfloor \log_2(1+e^{\varepsilon}) \rfloor$ and $\rho = \frac{1}{2^k}$
- Let $\mathcal{I}$ be the set of users whose input x_i matches the $x_i^{\mathcal{A}}$ from the authorizer $\mathcal{A}$
- For every user $\mathcal{U}_i \in \mathcal{I}$
  - Generate a sample of $b_{i,\rho} \sim Ber(\rho)$
  - Set $y_i = (1-x_i^{\mathcal{A}})b_{i,\rho} + (1-b_{i,\rho})x_i^{\mathcal{A}}$
  - If $\mathcal{U}_i$ is corrupt, send them y_i and give them an opportunity to drop out
- Send $(y_i, \mathsf{payload}_i)_{i \text{ s.t. } \mathcal{U}_i \text{ hasn't dropped}}$ to the server $\mathcal{S}$

Figure 4: Ideal Functionality for *Vɛrity-Auth*

Parameters. ε - Privacy parameter
User's Input. Each user $\mathcal{U}_i$ inputs a bit x_i.
Functionality.

- Set $k \leftarrow \lfloor \log_2(1+e^{\varepsilon}) \rfloor$ and $\rho = \frac{1}{2^k}$
- For every user $\mathcal{U}_i$
  - Generate a sample of $b_{i,\rho} \sim Ber(\rho)$
  - Set $y_i = (1-x_i)b_{i,\rho} + (1-b_{i,\rho})x_i$
  - Send y_i to the server $\mathcal{S}$

Figure 5: Ideal Functionality for *Vɛrity*

required to submit a valid response.

Lemma 7 (*Vɛrity-Auth* Without drop-out)**.** *$\hat{s}$ has ℓ_1 error $\Theta(\frac{\sqrt{n}}{\varepsilon})$, regardless of the number of malicious users.*

Since *Vɛrity-Auth* can prevent both input tampering and randomizer tampering, poisoning attacks are completely prevented in the absence of drop outs. The only error term, $\frac{\sqrt{n}}{\varepsilon}$, is the inherent error introduced by the randomness required to satisfy *LDP* [5, 18, 23].

Next, we analyze the case when the users can drop out.

Lemma 8 (*Vɛrity-Auth*)**.** *With m malicious users, an adversary can skew $\hat{s}$ by an ℓ_1 error $\tilde{O}(w(1+\frac{1}{\varepsilon}) + \frac{\sqrt{n}}{\varepsilon})$, for $w \in [m]$, w.p. $\sum_{j=\max\{0,w+q-m\}}^{\min\{q,w\}} \left(\binom{q}{j}\binom{m-q}{w-j}(1-\rho)\right)^{m+2j-q-w} \rho^{q+w-2j}$ if dropping out based on the randomness, where $q = \sum_{j=1}^{m} x_i$. In particular, the malicious users can introduce the worst case error of $\tilde{O}\left(m(1+\frac{1}{\varepsilon}) + \frac{\sqrt{n}}{\varepsilon}\right)$ w.p. at most $\left(\frac{e^{\varepsilon}}{1+e^{\varepsilon}}\right)^{q}\left(\frac{1}{1+e^{\varepsilon}}\right)^{m-q}$.*

In *Vɛrity-Auth*, malicious users can observe the sampled noise and can drop out based on that. Specifically, let $w \in \{0, m\}$ be the sum of the noisy responses of the malicious users. An advsersary can then introduce error proportional to w. Recall that the strongest poisoning attack is when all the malicious users directly change their response (y_i) to be 1 (or 0 equivalently), which can introduce an error of $\tilde{O}(m(1+\frac{1}{\varepsilon}) + \frac{\sqrt{n}}{\varepsilon})$ (Lemma 2). In our context, the adversary can still induce this worst case error when $w = m$, albeit with a significantly lower probability. Specifically, without any verifiability, malicious users can *always* lie about all m of their responses, and thereby carry out the worst-case attack with arbitrarily high probability. In contrast, in *Vɛrity-Auth*, adversary can execute this attack only with probability bounded by that of obtaining $w = m$ by running RR on the authorized inputs. Thus, the effectiveness of the attack is now determined by both the randomness of RR and the distribution of the authorized inputs of the malicious users.

Lemma 9 (*Vɛrity-Auth*$^+$)**.** *With m malicious users, an adversary can skew $\hat{s}$ by an ℓ_1 error of $\tilde{O}(q + \frac{\sqrt{n-m}}{\varepsilon})$ with arbitrarily high probability, where $q = \sum_{i=1}^{m} x_i$.*

By encrypting the noise, the adversary is restricted to drop out only based on its input (which cannot be tampered with) in *Vɛrity-Auth*$^+$. The adversary's influence is then bounded by the sum of the authorized inputs of the malicious users, q.

Without Authorized Input. The robustness of *Vɛrity* is:

Lemma 10 (*Vɛrity*)**.** *With m malicious users, an adversary can skew $\hat{s}$ by an ℓ_1 error of $\tilde{O}(m + \frac{\sqrt{n}}{\varepsilon})$ with arbitrarily high probability.*

In the absence of authorized inputs, the best one can do is limit the adversary to input poisoning attacks. This is precisely what *Vɛrity* achieves – Observe that the resulting error aligns with the one from input poisoning attacks as in Lemma 3.

Proofs of all the theorems are in the full paper [8].

