Efficient Verifiable Differential Privacy with Input Authenticity in the Local and Shuffle Model

Tariq Bontekoe
University of Groningen
Groningen, The Netherlands
t.h.bontekoe@rug.nl

Hassan Jameel Asghar
Macquarie University
Sydney, Australia
hassan.asghar@mq.edu.au

Fatih Turkmen
University of Groningen
Groningen, The Netherlands
f.turkmen@rug.nl

Abstract

Local differential privacy (LDP) enables the efficient release of aggregate statistics without having to trust the central server (aggregator), as in the central model of differential privacy, and simultaneously protects a client's sensitive data. The shuffle model with LDP provides an additional layer of privacy, by disconnecting the link between clients and the aggregator. However, LDP has been shown to be vulnerable to malicious clients who can perform both input and output manipulation attacks, i.e., before and after applying the LDP mechanism, to skew the aggregator's results.

In this work, we show how to prevent malicious clients from compromising LDP schemes. Our only realistic assumption is that the initial raw input is authenticated; the rest of the processing pipeline, e.g., formatting the input and applying the LDP mechanism, may be under adversarial control. We give several real-world examples where this assumption is justified. Our proposed schemes for verifiable LDP (VLDP), *prevent both input and output manipulation attacks* against generic LDP mechanisms, requiring *only one-time interaction between client and server*, unlike existing alternatives [37, 43]. Most importantly, we are the first to provide an efficient scheme for *VLDP in the shuffle model*. We describe, and prove security of, two schemes for VLDP in the local model, and one in the shuffle model. We show that all schemes are *highly practical*, with client run times of less than 2 seconds, and server run times of 5–7 milliseconds per client.

Keywords

differential privacy, shuffle model, verifiable computing

1 Introduction

Most distributed data sharing applications either assume that the data obtained from a source is honestly obtained via the true input, or deviates arbitrarily from it. Accordingly, one abstracts these sources as either honest or malicious, with the received data inheriting corresponding labels. However, we argue that in many practical scenarios, the data processing pipeline at the source, from gathering raw input to formatted data to be sent to a central server, has more structure, which is not captured by such a simple abstraction [15]. The pipeline consists of several sequential components, that pass information to one another culminating in the final formatted data. In this case, we may realistically assume that the adversary only controls some components of the source, rather than the entire pipeline. This makes it possible to verify the validity of the claimed raw input and any subsequent processing on it, if the component receiving the raw input is outside adversarial reach. Our target scenario is collection and release of data from multiple distributed sources by a central server using differential privacy (DP) [27]. More specifically, we focus on the *local* [36] and *shuffle* [22] models of DP where the distributed nodes (data holders) do not trust the server (aggregator) with their formatted inputs in the clear. On the other hand, the server needs to ensure that the data received from the clients is correctly obtained from the true, raw input. The following *use cases* further motivate the aforementioned threat model.

In *sensor networks*, the main program of a sensor device decides which sensors to read data from and what data to send in a prescribed format to the server. The sensor device contains various sensors, which are individual hardware components, e.g., chips (see for example [40]). An adversary could (indirectly) take control of the sensor device by replacing this main program with its own malicious program, which may influence the local data processing pipeline. But, such a program does not corrupt the physical sensor itself, nor the raw data it produces.

Consider the case of energy companies obtaining the total (or average) power consumption of a group of households fitted with *smart meters* at regular time intervals. This data may reveal private information such as the sleeping patterns of house occupants [42]. Privacy to individual households can be provided by applying LDP to smart meter readings. Smart meters do not currently support such functionality, and implementing it in all of them is not cost-efficient. Fortunately, LDP could be applied via an app outside the smart meter. However, as this app is outside the trusted environment, it can be manipulated to serve a malicious purpose, i.e., it might output completely different data, and thereby skew statistics.

Smartphones and *wearables* share their location via GPS sensors, which can be used for crowd estimation to identify hotspots or for crowd control. Crowd estimation does not require exact GPS coordinates of individual devices; a coarse-grained aggregate location distribution often suffices. This is an excellent use case of LDP to release a location histogram. But, naive use of LDP may allow an attacker to send arbitrary locations, skewing the distribution. Here again we can assume raw GPS coordinates from the sensors as being true (operating system space), but the application sending location information to the server may be malicious (user space).

Many *other applications* fit this narrative, such as smartphone (e.g., accelerometer) or smart home (e.g., temperature) sensors. Raw values (events) from such sensors are collected in the operating system (OS) space, before the applications running in the user space can process them further for a given task, e.g., gesture detection. Last but not least, the raw inputs may be generated within a trusted

Proceedings on Privacy Enhancing Technologies YYYY(X), 1–21

https://doi.org/XXXXXXX.XXXXXXX

computing module. We give several concrete examples of trusted components in Section 5.1. Thus, we assume a setup where raw data is processed securely and correctly via one component (e.g., a secure enclave, OS space, a hardware module) before another program, possibly adversarial, further processes it before sending the formatted input to a server.

Assuming that the raw data is produced by a *trusted component* at a client enables us to verify it using *digital signatures*. However, we also need to verify that the same input is used in the rest of the processing pipeline; all without revealing the raw input!

Our Contributions.

- We propose three schemes for efficient verifiable LDP (VLDP). Our first (baseline) scheme requires multiple interactive rounds between client and server, similar to comparable works [37, 43]. However, unlike these works, our second scheme reduces the server load by requiring only one such round through randomness expansion techniques. Our final scheme further decreases the load on the server while simultaneously offering security in the shuffle model and in the presence of colluding clients.
- We present the *first* scheme for VLDP in the *shuffle model* [22]. In this model, a trusted shuffler shuffles the locally randomized inputs from the nodes, with the net effect of privacy amplification [8]. The shuffle model scheme cannot be straightforwardly constructed through our LDP schemes, since the requirements for verifiability and input authenticity on the one hand, and unlinkability (necessary for the shuffle model) on the other hand oppose one another. Our VLDP scheme in the shuffle model only adds marginal overhead for the client: approximately 1.8 seconds versus 0.6 and 1.1 seconds for our other schemes.
- All our schemes protect against both *input* and *output manipulation attacks* as defined in [37]. In fact, we are the first to deal with input manipulation attacks, i.e., where an attacker can arbitrarily change the initial input while carrying out the rest of the LDP algorithm faithfully. We do this by relying on digital signatures created by a trusted component. Similarly, we ensure protection against output manipulation, in which the attacker tries to send arbitrary outputs to the server, by using verifiable randomness and zero-knowledge proofs.
- We implement our VLDP schemes for the k-ary randomized response (k-RR) mechanisms for both histograms and real-valued data as described in [8]. Unlike existing works, which only target specific randomizers, our schemes can be applied to generic randomizers. In fact, as long as the randomization used in the LDP mechanism can be approximated using a fixed number of uniformly random bits, our protocol can accommodate it. Hence, our solutions can be *extended to many other LDP mechanisms*, e.g., Laplace or Staircase RR [27, 54], as detailed in Section 4.1.
- We implement and evaluate our protocols on two real-world datasets: a smart meter dataset to get the approximate energy consumption per household, and a GPS dataset to obtain the histogram of locations. Our results show that the protocols are highly practical and scalable. Each client takes a maximum of 2 seconds for a single LDP message, and the server takes less than 7 milliseconds per client. Furthermore, the communication cost is only 200–485 bytes per client value (plus a small additional one-time message), as we show in Section 7.3.

2 Related Work

In alignment with our scope, we restrict our discussion to protocols for verifiable differential privacy [15], while observing that, to the best of our knowledge, no constructions for verifiable DP in the shuffle model can be found in existing academic literature.

Local Model. The earliest work in this area appears to be on cryptographic k-RR from Ambainis et al. [3]. They do mention an even earlier work by Kikuchi et al. [38], who reinvented the notion of RR for voting and provided cryptographic constructions to protect against cheating voters. According to [3], the schemes from [38] are less efficient and provide weaker security guarantees than theirs.

The main security concern addressed by [3] is the privacy of the server (interviewer). Namely, the client (respondent) should not know the randomized outcome of her true input, because otherwise the respondent may end the protocol. To ensure this and to verify that the RR mechanism is correctly computed, they propose several protocols based on oblivious transfer, Pedersen commitments and zero-knowledge proofs. Randomness in the protocol is guaranteed by ensuring that the commitment parameters evaluate exactly to the probability of the correct or wrong response, requiring this probability to be a rational number. The communication cost therefore suffers for high precision. Their threat model is different from ours since we do not require privacy of the outcome of the LDP mechanism, and furthermore, it is not clear how their protocol can be extended to the shuffle model.

Kato et al. [37] extend [3] to several other variants of LDP (k-RR, unary encoding (OUE), local hashing (OLH)). However, their techniques are similar and once again assume that only the output can be manipulated, and the user otherwise uses the true value. Therefore, they do not ensure that the correct input is being used, which, in our case, can be verified through digital signatures.

The constructions of [3] and [37] are improved by Song et al. [51], who also use an approach based on random sampling. A client commits to a vector of entries that corresponds to the distribution of the randomized raw input. Subsequently, the server requests openings of a subset of these commitments, to obtain a perturbed sample, and obtain sufficient guarantees about the correctness of the received commitment vector. The authors show how to implement their techniques for k-RR and OUE randomizers. The communication and computation costs are clearly an improvement over prior work, however due to the used approach, the communication costs still suffer for high precision. Additionally, we do not see a clear way to efficiently extend their work to other randomizers.

In [44], the authors propose LDP with verifiable computing to extract binary attributes from anonymous credentials (e.g., older than 18). These binary attributes are certified through a third party using signatures, e.g., government or bank issued anonymous credentials. They do not give details on how this signature verification is done. They do provide detailed verifiable algorithms for binary RR and the exponential mechanism [41] to sample attributes in a range (e.g., the age). Unlike us, they do not provide protocols for k-RR, the shuffle model, and their protocols are not scrutinized using rigorous security definitions.

The closest work to ours is from [43] who tackle the problem of releasing an attribute associated with a transaction in a differentially private manner while maintaining the anonymity of a transaction

in a blockchain system for cryptocurrencies. They demonstrate their scheme using binary RR, although they do mention that the scheme can be generalized to non-binary attributes, without giving further details (as we show in Section 4.1, this is not trivial). The private attribute is signed by a registration authority, so that it can be verified if the correct input was used in the RR mechanism. They also check if the random coins used for RR are unbiased, through an interactive protocol between client and server. Finally, the client provides a NIZK proof as a proof of correct application of the RR mechanism. Our protocol has one element inspired by [43], namely generating joint randomness. But, unlike them, our protocols apply to generic LDP randomizers, protect against input manipulation, and provide LDP in the shuffle model. Moreover, apart from our baseline protocol, we only require a single round of interaction between client and server, which greatly reduces the communication load of both. This does come at the cost of a more expensive NIZK proof for each client, but this trade-off pays off quickly (see Section 7). Additionally, we do not suffer from the latency and scalability drawbacks caused by their dependence on blockchain.

Central Model. The construction from [45] is for verifying DP in the central model as opposed to the local model. The main threat model tackled in their paper is a dishonest analyst who may publish wrong results, banking on the inherent noise in DP, which is different than ours. The work from [53] uses a similar threat model to [45]. Randomness, however, is generated interactively between the curator and a "reader" (an entity interested in verifying the claims of the analyst). The work from [10] tackles the problem of verifiable DP in the single curator and multiparty setting. In the former, one server collects all client inputs. The server then provides differentially private answers to a third party, the analyst. In the multiparty setting, multiple servers receive inputs (secret shared) from clients and then compute the differentially private answer to a query that is then presented to the analyst. Input verification is limited to range checks, rather than verifying their exact values.

Concurrent to our work, Bell et al. [9] present their construction for verifying private probabilistic mechanisms. They consider a similar setting to [10] but provide stronger security guarantees. However, their solution only enables the server to answer counting queries and provide differential privacy through additive noise. Specifically, they show how to use the binomial mechanism, and leave possible extensions to other randomizers for future work. Contrary to [9], we do offer verifiability for generic LDP randomizers. Our techniques could potentially be extended to their work to support a wider variety of randomizers.

Finally, we note the scheme for confidential proofs of DP training from [50]. The authors show how to prevent output manipulation in DP-SGD [1] training of machine learning models in the central model. They use zero-knowledge proofs in combination with commitments to the entire dataset. However, due to enormous circuit size of a zero-knowledge proof for DP-SGD training, their method relies on interactive schemes with a heavy communication load. This makes their approach unsuitable for the local model.

Other Works. Another related work to ours, but which does not consider DP, is the ADSNARK system [7] for proving computation on authenticated data while maintaining privacy. Like us, they assume a trusted source that can provide authenticated initial data. The client is required to compute a function of this data and send the result to the server. To verify that the client has done the computation correctly, they propose their ADSNARK protocol based on Succinct Non-Interactive Arguments of Knowledge (SNARKs). Unlike us however, their setting is not distributed, and does not consider DP client inputs. Finally, we would like to point out several works showing the susceptibility of LDP to input manipulation (or data poisoning) attacks [19, 21, 39, 58], which highlight the need for cryptographic solutions for the integrity of initial data and their subsequent processing like our schemes.

3 Preliminaries and Building Blocks

We describe the building blocks used in our protocols with specific attention to zero-knowledge proofs and differential privacy.

PRGs from PRFs. A pseudo-random function (PRF) family is defined as a family of functions implemented by a key $k \in \mathcal{K}$, where $\mathcal{K}$ is the key space. A function $\mathsf{PRF}(k, x)$ from this family deterministically maps an input $x \in \mathcal{X}$ to an output $y \in \mathcal{Y}$. For a randomly chosen k, $\mathsf{PRF}(k, \cdot)$ is indistinguishable from a true random function. We can use PRFs to construct *pseudo-random number generators (PRGs)* [14, Section 4.4]: if we wish to sample a random bitstring from $\mathcal{Y}^\ell$, we define ℓ distinct, fixed input values $x_1, \ldots, x_\ell \in \mathcal{X}$. Then, we can define a PRG with seed $k \in \mathcal{K}$ as $\mathsf{PRG}(k) = \mathsf{PRF}(k, x_1)|| \ldots ||\mathsf{PRF}(k, x_\ell)$. We use this PRG construction to realize our schemes.

Commitment Schemes. A commitment scheme $C(x, r)$ takes as input a value x and randomness r, and outputs a commitment cm. The pair (x, r) is called the opening of the commitment. A secure commitment scheme should be both hiding and binding. *Binding* means that, given a commitment $C(x, r)$, it should be hard to output a pair (x', r'), with $x' \neq x$, such that $C(x', r') = C(x, r)$. *Hiding* implies that, given two commitments to distinct input values, it should be hard to determine which commitment belongs to which input value, i.e., given $x_0 \neq x_1$ and $\mathsf{cm}_b = C(x_b, r)$, for a random secret bit b and random secret r, it should be hard to determine b.

Digital Signature Schemes. A signature scheme Sig is a 3-tuple of p.p.t. algorithms (KeyGen, Sign, Verify), where KeyGen generates the keys, Sign creates a signature σ for a message m, and Verify asserts whether σ is valid for m. We only require that the signature scheme be secure against *existential forgeries under a chosen message attack (EUF-CMA)*, i.e., an adversary $\mathcal{A}$ should not be able to create a valid message-signature pair (m', σ') for a *new* message $m' \neq m$.

3.1 Zero-Knowledge Proofs

In our constructions, we rely upon *non-interactive zero-knowledge proofs (NIZKs).* NIZKs are used to prove the existence of a secret witness w for a given, public *statement* x, such that the pair satisfies some NP-relation $\mathcal{R}$, i.e., $(x, w) \in \mathcal{R}$. Specifically, we consider NIZKs in the common reference string (CRS) model [13, 25, 33], which can be defined as a 4-tuple of p.p.t. algorithms (Setup, Prove, Verify, Sim). The Setup algorithm generates the evaluation ek and verification vk keys (and a simulation trapdoor trap) for a given relation $\mathcal{R}$. Prove uses ek to create a valid proof π for a given statement-witness pair (x, w), and Verify uses vk to assert the correctness of π for a given statement x. Finally, Sim uses the trapdoor trap and ek to create a simulated proof for a statement x.

A secure NIZK scheme should satisfy the following (informal) properties. *Completeness:* given a true statement, an honest prover should be able to convince an honest verifier. *Soundness:* if the statement is false, no prover should be able to convince the verifier that it is true. *Zero-knowledge:* a proof π should reveal no information other than the truth of the public statement x, specifically it should leak no information about the witness w. (Note: our constructions only require honest-verifier zero-knowledge.)

However, we are not just interested in knowing that a witness exists, we also want to confirm that the prover *knows* this witness. Therefore, in the remainder of our work we will look at *NIZK proofs of knowledge (NIZK-PKs)*, which are NIZKs that additionally also satisfy knowledge soundness. *Knowledge soundness* is a stronger version of soundness that additionally requires the existence of an extractor $\mathcal{E}_{\mathcal{A}}$ that can produce a valid witness given complete access to the adversary $\mathcal{A}$'s state. In our implementation, we use zk-SNARKs [11]: NIZK-PK schemes that are also succinct, i.e., the verifier runs in $\text{poly}(\lambda + |x|)$ time and the proof size is $\text{poly}(\lambda)$.

3.2 Differential Privacy

Differential privacy (DP) [27] is a formal way of describing database privacy. It provides precise measures for how much information about a dataset is leaked by (partial) disclosure through queries on the dataset. Consider a database X containing n entries from the domain $\mathcal{X}$, i.e., $X \in \mathcal{X}^n$. We consider two databases $X, X' \in \mathcal{X}^n$ as neighbors, denoted $X \sim X'$, if they differ in exactly one entry.

Definition 1 (Differential Privacy [27])**.** A randomized algorithm $\mathcal{M} \colon \mathcal{X}^n \to \mathcal{Y}$ is (ϵ, δ)-differentially private, if for all $X \sim X' \in \mathcal{X}^n$ and for all $T \subseteq \mathcal{Y}$, we have $\Pr[\mathcal{M}(X) \in T] \leq e^{\epsilon} \Pr[M(X') \in T] + \delta$.

Any (ϵ, δ)-DP randomization algorithm has two highly useful properties, following [8]:

Lemma 1 (Post-processing [27]). *If $\mathcal{M}$ is (ϵ, δ)-DP, then for every (deterministic or randomized) $\mathcal{M}'$, $\mathcal{M}' \circ \mathcal{M}$ is also (ϵ, δ)-DP.*

Lemma 2 (Sequential composition [28]). *If $\mathcal{M}_1, \ldots, \mathcal{M}_n$ are (ϵ, δ)-DP, then the composed algorithm $\mathcal{M}' = (\mathcal{M}_1, \ldots, \mathcal{M}_n)$ is $(\epsilon', \delta' + n\delta)$-DP for any $\delta' > 0$ and $\epsilon' = \epsilon(e^{\epsilon} - 1)n + \epsilon\sqrt{2n \log(1/\delta')}$.*

Shuffle Model. In the shuffle model, there are n clients, each of whom holds a data entry $x_i \in \mathcal{X}$. The shuffle model considers three algorithms, following the definitions of [22]:

- A randomizer $\mathcal{R} \colon \mathcal{X} \to \mathcal{Y}$ that takes as input a data entry x_i and outputs a value $\tilde{x}_i \in \mathcal{Y}$.[1]
- A shuffler $\mathcal{S} \colon \mathcal{Y}^n \to \mathcal{Y}^n$ that takes as input a vector of n messages and outputs these in a random order. Specifically, on input $(\tilde{x}_1, \ldots, \tilde{x}_n)$, $\mathcal{S}$, outputs $(\tilde{x}_{\pi_1}, \ldots, \tilde{x}_{\pi_n})$, where π is a uniform random permutation of $[n]$.
- An aggregator, or analyst, $C \colon \mathcal{Y}^n \to \mathcal{Z}$, that takes as input a vector of n messages $(\tilde{x}_{\pi_1}, \ldots, \tilde{x}_{\pi_n})$ and outputs an estimation of $f(x_1, \ldots, x_n)$.

