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) is an efficient solution for providing privacy to client's sensitive data while simultaneously releasing aggregate statistics without relying on a trusted central server (aggregator) as in the central model of differential privacy. The shuffle model with LDP provides an additional layer of privacy, by disconnecting the link between clients and the aggregator, further improving the utility of LDP. 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. Specifically, we give efficient constructions to *prevent both input ánd output manipulation attacks* from malicious clients for generic LDP algorithms. Our proposed schemes for verifiable LDP (VLDP), completely protect from output manipulation attacks, and prevent input attacks using signed data, requiring *only one-time interaction between client and server*, unlike existing alternatives [28, 33]. Most importantly, we are the first to provide an efficient scheme for *VLDP in the shuffle model*. We describe and prove secure, two schemes for VLDP in the regular model, and one in the shuffle model. We show that all schemes are *highly practical*, with client runtimes of < 2 seconds, and server runtimes 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 the true input. 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 (see also [11]). This consists of several sequential sub-components which pass information to one another resulting in the final formatted data. In these cases, we may realistically assume that the adversary only controls some components of the source, rather than the entire source itself. 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 [21]. More specifically, we focus on the *local* [27] and *shuffle* [17] models of differential privacy (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 nodes 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 from which sensors to read and what data to send in a prescribed format to the server. The sensor device itself consists of various sensors, which are individual hardware components (chips).[1] An adversary could take control of the sensor device by replacing this main program with its own malicious program, which could influence this local data processing pipeline. However, such a program does not corrupt the physical sensors themselves, nor the raw data it produces.

Energy companies want to obtain 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 [32]. 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 controlled/trusted environment, it may be malicious, 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 in the city or to regulate crowd during busy events. Crowd estimation does not necessarily require exact GPS coordinates of individual devices, and a coarse-grained aggregate location distribution often suffices. This is an excellent use case of LDP to release a histogram of hotspots. However, 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 device's sensors as being true/correct (operating system (OS) 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 a sensor are collected in the OS space, before the applications running in the user space can process them further for a given task, e.g., gesture detection. 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 to send the formatted input to a server.

[1] See for example: https://os.mbed.com/platforms/NAMote-72/.

Our Contributions. Assuming that the raw data is produced by a *trusted component* at a node, we can attach a *digital signature* to the data item. The server can then verify if a received input from a node is correct by verifying the signature of the initial raw input, and if subsequent processing is done correctly. Our contributions are as follows:

- We propose three schemes for efficient verifiable LDP (VLDP). For each LDP message sent to the server, the client only spends *1–3 seconds*. Furthermore, the load at the server is *5–7 milliseconds* per client, which can be parallelized. Similar to comparable works [28, 33], our baseline scheme requires multiple interactive rounds between client and server. Our other, improved schemes reduce the server load by requiring only one such round, at a very small cost, by using randomness expansion techniques. All our schemes protect against both *input* and *output manipulation attacks* as defined in [28]. In the input manipulation attack, the attacker can arbitrarily change the initial input while carrying out the rest of the LDP algorithm faithfully. We ensure protection against input manipulation using digital signatures with the assumption that it is signed by a trusted component. In output manipulation, the attacker can send arbitrary outputs to the server. We ensure protection against this attack using verifiable randomness and zero-knowledge proofs.
- We present the *first* scheme for VLDP in the *shuffle model* [17]. In this model a trusted shuffler shuffles the locally randomized inputs from the nodes, with the net effect of privacy amplification [5]. The shuffle model scheme is not a straightforward generalization of the LDP constructions as we need to ensure that the shuffled messages cannot be linked to specific nodes. 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. The computation load of the server is even slightly decreased.
- Our VLDP protocols are designed for the k-ary randomized response (k-RR) mechanisms for both histograms and real-valued data as described in [5]. Comparable works to ours on verifiable differential privacy have mostly targeted binary randomized response. 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., Exponential or Laplace mechanisms [21] (by using a piecewise approximation of $\ln(\cdot)$). The requirement for randomization approximation using fixed uniform random bits is due to the use of non-interactive zero knowledge proofs discussed in detail in Section 4 and Appendix C.2.
- 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 only 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 this discussion to protocols for verifiable differential privacy (see also [11]), 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 [2]. They mention an even earlier work by Kikuchi et al [29], who reinvented the notion of randomized response (RR) for voting applications and provided cryptographic constructions to protect against cheating voters. According to [2] the earlier constructions from [29] are less efficient and provide weaker security than their constructions.

The main security concern addressed by [2] is 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, Cao and Yoshikawa [28] extend the work from [2] to several other variants of LDP. However, their techniques are similar and once again they 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. Furthermore, their protocols do not cover the shuffle model.

In [34], 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 provide detailed verifiable algorithms for binary RR and the exponential mechanism [31] 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 scrutinize using rigorous security definitions.

The closest work to ours is from [33] 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 specifying this further (as we show in Appendix C.2 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. Verification of unbiased randomness in the RR mechanism is done through an interactive protocol between the user and the server which works if at least one of the two is honest. The user provides a NIZK proof as a proof of correct application of the RR mechanism.

Our protocol has one significant element built on their work, i.e., generating joint randomness. However, unlike them, our protocols can process the RR mechanism for general k, allowing composition of several LDP mechanisms, protect against input manipulation, do not require a specific blockchain infrastructure, and provide LDP in the shuffle model. Moreover, apart from our baseline protocol, we only require one single round of interaction between client and server (unlike [33]), 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, which [33] uses for storing the private attributes and transferring their DP versions.

Central model. The construction from Narayan et al [35] 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 to ours. The work from [42] uses a similar threat model to [35]. Randomness, however, is generated interactively between the curator and a "reader" (an entity who is interested in verifying the claims of the analyst). The work from [6] tackles the problem of verifiable DP in the single curator and multiparty setting. In the former, one server collects all inputs from clients. 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 which is then presented to the analyst. The servers in both settings may be actively corrupt. Verification of inputs from the clients is limited to range checks (i.e., whether it lies in a given range). This is unlike our schemes which operate in the LDP setting, and allow verification of exact inputs rather than range proofs.

Other works. Another related work to ours, but which does not consider DP, is the ADSNARK system [4] for proving computation on authenticated data while maintaining privacy. Like our work, 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 SNARKs. Unlike us however, their setting is not distributed, and does not consider DP inputs from the client. Finally we would like to point out several works showing the susceptibility of LDP to input manipulation (or data poisoning) attacks [16, 30, 43], which highlight the need for cryptographic solutions for the integrity of initial data and 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 in both the local and shuffle model.

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, $\mathsf{PRF}(k, \cdot)$ should be indistinguishable from a true random function. We can use PRFs to construction *pseudo-random number generators (PRGs)* following [10, Section 4.4]: if we require a random bitstring from $\mathcal{Y}^\ell$, we define ℓ distinct, fixed input values $x_1, \ldots, x_\ell \in \mathcal{X}$. Subsequently, we can create a PRG with seed $k \in \mathcal{K}$ as $\mathsf{PRG}(k) = \mathsf{PRF}(k, x_1)|| \ldots ||\mathsf{PRF}(k, x_\ell)$. This PRG construction will be implicitly used in the remainder of this work.

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 $(\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$, where KeyGen is the key generation algorithm, Sign is used to create a signature σ for a message m, and Verify is used to assert that σ is a valid signature for m. We only require that the digital 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.

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 [9, 20, 25] (where we can also reuse this CRS for any polynomial number of proofs), which can be defined as a 4-tuple of p.p.t. algorithms $(\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify}, \mathsf{Sim})$. The Setup algorithm generates the evaluation ek and verification vk key pair (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, the algorithm Sim uses the simulation trapdoor trap and ek to create a simulated proof for a statement x.

A secure NIZK scheme should satisfy the following 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: for our constructions we 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. Formal definitions of the above mentioned properties are given in Appendix A. In our implementation, we make use of zk-SNARKs [7]: 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) [21] 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 [21])**.** A randomized algorithm $\mathcal{M} : \mathcal{X}^n \to \mathcal{Y}$ is (ϵ, δ)-differentially private, if for all $X \sim X' \in \mathcal{X}^n$ and for all $T \subseteq \mathcal{Y}$, we have

$$\Pr[\mathcal{M}(X) \in T] \leq e^{\epsilon} \Pr\left[M(X') \in T\right] + \delta.$$

Any (ϵ, δ)-DP algorithm possesses two properties that are useful for defining DP in the local and shuffle model, following [5]:

Lemma 1 (Post-processing [21]). *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 [22]). *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 [17]:

- A randomizer $\mathcal{R} : \mathcal{X} \to \mathcal{Y}$ that takes as input a data entry x^i and outputs a value $\tilde{x}^i \in \mathcal{Y}$.[2]
- A shuffler $\mathcal{S} : \mathcal{Y}^n \to \mathcal{Y}^n$ that takes as input a vector of n messages and outputs these messages 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, $\mathcal{C} : \mathcal{Y}^n \to \mathcal{Z}$, that takes as input a vector of n messages $(\tilde{x}_{\pi_1}, \ldots, \tilde{x}_{\pi_n})$ and outputs an estimation of $f(x_1, \ldots, x_n)$.

As observed in [17], by Lemma 1, the shuffler $\mathcal{S}$ does not affect the (ϵ, δ)-DP property of the randomizer $\mathcal{R}$. Therefore, if $\mathcal{R}$ is (ϵ, δ)-DP as a standalone algorithm, it is also (ϵ, δ)-DP in the shuffle model. Clearly, Lemma 2 therefore also holds in the shuffle model [17].