9 Implementation and Evaluation

We benchmarked both our protocols on an Intel Ice Lake CPU running at 2.6GHz with AVX512 instructions. We computed the Legendre PRF as $\mathcal{F}(x, \mathsf{K}) = 2^{-1} \cdot \left(((\mathsf{K}+x)^{(p-1)/2} \mod p) + 1\right) \mod p$.

9.1 *Vɛrity-Auth*

Implementation. We used an 120-bit Legendre PRF modulus $p = (2^{16} + 2^{18} + 2^{19} + 2^{119}) \cdot 2 + 1$. As shown by [33, 34], it takes $2^{89.7}$ time to recover the secret key when given 2^{20} Legendre PRF values. So, the user should rerun the setup phase and refresh its PRF key every $2^{20}/k$ sessions, where k is the number of PRF bits per protocol invocation. We instantiated the EQS signature using the Tate Pairing over curveFp254BNb [30], and we used Bulletproofs [16, 45] to implement the ZK proofs for correct PRF and RR computation. Bulletproofs operates directly on committed inputs without a setup process. As a result, we no longer need a separate proof (π_{x_i}; Eq. 10) showing the consistency between the bit used for evaluating RR and the authroized bit, and instead, the server verifies π_{y_i} (Eq. 9) directly using the commitment $\overrightarrow{M_i'}[1]$. Additionally, we moved the ZKPoK of $\mathsf{com}_{\mathsf{sk}_i}$ to the setup phase instead of sending it as part of $\pi_{b_{ij}}$ for every message.

Results. We report the mean of 100 runs in Table 2. *Vɛrity-Auth* is independent of the number of PRF bits, k, up to the noise sampling phase. Our results show that *Vɛrity-Auth* is

Phase	Party	Description	Running Time / Communication Cost				
			$k=2$ ($\varepsilon=1.1$)	$k=3$ ($\varepsilon=2$)	$k=4$ ($\varepsilon=2.8$)	$k=5$ ($\varepsilon=3.5$)	$k=6$ ($\varepsilon=4.2$)
Setup	Authorizer	Generate parameters	0.48ms / 0.86KB				
	User	Generate ZKPoK of sk_i	0.35ms / 0.38KB				
	Server	Verify ZKPoK of sk_i	0.60ms / -				
Input Authorization	User	Generate sign request	1.62ms / 1.12KB				
	Authorizer	Verify sign request & issue blind signature	2.05ms / 0.32KB				
	User	Unblind the signature	3.05ms / -				
Noise Sampling	User	Compute PRFs and RR bit	200 μs / -	300 μs / -	400 μs / -	500 μs / -	600 μs / -
		Generate ZK proof of PRF and RR computation	5.22ms 1.02KB	8.37ms 1.19KB	9.15ms 1.30KB	9.92ms 1.40KB	15.33ms 1.56KB
Verification	Server	Verify ZK proofs	0.90ms	1.38ms	1.44ms	1.50ms	2.26ms
		Verify signature	2.16ms				

Table 2: Results for *Vɛrity-Auth* when the RR bit is computed using different numbers (k) of PRF evaluations. The protocol is independent of k until noise sampling. Communication cost is measured by outbound bandwidth, marked with a "-" if no outbound communication occurs.

Phase	Party	Description	Running Time / Communication Cost		
			$k=2(\varepsilon=1.1)$	$k=4(\varepsilon=2.8)$	$k=6(\varepsilon=4.2)$
Setup	Server & User	Generate HE keys	21.41 s / 9126.88 MB	28.31 s / 9301.44 MB	35.16 s / 9476 MB
	Server	Sample and encrypt PRF key share	0.89 s / 78.44 MB	0.91 s / 80.22 MB	0.93 s / 82 MB
		Generate ZKPoK of encryption	**1216.23 s / 3.17 KB**		
	User	Verify Server's ZKPoK	**1282.32 s / -**		
		Sample and encrypt PRF key share	0.89 s / 78.44 MB	0.91 s / 80.22 MB	0.93 s / 82 MB
		Generate ZKPoK of encryption	**1216.23 s / 3.17 KB**		
	Server	Verify User's ZKPoK	**1282.32 s / -**		
Noise Sampling	User	Encrypt input bits	0.89 s / 78.44 MB	0.91 s / 80.22 MB	0.93 s / 82 MB
		Generate ZKPoK of encryption	**1216.23 s / 3.17 KB**		
		Homomorphic evaluation of PRF and RR bit	69.48 s / 4.03 MB	73.45 s / 4.03 MB	78.17 s / 4.03 MB
		Partial decryption	**0.26 s / 2.02 MB**		
		Generate ZKPoK of partial decryption	**133.57 s / 1.76 KB**		
Verification	Server	Verify ZKPoK of encryption	**1282.32 s / -**		
		Re-compute homomorphic evaluation	69.48 s / -	73.45 s / -	78.17 s / -
		Verify ZKPoK of partial decryption	**117.0 s / -**		
		Partial decryption and recover result	**0.26 s / -**		

Table 3: Results for *Vɛrity* when the RR bit uses different numbers (k) of PRF evaluations. Steps with identical performance across k are highlighted in bold. $k=\{3,4\}$ and $k=\{5,6,7,8\}$ have identical costs because of the same complexity of the circuit for HE. Communication cost is measured by outbound bandwidth, marked with a "-" if no outbound communication occurs.

very efficient: the setup phase takes < 1 ms for each party; the user spends < 5 ms during input authorization, while the authorizer spends 2 ms. The costliest step is to generate the ZK proof of correct PRF evaluations: for $k \in [2,6]$, the prover's (user) cost is ≤ 15 ms, and the verifier's (server) cost is ≤ 4ms. The communication costs are also low – the respective costs for the authorizer, user and the server are 2.3 KB, ≤ 5.02 KB and ≤ 3.58 KB. Our performance improves with lower ε since the flipping probability is given by $\rho = 1/2^k$.

