Secure Noise Sampling for DP in MPC with Finite Precision[*]

Hannah Keller
hkeller@cs.au.dk
Aarhus University
Denmark

Helen Möllering
moellering@encrypto.cs.tu-darmstadt.de
McKinsey & Company
Germany

Thomas Schneider
schneider@encrypto.cs.tu-darmstadt.de
Technical University of Darmstadt
Germany

Oleksandr Tkachenko
oleksandr.tkachenko1@gmail.com
DFINITY Foundation
Germany

Liang Zhao
liang.zhao@tu-darmstadt.de
Technical University of Darmstadt
Germany

ABSTRACT

While secure multi-party computation (MPC) protects the privacy of inputs and intermediate values of a computation, differential privacy (DP) ensures that the output itself does not reveal too much about individual inputs. For this purpose, MPC can be used to generate noise and add this noise to the output. However, securely generating and adding this noise is a challenge considering real-world implementations on finite-precision computers, since many DP mechanisms guarantee privacy only when noise is sampled from continuous distributions requiring infinite precision.

We introduce efficient MPC protocols that securely realize noise sampling for several plaintext DP mechanisms that are secure against existing precision-based attacks: the discrete Laplace and Gaussian mechanisms, the snapping mechanism, and the integer-scaling Laplace and Gaussian mechanisms. Due to their inherent trade-offs, the favorable mechanism for a specific application depends on the available computation resources, type of function evaluated, and desired (ϵ, δ)-DP guarantee.

The benchmarks of our protocols implemented in the state-of-the-art MPC framework MOTION (Braun et al., TOPS'22) demonstrate highly efficient online runtimes of less than 32 ms/query and down to about 1ms/query with batching in the two-party setting. Also the respective offline phases are practical, requiring only 51 ms to 5.6 seconds/query depending on the batch size.

KEYWORDS

Secure Multi-party Computation, Differential Privacy, Noise Sampling, Secure Implementations, Finite-Precision Computing

1 MOTIVATION

The aggregation and statistical analysis of many individuals' data became common across multiple industries, e.g., for the detection of financial frauds [10], the improvement of disease diagnostics [29], or the matching of organ donors with patients [17]. Such analyses may, however, conflict with data privacy. To complicate the matter, data is often stored at different locations, e.g., hospitals or banks, and those parties do not wish or are legally not permitted to share their information with the other parties. However, collaboration is imperative for effectively gaining new insights from distributed data. Secure multi-party computation (MPC) [6, 21, 36] offers a cryptographic approach to execute an analysis on data in a privacy-preserving manner, even when the data is distributed among multiple parties. MPC guarantees that no party learns more than what is revealed by the published analysis output.

MPC protocols usually yield exact results, but provide no protection against attacks that use these outputs to make inferences about the individuals that contributed their data to the analysis. Without additional output privacy measures, published results may be vulnerable to multiple attacks ranging from simple linkage and reconstruction attacks [81, 92] to inference and model inversion attacks [49, 58, 91] in machine learning applications. In many of these scenarios, an adversary may use publicly available information to fully reconstruct the input data used for the analysis. Differential privacy (DP) [43] has emerged as the de facto standard for guaranteeing that the published output of a function does not reveal too much information about its input. In fact, companies such as Google [97] and Apple [93], as well as the US Census Bureau [37], already use DP in practice.

In many cases, DP is guaranteed by mechanisms that sample noise from some distribution and add this noise to some computed function result. When the output of the function to be released is an integer, noise can be sampled from a discrete distribution, like the discrete Laplace [52] or discrete Gaussian distribution [26]. If the function output is not an integer, many DP mechanisms require noise sampled from a continuous distribution. Representing real numbers from continuous distributions with only finite precision, as is necessary on computers, leads to a violation of the DP guarantee in general [50]. In fact, textbook Laplace and Gaussian mechanism implementations allow an attacker to recover the entire database [64, 77]. Therefore, we consider five DP mechanisms in our work that are known to guarantee DP when implemented on finite-precision computers, two that sample from discrete distributions, and three that sample from approximated versions of continuous distributions.

Those five DP mechanisms are, however, designed for a scenario with a central database which is often not available for real-world applications where data is held by multiple parties. This scenario is commonly known as the central DP model [42]. As an alternative model, local DP [68] drops this assumption, instead requiring each individual to add noise necessary for preserving DP to their own data and publish the result, after which the function is computed on this noisy published data. This approach adds significantly more noise, resulting in published results with more error and less utility.

[*] Please cite the conference version of this paper published at ARES'24 [69].

Using MPC, the trusted aggregator in the central DP model can be replaced by a distributed protocol run by multiple non-colluding parties. We focus on an outsourcing scenario, where a large number of data owners secret share their data to a small number of non-colluding parties. We realize the central DP model where the non-colluding parties jointly sample noise from a distribution under MPC with limited precision, such that DP is securely guaranteed and good utility is preserved. We choose floating-point arithmetic to implement the above five secure DP mechanisms for two reasons: (1) it satisfies the accuracy requirement of these DP mechanisms which is not the case for fixed-point arithmetic; (2) it is more efficient than the noise sampling methods of [30, 42, 46] that are based on processing a large sequence of Boolean bits when the required sampling interval is fairly large, e.g., sampling a geometric random value in $[0, 2^{128})$. Besides, existing plaintext sampling algorithms for DP mechanisms cannot simply be "translated" into an MPC protocol, so we add multiple optimizations to yield a practically efficient solution.

Outline and Contributions. After introducing the baseline techniques used in our work in §2 and providing an overview of related work in §3, we present the following contributions:

(1) *MPC-based Noise Sampling with Finite Precision.* We introduce MPC protocols for sampling random integers and floating-point values from the Laplace and Gaussian distributions in §4. Notably, our protocols are not susceptible to precision-based attacks (cf. §3), which was not considered by the MPC community so far.

(2) *MPC-Protocols for Discrete Mechanisms.* Based on our secure random sampling protocols, we design MPC protocols for the discrete Laplace and Gaussian mechanisms [26] in §4.1, which satisfy DP for queries returning integer values. We introduce several protocol-level optimizations, that can halve the circuit size and reduce its depth by $36-88\%$ depending on the employed MPC technique in §4. We reduce runtimes by up to 8.6× (cf. §5.2) compared to a naive implementation.

(3) *MPC-Protocols for Continuous Mechanisms.* Similarly, we design the first MPC protocols for the snapping and integer-scaling Laplace and Gaussian mechanisms with floating-point arithmetic. Our MPC protocols for the snapping mechanism by Mironov [77] in §4.2 and Google's integer-scaling Laplace and Gaussian mechanisms [55] in §4.3 are applicable for floating-point arithmetic.

(4) *Open-source implementation and benchmarks.* We implement our five MPC-protocols for DP mechanisms with both integer and floating-point arithmetic in the state-of-the-art MPC framework MOTION [21]. The implementation will be open-sourced upon acceptance of our work. Furthermore, we extensively benchmark all algorithms and experimentally compare them to related work in §5. All but one of our MPC-based sampling protocols are independent of the input data, so they can be entirely pre-computed in an offline phase. The online phases of our MPC-based DP mechanisms are highly efficient using batching, e.g., 1.4-9.5 ms per query with a batch size of 30 or 40 (cf. §5.2).

2 PRELIMINARIES

In this section, we summarize the concept of differential privacy and the cryptographic building blocks used in our work. We consider security from two different perspectives: First, we achieve *computational privacy* via MPC, i.e., protecting the input data X of different data owners during the computation. Second, we achieve *output privacy* via DP, i.e., the information leakage about the input data X that can be inferred from the output $f(X)$ is limited by the DP guarantees.

2.1 Output Privacy

Differential Privacy (DP). Intuitively, DP guarantees that including an individual's data record in an analysis has only a limited impact on the result of the analysis. Let $X \in D^n$ be a database represented as a vector of n entries from some domain D. Typically a domain will be of types $\{0,1\}^d$ or $\mathbb{R}^d$. The size of a database X is measured by its ℓ_1 norm: $\|X\|_1 = n$. X' is a neighboring database of X if it differs by one record, such that X' is constructed by replacing one record of X with a different record. For DP, randomized mechanisms M are used to ensure the similarity of the outputs of a statistical analysis on two neighboring databases X and X'. Differential privacy is formally defined as follows:

Definition 2.1 (Differential Privacy [42, 43]). A randomized algorithm $M : D^n \rightarrow \mathcal{Y}$ is (ε, δ)-differentially private if for any two neighboring databases $X, X' \in D^n$, and all sets $T \subseteq \mathcal{Y}$,

$$\Pr\left[M(X) \in T\right] \leq e^{\varepsilon} \cdot \Pr\left[M\left(X'\right) \in T\right] + \delta. \qquad (1)$$

Smaller ε values provide a stronger privacy guarantee. ε is chosen to be a small, non-negligible constant. δ should be much smaller than $\frac{1}{\|X\|_1}$ to prevent the leakage of complete data records.

Discrete Outputs. When functions have discrete outputs, it is possible to add noise sampled from a distribution of discrete values, avoiding precision-based attacks on some implementations of continuous distributions. Take integer outputs for example, the discrete Laplace or discrete Gaussian distributions can be used for noise addition.

Laplace and Gaussian Mechanisms. Additive Laplace [43] and Gaussian [44] noise satisfy DP by adding noise to the output of a function $f : D^n \rightarrow \mathbb{R}^k$, that maps a database $X \in D^n$ to k real numbers. Thereby, the noise's magnitude must be chosen based on the ℓ_1- and ℓ_2-sensitivities of the function f: $\Delta_1 f = \max\|f(X) - f(X')\|_1$ and $\Delta_2(f) = \max\|f(X) - f(X')\|_2$ for any two neighboring databases X and X'. I.e., it measures the maximum difference in the output that modifying a single record in a database can cause.

While the Laplace mechanism guarantees ϵ-DP, the Gaussian mechanism can only offer (ϵ, δ)-DP, a weaker guarantee. The Laplace distribution overall samples less noise than the Gaussian distribution for one-dimensional single outputs; however, the Gaussian mechanism allows for better utility when releasing high-dimensional outputs, which is relevant for machine learning applications [1, 61, 76]. Furthermore, Gaussian noise has the same distribution as much naturally occurring noise in data measurements, and several sources of Gaussian noise add nicely.

Lemma 2.2 (Laplace Mechanism [43]). *Given any function f : $D^n \rightarrow \mathbb{R}^k$ with ℓ_1-sensitivity $\Delta_1 f$ and a privacy parameter ε, the Laplace mechanism satisfies $(\epsilon, 0)$-DP and is defined as:*

$$M_{\mathsf{Lap}}(X) = f(X) + (Y_1, \ldots, Y_k), \qquad (2)$$

where Y_i are i.i.d. random values drawn from a Laplace distribution $\mathsf{Lap}(b)$ *with probability density function* $P(x \mid b) = \frac{1}{2b} e^{-\frac{|x|}{b}}$ *and* $b = \Delta_1 f/\epsilon$.

Lemma 2.3 (Gaussian Mechanism [44]). *Given any function* $f : D^n \to \mathbb{R}^k$ *with* ℓ_2*-sensitivity* $\Delta_2(f)$ *and privacy parameters* (ε, δ)*, the Gaussian mechanism is defined as:*

$$M_{\text{Gauss}}(X) = f(X) + (Y_1, \ldots, Y_k), \tag{3}$$

where Y_i are i.i.d. random values drawn from a Gaussian distribution $\mathcal{N}\left(\mu = 0, \sigma^2\right)$ *with probability density function* $P\left(x \mid \mu = 0, \sigma^2\right) = \frac{1}{\sigma\sqrt{2\pi}} \cdot e^{-(x-\mu)^2/(2\sigma^2)}$ *and* $\sigma^2 > \frac{2\ln\left(\frac{1.25}{\delta}\right) \cdot (\Delta_2(f))^2}{\varepsilon^2}$.

Distributed Computational Differential Privacy. The original definition of differential privacy holds against computationally unbounded adversaries. However, efficient MPC protocols are often only secure against computationally bounded adversaries. Hence, we use the definition of distributed or computational differential privacy introduced in [13] that considers computationally bounded adversaries.

Definition 2.4 (Distributed Computational Differential Privacy (CDP) [13]). A randomized protocol Π implemented among N computation parties $\mathcal{P} = \{\mathcal{P}_1, \ldots, \mathcal{P}_{\mathsf{N}}\}$ satisfies $(\epsilon, \delta = \mathsf{negl}(\kappa))$ computational distributed differential privacy w.r.t. a coalition $C \subset \mathcal{P}$ of semi-honest computation parties of size t, if for every probabilistic polynomial-time distinguisher D and for any neighboring databases X, X' and any possible set of views (e.g., internal state and exchanged messages of party $\mathcal{P}_i$) for protocol Π,

$$\Pr\left[D(\mathsf{VIEW}_{\Pi}^{C}(X)) = 1\right] \leq e^{\varepsilon} \cdot \Pr\left[D(\mathsf{VIEW}_{\Pi}^{C}(X')) = 1\right] + \mathsf{negl}(\kappa),$$

where κ is a security parameter for computationally bounded adversaries.

2.2 Computational Privacy

Secure Multi-Party Computation (MPC). MPC enables $\mathsf{N} \geq 2$ parties $\mathcal{P}_1, \ldots, \mathcal{P}_{\mathsf{N}}$ to securely compute any function $y = f(x_1, \ldots, x_{\mathsf{N}})$ while revealing nothing but the output y. There are two typical MPC deployment scenarios: Either, the N data owners jointly run the computation taking their data as input which is, however, kept private from each other. Alternatively, MPC can also be used in an *outsourcing* scenario [67], where M data owners generate shares of their private inputs and send the shares to the N non-colluding computing parties. Those evaluate f on the received secret shared inputs using MPC and obtain the secret shared output while learning no information. Our benchmarks in §5 use the MPC framework MOTION [21] that implements mixed-protocol MPC tolerating up to $\mathsf{N} - 1$ passively corrupted parties. MOTION combines the GMW protocol [53] in its Boolean and arithmetic versions with the constant round BMR protocol [15].

We can use the result from [13, 42, 95] to prove that a mechanism remains computationally differentially private when implemented using a computationally secure MPC protocol.

Theorem 2.5. *Define some function* $\mathcal{M} : D^n \to \mathcal{Y}$*, with input domain D and output domain* $\mathcal{Y}$*. Let* $\Pi_{\mathcal{M}} : (D^{n/m})^m \to \mathcal{Y}$ *be an m-party protocol that computationally securely computes* $\mathcal{M}$ *with security parameter* κ *and security against up to t corruptions. If* $\mathcal{M}$ *is* (ϵ, δ)*-DP, and* $\delta_\kappa = \mathsf{negl}(\kappa)$*, then* $\Pi_{\mathcal{M}}$ *is* $(\epsilon, (\exp(\epsilon) + 1)\delta_\kappa + \delta)$*-CDP with security against up to t corruptions.*

A proof sketch is given in Appendix C.