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 of local differential privacy [27]. The purpose of the shuffle mechanism is to amplify the privacy achievable via LDP, as we explain in Section 4. In the remainder of this work, when we refer to an LDP algorithm, we will only denote the local randomizer, unless explicitly stated otherwise.

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

4 DP Algorithms

We consider two LDP algorithms, both of which appear in [5]. 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 that the input $x \in [0, 1]$. For a precision level k, we first encode x as an integer as follows [5]:

$$\overline{x} = \lfloor xk \rfloor + \text{Ber}(xk - \lfloor xk \rfloor)$$

It can be easily verified that encoding in this way 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 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 $\mathcal{M}(x_1, \ldots, x_n) = (\mathcal{R}(x_1), \ldots, \mathcal{R}(x_n))$ is also (ϵ, δ)-DP.

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 needs to do a de-biasing step. Let x_i be the input of user i, let $\overline{x}_i$ be the same input with precision k, and $\tilde{x}_i$ be the input received from user i through the LDP algorithm. Then, as shown in Appendix C.1, the aggregator outputs:

$$\frac{1}{1-\gamma}\left(\frac{\sum_{i=1}^{n} \tilde{x}_i}{k} - \frac{\gamma n}{2}\right),$$

as the estimate of $\frac{1}{k}\sum_{i=1}^{n} \overline{x}_i$ [5], 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 (single-message) shuffle model of DP employs a shuffler $\mathcal{S} : \mathcal{Y}^n \to \mathcal{Y}^n$ which is a random permutation of its inputs. The algorithm $\mathcal{M} : \mathcal{S} \circ \mathcal{R}^n : \mathcal{X}^n \to \mathcal{Y}^n$ then provides (ϵ, δ)-DP against the curator, but with the advantage that the local randomizer 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 [5] 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 randomized response, with $n = 100$ participants, $\delta = 10^{-6}$ and $\epsilon = 0.1$, we get $\epsilon_0 \approx 1.0032$ through the Bennett bound. This means, we can use the mechanism *10 times more* than the LDP mechanism alone.

LDP Algorithm for Reals [5]	LDP Algorithm for Histograms [5]
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**	$\tilde{x} \leftarrow \overline{x}$
$\tilde{x} \leftarrow \overline{x}$	**else**
else	$\tilde{x} \leftarrow\!\!\$\ \{1, \ldots, k\}$
$\tilde{x} \leftarrow\!\!\$\ \{0, 1, \ldots, k\}$	**return** $\tilde{x}$
return $\tilde{x}$	

Figure 1: LDP algorithms for reals and histograms.

LDP inside NIZK. To verify the above LDP algorithms inside a NIZK circuit, we need to define how we evaluate the 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 the 'recreate' the random sampling inside the NIZK circuit. Next, we observe that the NIZK proof is computed over a given, fixed, agreed upon relation $\mathcal{R}$. Therefore, the 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 $\text{LDP.Apply}(x; \rho)$ approximately equals that of the true randomized LDP algorithm. In Appendix C.2 we specify approximate distributions for the algorithms described above, such that the statistical distance decreases exponentially in ℓ.

5 Verifiable DP in the Local and Shuffle Model

We first sketch our system model and give a formal definition for a VLDP scheme that is applicable to both the local and the shuffle model. Subsequently, we elaborate the threat model and formally describe the necessary security properties.

5.1 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 2. 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 the client and server together generate a random value to be used at time t_j, for $j \in [T]$.

The process with which a client i generates a VLDP value $\tilde{x}$ at time step t_x starts with the client requesting a fresh raw input value x from the trusted environment. In response, the trusted environment returns a signed input value x with signature $\sigma_x = \text{Sig.Sign}_{\text{sk}_i}(x||t_x)$, where sk_i is the secret key of the trusted environment of the i-th client, which we assume has already been generated beforehand. This signature can be verified using pk_i. Subsequently, the client i calls Randomize to apply verifiable LDP to x and obtain $\tilde{x}$ and a correctness proof π, both of which are sent to the shuffler (or directly to the server in the local model). The shuffler collects all messages $((\tilde{x}_1, \pi_1), (\tilde{x}_2, \pi_2), \ldots, (\tilde{x}_n, \pi_n))$ and forwards these in a random order $((\tilde{x}_{?_1}, \pi_{?_1}), (\tilde{x}_{?_2}, \pi_{?_2}), \ldots, (\tilde{x}_{?_n}, \pi_{?_n}))$ to the server, thereby ensuring that the server cannot determine which message belongs to which client.

For each received $(\tilde{x}_{?_i}, \pi_{?_i})$, the server runs Verify, to ensure that $\tilde{x}_{?_i}$ is created from a value x signed by a valid $\text{pk}_{?_i}$, using a true random value. Finally, the server uses all valid values $(\tilde{x}_{?_1}, \ldots, \tilde{x}_{?_n})$ to compute and publish its desired output $f(\tilde{x}_{?_1}, \ldots, \tilde{x}_{?_n})$.

Definition 2 (VLDP Scheme)**.** A VLDP scheme for an LDP algorithm $\text{LDP.Apply} : \mathcal{X} \to \mathcal{Y}$ is a 5-tuple of p.p.t. algorithms $\mathcal{VLDP}$ for any number $n \geq 1$ of clients and one prover:

- $\text{Setup}(1^\lambda) \to \text{pp}$: 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 pp_{sig} and a commitment scheme pp_{comm}. Optionally, it also returns a vector $\vec{s}$ of T PRF seeds.
- $\text{KeyGen}(\text{pp}) \to (\text{ek}, \text{vk}, \text{pk}_s, \text{sk}_s, L)$: Given the public parameters pp, this algorithm returns the evaluation ek and verification key vk for the NIZK proof, and the server's public pk_s and secret sk_s signature keys, together with a list L. This list is populated with the identities of clients that have already been processed in a given time period.
- $\text{GenRand}(\text{pp}, \text{aux}) \to \text{out}_c$: This interactive protocol between a single client and server takes as input the public parameters pp and optional auxiliary information. The output of the client is defined as out_c, 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.
- $\text{Randomize}(\text{pp}, \text{ek}, t_j, \text{out}_c, x, t_x, \sigma_x) \to (\tilde{x}, \pi, \tau_x)$: The client i uses pp, ek, a timestamp t_j, its output from the GenRand protocol out_c, the true input value x with timestamp t_x and their signature $\sigma_x = \text{Sig.Sign}_{\text{sk}_i}(x||t_x)$ to compute an LDP value $\tilde{x}$, a NIZK proof π, and a vector of public values τ_x.
- $\text{Verify}(\text{pp}, \text{vk}, t_j, \tilde{x}, \pi, \tau_x) \to \tilde{x} \cup \perp$: The server uses pp, ek, a timestamp t_j, out_s, $\tilde{x}$, π, and τ_x to verify whether $\tilde{x}$ was computed honestly. It returns $\tilde{x}$ if this is the case and $\perp$ otherwise.

5.2 Threat model

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

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 $\sigma_x = \text{Sig.Sign}_{\text{sk}_i}(x||t_x)$. sk_i is contained inside the trusted environment, and cannot be accessed by the potentially malicious client program. Recall that we use the term trusted environment to mean any controlled environment outside adversarial reach such as a secure enclave, OS space, or hardware module.

Server: The server is assumed to behave *semi-honestly*, i.e., it will not deviate from the scheme, but does try to obtain as much information as possible within the bounds set by the scheme. Moreover, the server is assumed to be a non-colluding entity. Finally, we

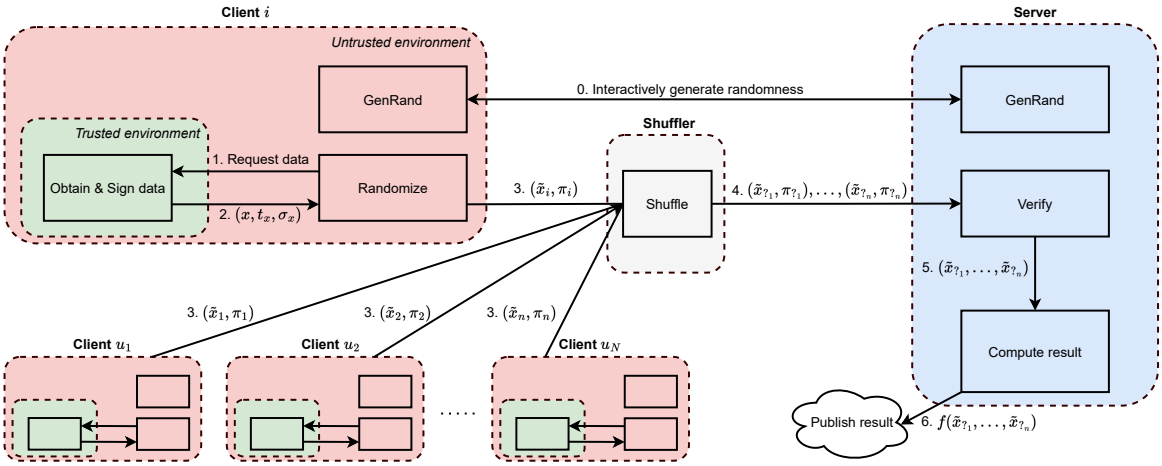

Figure 2: System model for the VLDPPipeline **in the shuffle model. When using the 'regular' local model, the shuffler is removed and the messages of step 3 are sent directly to the server instead.**

assume that the server can verify which pk_i's are known, trusted public keys belonging to a trusted environment, e.g., via a public key infrastructure or whitelist of trusted keys.