Baseline. The closest prior work to ours is a concurrent work by Bontekoe et al. [14] that also uses authorized inputs but in a weaker threat model (see Sec. 10). Reporting a single instance of verifiable RR takes each user 1.31s, which is 100× slower than *Vɛrity-Auth*. Complete details are in the full paper [8].

Scalability. Note that our protocols operate independently for each client. Consequently, a client's cost is completely *scale-free*–it remains the same regardless of the total number of participants n. For the server, the most expensive component of the protocol is the verification of the ZK proofs. However, efficient batched verification techniques exist [16] that enable substantial amortization when verifying proofs from a large number of clients. To further demonstrate the practicality of *Vɛrity-Auth*, we provide microbenchmark results on a Pixel 9 smartphone in the full paper [8].

9.2 *Vɛrity*

Implementation. Our $(2,2)$-threshold HE scheme is based on the BGV leveled HE scheme [15] and is implemented using OpenFHE [3]. Our ZK proofs are again implemented using Bulletproofs. In particular, partial decryption applies noise flooding to statistically hide the secret key $\mathsf{sk}_{i,P}^{\mathsf{HE}}$ from the output. We used a 40-bits Legendre PRF modulus $p = (2^{17}+2^{19}+2^{39})\cdot 2+1$ to be compatible with OpenFHE. Although p is much smaller than the one in *Vɛrity-Auth*, the key recovery attack in [34] still requires roughly 2^{28} word operations when given 2^{10} PRF values. As a result, the user must refresh its secret key more frequently. Since p is an NTT-friendly prime, multiple Legendre PRFs and the corresponding Bernoulli bits can be evaluated in the "SIMD" slots in parallel. We set the RLWE ring degree to 2^{17}, the

ciphertext modulus to 3104 bits, and the discrete Gaussian with parameter 3.2 for the RLWE secret and error distributions. Such parameters provide roughly 146 bits of classical security [1], and they support Bernoulli bits that are products of at most 8 PRF bits. To prove RLWE encryption and partial decryption in ZK, we express these algorithms using inner product constraints over the secret values as in [7]. Since the Bulletproofs implementation operates in a 252-bit group that is smaller than the HE ciphertext modulus and insufficient for expressing RLWE relations as is, we partition the RNS moduli into groups such that the product Q_j of moduli in each group j is small enough to prove RLWE relations modulo Q_j.

Results. Table 3 shows our results. For $k = 6$, the server and the user both take 2535s in the setup phase, mostly spent on generating and verifying the ZK proof of encryption. The user's online time is 1429.16s while that of the server is 1477.75s. Thanks to the SIMD feature, with the same cost we can generate up to 2^{13} Bernoulli bits with $k \leq 8$. For $k = 4$ and $k = 2$, $\mathcal{F}(x, \mathsf{K})$ has shallower circuits and requires smaller ciphertext moduli, resulting in less expensive HE setup and evaluation. On the other hand, the ZK proofs have the same cost as we use the same number of groups to partition the RNS moduli.

Baseline. A recent work [38] considers the alternative approach of directly proving the correct homomorphic evaluation via ZKP. However, even proving a *single* naive homomorphic multiplication followed by modulus switching using small RLWE parameters ($N = 2^{13}$ and $\log Q = 137$) requires 433s. In fact, such RLWE parameters are insufficient for our circuits. The inefficiency stems from the numerous non-arithmetic operations such as gadget decomposition and modulus switching, resulting in a massive and practically infeasible ZK statement. See the full paper [8] for more details.

Scalability. As before, *Vɛrity* ensures that a client's cost is completely *scale-free*. Moreover, since *Vɛrity* can support multiple messages per user, we can obtain substantial amortization by leveraging SIMD-style parallelism of HE, which allows a *single* user to batch many reports in a single ciphertext. Specifically, *Vɛrity* can generate 2^{13} Bernoulli bits within the reported online time. Similarly, on the server side, verification of proofs across multiple clients can be batched for significant efficiency gains.

10 Related Work

A common point of difference with *all* prior work is that *Vɛrity* is the first non-interactive protocol that offers the strongest protection where users lack authorized inputs, by keeping the randomness encrypted. Additionally, our work is the first to analyze drop outs for verifiable DP. Existing works on verifiable RR [2,35,37,49] are interactive in nature. Biswas et al. [10] proposed an interactive protocol for the Binomial mechanism. Moran et. al. proposed a human-centric solution using physical envelopes [40]. Furthermore, these works do not consider authorized inputs. A few prior works [14,41,42] have considered authorized inputs, but they either fail to provide unlinkability [42] or require interaction [41]. Closest to our proposal is a concurrent work by Bontekoe et al. [14]; however, our work differs from theirs in several ways. First, they assume a weaker threat model of non-colluding authorizer and server. Second, their proposed protocols are either interactive or can support only a fixed number of reports, at a higher cost. Finally, our solution outperforms theirs by 100× (Sec. 9). Additionally, some works address the problem in the central model [6,43,48], assuming a trusted, centralized server and focusing on additive noise mechanisms like Laplace. Other approaches for generating noise in distributed DP settings [17,24,36] also require interaction.