Number Representations. While the MPC protocols in MOTION [21] operate over rings ($\mathbb{Z}_{2^\ell}$, where $\ell \geq 1$), DP mechanisms require drawing random values from probability distributions [44]. To represent those, we use integer or floating-point representations.
Floating-Point Representation. According to the IEEE floating-point standard [73], a floating-point number u is represented with a sign bit S, an exponent E and significant bits $d_1, \ldots, d_p \in \{0, 1\}^p$, where $u = (-1)^S \times (1.d_1 \ldots d_p)_2 \times 2^E$. Essentially, floating-point numbers can only represent a subset of real numbers leading to limited precision for the others. A series of works [27, 57, 64, 77] present attacks against insecure implementations of plaintext DP mechanisms using floating-point representations (cf. §3).

Since we have to sample from a distribution with finite domain for MPC, we must account for an error probability when operating over rings $\mathbb{Z}_{2^\ell}$ as it is not possible to sample values greater than 2^ℓ. Therefore, with some small probability δ', a sample will lie outside the range $[0, 2^\ell)$, which we have to account for in the failure probability. The following lemma follows directly from the privacy of $\mathcal{M}$ and Definition 2.1.

Lemma 2.6. *Consider an additive noise mechanism* $\mathcal{M} = f(x) + n$*, where additive noise* $n \leftarrow \mathcal{D}$ *is sampled from a distribution that satisfies* (ϵ, δ)*-DP. Then a mechanism* $\mathcal{M}'$ *satisfies* $(\epsilon, \delta + \delta')$*-DP, where* δ' *is the probability with which* $n \leftarrow \mathcal{D}$ *is out of the range* $[0, 2^\ell)$*:*

$$\mathcal{M}'(x) = \begin{cases} f(x) + n, n \leftarrow \mathcal{D} & \text{w.p. } 1 - \delta' \\ f(x) & \text{w.p. } \delta'. \end{cases}$$

3 RELATED WORK

In this section, we discuss related works on combining DP and MPC, as well as on the privacy issues of finite-precision implementations of DP mechanisms.

Combination of DP and MPC. When multiple parties own data, DP protocols need to either be interactive, assume computationally bounded adversaries or both [33], as we consider in our work, to avoid large errors in the released output. Multiple works, e.g., [19, 30, 32, 34, 42, 45, 46, 63, 71, 83, 98] combine MPC techniques with DP to achieve a similar utility as in the central DP model while removing the requirement of a trusted aggregator. Dwork et al. [42] use secret sharing-based MPC for noise generation. Their protocols generate noise from a Gaussian distribution (approximated with a binomial distribution) and a discrete Laplace distribution (approximated with a Poisson distribution) by processing a sequence of unbiased/biased Boolean bits in MPC. Eriguchi et al. [46] improve the noise generation of [42] by reducing the communication and round complexity of the MPC protocols as well as removing the failure probability of the sampling algorithms. However, the protocols of Dwork et al. [42] and Eriguchi et al. [46] rely on Shamir's secret sharing [88], which is defined on a field. In contrast, the MPC techniques used in our work are defined on a ring, significantly improving the efficiency of our MPC-based DP protocols. Champion et al. [30] propose secure computation methods for sampling biased bits improving previous work by Dwork et al. [42]. However, their sampling domain is not large enough to support our MPC-based DP mechanism. For example, for geometric sampling, we require the sampling domain to be $[0, 2^{128})$, while a solution based on the biased bits protocols of [30] can only efficiently support a domain of size $[0, 85)$. An extension of their protocol to our required domain size would have exponential time overhead in MPC. Eigner et al. [45] propose an architecture called PrivaDA to realize the Laplace, discrete Laplace, and exponential mechanisms in MPC using a combination of floating and fixed-point arithmetic operations for efficiency reasons. Knott et al. [71] propose a machine learning framework that implements the Gaussian mechanism in MPC. Wu et al. [98] use Shamir's secret sharing [88]-based protocols that approximate the Laplace and Gaussian distributions using the central limit theorem [9]. However, Eigner et al. [45], Wu et al. [98] and Knott et al. [71] do not provide an analysis regarding whether the implementation of arithmetic operations affects the DP guarantee. In an orthogonal line of work, Pettai et al. [83] and Choquette-Choo et al. [34] present frameworks where a trusted third party generates and adds DP noise to the query result which is computed under MPC. In contrast, our protocols do not rely on a trusted third party.

Although we focus on the central DP model due to its favorable properties with respect to the accuracy, the local DP model has also been combined with cryptographic techniques such as homomorphic encryption [2, 3, 16, 31, 54, 65, 84, 89, 90, 94], functional encryption [99, 101], authenticated encryption [14, 25], secret sharing [59], or other MPC techniques [66] to enhance data privacy.

Attacks against DP Mechanisms. The privacy of many DP mechanisms is based on two implicit assumptions [44]: (1) computations are performed on real numbers with infinite precision; (2) the noise is accurately sampled from a probability distribution (e.g., Laplace or Gaussian distribution). However, an implementation of DP mechanism is typically done with floating-point or fixed-point arithmetic that only provides finite precision. Mironov [77] demonstrates that the Laplace mechanism implementations using Laplace noise $Y \sim \mathsf{Lap}(\lambda)$ sampled with the textbook algorithms (i.e., compute $Y = (2Z - 1) \cdot \lambda \ln(U)$ using double-precision floating-point arithmetic, where $U \in (0, 1]$ and $Z \in \{0, 1\}$) enables an attacker to recover the entire database. Concretely, Mironov [77] shows that outputs of such implementations concentrate on a small subset of floating-point values which are correlated for inputs, leading to a breach of the DP guarantee. Jin et al. [64] extend the floating-point attack of Mironov [77] to the implementations of Gaussian mechanisms that generate Gaussian noise using the Marsaglia polar method [74], Box-Muller method [72], Ziggurat method [75], etc. Haney et al. [57] present precision-based floating-point attacks against implementations of several DP mechanisms (e.g., Laplace, Gaussian, Staircase [51], etc.) that enable an adversary to infer if the query result $f(D)$ equals 1. Casacuberta et al. [27] introduce another type of attack on DP mechanisms that fail to add a sufficient amount of noise. They exploit the properties (e.g., overflow or rounding) of integer and floating-point arithmetic operations that can lead to an underestimation of the sensitivity (cf. §2.1) in DP mechanisms.

The discussed MPC-based Laplace or Gaussian mechanisms either do not specify how to correctly/securely sample random noise and perturb the input [45, 63, 71], assume infinite precision [83, 98], or rely on a trusted third party that generates and adds DP noise [34, 83]. We implement existing DP mechanisms that are known not to be vulnerable to precision based attacks [57, 64, 77] in MPC protocols.

4 PROTOCOLS

In this section, we present our tailored MPC protocols for three types of DP mechanisms not vulnerable to the precision-based attacks [57, 64, 77], namely the discrete Laplace and discrete Gaussian mechanisms [26], the snapping mechanism [77], and the integer-scaling mechanism [55].

Notation. Here, we introduce the notation used in the remainder of this work. The shares of a secret value x held by N parties $\mathcal{P}_1, \dots, \mathcal{P}_\mathsf{N}$ are denoted by $\langle \mathsf{x}^\mathsf{d} \rangle^\mathsf{s} = \left(\langle \mathsf{x}^\mathsf{d} \rangle^\mathsf{s}_1, \dots, \langle \mathsf{x}^\mathsf{d} \rangle^\mathsf{s}_\mathsf{N} \right)$, where $\langle \mathsf{x}^\mathsf{d} \rangle^\mathsf{s}_i$ is held by party $\mathcal{P}_i$, $i \in [\mathsf{N}]$. The superscript $\mathsf{s} \in \{\mathsf{A}, \mathsf{B}, \mathsf{Y}\}$ is the secret sharing type. Here, A is Arithmetic sharing and B is Boolean sharing; both are based on the GMW protocol [53]. Y is the BMR protocol [15], an extension of Yao's Garbled Circuits protocol [100] from two to multiple parties. The second superscript $\mathsf{d} \in \{\mathbb{N}, \mathbb{Z}, \mathbb{L}\}$ indicates the data type in finite domains: $\mathbb{N}$ are unsigned integers, $\mathbb{Z}$ are signed integers, and $\mathbb{L}$ are floating-point values. We omit the superscript or subscript of share $\langle \mathsf{x}^\mathsf{d} \rangle^\mathsf{s}_i$ when it is clear from the context. $\langle \mathsf{x} \rangle^\mathsf{D} \leftarrow \mathsf{S2D}\left(\langle \mathsf{x} \rangle^\mathsf{S} \right)$ is the conversion from one secret sharing technique S to another D, $\mathsf{S} \neq \mathsf{D}$ and $\mathsf{S}, \mathsf{D} \in \{\mathsf{A}, \mathsf{B}, \mathsf{Y}\}$. Conversions between different data representations are presented in the same style. A multiplexer gate $\mathsf{MUX}(a, b, c)$ returns b if a is true or otherwise c. An AND gate is denoted by $\wedge$, an OR gate by $\vee$, a NOT gate by $\neg$, and an XOR gate by $\oplus$.

Random Number Generation. In our secure DP mechanisms in §4.1-4.3, we rely on three types of random number generators:

- $\left(\langle b_0 \rangle^\mathsf{B}, \dots, \langle b_{\ell-1} \rangle^\mathsf{B} \right) \leftarrow \mathsf{RandBits}(\ell)$ (cf. §B.3) generates secret-shares of an ℓ-bit uniformly random Boolean string $b \in \{0, 1\}^\ell$.
- $\langle x \rangle^\mathsf{B} \leftarrow \mathsf{RandInt}(m)$ (cf. §4.1) generates secret-shares of an uniformly random unsigned integer $x \in [0, m - 1]$ for $m \in \mathbb{Z}$.
- $\langle u^\mathbb{L} \rangle^\mathsf{B} \leftarrow \mathsf{RandFloat}(l, k)$ (cf. §B.7) generates uniformly random floating-point values $u \in [0, 1)$, where u has a l-bit mantissa and a k-bit exponent.

Computational Privacy. Our protocols have to fulfill both computational as well as output privacy. With respect to *computational privacy*, our protocols leak no intermediate values, i.e., all computation is run in MPC. Thus, computational privacy follows directly from the provable security of the employed MPC techniques, namely the Arithmetic (A), and the Boolean variant (B) of GMW [53] and BMR [15] (cf. §2) as well as private conversions [21]. We discuss the *output privacy* of the protocols in their respective sections.

In Appendix G we give an overview figure showing the noise generation procedure for the DP mechanisms (cf. §4 and §5).

4.1 MPC-Based Discrete Laplace/Gaussian Mechanisms

The discrete Laplace [52] and Gaussian DP mechanisms [26] are formulated as: $M_{\mathsf{Discrete}}(D) = f(D) + Q$, where $f(D)$ is the output of a query function and Q is additive noise sampled from the discrete Laplace or discrete Gaussian distributions (cf. §A). M_{Discrete} requires that both the function output $f(D) \in \mathbb{Z}^k$ and the noise $Q \in \mathbb{Z}^k$ are integers.

From an MPC point of view, the critical step is to sample a random integer from the discrete Laplace and Gaussian distributions (cf. §A). Afterwards, the secret-shared random value is securely added to the share of query output $f(D)$, which is a single operation in MPC. Consequently, we focus on the MPC-based sampling protocols in the following.

The discrete Laplace and Gaussian distributions (cf. §A) turn out to be closely related, such that a value sampled from a discrete Laplace distribution can be transformed into a sample from a discrete Gaussian distribution [26]. First, Prot. 2 generates shares of a randomly sampled discrete Laplace value $\langle Y^{\mathbb{Z}} \rangle^{\mathsf{B}}$, $Y \sim \mathsf{DLap}\left(\frac{t}{s}\right)$. Next, Prot. 3 converts those discrete Laplace value shares $\langle Y^{\mathbb{Z}} \rangle^{\mathsf{B}}$ into shares of discrete Gaussian value $\langle G^{\mathbb{Z}} \rangle^{\mathsf{B}}$ sampled from a discrete Gaussian distribution $G \sim \mathsf{DGauss}(\mu = 0, \sigma^2)$, where the variance σ^2 of added noise is calculated from the desired (ϵ, δ)-DP guarantee [11].

Our sampling of random discrete Laplace values in Prot. 2 follows the idea of Canonne et al. [26] because it is more efficient than the sampling methods of [30, 42, 46] that are based on processing a sequence of Boolean bits (cf. §1). Besides, we introduce several optimizations on the MPC side. The authors introduce a rejection sampling-based [41] approach that first generates geometric random values and converts them into discrete Laplace random values. Therefore, we also split the protocol for sampling from a discrete Laplace distribution into two sub-protocols. The first one, shown in Prot. 1, generates secret shares $\langle X^{\mathbb{N}} \rangle^{\mathsf{B}}$ of a geometric random value $X \sim \mathsf{Geo}\left(p = 1 - e^{-\frac{s}{t}}\right)$. Afterwards, Prot. 2 converts $\langle X^{\mathbb{N}} \rangle^{\mathsf{B}}$ into shares of a discrete Laplace random value $\langle Y^{\mathbb{Z}} \rangle^{\mathsf{B}}$, $Y \sim \mathsf{DLap}\left(\frac{t}{s}\right)$.

In this section, we will first present the MPC protocol for sampling from a geometric distribution (cf. §A) including several optimizations enhancing its efficiency. Next, we present the MPC protocol for sampling from a discrete Laplace distribution, which uses the geometric sampling protocol as a sub-protocol. Last, we present the discrete Gaussian sampling protocol, which uses the discrete Laplace sampling protocol as a sub-protocol.