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 D. 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 [5, 17].

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 E, 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 3 (Completeness)**.** A scheme $\mathcal{VLDP}$ for an LDP method $\mathsf{LDP.Apply} : \mathcal{X} \to \mathcal{Y}$ 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 $\Pr\left[\perp \in \mathbf{Exp}^{\text{Comp}}_{\mathcal{A}}(1^\lambda, n, T)\right] \leq \mathsf{negl}(\lambda)$, with $\mathbf{Exp}^{\text{Comp}}_{\mathcal{A}}$ 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 4 (Soundness)**.** A scheme $\mathcal{VLDP}$ for an LDP method $\mathsf{LDP.Apply} : \mathcal{X} \to \mathcal{Y}$ 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

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

with $\mathbf{Exp}^{\text{Snd-Real}}_{\mathcal{A}}$ as defined in Figure 11, 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 5 (Zero-knowledge)**.** A scheme $\mathcal{VLDP}$ for an LDP method $\mathsf{LDP.Apply} : \mathcal{X} \to \mathcal{Y}$ 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

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

with $\mathbf{Exp}^{\text{Zk-Real}}_{\mathcal{A}}$ and $\mathbf{Exp}^{\text{Zk-Sim}}_{\mathcal{A},\mathcal{S}}$ as defined in Figure 10.

Additionally, for a VLDP scheme to be secure in the shuffle model, we need the property of *shuffle indistinguishability* to hold. This guarantees that the server cannot distinguish an output that was sent by client i from an output sent by client i'.

Definition 6 (Shuffle indistinguishability)**.** A scheme $\mathcal{VLDP}$ for an LDP method $\mathsf{LDP.Apply} : \mathcal{X} \to \mathcal{Y}$ with security parameter λ has shuffle indistinguishability if for every p.p.t. adversary $\mathcal{A}$: $2|\Pr[\mathbf{Exp}^{\text{Ind}}_{\mathcal{A}}(\lambda) = 1 | \mathbf{Exp}^{\text{Ind}}_{\mathcal{A}}(\lambda) \neq \perp] - \frac{1}{2}| \leq \mathsf{negl}(\lambda)$, with $\mathbf{Exp}^{\text{Ind}}_{\mathcal{A}}$ 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 the respective construction of the VLDPPipeline. Each scheme will improve upon the construction of the previous one, culminating in an efficient VLDP scheme that can be applied in the shuffle model, with minimal interaction between server and clients. A formal security analysis of each scheme is provided in Appendix E.

(1) The Base scheme achieves verifiable LDP in the local model. Its GenRand protocol is loosely inspired by the VerRR algorithm in [33]. The other algorithms are novel constructions which together form a scheme that unlike [33] also provides security against input manipulation attacks for authenticated data, supports generic LDP algorithms, and does not require a blockchain infrastructure.
(2) The Expand scheme provides the same guarantees, but only requires the interactive GenRand protocol to be run once per client, rather than for each new input value of each client. 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 the second version, but also achieves verifiable local differential privacy in the shuffle model.

6.1 Base scheme

Figure 3 describes the Base scheme in detail. During the protocol $\mathsf{GenRand}_{\text{base}}$, the client and server together compute the necessary values to construct a true random value ρ for later use in $\mathsf{Randomize}_{\text{base}}$. The bit length of ρ will be equal to the output of the PRF that we use to generate ρ, let $|\rho|$ denote this length. In case the required number of bits ℓ needed to evaluate LDP.Apply is lower than $|\rho|$ we can simply ignore the left-over 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 the experimental evaluation (Section 7) we will evaluate how the performance depends on ℓ.

Now, in $\mathsf{GenRand}_{\text{base}}$, the client i first generates its own random bits ρ_c (we explicitly show the use of a PRF in step 1 and 2 of Figure 3 to resemble the later schemes). Subsequently, i computes a commitment cm_{ρ_c} to ρ_c, and shares this commitment along with the public key from its trusted environment pk_i, and a time marker t_j with the server. The eventual randomness will also be 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 user i, and subsequently verifies that user i did not yet construct a random value for t_j, i.e., whether $(i, t_j) \notin L$. Next, the server generates a valid PRF seed k_s and computes $\sigma_s = \mathsf{Sig.Sign}_{\mathsf{sk}_s}(\mathsf{pk}_i||\mathsf{cm}_{\rho_c}||k_s||t_j)$. The server then sends (k_s, σ_s) to i, who verifies σ_s. Note that, rather than using a signature, the server might instead also maintain a state of $(\mathsf{pk}_i, \mathsf{cm}_{\rho_c})$ for each client and compare this state in $\mathsf{Verify}_{\text{base}}$ later. We, however, choose this approach to minimize the storage load for the server.

In $\mathsf{Randomize}_{\text{base}}$, the client computes the server part of the randomness ρ_s from k_s, and combines the client and server parts to obtain a true random value $\rho = \rho_c \oplus \rho_s$. Subsequently, the client uses ρ to transform x into a differentially private value $\tilde{x}$ 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 was signed using pk_i and obtained at a time t_x, such that $t_{j-1} < t_x \leq t_j$; (2) ρ is a true random value, i.e., $\mathsf{cm}_{\rho_c} = \mathsf{Comm}(\rho_c; r_{\rho_c})$ and $\rho = \rho_c \oplus \rho_s$; and (3) $\tilde{x}$ is the result of $\mathsf{LDP.Apply}(x; \rho)$.

The server receives the LDP value $\tilde{x}$, the proof π and the other required public values $(\mathsf{pk}_i, \mathsf{cm}_{\rho_c}, k_s, \sigma_s)$ from the client, and verifies correctness of ρ_s, σ_s and π. If both hold, it knows that $\tilde{x}$ is a correct differentially private version of an authentic input.

As mentioned, this scheme shows some similarities to [33]. Specifically, they also let the client commit to a random value, and let the server subsequently generate an independent random value and bind both values using a signature under the server's key. Next, they combine these two random values and use that to generate a randomized response for a single bit. Finally, they use zk-SNARKs (a specific form of NIZK-PKs) to guarantee correctness.

The main differences between our Base scheme and [33] are as follows. First, we allow for generic, more complex LDP algorithms, rather than binary RR. Moreover, we do not require any DLT/blockchain infrastructure for our scheme. Lastly, our scheme works on authenticated data, unlike [33].

6.2 Randomness expansion (expand) scheme

The Base scheme requires one execution of GenRand for each call to Randomize. For a large number of clients, this would put an impractically large load on the server, due to the interactive nature of GenRand. However, we can employ a more efficient version of GenRand to minimize the number of runs to only one per client. The Expand scheme uses Merkle trees as compact commitments to multiple random values, in order to reduce the load on the server. Specifically, we update step 2–4 of GenRand by creating T commitments to T randomly generated values ρ_c^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 equivalent to that of $\mathsf{GenRand}_{\text{base}}$. The remainder of $\mathsf{GenRand}_{\text{expand}}$ is equivalent to $\mathsf{GenRand}_{\text{base}}$. The main improvement is that we can now generate T true random values with only one round of communication, with communication and server-side cost independent of T.

This improvement does require some changes and additional computations for the client in $\mathsf{Randomize}_{\text{expand}}$. Following Section 3, given an array of distinct, public values $\vec{s} = (s^1, \ldots, s^T)$, we can define a secure PRG as $\mathsf{PRG}(k) := \mathsf{PRF}(k||s^1)|| \ldots ||\mathsf{PRF}(k||s^T)$. Thus, if we consider the j-th call to $\mathsf{Randomize}_{\text{expand}}$, we can compute the server part of the randomness (line 1) as $\rho_s = \mathsf{PRF}(k_s||s^j)$, where k_s is the server seed. 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 $\mathsf{Randomize}_{\text{expand}}$ is indeed the j-th entry of the Merkle tree with root rt. This ensures not only that the client uses a random value that was committed to before seeing k_s, but also ensures that the client has no choice in which random value in the Merkle tree it uses. Allowing the client