11 Ethical Considerations

Stakeholders. Following frameworks from ethical philosophy [39] and the Menlo Report's principles for ICT research [4], we present a structured ethical analysis of our work. We begin by identifying key stakeholders.

- **Data Owners.** The users who hold their private bits x_i and report the corresponding noisy bits under randomized response mechanism.
- **Data Subjects.** The individuals whose private bits constitute the local input.
- **Analytics Users.** This can be private entities, such as companies or the broader public, which may benefit from the verifiable and local DP mechanisms.

Impacts. The ethical and harm impact of our research concerns the data owners and the data subjects. The relevant ethical principles here are Respect for the Person and Beneficence.

We analyze the impact of our research on the above stakeholders according to the following guiding principles.

Respect for Persons. asserts that individuals should be treated as autonomous agents and that those with diminished autonomy are entitled to protection [4]. It manifests as the right to privacy and informed consent. *Vɛrity-Auth* and *Vɛrity* both support respect for persons by enhancing the autonomy of data owners, enabling them to report statistics without leaking individual data points. By leveraging local differential privacy out system enables much more control on their privacy in the hands of the user who enforces the privacy protection by applying the local DP processing without having to rely on any other entity.

Beneficence. Beneficence is concerned with the appropriate balance of probable harm and likelihood of enhanced welfare resulting from the research. Our paper provides techniques

that enable computation of aggregate insights across large population that can be beneficial to understand the needs across users while protecting the privacy of individual contributors. This constitutes a tool that balances privacy and utility goals.

Potential harms to the right of privacy for the data subjects and owners could stem from a misuse of our solution with inappropriate privacy parameters while still claiming privacy for the users to justify their participation.

Mitigations. Mitigations for potential misuses of our privacy preserving data collection solution could be done via establishing privacy policies that the users of such systems need to follow and through auditing mechanisms for deployed solutions. The need for such mechanisms is already present in order to regulate data collection in general and is not specific to our solution. Tools like ours create more flexible mechanisms that could be used to enforce a wider range of privacy regulations.

Decision. Our protocols *Vɛrity-Auth* and *Vɛrity* offer privacy preserving means to collect statistics of users. They are tools that can be used to protect user privacy while enabling the collection of population metrics that can help serve the needs of the users. This work is evaluated on synthesized data and has no ethical implications associated with user data. We made the decision to pursue and publish our research since we believe its benefit a new privacy enhancing technology outweigh the risks of potential misuses.

12 Open Science

The implementations of our protocols *Vɛrity-Auth* and *Vɛrity* are available[6] under the Apache 2.0 license. The experiments appearing in this work may be fully reproduced with available source code. Our source code are organized into two subdirectories.

- `verity-auth`. This subdirectory contains source code for *Vɛrity-Auth*. The main cryptographic primitives are the EQS-based blind signature, Legendre PRF, and ZK proof of correct PRF and RR bit computation.
  - Our implementation of blind signature scheme are located directly under `verity-auth`.
  - Our implementation of Legendre PRF and the Bulletproofs-based ZK proof of PRF and RR bit computation are located under `verity-auth/zkp`.
- `verity`. This subdirectory contains source code for *Vɛrity*. The main cryptographic primitives are BGV-based leveled homomorphic encryption scheme, and ZK proof of correct HE encryption and partial decryption.
  - Our OpenFHE-based homomorphic Legendre PRF implementations are located directly under `verity`.
  - Our Bulletproofs-based implementation of the ZK proof of HE encryption and partial decryption are located under `verity/zkp`. In particular we used the approximate range proof and linear proof optimizations in [20].

[6] https://doi.org/10.5281/zenodo.17959962

References

[1] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *J. Math. Cryptol.*, 9(3):169–203, 2015.

[2] Andris Ambainis, Markus Jakobsson, and Helger Lipmaa. Cryptographic randomized response techniques, 2003.

[3] Ahmad Al Badawi, Andreea Alexandru, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Carlo Pascoe, Yuriy Polyakov, Ian Quah, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. OpenFHE: Open-source fully homomorphic encryption library. Cryptology ePrint Archive, Paper 2022/915, 2022. https://eprint.iacr.org/2022/915.

[4] Michael Bailey, David Dittrich, Erin Kenneally, and Doug Maughan. The menlo report. page 71–75, March 2012.

[5] Amos Beimel, Kobbi Nissim, and Eran Omri. Distributed private data analysis: Simultaneously solving how and what. In *Proceedings of the 28th Annual Conference on Cryptology: Advances in Cryptology*, CRYPTO 2008, pages 451–468, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] Zoë Ruha Bell, Shafi Goldwasser, Michael P. Kim, and Jean-Luc Watson. Certifying private probabilistic mechanisms. Cryptology ePrint Archive, Paper 2024/938, 2024.

[7] James Bell-Clark, Adrià Gascón, Baiyu Li, Mariana Raykova, and Phillipp Schoppmann. Willow: Secure aggregation with one-shot clients. *IACR Cryptol. ePrint Arch.*, page 936, 2024.

[8] James Bell-Clark, Adrià Gascón, Baiyu Li, Mariana Raykova, and Amrita Roy Chowdhury. Vɛrity: Verifiable local differential privacy. Cryptology ePrint Archive, Paper 2025/851, 2025.