Geometric Sampling. We begin by presenting the challenges and necessary optimizations involved in generating secret shares of geometric random values, the protocol for which is specified in Prot. 1. Our sub-protocols for oblivious selection Sel and Boolean-String multiplication BoolStrMul are presented in §B.6. More concretely, $\langle u^{\mathbb{N}} \rangle^{\mathsf{B}}, \langle b \rangle^{\mathsf{B}} \leftarrow \mathsf{Sel}\left(\langle u_0^{\mathbb{N}} \rangle^{\mathsf{B}}, \ldots, \langle u_{\ell-1}^{\mathbb{N}} \rangle^{\mathsf{B}}, \langle b_0 \rangle^{\mathsf{B}}, \ldots, \langle b_{\ell-1} \rangle^{\mathsf{B}}\right)$ outputs a bit-string $u^{\mathbb{N}} = u_i^{\mathbb{N}}$ and a bit $b = b_i$, where i is the index of the first non-zero bit b_i for $i \in [0, \ell-1]$. Protocol $\mathsf{BoolStrMul}\left(\langle a \rangle^{\mathsf{B}}, \langle b_0 \rangle^{\mathsf{B}}, \ldots, \langle b_{\ell-1} \rangle^{\mathsf{B}}\right)$ [8] computes the multiplication of one Boolean bit a with a set of ℓ Boolean bits $b_0, \ldots, b_{\ell-1}$.

```
Input  : $s, t$ // Parameters of $\mathsf{Geo}\left(1 - e^{-\frac{s}{t}}\right)$
Output: $\langle X^{\mathbb{N}} \rangle^{\mathsf{B}}$ // $X \sim \mathsf{Geo}\left(1 - e^{-\frac{s}{t}}\right)$ or $X = 0$
// Skip 1-st loop if $t == 1$
if $t == 1$ then
    $\langle u^{\mathbb{N}} \rangle^{\mathsf{B}} \leftarrow \langle 0^{\mathbb{N}} \rangle^{\mathsf{B}}$
    $\langle b \rangle^{\mathsf{B}} = 1$
end
else
    // Draw $\kappa_1$ random integers from interval $[0, t-1]$, and $\kappa_1$
    // random bits from a Bernoulli distribution
    for $i \leftarrow 0$ to $\kappa_1 - 1$ do
        $\langle u_i^{\mathbb{N}} \rangle^{\mathsf{B}} \leftarrow \mathsf{RandInt}(t)$
        $\langle b_i \rangle^{\mathsf{B}} \leftarrow \mathsf{Bernoulli}\left(e^{\frac{\mathsf{UINT2FL}\left(\langle u_i^{\mathbb{N}} \rangle^{\mathsf{B}}\right)}{-t}}\right)$
    end
    // Set $u = u_k$ and $b = b_k$ if $b_k == 1$ for $k \in [0, \kappa_1 - 1]$
    $\langle u^{\mathbb{N}} \rangle^{\mathsf{B}}, \langle b \rangle^{\mathsf{B}} \leftarrow$
      $\mathsf{Sel}\left(\langle u_0^{\mathbb{N}} \rangle^{\mathsf{B}}, \ldots, \langle u_{\kappa_1-1}^{\mathbb{N}} \rangle^{\mathsf{B}}, \langle b_0 \rangle^{\mathsf{B}}, \ldots, \langle b_{\kappa_1-1} \rangle^{\mathsf{B}}\right)$
end
// Draw $\kappa_2$ random bits from a Bernoulli distribution
for $j \leftarrow 0$ to $\kappa_2 - 1$ do
    $\langle c_j \rangle^{\mathsf{B}} \leftarrow \mathsf{Bernoulli}\left(e^{-1}\right)$
    $\langle d_j \rangle^{\mathsf{B}} \leftarrow \neg \langle c_j \rangle^{\mathsf{B}}$
end
// Draw $v = v_k \sim \mathsf{Geo}\left(1 - e^{-1}\right)$ and set $d = d_k$ if $d_k == 1$ for
// $k \in [0, \kappa_2 - 1]$
$\langle v^{\mathbb{N}} \rangle^{\mathsf{B}}, \langle d \rangle^{\mathsf{B}} \leftarrow$
  $\mathsf{Sel}\left(v_0 \leftarrow 0, \ldots, v_{\kappa_2-1} \leftarrow \kappa_2 - 1, \langle d_0 \rangle^{\mathsf{B}}, \ldots, \langle d_{\kappa_2-1} \rangle^{\mathsf{B}}\right)$
// Output $X = \left\lfloor \frac{u + v \cdot t}{s} \right\rfloor$ or $X = 0$ if $\mathsf{GeometricExp}(s, t)$ fails
$\langle X^{\mathbb{N}} \rangle^{\mathsf{B}} \leftarrow$
  $\mathsf{BoolStrMul}\left(\mathsf{Floor}\left(\frac{\langle u^{\mathbb{N}} \rangle^{\mathsf{B}} + \langle v^{\mathbb{N}} \rangle^{\mathsf{B}} \cdot t}{s}\right), \langle b \rangle^{\mathsf{B}} \wedge \langle d \rangle^{\mathsf{B}}\right)$
```

Protocol 1: GeometricExp — our MPC protocol realizing Geometric sampling [26].

Our MPC protocols directly implement the steps of the plaintext sampling algorithms in [26]. We chose algorithms from Canonne et al. [26] for MPC efficiency reasons. For example, Google's DP library [56] also contains securely implemented plaintext discrete Laplace sampling, but it is based on binary search geometric sampling. A realization in MPC would require computing 52 iterations (each computing floating-point natural logarithm and exponentiations, thus, an impracticable overhead) as integers up to 2^{52} can be represented precisely in floating-point arithmetic.

To generate samples from a geometric distribution (cf. §A), the while-loops in Algorithm 2 in [26] repeatedly sample from Bernoulli distributions until termination conditions are met. Thus, each loop generating a random value can in theory run for an infinite number of iterations with negligible probability. As the number of iterations is not fixed, a key challenge is to realize it in MPC (1) without information leakage and (2) in an efficient manner.

An idea is to limit the number of iterations by fixing it to a predefined number κ of iterations. A check in MPC determines whether the random number has been generated; if it has not, κ additional iterations are run. Unfortunately, this strategy is not fully privacy-preserving, as it leaks one bit of information indicating whether to continue the computation. Critically, Jin et al. [64] construct a timing attack against the discrete Laplace mechanism [26] and show that the magnitude of the generated random value exhibits a positive linear relation to the number of required iterations, i.e., a large random value is usually generated in more iterations.

Instead, we omit the check and run the protocol for a fixed number of iterations κ. This, however, means that the output may be zero with some failure probability p_{failure}. We derive p_{failure} from κ for our MPC-protocols that sample random values from the geometric, discrete Laplace, and Gaussian distributions in §D. We can guarantee a small failure probability, i.e., $p_{\text{failure}} < 2^{-40}$, by choosing an appropriate number of iterations κ. The DP guarantee will have $\delta = p_{\text{failure}} < 2^{-40}$, which protects individual privacy well when noise is added to functions of datasets with size $\|D\|_1 \ll 2^{40}$. Different choices for κ allow δ to be adjusted.

Theorem 4.1. *Consider mechanism $\mathcal{M}$ that is (ϵ, δ)-DP, as well as mechanism $\mathcal{M}'$ defined as follows:*

$$\mathcal{M}'(x) = \begin{cases} \mathcal{M}(x) & \text{w.p. } 1 - p_{\text{failure}} \text{ (event } \neg E) \\ f(x) & \text{w.p. } p_{\text{failure}} \text{ (event } E). \end{cases}$$

Then mechanism $\mathcal{M}'$ is $(\epsilon, \delta + p_{\text{failure}})$-DP. The proof is given §C.

We additionally introduce five optimizations on protocol level to further improve the efficiency of our MPC protocol:

```
Input  : $t, s$ // Parameters of $\mathsf{DLap}\left(\frac{t}{s}\right)$
Output: $\langle Y^{\mathbb{Z}} \rangle^{\mathsf{B}}$ // $Y \sim \mathsf{DLap}\left(\frac{t}{s}\right)$ or $Y = 0$
for $i \leftarrow 0$ to $\kappa_3 - 1$ do
    // Generate sign $S_i$ for $Y \sim \mathsf{DLap}\left(\frac{t}{s}\right)$
    $\langle S_i \rangle^{\mathsf{B}} \leftarrow \mathsf{RandBits}(1)$
    // Generate the integer part $m_i \sim \mathsf{Geo}\left(1 - e^{-\frac{s}{t}}\right)$ for $Y$
    $\langle m_i{}^{\mathbb{N}} \rangle^{\mathsf{B}} \leftarrow \mathsf{GeometricExp}(s, t)$
    // Check if $Y = (1 - 2S_i) \cdot m_i = (-1) \cdot 0 = -0$
    $\langle f_i \rangle^{\mathsf{B}} \leftarrow \neg \left( \left( \langle S_i \rangle^{\mathsf{B}} \right) \wedge \left( \langle m_i{}^{\mathbb{N}} \rangle^{\mathsf{B}} == 0 \right) \right)$
end
// If $f_k == 1$, set $m = m_k$ and $b = b_k$ for $k \in [0, \kappa_3 - 1]$
$\langle m^{\mathbb{N}} \rangle^{\mathsf{B}} \,\|\, \langle S \rangle^{\mathsf{B}} \leftarrow \mathsf{Sel}\left( \langle m_0{}^{\mathbb{N}} \rangle^{\mathsf{B}} \,\|\, \langle S_0 \rangle^{\mathsf{B}}, \ldots, \langle f_0 \rangle^{\mathsf{B}}, \ldots \right)$
Set $\neg \langle S \rangle^{\mathsf{B}}$ as the sign bit of $\langle m^{\mathbb{N}} \rangle^{\mathsf{B}}$
// Output $Y = (1 - 2S) \cdot m$ or $Y = 0$ if $\mathsf{DiscreteLaplace}(t, s)$
// fails
$\langle Y^{\mathbb{Z}} \rangle^{\mathsf{B}} \leftarrow \langle m^{\mathbb{N}} \rangle^{\mathsf{B}}$
```

Protocol 2: DiscreteLaplace — our MPC protocol realizing discrete Laplace sampling [26].

(1) *Parallelization.* Canonne et al.'s [26] geometric sampling algorithm uses two nested while-loops. The inner loop is run only if the outer loop was successful. In MPC, we cannot leak if the generation was successful. Hence, we fix the number of iterations to κ_1 and κ_2 iterations, which leads to $\kappa_1 \cdot \kappa_2$ iterations in total. Since the operations inside the loops are independent, we can run both loops independently and in parallel, leading to only $\kappa_1 + \kappa_2$ iterations. We choose κ_1 and κ_2 to guarantee that both loops output the required random values with high probability (cf. §D.1). Each loop can be run using Single Instruction Multiple Data (SIMD [18, 40, 87]), which fully parallelizes our protocol, enhances computation efficiency, and reduces memory consumption [40].

(2) *Random Integer Generation.* The random integer generation in line 7 of Prot. 1, indicated by $\mathsf{RandInt}(t)$, generates a uniformly random integer $u \in [0, t-1]$. It uses the Simple Modular Method [12] to generate secret shares of an ℓ-bit random unsigned integer $x \in \{0, \ldots, t-1\}$ for $t \in \mathbb{Z}$. We observe that when $t = 2^k$, the expensive modular reduction (cf. Tab. 11) operation can be omitted. Now each party can generate k uniform random bits $\left(\langle b_0 \rangle^{\mathsf{B}}, \ldots, \langle b_{k-1} \rangle^{\mathsf{B}} \right) \leftarrow \mathsf{RandBits}(k)$ (cf. §B.3) and sets $\langle x \rangle^{\mathsf{B},\mathbb{N}} = \langle b_0 \rangle^{\mathsf{B}}, \ldots, \langle b_{k-1} \rangle^{\mathsf{B}}$ *locally* to create the secret-shared random integer.

(3) *Floating-Point Division.* We avoid the floating-point division in line 8 of Prot. 1 by first computing $e^{-\frac{1}{t}}$ in plaintext, followed with $e^{-\frac{1}{t}} \cdot e^{\mathsf{UINT2FL}\left(\langle u_i^{\mathbb{N}} \rangle^{\mathsf{B}} \right)}$. The floating-point multiplication is up to 4× faster than the floating-point division in $\{\mathsf{B}, \mathsf{Y}\}$ (cf. Tab. 11).

(4) *Integer Division.* Integer division (Line 17 in Prot. 1) is a very expensive operation in MPC [21] (e.g., about $54.40 - 70.15$ ms for a single division in B-sharing, cf. Tab. 11). Thus, we first convert integers from B-sharing to Y-sharing before dividing and rounding down to the next integer. This approach is up to 20× faster than integer division in B (cf. §5.2).

(5) *Bernoulli Sampling.* Canonne et al. [26] propose a Bernoulli sampling algorithm. However, similar to Prot. 1, it requires a large number of iterations to guarantee a negligible failure probability. Instead, we adopt a protocol from CrypTen [71] for sampling a random value b from a Bernoulli distribution with parameter p and transfer it from fixed-point to floating-point arithmetic. A random value b drawn from the Bernoulli distribution equals 1 with probability p and 0 with probability $1 - p$. This is equivalent to $b = (x < p)$, where x is a uniformly random value in $(0, 1)$. We further optimize the efficiency of the comparison by purely relying on integer arithmetic. Concretely, each party locally generates κ random bits and interprets those as a random unsigned integer x. Then, it computes $b = x < (p_{\ll \kappa})$, where $p_{\ll \kappa}$ is the binary representation of p after a left-shift of κ bits. Depending on the MPC technique, our integer comparison-based Bernoulli sampling protocol (Line 8, 13 in Prot. 1) is up to $4.8 - 6.5\times$ faster than the naive floating-point comparison-based protocol (cf. Tab. 11).

Discrete Laplace Sampling. Next, we show how to convert the samples from a geometric distribution to samples from a Laplace distribution (cf. §A), as specified in Prot. 2. The protocol generates secret shares of a discrete Laplace random value $Y = (1 - 2S) \cdot m \sim \mathsf{DLap}\left(\frac{t}{s}\right)$ using the random values drawn from the geometric distribution $m_i \sim \mathsf{Geo}\left(1 - e^{-\frac{s}{t}}\right)$ with Prot. 1, and a random sign bit $S_i \in \{0,1\}$ (cf. Lines $2-3$). However, $Y = (1-2S) \cdot m$ would equal 0 with twice the probability as it would occur in the discrete Laplace distribution [26]. Hence, we check in line 4 if $S_i == 1$ and $m_i == 0$. If it is the case, we re-sample both values. Otherwise, we either output $Y = (1-2S) \cdot m$ for $m = m_i$ and $S = S_i$, if $f_i == 1$ or $Y = 0$ otherwise (cf. Lines $6-8$). κ_3 is again set such that p_{fail} (DiscreteLaplace, Prot. 2) $< 2^{-40}$ (cf. §D.2).

Discrete Gaussian Sampling. We generate shares of a discrete Gaussian random value (cf. §A) using Prot. 3 by adapting the plaintext Algorithm 3 of Canonne et al. [26] and set κ_4 such that Prot. 3 fails with a probability less than 2^{-40} (cf. §D.2).

Correctness. Correctness can be derived from the correctness of outputs from the plaintext algorithms and the MPC techniques employed. Our MPC-based sampling protocols execute all steps as specified by the respective plaintext algorithm. The sampling procedure fails to generate noise from the specified distribution with only a tunable, small probability, which we set to be negligible ($< 2^{-40}$) and incorporate into (ϵ, δ)-DP. This negligible failure probability ensures correctness. Next, we analyze the five optimizations we introduced for the noise sampling w.r.t. their effect on correctness. Due to the obliviousness requirement of MPC, i.e., the control flow of the computation must be independent of the input data, MPC protocols cannot check and terminate after a successful random integer generation. Instead, we fix the number of iterations of both loops generating random integers in Prot. 1, which increases computation overhead, keeping the behavior of the protocol unchanged except for the small failure probability. Our parallelization improves efficiency, but does not affect correctness. The random integer generation as well as the floating-point and integer divisions optimizations are only transformations of the original formulas, i.e., the computation is unchanged. The Bernoulli Sampling protocol was shown to be correct in [71]. Except for fixing the number of iterations (where correctness follows from the same arguments as discussed above[1]), our MPC-based discrete Laplace mechanism M_{DLap} and discrete Gaussian mechanism M_{DGauss} in §4.1 are directly derived from the algorithms in [26].

Output Privacy. Output privacy (quantified by (ϵ, δ)-CDP) follows directly from the correctness and output privacy of the plaintext algorithms, as well as Lemma 2.6 and Theorems 2.5 and 4.1. Concretely, [26] proves that the original mechanisms satisfy (ϵ, δ)-DP, and our protocols correctly and securely generate noise from the same distributions except with a small failure probability. Taking into account the finite domain, our protocols satisfy $(\epsilon, \delta + \delta_\lambda + \delta' + p_{\text{failure}})$-CDP, where δ_λ is negligible in the security parameter λ from Theorem 2.5, p_{failure} is set to some small failure probability from Theorem 4.1, and δ' is the probability with which the sampled noise exceeds the maximum value from Lemma 2.6. This bound is based on the tail bounds of the discrete Laplace and Gaussian distributions.