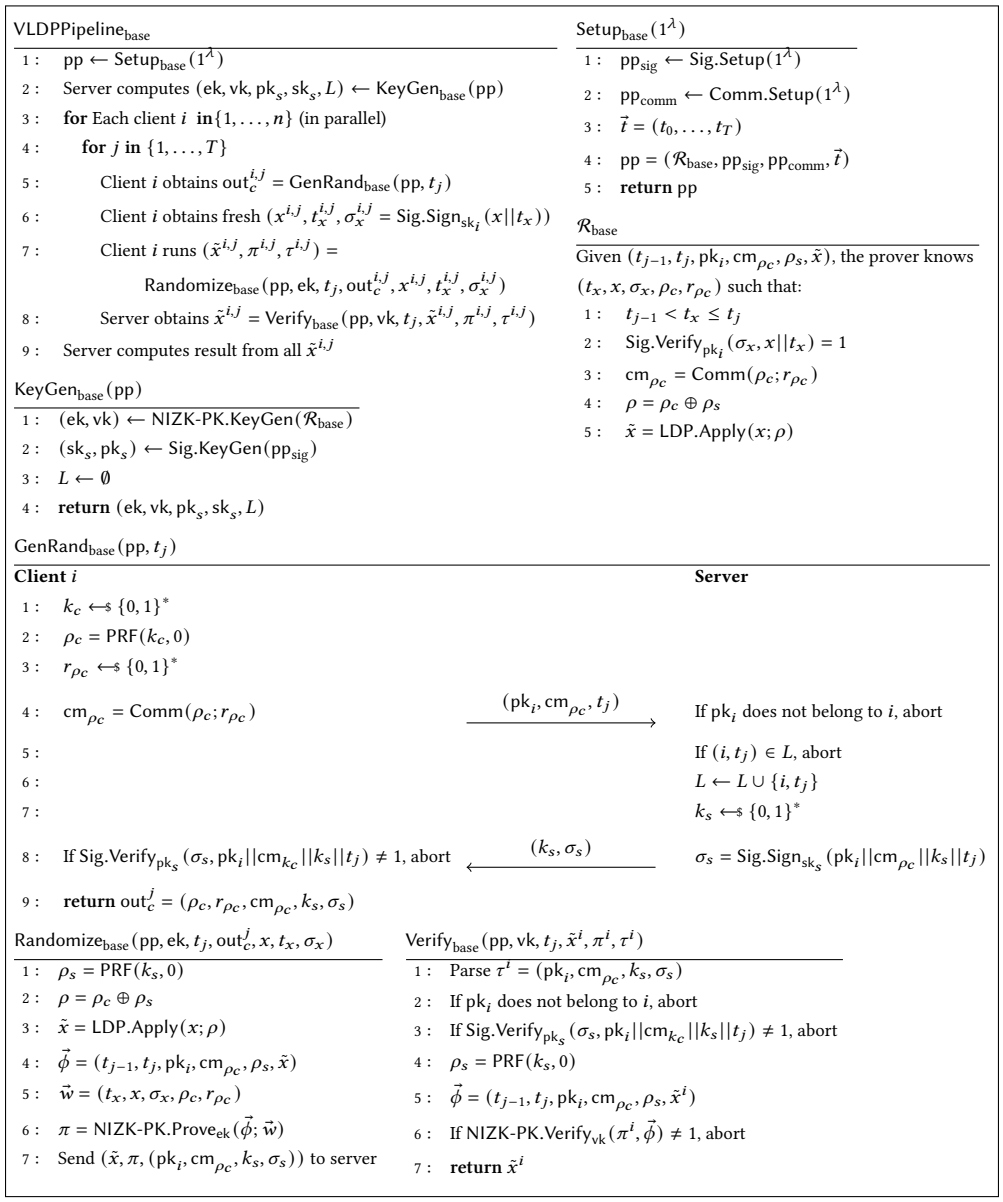

Figure 3: Base **scheme for VLDP between one server and multiple clients.**

to choose which value it uses could make it possible to influence the value of $\tilde{x}$, by cleverly constructing the random elements in $\vec{\rho}_c$.

Due to space constraints, and the nature of the differences with Base and Shuffle, we include a detailed definition of the Expand scheme in Figure 6 in Appendix B.

6.3 Shuffle model scheme

An interesting question to ask is whether either of the previous schemes also work in the shuffle model. In this model, the client could answer more server queries within the same privacy budget, since it decreases less quickly (see Section 4). However, we observe that neither the Base nor the Expand scheme can be applied directly in the shuffle model, since the public values pk_i, σ_x, $\mathsf{cm}_{\rho_c}/\mathsf{rt}$, k_s, and σ_s 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.

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 $\mathcal{R}_{\text{shuffle}}$.[3] This transformation clearly guarantees unlinkability of different Randomize messages of the same client. Moreover, verifiable correctness is still guaranteed, which can be seen intuitively as follows. First, observe that, since pk_i is included

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

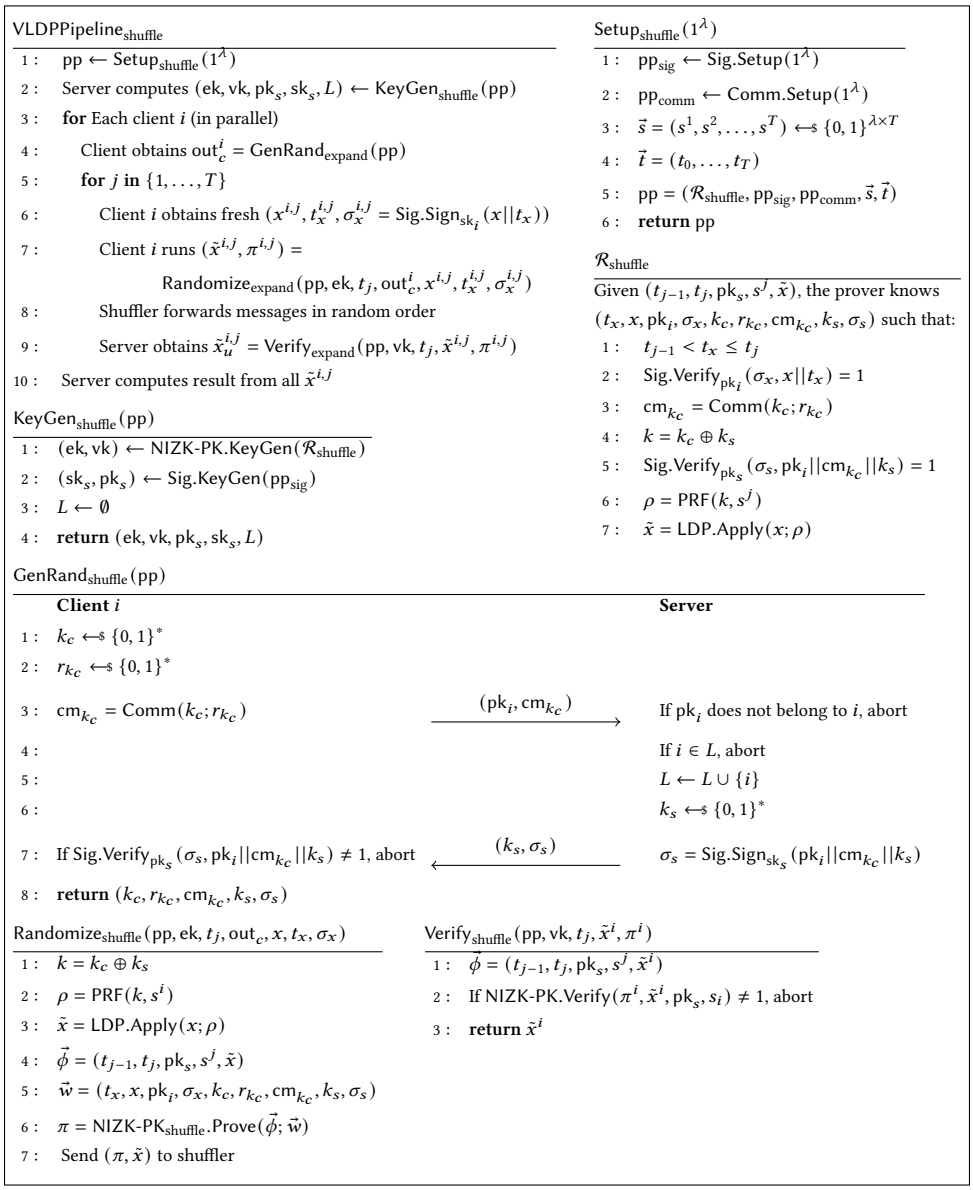

Figure 4: Shuffle **scheme for efficient VLDP in the shuffle model between one server and multiple clients.**

in σ_s, and we verify σ_s inside the NIZK-PK for $\mathcal{R}_{\text{shuffle}}$, the client has to use a pair (x, t_x), signed by its own trusted environment. Second, the inclusion of pk_i inside σ_s also guarantees that the randomness is bound to a specific client, and thus a set of colluding clients could not interchange their random values.

The transformation as described above is secure in the shuffle model, however we can still make some optimizations to reduce the computational cost for the client. We note that by moving the verification of $\rho_s = \text{PRF}(k_s||s^i)$ to the NIZK-PK, we can remove the Merkle tree. This is done by having both the client and server generate a random value for the PRF seed, respectively k_s and k_c. The full PRF seed is defined as $k = k_c \oplus k_s$. Next, we compute a random value $\rho = \text{PRF}(k, s^i)$ 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 [26].

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 the experiment setup and results we obtained to support our efficiency and practicality claims.

7.1 Implementation

Each scheme was implemented in Rust using the Arkworks v0.4 library [3]. This library provides efficient implementations for zk-SNARK schemes and other cryptographic primitives with gadgets to evaluate these primitives inside a zk-SNARK proof circuit. We used the following instantiations for the cryptographic building blocks used in our constructions, targeting 128-bit security:

- NIZK-PK: The *Groth16* zk-SNARK [25] 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. We note that 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 does not collude with the clients. The zk-SNARK elements are chosen to be on the *BLS12-381* elliptic curve (EC) [12], 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* [13], which allows for efficient, secure evaluation of EC-primitives inside zk-SNARK circuits.
- Sig: The signature schemes used by client and server are both implemented using Schnorr signatures [39]. Specifically, we use EC-Schnorr signatures over the Jubjub curve, due to its efficient verification inside a zk-SNARK circuit [40]. Additionally, we use the Blake2s-256 collision resistant hash (CRH) [37] to hash the input message to a fixed length digest. This CRH was chosen due to its good security (128 bits against collision attacks), and efficient use inside a zk-SNARK.
- Comm: Our commitment scheme is instantiated using Pedersen vector commitments [36] (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 [37], which gives a PRF output of 256 bits, or 32 bytes.[4] 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 [26] 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.