[9] Ward Beullens, Tim Beyne, Aleksei Udovenko, and Giuseppe Vitto. Cryptanalysis of the legendre PRF and generalizations. *IACR Trans. Symmetric Cryptol.*, 2020(1):313–330, 2020.

[10] Ari Biswas and Graham Cormode. Interactive proofs for differentially private counting. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23. ACM, November 2023.

[11] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnes, and Bernhard Seefeld. Prochlo: Strong privacy for analytics in the crowd. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 441–459, New York, NY, USA, 2017. Association for Computing Machinery.

[12] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[13] Dan Boneh and David Mandell Freeman. Homomorphic signatures for polynomial functions. In Kenneth G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, pages 149–168, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[14] Tariq Bontekoe, Hassan Jameel Asghar, and Fatih Turkmen. Efficient verifiable differential privacy with input authenticity in the local and shuffle model, 2024.

[15] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *ITCS 2012*, pages 309–325, Cambridge, MA, USA, January 8–10, 2012. ACM.

[16] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy*, pages 315–334, San Francisco, CA, USA, May 21–23, 2018. IEEE Computer Society Press.

[17] Jeffrey Champion, abhi shelat, and Jonathan Ullman. Securely sampling biased coins with applications to differential privacy. Cryptology ePrint Archive, Paper 2019/823, 2019.

[18] T-H. Hubert Chan, Elaine Shi, and Dawn Song. Optimal lower bound for differentially private multi-party aggregation. In *Proceedings of the 20th Annual European Conference on Algorithms*, ESA'12, pages 277–288, Berlin, Heidelberg, 2012. Springer-Verlag.

[19] Albert Cheu, Adam Smith, and Jonathan Ullman. Manipulation attacks in local differential privacy. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 883–900. IEEE, 2021.

[20] C++ implementation of practical non-interactive publicly verifiable secret sharing with thousands of parties. https://github.com/shaih/cpp-lwevss, 2021.

[21] Ivan Bjerre Damgård. On the randomness of legendre and jacobi sequences. In Shafi Goldwasser, editor, *Advances in Cryptology — CRYPTO' 88*, pages 163–172, New York, NY, 1990. Springer New York.

[22] Bolin Ding, Janardhan Kulkarni, and Sergey Yekhanin. Collecting telemetry data privately. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 3574–3583, Red Hook, NY, USA, 2017. Curran Associates Inc.

[23] J. C. Duchi, M. I. Jordan, and M. J. Wainwright. Local privacy and statistical minimax rates. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 429–438, Oct 2013.

[24] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. Our data, ourselves: privacy via distributed noise generation. In *Proceedings of the 24th Annual International Conference on The Theory and Applications of Cryptographic Techniques*, EUROCRYPT'06, page 486–503, Berlin, Heidelberg, 2006. Springer-Verlag.

[25] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 9:211–407, August 2014.

[26] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 1054–1067, New York, NY, USA, 2014. Association for Computing Machinery.

[27] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, page 75–88, New York, NY, USA, 2014. Association for Computing Machinery.

[28] Dario Fiore, Rosario Gennaro, and Valerio Pastro. Efficiently verifiable computation on encrypted data. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 844–855, New York, NY, USA, 2014. Association for Computing Machinery.

[29] Georg Fuchsbauer, Christian Hanser, and Daniel Slamanig. Structure-preserving signatures on equivalence classes and constant-size anonymous credentials. *Journal of Cryptology*, 32(2):498–546, April 2019.

[30] C. C. F. Pereira Geovandro, Marcos A. Simplício Jr., Michael Naehrig, and Paulo S. L. M. Barreto. A family of implementation-friendly BN elliptic curves. *J. Syst. Softw.*, 84(8):1319–1326, 2011.

[31] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions (extended abstract). In *25th FOCS*, pages 464–479, Singer Island, Florida, October 24–26, 1984. IEEE Computer Society Press.

[32] Lorenzo Grassi, Christian Rechberger, Dragos Rotaru, Peter Scholl, and Nigel P. Smart. MPC-friendly symmetric key primitives. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 430–443, Vienna, Austria, October 24–28, 2016. ACM Press.

[33] Novak Kaluđerović, Thorsten Kleinjung, and Dušan Kostić. Cryptanalysis of the generalised legendre pseudorandom function. *ANTS XIV*, 4(1):267–282, 2020.

[34] Novak Kaluđerović, Thorsten Kleinjung, and Dušan Kostić. Improved key recovery on the legendre PRF. *IACR Cryptol. ePrint Arch.*, page 98, 2020.

[35] Fumiyuki Kato, Yang Cao, and Masatoshi Yoshikawa. Preventing manipulation attack in local differential privacy using verifiable randomization mechanism. *CoRR*, abs/2104.06569, 2021.

[36] Hannah Keller, Helen Möllering, Thomas Schneider, Oleksandr Tkachenko, and Liang Zhao. Secure noise sampling for DP in MPC with finite precision. Cryptology ePrint Archive, Paper 2023/1594, 2023.

[37] H. Kikuchi, J. Akiyama, H. Gobioff, and G. Nakamura. Stochastic voting protocol to protect voters privacy. In *Proceedings 1999 IEEE Workshop on Internet Applications (Cat. No.PR00197)*, pages 103–111, 1999.