Definition 2 (DP in the Shuffle Model [22])**.** A protocol $(\mathcal{R}, \mathcal{S}, C)$ is (ϵ, δ)-DP if the protocol $\mathcal{S}(\mathcal{R}(x_1), \ldots, \mathcal{R}(x_n))$ is (ϵ, δ)-DP.

As a consequence of Lemma 2 and Lemma 1, there is a composition property equivalent to Lemma 2 for the shuffle model [22].

[1] We only consider the single-message shuffle model. The more general shuffle model allows for an array of m messages to be output by $\mathcal{M}$.

LDP Algorithm for Reals [8]	LDP Algorithm for Histograms [8]
input: $k \in \mathbb{N}, x \in [0,1], \gamma \in [0,1]$	**input:** $k \in \mathbb{N}, x \in [k], \gamma \in [0,1]$
$\overline{x} \leftarrow \lfloor xk \rfloor + \text{Ber}(xk - \lfloor xk \rfloor)$	$b \leftarrow \text{Ber}(\gamma)$
$b \leftarrow \text{Ber}(\gamma)$	**if** $b = 0$ **do**
if $b = 0$ **do**	$\tilde{x} \leftarrow x$
$\tilde{x} \leftarrow \overline{x}$	**else**
else	$\tilde{x} \leftarrow\!\!\$\ \{1, \ldots, k\}$
$\tilde{x} \leftarrow\!\!\$\ \{0, 1, \ldots, k\}$	**return** $\tilde{x}$
return $\tilde{x}$	

Figure 1: LDP randomizers for reals and histograms.

Local Differential Privacy (LDP). When one replaces the shuffler $\mathcal{S}$ by an identity function, i.e., the vector of messages is not shuffled, we are left with the well-known model for LDP [36]:

Definition 3 (Local Differential Privacy)**.** A randomized algorithm $\mathcal{R} : \mathcal{X} \to \mathcal{Y}$ is $(\epsilon, \delta) - LDP$, if for all pairs $x, x' \in \mathcal{X}$, and for all $T \subseteq \mathcal{Y}$, we have $\Pr[\mathcal{R}(x) \in T] \leq e^{\epsilon} \Pr[\mathcal{R}(x') \in T] + \delta$.

The purpose of the shuffle mechanism is to amplify the privacy achievable via LDP. We give a concrete example of this in Section 4. In the remainder of this work, when we refer to an LDP algorithm, we will only denote the local randomizer, unless stated otherwise.

4 DP Algorithms

We consider two LDP algorithms, both of which appear in [8]. The first locally randomizes a real-number input $x \in [0, 1]$. The goal of the aggregator is to output the sum of these inputs from n users. The second algorithm takes as input an integer $x \in [k]$ for $k \geq 2$, and locally randomizes it. The application in this case is a histogram of values in $[k]$. The algorithms are shown in Figure 1.

In the LDP algorithm for reals, without loss of generality, we assume $x \in [0, 1]$. For a precision level k, we first encode x as an integer as $\overline{x} = \lfloor xk \rfloor + \text{Ber}(xk - \lfloor xk \rfloor)$ [8]. It is easily verified that this encoding ensures that $\mathbb{E}(\overline{x}/k)$, which is the expected value of the decoded $\overline{x}$, is exactly $\mathbb{E}(x)$. This makes the range of $\overline{x}$ equal to $\{0, 1, \ldots, k\}$. This algorithm is ϵ-DP, as long as $\frac{1-k\gamma/(k+1)}{\gamma/(k+1)} \leq e^{\epsilon}$. Equating the left hand side to the right hand side, we get $\gamma = \frac{k+1}{e^{\epsilon}+k}$.

Thus, we can set γ to this value given ϵ and k. If $\mathcal{R}$ is (ϵ, δ)-LDP, then the mechanism $\mathcal{M} : \mathcal{X}^n \to \mathcal{Y}^n$ defined as $\mathcal{M}(x_1, \ldots, x_n) = \mathcal{R}^n = (\mathcal{R}(x_1), \ldots, \mathcal{R}(x_n))$ is also (ϵ, δ)-DP.

For the LDP algorithm for histograms we can determine γ using the same equation as above. However, we need to replace each occurrence of k by $k - 1$, due to the different range for x.

Aggregator. The aggregator for the LDP histogram algorithm simply outputs the histogram, i.e., the number of inputs for each $i \in [k]$. For the LDP algorithm for reals, the aggregator should de-bias first. Let x_i be the i-th user's input, let $\overline{x}_i$ be the same input with precision k, and $\tilde{x}_i$ the i-th user's output of the LDP algorithm. Then, as shown in Appendix A.1, the aggregator outputs $\frac{1}{1-\gamma}\left(\frac{\sum_{i=1}^n \tilde{x}_i}{k} - \frac{\gamma n}{2}\right)$, as estimate of $\frac{1}{k}\sum_{i=1}^n \overline{x}_i$ [8], which itself estimates $\sum_{i=1}^n x_i$.

Shuffle Model. In the shuffle model, the inputs from all parties are first shuffled randomly, and then given to the aggregator. This

results in privacy amplification, as the aggregator now does not know which input belongs to which user. The shuffle model of DP employs a shuffler $\mathcal{S}\colon \mathcal{Y}^n \to \mathcal{Y}^n$, which is a random permutation of its inputs. The algorithm $\mathcal{M} \coloneqq \mathcal{S} \circ \mathcal{R}^n\colon \mathcal{X}^n \to \mathcal{Y}^n$ then provides (ϵ, δ)-DP against the curator, but with the advantage that the local randomizer $\mathcal{R}$ need only be $(\epsilon_0, 0)$-LDP, with ϵ_0 greater than ϵ. Ignoring logarithmic terms, ϵ_0 is proportional to n and inversely proportional to δ. Given a value of ϵ, δ and n, we can use the script provided by [8] to obtain a value of ϵ_0 which uses a tighter analysis than given by the implicit bounds in the paper. For instance, for the LDP histogram mechanism described above with $k = 10$, i.e., k-ary RR, with $n = 100$ participants, $\delta = 10^{-6}$ and $\epsilon = 0.1$, we get $\epsilon_0 \approx 1.0032$ through the Bennett bound. Thus, we can use the mechanism *10 times more* than the LDP mechanism alone.

4.1 LDP inside NIZK

To verify the above LDP algorithms inside a NIZK circuit, we need to define how we evaluate an LDP algorithm in a *deterministic* fashion, given a *fixed* number of uniform random bits. It must be deterministic in the sense that we need to be able to 'recreate' random sampling inside the NIZK circuit. Moreover, we observe that the NIZK proof is computed over a given, fixed, agreed upon relation $\mathcal{R}$. Therefore, the (maximum) number of random bits used should be fixed and known up front. This has the downside that we cannot sample exactly from each distribution, but rather need to sample from *approximate* distributions. We tackle both issues simultaneously, by defining how to use a uniform random bitstring ρ of length ℓ, such that the distribution of $\mathsf{LDP.Apply}(x; \rho)$ is statistically close to the true randomized LDP algorithm.

Our Approximate Sampling Methods. Approximations for NIZK encoding of LDP randomizers can be designed for most well-known distributions. For the LDP randomizers defined in Figure 1, we only need to approximate the Bernoulli distribution and the Discrete Uniform distribution. Figure 2 shows two algorithms for sampling from these distributions. We give an additional example in Appendix A.2.

These sampling methods clearly match our requirements, and are also statistically close to the true distributions, with the statistical distance decreasing exponentially in ℓ. These approximate sampling algorithms replace the random sampling steps in Figure 1. We give an exact specification of the resulting algorithms in Appendix A.3. For a sufficiently large bitsize of ρ, these algorithms are statistically close approximations of the true LDP algorithms. Moreover, the approximation error decreases exponentially in $|\rho|$.

Generalization to Other Randomizers. The previous construction can be generalized to many other distributions, thereby supporting our claim that our schemes are applicable to a wide class of local randomizers. The essential difference in the construction across randomizers is showing how to efficiently (approximately) sample from the underlying distributions used by the randomizer. The rest of the adaptations to the NIZK proof are straightforward. Thus, below we discuss (at a high level) approaches for approximate sampling from other representative or state-of-the-art LDP randomizers to provide further evidence for the feasibility of encoding them inside NIZK circuits. Additionally, we discuss estimates for $|\rho|$, since this dominates the computation cost (see also Section 7).

$\widetilde{\text{Ber}}(\gamma; \rho)$

input: $\gamma \in [0,1], \rho \in \{0,1\}^\ell$
Interpret ρ as an integer
if $\rho \leq \lfloor \gamma \cdot (2^\ell - 1) \rfloor$ **do**
  $b \leftarrow 1$
else
  $b \leftarrow 0$
return b

$\widetilde{\text{Unif}}([lb, ub]; \rho)$

input: $lb, ub \in \mathbb{Z} : lb < ub, \rho \in \{0,1\}^\ell$
Interpret ρ as an integer
$\Delta \leftarrow \lfloor 2^\ell / (ub - lb + 1) \rfloor$
for j **in** $\{0, \ldots, ub - lb - 1\}$
  if $j \cdot \Delta \leq \rho < (j+1) \cdot \Delta$ **do**
    return $lb + j$
return ub

Figure 2: Algorithms for approximately sampling from the Bernoulli and Discrete Uniform distribution.

- *Laplace noise* [27] is often used in LDP to perturb continuous input values. For approximate Laplace sampling, one can first sample a sign bit (by taking the first bit of ρ) and then sample an exponentially distributed 'distance' ℓ. The latter can be approximated arbitrarily closely by sampling ℓ from a Poisson distribution as described in [26]: one samples ℓ in binary, where the i-th bit has a predefined bias γ_i, i.e., it is sampled from $\widetilde{\text{Ber}}(\gamma_i)$. The size of ρ for this approach is $|\rho| \approx |\ell| \cdot \text{precision}(\widetilde{\text{Ber}})$, e.g., for 64-bit precision, we have $|\rho| \approx 512$ bytes.
- *RAPPOR* [31] was developed by Google and used as part of the Chrome browser. It adds LDP noise to users' responses to questions. First it encodes a raw input using a Bloom filter of size ℓ_B, i.e., by hashing the input to a bit vector using different hash functions for each vector entry. Bloom filters generally use simple hash functions, that are evaluated at little cost inside a NIZK circuit. The resulting bit-vector is subsequently transformed using Bernoulli random sampling. Thus, we can use $\widetilde{\text{Ber}}$ (Figure 2) to approximate this efficiently. The amount of random bits required is $|\rho| \approx \ell_B \cdot \text{precision}(\widetilde{\text{Ber}})$, e.g., for 64-bit precision and a 20-bit Bloom filter, we have $|\rho| \approx 160$ bytes.
- *Staircase Randomized Response (SRR)* [54] is a recent LDP algorithm for perturbing location data. It requires sampling from a 'staircase' shaped distribution, where locations close to the true one are more likely to be sampled than locations further away. SRR relies on a specific bit-string encoding of locations with length ℓ (no more than 46 bits in practice), where the common prefix length is inversely proportional to the distance between two locations. Thus, given a true location x, we compute three values from ρ; the length of the common prefix, the value of the first different bit-pair, and the remaining bits. The first two can be sampled using $\widetilde{\text{Unif}}$ (Figure 2) and the latter can be taken directly from ρ. The resulting LDP value $\tilde{x}$ is then computed using simple if-else statements. We need $|\rho| \approx \ell + 2 \cdot \text{precision}(\widetilde{\text{Unif}})$ for this approach, e.g., for 64-bits precision, we have $|\rho| \approx 22$ bytes.

Many more randomizers can be approximated similarly. *Gaussian noise* can be approximated by repeated Bernoulli random sampling [26]. Similarly, generic *Randomized Response (RR)* [35, 57] or *Subset Selection mechanisms* [55, 59] are easily computed using (repeated) Bernoulli or Uniform random sampling. [56] presents a general framework for LDP randomizers for frequency estimation. This framework splits randomizers in an encoding and a perturbation step. The listed encodings — direct, histogram, unary, and

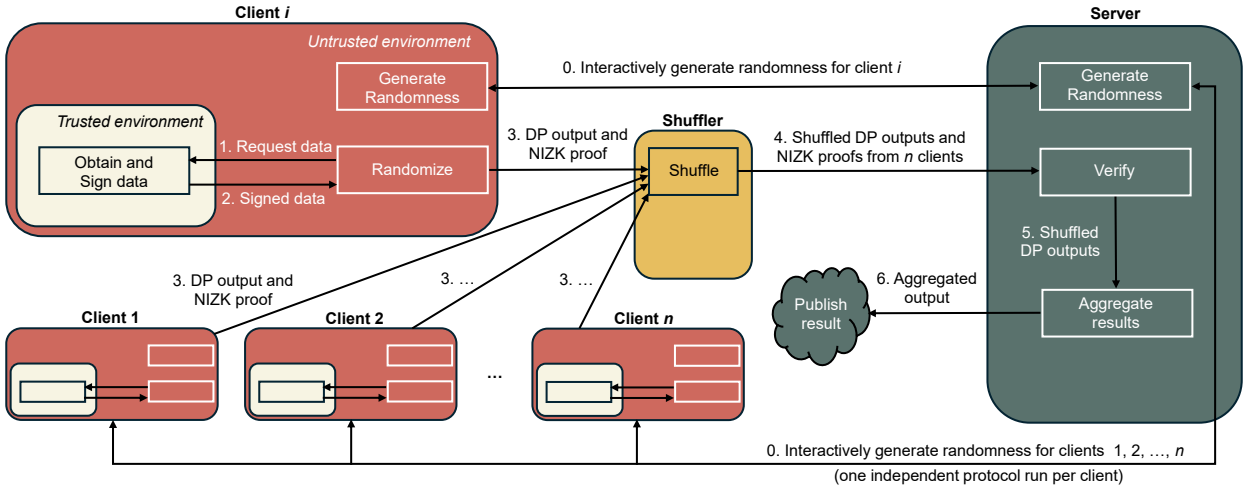

Figure 3: System model for the VLDPPipeline**. For multiple time steps j, the clients reiterate the steps as explained further on. When using the 'regular' local model, the shuffler is removed and the messages of step 3 are sent directly to the server instead.**

local hashing (like in RAPPOR) — are all easily expressed in NIZK circuit constraints and will have small to negligible overhead. The listed perturbations are either based on RR or Laplace noise, which we have discussed above.

Finally, we observe that stateful LDP randomizers (such as [30]), i.e., randomizers whose behavior depends on previous calls to it, would require the NIZK proof to additionally verify the state update. Whilst verifying this update is not a problem per se, it would require the previous state as input to the proof, which is not directly supported by our constructions (see Section 6). Since the overwhelming majority of LDP randomizers is not stateful, we leave a solution to this challenge as future work.

5 Verifiable DP in the Local and Shuffle Model

First, we describe the threat model. Next, we sketch our system model and give a formal definition for a VLDP scheme that is applicable to both the local and the shuffle model. Finally, we present formal security definitions.

5.1 Threat Model

There are three types of actors in the shuffle model: clients, shuffler and server (see also Figure 3). The shuffler can be ignored for the local model. We describe the threat model according to each actor.

5.1.1 Clients. We assume that all client programs may potentially behave *maliciously*, or collude with other clients, meaning that they could deviate from our scheme arbitrarily, or attempt to use false input data. However, client programs have no control over the trusted environment and can only obtain signed input data x from it, with signature $\sigma_x = \mathsf{Sig.Sign}_{\mathsf{sk}_i}(x||t_x)$. Each client i's trusted environment contains a unique secret key sk_i, which cannot be accessed by the potentially malicious client program. Recall that we use the term trusted environment to denote any controlled environment outside adversarial reach, e.g., secure enclave, OS space, or hardware module.

Examples of Trusted Environments. Our model is applicable in situations where a trusted environment is viable. Given that the presence of trusted environments in consumer hardware is increasingly strong, this is a reasonable requirement. A concrete example is Apple's Secure Enclave [5]; implemented on iPhones and wearables like Apple watches and HomePods. The enclave supports EdDSA and ECDSA signatures (see our discussion in Section 7.1). Our protocol minimizes processing within the enclave to input signing only. Thus, the trusted module can be small. The rest of the pipeline is executed outside the enclave, and is not assumed to be trustworthy.

Another example is kernel-space vs user-space in Linux-based OSes. Apps can only access user-space memory. Thus any malicious app on a victim phone can not directly access hardware; only indirectly via the kernel [4]. We recognize that attacks on trusted environments or kernels (jailbreaks) exist. Yet, they would also apply to any work based on trusted execution environments (TEEs). Additionally, it is not straightforwardly clear if there is an approach for significantly reducing the reliance on some sort of trusted environment. The idea of loosening or removing this assumption is left to future work.

5.1.2 Server. We assume the server to be *semi-honest*, i.e., it will not deviate from the scheme, but does try to obtain as much information as possible whilst following the scheme. Moreover, the server is assumed to be a non-colluding entity. Finally, we assume that the server can verify which pk_i's are known/trusted public keys belonging to a trusted environment, e.g., by means of a public key infrastructure or whitelist of trusted keys.

5.1.3 Shuffler. For the scheme in the shuffle model, we assume that the shuffler is an *honest-but-curious*, independent, *non-colluding* party. In this work, for the sake of clarity, we will assume that the shuffler is a trusted third party. In practice, different methods exist for implementing a shuffler, e.g., using mixnets. We discuss these in Appendix B. The actual choice of implementation for the shuffler is out of scope for this work, as our focus lies on constructing efficient,

implementation-agnostic, secure VLDP schemes for the local and shuffle model. Note that it is common for works in the shuffle model to leave this discussion out of scope [8, 22].

5.2 System Model

Let VLDPPipeline denote the high-level structure of a VLDP scheme, which describes the workings of the VLDP scheme with 1 server and n clients for T time steps (one for each message). A schematic overview is shown in Figure 3. First, GenRand ensures that the client has the necessary inputs to construct verifiably true random values later on. It can be seen as a sort of preprocessing, where client and server together generate client-specific randomness to be used in the j-th time interval $(t_{j-1}, t_j]$, for $j \in [T]$.

Client i generates a VLDP value $\tilde{x}_{i,j}$ for the j-th time interval by first requesting a fresh raw input value $x_{i,j}$ from the trusted environment at time $t_x^{i,j} \in (t_{j-1}, t_j]$. In response, the trusted environment returns a signed input value x with signature $\sigma_x^{i,j} = \mathsf{Sig.Sign}_{\mathsf{sk}_i}(x_{i,j} || t_x^{i,j})$, where sk_i is the secret key of i's trusted environment, which we assume has been generated beforehand. This signature can be verified using the corresponding pk_i. Subsequently, i calls Randomize to verifiably perturb $x_{i,j}$ and obtain $\tilde{x}_{i,j}$ and a correctness proof $\pi_{i,j}$, both of which are sent to the shuffler (or directly to the server in the local model). The shuffler collects all messages for $(t_{j-1}, t_j]$: $((\tilde{x}_{1,j}, \pi_{1,j}), (\tilde{x}_{2,j}, \pi_{2,j}), \ldots, (\tilde{x}_{n,j}, \pi_{n,j}))$ and forwards these in random order $((\tilde{x}_{?_1,j}, \pi_{?_1,j}), (\tilde{x}_{?_2,j}, \pi_{?_2,j}), \ldots, (\tilde{x}_{?_n,j}, \pi_{?_n,j}))$ to the server, thus ensuring that the server cannot determine which message belongs to which client.