For all primitives we make use of the implementations provided in Arkworks, where we note that the implementation of Blake2s in Arkworks is provided by RustCrypto [41]. Our open-source code was written in such a way that it is simple to change the currently used primitives by other primitives of ones choosing, as long as they are compatible with the Arkworks framework. In Appendix F.3, we discuss alternative choices with respect to the building blocks we used in our implementation, and how these would influence security, efficiency, and practicality.

7.2 Experiment 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.

7.2.1 Datasets. We use two datasets in our experiments. The first dataset is based on the *Geolife GPS Trajectory dataset*, from which we extracted a dataset of 182 users with locations for 5 days divided over 8 bins. The second *Smart Meter Dataset* contains real-valued smart-meter reading from 5,567 households. For our experiments, we consider 5 days of readings from this dataset. The first dataset uses the LDP algorithm for histograms, whereas the second uses the one for reals (see Figure 1). Both datasets are described in more detail in Appendix F.1.

7.2.2 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 both the GenRand and Randomize/Verify 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 a stable Rust 1.77.2.

7.3 Concrete applications

For both datasets (and corresponding use cases) the timestamp is encoded using one byte. Next to this, we use 8 bytes (64 bits) for each random value we sample. By using 64 bits to sample a random value from a Bernoulli or Discrete (with $k \ll 2^{64}$) distribution, we sample from approximated distributions that are statistically close to the true distributions. 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 ($|\rho| = 24$), 8 bytes each for the two Bernoulli distributions and another 8 bytes for the uniform distribution. Both are below the 32 bytes that we get as output from one PRF evaluation. We expect this to be the case for most real-world applications of LDP. Nonetheless, our general experiments in the next section evaluate the computation times and message sizes for larger randomness requirements. The performance of our schemes is not impacted by any particular value of ϵ and δ used in the DP mechanism. For

[4] Recall that we can evaluate the PRF at more than one point in situations where more than 32 bytes of randomness is required for evaluating LDP.Apply.

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 metrics of all schemes for two real-world applications: client and server computation and communication costs of a *single* evaluation of an algorithm/protocol and 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 applications, over all time steps ($T = 5$) (and for the server also over all clients).

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, we note that for the Expand scheme we used a Merkle tree of depth $d_{\mathsf{MT}} = 4$, i.e., it has $2^{4-1} = 8$ leaves, since we run both datasets for $T = 5$ time steps.

Results. We show the median computation times and message sizes for a single run of each algorithm/protocol 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, several megabytes (MB), and needs to be communicated with each client. Fortunately, its generation is part of the setup and can be communicated as part of the public parameters long before the actual protocol evaluation starts. 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 each individual 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 $96T = 480$ bytes. 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 $64 + 200T = 1,064$ bytes.

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 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 the results shown in Table 2 do not contain any optimizations on the server side, e.g. parallelization. Hence, the server's runtime 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 t_j, instead of once overall, Base has T times the communication cost of the other schemes for GenRand.

7.4 General performance

The amount of randomness required for LDP.Apply will have the largest influence on the computation costs. Thus, 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. Next to this, we investigate the performance of the Expand scheme for different Merkle tree depths. We will vary d_{MT} between 2 and 11, i.e., from 2 to 1,024 leaves, which should be more than sufficient in realistic settings (a total of 1,024 sequential compositions of the LDP/shuffle mechanisms). In all experiments, we use randomly generated data, and encode the timestamps and input values using 8 bytes, which is similar to byte sizes that are often used in practice.

Results. First, we observe that the communication size is independent of both $|\rho|$ and d_{MT}, and thus only look at the influence on the runtime. Figure 5 shows the relation between the number of constraints and the computation times for the phases that are influenced by changing $|\rho|$. We observe that 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. With 1,024 bytes, one could sample 128 random values with the same statistical distance ($\sim 2^{-64}$) that we have, or 64 with half that distance. Note that increasing k will at most require one or two extra random bytes per random sampling step, to maintain the same precision. 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 it will not form

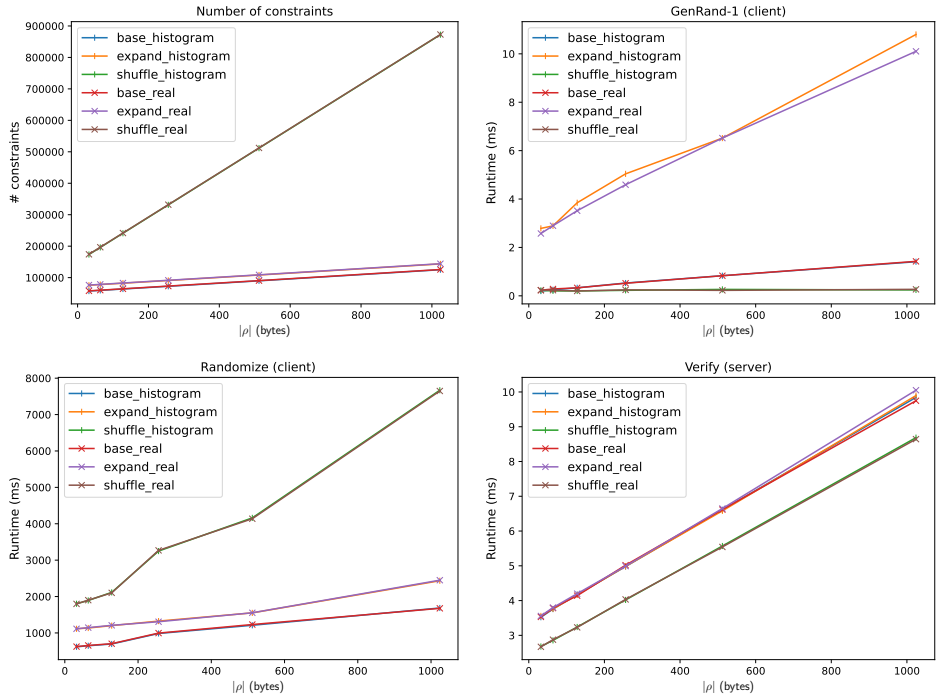

Figure 5: Graphs detailing the influence of $|\rho|$ on the number of constraints (topleft), runtime for the client in GenerateRandomness **(topright) &** Randomize **(bottomright), and runtime for the server in** Verify **(bottomleft).**

a limiting factor. For the Expand scheme, we observe a linear increase of the number of constraints and an exponential increase of the duration of GenRand in d_{MT} (Figure 14 in Appendix F.2). This clearly 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.

7.5 Comparison

In conclusion, we see that the total runtime of each of our schemes 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 protocol puts a comparatively high communication and computation load on the server, and in all but some cases performs worse than Expand.

For comparison with related work, we consider [33] 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 [33] will also come with its own latency and scalability issues, which our scheme does not suffer from. Moreover, their scheme does not consider authenticated inputs, nor does it work in the shuffle model. On top of that, it is only evaluated for binary RR, which is much simpler than our construction. Finally, [33] does not discuss the performance of their approach to GenRand. However, as it is similar in nature to $\mathsf{GenRand}_{\mathsf{base}}$, it will suffer from the same drawbacks when compared to Expand and Shuffle.

8 Conclusion

In this work, 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 believe that our schemes can be efficiently implemented for a wide variety of LDP algorithms, due to the generic design of our schemes and the capabilities of modern NIZK-PK schemes, such as zk-SNARKs.

References

[1] 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

[2] 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

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

[4] 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

[5] 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

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

[7] 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

[8] 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 (SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 441–459. https://doi.org/10.1145/3132747.3132769

[9] 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

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

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

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

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

[14] 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

[15] 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

[16] 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

[17] 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

[18] 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

[19] 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

[20] 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

[21] 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

[22] 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

[23] 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

[24] 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. https://www.usenix.org/conference/usenixsecurity21/presentation/grassi

[25] 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

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

[27] 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.

[28] 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

[29] 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.

[30] 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 (SEC '23)*. USENIX Association, USA, 1739–1756.

[31] Frank McSherry and Kunal Talwar. 2007. Mechanism Design via Differential Privacy. In *48th Annual IEEE Symposium on Foundations of Computer Science (FOCS 2007), October 20-23, 2007, Proceedings*. IEEE Computer Society, Providence, RI, USA, 94–103. https://doi.org/10.1109/FOCS.2007.41

[32] 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

[33] 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

[34] 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

[35] 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

[36] 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

[37] 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

[38] 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

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

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

[41] The RustCrypto Project Developers. 2015. RustCrypto: BLAKE2. https://github.com/RustCrypto/hashes/tree/master/blake2

[42] 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. https://www.scitepress.org/PublicationsDetail.aspx?ID=HUXWvgGpS1M=&t=1

[43] 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. https://www.usenix.org/conference/usenixsecurity22/presentation/wu-yongji

A Formal definitions for NIZKs

A NIZK scheme can be formally defined as follows.