[38] Christian Knabenhans, Alexander Viand, Antonio Merino-Gallardo, and ppp Anwar Hithnawi. vfhe: Verifiable fully homomorphic encryption. In *WAHC@CCS*, pages 11–22. ACM, 2024.

[39] Tadayoshi Kohno, Yasemin Acar, and Wulf Loh. Ethical frameworks and computer security trolley problems: Foundations for conversations. pages 5145–5162, August 2023.

[40] Tal Moran and Moni Naor. Polling with physical envelopes: A rigorous analysis of a human-centric protocol. In *Proceedings of the 24th Annual International Conference on The Theory and Applications of Cryptographic Techniques*, EUROCRYPT'06, page 88–108, Berlin, Heidelberg, 2006. Springer-Verlag.

[41] Danielle Movsowitz-Davidow, Yacov Manevich, and Eran Toch. Privacy-preserving transactions with verifiable local differential privacy. In *Conference on Advances in Financial Technologies*, 2023.

[42] Gonzalo Munilla Garrido, Johannes Sedlmeir, and Matthias Babel. Towards verifiable differentially-private polling. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*, ARES '22, New York, NY, USA, 2022. Association for Computing Machinery.

[43] Arjun Narayan, Ariel Feldman, Antonis Papadimitriou, and Andreas Haeberlen. Verifiable differential privacy. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[44] Goldreich Oded. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[45] Rust bulletproofs library. https://github.com/dalek-cryptography/bulletproofs, 2023. Dalek cryptography Project.

[46] Google Privacy Sandbox. Event-level report, attribution reporting api, 2023.

[47] István András Seres, Máté Horváth, and Péter Burcsi. The legendre pseudorandom function as a multivariate quadratic cryptosystem: security and applications. *Applicable Algebra in Engineering, Communication and Computing*, pages 1–31, 2023.

[48] Ali Shahin Shamsabadi, Gefei Tan, Tudor Ioan Cebere, Aurélien Bellet, Hamed Haddadi, Nicolas Papernot, Xiao Wang, and Adrian Weller. Confidential-DPproof: Confidential proof of differentially private training. In *The Twelfth International Conference on Learning Representations*, 2024.

[49] Shaorui Song, Lei Xu, and Liehuang Zhu. Efficient defenses against output poisoning attacks on local differential privacy. *IEEE Transactions on Information Forensics and Security*, 18:5506–5521, 2023.

[50] Kunal Talwar, Shan Wang, Audra McMillan, Vitaly Feldman, Pansy Bansal, Bailey Basile, Aine Cahill, Yi Sheng Chan, Mike Chatzidakis, Junye Chen, Oliver R. A. 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, Congzheng Song, Linmao Song, Karl Tarbe, Sebastian Vogt, Shundong Zhou, Vojta Jina, Michael Scaria, and Luke Winstrom. Samplable anonymous aggregation for private federated data analysis. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 2859–2873, New York, NY, USA, 2024. Association for Computing Machinery.

[51] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 137–152, New York, NY, USA, 2015. Association for Computing Machinery.

[52] Stanley L. Warner. Randomized response: A survey technique for eliminating evasive answer bias. *Journal of the American Statistical Association*, 60(309):63–69, 1965.

A Probability of flipping in RR_ρ

We begin by recalling the RR_ρ mechanism. To satisfy ε-LDP, an input bit x must be flipped with probability $\rho = \frac{1}{1+e^\varepsilon}$ [25]. However, as explained in Sec. 4.2, for computational efficiency we restrict our protocols to probabilities of the form $\rho = \frac{1}{2^k}$. In particular, given a target privacy parameter ε, we choose the largest k such that $\frac{1}{2^k} \le \frac{1}{1+e^\varepsilon}$, i.e., $k = \lfloor \log(1+e^\varepsilon) \rfloor$ and then set $\rho = 1/2^k$.

Since an ε-LDP mechanism also satisfies ε'-LDP for all $\varepsilon' \ge \varepsilon$, our choice of ρ is conservative: the privacy parameter we report corresponds to the *effective* value ε' implied by the chosen ρ. For $\rho = 1/2^k$, this effective privacy parameter is $\varepsilon' = \ln(2^k - 1)$. For example, if the desired privacy parameter is $\varepsilon = 5$, then we set $k = \lfloor (1+e^5) \rfloor = 7$. This results in an effective privacy parameter of $\varepsilon' = \ln(127) \approx 4.84$. In our evaluation (Tables 2 and 3), we report the exact value $\varepsilon' = \ln(2^k - 1)$ induced by the chosen k.

B Input Swapping Attack

We illustrate the attack with an example. Consider two malicious users, $\mathcal{U}_1$ and $\mathcal{U}_2$, with inputs $x_1 = 0$ and $x_2 = 1$, who want to report 1s. Suppose the honestly generated noise samples for them are $b_{\rho,1} = 1$ and $b_{\rho,2} = 0$, which forces both to report 0s. However, if they swap their inputs, they can both report 1s. Specifically, they can do this by swapping their message and signature pairs when submitting to the server, which is possible due to the unlinkability of the signatures. Note that this attack is randomness-dependent, which should be disallowed by our protocol. By binding the secret key to the message, we prevent such an attack, as the user's identity is tied to the input they report *before* observing the randomness.