For each received $(\tilde{x}_{?_i,j}, \pi_{?_i,j})$, the server runs Verify, to ensure that $\tilde{x}_{?_i,j}$ is correctly randomized from a value x, with $t_x \in (t_{j-1}, t_j]$ signed by a valid $\mathsf{pk}_{?_i}$. Finally, the server uses all valid values to evaluate and publish its desired output $f(\tilde{x}_{?_1,j}, \ldots, \tilde{x}_{?_n,j})$.

Definition 4 (VLDP Scheme)**.** A VLDP scheme for an LDP algorithm $\mathsf{LDP.Apply}\colon \mathcal{X} \to \mathcal{Y}$ is a 5-tuple of p.p.t. algorithms $\mathcal{VLDP}$ for any number $n \geq 1$ of clients and one prover:

- $\mathsf{Setup}(1^\lambda) \to \mathsf{pp}$: Given the security parameter λ, this algorithm returns public parameters pp. This is a tuple containing the NIZK relation $\mathcal{R}$, parameters of a public key signature scheme $\mathsf{pp}_{\mathsf{sig}}$ and a commitment scheme $\mathsf{pp}_{\mathsf{comm}}$. Optionally, it also returns a vector $\vec{s}$ of T PRF seeds.
- $\mathsf{KeyGen}(\mathsf{pp}) \to (\mathsf{ek}, \mathsf{vk}, \mathsf{pk}_s, \mathsf{sk}_s, L)$: Given the public parameters pp, this algorithm returns the evaluation ek and verification key vk for the NIZK proof, and the server's public and secret signature keys $(\mathsf{pk}_s, \mathsf{sk}_s)$ along with a list L. The list L is populated with the identities of clients that have already been processed in a given time interval.
- $\mathsf{GenRand}(\mathsf{pp}, \mathsf{aux}) \to \mathsf{out}_c^i$: This interactive protocol between a single client and server takes as input the public parameters pp and optional auxiliary information aux. The output of the client is defined as out_c^i, which contains client-generated randomness, commitment to this randomness, server generated randomness, and a server signed signature binding the server generated randomness with the commitment to client's randomness. Depending on the scheme's instantiation, out_c^i can be used for multiple time intervals or only for one.
- $\mathsf{Randomize}(\mathsf{pp}, \mathsf{ek}, t_j, \mathsf{out}_c^i, x, t_x, \sigma_x) \to (\tilde{x}, \pi, \tau_x)$: Client i uses pp, ek, a timestamp t_j, its output out_c^i from GenRand, the true input value x with timestamp t_x and signature σ_x to compute an LDP value $\tilde{x}$, a NIZK proof π, and a vector of public values τ_x. [2]
- $\mathsf{Verify}(\mathsf{pp}, \mathsf{vk}, t_j, \tilde{x}, \pi, \tau_x) \to \tilde{x} \cup \perp$: The server uses pp, vk, a timestamp t_j, $\tilde{x}$, π, and τ_x to verify whether $\tilde{x}$ was computed honestly. If so, it returns $\tilde{x}$, and $\perp$ otherwise.[2]

5.3 Security definitions

A VLDP scheme should satisfy at least *completeness*, *soundness*, and *zero-knowledgeness*. Below, we provide the formal definitions of all these properties. The experiments used in the definitions are detailed in Appendix C.1, together with our formal security proofs.

Completeness guarantees that for any authenticated input x, created in the right time interval, the output of an honest client will be accepted by an honest server with probability 1.

Definition 5 (Completeness)**.** A scheme $\mathcal{VLDP}$ for an LDP method $\mathsf{LDP.Apply}\colon \mathcal{X} \to \mathcal{Y}$ with security parameter λ is complete if for any $n = \mathsf{poly}(\lambda)$, any $T = \mathsf{poly}(\lambda)$, and for all p.p.t. $\mathcal{A}$, we have $\Pr\left[\perp \in \mathbf{Exp}_{\mathcal{A}}^{\text{Comp}}(1^\lambda, n, T)\right] \leq \mathsf{negl}(\lambda)$, with $\mathbf{Exp}_{\mathcal{A}}^{\text{Comp}}$ as defined in Figure 9.

On the other hand, soundness guarantees that no dishonest client can make a server accept an output, that is not an honest randomization of an authentic input x, except with negligible probability.

Definition 6 (Soundness)**.** A scheme $\mathcal{VLDP}$ for an LDP method $\mathsf{LDP.Apply}\colon \mathcal{X} \to \mathcal{Y}$ with security parameter λ is sound if, for any $n = \mathsf{poly}(\lambda)$, any $T = \mathsf{poly}(\lambda)$, for all p.p.t. $\mathcal{A}$, and $\forall (x_{i,j}, y_{i,j}) \in \mathcal{X} \times \mathcal{Y}$, we have

$$\Big|\Pr\big[\mathsf{LDP.Apply}(x_{i,j}; \rho_{i,j}) = \{y_{i,j}\}_{i,j} \big| \rho_{i,j} \leftarrow\!\$\ \{0,1\}^*\big] - \Pr\Big[\mathbf{Exp}_{\mathcal{A},S^*}^{\text{Snd-Real}}(1^\lambda, n, T, \{x_{i,j}\}_{i,j}) = \{y_{i,j}\}_{i,j} \Big| \perp \notin \mathbf{Exp}_{\mathcal{A},S^*}^{\text{Snd-Real}}(1^\lambda, n, T, \{x_{i,j}\}_{i,j})\Big]\Big| \leq \mathsf{negl}(\lambda),$$

with $\mathbf{Exp}_{\mathcal{A}}^{\text{Snd-Real}}$ as defined in Figure 10, where S^* denotes an honest server that the adversary can interact with.

The zero-knowledge property guarantees that the server learns nothing about the original input value x, other than what could already be learned from its randomization $\tilde{x}$.

Definition 7 (Zero-knowledge)**.** A scheme $\mathcal{VLDP}$ for an LDP method $\mathsf{LDP.Apply}\colon \mathcal{X} \to \mathcal{Y}$ with security parameter λ is zero-knowledge if for any $n = \mathsf{poly}(\lambda)$, any $T = \mathsf{poly}(\lambda)$, there exists a p.p.t. simulator $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2)$, such that for all p.p.t. adversaries $\mathcal{A}$, and $\forall (x_{i,j}, y_{i,j}) \in \mathcal{X} \times \mathcal{Y}$, we have

$$\Big\{\mathbf{Exp}_{\mathcal{A}}^{\text{Zk-Real}}(1^\lambda, n, T, \{x_{i,j}\}_{i,j}) = (\cdot, \{y_{i,j}\}_{i,j})\Big\} \overset{c}{\equiv} \Big\{\mathbf{Exp}_{\mathcal{A},\mathcal{S}}^{\text{Zk-Sim}}(1^\lambda, n, T, \{y_{i,j}\}_{i,j})\Big\},$$

with $\mathbf{Exp}_{\mathcal{A}}^{\text{Zk-Real}}$ and $\mathbf{Exp}_{\mathcal{A},\mathcal{S}}^{\text{Zk-Sim}}$ as defined in Figure 11.

Additionally, for a VLDP scheme to be secure in the shuffle model, we require *shuffle indistinguishability*, i.e., the server cannot discern an output sent by client i from an output sent by client i'.

[2] We leave out the index-pair (i, j) for $x, t_x, \sigma_x, \tilde{x}, \pi, \tau_x$ to improve legibility.

Definition 8 (Shuffle indistinguishability). A scheme $\mathcal{VLDP}$ for an LDP method $\mathsf{LDP.Apply}\colon \mathcal{X} \to \mathcal{Y}$ with security parameter λ has shuffle indistinguishability if for every p.p.t. adversary $\mathcal{A}$: $2|\Pr[\mathbf{Exp}^{\mathrm{Ind}}_{\mathcal{A}}(\lambda) = 1 | \mathbf{Exp}^{\mathrm{Ind}}_{\mathcal{A}}(\lambda) \neq \perp] - \frac{1}{2}| \leq \mathsf{negl}(\lambda)$, with $\mathbf{Exp}^{\mathrm{Ind}}_{\mathcal{A}}$ as defined in Figure 12.

6 Our Constructions for VLDP

In this section, we present our three VLDP schemes and explain the components that together form their respective construction of VLDPPipeline. Each scheme improves upon the previous one, culminating in an efficient VLDP scheme that can be applied in the shuffle model. Appendix C.2 contains formal security analyses.

(1) The Base scheme achieves verifiable LDP in the local model. Its GenRand protocol is loosely inspired by the VerRR algorithm in [43] and should be run once per time interval (and client). The other algorithms are novel constructions, which together form a scheme that, unlike [43], also provides security against input manipulation attacks for authenticated data, supports generic LDP algorithms, and does not require a blockchain.
(2) The Expand scheme provides the same guarantees, but enables clients to reuse their output out^i_c of GenRand for every time interval. This significantly decreases the computation and communication load of the server, making the scheme suitable for sequential composition of DP.
(3) The Shuffle scheme has the same communication efficiency as Expand, but also achieves VLDP in the shuffle model.

6.1 Base Scheme

Figure 4 describes the Base scheme in detail. Each client i obtains fresh randomness for time interval j by running an independent instance of $\mathsf{GenRand}_{\mathsf{base}}$ with the server. Together, they compute the necessary values to construct a true random value $\rho_{i,j}$ for later use in $\mathsf{Randomize}_{\mathsf{base}}$. The bit length of ρ will be equal to the output of the PRF used to generate ρ, and is denoted by $|\rho|$. In case the required number of bits ℓ needed to evaluate $\mathsf{LDP.Apply}()$ is lower than $|\rho|$, we can simply ignore the unused bits. However, in case $\ell > |\rho|$, we need to evaluate the PRF on one or more additional inputs, depending on ℓ, and concatenate the results. For clarity, we assume that $\ell \leq |\rho|$ in our scheme definitions, since it can be extended easily using this method. In our experimental evaluations (Section 7), we evaluate the influence of ℓ on the performance.

In an instance of $\mathsf{GenRand}_{\mathsf{base}}$, client i first generates its own random bits $\rho^{i,j}_c$ (we explicitly show the use of a PRF in step 1 and 2 of Figure 4 to resemble the later schemes). Subsequently, i computes a commitment $\mathsf{cm}^{i,j}_{\rho_c}$ to $\rho^{i,j}_c$, and shares it along with its trusted environment's public key pk_i, and a time interval marker t_j with the server. The eventual randomness is then also bound to t_j, such that the client cannot create a large batch of random values, and then pick a specific value from this batch. That would clearly violate the requirements for verifiable randomization.

The server first checks whether pk_i indeed belongs to i, and that i did not previously construct a random value for t_j, i.e., whether $(i, t_j) \notin L$. Next, the server generates a valid PRF seed $k^{i,j}_s$ and computes $\sigma^{i,j}_s = \mathsf{Sig.Sign}_{\mathsf{sk}_s}(\mathsf{pk}_i || \mathsf{cm}^{i,j}_{\rho_c} || k^{i,j}_s || t_j)$. The server then sends $(k^{i,j}_s, \sigma^{i,j}_s)$ to i, who verifies $\sigma^{i,j}_s$. Note that, rather than using a signature, the server could instead maintain a list of $(\mathsf{pk}_i, \mathsf{cm}^{i,j}_{\rho_c})$ for each client and compare this state in $\mathsf{Verify}_{\mathsf{base}}$ later. We, however, choose this approach to minimize the server's storage load.

In $\mathsf{Randomize}_{\mathsf{base}}$, the client computes the server part of the randomness $\rho^{i,j}_s$ from $k^{i,j}_s$, and combines the client and server

$\mathsf{VLDPPipeline}_{\mathsf{base}}$

1: $\mathsf{pp} \leftarrow \mathsf{Setup}_{\mathsf{base}}(1^\lambda)$
2: Server computes $(\mathsf{ek}, \mathsf{vk}, \mathsf{pk}_s, \mathsf{sk}_s, L) \leftarrow \mathsf{KeyGen}_{\mathsf{base}}(\mathsf{pp})$
3: **for** Each client i **in** $\{1, \ldots, n\}$ (in parallel)
4: **for** j **in** $\{1, \ldots, T\}$
5: Client i obtains $\mathsf{out}^{i,j}_c = \mathsf{GenRand}_{\mathsf{base}}(\mathsf{pp}, t_j)$
6: Client i obtains fresh $(x_{i,j}, t^{i,j}_x, \sigma^{i,j}_x = \mathsf{Sig.Sign}_{\mathsf{sk}_i}(x||t_x))$
7: Client i runs $(\tilde{x}_{i,j}, \pi_{i,j}, \tau_{i,j}) = \mathsf{Randomize}_{\mathsf{b.}}(\mathsf{pp}, \mathsf{ek}, t_j, \mathsf{out}^{i,j}_c, x_{i,j}, t^{i,j}_x, \sigma^{i,j}_x)$
8: Server obtains $\tilde{x}_{i,j} = \mathsf{Verify}_{\mathsf{base}}(\mathsf{pp}, \mathsf{vk}, t_j, \tilde{x}_{i,j}, \pi_{i,j}, \tau_{i,j})$
9: Server computes result from all $\tilde{x}_{i,j}$

$\mathsf{KeyGen}_{\mathsf{base}}(\mathsf{pp})$

1: $(\mathsf{ek}, \mathsf{vk}) \leftarrow \mathsf{NIZK\text{-}PK.KeyGen}(\mathcal{R}_{\mathsf{base}})$
2: $(\mathsf{sk}_s, \mathsf{pk}_s) \leftarrow \mathsf{Sig.KeyGen}(\mathsf{pp}_{\mathsf{sig}})$
3: $L \leftarrow \emptyset$
4: **return** $(\mathsf{ek}, \mathsf{vk}, \mathsf{pk}_s, \mathsf{sk}_s, L)$

$\mathsf{Setup}_{\mathsf{base}}(1^\lambda)$

1: $\mathsf{pp}_{\mathsf{sig}} \leftarrow \mathsf{Sig.Setup}(1^\lambda)$
2: $\mathsf{pp}_{\mathsf{comm}} \leftarrow \mathsf{Comm.Setup}(1^\lambda)$
3: $\vec{t} = (t_0, \ldots, t_T)$
4: $\mathsf{pp} = (\mathcal{R}_{\mathsf{base}}, \mathsf{pp}_{\mathsf{sig}}, \mathsf{pp}_{\mathsf{comm}}, \vec{t})$
5: **return** pp

$\mathcal{R}_{\mathsf{base}}$

Given $(t_{j-1}, t_j, \mathsf{pk}_i, \mathsf{cm}^{i,j}_{\rho_c}, \rho^{i,j}_s, \tilde{x}^{i,j})$, the prover knows $(t^{i,j}_x, x_{i,j}, \sigma^{i,j}_x, \rho^{i,j}_c, r^{i,j}_{\rho_c})$ s.t.:
1: $t^{i,j}_x \in (t_{j-1}, t_j]$
2: $\mathsf{Sig.Verify}_{\mathsf{pk}_i}(\sigma^{i,j}_x, x_{i,j} || t^{i,j}_x) = 1$
3: $\mathsf{cm}^{i,j}_{\rho_c} = \mathsf{Comm}(\rho^{i,j}_c; r^{i,j}_{\rho_c})$
4: $\rho_{i,j} = \rho^{i,j}_c \oplus \rho^{i,j}_s$
5: $\tilde{x}_{i,j} = \mathsf{LDP.Apply}(x_{i,j}; \rho_{i,j})$

$\mathsf{GenRand}_{\mathsf{base}}(\mathsf{pp}, t_j)$ – **Client** i

1: $k^{i,j}_c \leftarrow\!\!\$\ \{0,1\}^*$
2: $\rho^{i,j}_c = \mathsf{PRF}(k^{i,j}_c, 0)$
3: $r^{i,j}_{\rho_c} \leftarrow\!\!\$\ \{0,1\}^*$
4: $\mathsf{cm}^{i,j}_{\rho_c} = \mathsf{Comm}(\rho^{i,j}_c; r^{i,j}_{\rho_c})$
5: Send $(\mathsf{pk}_i, \mathsf{cm}^{i,j}_{\rho_c}, t_j)$ to server
6: Receive $(k^{i,j}_s, \sigma^{i,j}_s)$ from server
7: If $\mathsf{Sig.Verify}_{\mathsf{pk}_s}(\sigma^{i,j}_s, \mathsf{pk}_i || \mathsf{cm}^{i,j}_{k_c} || k^{i,j}_s || t_j) \neq 1$, abort
8: **return** $\mathsf{out}^{i,j}_c = (\rho^{i,j}_c, r^{i,j}_{\rho_c}, \mathsf{cm}^{i,j}_{\rho_c}, k^{i,j}_s, \sigma^{i,j}_s)$

$\mathsf{GenRand}_{\mathsf{base}}(\mathsf{pp}, t_j)$ – **Server**

1: Receive $(\mathsf{pk}_i, \mathsf{cm}^{i,j}_{\rho_c}, t_j)$ from client i
2: If pk_i does not belong to i, abort
3: If $(i, t_j) \in L$, abort
4: $L \leftarrow L \cup \{i, t_j\}$
5: $k_s \leftarrow\!\!\$\ \{0,1\}^*$
6: Send $(k^{i,j}_s, \sigma^{i,j}_s)$ to client i

$\mathsf{Randomize}_{\mathsf{b.}}(\mathsf{pp}, \mathsf{ek}, t_j, \mathsf{out}^{i,j}_c, x_{i,j}, t^{i,j}_x, \sigma^{i,j}_x)$

1: $\rho^{i,j}_s = \mathsf{PRF}(k^{i,j}_s, 0)$
2: $\rho_{i,j} = \rho^{i,j}_c \oplus \rho^{i,j}_s$
3: $\tilde{x}_{i,j} = \mathsf{LDP.Apply}(x_{i,j}; \rho_{i,j})$
4: $\vec{\phi}_{i,j} = (t_{j-1}, t_j, \mathsf{pk}_i, \mathsf{cm}^{i,j}_{\rho_c}, \rho^{i,j}_s, \tilde{x}_{i,j})$
5: $\vec{w}_{i,j} = (t^{i,j}_x, x_{i,j}, \sigma^{i,j}_x, \rho^{i,j}_c, r^{i,j}_{\rho_c})$
6: $\pi_{i,j} = \mathsf{NIZK\text{-}PK.Prove}_{\mathsf{ek}}(\vec{\phi}_{i,j}; \vec{w}_{i,j})$
7: $\tau_{i,j} = (\mathsf{pk}_i, \mathsf{cm}^{i,j}_{\rho_c}, k^{i,j}_s, \sigma^{i,j}_s)$
8: Send $(\tilde{x}_{i,j}, \tau_{i,j})$ to server

$\mathsf{Verify}_{\mathsf{base}}(\mathsf{pp}, \mathsf{vk}, t_j, \tilde{x}_{i,j}, \pi_{i,j}, \tau_{i,j})$

1: Parse $\tau_{i,j} = (\mathsf{pk}_i, \mathsf{cm}^{i,j}_{\rho_c}, k^{i,j}_s, \sigma^{i,j}_s)$
2: If pk_i does not belong to i, abort
3: If $\mathsf{Sig.Verify}_{\mathsf{pk}_s}(\sigma^{i,j}_s, \mathsf{pk}_i || \mathsf{cm}^{i,j}_{k_c} || k^{i,j}_s || t_j) \neq 1$, abort
4: $\rho^{i,j}_s = \mathsf{PRF}(k^{i,j}_s, 0)$
5: $\vec{\phi}_{i,j} = (t_{j-1}, t_j, \mathsf{pk}_i, \mathsf{cm}^{i,j}_{\rho_c}, \rho^{i,j}_s, \tilde{x}_{i,j})$
6: If $\mathsf{NIZK\text{-}PK.Vfy}_{\mathsf{vk}}(\pi_{i,j}, \vec{\phi}_{i,j}) \neq 1$, abort
7: **return** $\tilde{x}_{i,j}$

Figure 4: Base **scheme: VLDP with one server and n clients.**

parts to obtain a true random value $\rho_{i,j} = \rho_c^{i,j} \oplus \rho_s^{i,j}$. Subsequently, the client uses $\rho_{i,j}$ to transform $x_{i,j}$ into a differentially private value $\tilde{x}_{i,j}$ using LDP.Apply(). Finally, the client computes the NIZK-PK for $\mathcal{R}_{\text{base}}$ to attest to a number of statements: (1) the true value $x_{i,j}$ was signed using pk_i and obtained at a time $t_x^{i,j}$, such that $t_x^{i,j} \in (t_{j-1}, t_j]$; (2) $\rho_{i,j}$ is a true random value, i.e., $\text{cm}_{\rho_c}^{i,j} = \text{Comm}(\rho_c^{i,j}; r_{\rho_c}^{i,j})$ and $\rho_{i,j} = \rho_c^{i,j} \oplus \rho_s^{i,j}$; and (3) $\tilde{x}_{i,j}$ is the result of LDP.Apply$(x_{i,j}; \rho_{i,j})$.