Definition 7 (NIZK)**.** We say that (Setup, Prove, Verify, Sim) is a NIZK scheme with security parameter λ for a relation $\mathcal{R}$ if it has the following properties:
Completeness: Given a true statement, an honest prover should be able to convince an honest verifier, i.e., $\forall(x, w) \in \mathcal{R}$:

$$\Pr[\mathsf{Verify}(\mathsf{vk}, x, \pi) \neq 1 | (\mathsf{ek}, \mathsf{vk}, \mathsf{trap}) \leftarrow \mathsf{Setup}; \pi \leftarrow \mathsf{Prove}(\mathsf{ek}, x, w)] \leq \mathsf{negl}(\lambda).$$

Soundness: If the statement is false, no prover should be able to convince the verifier that it is true, i.e., for all p.p.t. adversaries $\mathcal{A}$:

$$\Pr[\mathsf{Verify}(\mathsf{vk}, x, \pi) = 1 \wedge \forall w : (x, w) \notin \mathcal{R} \mid (\mathsf{ek}, \mathsf{vk}, \mathsf{trap}) \leftarrow \mathsf{Setup}; (x, \pi) \leftarrow \mathcal{A}(\mathsf{ek}, \mathsf{vk})] \leq \mathsf{negl}(\lambda).$$

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, i.e., for all $(x, w) \in \mathcal{R}$:

$$\{(\mathsf{ek}, \mathsf{vk}, \mathsf{trap}, x, \pi) | (\mathsf{ek}, \mathsf{vk}, \mathsf{trap}) \leftarrow \mathsf{Setup}; \pi \leftarrow \mathsf{Prove}(\mathsf{ek}, x, w)\} \overset{c}{\equiv} \{(\mathsf{ek}, \mathsf{vk}, \mathsf{trap}, x, \pi) | (\mathsf{ek}, \mathsf{vk}, \mathsf{trap}) \leftarrow \mathsf{Setup}; \pi \leftarrow \mathsf{Sim}(\mathsf{trap}, x)\}$$

A NIZK-PK scheme requires the same properties as a NIZK scheme, but additionally also requires knowledge soundness.

Definition 8 (NIZK-PK)**.** We say that (Setup, Prove, Verify, Sim) is a NIZK-PK scheme with security parameter λ for a relation $\mathcal{R}$ if it is a NIZK that additionally satisfies *knowledge soundness*:
There exists an extractor $\mathcal{E}_{\mathcal{A}}$ that can produce a valid witness given complete access to the adversary's $\mathcal{A}$ state, i.e., for all p.p.t. adversaries $\mathcal{A}$, there exists a p.p.t. extractor $\mathcal{E}_{\mathcal{A}}$ such that:

$$\Pr[\mathsf{Verify}(\mathsf{vk}, x, \pi) = 1 \wedge (x, w) \notin \mathcal{R} \mid (\mathsf{ek}, \mathsf{vk}, \mathsf{trap}) \leftarrow \mathsf{Setup}; (x, \pi) \leftarrow \mathcal{A}(\mathsf{ek}, \mathsf{vk}); w \leftarrow \mathcal{E}_{\mathcal{A}}(\mathsf{ek}, \mathsf{vk}, \mathsf{trap}, x)] \leq \mathsf{negl}(\lambda).$$

B Expand protocol

The precise definition of the Expand scheme is shown in Figure 6.

C Appendix to Section 4

C.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.

C.2 LDP inside NIZK

As discussed in Section 4, we should be able to perform random sampling given a fixed number of random bits.

Example. To better explain this requirement, 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 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. Subsequently, we determine 3 intervals: $[0, \lfloor 2^\ell/3 \rfloor)$, $[2^\ell/3, 2 \cdot \lfloor 2^\ell/3 \rfloor)$, and $[2 \cdot \lfloor 2^\ell/3 \rfloor, 2^\ell - 1]$, and 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 $2^\ell - 3 \cdot \lfloor 2^\ell/3 \rfloor \leq 2$ additional elements. Thus, for sufficiently large ℓ, the distribution generated

[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.

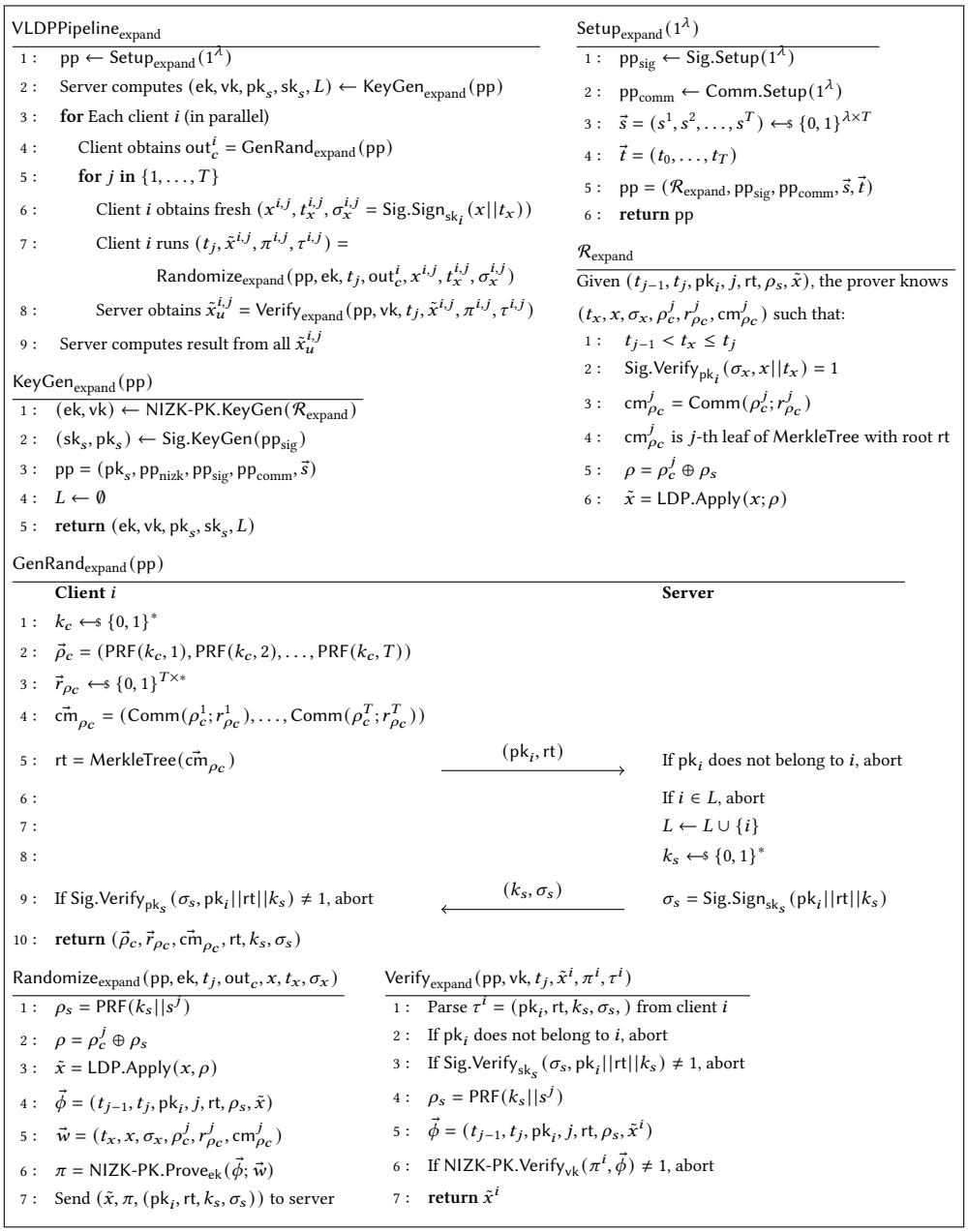

Figure 6: Expand **scheme for VLDP between one server and multiple clients, requiring only round of** GenRand **per client.**

by this sampling method is statistically close to the true distribution we wish to sample from.

Our approximate sampling methods. Such approximations can be easily defined for most well-known distributions. For the LDP algorithms defined in Figure 1, we only need to approximate the Bernoulli distribution and the Discrete Uniform distribution. Figure 8 shows two algorithms for sampling from these distributions.

It is evident that these sampling methods match our requirements, and moreover are also statistically close to the true distributions, with the statistical distance decreasing exponentially in ℓ. Using these approximate distributions, we define our approximate LDP algorithms as in Figure 7.

Clearly, for sufficiently large bitsizes of all ρ_j, these algorithms are statistically close approximations of the true LDP algorithms. Moreover, the approximation error decreases exponentially in the bitsize of the different ρ_j.

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)	1 : Split ρ into (ρ_1, ρ_2)
$\overline{x} \leftarrow \lfloor xk \rfloor + \widetilde{\text{Ber}}(xk - \lfloor xk \rfloor; \rho_1)$	2 : $b \leftarrow \widetilde{\text{Ber}}(\gamma; \rho_1)$
$b \leftarrow \widetilde{\text{Ber}}(\gamma; \rho_2)$	3 : **if** $b = 0$ **do**
if $b = 0$ **do**	4 : $\tilde{x} \leftarrow \overline{x}$
$\tilde{x} \leftarrow \overline{x}$	5 : **else**
else	6 : $\tilde{x} \leftarrow \widetilde{\text{Unif}}([1,k]; \rho_2)$
$\tilde{x} \leftarrow \widetilde{\text{Unif}}([0,k]; \rho_3)$	7 : **return** $\tilde{x}$
return $\tilde{x}$	

Figure 7: Approximate LDP algorithms for reals and histograms.

$\widetilde{\text{Ber}}(\gamma;\rho)$	$\widetilde{\text{Unif}}([lb,ub];\rho)$
input: $\gamma \in [0,1], \rho \in \{0,1\}^\ell$	**input:** $lb, ub \in \mathbb{Z} : lb < ub, \rho \in \{0,1\}^\ell$
Interpret ρ as an integer	Interpret ρ as an integer
if $\rho \leq \lfloor \gamma \cdot (2^\ell - 1) \rfloor$	$\Delta \leftarrow \lfloor 2^\ell / (ub - lb + 1) \rfloor$
$b \leftarrow 1$	**for** j **in** $\{0, \ldots, ub - lb - 1\}$
else	**if** $j \cdot \Delta \leq \rho < (j+1) \cdot \Delta$
$b \leftarrow 0$	**return** $lb + j$
return b	**return** ub

Figure 8: Algorithms for approximately sampling from the Bernoulli and Discrete Uniform distribution.