4.2 MPC-Based Snapping Mechanism

Mironov [77] proposes the snapping mechanism M_{Snap} as a remedy for the insecure implementations of the Laplace mechanism using *floating-point arithmetic*. The snapping mechanism is parameterized by B and λ, which can be chosen in advance, and satisfies $\left(\frac{1}{\lambda} + 2^{-49} \cdot \frac{B}{\lambda}\right)$-DP for $\lambda < B < 2^{46} \cdot \lambda$. The mechanism is defined as follows:

$$M_{\text{Snap}}(f(D), \lambda, \Lambda, B) = \mathsf{clamp}_B\left(\left\lfloor \mathsf{clamp}_B(f(D)) + Y_{\text{noise}} \right\rceil_\Lambda\right), \quad Y_{\text{noise}} = S \cdot \lambda \cdot \ln(U). \tag{4}$$

In Eq. 4, $f(D)$ is the output of a query function that takes database D as input. Y_{noise} is the Laplace noise, S is the sign of the noise and uniformly distributed over $\{-1, 1\}$, and U is a random real

```
Input  : $\sigma$ // Parameter of DGauss $(\mu = 0, \sigma^2)$
Output : $\langle G^{\mathbb{Z}} \rangle^B$ // $G \sim$ DGauss $(\mu = 0, \sigma^2)$ or $G = 0$
$t \leftarrow \lfloor \sigma \rfloor + 1$
for $j \leftarrow 0$ to $\kappa_4 - 1$ do
    $\langle Y_j^{\mathbb{Z}} \rangle^B \leftarrow \mathsf{DiscreteLaplace}(t, s = 1)$
    $\langle a^{\mathbb{L}} \rangle^B \leftarrow \frac{\left(\mathsf{UINT2FL}\left(\left|\langle Y_j^{\mathbb{Z}} \rangle^B\right|\right) - \frac{\sigma^2}{t}\right)^2}{2\sigma^2}$
    $\langle b_j \rangle^B \leftarrow \mathsf{Bernoulli}\left(e^{-\langle a^{\mathbb{L}} \rangle^B}\right)$
end
$\langle G^{\mathbb{Z}} \rangle^B, \langle b \rangle^B \leftarrow \mathsf{Sel}\left(\langle Y_0^{\mathbb{Z}} \rangle^B, \ldots, \langle b_0 \rangle^B, \ldots\right)$
```

Protocol 3: DiscreteGaussian — our MPC protocol realizing discrete Gaussian sampling [26] .

```
// Secret-shared exact query result, parameters of $M_{\text{Snap}}$
Input  : $\langle f(D)^{\mathbb{L}} \rangle^B, \lambda, \Lambda, B$
// Secret-shared DP query result
Output : $\langle y_{\text{Snap}}^{\mathbb{L}} \rangle^B$
// Generate a random floating-point number $U \in (0,1)$ and a
// random bit
$\langle U^{\mathbb{L}} \rangle^B \leftarrow \mathsf{RandFloat}$
$\langle S \rangle^B \leftarrow \mathsf{RandBits}(1)$
 // Bound $f(D)$ to $[-B, B]$
$\langle f(D)_{\text{clamp}}^{\mathbb{L}} \rangle^B \leftarrow \mathsf{Clamp}\left(B, \langle f(D)^{\mathbb{L}} \rangle^B\right)$
 // Generate Laplace noise $Y = S \cdot \lambda \cdot \mathsf{LN}(U)$
$\langle Y_{\text{noise}}^{\mathbb{L}} \rangle^B \leftarrow \lambda \cdot \mathsf{LN}\left(\langle U^{\mathbb{L}} \rangle^B\right)$
$\mathsf{Sign}\left(\langle S \rangle^B, \langle Y_{\text{noise}}^{\mathbb{L}} \rangle^B\right)$
// Add noise $Y_{\text{noise}}$ to the bounded query result $f(D)_{\text{clamp}}$
$\langle x^{\mathbb{L}} \rangle^B \leftarrow \langle f(D)_{\text{clamp}}^{\mathbb{L}} \rangle^B + \langle Y_{\text{noise}}^{\mathbb{L}} \rangle^B$
 // Round perturbation result $x$ to multiple of $\Lambda$
$\langle x_\Lambda^{\mathbb{L}} \rangle^B \leftarrow \left\lfloor \langle x^{\mathbb{L}} \rangle^B \right\rceil_\Lambda$
 // Bound $x_\Lambda$ to $[-B, B]$
$\langle y_{\text{Snap}}^{\mathbb{L}} \rangle^B \leftarrow \mathsf{Clamp}\left(B, \langle x_\Lambda^{\mathbb{L}} \rangle^B\right)$
```

Protocol 4: M_{Snap} — our MPC protocol realizing the snapping mechanism [77].

[1] Note that this is true for all cases where the number of iteration is fixed in our protocols.

number over $(0, 1)$ represented as a floating-point number that is output with probability proportional to its unit in the last place. $\mathsf{ln}(x)$ is the floating-point natural logarithm with exact rounding, i.e., $\mathsf{ln}(x)$ always rounds the output to the closest floating-point number. The addition (+) and multiplication ($\cdot$) operations in Eq. 4 are floating-point arithmetic operations. Function $\mathsf{clamp}_B(x)$ limits the output to the interval $[-B, B]$ by outputting B if $x > B$, $-B$ if $x < -B$, and x otherwise. Function $\lfloor x \rceil_\Lambda$ rounds x to the nearest multiple of Λ, where Λ is the smallest power of 2 greater than or equal to λ. Notice that clamping can be done without introducing additional error based on the choice of parameter B. To create the MPC version for the snapping mechanism [77] in Prot. 4, we first sample Y_{noise}, which can be done independently of the input and pre-computed in the offline phase. In the online phase, we must compute and clamp the function output, add Y_{noise}, and clamp the result $\mathsf{clamp}_B\left(\lfloor \mathsf{clamp}_B(f(D)) + Y_{\mathsf{noise}} \rceil_\Lambda\right)$. We also encounter some challenges and introduce protocol optimizations to improve efficiency as follows:

(1) *Floating-Point Arithmetic.* The snapping mechanism [77] requires floating-point arithmetic to generate the random noise values added to the query result. However, floating-point arithmetic in MPC is expensive [4, 7], so most MPC frameworks [71, 78, 80] prefer to use fixed-point arithmetic [28]. We extend the MOTION framework [21] to support floating-point arithmetic in $\{\mathsf{A}, \mathsf{B}, \mathsf{Y}\}$-sharing. Specifically, we convert circuits from [38] that support IEEE 754 compliant floating-point arithmetic operations to the Bristol circuit format used in MOTION for $\{\mathsf{B}, \mathsf{Y}\}$-sharing.

We also implement MPC protocols for logical/arithmetic shifting from [47] that are needed for the floating-point operations in A-sharing following ideas of Aliasgari et al. [4]. The difference is that the protocols in [4] are designed for Shamir's secret sharing [88], whereas the operations are performed over a prime field $\mathbb{F}_p$ (modulo p) that supports the inverse operation, while the A-sharing operations in the MOTION framework [21] are performed over a ring (modulo $\mathbb{Z}_{2^\ell}$).

(2) *Input Bounding.* MPC requires an input-independent program flow, i.e., branching depending on input values is not possible. Thus, the MPC-protocol realizing clamp_B must ensure that it does not leak the interval of the input x. We realize this by a multiplexer-based floating-point comparison protocol.

(3) *Rounding to Nearest Multiple of* Λ. $\lfloor x \rceil_\Lambda$ rounds floating-point inputs x to the nearest multiple of Λ. As Λ is a power of two, our protocol can directly compute the rounding on the binary representation of input x. We create new depth- and size-optimized Boolean circuits with the CBMC-GC circuit compiler [24] to realize $\lfloor x \rceil_\Lambda$ in MPC using $\{\mathsf{B}, \mathsf{Y}\}$-sharing. The program for CBMC-GC [24] is inspired by Covington's work [35] that relies on bit manipulation of the binary representation of floating-point numbers x. The bit manipulation operations are equivalent to first computing $x' = \frac{x}{\Lambda}$ using floating-point division and then rounding x' to the nearest integer. The rounded result can then be obtained by a simple multiplication: $\lfloor x \rceil_\Lambda = x' \cdot \Lambda$. Evaluating our circuit for rounding with MPC is $4.5 - 15.0\times$ faster than the above floating-point operations (cf. Tab. 11), as it only requires bit-level logical operations such as bit-shifting, AND, and XOR.

(4) *Sign Setting.* In Eq. 4, the sign of the Laplace random value Y_{noise} is multiplied by a random value $S \in \{-1, 1\}$. This expensive floating-point multiplication can be avoided by directly manipulating the sign bit of Y_{noise}. Concretely, in $\{\mathsf{B}, \mathsf{Y}\}$-sharing (resp. A-sharing) the Boolean sign bit (resp. the arithmetic share containing the sign) of $\langle Y_{\mathsf{noise}} \rangle$ is simply replaced by a freshly generated random secret-shared Boolean bit $\langle s \rangle$ (resp. random arithmetic share), where $s \in \{0, 1\}$. We indicate this operation as $\mathsf{Sign}(\langle s \rangle, \langle Y_{\mathsf{noise}} \rangle)$.

(5) *Secret Sharing.* A difficult open problem in MPC research is how to effectively determine the best mix of MPC-techniques for the most efficient, privacy-preserving instantiation of an algorithm. First attempts to create automatic compilers [23, 24, 38, 48, 60, 82] still exhibit significant shortcomings with respect to efficiency compared to protocols where the MPC-techniques have been carefully combined by hand. Thus, to find the most efficient instantiation for our MPC protocol of the snapping mechanism [77] (as well as for our other protocols), we micro-benchmark all relevant sub-protocols (i.e., floating-point addition, multiplication, and natural logarithm, Clamp and $\lfloor x \rceil_\Lambda$) with each sharing ($\{\mathsf{A}, \mathsf{B}, \mathsf{Y}\}$) to compose the most efficient mix. The results can be found in §F. Taking conversion cost into account, using Y in the two-party setting and B-sharing in the multi-party setting for all parts in a LAN network (cf. §5.1) leads to the most efficient solution. The benchmark result of Prot. 4 is given in §5.2.

Correctness. Our snapping mechanism M_{Snap} (cf. Prot. 4) directly realizes the plaintext algorithm by Mironov [77] in MPC. Our five optimizations discussed in §4.2 are MPC protocol optimizations, i.e., formula transformations, efficient circuit generations, and efficient combinations of MPC techniques, which do not change the underlying computation.

Output Privacy. [77] proves that the original mechanism satisfies (ϵ, δ)-DP. Taking into account the finite domain, our protocols satisfy $(\epsilon, \delta + \delta_\lambda + \delta')$-CDP, where δ_λ is negligible in the security parameter λ from Theorem 2.5, and δ' is the probability with which the sampled noise exceeds the maximum value from Lemma 2.6, which is based on the tail bounds of the Laplace distribution.

4.3 MPC-Based Integer-Scaling Mechanisms

Google [55] introduced a framework to securely realize DP by adding appropriately *scaled* discrete noise using floating-point arithmetic operations. Their integer-scaling Laplace mechanism was built to address some of the challenges associated with the snapping mechanism [77] and the precision-based attacks [57, 64, 77]. Namely, the snapping mechanism adds more noise than would theoretically be necessary for a given DP guarantee, offering a weaker privacy-utility trade-off than the integer-scaling mechanism [26]. The integer-scaling mechanism also offers a Gaussian variant. In the following, we call the two mechanisms introduced in [55] the integer-scaling Laplace mechanism (cf. §4.3) and the integer-scaling Gaussian mechanism (cf. §E). Both are defined as follows:

$$M_{\mathsf{IS}}(f(D), r, \varepsilon, \delta) = f_r(D) + i \cdot r, \tag{5}$$

where discrete random values i are scaled by a resolution parameter $r = 2^k$ (for $k \in [-1022, 970]^2$) which controls the discretization of the simulated continuous noise. M_{IS} satisfies (ε, δ)-DP and the function $f_r(D)$ rounds the output of a query function $f(D)$ to the nearest multiple of r. The scaled discrete noise $i \cdot r$ is used to simulate the continuous noise, e.g., the Laplace noise in Lemma 2.2 or the Gaussian noise Lemma 2.3 with the resolution parameter r. To create MPC protocols for the integer-scaling mechanisms, we first sample $i \cdot r$, which can be done independently of the input and pre-computed in the offline phase. In the online phase, we must compute and clamp the function output $f_r(D)$ and add the noise.

Integer-Scaling Laplace Mechanism. Here, we introduce our MPC protocol for the integer-scaling Laplace mechanism M_{ISLap} [55]. Prot. 5 presents our MPC protocol for the integer-scaling Laplace mechanism using the previously presented sub-protocol (cf. Prot. 2). The resolution parameter r is used to re-scale the exact query result $f(D)$ and DP noise i. It is set to the smallest power of 2 that is greater than $\frac{\Delta_1 f}{2^\gamma \epsilon}$ for $\gamma \in [10, 45]$, where γ controls accuracy and discretization, ε is the DP parameter, and $\Delta_1 f$ is the ℓ_1-sensitivity of $f(D)$. We first generate shares of a discrete random value $\langle i_{\text{DLap}} \rangle$ using Prot. 2 (cf. Line 1). Then, in line 2, $\langle i_{\text{DLap}} \rangle$ is converted from integer to floating-point representation. To prevent precision-based attacks [57, 64, 77], integer i_{DLap} has to be scaled by r without precision loss, which requires $i_{\text{DLap}} \in [2^{-52}, 2^{52}]$. We use our MulPow2 protocol (cf. §B.8) to re-scaled i_{DLap} by a factor $r = 2^k$, that is 3.1 − 20.5× faster than the direct floating-point multiplication (cf. Tab. 11), as it operates on the exponent part of the floating-point numbers. Lastly, we compute the DP query result $\langle f_r(D) \rangle$ after rounding $\langle f(D) \rangle$ to the nearest multiple of r by adding the scaled Laplace noise Y_{Lap}. We use the same rounding operation $\lfloor \cdot \rceil_r$ we presented in §4.2.

We empirically evaluate the efficiency for each sub-protocol for the different secret sharing techniques. The most efficient approach is to fully run the protocol in Y-sharing in a LAN network (cf. §5.1) in the two-party setting or in B in the multi-party setting. The benchmark results of Prot. 5 are given in §5.2.

Integer-Scaling Gaussian Mechanism. The integer-scaling Gaussian and Laplace mechanisms both use symmetrical binomial DP noise i (cf. Prot. 8). We refer the readers to §E for details.

Correctness. The integer-scaling Laplace mechanism M_{ISLap} (cf. Prot. 5) and integer-scaling Gaussian mechanism M_{ISGauss} (cf. Prot. 7) are directly based on [55].