When the server receives an LDP value $\tilde{x}^{i,j}$, proof $\pi^{i,j}$ and public values $(\text{pk}_i, \text{cm}_{\rho_c}^{i,j}, k_s^{i,j}, \sigma_s^{i,j})$ from the i-th client, it verifies correctness of $\rho_s^{i,j}$, $\sigma_s^{i,j}$ and $\pi_{i,j}$ for t_j. If all hold, it knows that $\tilde{x}^{i,j}$ is a correct DP version of an authentic input.

6.2 Randomness Expansion (Expand) Scheme

The Base scheme requires one execution of GenRand for each call to Randomize, i.e., one per client, per time interval. Due to the interactive nature of GenRand, this becomes impractical when the number of clients increases. The Expand scheme (Figure 5) uses Merkle trees as compact commitments to multiple random values, to reduce the number of GenRand executions to only one per client. Specifically, we update steps 2–4 of GenRand by creating T commitments to T randomly generated values $\rho_c^{i,j}$, for $j \in [T]$. Subsequently, we encode all these commitments inside a Merkle tree with root rt to keep the message size constant and equal to that of $\text{GenRand}_{\text{base}}$. The main advantage is that we can now generate T random values with only one round of communication, with communication and server-side cost independent of T.

This improvement requires some changes and additional computations for the client in $\text{Randomize}_{\text{expand}}$. Following Section 3, given an array of distinct, public values $\vec{s} = (s_1, \ldots, s_T)$, we can define a secure PRG as $\text{PRG}(k) \coloneqq \text{PRF}(k||s_1)|| \ldots ||\text{PRF}(k||s_T)$. Thus, if we consider the j-th call to $\text{Randomize}_{\text{expand}}$, we can compute the server part of the randomness (line 1) as $\rho_s^{i,j} = \text{PRF}(k_s^i||s_j)$, where k_s^i is the server seed for client i. Observe that the vector s is identical for all clients. The remainder of Randomize follows the same structure as in Base. However, we do have to add an additional statement to our NIZK-PK for $\mathcal{R}_{\text{expand}}$, verifying that the client randomness used in the j-th call of $\text{Randomize}_{\text{expand}}$ is indeed the j-th entry of the Merkle tree with root rt_i. This ensures not only that the i-th client uses a random value that was committed to before seeing k_s^i, but also ensures that i has no choice in which random value in the Merkle tree it uses. Allowing the client to choose which value it uses could make it possible to influence the value of $\tilde{x}^{i,j}$ for at least one j, by cleverly constructing $\vec{\rho}_c$.

6.3 Shuffle Model Scheme

In both the Base and Expand scheme, we consider the regular LDP model. In this model, at time step j the server receives n differentially private values $\tilde{x}_{i,j}$, each of which is directly linkable to the client who sent it. However, in the shuffle model at time step j, the server instead receives a vector $\tilde{\mathbf{x}}_\mathbf{j}$ of n differentially private values, which are not linkable to a particular client. The server at most knows the group of clients that is collectively responsible for sending this vector of differentially private data. For simplicity, we assume that there is a trusted shuffler who first collects all the clients' messages and then sends them in random order to the server. In practice, this may be implemented using, e.g., mixnets (see Appendix B).

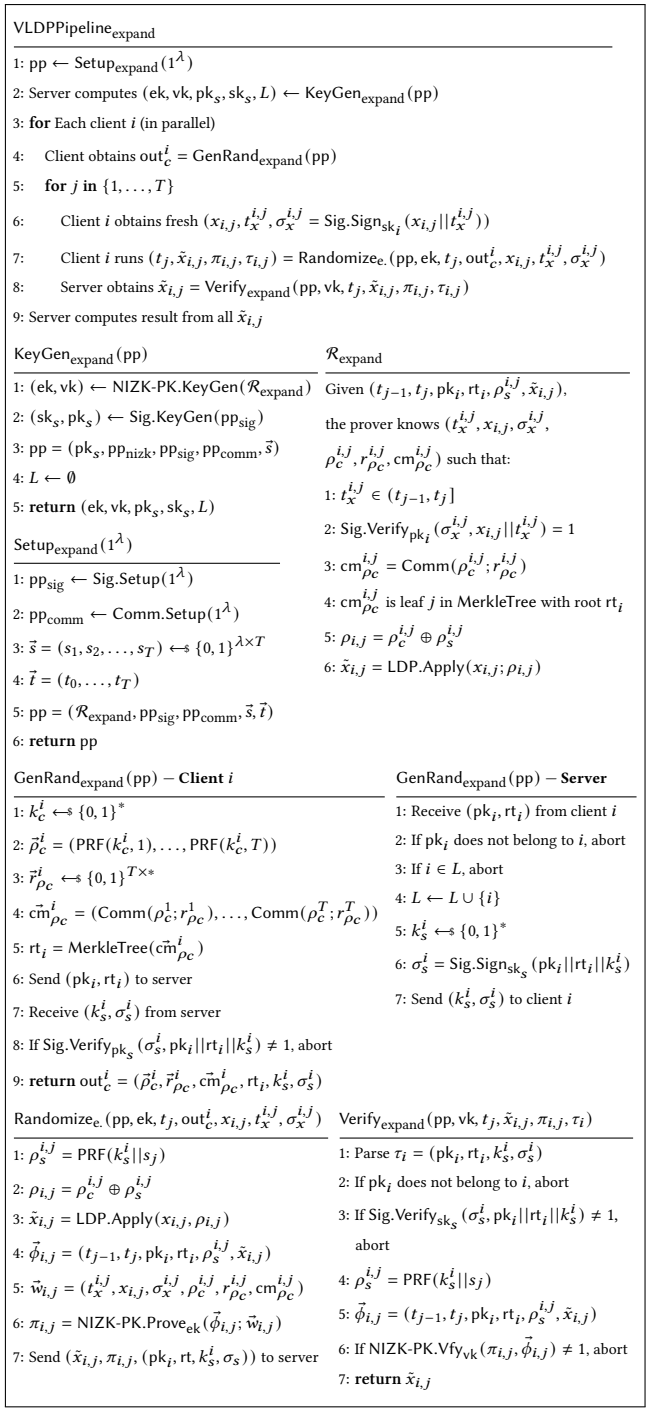

Figure 5: Expand **scheme: only one call to** GenRand **per client.**

In other words, rather than receiving n differentially private values from n identified parties, the server now only receives a differentially private histogram representing the collective response of a (known) group of n clients. Clearly, in this situation, the client could answer more server queries within the same privacy budget, since the budget decreases less quickly (see Section 4). One can determine the influence on the privacy budget decrease for a particular randomizer in the shuffle model following, e.g., [8].

An interesting question to ask is whether either of the previous schemes can be transformed to work in the shuffle model. Whilst we assume that the shuffler properly randomizes all messages and does not provide the server with any other information, the messages themselves might still be linkable to the client who sent them. In fact, we observe that neither the Base nor the Expand scheme can be applied directly in the shuffle model, since the public values pk_i, $\sigma_x^{i,j}$, $\mathsf{cm}_{\rho_c}^{i,j}/\mathsf{rt}_i$, $k_s^{i,j}$, and $\sigma_s^{i,j}/\sigma_s^i$ are the same for different runs of Randomize. This would allow the server to easily link several messages to the same client by simply comparing these public values. Even worse, pk_i is directly linkable to the i-th client.

Fortunately, we can solve this, by moving these values to the witness part of the NIZK-PK statement and include the verification statements on line 3 and 4 into $\mathcal{R}_{\mathsf{shuffle}}$.[3] This transformation clearly guarantees unlinkability of different Randomize messages of the same client. Also, verifiable correctness is still guaranteed, which can be seen intuitively as follows. First, observe that, since pk_i is included in σ_s^i, and we verify σ_s^i inside the NIZK-PK for $\mathcal{R}_{\mathsf{shuffle}}$, the client has to use a pair $(x_{i,j}, t_x^{i,j})$, signed by its own trusted environment. Second, the inclusion of pk_i inside σ_s^i also guarantees that the randomness is bound to a specific client, and thus a set of colluding clients could not interchange their random values.

In summary, this gives us the following high-level protocol execution. At time step j each client sends a message containing a differentially private value $\tilde{x}_{i,j}$ and a proof $\pi_{i,j}$ to the shuffler. Next, the shuffler collects all these messages and forwards them in random order to the server. I.e., the server receives n value-proof pairs. Finally, the server verifies the proof of each value, and accepts the values with a correct proof. We note that for proof verification, the server only requires the corresponding $\tilde{x}$, public parameters pp, it's own public key pk_s and the verification key vk. Since $\tilde{x}$ is the requested value and all other values are identical for all messages, we are certain that we do not compromise the unlinkability that is required in the shuffle model.

However, we do not only want a secure protocol. The above construction still requires careful consideration to keep the client-side performance practical. We note that by moving the verification of $\rho_s^{i,j} = \mathsf{PRF}(k_s^i || s_j)$ to the NIZK-PK, we can remove the Merkle tree. This is done by having both the server and client i generate a random value for the PRF seed, respectively k_s^i and k_c^i. The full PRF seed is defined as $k_i = k_c^i \oplus k_s^i$. Next, we compute a random value $\rho = \mathsf{PRF}(k_i, s_j)$ and verify this inside the NIZK-PK. By doing this, we only require one verification of a PRF, rather than requiring both a PRF verification and verifying the presence of a commitment in a Merkle tree. We observe that we could also have used this construction in our Expand scheme, however the NIZK-PK for practically sized Merkle trees is more efficient than that for a secure PRF evaluation [34]. A precise specification of the Shuffle scheme is given in Figure 6.

[3] The statement on line 2 is implicitly guaranteed by the check in GenRand.

$\mathsf{VLDPPipeline}_{\mathsf{shuffle}}$

1: $\mathsf{pp} \leftarrow \mathsf{Setup}_{\mathsf{shuffle}}(1^\lambda)$
2: Server computes $(\mathsf{ek}, \mathsf{vk}, \mathsf{pk}_s, \mathsf{sk}_s, L) \leftarrow \mathsf{KeyGen}_{\mathsf{shuffle}}(\mathsf{pp})$
3: **for** Each client i (in parallel)
4: Client obtains $\mathsf{out}_c^i = \mathsf{GenRand}_{\mathsf{shuffle}}(\mathsf{pp})$
5: **for** j **in** $\{1, \ldots, T\}$
6: Client i obtains fresh $(x_{i,j}, t_x^{i,j}, \sigma_x^{i,j} = \mathsf{Sig.Sign}_{\mathsf{sk}_i}(x_{i,j} || t_x^{i,j}))$
7: Client i runs $(\tilde{x}_{i,j}, \pi_{i,j}) = \mathsf{Randomize}_{\mathsf{shuffle}}(\mathsf{pp}, \mathsf{ek}, t_j, \mathsf{out}_c^i, x^{i,j}, t_x^{i,j}, \sigma_x^{i,j})$
8: Shuffler forwards messages in random order
9: Server obtains $\tilde{x}_{?i,j} = \mathsf{Verify}_{\mathsf{shuffle}}(\mathsf{pp}, \mathsf{vk}, t_j, \tilde{x}^{?i,j}, \pi_{?i,j})$
10: Server computes result from all $\tilde{x}_{?i,j}$

$\mathsf{KeyGen}_{\mathsf{shuffle}}(\mathsf{pp})$

1: $(\mathsf{ek}, \mathsf{vk}) \leftarrow \mathsf{NIZK\text{-}PK.KeyGen}(\mathcal{R}_{\mathsf{shuffle}})$
2: $(\mathsf{sk}_s, \mathsf{pk}_s) \leftarrow \mathsf{Sig.KeyGen}(\mathsf{pp}_{\mathsf{sig}})$
3: $L \leftarrow \emptyset$
4: **return** $(\mathsf{ek}, \mathsf{vk}, \mathsf{pk}_s, \mathsf{sk}_s, L)$

$\mathsf{Setup}_{\mathsf{shuffle}}(1^\lambda)$

1: $\mathsf{pp}_{\mathsf{sig}} \leftarrow \mathsf{Sig.Setup}(1^\lambda)$
2: $\mathsf{pp}_{\mathsf{comm}} \leftarrow \mathsf{Comm.Setup}(1^\lambda)$
3: $\vec{s} = (s_1, \ldots, s_T) \leftarrow\!\$ \{0,1\}^{\lambda \times T}$
4: $\vec{t} = (t_0, \ldots, t_T)$
5: $\mathsf{pp} = (\mathcal{R}_{\mathsf{shuffle}}, \mathsf{pp}_{\mathsf{sig}}, \mathsf{pp}_{\mathsf{comm}}, \vec{s}, \vec{t})$
6: **return** pp

$\mathcal{R}_{\mathsf{shuffle}}$

Given $(t_{j-1}, t_j, \mathsf{pk}_s, s_j, \tilde{x}_{i,j})$,
the prover knows$(t_x^{i,j}, x^{i,j}, \mathsf{pk}_i, \sigma_x^{i,j}, k_c^i,$
$r_{k_c}^i, \mathsf{cm}_{k_c}^i, k_s^i, \sigma_s^i)$ such that:
1: $t_x^{i,j} \in (t_{j-1}, t_j]$
2: $\mathsf{Sig.Verify}_{\mathsf{pk}_i}(\sigma_x^{i,j}, x_{i,j} || t_x^{i,j}) = 1$
3: $\mathsf{cm}_{k_c}^i = \mathsf{Comm}(k_c^i; r_{k_c}^i)$
4: $k_i = k_c^i \oplus k_s^i$
5: $\mathsf{Sig.Verify}_{\mathsf{pk}_s}(\sigma_s^i, \mathsf{pk}_i || \mathsf{cm}_{k_c}^i || k_s^i) = 1$
6: $\rho_{i,j} = \mathsf{PRF}(k_i, s_j)$
7: $\tilde{x}_{i,j} = \mathsf{LDP.Apply}(x_{i,j}; \rho_{i,j})$

$\mathsf{GenRand}_{\mathsf{shuffle}}(\mathsf{pp})$ – **Client** i

1: $k_c^i \leftarrow\!\$ \{0,1\}^*$
2: $r_{k_c}^i \leftarrow\!\$ \{0,1\}^*$
3: $\mathsf{cm}_{k_c}^i = \mathsf{Comm}(k_c^i; r_{k_c}^i)$
4: Send $(\mathsf{pk}_i, \mathsf{cm}_{k_c}^i)$ to server
5: Receive (k_s^i, σ_s^i) from server
6: If $\mathsf{Sig.Verify}_{\mathsf{pk}_s}(\sigma_s^i, \mathsf{pk}_i || \mathsf{cm}_{k_c} || k_s^i) \neq 1$, abort
7: **return** $\mathsf{out}_c^i = (k_c^i, r_{k_c}^i, \mathsf{cm}_{k_c}^i, k_s^i, \sigma_s^i)$

$\mathsf{GenRand}_{\mathsf{shuffle}}(\mathsf{pp})$ – **Server**

1: Receive $(\mathsf{pk}_i, \mathsf{cm}_{k_c}^i)$ from client i
2: If pk_i does not belong to i, abort
3: If $i \in L$, abort
4: $L \leftarrow L \cup \{i\}$
5: $k_s^i \leftarrow\!\$ \{0,1\}^*$
6: $\sigma_s^i = \mathsf{Sig.Sign}_{\mathsf{sk}_s}(\mathsf{pk}_i || \mathsf{cm}_{k_c}^i || k_s^i)$
7: Send (k_s^i, σ_s^i) to client i

$\mathsf{Randomize}_{\mathsf{shuffle}}(\mathsf{pp}, \mathsf{ek}, t_j, \mathsf{out}_c^i, x_{i,j}, t_x^{i,j}, \sigma_x^{i,j})$

1: $k_i = k_c^i \oplus k_s^i$
2: $\rho_{i,j} = \mathsf{PRF}(k_i, s_j)$
3: $\tilde{x}_{i,j} = \mathsf{LDP.Apply}(x; \rho)$
4: $\vec{\phi}_{i,j} = (t_{j-1}, t_j, \mathsf{pk}_s, s_j, \tilde{x}_{i,j})$
5: $\vec{w}_{i,j} = (t_x^{i,j}, x_{i,j}, \mathsf{pk}_i, \sigma_x^{i,j}, k_c^i, r_{k_c}^i, \mathsf{cm}_{k_c}^i, k_s^i, \sigma_s^i)$
6: $\pi = \mathsf{NIZK\text{-}PK}_{\mathsf{shuffle}}.\mathsf{Prove}(\vec{\phi}_{i,j}; \vec{w}_{i,j})$
7: Send $(\pi_{i,j}, \tilde{x}_{i,j})$ to shuffler

$\mathsf{Verify}_{\mathsf{shuffle}}(\mathsf{pp}, \mathsf{vk}, t_j, \tilde{x}_{i,j}, \pi_{i,j})$

1: $\vec{\phi} = (t_{j-1}, t_j, \mathsf{pk}_s, s_j, \tilde{x}_{i,j})$
2: If $\mathsf{NIZK\text{-}PK.Verify}($
$\pi^i, \tilde{x}_{i,j}, \mathsf{pk}_s, s_j) \neq 1$, abort
3: **return** $\tilde{x}_{i,j}$

Figure 6: Shuffle **scheme: efficient VLDP in the shuffle model.**

7 Experimental Evaluation

To assess the practical performance of our constructions and to compare different versions, we conducted various experiments on synthetic and real data, and report the communication costs and computation times. We first describe our implementation of

the schemes, including how the different building blocks were instantiated. This is followed by a description of our experiments and their results to support our efficiency and practicality claims.