D Implementing the Shuffler

In practice a trusted shuffler can be implemented in a number of ways. One way to do this is by means of 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 [15] to realize untraceable e-mail and can be implemented in a variety of ways. An overview can be found in, e.g., [38]. 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 [8] for an implementation of verifiable, oblivious shuffling using trusted hardware.

In conclusion, following also the discussion in [8], 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 would work with any implementation of the shuffler.

E 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.

E.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 $\tilde{x}$, with accompanying NIZK-PK proof π, and public values τ_x, generated by an honest client, is accepted by an honest server, even when an adversary $\mathcal{A}$ chooses the client's inputs (x, t_x).[7]

$\mathbf{Exp}^{\text{Comp}}_{\mathcal{A}}(1^\lambda, n, T)$
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 11) guarantees that no malicious, possibly colluding, clients are able to return a value $\tilde{x}$ that is

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

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

$\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$
$\{\mathsf{sk}_i, \mathsf{pk}_i\}_i \leftarrow \mathsf{Sig.KeyGen}(\mathsf{pp})$
$(\mathsf{ek}, \mathsf{vk}, \mathsf{pk}_s, \mathsf{sk}_s, L, \mathsf{trap}) \leftarrow \mathcal{A}(\mathsf{pp})$
$\mathsf{out}^i_c \leftarrow \mathsf{GenRand}^{\mathcal{A}}(\mathsf{pp}, \mathsf{aux})$
$\{t^{i,j}_x\}_{i,j} \leftarrow \mathcal{A}(\mathsf{pp}, \mathsf{ek}, \mathsf{vk}, \mathsf{pk}_s, \mathsf{sk}_s, L)$
$\sigma^{i,j}_x \leftarrow \mathsf{Sig.Sign}_{\mathsf{sk}_i}(x^{i,j} || t^{i,j}_x)$
$\{(\tilde{x}^{i,j}, \pi^{i,j}, \tau^{i,j}_x)\}_{i,j} \leftarrow \mathsf{Randomize}(\mathsf{pp}, \mathsf{ek}, t_j, \mathsf{out}^i_c, x^{i,j}, \sigma^{i,j}_x)$
if trap is not valid trapdoor for $(\mathcal{R}, \mathsf{ek}, \mathsf{vk})$
  $\vee\ \exists i \in [n], j \in [T] : t^{i,j}_x \leq t_{j-1} \vee t^{i,j}_x > t_j$
  return $\perp$
$\mathcal{A} \leftarrow \{\tilde{x}^{i,j}, \pi^{i,j}, \tau^{i,j}\}_{i,j}$
return $(\mathsf{view}_{\mathcal{A}}, \{\tilde{x}^{i,j}\}_{i,j})$

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

$\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$
$\{\mathsf{sk}_i, \mathsf{pk}_i\}_i \leftarrow \mathsf{Sig.KeyGen}(\mathsf{pp}, \{\mathsf{pk}_i\}_i)$
$(\mathsf{ek}, \mathsf{vk}, \mathsf{pk}_s, \mathsf{sk}_s, L, \mathsf{trap}) \leftarrow \mathcal{A}(\mathsf{pp}, \{\mathsf{pk}_i\}_i)$
$\mathsf{out}^i_c \leftarrow \mathcal{S}^{\mathcal{A}}_1(\mathsf{pp}, \mathsf{pk}_s)$
$\{t^{i,j}_x\}_{i,j} \leftarrow \mathcal{A}(\mathsf{pp}, \mathsf{ek}, \mathsf{vk}, \mathsf{pk}_s, \mathsf{sk}_s, L)$
$\{(\pi^{i,j}, \tau^{i,j}_x)\}_{i,j} \leftarrow \mathcal{S}_2(\mathsf{pp}, \mathsf{ek}, \mathsf{trap}, t_j, \mathsf{out}^i_c, y^{i,j})$
if trap is not valid trapdoor for $(\mathcal{R}, \mathsf{ek}, \mathsf{vk})$
  $\vee\ \exists i \in [n], j \in [T] : t^{i,j}_x \leq t_{j-1} \vee t^{i,j}_x > t_j$
  return $\perp$
$\mathcal{A} \leftarrow \{\tilde{y}^{i,j}, \pi^{i,j}, \tau^{i,j}\}_{i,j}$
return $(\mathsf{view}_{\mathcal{A}}, \{y^{i,j}\}_{i,j})$

Figure 10: Experiments for the zero-knowledge definition.

not an honest evaluation of $\mathsf{LDP.Apply}(x; \rho)$, for a truly random, independently sampled ρ. In this experiment, the adversary is allowed to choose t_x and controls all clients, who may deviate from the protocol arbitrarily. The goal of the adversary is to let the server accept a tuple $(\tilde{x}, \pi, \tau_x)$, where $\tilde{x}$ is not honestly computed.

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

$\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$
$\{\mathsf{sk}_i, \mathsf{pk}_i\}_i \leftarrow \mathsf{Sig.KeyGen}(\mathsf{pp})$
$(\mathsf{ek}, \mathsf{vk}, \mathsf{pk}_s, \mathsf{sk}_s, L, \{\mathsf{pk}_i\}_i) \leftarrow \mathsf{KeyGen}(\mathsf{pp})$
$\{t^{i,j}_x\}_{i,j} \leftarrow \mathcal{A}(\mathsf{pp}, \mathsf{ek}, \mathsf{vk}, \mathsf{pk}_s, \{x^{i,j}\}_{i,j})$
if $\exists i \in [n], j \in [T] : t^{i,j}_x \leq t_{j-1} \vee t^{i,j}_x > t_j$
  return $\{\perp\}_{i,j}$
$\sigma^{i,j}_x \leftarrow \mathsf{Sig.Sign}_{\mathsf{sk}_i}(x^{i,j} || t^{i,j}_x)$
$\{(\tilde{x}^{i,j}, \pi^{i,j}, \tau^{i,j}_x)\}_{i,j} \leftarrow \mathcal{A}^{\mathcal{S}^*}(\mathsf{pp}, \mathsf{ek}, \mathsf{vk}, \mathsf{pk}_s, \{x^{i,j}, t^{i,j}_x, \sigma^{i,j}_x, \mathsf{pk}_i\}_{i,j})$
return $\{\mathsf{Verify}(\mathsf{pp}, \mathsf{vk}, t_j, \tilde{x}^{i,j}, \pi^{i,j}, \tau^{i,j}_x)\}_{i,j}$

Figure 11: Experiment for soundness definition.

The experiments for zero-knowledge (Figure 10), 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 (portrayed by the environment). In Zk-sim, $\mathcal{A}$ acts as a server also, but instead interacts with a simulator $\mathcal{S}$. This simulator pretends to be an honest client, and should be able to generate messages with the same distribution as an actual client would, but does not have access to the input values (x, t_x). The adversary wins this game, if it is able to distinguish between the different worlds.

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.

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

$\mathsf{pp} \leftarrow \mathsf{Setup}(1^\lambda)$
$\{\mathsf{sk}_i, \mathsf{pk}_i\}_i \leftarrow \mathsf{Sig.KeyGen}(\mathsf{pp})$
$(\mathsf{ek}, \mathsf{vk}, \mathsf{pk}_s, \mathsf{sk}_s, L, \mathsf{trap}) \leftarrow \mathcal{A}(\mathsf{pp}, \{\mathsf{pk}_i\}_i)$
$\mathcal{A}(\mathsf{pp}) \rightarrow t_j$
for $i \in \{0, 1\}$
  $\mathsf{GenRand}^{\mathcal{A}}(\mathsf{pp}, t_j) \rightarrow \mathsf{out}^i_c$
$\mathcal{A}(\mathsf{pp}) \rightarrow (x, t_x)$
$b \leftarrow\!\$\ \{0, 1\}$
$\mathsf{Sig.Sign}_{\mathsf{sk}_b}(x, t_x) \rightarrow \sigma^b_x$
$\mathsf{Randomize}(\mathsf{pp}, \mathsf{ek}, t_j, \mathsf{out}^b_c, x, t_x, \sigma^b_x) \rightarrow (\tilde{x}^b, \pi^b, \tau^b_x)$
$\mathcal{A}(\tilde{x}^b, \pi^b, \tau^b_x) \rightarrow b'$
if trap is not valid trapdoor for $(\mathcal{R}, \mathsf{ek}, \mathsf{vk})$
  $\vee\ \exists i \in [n], j \in [T] : t^{i,j}_x \leq t_{j-1} \vee t^{i,j}_x > t_j$
  return $\perp$
return $b' = b$

Figure 12: Experiment for shuffle indistinguishability.

E.2 Proofs

We focus on a full proof for the Shuffle scheme, as this is our main result, and the proofs for the other schemes are analogous. Then, for brevity, we describe proof sketches for the other schemes, highlighting the differences with the proof for the Shuffle scheme.

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 $\mathsf{Randomize}_{\mathsf{shuffle}}$ procedure outputs a valid NIZK-PK proof π for $\mathcal{R}_{\mathsf{shuffle}}$. It therefore suffices to show that the proof for $\mathcal{R}_{\mathsf{shuffle}}$ 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, due to the fact that our NIZK-PK scheme is complete itself, i.e., $\text{Verify}_{\text{shuffle}} \neq \perp$, except for some probability that is $\text{negl}(\lambda)$. Since, i and j were picked arbitrarily, and both T and n are $\text{poly}(\lambda)$, we can conclude that the total probability is also $\text{negl}(\lambda)$.