Output Privacy. [55] proves that the original mechanisms satisfy (ϵ, δ)-DP. Since our protocols correctly and securely generate noise from the same distributions except with a small failure probability, taking into account the finite domain, our protocols satisfy $(\epsilon, \delta + \delta_\lambda + \delta' + p_{\text{failure}})$-CDP, where δ_λ is negligible in the security parameter λ from Theorem 2.5, p_{failure} is set to some small failure probability from Theorem 4.1, and δ' is the probability with which the sampled noise exceeds the maximum value from Lemma 2.6, based on the tail bounds of the Laplace and Gaussian distributions.

```
    // Secret-shared exact query result, resolution and DP
    // parameters of $M_{\text{ISLap}}$, $\ell_1$-sensitivity of query function $f$
    Input  : $\left\langle f(D)^{\mathbb{L}} \right\rangle^{\text{B}}, r, \varepsilon, \Delta f$
    // Secret-shared DP query result
    Output : $\left\langle y_{\text{ISLap}}^{\mathbb{L}} \right\rangle^{\text{B}}$
    // Generate $i_{\text{DLap}} \sim \text{DLap}\left(\frac{t}{s}\right)$, where $\frac{t}{s} = \frac{\Delta f + r}{r \cdot \varepsilon}$
1   $\left\langle i_{\text{DLap}}^{\mathbb{N}} \right\rangle^{\text{B}} \leftarrow \text{DiscreteLaplace}(t, s)$
    // Re-scale $Y_{\text{Lap}} = i_{\text{DLap}} \cdot 2^k$
2   $\left\langle Y_{\text{Lap}}^{\mathbb{L}} \right\rangle^{\text{B}} \leftarrow \text{MulPow2}\left(\text{UINT2FL}\left(\left\langle i_{\text{DLap}}^{\mathbb{N}} \right\rangle^{\text{B}}\right), k = \log_2 r\right)$
    // Round $f(D)$ to nearest multiple of $r$
3   $\left\langle f_r(D)^{\mathbb{L}} \right\rangle^{\text{B}} \leftarrow \left\lfloor \left\langle f(D)^{\mathbb{L}} \right\rangle^{\text{B}} \right\rceil_r$
    // Perturb $f_r(D)$ with Laplace noise $Y_{\text{Lap}}$
4   $\left\langle y_{\text{ISLap}}^{\mathbb{L}} \right\rangle^{\text{B}} \leftarrow \left\langle f_r(D)^{\mathbb{L}} \right\rangle^{\text{B}} + \left\langle Y_{\text{Lap}}^{\mathbb{L}} \right\rangle^{\text{B}}$
```

Protocol 5: M_{ISLap} — our MPC protocol realizing the integer-scaling Laplace mechanism [55].

4.4 Summary

In this section, we have introduced MPC protocols for the discrete Laplace and Gaussian mechanisms (cf. §4.1), the snapping mechanism (cf. Prot. 4), and the integer-scaling Laplace and Gaussian mechanisms (cf. Prot. 5 and Prot. 7).

The best mechanism in practice depends on the type of function, the available computational resources, as well as the desired utility and DP guarantees. Therefore, the best choice depends on similar factors as in implementations of these DP mechanisms without MPC.

For functions that output integer values, the discrete Laplace and Gaussian mechanisms are a good choice in general. These mechanisms offer a privacy-utility trade-off corresponding to the bounds offered by DP, and the computational bottleneck of these protocols is the large number of floating-point exponentiations. The noise is also sampled from well-studied discrete distributions, samples of which are not vulnerable to precision-based attacks in practical implementations.

For functions that output floating-point values, the snapping mechanism or integer scaling Laplace or Gaussian mechanisms can be used. The protocol for the snapping mechanism is the most efficient (cf. §5.2). However, it offers a sub-optimal privacy-utility trade-off. The integer-scaling Laplace and Gaussian mechanisms offer a better privacy-utility trade-off, but they require more iterations with floating-point exponentiation, making the protocols less efficient.

5 EXPERIMENTAL EVALUATION

To evaluate the efficiency of our MPC protocols for secure DP mechanisms presented in §4, we provide an implementation and extensive benchmarks. We aim at high usability and will release

[2] [55] sets $k \in [-1022, 1023]$, but if $i = 2^{52}$ and $r = 2^k = 2^{1023}$, $i \cdot r$ cannot be represented correctly as a double-precision floating point number.

our code[3] as open-source software under the permissive MIT license[4]. In this section, we first present our benchmark setup and give a security and complexity analysis of our MPC protocols before discussing the experimental results. Our benchmarks evaluate computation and communication efficiency, including comparisons to existing MPC protocols for *insecure* DP mechanisms [45, 71].

5.1 Experimental Setup

Server Configuration. The experiments are run on five servers equipped with Intel Core i9-7960X processors and 128GB RAM. We consider three network environments: (1) LAN10: 10-Gbit/s with 1ms RTT, (2) LAN1: 1-Gbit/s with 1ms RTT, and (3) WAN100: 1-Mbit/s with 100ms RTT. We extend MOTION [21] to 8/16/32/64/128-bit signed integer and 32/64-bit floating-point arithmetic in $\{\mathsf{A}, \mathsf{B}, \mathsf{Y}\}$ as well as conversions between those.

Setting. The implementation of our MPC-based DP mechanisms generate 64-bit random integer/floating-point values used as DP noise. They can be run in an outsourcing [67] or a multi-party computation (N-PC) scenario where N data owners run the computation among themselves. In our experiments, we benchmark the outsourcing setting where an arbitrary number of data owners secret share their data (i.e., the query results) and send the shares to the computing parties instantiated by the servers. Those then jointly add the noise for guaranteeing DP. Note that the noise shares can be generated *offline* (i.e., independently of the input data and, thus, *before* receiving the input data shares from the data owners). Moreover, in this scenario, the noise generation is *independent* of the number of data owners. Additionally, pre-computed DP noise shares can also be transferred to N-PC scenarios as long as the magnitude of the noise, the MPC protocols, and data types are correctly configured.

Implementation. In our experiments, we compare with the state-of-the-art, but *insecure* MPC-based (discrete) Laplace mechanisms in PrivaDA [45] and Gaussian mechanisms in CrypTen [71]. Note that although M_{DLap} [45] is not susceptible to the precision-based attacks [57, 64, 77] discussed in §3, its correctness relies on real number arithmetic that is not satisfied when using floating-point arithmetic. We (re-)implement our protocols as well as previous work in the state-of-the-art full threshold passively secure MPC framework MOTION [21] for a fair comparison. Note that our proposed MPC protocols can naturally be translated to frameworks with stronger security such as ABY3 [78] or MP-SDPZ [70]. We provide a complexity analysis of circuit size and depth, as well as experimental communication costs in Appendix I.

5.2 Performance

We benchmark the efficiency of our protocols for the *secure* DP mechanisms presented in §4 and §E.

Sharing Techniques. In the two-party (2PC) setting, Yao's Garbled Circuit [100] with Three-Halves garbling [86] turned out to be the most efficient technique for our building blocks in the microbenchmarks (cf. §F). With N ≥ 3 parties, B-sharing was more efficient. All microbenchmarks for our building blocks can be found in §F.

Optimization Effects. To evaluate the performance of optimizations presented in §4, we implement the *naive* and *optimized* versions of M_{DLap}, M_{DGauss} (§4.1), M_{Snap} (§4.2), and M_{ISLap} (§4.3) and M_{ISGauss} (§E). The *naive* version refers to the protocols that are transferred from the plaintext algorithms without optimizations.

Runtimes. The benchmark result of our MPC-based DP mechanisms in LAN10, LAN1 and WAN100 can be found at Tab. 1, Tab. 9 and Tab. 10. We test multiple batch sizes in our experiments, i.e., the number of independent random values of DP noise generated in parallel. As the batch size increases, the overhead per sampled value decreases, so computation cost amortizes. Therefore, we choose the largest possible batch size that does not cause a memory overflow (which is denoted by "-" for no result).

Discrete Laplace/Gaussian Mechanisms. The upper part of Tab. 1 contains the runtime for the naive, optimized, and vulnerable [45] discrete Laplace mechanisms M_{DLap} (§4.1). The results in Tab. 1 show that M_{DLap} (§4.1) runs out of memory when trying to generate more than one random noise value in the 5PC setting. The offline runtime of our secure optimized M_{DLap} (§4.1) is $26.8 - 42.8\times$ slower than the insecure M_{DLap} [45]. Tab. 1 also presents the discrete Gaussian mechanism M_{DGauss} (§4.1). In our experiments, the memory restrictions of our hardware allow us to only generate Gaussian random value when $\sigma \leq 1$ (cf. §D.3).

Snapping and Integer-Scaling Laplace Mechanisms. In the lower half, Tab. 1 contains the runtimes of our MPC protocol for the snapping mechanism [77] M_{Snap} (Prot. 4) and the integer-scaling Laplace mechanism [55] M_{ISLap} (Prot. 5) as well as of the *vulnerable* version of the Laplace mechanism M_{Lap} [45] from PrivaDA. We implement all three mechanisms using floating-point arithmetic in $\{\mathsf{B}, \mathsf{Y}\}$, i.e., when the exact query result $f(D)$ are 64-bit floating-point numbers. In the 2PC setting, our optimized version of M_{Snap} has the best total runtime. Concretely, it is about 47% faster than our optimized M_{ISLap} and about 19% faster than the insecure M_{Lap} [45]. In the 5PC setting, our optimized M_{Snap} is 28% faster and and our optimized M_{ISLap} is 1.9× slower than the insecure M_{Lap} [45].

Integer-Scaling Gaussian Mechanism. The lower part of Tab. 1 also presents the runtime of our MPC-based integer-scaling mechanism M_{ISGauss} (Prot. 7) and the *insecure* Gaussian Mechanisms M_{Gauss} [71] of CrypTen. We implement these mechanisms using floating-point arithmetic in Y in the 2PC setting and in B in the 3PC and 5PC settings for efficiency reasons (cf. §F). Due to memory restrictions of our hardware, we were not able to run M_{ISGauss} with more than 2 parties and a batch size of 30 or with 5 parties and a batch size of 2. The offline runtime of our secure M_{ISGauss} is $44.5 - 287.1\times$ slower than the insecure M_{Gauss} [71].

Offline vs. Online Phase. Note that our MPC-based noise sampling protocols can fully be run offline before the actual input is available. Thus, in the time-critical online phase, for the discrete DP mechanisms (cf. §4.1) only a secure addition of the noise to the secret-shared function output has to be performed which is a single efficient local MPC operation. For the other three mechanisms (cf. §4.2-4.3), an additional rounding and scaling is needed, but this only needs between 9 and 55 ms in our MPC-based Laplace mechanisms (cf. Tab. 1) and between 8 ms and 1.1 seconds for the Gaussian mechanisms depending on the number of parties.

[3] https://github.com/liangzhao2048/Securely-Realizing-Output-Privacy-in-MPC-using-Differential-Privacy.git

[4] https://choosealicense.com/licenses/mit/

Table 1: Total runtimes (ms) per generated noise value in LAN10 (10-Gbit/s with 1ms RTT) averaged over 10 protocols runs of our MPC-based DP mechanisms using B and Y-sharing among N parties. ✓ are protocols not susceptible to the finite precision attacks discussed in §3, while ✗ are vulnerable solutions. The best secure results are marked in bold. - denotes memory overflow.

				N = 2			N = 3			N = 5		
	Mechanisms	Security	Batch	Prot.	Offline	Online	Prot.	Offline	Online	Prot.	Offline	Online
Discrete	M_{DLap} [45]	✗	40	Y	37.05	1.48	B	225.89	4.24	B	467.88	17.13
	M_{DLap} (cf. §4.1, naive)	✓	1	Y	14051.48	44.46	B	313628.29	376.21	B	472429.32	382.26
	M_{DLap} (cf. §4.1, optimized)	✓	1	Y	1606.00	37.71	B	12266.09	356.29	B	**17953.73**	**378.13**
	M_{DLap} (cf. §4.1, naive)	✓	5	Y	8690.86	4.72	B	36109.00	18.07	B	—	—
	M_{DLap} (cf. §4.1, optimized)	✓	5	Y	1273.36	5.55	B	**9662.55**	**15.46**	B	—	—
	M_{DLap} (cf. §4.1, naive)	✓	40	Y	7235.65	1.46	B	—	—	B	—	—
	M_{DLap} (cf. §4.1, optimized)	✓	40	Y	**991.78**	**1.41**	B	—	—	B	—	—
	M_{DGauss} (cf. §4.1)	✓	1	Y	6222.30	32.09	B	**12459.71**	**306.06**	B	**21169.71**	**414.90**
	M_{DGauss} (cf. §4.1)	✓	5	Y	**5584.64**	**31.48**	B	—	—	B	—	—
Continuous	M_{Lap} [45]	✗	30	Y	62.84	6.57	B	361.54	18.43	B	652.93	20.53
	M_{Snap} (cf. §4.2, naive)	✓	30	Y	72.05	38.64	B	275.51	128.55	B	685.93	167.41
	M_{Snap} (cf. §4.2, optimized)	✓	30	Y	**50.63**	10.20	B	**261.94**	33.56	B	**468.54**	44.28
	M_{ISLap} (cf. §4.3, naive)	✓	30	Y	817.72	55.07	B	12880.42	142.37	B	16836.56	224.10
	M_{ISLap} (cf. §4.3, optimized)	✓	30	Y	95.05	**9.52**	B	793.43	**32.61**	B	1239.24	**44.17**
	M_{Gauss} [71]	✗	30	Y	106.92	7.15	B	397.21	18.07	B	689.55	29.96
	M_{ISGauss} (cf. §4.3)	✓	2	Y	10696.00	408.37	B	97190.00	843.96	B	**197842.00**	**1141.75**
	M_{ISGauss} (cf. §4.3)	✓	4	Y	7836.50	166.82	B	**82901.92**	**492.96**	B	—	—
	M_{ISGauss} (cf. §4.3)	✓	30	Y	**4712.24**	**8.56**	B	—	—	B	—	—

Table 2: Total runtimes (ms) per generated noise value in LAN1 (1-Gbit/s with 1ms RTT) averaged over 10 protocols runs of our MPC-based DP mechanisms using B and Y-sharing among N parties. ✓ are protocols not susceptible to the finite-precision attacks discussed in §3, while ✗ are vulnerable solutions. The best secure results are marked in bold. - denotes memory overflow.

				N = 2			N = 3		
	Mechanisms	Security	Batch	Prot.	Offline	Online	Prot.	Offline	Online
Discrete	M_{DLap} [45]	✗	10	Y	196.03	5.44	B	823.13	23.73
	M_{DLap} (cf. §4.1, naive)	✓	10	Y	12 221.81	**4.55**	B	—	—
	M_{DLap} (cf. §4.1, optimized)	✓	10	Y	**4 707.46**	4.81	B	**13 474.59**	**22.33**
	M_{DGauss} (cf. §4.1)	✓	5	Y	**11 081.80**	**17.72**	B	**17 333.22**	**42.11**
Continuous	M_{Lap} [45]	✗	30	Y	68.19	6.94	B	380.91	22.65
	M_{Snap} (cf. §4.2, naive)	✓	30	Y	75.46	45.90	B	302.88	140.22
	M_{Snap} (cf. §4.2, optimized)	✓	30	Y	**56.13**	5.63	B	**291.73**	37.79
	M_{ISLap} (cf. §4.3, naive)	✓	30	Y	1 101.77	50.05	B	11 877.52	157.30
	M_{ISLap} (cf. §4.3, optimized)	✓	30	Y	373.54	**5.23**	B	1 114.22	**34.50**
	M_{Gauss} [71]	✗	30	Y	118.08	4.75	B	381.59	16.58
	M_{ISGauss} (cf. §E)	✓	30	Y	**14 219.45**	**6.10**	B	—	—

6 CONCLUSION

In this work, we introduced five MPC protocols that implement DP mechanisms that approximate discrete and continuous Laplace and Gaussian samples. Their different output formats and trade-offs between utility and efficiency enable a favorable selection based on the requirements of specific applications. In contrast to prior works that combine DP and MPC, our protocols are secure against finite precision attacks [57, 64, 77], which are even able to recover the entire database. Our MPC-based DP mechanisms transfer and optimize previous works for finite precision noise generation by Mironov [77], Canonne et al. [26], and Google [55] from the plaintext domain to the MPC setting. They offer extremely efficient online runtimes of only a few milliseconds, and the computation for sampling noise is largely independent of the function input and can be pre-computed.