C Extension to r-ary RR

Here, we detail how to extend our protocols to support r-ary RR.When the input takes values in a set of size r, ε-DP RR consists of outputting the true input with probability at most $p_{r,\varepsilon} = (e^\varepsilon - 1)/(e^\varepsilon - 1 + r)$ and otherwise outputting a uniformly random value from the set. In our setting, we can choose k (the number of PRF bits used to sample the biased bit) so that our biased bit $b_{\rho,i} \le p_{r,\varepsilon}$ as before (note the biased bit is now the probability of not randomizing the output rather than of flipping).

Power-of-Two Domain. If $r = 2^m$ for some integer m then we can just compute $b_{ij} = \mathcal{F}(\mathsf{K}_i, j)$ for $j \in [k+1, k+m]$ and use those as the bits of the binary representation of the random output. Formally, we encode the input in binary as $x_{i,1} \ldots x_{i,m}$ and commit to it in this form at the start of the protocol. For $l \in [m]$ we set $y_{i,l} = b_{\rho,i} x_{i,l} + (1 - b_{\rho,i}) b_{i,k+l}$ and include zero-knowledge proofs to attest to the value of all of the $y_{i,l}$.

Non Power-of-Two Domain. Choose m such that $2^m \ge r$, and let a be the largest integer satisfying $2^m \ge ar$. Set $q = (e^\varepsilon - 1)/(e^\varepsilon - 1 + 2^m/a)$ and choose the biased bit $b_{\rho,i}$ such that $b_{\rho,i} \le q$. We encode an input $x_i \in [r]$ by first mapping it to a uniformly random value in $[(x_i - 1)a, x_i a)$ and then encoding this value in binary as $x_{i,1} \ldots x_{i,m}$ which are committed to at the start of the protocol. As in the power of two case, for $l \in [m]$, we set $y_{i,l} = b_{\rho,i} x_{i,l} + (1 - b_{\rho,i}) b_{i,k+l}$ and prove this in zero-knowledge.

Lemma 11. *The above mechanism implements ε-LDP r-ary randomized response, except with probability at most $(1 - ar2^{-m})$, in which case the output is independent of the input.*

Proof. We first analyze the case where the server receives an encoding corresponding to a value $v \ge ar$. The server can conclude that $b_{\rho,i} = 0$, and hence that the output is independent of the true input x_i. In other words, this is equivalent to providing no response and this with probability $(1-q)(1-ar2^{-m}) < (1-ar2^{-m})$. On the other hand, if the server receives an encoding of a value $< ar$, it corresponds to some $x' \in [r]$. Crucially, which of the a encodings associated with x' is observed carries no information, since all such encodings are chosen uniformly regardless of the value of $b_{\rho,i}$. To establish equivalence with r-ary RR, it therefore suffices to verify that the probability of observing x' given x depends only on whether $x = x'$, and that this probability is at most an e^ε factor larger when $x = x'$ than when $x \ne x'$. The former follows because q is independent of x and the output is uniform when $b_{\rho,i} = 0$, while the latter is ensured by our choice of q. □

Parameters. ε - Privacy parameter; κ - Security parameter

- **Setup Phase.**
  Server and User:
  - The server $\mathcal{S}$ and user $\mathcal{U}_i$ jointly generates the keys of the threshold HE scheme: $(\mathsf{pk}_i^{\mathsf{HE}}, \mathsf{sk}_{i,\mathcal{U}}^{\mathsf{HE}}, \mathsf{sk}_{i,\mathcal{S}}^{\mathsf{HE}}) \leftarrow \textbf{HE.GenParam}(1^\kappa)$
    where $\mathsf{sk}_{i,\mathcal{S}}^{\mathsf{HE}}$ and $\mathsf{sk}_{i,\mathcal{U}}^{\mathsf{HE}}$ is known only to the server and the user $\mathcal{U}_i$
  - Initialize $\mathsf{cnt}_i = 0$

  Server:
  - Samples its share of the secret key $\mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}}$ for user $\mathcal{U}_i$'s PRF as $\mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}} \leftarrow\!\$\, \mathbb{F}_p$
  - Encrypts the sampled key and generates a ZK proof of the correctness of the encryption

$$\mathsf{ct}_{\mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}}} \leftarrow \textbf{HE.Enc}(\mathsf{pk}_i^{\mathsf{HE}}, \mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}}), \pi_{\mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}}} \leftarrow \mathsf{ZKPoK}\{\mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}} \in \mathbb{F}_p \mid \mathsf{ct}_{\mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}}} = \textbf{HE.Enc}(\mathsf{pk}_i^{\mathsf{HE}}, \mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}})\}$$

  - Sends the generated ciphertext and ZK proof to user: $\mathcal{U}_i$, $\mathcal{S} \xrightarrow{\mathsf{ct}_{\mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}}}, \pi_{\mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}}}} \mathcal{U}_i$

  User: Each user $\mathcal{U}_i$:
  - Verifies the proof $\pi_{\mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}}}$ and aborts in case of the check fails
  - Samples its share of the secret key for the PRF $\mathsf{sk}_{i,\mathcal{U}}^{\mathcal{F}} \leftarrow\!\$\, \mathbb{F}_p$
  - Encrypts the sampled key and generates a ZK proof of the correctness of the encryption

$$\mathsf{ct}_{\mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}}} \leftarrow \textbf{HE.Enc}(\mathsf{pk}_i^{\mathsf{HE}}, \mathsf{sk}_{i,\mathcal{U}}^{\mathcal{F}}), \pi_{\mathsf{sk}_{i,\mathcal{U}}^{\mathcal{F}}} = \mathsf{ZKPoK}\{\mathsf{sk}_{i,\mathcal{U}}^{\mathcal{F}} \in \mathbb{F}_p | \mathsf{ct}_{\mathsf{sk}_{i,\mathcal{U}}^{\mathcal{F}}} = \textbf{HE.Enc}(\mathsf{pk}_i^{\mathsf{HE}}, \mathsf{sk}_{i,\mathcal{U}}^{\mathcal{F}})\}$$