(2) Soundness: In order to prove soundness, we will describe a series $\mathbf{Exp}_0$ to $\mathbf{Exp}_3$ of hybrid experiments, where $\mathbf{Exp}_0$ is equal to $\mathbf{Exp}^{\text{Snd-Real}}_{\mathcal{A},\mathcal{S}^*}(1^\lambda, n, T, \{x^{i,j}\}_{i,j})$ (from Definition 4 as defined in Figure 11) 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 $\mathcal{VLDP}_{\text{shuffle}}$ is sound.

$\mathbf{Exp}_0$: The original experiment $\mathbf{Exp}^{\text{Snd-Real}}_{\mathcal{A},\mathcal{S}^*}(1^\lambda, n, T, \{x^{i,j}\}_{i,j})$.

$\mathbf{Exp}_1$: 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 $\mathcal{R}_{\text{shuffle}}$ 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:

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

$\mathbf{Exp}_2$: 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 $\text{fail}_2 = \text{true}$. 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 $\Pr[\mathbf{Fail}_2] \leq \text{negl}(\lambda)$.

$\mathbf{Exp}_3$: This is the same experiment as $\mathbf{Exp}_2$, except that the server S^* now also maintains a list R of entries $(i, (\text{pk}_i, \text{cm}^i_{k_c}, k^i_s))$. These entries correspond to messages $(\text{pk}_i, \text{cm}_{k_c})$ received during occurrences of the $\text{GenRand}_{\text{shuffle}}$ 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 $\text{GenRand}_{\text{shuffle}}$. Now, if fail_2 has not been set, the server asserts, in $\text{Verify}_{\text{shuffle}}$, that R indeed contains an entry $(\star, (\text{pk}_i, \text{cm}^i_{k_c}, k^i_s))$, where $(\text{pk}_i, \text{cm}^i_{k_c}, k^i_s)$ are the corresponding elements of $\vec{w}^{i,j}$. If this is not the case, we set $\text{fail}_3 = \text{true}$. 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 $\mathbf{Fail}_3$ can occur, is if the client can produce a tuple $(\text{pk}_i, \text{cm}^i_{k_c}, k^i_s)$ with signature σ^i_s, such that $\text{Sig.Verify}_{\text{pk}_s}(\sigma^i_s, \text{pk}_i || \text{cm}^i_{k_c} || k^i_s) = 1$. $\mathcal{S}^*$ will only have generated one signature for each pk_i, due to step 4 in $\text{GenRand}_{\text{shuffle}}$, and this tuple is on R. Therefore, for $\mathbf{Fail}_3$ to occur, the client must have forged a signature on a tuple $(\text{pk}'_i, \text{cm}^{i\prime}_{k_c}, k^{i\prime}_s)$, where at least one element differs from $(\text{pk}_i, \text{cm}^i_{k_c}, k^i_s)$. 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 $\Pr[\mathbf{Fail}_3] \leq \text{negl}(\lambda)$.

Finally, we observe that if Fail_3 does not occur, we are essentially at a point where $\tilde{x}^{i,j} = \text{LDP.Apply}(x^{i,j}; \rho^i)$, where $\rho^i = \text{PRF}(k^i_c \oplus k^i_s, s^j)$. 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 cm_{k_c} is fixed before k_s is chosen uniformly at random, $k_c \oplus k_s$ is also be a uniform random bitstring, and by the definition of a secure PRF, ρ will also be distributed at random, except with negligible probability. Therefore it follows that the following probability is negligible in λ:

$$|\Pr[\mathbf{Exp}_3] - \Pr\left[\text{LDP.Apply}(x^{i,j}; \rho^{i,j}) = \{y^{i,j}\}_{i,j} \middle| \rho^{i,j} \leftarrow\!\!\$\ \{0,1\}^*\right]|.$$

(3) Zero-knowledge: To show that $\mathcal{VLDP}_{\text{shuffle}}$ has 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 behave like an honest-but-curious adversary, and thus follows the protocol. The first simulator algorithm $\mathcal{S}_1$, simulates a client in $\text{GenRand}_{\text{shuffle}}$, thereby also interacting with $\mathcal{A}$, who represent the server. The second simulator algorithm $\mathcal{S}_2$, simulates the $\text{Randomize}_{\text{shuffle}}$ algorithm.

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 $\text{GenRand}_{\text{shuffle}}$. 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

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

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

the real world and the simulator. Since NIZK-PK is a secure scheme for $\mathcal{R}_{\text{shuffle}}$ 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.

$\mathbf{Exp}_0$: This is the original experiment $\mathbf{Exp}^{\text{Sh-Ind}}_{\mathcal{A}}$ in Definition 6.

$\mathbf{Exp}_1$: 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 bit-string 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

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

□

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. □

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 $\mathsf{GenRand}_{\mathsf{base}}$, 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. □

F Appendix to Section 7

F.1 Datasets

Below, we give more detail on the datasets used to evaluate the performance of our schemes on concrete, realistic applications.

Geolife GPS Trajectory Dataset. 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 very 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 few 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 histogram 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).

Smart Meter Dataset. 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 took part in 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 which 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] See 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

F.2 Additional graphs

Figure 14 displays additional graphs showing the impact on the number of constraints, and runtime of the GenRand and Randomize routines as we increase the depth d_{MT} of the Merkle tree used in the Expand scheme. Details are discussed in Section 7.4.

F.3 Discussion on possible optimizations and alternatives

While the experiments in Section 7 show the practicality of our schemes, certain optimizations and/or alternative 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 [24], MiMC [1], and Pedersen [26] hashes. These schemes can reduce prover times by as much as a factor of 10 [40]. However, this often comes at the cost of reduced security. Poseidon and MiMC are still very 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 [23]. 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.

On the other hand, 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 scheme 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

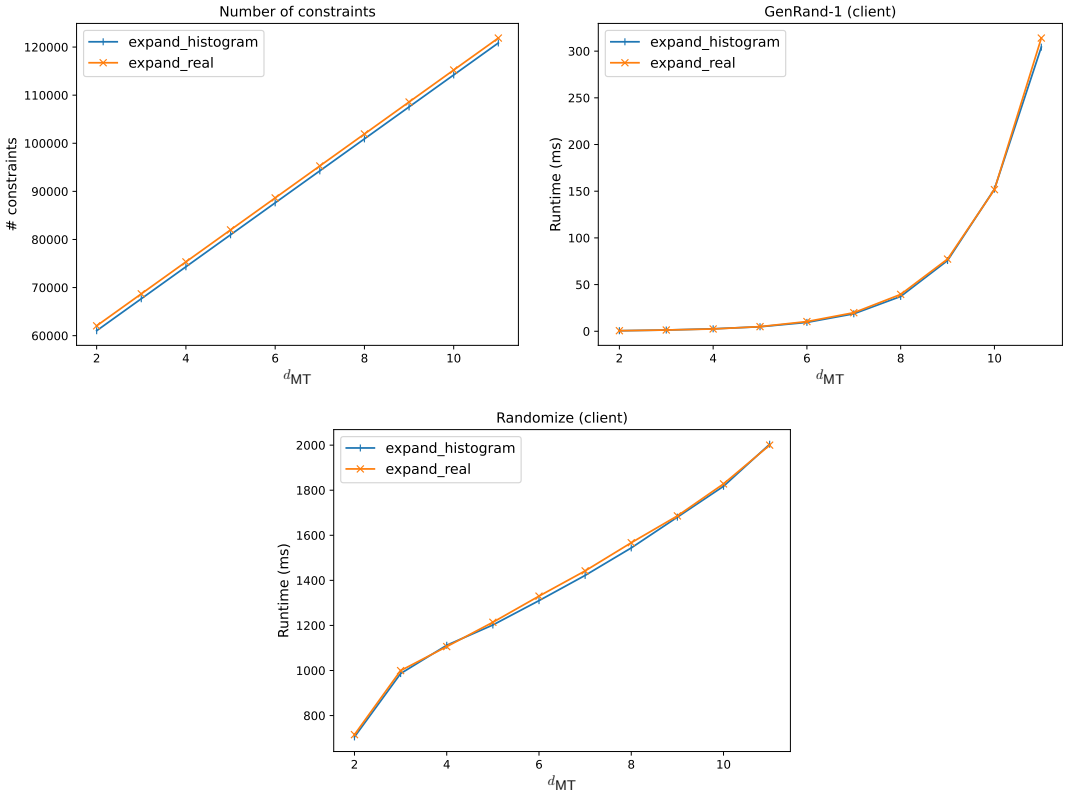

Figure 14: Influence of d_{MT} on the number of constraints (topleft), client runtime for GenRand **(topright) and** Randomize **(bottom).**

hash functions. However, in cases where such primitives are available, they could be used to improve effiency 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 already very efficient by itself. E.g., it is around 2-2.5x times faster than EdDSA inside a zk-SNARK circuit [40].

zk-SNARK scheme. 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 might be a significant drawback in practice.

Next to this, one could also consider using an alternative scheme to Groth16. For example, schemes with a *universal setup* (e.g., Marlin [18], *post-quantum security* (e.g., Fractal [19]) or a *transparent setup* (e.g., Fractal [19] or SuperSonic [14]) 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, due to the fact the 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.