Future Work. Besides random sampling, our MPC protocols and sub-protocols are deterministic, offering random samples from a specified distribution. If some additional inaccuracy in the result can be tolerated, approximating some expensive operations may improve the efficiency of our protocols. For example, piecewise

Table 3: Total runtimes (ms) per generated noise value in WAN100 (100-Mbit/s with 100ms RTT) averaged over 10 protocols runs of our MPC-based DP mechanisms using B and Y-sharing among N parties. ✓ are protocols not susceptible to the finite-precision attacks discussed in §3, while ✗ are vulnerable solutions. The best secure results are marked in bold. - denotes memory overflow.

				N = 2			N = 3		
	Mechanisms	Security	Batch	Prot.	Offline	Online	Prot.	Offline	Online
Discrete	M_{DLap} [45]	✗	10	Y	622.97	45.89	B	23 436.08	150.22
	M_{DLap} (cf. §4.1, naive)	✓	10	Y	88 752.83	**46.96**	B	—	—
	M_{DLap} (cf. §4.1, optimized)	✓	10	Y	**42 352.58**	47.99	B	**82 289.22**	**153.15**
	M_{DGauss} (cf. §4.1)	✓	5	Y	**66 792.65**	**86.63**	B	**89 240.04**	**355.12**
Continuous	M_{Lap} [45]	✗	30	Y	301.17	38.13	B	11 520.13	263.17
	M_{Snap} (cf. §4.2, naive)	✓	30	Y	266.34	127.13	B	7 977.09	4 640.57
	M_{Snap} (cf. §4.2, optimized)	✓	30	Y	**241.53**	42.89	B	**7 656.29**	572.26
	M_{ISLap} (cf. §4.3, naive)	✓	30	Y	6 862.04	123.97	B	71 937.68	4 477.84
	M_{ISLap} (cf. §4.3, optimized)	✓	30	Y	3 284.79	**41.00**	B	10 415.45	**427.07**
	M_{Gauss} [71]	✗	30	Y	370.97	41.90	B	10 360.31	262.90
	M_{ISGauss} (cf. §E)	✓	30	Y	**121 444.31**	**56.75**	B	—	—

polynomials might be used in A-sharing for floating-point exponentiation and division, building on the approach of [85]. New state-of-art circuit compilers, such as [39, 82], may also reduce the circuit sizes or multiplicative gate depth of our protocols, further improving efficiency. MPC protocols for noise generation secure against malicious adversaries, as well as protocols for other types of DP mechanisms also remain open directions for the future.

ACKNOWLEDGMENTS

This project received funding from: the Independent Research Fund Denmark under grant ID 10.46540/2032-00122B (YOSO); the ERC under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850990 PSOTI). It was co-funded by the DFG within SFB 1119 CROSSING/236615297 and GRK 2050 Privacy & Trust/251805230.

REFERENCES

[1] Martín Abadi, Andy Chu, Ian Goodfellow, H. Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. 2016. Deep Learning with Differential Privacy. In *CCS*.

[2] Abbas Acar, Berkay Celik, Hidayet Aksu, Selcuk Uluagac, and Patrick McDaniel. 2017. Achieving Secure and Differentially Private Computations in Multiparty Settings. In *PAC*.

[3] Gergely Ács and Claude Castelluccia. 2011. I Have a DREAM! (DiffeRentially privatE smArt Metering). In *Information Hiding (IH)*.

[4] Mehrdad Aliasgari, Marina Blanton, Yihua Zhang, and Aaron Steele. 2013. Secure Computation on Floating Point Numbers. In *NDSS*.

[5] Abdelrahaman Aly, Kelong Cong, D Cozzo, Marcel Keller, E Orsini, Dragos Rotaru, O Scherer, Peter Scholl, Nigel Smart, Titouan Tanguy, et al. 2021. Scale–mamba v1. 14: Documentation. https://homes.esat.kuleuven.be/~nsmart/SCALE/Documentation.pdf, Accessed 2022-09-29.

[6] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. 2016. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In *CCS*.

[7] David Archer, Shahla Atapoor, and Nigel Smart. 2021. The Cost of IEEE Arithmetic in Secure Computation. In *LATINCRYPT*.

[8] Gilad Asharov, Shai Halevi, Yehuda Lindell, and Tal Rabin. 2018. Privacy-Preserving Search of Similar Patients in Genomic Data. In *PETS*.

[9] Krishna Athreya and Soumendra Lahiri. 2006. *Measure theory and probability theory.* Springer Texts in Statistics.

[10] John Awoyemi, Adebayo Adetunmbi, and Samuel Oluwadare. 2017. Credit card fraud detection using machine learning techniques: A comparative analysis. In *International Conference on Computing Networking and Informatics (ICCNI)*.

[11] Borja Balle and Yu-Xiang Wang. 2018. Improving the Gaussian Mechanism for Differential Privacy: Analytical Calibration and Optimal Denoising. In *ICML*.

[12] Elaine Barker and John Kelsey. 2015. *Recommendation for Random Number Generation Using Deterministic Random Bit Generators.* Special Publication NIST.

[13] Amos Beimel, Kobbi Nissim, and Eran Omri. 2008. Distributed Private Data Analysis: Simultaneously Solving How and What. In *Advances in Cryptology – CRYPTO 2008*, David Wagner (Ed.). Springer Berlin Heidelberg.

[14] James Bell, Kallista Bonawitz, Adrià Gascón, Tancrède Lepoint, and Mariana Raykova. 2020. Secure Single-Server Aggregation with (Poly)Logarithmic Overhead. In *CCS*.

[15] Aner Ben-Efraim, Yehuda Lindell, and Eran Omri. 2016. Optimizing Semi-Honest Secure Multiparty Computation for the Internet. In *CCS*.

[16] Igor Bilogrevic, Julien Freudiger, Emiliano De Cristofaro, and Ersin Uzun. 2014. What's the Gist? Privacy-Preserving Aggregation of User Profiles. In *ESORICS*.

[17] Timm Birka, Kay Hamacher, Tobias Kussel, Helen Möllering, and Thomas Schneider. 2022. SPIKE: secure and private investigation of the kidney exchange problem. *BMC Medical Informatics Decision Making* (2022).

[18] Dan Bogdanov, Sven Laur, and Jan Willemson. 2008. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *ESORICS*.

[19] Jonas Böhler and Florian Kerschbaum. 2021. Secure Multi-party Computation of Differentially Private Heavy Hitters. In *CCS*.

[20] Joan Boyar and René Peralta. 2008. Tight bounds for the multiplicative complexity of symmetric functions. In *TCS*.

[21] Lennart Braun, Daniel Demmler, Thomas Schneider, and Oleksandr Tkachenko. 2022. MOTION–A Framework for Mixed-Protocol Multi-Party Computation. In *TOPS*.

[22] Karl Bringmann, Fabian Kuhn, Konstantinos Panagiotou, Ueli Peter, and Henning Thomas. 2014. Internal DLA: Efficient simulation of a physical growth model. In *International Colloquium on Automata, Languages, and Programming (ICALP)*.

[23] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. 2018. HyCC: Compilation of Hybrid Protocols for Practical Secure Computation. In *CCS*.

[24] Niklas Büscher, Andreas Holzer, Alina Weber, and Stefan Katzenbeisser. 2016. Compiling Low Depth Circuits for Practical Secure Computation. In *ESORICS*.

[25] David Byrd and Antigoni Polychroniadou. 2020. Differentially private secure multi-party computation for federated learning in financial applications. In *ICAIF*.

[26] Clément Canonne, Gautam Kamath, and Thomas Steinke. 2020. The Discrete Gaussian for Differential Privacy. In *NeurIPS*.

[27] Sílvia Casacuberta, Michael Shoemate, Salil Vadhan, and Connor Wagaman. 2022. Widespread Underestimation of Sensitivity in Differentially Private Libraries and How to Fix It. In *CCS*.

[28] Octavian Catrina and Amitabh Saxena. 2010. Secure Computation with Fixed-Point Numbers. In *FC*.

[29] Berkay Celik, David Lopez-Paz, and Patrick McDaniel. 2017. Patient-Driven Privacy Control through Generalized Distillation. In *PAC*.

[30] Jeffrey Champion, Abhi Shelat, and Jonathan R. Ullman. 2019. Securely Sampling Biased Coins with Applications to Differential Privacy. In *CCS*.

[31] Hubert Chan, Elaine Shi, and Dawn Song. 2012. Privacy-Preserving Stream Aggregation with Fault Tolerance. In *FC*.

[32] Melissa Chase, Ran Gilad-Bachrach, Kim Laine, Kristin E. Lauter, and Peter Rindal. 2017. Private Collaborative Neural Network Learning. Cryptology ePrint Archive, Paper 2017/762, https://eprint.iacr.org/2017/762.

[33] Albert Cheu and Chao Yan. 2022. Necessary Conditions in Multi-Server Differential Privacy. In *ITCS*.

[34] Christopher A. Choquette-Choo, Natalie Dullerud, Adam Dziedzic, Yunxiang Zhang, Somesh Jha, Nicolas Papernot, and Xiao Wang. 2021. CaPC Learning: Confidential and Private Collaborative Learning. In *ICLR*.

[35] Christian Covington. 2019. Snapping Mechanism Notes. https://github.com/ctcovington/floating_point/blob/master/snapping_mechanism/notes/snapping_implementation_notes.pdf, Accessed 2022-09-29.

[36] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. 2018. SPD$\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for Dishonest Majority. In *CRYPTO*.

[37] Aref Dajani, Amy Lauger, Phyllis Singer, Daniel Kifer, Jerome Reiter, Ashwin Machanavajjhala, Simson Garfinkel, Scot Dahl, Matthew Graham, Vishesh Karwa, Hang Kim, Philip Leclerc, Ian Schmutte, William Sexton, Lars Vilhuber, and John Abowd. 2020. The modernization of statistical disclosure limitation at the U.S. Census Bureau. U.S. Census Bureau. https://www2.census.gov/cac/sac/meetings/2017-09/statistical-disclosure-limitation.pdf

[38] Daniel Demmler, Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, and Shaza Zeitouni. 2015. Automated Synthesis of Optimized Circuits for Secure Computation. In *CCS*.

[39] Daniel Demmler, Stefan Katzenbeisser, Thomas Schneider, Tom Schuster, and Christian Weinert. 2021. Improved Circuit Compilation for Hybrid MPC via Compiler Intermediate Representation. In *SECRYPT*.

[40] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *NDSS*.

[41] Luc Devroye. 1986. *Non-Uniform Random Variate Generation.* Springer Book Archive.

[42] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. 2006. Our Data, Ourselves: Privacy Via Distributed Noise Generation. In *CRYPTO*.

[43] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam D. Smith. 2006. Calibrating Noise to Sensitivity in Private Data Analysis. In *TCC*.

[44] Cynthia Dwork and Aaron Roth. 2014. The Algorithmic Foundations of Differential Privacy. *Foundations and Trends in Theoretical Computer Science* (2014).

[45] Fabienne Eigner, Matteo Maffei, Ivan Pryvalov, Francesca Pampaloni, and Aniket Kate. 2014. Differentially private data aggregation with optimal utility. In *ACSAC*.

[46] Reo Eriguchi, Atsunori Ichikawa, Noboru Kunihiro, and Koji Nuida. 2021. Efficient Noise Generation to Achieve Differential Privacy with Applications to Secure Multiparty Computation. In *FC*.

[47] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. 2020. Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits. In *CRYPTO*.

[48] Vivian Fang, Lloyd Brown, William Lin, Wenting Zheng, Aurojit Panda, and Raluca Ada Popa. 2022. CostCO: An automatic cost modeling framework for secure multi-party computation. In *IEEE EuroS&P*.

[49] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. 2015. Model Inversion Attacks That Exploit Confidence Information and Basic Countermeasures. In *CCS*.

[50] Ivan Gazeau, Dale Miller, and Catuscia Palamidessi. 2013. Preserving differential privacy under finite-precision semantics. In *International Workshop on Quantitative Aspects of Programming Languages and Systems (QAPL).*

[51] Quan Geng, Peter Kairouz, Sewoong Oh, and Pramod Viswanath. 2015. The Staircase Mechanism in Differential Privacy. *IEEE Journal of Selected Topics in Signal Processing* (2015).

[52] Arpita Ghosh, Tim Roughgarden, and Mukund Sundararajan. 2009. Universally utility-maximizing privacy mechanisms. In *STOC*.

[53] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *STOC*.

[54] Maoguo Gong, Jialun Feng, and Yu Xie. 2020. Privacy-enhanced multi-party deep learning. *Neural Networks* (2020).

[55] Google. 2020. Secure noise generation. https://github.com/google/differential-privacy/blob/main/common_docs/Secure_Noise_Generation.pdf, Accessed 2022-09-29.

[56] Google. 2022. Google's differential privacy libraries. https://github.com/google/differential-privacy.git, Accessed 2022-09-29.

[57] Samuel Haney, Damien Desfontaines, Luke Hartman, Ruchit Shrestha, and Michael Hay. 2022. Precision-based attacks and interval refining: how to break, then fix, differential privacy on finite computers. *Journal of Privacy and Confidentiality* (2022).

[58] Jamie Hayes, Luca Melis, George Danezis, and Emiliano De Cristofaro. 2019. LOGAN: Membership Inference Attacks Against Generative Models. In *PETS*.

[59] Mikko Heikkilä, Eemil Lagerspetz, Samuel Kaski, Kana Shimizu, Sasu Tarkoma, and Antti Honkela. 2017. Differentially private Bayesian learning on distributed data. In *NeurIPS*.

[60] Muhammad Ishaq, Ana L. Milanova, and Vassilis Zikas. 2019. Efficient MPC via Program Analysis: A Framework for Efficient Optimal Mixing. In *CCS*.

[61] Matthew Jagielski, Jonathan Ullman, and Alina Oprea. 2020. Auditing Differentially Private Machine Learning: How Private is Private SGD?. In *NeurIPS*.

[62] Kimmo Järvinen, Helena Leppäkoski, Elena Simona Lohan, Philipp Richter, Thomas Schneider, Oleksandr Tkachenko, and Zheng Yang. 2019. PILOT: Practical Privacy-Preserving Indoor Localization Using OuTsourcing. In *IEEE EuroS&P*.