7.1 Implementation

Each scheme was implemented in Rust using the Arkworks v0.4 library [6]. It provides efficient implementations for zk-SNARKs and other cryptographic primitives with gadgets to evaluate these primitives inside a zk-SNARK circuit. Our cryptographic building blocks are instantiated as follows, targeting 128-bit security:

- NIZK-PK: The *Groth16* zk-SNARK [33] is used to generate the NIZK-PKs. This specific pairing-based, circuit zk-SNARK scheme has gained widespread adoption in real-world applications due to its efficiency and constant proof size. This scheme does rely on a trusted setup, which, if broken, would allow anyone to create false proofs. However, this is not an issue in our constructions, since the server can execute this trusted setup by itself. Furthermore, the server is assumed to behave semi-honestly and to be non-colluding. The zk-SNARK elements are chosen to be on the *BLS12-381* elliptic curve (EC) [16], which is a known pairing-friendly curve with good (estimated 128-bit) security. Moreover, there is a known embedded curve for BLS12-381, called *Jubjub* [17], which allows for efficient and secure evaluation of EC-primitives inside zk-SNARK circuits.
- Sig: The signature schemes used by client and server are both implemented using Schnorr signatures [49]. Specifically, we use EC-Schnorr signatures over the Jubjub curve, due to its efficient verification inside a zk-SNARK circuit [52]. Moreover, this scheme is often used in practice, and other popular schemes, such as EdDSA (~1,000 more constraints) and ECDSA (~10,000 more) would only have a small to negligible performance impact [52]. Additionally, we use the Blake2s-256 collision resistant hash (CRH) [47] to hash the input message to a fixed length digest. This CRH was chosen for its good security (128 bits against collision attacks), and efficiency inside a zk-SNARK.
- Comm: Our commitment scheme is instantiated using Pedersen vector commitments [46] (with 4-bit windows) over the Jubjub curve. This instantiation is very efficient inside a zk-SNARK circuit, is information-theoretically hiding, and targets the required bit security for the binding property.
- PRF: We construct a PRF using keyed Blake2s-256 [47], which gives a PRF output of 256 bits, or 32 bytes. Also here, Blake2s was chosen to fit the targeted security level, whilst still being practical inside a zk-SNARK circuit.
- MerkleTree: This primitive is only used inside the Expand scheme. By using Pedersen commitments to instantiate Comm, we can use these commitments directly as the leaves of the Merkle tree due to their fixed size (which is no more than 256 bits in our case). To compute the higher level nodes and root, we use the Pedersen hash function [34] to hash the concatenation of both its children. We use a Pedersen hash rather than Blake2s here, since it is significantly more efficient inside a zk-SNARK circuit, and has security guarantees similar to that of Groth16, thereby not decreasing the security of our scheme. Finally, we note that the tree depth d has to be the smallest power of 2 such that $2^{d-1} \geq T$, where T is the total number of time steps we wish to run.

Our open-source code[4] was written in such a way that it is simple to replace a specific building block by another. In Appendix D.2, we discuss alternative building block choices and their impact on security, efficiency, and practicality.

7.2 Experimental Setup

We perform two sets of experiments to evaluate and compare the practical performance of our constructions. The first set uses two real datasets to evaluate and validate the performance of each scheme in a real-world setting. The second set of experiments uses synthetic data to evaluate the scalability of our schemes.

We use two datasets in our experiments, and consider $T = 5$ days of readings from each. The first dataset (*Geolife GPS Trajectory*) is a location dataset of 182 users, which after pre-processing, gave us 8 potential postcode locations per day for the subject group. With respect to the algorithm for histograms (see Figure 1), we thus have $k = 8$, where $\{1, \ldots, k\}$ represent the respective postcodes.

The second dataset (*Smart meter*) contains smart meter energy readings (floating points) of 5,567 households. We use a precision level of $k = 10$ (see the algorithm for reals in Figure 1) for our experiments. Both datasets are described in more detail in Appendix D.1.

Experiments. For our experiments, we determine the median runtime of 100 runs (after discarding three warm-up runs), of each of the algorithms at the client and server side. Specifically, we look at the computation time for individual clients and the server in the different phases. Next to this, we also measure the byte size of all (compressed) messages. The experiments were run on a desktop computer with Windows 10 desktop PC with a Ryzen 3600 CPU with 6 cores and 12 threads @4.0GHz and 16GB dual-channel DDR4 RAM at 3600MHz. The experiments were run using Rust 1.77.2.

7.3 Concrete Applications

For both datasets, the timestamp is encoded using one byte. Next to this, we use 8 bytes (64 bits) for each random value we sample. This guarantees statistical closeness to the true distributions, since $k \ll 2^{64}$. Thus, for the Geolife GPS dataset, i.e., histogram, we require 16 bytes of randomness ($|\rho| = 16$). For the smart meter dataset, i.e., real valued data, we require 24 bytes of randomness ($|\rho| = 24$), $2 \cdot 8$ bytes for both Bernoulli samples and another 8 bytes for one uniformly random sample. Both are below the 32 bytes that we get as output from one PRF evaluation. In Section 7.4, we evaluate the computation times and message sizes for larger values of $|\rho|$. The performance of our schemes is not impacted by any particular value of ϵ and δ used in the DP mechanism. For completeness, we shall use 5 runs of the LDP mechanism, with the privacy budget per run, i.e., ϵ_0, determined as in Section 4. Finally, for the Expand scheme we set the Merkle tree depth to $d_{\mathsf{MT}} = 4$, i.e., it has $2^{4-1} = 8$ leaves, since we run both datasets for $T = 5$ steps.

Results. The median computation times and message sizes for a single run of each algorithm are shown in Table 1. This table also includes the size of the NIZK-PK evaluation/verification key, and the number of constraints. The evaluation key is relatively large, and needs to be communicated with each client. Fortunately, its

[4]The source code of our implementation is available at https://github.com/xQiratNL/VLDP.

Dataset	Scheme	Client			Server		Communication			NIZK-PK		
		GenRand-1	GenRand-2	Randomize	GenRand	Verify	GenRand-1	GenRand-2	Randomize	|ek|	|vk|	# constraints
Geolife GPS	Base	0.218 ms	0.302 ms	0.610 s	2.494 ms	3.454 ms	65 B	96 B	360 B	16.1 MB	776 B	55 884
	Expand	3.033 ms	0.267 ms	1.213 s	2.005 ms	4.425 ms	64 B	96 B	360 B	23.4 MB	824 B	74 322
	Shuffle	0.230 ms	0.305 ms	1.798 s	2.413 ms	2.680 ms	64 B	96 B	200 B	53.2 MB	728 B	173 460
Smart meter	Base	0.223 ms	0.213 ms	0.619 s	2.146 ms	3.484 ms	65 B	96 B	360 B	16.3 MB	776 B	56 903
	Expand	2.475 ms	0.211 ms	1.106 s	1.271 ms	3.540 ms	64 B	96 B	360 B	23.7 MB	824 B	75 341
	Shuffle	0.225 ms	0.293 ms	1.821 s	2.119 ms	2.659 ms	64 B	96 B	200 B	53.3 MB	728 B	174 095

Table 1: Performance of all schemes for two real-world applications: client and server computation and communication costs of a single evaluation of an algorithm/protocol, byte size of ek and vk, and number of constraints in the NIZK-PK.

Dataset	Scheme	Client			Server	
		GenRand-1	GenRand-2	Rand.	GenRand	Verify
Geolife GPS	Base	1.092 ms	1.508 ms	3.032 s	2.392 s	3.316 s
	Expand	3.033 ms	0.267 ms	6.045 s	0.385 s	4.425 s
	Shuffle	0.230 ms	0.305 ms	8.973 s	0.463 s	2.572 s
Smart meter	Base	1.113 ms	1.064 ms	3.078 s	59.734 s	96.977 s
	Expand	2.475 ms	0.211 ms	5.510 s	7.076 s	98.536 s
	Shuffle	0.225 ms	0.293 ms	9.090 s	11.796 s	74.999 s

Table 2: Total computation time for both use cases, over all time steps ($T = 5$) (for the server also over all clients).

generation is part of the setup and can be communicated as part of the public parameters beforehand. The number of constraints gives an implementation-independent view on the proof generation and verification costs and is the most fair way to compare different schemes. Especially, since proof generation and verification are the dominating factors in Randomize and Verify. To better understand the cost in both use cases, we report the overall computation time for the server and of a single client in Table 2.

Regarding the communication costs of Base, we see that each client sends $(65+360)T = 2{,}125$ bytes (B) to the server, and receives $96T = 480$B. In the Expand scheme, this reduces to $64+360T = 1{,}864$ sent and 96 received bytes. For the Shuffle scheme, the amount of bytes sent by each client reduces further to only $64+200T = 1{,}064$B.

Moreover, we observe that the Base scheme puts a much higher load on the server, in both computation and communication[5] costs, in the GenRand phase. This is due to the fact that this phase needs to be run again for each time step. Expand requires slightly more computational effort from each client, however, this is negligible when compared to the reduction in server computation time, and overall communication costs. The Shuffle scheme requires the least effort in this phase, but puts clearly higher cost on the client in Verify. It should be noted, however, that the computational cost for the client is very practical and lies in the 0.5–2 seconds range for all schemes. Moreover, the computation and communication costs of Verify are significantly lower for Shuffle, which makes it more attractive even in the 'regular' local model. We remark that we did not implement server-side parallelization. Hence, the server's runtime (Table 2) could be further reduced by, e.g., distributing the client messages over different processes.

[5]Since GenRand has to be run once per time step, instead of once overall, Base has $T\times$ more communication than other schemes for GenRand.

Additionally, we note that the shuffle model does introduce some additional latency, when compared to the 'regular' local model. The amount of latency depends on how the shuffler is implemented (see Appendix B). But, given that the message size is small, we expect this to be of little influence in most applications. The introduced latency will be in the same order as in the shuffle model without verifiability, i.e., our introduction of verifiability does not in itself restrict the usage of a shuffler.

7.4 General Performance & Comparison

Above, we evaluated our schemes on common LDP algorithms and datasets, thereby reflecting the expected application settings for our protocols. As discussed, the amount of randomness required ($|\rho|$) for LDP.Apply() determines the majority of the computation cost, due to the cost of evaluating PRF. Here, $|\rho|$ depends on two factors:

- *The randomizer:* The amount of random variables used, and the number of bits to (accurately) sample them, predominantly determines the size of $|\rho|$. In other words, more random values or sampling with higher accuracy leads to an increase in $|\rho|$.
- *Entropy of released data:* The data that is released by the client also has an effect on $|\rho|$, albeit indirectly. Namely, when releasing data with higher entropy (more records, larger domain size) more randomness is needed to ensure differential privacy. For example, privately releasing a bit requires less randomness than releasing an 8-bit integer. Similarly, releasing 10 values requires more randomness than only 1. Thus, when considering higher-dimensional datasets one often also releases data with higher entropy and thus indirectly requires more randomness. Finally, we note that using higher-dimensional input data could lead to reduced performance due to increasing the signature input size. Fortunately, this can be counteracted by using SNARK-friendly signatures (see Appendix D.2).

To better investigate the performance impact of the amount of randomness used, we vary $|\rho|$ in steps of 32, which is the output size of our choice of PRF, i.e., smaller step sizes will show negligible differences in performance. This gives insight into the performance of other LDP mechanisms as discussed in Section 4.1. Next to this, we investigate the performance of the Expand scheme for different Merkle tree depths. We vary d_{MT} between 2 and 11, i.e., from 2 to 1,024 leaves, which should be more than sufficient in realistic settings. In all experiments, we use randomly generated data, and encode the timestamps and input values as 64-bit values.

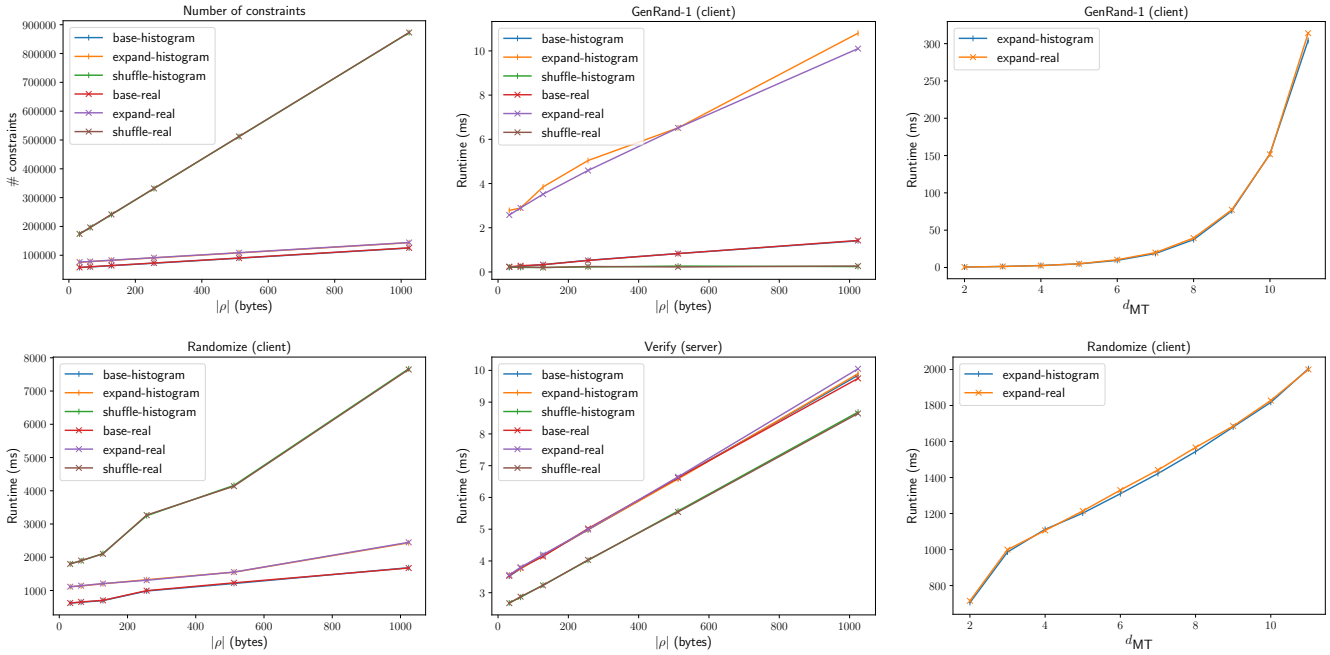

Figure 7: Impact of $|\rho|$ on # constraints (topleft), client runtime in GenerateRandomness **(topcenter) &** Randomize **(bottomcenter), and server runtime in** Verify **(bottomleft); Impact of d_{MT} on client runtime in** GenRand **(topright) and** Randomize **(bottomright).**

Results. First, we observe that the communication size is independent of both $|\rho|$ and d_{MT}, and thus only look at their influence on the runtime (see Figure 7). An increase of $|\rho|$ leads to a significant increase in the constraint count and duration of Randomize for the Shuffle scheme. However, even for as much as 1,024 bytes of randomness, the computation time remains below 8 seconds, which is still very practical. As shown in Section 4.1, 1,024 is more than sufficient for 64-bit precision in many representative and state-of-the-art LDP algorithms. Additionally, we observe a significant, but approximately linear, increase in computation time for the client in GenRand. Since the duration is in the millisecond range, this will not cause any practical issues. Finally, we see a linear increase in the verification time for the server. However, this verification time is so small that we do not consider it to be an issue. For the Expand scheme, we observe a linear increase of the number of constraints (from $\sim$60, 000 to $\sim$120, 000) and an exponential increase of the duration of GenRand in d_{MT} (Figure 7). This agrees with the fact that the amount of random values grows exponentially in d_{MT}. The increase in runtime for Randomize is also approximately linear, and only takes around 2 seconds for a Merkle tree with 1,024 random values.

Comparison. In conclusion, we see that the total runtime of each scheme scales approximately linearly in the amount of randomness required, for both client and server, and that the runtime of each schemes is very practical for realistically sized parameters. Clearly, the Shuffle scheme has the lowest communication cost and server load, in addition to being secure in the shuffle model. Conversely, Expand puts a smaller load on the client, and slightly higher on the server, but is not secure in the shuffle model. Finally, the Base scheme puts a comparatively high communication and computation load on the server, making it less practical than the other schemes.

For comparison with related work, we consider [43] which is the work closest to our construction. As mentioned, the number of constraints provides a fair comparison for different schemes. Their scheme requires two NIZK-PK proofs for one transfer of LDP values. For one input their proofs have 9,769 and 12,882 constraints, i.e., the combined number of constraints is around 2.5—7.5 times smaller than our scheme, which means the computational effort for both client and server will also be smaller by a similar factor. However, the underlying blockchain structure used in [43] will also come with its own latency and scalability issues, which our scheme does not suffer from. On top of that, it is only evaluated for binary RR, which is much simpler than our construction. Finally, [43] does not discuss the performance of their approach to GenRand. However, as it is similar in nature to $\text{GenRand}_{\text{base}}$, it will suffer from the same drawbacks when compared to Expand and Shuffle.

8 Conclusion

We showed how to construct verifiable LDP schemes for both the local and, most interestingly, the shuffle model, which guarantee security against data manipulation attacks. Experimental evaluation of our schemes on realistic use cases underscores their practicality. Especially the Expand and Shuffle schemes put a very low load (5–7 ms) on the server, whilst keeping client computation times down to < 2 seconds. Moreover, we showed the scalability of our schemes using generic benchmarks. Finally, we discuss how our schemes can be efficiently adopt a wide variety of LDP algorithms, due to their generic design.

Acknowledgments

We thank the anonymous reviewers for their insightful feedback, which helped improve our work. This research received no specific grant from any funding agency in the public, commercial, or not-for-profit sectors.

References

[1] Martin Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. 2016. Deep Learning with Differential Privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. Association for Computing Machinery, New York, NY, USA, 308–318. https://doi.org/10.1145/2976749.2978318

[2] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. 2016. MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity. In *Advances in Cryptology – ASIACRYPT 2016*, Jung Hee Cheon and Tsuyoshi Takagi (Eds.). Springer, Berlin, Heidelberg, 191–219. https://doi.org/10.1007/978-3-662-53887-6_7

[3] Andris Ambainis, Markus Jakobsson, and Helger Lipmaa. 2004. Cryptographic Randomized Response Techniques. In *Public Key Cryptography – PKC 2004 (Lecture Notes in Computer Science)*, Feng Bao, Robert Deng, and Jianying Zhou (Eds.). Springer, Berlin, Heidelberg, 425–438. https://doi.org/10.1007/978-3-540-24632-9_31

[4] Apple. 2013. Kernel Architecture Overview. https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/KernelProgramming/Architecture/Architecture.html

[5] Apple. 2024. Secure Enclave. https://support.apple.com/en-au/guide/security/sec59b0b31ff/web

[6] arkworks contributors. 2022. *arkworks zkSNARK ecosystem*. arkworks. https://arkworks.rs

[7] Michael Backes, Manuel Barbosa, Dario Fiore, and Raphael M. Reischuk. 2014. ADSNARK: Nearly Practical and Privacy-Preserving Proofs on Authenticated Data. https://eprint.iacr.org/2014/617

[8] Borja Balle, James Bell, Adrià Gascón, and Kobbi Nissim. 2019. The Privacy Blanket of the Shuffle Model. In *Advances in Cryptology – CRYPTO 2019*, Alexandra Boldyreva and Daniele Micciancio (Eds.). Springer International Publishing, Cham, 638–667. https://doi.org/10.1007/978-3-030-26951-7_22

[9] Zoë Ruha Bell, Shafi Goldwasser, Michael P. Kim, and Jean-Luc Watson. 2024. Certifying Private Probabilistic Mechanisms. In *Advances in Cryptology – CRYPTO 2024*, Leonid Reyzin and Douglas Stebila (Eds.). Springer Nature Switzerland, Cham, 348–386. https://doi.org/10.1007/978-3-031-68391-6_11

[10] Ari Biswas and Graham Cormode. 2023. Verifiable Differential Privacy. https://doi.org/10.48550/arXiv.2208.09011 arXiv:2208.09011 [cs]

[11] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. 2012. From Extractable Collision Resistance to Succinct Non-Interactive Arguments of Knowledge, and Back Again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference (ITCS '12)*. Association for Computing Machinery, New York, NY, USA, 326–349. https://doi.org/10.1145/2090236.2090263

[12] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnes, and Bernhard Seefeld. 2017. Prochlo: Strong Privacy for Analytics in the Crowd. In *Proceedings of the 26th Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, USA, 441–459. https://doi.org/10.1145/3132747.3132769

[13] Manuel Blum, Paul Feldman, and Silvio Micali. 1988. Non-Interactive Zero-Knowledge and Its Applications. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing (STOC '88)*. Association for Computing Machinery, New York, NY, USA, 103–112. https://doi.org/10.1145/62212.62222

[14] Dan Boneh and Victor Shoup. 2023. A Graduate Course in Applied Cryptography v0.6. (Jan. 2023). http://toc.cryptobook.us/ Book.