  - Sends the generated ciphertext and ZK proof to the server $\mathcal{U}_i \xrightarrow{\mathsf{ct}_{\mathsf{sk}_{i,\mathcal{U}}^{\mathcal{F}}}, \pi_{\mathsf{sk}_{i,\mathcal{U}}^{\mathcal{F}}}} \mathcal{S}$

  Server: For each user $\mathcal{U}_i$:
  - Continues if $\pi_{\mathsf{sk}_{i,\mathcal{U}}^{\mathcal{F}}}$ verifies, reject otherwise.
- **Noise Sampling Phase.**
  User: Each user $\mathcal{U}_i$
  - Encrypts the input bit x_i and generate a ZK proof of the correctness of the encryption

$$\mathsf{ct}_{x_i} \leftarrow \textbf{HE.Enc}(\mathsf{pk}_i^{\mathsf{HE}}, x_i); \pi_{x_i} \leftarrow \mathsf{ZKPoK}\{x_i \in \{0,1\} \mid \mathsf{ct}_{x_i} = \textbf{HE.Enc}(\mathsf{pk}_i^{\mathsf{HE}}, x_i)\}$$

  - Sets $k \leftarrow \lfloor \log_2(1+e^\varepsilon) \rfloor$ and let $f_{\mathsf{cnt}_i}(\mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}}, \mathsf{sk}_{i,\mathcal{U}}^{\mathcal{F}}, x_i) = x_i + (1-2x_i)\prod_{j=1}^{k} \mathcal{F}(\mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}} + \mathsf{sk}_{i,\mathcal{U}}^{\mathcal{F}} + \mathsf{cnt}_i, j)$
  - Homomorphically evaluates f_{cnt_i} to generate the ciphertext of the response: $\mathsf{ct}_{y_i} \leftarrow \textbf{HE.Eval}(\mathsf{pk}_i^{\mathsf{HE}}, \mathsf{ct}_{\mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}}}, \mathsf{ct}_{\mathsf{sk}_{i,\mathcal{U}}^{\mathcal{F}}}, \mathsf{ct}_{x_i}, f_{\mathsf{cnt}_i})$
  - Partially decrypts ct_{y_i} using the secret key $\mathsf{sk}_{i,\mathcal{U}}^{\mathsf{HE}}$ and generate a ZK proof showing the correctness of this step

$$d_{i,\mathcal{U}} \leftarrow \textbf{HE.PartialDec}(\mathsf{sk}_{i,\mathcal{U}}^{\mathsf{HE}}, \mathsf{ct}_{y_i}), \pi_{y_i} \leftarrow \mathsf{ZKPoK}\{\mathsf{sk}_{i,\mathcal{U}}^{\mathsf{HE}} \mid d_{i,\mathcal{U}} = \textbf{HE.PartialDec}(\mathsf{sk}_{i,\mathcal{U}}^{\mathsf{HE}}, \mathsf{ct}_{y_i})\}$$

  - Updates $\mathsf{cnt}_i = \mathsf{cnt}_i + 1$
  - Sends all the ciphertexts, the partial decryption of the response and the ZK proofs to the server: $\mathcal{U}_i \xrightarrow{\mathsf{ct}_{y_i}, \mathsf{ct}_{x_i}, d_{i,\mathcal{U}}, \pi_{x_i}, \pi_{y_i}} \mathcal{S}$
- **Verification of Computation Phase.**
  Server: For each user $\mathcal{U}_i$, the server
  - Verifies the proof π_{x_i} on ct_{x_i} and reject in case of an invalid proof.
  - Re-computes the ciphertext for the response: $\mathsf{ct}'_{y_i} = \textbf{HE.Eval}(\mathsf{pk}_i^{\mathsf{HE}}, \mathsf{ct}_{\mathsf{sk}_{i,\mathcal{S}}^{\mathcal{F}}}, \mathsf{ct}_{\mathsf{sk}_{i,\mathcal{U}}^{\mathcal{F}}}, \mathsf{ct}_{x_i}, f_{\mathsf{cnt}_i})$.
  - Continue only if $\mathsf{ct}_{y_i} = \mathsf{ct}'_{y_i}$ and π_{y_i} verifies. Rejects otherwise.
  - Generates its decryption share for ct_{y_i} and combines $(d_{i,\mathcal{U}}, d_{i,\mathcal{S}})$ for full decryption:

$$d_{i,\mathcal{S}} \leftarrow \textbf{HE.PartialDec}(\mathsf{sk}_{i,\mathcal{S}}^{\mathsf{HE}}, \mathsf{ct}_{y_i}); y_i \leftarrow \textbf{HE.Combine}(\mathsf{pk}_i^{\mathsf{HE}}, d_{i,\mathcal{U}}, d_{i,\mathcal{S}})$$

  - Updates $\mathsf{cnt}_i = \mathsf{cnt}_i + 1$

Figure 6: *Vɛrity*: Verifiable randomized response where the users do not have any authorized input.