[63] Bargav Jayaraman, Lingxiao Wang, David Evans, and Quanquan Gu. 2018. Distributed Learning without Distress: Privacy-Preserving Empirical Risk Minimization. In *NeurIPS*.

[64] Jiankai Jin, Eleanor McMurtry, Benjamin Rubinstein, and Olga Ohrimenko. 2022. Are We There Yet? Timing and Floating-Point Attacks on Differential Privacy Systems. In *IEEE S&P*.

[65] Marc Joye and Benoît Libert. 2013. A Scalable Scheme for Privacy-Preserving Aggregation of Time-Series Data. In *FC*.

[66] Peter Kairouz, Ziyu Liu, and Thomas Steinke. 2021. The Distributed Discrete Gaussian Mechanism for Federated Learning with Secure Aggregation. In *ICML*.

[67] Seny Kamara and Mariana Raykova. 2011. Secure outsourced computation in a multi-tenant cloud. In *IBM Workshop on Cryptography and Security in Clouds*.

[68] Shiva Prasad Kasiviswanathan, Homin K. Lee, Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. 2008. What Can We Learn Privately?. In *FOCS*.

[69] Hannah Keller, Helen Möllering, Thomas Schneider, Oleksandr Tkachenko, and Liang Zhao. 2024. Secure Noise Sampling for DP in MPC with Finite Precision. In *ARES*.

[70] Marcel Keller. 2020. MP-SPDZ: A versatile framework for multi-party computation. In *CCS*.

[71] Brian Knott, Shobha Venkataraman, Awni Y. Hannun, Shubho Sengupta, Mark Ibrahim, and Laurens van der Maaten. 2021. CrypTen: Secure Multi-Party Computation Meets Machine Learning. In *NeurIPS*.

[72] D-U Lee, John D Villasenor, Wayne Luk, and Philip Heng Wai Leong. 2006. A hardware Gaussian noise generator using the Box-Muller method and its error analysis. *IEEE transactions on computers* (2006).

[73] Peter W. Markstein. 2008. The New IEEE-754 Standard for Floating Point Arithmetic. In *Numerical Validation in Current Hardware Architectures.*

[74] George Marsaglia and Thomas A Bray. 1964. A convenient method for generating normal variables. In *SIAM Review*.

[75] George Marsaglia and Wai Wan Tsang. 2000. The ziggurat method for generating random variables. *Journal of statistical software* (2000).

[76] Brendan McMahan, Galen Andrew, Ulfar Erlingsson, Steve Chien, Ilya Mironov, Nicolas Papernot, and Peter Kairouz. 2018. A General Approach to Adding Differential Privacy to Iterative Training Procedures. arXiv, http://arxiv.org/abs/1812.06210.

[77] Ilya Mironov. 2012. On significance of the least significant bits for differential privacy. In *CCS*.

[78] Payman Mohassel and Peter Rindal. 2018. ABY^3: A Mixed Protocol Framework for Machine Learning. In *CCS*.

[79] Payman Mohassel, Mike Rosulek, and Ni Trieu. 2020. Practical Privacy-Preserving K-means Clustering. In *PETS*.

[80] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE S&P*.

[81] Arvind Narayanan and Vitaly Shmatikov. 2008. Robust De-anonymization of Large Sparse Datasets. In *IEEE S&P*.

[82] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2021. SynCirc: Efficient Synthesis of Depth-Optimized Circuits for Secure Computation. In *HOST*.

[83] Martin Pettai and Peeter Laud. 2015. Combining Differential Privacy and Secure Multiparty Computation. In *CCS*.

[84] Vibhor Rastogi and Suman Nath. 2010. Differentially private aggregation of distributed time-series with transformation and encryption. In *SIGMOD*.

[85] Deevashwer Rathee, Anwesh Bhattacharya, Rahul Sharma, Divya Gupta, Nishanth Chandran, and Aseem Rastogi. 2022. SecFloat: Accurate Floating-Point meets Secure 2-Party Computation. In *IEEE S&P'*.

[86] Mike Rosulek and Lawrence Roy. 2021. Three Halves Make a Whole? Beating the Half-Gates Lower Bound for Garbled Circuits. In *CRYPTO*.

[87] Thomas Schneider and Michael Zohner. 2013. GMW vs. Yao? Efficient Secure Two-Party Computation with Low Depth Circuits. In *FC*.

[88] Adi Shamir. 1979. How to Share a Secret. *Commun. ACM* (1979).

[89] Elaine Shi, T.-H. Hubert Chan, Eleanor Gilbert Rieffel, Richard Chow, and Dawn Song. 2011. Privacy-Preserving Aggregation of Time-Series Data. In *NDSS*.

[90] Elaine Shi, T.-H. Hubert Chan, Eleanor Gilbert Rieffel, and Dawn Song. 2017. Distributed Private Data Analysis: Lower Bounds and Practical Constructions. *ACM Transactions on Algorithms* (2017).

[91] Reza Shokri and Vitaly Shmatikov. 2015. Privacy-preserving Deep Learning. In *CCS*.

[92] Latanya Sweeney. 1997. Weaving Technology and Policy Together to Maintain Confidentiality. *The Journal of Law, Medicine & Ethics* (1997).

[93] Jun Tang, Aleksandra Korolova, Xiaolong Bai, Xueqiang Wang, and Xiaofeng Wang. 2017. Privacy Loss in Apple's Implementation of Differential Privacy on MacOS 10.12. arXiv, http://arxiv.org/abs/1709.02753.

[94] Stacey Truex, Nathalie Baracaldo, Ali Anwar, Thomas Steinke, Heiko Ludwig, Rui Zhang, and Yi Zhou. 2019. A Hybrid Approach to Privacy-Preserving Federated Learning. In *CCS*.

[95] Salil Vadhan. 2017. The Complexity of Differential Privacy. In *Tutorials on the Foundations of Cryptography*. Springer International Publishing, 347–450.

[96] Alastair J Walker. 1974. Fast generation of uniformly distributed pseudorandom numbers with floating-point representation. *Electronics Letters* (1974).

[97] Royce Wilson, Celia Yuxin Zhang, William Lam, Damien Desfontaines, Daniel Simmons-Marengo, and Bryant Gipson. 2020. Differentially Private SQL with Bounded User Contribution. In *PETS*.

[98] Genqiang Wu, Yeping He, JingZheng Wu, and Xianyao Xia. 2016. Inherit Differential Privacy in Distributed Setting: Multiparty Randomized Function Computation. In *IEEE Trustcom/BigDataSE/ISPA*.

[99] Runhua Xu, Nathalie Baracaldo, Yi Zhou, Ali Anwar, and Heiko Ludwig. 2019. HybridAlpha: An Efficient Approach for Privacy-Preserving Federated Learning. In *Workshop on Artificial Intelligence and Security (AISec@CCS)*.

[100] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets. In *FOCS*.

[101] Lihua Yin, Jiyuan Feng, Hao Xun, Zhe Sun, and Xiaochun Cheng. 2021. A Privacy-Preserving Federated Learning for Multiparty Data Sharing in Social IoTs. *IEEE Transactions on Network Science and Engineering* (2021).

A STATISTICAL DISTRIBUTIONS

We define the following statistical distributions used in our work with their probability density functions:

(1) Geometric distribution: $\mathsf{Geo}\left(x \mid p\right) = (1-p)^x \cdot p$.

(2) Discrete Laplace distribution [52]:

$$\mathsf{DLap}\left(x \mid t\right) = \frac{e^{\frac{1}{t}} - 1}{e^{\frac{1}{t}} + 1} \cdot e^{-\frac{|x|}{t}}.$$

(3) Discrete Gaussian distribution [26]:

$$\mathsf{DGauss}\left(x \mid \mu, \sigma^2\right) = \frac{e^{-\frac{(x-\mu)^2}{2\sigma^2}}}{\sum_{y \in \mathbb{Z}} e^{-\frac{(y-\mu)^2}{2\sigma^2}}}.$$

(4) Binomial distribution:

$$\mathsf{Bino}\left(x \mid n, p\right) = \frac{n!}{x!\,(n-x)!} \cdot p^x \cdot (1-p)^{n-x}.$$

(5) Symmetrical binomial distribution:

$$\mathsf{SymmBino}\left(x \mid n, p = 0.5\right) = \mathsf{Bino}\left(x \mid n, p = 0.5\right) - \frac{n}{2}.$$

B MPC SUB-PROTOCOLS

We construct our MPC-based DP mechanism presented in §4 by using the following building blocks.

B.1 Prefix-OR.

The function $\left(\langle y_0 \rangle^{\mathsf{B}}, \dots, \langle y_{\ell-1} \rangle^{\mathsf{B}}\right) \leftarrow \mathsf{PreOr}\left(\langle x_0 \rangle^{\mathsf{B}}, \dots, \langle x_{\ell-1} \rangle^{\mathsf{B}}\right)$ outputs the secret-shares of $y_j = \vee_{k=0}^{j} x_k$ for $j \in [0, \ell-1]$, $y_0 = x_0$, i.e., output y_j is the prefix OR of the ℓ bits $x_0, \dots, x_{\ell-1}$. We re-implement the Prefix-OR protocol by Aly et al.[5] in MOTION.

B.2 Hamming Weight

The function $\left\langle y^{\mathbb{N}} \right\rangle^{\mathsf{B}} \leftarrow \mathsf{HW}\left(\langle x_0 \rangle^{\mathsf{B}}, \dots, \langle x_{\ell-1} \rangle^{\mathsf{B}}\right)$ computes the secret-shared Hamming weight (i.e., the number of bits equal to 1) of the ℓ input bits $x_0, \dots, x_{\ell-1}$. We use the protocol based on the plaintext algorithm from Boyar et al. [20].

B.3 Uniform Random Bits

The function $\left(\langle b_0 \rangle^{\mathsf{B}}, \dots, \langle b_{\ell-1} \rangle^{\mathsf{B}}\right) \leftarrow \mathsf{RandBits}\left(\ell\right)$ generates secret-shares of an ℓ-bit random string $b = (b_0, \dots, b_{\ell-1}) \in \{0,1\}^\ell$ hold by N parties. Specifically, each party P_i locally generates a random ℓ-bit string and sets it as its Boolean shares $\langle b \rangle_i^{\mathsf{B}} = \left(\langle b_0 \rangle_i^{\mathsf{B}}, \dots, \langle b_{\ell-1} \rangle_i^{\mathsf{B}}\right)$ of the secret-shared bit-string b, where $i \in \{1, \dots, \mathsf{N}\}$. The XOR of the independently generated Boolean shares $b = \oplus_{i=1}^{\mathsf{N}} \langle b \rangle_i^{\mathsf{B}}$ is uncorrelated with any input $\langle b \rangle_i^{\mathsf{B}}$ as long as at least one party is not corrupted.

B.4 Geometric Random Sampling

The function $\left\langle y^{\mathbb{N}} \right\rangle^{\mathsf{B}} \leftarrow \mathsf{Geometric}\left(\kappa\right)$ generates secret-shares of a random value $y \sim \mathsf{Geo}\left(p = 0.5\right)$ drawn from a geometric distribution (cf. §A), where κ is a security parameter, i.e, $\mathsf{Geometric}\left(\kappa\right)$ fails with a probability of $p = 1 - \frac{1}{2^\kappa}$. Prot. 6 is based on the plaintext sampling algorithm of Google's DP library [56].

```
Input  : $\kappa$ // Length of the input bit-string
Output: $\langle x^{\mathbb{N}} \rangle^{\mathsf{B}}$ // $x \sim \mathsf{Geo}\,(p = 0.5)$ or $x = \kappa$
$\left(\langle u_0 \rangle^{\mathsf{B}}, \ldots, \langle u_{\kappa-1} \rangle^{\mathsf{B}}\right) \leftarrow \mathsf{RandBits}\,(\kappa)$
$\left(\langle p_0 \rangle^{\mathsf{B}}, \ldots, \langle p_{\kappa-1} \rangle^{\mathsf{B}}\right) \leftarrow \mathsf{PreOr}\left(\langle u_0 \rangle^{\mathsf{B}}, \ldots, \langle u_{\kappa-1} \rangle^{\mathsf{B}}\right)$
$\left(\langle b_0 \rangle^{\mathsf{B}}, \ldots, \langle b_{\kappa-1} \rangle^{\mathsf{B}}\right) \leftarrow \left(\neg\left(\langle p_0 \rangle^{\mathsf{B}}\right), \ldots, \neg\left(\langle p_{\kappa-1} \rangle^{\mathsf{B}}\right)\right)$
$\langle x \rangle^{\mathsf{B},\mathbb{N}} \leftarrow \mathsf{HW}\left(\langle b_0 \rangle^{\mathsf{B}}, \ldots, \langle b_{\kappa-1} \rangle^{\mathsf{B}}\right)$
```

Protocol 6: Geometric — our MPC protocol realizing Geometric sampling [56].

B.5 Boolean-String Multiplication

The function $\left(\langle x_0 \rangle^{\mathsf{B}}, \ldots, \langle x_{\ell-1} \rangle^{\mathsf{B}}\right) \leftarrow \mathsf{BoolStrMul}\Big(\langle a \rangle^{\mathsf{B}}, \langle b_0 \rangle^{\mathsf{B}}, \ldots, \langle b_{\ell-1} \rangle^{\mathsf{B}}\Big)$ [8] computes the multiplication of one Boolean bit a by a set of ℓ Boolean bits $b_0, \ldots, b_{\ell-1}$, i.e., $x_0 = a \wedge b_0, \ldots, x_{\ell-1} = a \wedge b_{\ell-1}$.

B.6 Oblivious Selection

The function $\left(\langle y \rangle^{\mathsf{B}}, \langle c \rangle^{\mathsf{B}}\right) \leftarrow \mathsf{Sel}\Big(\langle x_0 \rangle^{\mathsf{B}}, \ldots, \langle x_{\ell-1} \rangle^{\mathsf{B}}, \langle c_0 \rangle^{\mathsf{B}}, \ldots, \langle c_{\ell-1} \rangle^{\mathsf{B}}\Big)$ outputs a bit-string $y = x_i$ and a bit $c = c_i$, where i is the index of the first non-zero bit c_i for $i \in [0, \ell - 1]$. If all bits $c_0, \ldots, c_{\ell-1}$ are 0, bit-string y is set to a string of 0s with the length of x_0, and bit c is set to 0 as well.