[15] Tariq Bontekoe, Dimka Karastoyanova, and Fatih Turkmen. 2024. Verifiable Privacy-Preserving Computing. https://doi.org/10.48550/arXiv.2309.08248 arXiv:2309.08248 [cs]

[16] Sean Bowe. 2017. BLS12-381: New Zk-SNARK Elliptic Curve Construction. https://electriccoin.co/blog/new-snark-curve/

[17] Sean Bowe. 2024. Zkcrypto/Jubjub. Zero-knowledge Cryptography in Rust. https://github.com/zkcrypto/jubjub

[18] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. 2020. Transparent SNARKs from DARK Compilers. In *Advances in Cryptology – EUROCRYPT 2020 (Lecture Notes in Computer Science)*, Anne Canteaut and Yuval Ishai (Eds.). Springer International Publishing, Cham, 677–706. https://doi.org/10.1007/978-3-030-45721-1_24

[19] Xiaoyu Cao, Jinyuan Jia, and Neil Zhenqiang Gong. 2021. Data Poisoning Attacks to Local Differential Privacy Protocols. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Virtual, 947–964. https://www.usenix.org/conference/usenixsecurity21/presentation/cao-xiaoyu

[20] David L. Chaum. 1981. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Commun. ACM* 24, 2 (Feb. 1981), 84–90. https://doi.org/10.1145/358549.358563

[21] Albert Cheu, Adam Smith, and Jonathan Ullman. 2021. Manipulation Attacks in Local Differential Privacy. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, San Francisco, CA, USA, 883–900. https://doi.org/10.1109/SP40001.2021.00001

[22] Albert Cheu, Adam Smith, Jonathan Ullman, David Zeber, and Maxim Zhilyaev. 2019. Distributed Differential Privacy via Shuffling. In *Advances in Cryptology – EUROCRYPT 2019*, Yuval Ishai and Vincent Rijmen (Eds.). Springer International Publishing, Cham, 375–403. https://doi.org/10.1007/978-3-030-17653-2_13

[23] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. 2020. Marlin: Preprocessing zkSNARKs with Universal and Updatable SRS. In *Advances in Cryptology – EUROCRYPT 2020 (Lecture Notes in Computer Science)*, Anne Canteaut and Yuval Ishai (Eds.). Springer International Publishing, Cham, 738–768. https://doi.org/10.1007/978-3-030-45721-1_26

[24] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. 2020. Fractal: Post-quantum and Transparent Recursive Proofs from Holography. In *Advances in Cryptology – EUROCRYPT 2020 (Lecture Notes in Computer Science)*, Anne Canteaut and Yuval Ishai (Eds.). Springer International Publishing, Cham, 769–793. https://doi.org/10.1007/978-3-030-45721-1_27

[25] Alfredo De Santis, Giovanni Di Crescenzo, Rafail Ostrovsky, Giuseppe Persiano, and Amit Sahai. 2001. Robust Non-interactive Zero Knowledge. In *Advances in Cryptology — CRYPTO 2001*, Joe Kilian (Ed.). Springer, Berlin, Heidelberg, 566–598. https://doi.org/10.1007/3-540-44647-8_33

[26] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. 2006. 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)*. Springer-Verlag, Berlin, Heidelberg, 486–503. https://doi.org/10.1007/11761679_29

[27] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating Noise to Sensitivity in Private Data Analysis. In *Theory of Cryptography*, Shai Halevi and Tal Rabin (Eds.). Springer, Berlin, Heidelberg, 265–284. https://doi.org/10.1007/11681878_14

[28] Cynthia Dwork, Guy N. Rothblum, and Salil Vadhan. 2010. Boosting and Differential Privacy. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*. IEEE, Las Vegas, NV, USA, 51–60. https://doi.org/10.1109/FOCS.2010.12

[29] Maria Eichlseder, Lorenzo Grassi, Reinhard Lüftenegger, Morten Øygarden, Christian Rechberger, Markus Schofnegger, and Qingju Wang. 2020. An Algebraic Attack on Ciphers with Low-Degree Round Functions: Application to Full MiMC. In *Advances in Cryptology – ASIACRYPT 2020*, Shiho Moriai and Huaxiong Wang (Eds.). Springer International Publishing, Cham, 477–506. https://doi.org/10.1007/978-3-030-64837-4_16

[30] Úlfar Erlingsson, Vitaly Feldman, Ilya Mironov, Ananth Raghunathan, Kunal Talwar, and Abhradeep Thakurta. 2019. Amplification by Shuffling: From Local to Central Differential Privacy via Anonymity. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '19)*. Society for Industrial and Applied Mathematics, USA, 2468–2479.

[31] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. 2014. RAPPOR: Randomized Aggregatable Privacy-Preserving Ordinal Response. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. Association for Computing Machinery, New York, NY, USA, 1054–1067. https://doi.org/10.1145/2660267.2660348

[32] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. 2021. Poseidon: A New Hash Function for Zero-Knowledge Proof Systems. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Virtual, 519–535.

[33] Jens Groth. 2016. On the Size of Pairing-Based Non-interactive Arguments. In *Advances in Cryptology – EUROCRYPT 2016 (Lecture Notes in Computer Science)*, Marc Fischlin and Jean-Sébastien Coron (Eds.). Springer, Berlin, Heidelberg, 305–326. https://doi.org/10.1007/978-3-662-49896-5_11

[34] Daira Emma Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. 2023. Zcash Protocol Specification, Version 2023.4.0 [NU5]. (2023). https://zips.z.cash/protocol/protocol.pdf Protocol specification.

[35] Peter Kairouz, Keith Bonawitz, and Daniel Ramage. 2016. Discrete Distribution Estimation under Local Privacy. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48 (ICML'16)*. JMLR.org, New York, NY, USA, 2436–2444.

[36] Shiva Prasad Kasiviswanathan, Homin K Lee, Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. 2011. What can we learn privately? *SIAM J. Comput.* 40, 3 (2011), 793–826.

[37] Fumiyuki Kato, Yang Cao, and Masatoshi Yoshikawa. 2021. Preventing Manipulation Attack in Local Differential Privacy Using Verifiable Randomization Mechanism. In *Data and Applications Security and Privacy XXXV (Lecture Notes in Computer Science)*, Ken Barker and Kambiz Ghazinour (Eds.). Springer International Publishing, Cham, 43–60. https://doi.org/10.1007/978-3-030-81242-3_3

[38] Hiroaki Kikuchi, Jin Akiyama, Gisaku Nakamura, and Howard Gobioff. 1999. Stochastic Voting Protocol To Protect Voters Privacy. In *Proceedings of the 1999 IEEE Workshop on Internet Applications (WIAPP '99)*. IEEE Computer Society, USA, 103.

[39] Xiaoguang Li, Ninghui Li, Wenhai Sun, Neil Zhenqiang Gong, and Hui Li. 2023. Fine-Grained Poisoning Attack to Local Differential Privacy Protocols for Mean and Variance Estimation. In *Proceedings of the 32nd USENIX Conference on Security Symposium*. USENIX Association, Anaheim, CA, USA, 1739–1756.

[40] Mbed. 2018. NAMote72 | Mbed. https://os.mbed.com/platforms/NAMote-72/

[41] Frank McSherry and Kunal Talwar. 2007. Mechanism Design via Differential Privacy. In *48th Annual IEEE Symposium on Foundations of Computer Science*. IEEE Computer Society, Providence, RI, USA, 94–103. https://doi.org/10.1109/FOCS.2007.41

[42] Andrés Molina-Markham, Prashant Shenoy, Kevin Fu, Emmanuel Cecchet, and David Irwin. 2010. Private Memoirs of a Smart Meter. In *Proceedings of the 2nd ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Building (BuildSys '10)*. Association for Computing Machinery, New York, NY, USA, 61–66. https://doi.org/10.1145/1878431.1878446

[43] Danielle Movsowitz Davidow, Yacov Manevich, and Eran Toch. 2023. Privacy-Preserving Transactions with Verifiable Local Differential Privacy. In *5th Conference on Advances in Financial Technologies (AFT 2023)*, Vol. 282. Schloss-Dagstuhl - Leibniz Zentrum für Informatik, Dagstuhl, Germany, 1–23. https://doi.org/10.4230/LIPIcs.AFT.2023.1

[44] Gonzalo Munilla Garrido, Johannes Sedlmeir, and Matthias Babel. 2022. Towards Verifiable Differentially-Private Polling. In *Proceedings of the 17th International Conference on Availability, Reliability and Security (ARES '22)*. Association for Computing Machinery, New York, NY, USA, 1–11. https://doi.org/10.1145/3538969.3538992

[45] Arjun Narayan, Ariel Feldman, Antonis Papadimitriou, and Andreas Haeberlen. 2015. Verifiable Differential Privacy. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, 1–14. https://doi.org/10.1145/2741948.2741978

[46] Torben Pryds Pedersen. 1992. Non-Interactive and Information-Theoretic Secure Verifiable Secret Sharing. In *Advances in Cryptology — CRYPTO '91*, Joan Feigenbaum (Ed.). Springer, Berlin, Heidelberg, 129–140. https://doi.org/10.1007/3-540-46766-1_9

[47] Markku-Juhani O. Saarinen and Jean-Philippe Aumasson. 2015. *The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)*. Request for Comments RFC 7693. Internet Engineering Task Force. https://doi.org/10.17487/RFC7693

[48] Krishna Sampigethaya and Radha Poovendran. 2006. A Survey on Mix Networks and Their Secure Applications. *Proc. IEEE* 94, 12 (Dec. 2006), 2142–2181. https://doi.org/10.1109/JPROC.2006.889687

[49] C. P. Schnorr. 1990. Efficient Identification and Signatures for Smart Cards. In *Advances in Cryptology — CRYPTO' 89 Proceedings*, Gilles Brassard (Ed.). Springer, New York, NY, 239–252. https://doi.org/10.1007/0-387-34805-0_22

[50] Ali Shahin Shamsabadi, Gefei Tan, Tudor Ioan Cebere, Aurélien Bellet, Hamed Haddadi, Nicolas Papernot, Xiao Wang, and Adrian Weller. 2024. Confidential-DPproof: Confidential Proof of Differentially Private Training. In *ICLR 2024 - 12th International Conference on Learning Representations*. HAL, Vienna, Austria, 1–16. https://hal.science/hal-04610635

[51] Shaorui Song, Lei Xu, and Liehuang Zhu. 2023. Efficient Defenses Against Output Poisoning Attacks on Local Differential Privacy. *IEEE Transactions on Information Forensics and Security* 18 (2023), 5506–5521. https://doi.org/10.1109/TIFS.2023.3305873

[52] Colin Steidtmann and Sanjay Gollapudi. 2023. Benchmarking ZK-Circuits in Circom. https://eprint.iacr.org/2023/681

[53] Georgia Tsaloli and Aikaterini Mitrokotsa. 2023. Differential Privacy Meets Verifiable Computation: Achieving Strong Privacy and Integrity Guarantees. In *16th International Conference on Security and Cryptography*. SciTePress, Prague, Czech Republic, 425–430.

[54] Han Wang, Hanbin Hong, Li Xiong, Zhan Qin, and Yuan Hong. 2022. L-SRR: Local Differential Privacy for Location-Based Services with Staircase Randomized Response. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*. Association for Computing Machinery, New York, NY, USA, 2809–2823. https://doi.org/10.1145/3548606.3560636

[55] Shaowei Wang, Liusheng Huang, Pengzhan Wang, Yiwen Nie, Hongli Xu, Wei Yang, Xiang-Yang Li, and Chunming Qiao. 2016. Mutual Information Optimally Local Private Discrete Distribution Estimation. https://doi.org/10.48550/arXiv.1607.08025 arXiv:1607.08025 [cs, math]

[56] Tianhao Wang, Jeremiah Blocki, Ninghui Li, and Somesh Jha. 2017. Locally Differentially Private Protocols for Frequency Estimation. In *Proceedings of the 26th USENIX Conference on Security Symposium (SEC'17)*. USENIX Association, USA, 729–745.

[57] Stanley L. Warner. 1965. Randomized Response: A Survey Technique for Eliminating Evasive Answer Bias. *J. Amer. Statist. Assoc.* 60, 309 (1965), 63–69. https://doi.org/10.2307/2283137 jstor:2283137

[58] Yongji Wu, Xiaoyu Cao, Jinyuan Jia, and Neil Zhenqiang Gong. 2022. Poisoning Attacks to Local Differential Privacy Protocols for Key-Value Data. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 519–536.

[59] Min Ye and Alexander Barg. 2018. Optimal Schemes for Discrete Distribution Estimation Under Locally Differential Privacy. *IEEE Trans. Inf. Theor.* 64, 8 (Aug. 2018), 5662–5676. https://doi.org/10.1109/TIT.2018.2809790

A Appendix to Section 4

A.1 De-Biased Output

Let X_i denote the random variable representing user i's output after running the LDP algorithm for reals. Let $X = \sum_i^n X_i$. We are interested in finding:

$$\mathbb{E}(X) = \sum_{i=1}^{n} \mathbb{E}(X_i)$$

Let p_j be the probability that user i outputs $j \in \{0, 1, \ldots, k\}$. Let q_j be the true probability of any user having input j. Then,

$$\begin{aligned} p_j &= \left(1 - \gamma + \frac{\gamma}{k+1}\right) q_j + \frac{\gamma}{k+1}(1 - q_j) \\ &= (1-\gamma) q_j + \frac{\gamma}{k+1} \end{aligned}$$

Then

$$\begin{aligned} \mathbb{E}(X_i) &= \sum_{j=0}^{k} j p_j \\ &= \sum_{j=0}^{k} j \left((1-\gamma) q_j + \frac{\gamma}{k+1} \right) \\ &= (1-\gamma) \left(\sum_{j=0}^{k} j q_j \right) + \frac{\gamma k}{2} \\ &= (1-\gamma)\mu + \frac{\gamma k}{2} \end{aligned}$$

where $\mu = \sum_{j=0}^{k} j q_j$ is the true expected input of any user. Thus,

$$\begin{aligned} \mathbb{E}(X) &= n \left((1-\gamma)\mu + \frac{\gamma k}{2} \right) \\ \Rightarrow \frac{n\mu}{k} &= \frac{1}{1-\gamma} \left(\frac{\mathbb{E}(X)}{k} - \frac{\gamma n}{2} \right). \end{aligned}$$

Therefore, the expected value of the sum to precision k output by the LDP algorithm, i.e., $\mathbb{E}(X)/k$, gives us the expectation of the sum of true inputs to precision k, i.e., $n\mu/k$. Thus, given the sum of these values for a sample, we can estimate the true sum as above.

A.2 Additional, Simple Example

To better explain the requirements above, consider the example where we sample a random element from $\{0, 1, 2\}$ using uniform random bits. In practice, an often used method to achieve this is to sample two random bits, and map this as follows $00 \rightarrow 0; 01 \rightarrow 1; 10 \rightarrow 2$. If the random bits are 11 we sample two new random bits and repeat the process, until we terminate.[6] This process terminates (with probability 1), after a finite number steps. However, due the variable requirement of random bits, we cannot use this sampling method inside a NIZK proof. When considering this more closely, it becomes evident that there is no way to sample a random element

[6] While the actual method might vary in practice, this simple version that we present here, is sufficient to describe the problem in the context of NIZK proofs.

from $\{0, 1, 2\}$ using a fixed number of random bits. This problem occurs in many random sampling problems, but can fortunately easily be solved, by sampling from an approximate distribution that is statistically close to the true distribution.

For this example, we can sample an ℓ-bit number ρ, such that 2^ℓ is sufficiently large. Then, we determine 3 intervals: $[0, \lfloor 2^\ell/3 \rfloor)$, $[2^\ell/3, 2 \cdot \lfloor 2^\ell/3 \rfloor)$, and $[2 \cdot \lfloor 2^\ell/3 \rfloor, 2^\ell - 1]$. If ρ is part of the j-th interval, we return j as our random sample. Observe, that all but the last interval have the exact same size of $\lfloor 2^\ell/3 \rfloor$, and only the final interval contains $2^\ell - 3 \cdot \lfloor 2^\ell/3 \rfloor \leq 2$ more elements. Thus, for sufficiently large ℓ, the distribution generated by this sampling method is statistically close to the true distribution we wish to sample from.

A.3 Approximate LDP Randomizers

In Figure 8, we give the precise specification of our approximate LDP randomizers for histograms and reals, based on the algorithms from Section 4.

LDP.Apply$(x;\rho)$ for Reals	LDP.Apply$(x;\rho)$ for Histograms
input: $k \in \mathbb{N}, \gamma \in [0,1]$, $x \in [0,1], \rho \in \{0,1\}^*$	**input:** $k \in \mathbb{N}, \gamma \in [0,1]$, $x \in [k], \rho \in \{0,1\}^*$
Split ρ into (ρ_1, ρ_2, ρ_3)	Split ρ into (ρ_1, ρ_2)
$\overline{x} \leftarrow \lfloor xk \rfloor + \widetilde{\text{Ber}}(xk - \lfloor xk \rfloor; \rho_1)$	$b \leftarrow \widetilde{\text{Ber}}(\gamma; \rho_1)$
$b \leftarrow \widetilde{\text{Ber}}(\gamma; \rho_2)$	**if** $b = 0$ **do**
if $b = 0$ **do**	$\tilde{x} \leftarrow \overline{x}$
$\tilde{x} \leftarrow \overline{x}$	**else**
else	$\tilde{x} \leftarrow \widetilde{\text{Unif}}([1,k]; \rho_2)$
$\tilde{x} \leftarrow \widetilde{\text{Unif}}([0,k]; \rho_3)$	**return** $\tilde{x}$
return $\tilde{x}$	

Figure 8: Approximate randomizers for reals and histograms.

B Implementing the Shuffler

In practice a trusted shuffler can be implemented in a number of ways. One way to do this is by using a *mixnet*, or mix network. A mixnet is a network involving several parties, that takes as input a list of messages and returns the same messages in a randomly permuted order. Mixnets were first introduced in [20] to realize untraceable e-mail and can be implemented in a variety of ways. An overview can be found in, e.g., [48]. In its most basic form, mixnets are implemented using a publicly known sequence of servers, whose public encryption keys are also available. Any client wishing to send a message, encrypts their message in a layered way, i.e., like an onion, using the public keys of the servers in reverse order. This encrypted 'onion' is then sent to the first server, who batches a certain buffer and messages and then forwards this buffer in a random order, stripping one layer of encryption. The following servers in the sequence repeat this process, until the final server sends the inside of the onion, the real message, to the recipient. Implementations of mixnets that produces verifiably random permutations also exist. See for example [12] for an implementation of verifiable, oblivious shuffling using trusted hardware.

In conclusion, following also the discussion in [12], there are three main options for implementing a true, honest-but-curious, non-colluding shuffler. The shuffler could be (1) a single trusted third-party; (2) a group of parties, in which trust is distributed; (3) one or more parties using trusted hardware. The schemes as presented in this work are implementation-agnostic, i.e., they work with any choice of implementation.

C Security Proofs and Experiments

In this section, we provide security proofs for the three protocols, according to our definitions in Section 5. Before we detail the proofs, we provide the explicit experiments in each of the definitions and give some intuition in their construction.

C.1 Experiments

Figures 9 to 12 describe the experiments used in the security definitions of Section 5.3. In all experiments, we explicitly describe the generation of the secret and public keys of each client's trusted environment on the second line of each experiment. In reality, this is a step separate from our system, however, for completeness of the experiment definitions, we explicitly define it here. Below, we give the formal definitions of these experiments, and provide some further intuition regarding their construction.

In the completeness experiment (Figure 9), we verify that the output $\tilde{x}_{i,j}$, with accompanying NIZK-PK proof $\pi_{i,j}$, and public values $\tau_x^{i,j}$, generated by an honest client i for time interval j, is accepted by an honest server, even when an adversary $\mathcal{A}$ chooses the client's inputs $(x_{i,j}, t_x^{i,j})$.[7]

$\textbf{Exp}^{\text{Comp}}_{\mathcal{A}}(1^\lambda, n, T)$

1: $\text{pp} \leftarrow \text{Setup}(1^\lambda)$
2: $\{\text{sk}_i, \text{pk}_i\}_i \leftarrow \text{Sig.KeyGen}(\text{pp})$
3: $(\text{ek}, \text{vk}, \text{pk}_s, \text{sk}_s, L) \leftarrow \text{KeyGen}(\text{pp})$
4: $\text{out}_c^i \leftarrow \text{GenRand}(\text{pp}, \text{aux})$
5: $\{x_{i,j}, t_x^{i,j}\}_{i,j} \leftarrow \mathcal{A}(\text{pp}, \text{ek}, \text{vk}, \text{pk}_s, \text{sk}_s, L, \{\text{pk}_i\}_i)$
6: **if** $\exists i \in [n], j \in [T] : t_x^{i,j} \leq t_{j-1} \vee t_x^{i,j} > t_j$
7: **return** $\{\top\}_{i,j}$
8: $\sigma_x^{i,j} \leftarrow \text{Sig.Sign}_{\text{sk}_i}(x_{i,j} || t_x^{i,j})$
9: $\tilde{x}_{i,j}, \pi_{i,j}, \tau_x^{i,j} \leftarrow \text{Randomize}(\text{pp}, \text{ek}, t_j, \text{out}_c^i, x_{i,j}, t_x^{i,j}, \sigma_x^{i,j})$
10: **return** $\{\text{Verify}(\text{pp}, \text{vk}, t_j, \tilde{x}_{i,j}, \pi_{i,j}, \tau_x^{i,j})\}_{i,j}$

Figure 9: Experiment for completeness definition.

The soundness experiment (Figure 10) guarantees that no malicious, possibly colluding, clients are able to return a value $\tilde{x}_{i,j}$ that is not an honest evaluation of LDP.Apply$(x_{i,j}; \rho_{i,j})$, for a truly random, independently sampled $\rho_{i,j}$, for some i and j. In this experiment, the adversary is allowed to choose $t_x^{i,j}$ and controls all clients, who may deviate from the protocol arbitrarily. The goal of the adversary is to let the server accept a tuple $(\tilde{x}_{i,j}, \pi_{i,j}, \tau_x^{i,j})$, where $\tilde{x}_{i,j}$ is not honestly computed.

[7] Completeness does not guarantee correctness of $\tilde{x}_{i,j}$. Correctness is implicitly guaranteed by soundness, which is defined below.

$\mathbf{Exp}^{\text{Snd-Real}}_{\mathcal{A},S^*}(1^\lambda, n, T, \{x_{i,j}\}_{i,j})$

1: $\text{pp} \leftarrow \text{Setup}(1^\lambda)$
2: $\{\text{sk}_i, \text{pk}_i\}_i \leftarrow \text{Sig.KeyGen}(\text{pp})$
3: $(\text{ek}, \text{vk}, \text{pk}_s, \text{sk}_s, L, \{\text{pk}_i\}_i) \leftarrow \text{KeyGen}(\text{pp})$
4: $\{t_x^{i,j}\}_{i,j} \leftarrow \mathcal{A}(\text{pp}, \text{ek}, \text{vk}, \text{pk}_s, \{x_{i,j}\}_{i,j})$
5: **if** $\exists i \in [n], j \in [T] : t_x^{i,j} \leq t_{j-1} \vee t_x^{i,j} > t_j$
6: **return** $\{\perp\}_{i,j}$
7: $\sigma_x^{i,j} \leftarrow \text{Sig.Sign}_{\text{sk}_i}(x_{i,j} || t_x^{i,j})$
8: $\{(\tilde{x}_{i,j}, \pi_{i,j}, \tau_x^{i,j})\}_{i,j} \leftarrow \mathcal{A}^{S^*}(\text{pp}, \text{ek}, \text{vk}, \text{pk}_s, \{x_{i,j}, t_x^{i,j}, \sigma_x^{i,j}, \text{pk}_i\}_{i,j})$
9: **return** $\{\text{Verify}(\text{pp}, \text{vk}, t_j, \tilde{x}_{i,j}, \pi_{i,j}, \tau_x^{i,j})\}_{i,j}$

Figure 10: Experiment for soundness definition.

The experiments for zero-knowledge (Figure 11), define two different worlds. Zk-real denotes the real world, in which the adversary $\mathcal{A}$ acts as the server, and interacts with honest clients (emulated by the environment). In Zk-sim, $\mathcal{A}$ acts as a server also, but instead interacts with a simulator $\mathcal{S}$. $\mathcal{S}$ simulates an honest client, and should be able to generate messages with the same distribution as an actual client would, but without access to the input values $(x_{i,j}, t_x^{i,j})$. The adversary wins this game, if it can distinguish between both worlds.

$\mathbf{Exp}^{\text{Zk-Real}}_{\mathcal{A}}(1^\lambda, n, T, \{x_{i,j}\}_{i,j})$

1: $\text{pp} \leftarrow \text{Setup}(1^\lambda)$
2: $\{\text{sk}_i, \text{pk}_i\}_i \leftarrow \text{Sig.KeyGen}(\text{pp})$
3: $(\text{ek}, \text{vk}, \text{pk}_s, \text{sk}_s, L, \text{trap}) \leftarrow \mathcal{A}(\text{pp})$
4: $\text{out}_c^i \leftarrow \text{GenRand}^{\mathcal{A}}(\text{pp}, \text{aux})$
5: $\{t_x^{i,j}\}_{i,j} \leftarrow \mathcal{A}(\text{pp}, \text{ek}, \text{vk}, \text{pk}_s, \text{sk}_s, L)$
6: $\sigma_x^{i,j} \leftarrow \text{Sig.Sign}_{\text{sk}_i}(x_{i,j} || t_x^{i,j})$
7: $\{(\tilde{x}_{i,j}, \pi_{i,j}, \tau_x^{i,j})\}_{i,j} \leftarrow \text{Randomize}(\text{pp}, \text{ek}, t_j, \text{out}_c^i, x_{i,j}, \sigma_x^{i,j})$
8: **if** trap is not valid trapdoor for $(\mathcal{R}, \text{ek}, \text{vk})$
9: $\quad \vee \exists i \in [n], j \in [T] : t_x^{i,j} \leq t_{j-1} \vee t_x^{i,j} > t_j$
10: **return** $\perp$
11: $\mathcal{A} \leftarrow \{\tilde{x}_{i,j}, \pi_{i,j}, \tau_{i,j}\}_{i,j}$
12: **return** $(\text{view}_{\mathcal{A}}, \{\tilde{x}_{i,j}\}_{i,j})$

$\mathbf{Exp}^{\text{Zk-Sim}}_{\mathcal{A},\mathcal{S}}(1^\lambda, n, T, \{y_{i,j}\}_{i,j}))$

1: $\text{pp} \leftarrow \text{Setup}(1^\lambda)$
2: $\{\text{sk}_i, \text{pk}_i\}_i \leftarrow \text{Sig.KeyGen}(\text{pp}, \{\text{pk}_i\}_i)$
3: $(\text{ek}, \text{vk}, \text{pk}_s, \text{sk}_s, L, \text{trap}) \leftarrow \mathcal{A}(\text{pp}, \{\text{pk}_i\}_i)$
4: $\text{out}_c^i \leftarrow \mathcal{S}_1^{\mathcal{A}}(\text{pp}, \text{pk}_s)$
5: $\{t_x^{i,j}\}_{i,j} \leftarrow \mathcal{A}(\text{pp}, \text{ek}, \text{vk}, \text{pk}_s, \text{sk}_s, L)$
6: $\{(\pi_{i,j}, \tau_x^{i,j})\}_{i,j} \leftarrow \mathcal{S}_2(\text{pp}, \text{ek}, \text{trap}, t_j, \text{out}_c^i, y_{i,j})$
7: **if** trap is not valid trapdoor for $(\mathcal{R}, \text{ek}, \text{vk})$
8: $\quad \vee \exists i \in [n], j \in [T] : t_x^{i,j} \leq t_{j-1} \vee t_x^{i,j} > t_j$
9: **return** $\perp$
10: $\mathcal{A} \leftarrow \{\tilde{y}_{i,j}, \pi_{i,j}, \tau_{i,j}\}_{i,j}$
11: **return** $(\text{view}_{\mathcal{A}}, \{y_{i,j}\}_{i,j})$

Figure 11: Experiments for the zero-knowledge definition.

Finally, in the shuffle indistinguishability experiment (Figure 12), the adversary $\mathcal{A}$ portrays the server and attempts to distinguish between two honest clients. These honest clients, get the same input (x, t_x), chosen by $\mathcal{A}$, after both having executed the GenRand protocol with $\mathcal{A}$. Subsequently, the environment only sends the outputs of Randomize for one of the clients to $\mathcal{A}$. $\mathcal{A}$ wins the game if it is able to successfully determine to which client those outputs belong.

$\mathbf{Exp}^{\text{Sh-Ind}}_{\mathcal{A}}(\lambda)$

1: $\text{pp} \leftarrow \text{Setup}(1^\lambda)$
2: $\{\text{sk}_i, \text{pk}_i\}_i \leftarrow \text{Sig.KeyGen}(\text{pp})$
3: $(\text{ek}, \text{vk}, \text{pk}_s, \text{sk}_s, L, \text{trap}) \leftarrow \mathcal{A}(\text{pp}, \{\text{pk}_i\}_i)$
4: $\mathcal{A}(\text{pp}) \rightarrow t_j$
5: **for** $i \in \{0, 1\}$
6: $\quad \text{GenRand}^{\mathcal{A}}(\text{pp}, t_j) \rightarrow \text{out}_c^i$
7: $\mathcal{A}(\text{pp}) \rightarrow (x, t_x)$
8: $b \leftarrow\!\!\$ \{0, 1\}$
9: $\text{Sig.Sign}_{\text{sk}_b}(x, t_x) \rightarrow \sigma_x^b$
10: $\text{Randomize}(\text{pp}, \text{ek}, t_j, \text{out}_c^b, x, t_x, \sigma_x^b) \rightarrow (\tilde{x}^b, \pi^b, \tau_x^b)$
11: $\mathcal{A}(\tilde{x}^b, \pi^b, \tau_x^b) \rightarrow b'$
12: **if** trap is not valid trapdoor for $(\mathcal{R}, \text{ek}, \text{vk})$
13: $\quad \vee \exists i \in [n], j \in [T] : t_x^{i,j} \leq t_{j-1} \vee t_x^{i,j} > t_j$
14: $\quad$ **return** $\perp$
15: **return** $b' = b$

Figure 12: Experiment for shuffle indistinguishability.

C.2 Proofs

We only provide a full proof for the Shuffle scheme, as this is our main result. Due to space constraints and similarity to the proof for the Shuffle scheme, we only provide sketches of the security proofs for the other schemes (Theorems 2 and 3).

Theorem 1. *$\mathcal{VLDP}_{shuffle}$ satisfies completeness, soundness, zero-knowledgeness and shuffle indistinguishability, given that* NIZK-PK *is secure for $\mathcal{R}_{Base}$,* Comm *a secure commitment scheme,* PRF *a secure pseudo-random function, and* Sig *an EUF-CMA secure digital signature scheme.*

Proof. We prove the properties one by one:

(1) Completeness. We see that the scheme satisfies completeness as long as the $\text{Randomize}_{\text{shuffle}}$ procedure outputs a valid NIZK-PK proof π for $\mathcal{R}_{\text{shuffle}}$. It therefore suffices to show that the proof for $\mathcal{R}_{\text{shuffle}}$ is correctly computed when all parties are honest.

To prove this, fix an arbitrary $i \in [n]$ and an arbitrary $j \in [T]$. First we observe that statement 1 of $\mathcal{R}_{\text{shuffle}}$ has to hold by the condition on $t_x^{i,j}$. Statement 2 of $\mathcal{R}_{\text{shuffle}}$ holds by construction of $\sigma_x^{i,j}$. Also, statements 3 and 5 hold, because these correspond exactly to the computations done in $\text{GenRand}_{\text{shuffle}}$ by the honest parties. Finally, statements 4, 6, and 7 hold by the fact that an honest party will evaluate these correctly in $\text{Randomize}_{\text{shuffle}}$.

Therefore, we can conclude that the proof π will be verified successfully, because our NIZK-PK scheme is complete itself, i.e.,

$\mathsf{Verify}_{\mathsf{shuffle}} \neq \bot$, except for some probability that is $\mathsf{negl}(\lambda)$. Since, i and j were picked arbitrarily, and both T and n are $\mathsf{poly}(\lambda)$, we can conclude that the total probability is also $\mathsf{negl}(\lambda)$.

(2) Soundness. To prove soundness, we define a series $\mathbf{Exp}_0$–$\mathbf{Exp}_3$ of hybrid experiments, where $\mathbf{Exp}_0$ is $\mathbf{Exp}^{\text{Snd-Real}}_{\mathcal{A},\mathcal{S}^*}(1^\lambda, n, T, \{x_{i,j}\}_{i,j})$ (from Definition 6 as defined in Figure 10) and $\mathbf{Exp}_3$ is close to the ideal. Recall, that by definition of soundness, the adversary $\mathcal{A}$, who controls all, potentially malicious, clients, can send messages to and receive corresponding replies from an honest server $\mathcal{S}^*$. We will show that all these experiments are (computationally) indistinguishable, and therefore $\mathcal{VLDP}_{\mathsf{shuffle}}$ is sound.

- $\mathbf{Exp}_0$: Equal to $\mathbf{Exp}^{\text{Snd-Real}}_{\mathcal{A},\mathcal{S}^*}(1^\lambda, n, T, \{x_{i,j}\}_{i,j})$.
- $\mathbf{Exp}_1$: This is the same experiment as $\mathbf{Exp}_0$, except that now we run a p.p.t. knowledge extractor $\mathcal{E}_{\mathcal{A}}$ to obtain the witness $\vec{w}$ that $\mathcal{A}$ used to generate the proof. We know that such an extractor exists, due to knowledge soundness of NIZK-PK. Now, instead of checking a proof $\pi_{i,j}$, the verifier uses $\vec{w}_{i,j}$ and $\vec{\phi}_{i,j}$ to check the statements of $\mathcal{R}_{\mathsf{shuffle}}$ directly. Clearly, both games are identical up to the probability that $\mathcal{A}$ wins the knowledge soundness game for one of the proofs. By knowledge soundness of NIZK-PK we know that this probability is negligible:

$$|\Pr[\mathbf{Exp}_0] - \Pr[\mathbf{Exp}_1]| \leq \mathsf{negl}(\lambda).$$

- $\mathbf{Exp}_2$: This is the same experiment as $\mathbf{Exp}_1$, except that now the verifier asserts that the values x and t_x in $\vec{w}_{i,j}$ are equal to $x_{i,j}$ and $t_x^{i,j}$. If this is not the case, we set $\mathsf{fail}_2 = \mathsf{true}$. Clearly both games are identical up to Fail_2:

$$|\Pr[\mathbf{Exp}_1] - \Pr[\mathbf{Exp}_2]| \leq \Pr[\mathbf{Fail}_2].$$

  Since Sig is an EUF-CMA secure signature scheme that has been used to generate σ_x and $\mathcal{A}$ does not know sk_i, it follows that $\Pr[\mathbf{Fail}_2] \leq \mathsf{negl}(\lambda)$.
- $\mathbf{Exp}_3$: This is the same experiment as $\mathbf{Exp}_2$, except that the server S^* now additionally maintains a list R containing entries of the form $(i, (\mathsf{pk}_i, \mathsf{cm}^i_{k_c}, k^i_s))$. These entries correspond to messages $(\mathsf{pk}_i, \mathsf{cm}_{k_c})$ received during occurrences of the $\mathsf{GenRand}_{\mathsf{shuffle}}$ protocol with user i. k^i_s corresponds to the server seed k_s that was generated by the server during this particular occurrence. Note, that each user i can only have one entry in R, due to step 4 in $\mathsf{GenRand}_{\mathsf{shuffle}}$. Now, if fail_2 has not been set, the server asserts, in $\mathsf{Verify}_{\mathsf{shuffle}}$, that R indeed has an entry $(\star, (\mathsf{pk}_i, \mathsf{cm}^i_{k_c}, k^i_s))$, where $(\mathsf{pk}_i, \mathsf{cm}^i_{k_c}, k^i_s)$ are the corresponding elements of $\vec{w}_{i,j}$. If this is not the case, we set $\mathsf{fail}_3 = \mathsf{true}$. Clearly both games are identical up to Fail_3:

$$|\Pr[\mathbf{Exp}_2] - \Pr[\mathbf{Exp}_3| \leq \Pr[\mathbf{Fail}_3].$$

  The only way, by which the event $\mathbf{Fail}_3$ can occur, is if the client can produce a tuple $(\mathsf{pk}_i, \mathsf{cm}^i_{k_c}, k^i_s)$ with signature σ^i_s, such that $\mathsf{Sig.Verify}_{\mathsf{pk}_s}(\sigma^i_s, \mathsf{pk}_i || \mathsf{cm}^i_{k_c} || k^i_s) = 1$. $\mathcal{S}^*$ will only have generated one signature for each pk_i, due to step 4 in $\mathsf{GenRand}_{\mathsf{shuffle}}$, and this tuple is on R. Therefore, for $\mathbf{Fail}_3$ to occur, the client must have forged a signature on a tuple $(\mathsf{pk}'_i, \mathsf{cm}^{i\prime}_{k_c}, k^{i\prime}_s)$, where at least one element differs from $(\mathsf{pk}_i, \mathsf{cm}^i_{k_c}, k^i_s)$. Since Sig is an EUF-CMA secure signature scheme that has been used to generate σ^i_s and $\mathcal{A}$ does not know sk_s, it follows that $\Pr[\mathbf{Fail}_3] \leq \mathsf{negl}(\lambda)$.

Finally, we observe that if Fail_3 does not occur, we are essentially at a point where $\tilde{x}_{i,j} = \mathsf{LDP.Apply}(x_{i,j}; \rho_{i,j})$, where $\rho_{i,j} = \mathsf{PRF}(k^i_c \oplus k^i_s, s_j)$. We observe that k_s is chosen uniformly at random and independently of $x_{i,j}$. Moreover, k^i_s is bound to all $x_{i,j}$, since the same public key pk_i is used inside σ_s and for the verification of σ_x. Also, s_j is public and independent of all $x_{i,j}$ and all $x_{i,j}$ are given. Next to this, k_c is uniquely determined by cm_{k_c}, except with negligible probability, according to the binding property of Comm. And, since $\mathsf{cm}^i_{k_c}$ is fixed before k^i_s is chosen uniformly at random, $k^i_c \oplus k^i_s$ is also be a uniform random bitstring, and by the definition of a secure PRF, $\rho_{i,j}$ will also be distributed at random, except with negligible probability. Therefore, it follows that:

$$\Big|\Pr[\mathbf{Exp}_3] - \Pr\big[\mathsf{LDP.Apply}(x_{i,j}; \rho_{i,j}) = \{y_{i,j}\}_{i,j} \;\big|\; \rho_{i,j} \leftarrow\!\!\$\ \{0,1\}^*\big]\Big| \leq \mathsf{negl}(\lambda).$$

(3) Zero-Knowledge. To show that $\mathcal{VLDP}_{\mathsf{shuffle}}$ satisfies the zero-knowledge property, we will show that the joint distribution of the output and all messages received by $\mathcal{A}$ in the real scheme is indistinguishable from those generated by the simulator $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2)$ (see Figure 13). Note, that in our model we assume that the verifier behaves like an honest-but-curious adversary, and thus follows the protocol. The first simulator algorithm $\mathcal{S}_1$, simulates a client in $\mathsf{GenRand}_{\mathsf{shuffle}}$, thereby also interacting with $\mathcal{A}$, representing the server. The second $\mathcal{S}_2$, simulates the $\mathsf{Randomize}_{\mathsf{shuffle}}$ algorithm.

$\mathcal{S}_1(\mathsf{pp}, \mathsf{pk}_s)$	$\mathcal{S}_2(\mathsf{pp}, \mathsf{ek}, \mathsf{trap}, t_j, \mathsf{out}^i_c, y_{i,j})$
1: $k_c \leftarrow\!\!\$\ \{0,1\}^*$	1: $\vec{\phi} = (t_{j-1}, t_j, \mathsf{pk}_s, s_j, y_{i,j})$
2: $r_{k_c} \leftarrow\!\!\$\ \{0,1\}^*$	2: $\pi \leftarrow \mathsf{Sim}_{\mathsf{nizk}}(\mathcal{R}, \mathsf{trap}, \vec{\phi})$
3: $\mathsf{cm}_{k_c} = \mathsf{Comm}(k_c; r_{k_c})$	3: **return** $(\pi, y_{i,j})$
4: Send $(\mathsf{pk}_i, \mathsf{cm}^i_{k_c})$ to $\mathcal{A}$ as client i.	
5: Receive (k_s, σ_s) from $\mathcal{A}$.	
6: **return** $(k_c, r_{k_c}, \mathsf{cm}_{k_c}, k_s, \sigma_s)$	

Figure 13: Simulator $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2)$ for the zero-knowledge proof of Theorem 1.

First, we observe that the message produced by $\mathcal{S}_1$ is distributed as in the real experiment, since our server is honest-but-curious and $\mathcal{S}_1$, follows the exact same steps as $\mathsf{GenRand}_{\mathsf{shuffle}}$. Second, we observe that $y_{i,j}$ is fixed. Third, we observe that the values in $\vec{\phi}$ are either public or fixed, and therefore are indistinguishable between the real world and the simulator. Since NIZK-PK is a secure scheme for $\mathcal{R}_{\mathsf{shuffle}}$ with the zero-knowledge property, and trap is a valid trapdoor, $\mathcal{S}_2$ can use the NIZK-PK simulator Sim to generate a proof π with the correct distribution, i.e., such that π passes verification for $\vec{\phi}$. From these observations it follows that the joint distribution of the real scheme and its output, is indistinguishable from that generated by $\mathcal{S}$, and therefore the scheme is zero-knowledge.

(4) Shuffle Indistinguishability. To prove shuffle indistinguishability, we describe a series of hybrid experiments $\mathbf{Exp}_0$ – $\mathbf{Exp}_3$, where $\mathbf{Exp}_0$ is equal to $\mathbf{Exp}^{\text{Sh-Ind}}_{\mathcal{A}}$, and $\mathbf{Exp}_3$ leaves $\mathcal{A}$ essentially random guessing. We will show that all these experiments are indistinguishable, and therefore $\mathcal{VLDP}_{\text{shuffle}}$ satisfies shuffle indistinguishability.

- $\mathbf{Exp}_0$: Equal to $\mathbf{Exp}^{\text{Sh-Ind}}_{\mathcal{A}}$ in Definition 8.
- $\mathbf{Exp}_1$: This is the same experiment as $\mathbf{Exp}_0$, except that now we also give trap as input to $\text{Randomize}_{\text{shuffle}}$ and replace the computation of π^b (step 6 in $\text{Randomize}_{\text{shuffle}}$), by a simulated proof using the NIZK-PK simulator Sim using trap. By the zero-knowledge property of NIZK-PK we conclude that both experiments are indistinguishable:
$$|\Pr[\mathbf{Exp}_0] - \Pr[\mathbf{Exp}_1]| \leq \text{negl}(\lambda).$$
- $\mathbf{Exp}_2$: This is the same experiment as $\mathbf{Exp}_1$, except that in step 1 of $\text{Randomize}_{\text{shuffle}}$ we replace k_c by a random bitstring of the same length. We will still pass verification, since π has been replaced by a simulated proof. Finally, by the hiding property of Comm, $\mathcal{A}$ cannot distinguish between the usage of k_c or some other random bitstring of the length, except with negligible probability. Therefore, we conclude that both experiments are indistinguishable:
$$|\Pr[\mathbf{Exp}_1] - \Pr[\mathbf{Exp}_2]| \leq \text{negl}(\lambda).$$
- $\mathbf{Exp}_3$: This is the same experiment as $\mathbf{Exp}_2$, except that we replace ρ in step 2 of $\text{Randomize}_{\text{shuffle}}$ by a random bit-string of the same length. We will still pass verification, since π has been replaced by a simulated proof. Finally, since k_c was already replaced by a uniform random bitstring, we know that k is a uniform random bitstring, and by the fact that PRF is secure, $\text{PRF}(k, s_j)$ is indistinguishable from a random bit-string of the same length. Therefore, we conclude that both experiments are indistinguishable:
$$|\Pr[\mathbf{Exp}_2] - \Pr[\mathbf{Exp}_3]| \leq \text{negl}(\lambda).$$

We observe that π^b is replaced by a simulated proof in $\mathbf{Exp}_3$, i.e., π^0 has the same distribution as π^1. Moreover, $\tilde{x}^b = \text{LDP.Apply}(x; r)$, where x is the same for both values of b and r is chosen uniformly at random, i.e., $\tilde{x}^0$ has the same distribution as $\tilde{x}^1$. Finally, τ^b_x is empty. Therefore, $(\tilde{x}^b, \pi^b, \tau^b_x)$ has the same distribution for either value of b, meaning the advantage of $\mathcal{A}$ in $\mathbf{Exp}_3$ is essentially the same as if $\mathcal{A}$ were random guessing. Thus, we conclude that

$$|\Pr[\mathbf{Exp}_3] - \frac{1}{2}| \leq \text{negl}(\lambda). \qquad \square$$

Theorem 2. *$\mathcal{VLDP}_{base}$ satisfies completeness, soundness, and zero-knowledgeness, given that* NIZK-PK *is secure for $\mathcal{R}_{Base}$,* Comm *a secure commitment scheme,* PRF *a secure pseudo-random function, and* Sig *an EUF-CMA secure digital signature scheme.*

Proof (Sketch). We prove the properties one by one:

(1) Completeness. By inspection of the protocol and completeness of NIZK-PK.

(2) Soundness. Also here, we define a series of hybrid experiments, where $\mathbf{Exp}_0$ is equal to $\mathbf{Exp}^{\text{Snd-Real}}_{\mathcal{A},S^*}$ and $\mathbf{Exp}_3$ is close to the ideal. In $\mathbf{Exp}_1$, we again use the knowledge extractor $\mathcal{E}_{\mathcal{A}}$ to obtain the witness $\vec{w}$ and verify the statements in $\mathcal{R}_{\text{base}}$ directly. $\mathbf{Exp}_2$ is analogous to that in the proof for Theorem 1.

$\mathbf{Exp}_3$ is similar to that in the proof for Theorem 1, however, R will now contain entries $((i, t_j), (\text{pk}_i, \text{cm}^{i,j}_{\rho_c}, k^{i,j}_s))$. I.e., each user i now has one entry for each t_j, rather than using the same entry for all t_j. Analogously to before, the server now verifies, in $\text{Verify}_{\text{base}}$ that $((\star, t_j), (\text{pk}_i, \text{cm}^{i,j}_{k_c}, k^{i,j}_s)$ is on R.

Finally, analogous to the proof for Theorem 1, by the binding property of Comm, we know that ρ is sampled independently and uniformly at random and, irrespective of $\mathcal{A}$. Therefore, we can conclude soundness.

(3) Zero-Knowledge. Just like in the proof for Theorem 1, $\mathcal{S}_1$ is identical to $\text{GenRand}_{\text{base}}$, i.e., for all t_j it computes cm_{ρ_c}, and receives (k_s, σ_s) from $\mathcal{A}$.

Just like in $\mathcal{S}_2$, we also create a simulated proof from the statement vector $\vec{\phi}$ alone. $\vec{\phi}$ consist of $\mathcal{S}_2$'s inputs, values from $\mathcal{S}_1$ and ρ_s. $\mathcal{S}_2$ simply computes ρ_s as in the real case. Note, that also here we use $y_{i,j}$ for $\tilde{x}$, rather than computing it from some signed input x and the randomness ρ.

Additionally, unlike the proof for Theorem 1, $\mathcal{S}_2$ outputs the values $(\text{pk}_i, \text{cm}_{\rho_c}, k_s, \sigma_s)$, where pk_i is known and fixed, and the other values come from $\mathcal{S}_1$.

Following arguments analogous to the proof for Theorem 1 and due to zero-knowledgeness of NIZK-PK we conclude that Base is zero-knowledge. $\square$

Theorem 3. *$\mathcal{VLDP}_{expand}$ satisfies completeness, soundness, and zero-knowledgeness, given that* NIZK-PK *is secure for $\mathcal{R}_{Base}$,* Comm *a secure commitment scheme,* PRF *a secure pseudo-random function,* Sig *an EUF-CMA secure digital signature scheme, and* CRH *(use to construct* MerkleTree*) a collision-resistant hash function.*

Proof (Sketch). We prove the properties one by one:

(1) Completeness. By inspection of the protocol and completeness of NIZK-PK.

(2) Soundness. Also here, we define a series of hybrid experiments, where $\mathbf{Exp}_0$ is equal to $\mathbf{Exp}^{\text{Snd-Real}}_{\mathcal{A},S^*}$ and $\mathbf{Exp}_3$ is close to the ideal. In $\mathbf{Exp}_1$, we again use the knowledge extractor $\mathcal{E}_{\mathcal{A}}$ to obtain the witness $\vec{w}$ and verify the statements in $\mathcal{R}_{\text{expand}}$ directly. $\mathbf{Exp}_2$ is analogous to the proof for Theorem 1.

$\mathbf{Exp}_3$ is analogous to that in the proof for Theorem 1, however, cm_{ρ_c} is replaced by rt.

Finally, analogous to the proof for Theorem 1, by the binding property of Comm, and by collision resistance of the hash function CRH used to construct the MerkleTree, we know that ρ is uniformly at random, irrespective of $\mathcal{A}$. Thus, we conclude that Expand is sound.

(3) Zero-Knowledge. Just like in the proof for Theorem 1, $\mathcal{S}_1$ acts identical to $\text{GenRand}_{\text{base}}$, i.e., for all t_j it computes rt, and receives (k_s, σ_s) from $\mathcal{A}$.

Just like in $\mathcal{S}_2$, we also create a simulated proof from the statement vector $\vec{\phi}$ alone. $\vec{\phi}$ consist of $\mathcal{S}_2$'s inputs, values from $\mathcal{S}_1$ and ρ_s. $\mathcal{S}_2$ simply computes ρ_s as in the real case. Note, that also here we use $y_{i,j}$ for $\tilde{x}$, rather than computing it from some signed input x and the randomness ρ. Additionally, unlike the proof for Theorem 1, $\mathcal{S}_2$ outputs the values $(\mathsf{pk}_i, \mathsf{rt}, k_s, \sigma_s)$, where pk_i is known and fixed, and the other values come from $\mathcal{S}_1$.

Following arguments analogous to the proof for Theorem 1 and due to zero-knowledgeness of NIZK-PK we obtain the zero-knowledge property for Base. □

D Appendix to Section 7

D.1 Dataset Description

Dataset 1: Geolife GPS Trajectory. This is a location dataset of 182 users, collected as part of Microsoft Research Asia's GeoLife project over the period of 2007 to 2012.[8] Each data point contains latitude, longitude (GPS coordinates) and altitude information of a user on a given day. Upon inspecting the data, we found that it was highly sparse. In particular, only a small subset of users had GPS coordinates recorded for any given day. We therefore decided to extract five readings from each user from five different days, assuming that the corresponding readings were taken from the same day. For each day, we only took the first GPS coordinates. We then used the Nominatim API[9] to obtain the address corresponding to each GPS coordinate using reverse geocoding. From the address thus returned, we retained only the postcode of each location. Most of the postcodes were only visited by a very small amount of users. We therefore took the 7 top postcodes and included the rest into a single postcode named `all_others`. Thus, in total we have 8 postcodes per day. This dataset is used for the LDP algorithm for histograms where the goal is to estimate the true histogram of users in each postcode per day. Note that we have $k = 8$, where $\{1, \ldots, k\}$ represent the respective postcodes in the algorithm for histograms (see Figure 1).

Dataset 2: Smart Meter. This dataset is extracted from the smart meters in London dataset.[10] Which was originally extracted from energy readings dataset of 5,567 London households, which were obtained during the Low Carbon London project led by UK Power Networks between 2011 and 2014. More specifically, we took the `daily_dataset.csv` file and take the mean energy reading from each household for the last five days of this dataset with the last date 25/02/2014. Households that did not have a mean energy reading for a particular day are assigned mean energy of 0 for that day. Finally, we normalized the readings by dividing each mean energy value by the maximum mean energy in `daily_dataset.csv` which was 6.928 kWh. This dataset is used for the LDP algorithms for reals where the goal is to estimate the average energy reading of a household per day, by estimating the sum of mean energy consumption across all households and then dividing it by the number of households, i.e., 5,567. We use a precision level of $k = 10$ (see the algorithm for reals in Figure 1) for our experiments.

[8]The original dataset can be found at https://www.microsoft.com/en-us/research/publication/geolife-gps-trajectory-dataset-user-guide/
[9]See https://nominatim.org/
[10]See https://www.kaggle.com/datasets/jeanmidev/smart-meters-in-london

D.2 Possible Optimizations and Alternatives

While the experiments in Section 7 show the practicality of our schemes, certain optimizations and/or different choices could be made with respect to our implementation. Specifically, we discuss how different choices would influence the trade-off between security, efficiency, and practicality.

Our current choices were based on primitives that provide a strong level of security (targeting 128 bits) within practical times. Given that our results show that performance is very practical, efficiency improvements are not a requirement for practical adoption, however may still be considered depending on the specific requirements of the use case.

SNARK-friendly CRH. In our current implementation, we rely on the Blake2s CRH inside our PRF, Sig scheme, and MerkleTree. Blake2s is a standardized scheme and its security level has been well vetted and confirmed. Moreover, it is more efficient to encode inside a zk-SNARK circuit than alternatives with a similar security level, e.g., SHA256. However, in some recent works we observe an increased interest in so-called SNARK-friendly hash functions, often algebraic hash functions that can be more efficiently computed inside a circuit. Examples of such schemes are Poseidon [32], MiMC [2], and Pedersen [34] hashes. These schemes can reduce prover times by as much as a factor of 10 [52]. However, this often comes at the cost of reduced security. Poseidon and MiMC are novel constructs and have not yet been properly vetted by the community. This novelty, in combination with their algebraic construction, might give rise to unforeseen attacks, such as the algebraic attacks shown against MiMC [29]. The Pedersen hash on the other hand is known to be secure, but has the downside that it relies on the discrete log assumption, whereas Blake2s requires no such assumption. We can safely use the Pedersen hash inside our Merkle tree, since breaking the discrete log assumption would require efforts similar to breaking the zk-SNARK scheme we use.

However, we have chosen to not use Pedersen hashes for our PRF and Sig schemes. The PRF is evaluated out-of-circuit in the Base and Expand schemes anyhow, and we chose to use the same primitive in the Shuffle scheme for comparability.

For the Sig schemes, we chose to use commonly available primitives, as in many use cases the trusted environment will not provide more novel or custom primitives, such as Pedersen or Poseidon hash functions. However, in cases where such primitives are available, they could be used to improve efficiency at the cost of (slightly) decreased security and general applicability.

SNARK-friendly signatures. Next to using a different hash function to transform the input message to a fixed digest, we could have also used a different signature scheme than Schnorr's. However, it should be noted that Schnorr is commonly available and very efficient. E.g., it is around 2-2.5x times faster than EdDSA inside a zk-SNARK circuit [52]. Although, due to the small number of constraints, the impact on the total performance of our scheme will be small.

zk-SNARK. The predominant component for our computation times is determined by the zk-SNARK scheme and the way our constraints are encoded inside the zk-SNARK circuit.

First, we note that we implemented our constraint circuit using readily available 'gadgets' from the Arkworks library. While these gadgets are of good quality and are implemented efficiently, we do get some more constraints than are strictly required in hand-optimized constraint systems. These constraints might possibly be reduced by manual inspection, however due to the large number of constraints we expect this to be a tedious task, for which we only get an (almost) negligible increase in performance. Moreover, manual optimization of these constraints would reduce modularity of our software implementation, which may be a significant practical drawback.

Next to this, one could also consider using an alternative scheme to Groth16. For example, schemes with a *universal setup* (e.g., Marlin [23]), *post-quantum security* (e.g., Fractal [24]) or a *transparent setup* (e.g., Fractal [24] or SuperSonic [18]) could be employed. Generally speaking, these alternatives are less efficient than Groth16, with respect to proof generation and verification time. However, in some use cases, universal setups can be used to prevent the server from having to run one setup per LDP algorithm. In practice, however we do not expect this trade-off to be beneficial. Moreover, the removal of a trusted setup is not necessary in our case, since we assume the server to behave semi-honestly and not collude with any of the clients, i.e., it will not give the trapdoor to any of the clients.