Sel deploys an inverted binary tree (inspired by Järvinen et al. and Mohassel et al. [62, 79] who use the structure for different functionalities) with depth $\lceil \log_2 \ell \rceil$, i.e., the leaves (as the 0-th layer) of the inverted binary tree represent ℓ-pair input elements $\left(\langle x_0 \rangle^{\mathsf{B}}, \langle c_0 \rangle^{\mathsf{B}}\right), \ldots, \left(\langle x_{\ell-1} \rangle^{\mathsf{B}}, \langle c_{\ell-1} \rangle^{\mathsf{B}}\right)$ and the root (as the last layer) is the (selected) output elements $\left(\langle y \rangle^{\mathsf{B}}, \langle c \rangle^{\mathsf{B}}\right)$. The inverted binary tree is evaluated from the 0-th layer (the layer with leaves) to the last layer (the root layer). Each intermediate node (between the leaves and root) $N_{(i,j)\to k}$ holds two elements $\left(\langle z_k \rangle^{\mathsf{B}}, \langle c_k \rangle^{\mathsf{B}}\right)$, where i and j are the index of the connected (intermediate or leaf) nodes in the upper layer and k is the index of node $N_{(i,j)\to k}$:

$$(z_k, c_k) = \begin{cases} (z_i, c_i), & \text{if } c_i == 1 \\ (z_j, c_j), & \text{if } c_i == 0 \text{ and } c_j == 1 \\ (0 \ldots, 0) & \text{if } c_i == 0 \text{ and } c_j == 0, \end{cases} \tag{6}$$

which is equivalent to

$$\begin{aligned} z_k &= \left((c_i \oplus c_j) \cdot \left((z_i \cdot c_i) \oplus (z_j \cdot c_j)\right)\right) \oplus \left((c_i \wedge c_j) \cdot z_i\right), \\ c_k &= \ c_i \oplus c_j \oplus (c_i \wedge c_j), \end{aligned} \tag{7}$$

where $z \cdot c$ denotes the multiplication (see Boolean-String Multiplication in §B.5) between a bit c and a bit-string z. Finally, the root node is evaluated in the same manner as the intermediate nodes and outputs $\left(\langle y \rangle^{\mathsf{B}}, \langle c \rangle^{\mathsf{B}}\right)$.

B.7 Uniform Random Floating-Point Numbers

The function $\left\langle u^{\mathbb{L}} \right\rangle^{\mathsf{B}} \leftarrow \mathsf{RandFloat}\,(l, k)$ generates secret-shares $\left\langle u^{\mathbb{L}} \right\rangle^{\mathsf{B}}$ of a random floating-point number $u \in [0, 1)$ (with a l-bit mantissa and a k-bit exponent) following the plaintext algorithms by Walker and Wu et al. [77, 96]. For instance, to generate B-shares of double-precision floating-point numbers $u \in [0, 1)$ (with $l = 53$, $k = 11$), each party generates a 52-bit random string $\langle d \rangle^{\mathsf{B}} \in \{0, 1\}^{52}$ and sets it as its secret-share of the mantissa of $\left\langle u^{\mathbb{L}} \right\rangle^{\mathsf{B}}$. Next, the party generates a geometric random value $\left\langle x^{\mathbb{Z}} \right\rangle^{\mathsf{B}}$ for $x \sim \mathsf{Geo}\,(0.5)$ (cf. §B.4) and sets $\left\langle e^{\mathbb{Z}} \right\rangle^{\mathsf{B}} = 1023 - \left(\left\langle x^{\mathbb{Z}} \right\rangle^{\mathsf{B}} + 1\right)$ as its share of the exponent of $\left\langle u^{\mathbb{L}} \right\rangle^{\mathsf{B}}$.

B.8 Multiplication with Power of Two

The function $\left\langle x^{\mathbb{L}} \right\rangle^{\mathsf{B}} \leftarrow \mathsf{MulPow2}\left(\left\langle \mathrm{a}^{\mathbb{L}} \right\rangle, \langle \mathrm{m} \rangle\right)$ computes the multiplication of a floating-point number a and 2^m for $m \in \mathbb{Z}$ in MPC. It first extracts the bits of the exponent $\left\langle e^{\mathbb{Z}} \right\rangle^{\mathsf{B}}$ of $\left\langle a^{\mathbb{L}} \right\rangle^{\mathsf{B}}$, computes $\left\langle E^{\mathbb{Z}} \right\rangle^{\mathsf{B}} = \left\langle e^{\mathbb{Z}} \right\rangle^{\mathsf{B}} + m$ as secure signed integer addition, and sets $\left\langle E^{\mathbb{Z}} \right\rangle^{\mathsf{B}}$ as the exponent's bits of the multiplication result $\left\langle x^{\mathbb{L}} \right\rangle^{\mathsf{B}}$, where $x = a \cdot 2^m$. Finally, the rest bits of $\left\langle x^{\mathbb{L}} \right\rangle^{\mathsf{B}}$ (i.e., mantissa and sign bits) are set the same value as $\left\langle a^{\mathbb{L}} \right\rangle^{\mathsf{B}}$. MulPow2 is more efficient than a secure floating-point multiplication as only a 16-bit integer addition is needed for 64-bit floating-point numbers.

C DP PROOFS

We present a proof sketch of Theorem 2.5.

Proof Sketch. Let $x \in D^n$ be an input dataset, where x_i is the input value contributed by client i and $\tilde{x}_i$ is the vector of input shares for server i that serves as input to Π. There is a mapping from $x \in D^n$ to $\tilde{x} \in (D^{n/m})^m$.

Each party learns nothing from an MPC protocol Π other than what is implied by the party's input and the function output. For every PPT adversary $\mathcal{A}$ controlling parties C, there is a simulator S such that $\mathsf{VIEW}_{\Pi}^{\mathcal{A}}(x)$ for any poly-time distinguisher T is indistinguishable from $S(\mathcal{M}(x), \{\tilde{x}_i\}_{i \in C})$. Since any $x \in D^n$ can be replaced with a neighboring $x' \in D^n$ that differs in a single row or x_i for party with x_i, we have:

$$\begin{aligned} &\Pr[T(\mathsf{VIEW}_{\Pi}^{\mathcal{A}}(x)) = 1] \\ &\quad \leq \Pr[T(S(\mathcal{M}(x), \{\tilde{x}_i\}_{i \in C}) = 1] + \mathsf{negl}(\kappa) \\ &\quad \leq (e^{\epsilon} \cdot \Pr[T(S(\mathcal{M}(x'), \{\tilde{x}'_i\}_{i \in C})) = 1] + \delta) + \mathsf{negl}(\kappa) \\ &\quad \leq (e^{\epsilon} \cdot (\Pr[T(\mathsf{VIEW}_{\Pi}^{\mathcal{A}}(x')) = 1] + \mathsf{negl}(\kappa)) + \delta) + \mathsf{negl}(\kappa) \\ &\quad \leq (e^{\epsilon} \cdot \Pr[T(\mathsf{VIEW}_{\Pi}^{\mathcal{A}}(x')) = 1] + \delta) + \mathsf{negl}(\kappa) + e^{\epsilon} \cdot \mathsf{negl}(\kappa) \end{aligned}$$

□

We present the proof of Theorem 4.1 (cf. §4.1).

Proof.

$$\begin{aligned} Pr[\mathcal{M}'(x) \in S] &= Pr[\mathcal{M}'(x) \in S \wedge E] + Pr[\mathcal{M}'(x) \in S \wedge \neg E] \\ &\leq Pr[E] + Pr[\neg E] \cdot Pr[\mathcal{M}'(x) \in S | \neg E] \\ &= p_{\text{failure}} + Pr[\neg E] \cdot Pr[\mathcal{M}'(x) \in S | \neg E] \\ &\leq p_{\text{failure}} + Pr[\neg E](Pr[\mathcal{M}'(x') \in S | \neg E] + \delta) \\ &\leq p_{\text{failure}} + e^{\epsilon} Pr[\mathcal{M}'(x') \in S \wedge \neg E] + (1 - p_{\text{failure}})\delta \\ &\leq e^{\epsilon} Pr[\mathcal{M}'(x') \in S] + p_{\text{failure}} + \delta \end{aligned}$$

□

Table 4: Optimization of the pre-computed parameters s, t, κ_1, κ_2 in Prot. 1. Given failure probability p_{fail} $(\mathsf{GeometricExp}, \mathsf{Prot.\ 1}) < 2^{-40}$, the min. values for κ_1, κ_2 w.r.t different $\frac{s}{t}$ values are given. We test 1 000 random values each for $\frac{s}{t}$ from $[0, 100]$, $\mathbb{Z}^+$, and the set $\{0.125, 0.25, 0.5, 0.75, 1.5, 2.5\}$. κ_1^* and κ_2^* are the maximal values tested. Best values of κ_1 and κ_2 are bold.

$\frac{s}{t}$	s	t	κ_1	κ_2
$(0, 100]$	—	—	28^*	30^*
$\mathbb{Z}^+$	—	1	**0**	**28**
0.125	1	8	25	30
0.25	1	4	23	29
0.5	1	2	18	**28**
0.75	3	4	23	29
1.5	3	2	18	**28**
2.5	5	2	23	29

D FAILURE PROBABILITY DETERMINATION AND EFFICIENCY OPTIMIZATION

As discussed in §4.1, we fix the number of iterations of our MPC-based DP mechanism in advance before executing the protocol to achieve (1) feasible efficiency as well as (2) full computational privacy, i.e., no leakage about the magnitude of the sampled random value. To ensure a negligible failure probability $p_{\text{failure}} < 2^{-40}$, we determine the number of iterations required for each protocol in the following.

D.1 Geometric

For sampling a random value from the geometric distribution in Prot. 1, we first derive the failure probability. It is then used to select the parameter values of Prot. 1 such that it fails with negligible probability and causes minimal computation costs.

Failure Probability. We compute the failure probability of Prot. 1 as follows: Let A_{κ_1} be the event that the first for-loop (Lines 6-9 of Prot. 1) fails to generate a secret-shared bit $b = 1$ (line 10 in Prot. 1) within κ_1 iterations and B_{κ_2} is the event that the second for-loop (Line 12-15 of Prot. 1) fails to generate $d = 1$ (line 16 in Prot. 1) within κ_2 iterations. As both loops are independent, we have:

$$\begin{aligned} p_{\text{fail}}\left(\mathsf{GeometricExp},\ \mathsf{Prot.\ 1}\right) &= p\left(A_{\kappa_1} \vee B_{\kappa_2}\right) \\ &= p\left(A_{\kappa_1}\right) + p\left(B_{\kappa_2}\right) - p\left(A_{\kappa_1} \wedge B_{\kappa_2}\right) \\ &= p\left(A_{\kappa_1}\right) + p\left(B_{\kappa_2}\right) - p\left(A_{\kappa_1}\right) \cdot p\left(B_{\kappa_2}\right) \\ &= \left(1 - \frac{1}{t}\frac{1-e^{-1}}{1-e^{-\frac{1}{t}}}\right)^{\kappa_1} + e^{-\kappa_2} \\ &\quad - \left(1 - \frac{1}{t}\frac{1-e^{-1}}{1-e^{-\frac{1}{t}}}\right)^{\kappa_1} \cdot e^{-\kappa_2}, \end{aligned} \tag{8}$$

where

$$\begin{aligned} p\left(A_{\kappa_1}\right) &= \prod_{i=1}^{\kappa_1} p\left(A_1\right) \\ &= \prod_{i=1}^{\kappa_1} \left(\sum_{k=0}^{t-1} p\left(u = k\right) \cdot p\left(b_0 = 0\right)\right) \\ &= \prod_{i=1}^{\kappa_1} \left(\sum_{k=0}^{t-1} \frac{1}{t} \cdot \left(1 - e^{-\frac{k}{t}}\right)\right) \\ &= \prod_{i=1}^{\kappa_1} \left(1 - \frac{1}{t}\sum_{k=0}^{t-1} e^{-\frac{k}{t}}\right) \\ &= \prod_{i=1}^{\kappa_1} \left(1 - \frac{1}{t}\frac{1-e^{-1}}{1-e^{-\frac{1}{t}}}\right) \\ &= \left(1 - \frac{1}{t}\frac{1-e^{-1}}{1-e^{-\frac{1}{t}}}\right)^{\kappa_1}, \end{aligned} \tag{9}$$

and

$$\begin{aligned} p\left(B_{\kappa_2}\right) &= \prod_{i=1}^{\kappa_2} p\left(B_1\right) \\ &= \prod_{i=1}^{\kappa_2} p\left(c_0 = 1\right) \\ &= e^{-\kappa_2}. \end{aligned} \tag{10}$$

Optimization. Recall that Prot. 1 generates geometric random values $X \sim \mathrm{Geo}\left(1 - e^{-\frac{s}{t}}\right)$, where s, t are positive integers. The efficiency of Prot. 1 can be improved if t is a power of 2 and κ_1 is small as we will show in the following.

Our micro-benchmark results in §F show that modular reduction based $\mathsf{RandInt}\left(t\right)$ [12] and floating-point exponentiations (Lines 7 and 8 in Prot. 1) are the most expensive primitives of Prot. 1 from a computational point of view. But, as discussed in §4.1, when $t = 2^k$ for $k \in \mathbb{Z}$, $\mathsf{RandInt}\left(t\right)$ becomes a local operation in MPC.

Additionally, we can further reduce the computational cost by requiring a smaller number of iterations κ_1 (line 6 in Prot. 1) as those determine the number of exponentiations. Following this idea, we exhaustively compute all combinations of values for κ_1, κ_2, s and $t = 2^k$ that achieve $p_{\text{fail}}\left(\mathsf{GeometricExp},\ \mathsf{Prot.\ 1}\right) < 2^{-40}$ (cf. Tab. 4) using Eq. 8 and choose the configuration with the smallest κ_1. Note that $\kappa_1 = 0$ means that the iteration from lines $6 - 9$ in Prot. 1 can be skipped.

D.2 Discrete Laplace

For sampling a random value from the discrete Laplace distribution in Prot. 2, we first derive the protocol's failure probability. It is then used to select the parameter values of Prot. 2 s.t. it fails with negligible probability and minimizes runtime.

Failure Probability. We compute the failure probability of Prot. 2 as follows: Suppose A_{κ_3} is the event that Prot. 2 fails to output $S = 1$ (Line 6 in Prot. 2) within κ_3 iterations. Since each iteration is

Table 5: Optimization of the pre-computed parameters s, t, κ_1, κ_2, κ_3 in Prot. 2. Given failure probability $p_{\text{failure}} < 2^{-40}$ of Prot. 1- 2, the min. values for $\kappa_1, \kappa_2, \kappa_3$ w.r.t. different $\frac{s}{t}$ values are given. Δf and ε are the sensitivity and DP parameter of discrete Laplace mechanisms that use the Laplace random value as DP noise generated with Prot. 2. We test 1 000 random values each for $\frac{s}{t}$ from $(0, 5]$, $(5, 10]$, $(10, 10000]$, $\mathbb{Z}^+$, and the set $\{0.125, 0.25, 0.5, 0.75, 1.5, 2.5\}$. $\kappa_i^*, i \in [3]$ are the maximal values tested. Best values of κ_1, κ_2, κ_3 are bold.

$\frac{s}{t} = \frac{\Delta f}{\varepsilon}$	s	t	κ_1	κ_2	κ_3	$\kappa_1 * \kappa_3$
$(0, 5]$	—	—	28^*	30^*	25^*	672
$(5, 10]$	—	—	28^*	30^*	40^*	1120
$(10, 10000]$	—	—	28^*	30^*	41^*	1148
$\mathbb{Z}^+$	—	1	**0**	**28**	41^*	0
0.125	1	8	25	30	**10**	250
0.25	1	4	23	29	13	299
0.5	1	2	18	**28**	18	324
0.75	3	4	23	29	21	483
1.5	3	2	18	**28**	30	540
2.5	5	2	18	**28**	36	648

Table 6: Optimization of the pre-computed parameters σ, κ_1, κ_2, κ_3, κ_4 in Prot. 3. Given failure probability $p_{\text{failure}} < 2^{-40}$ of Prot. 1-3, the min. values of κ_1, κ_2, κ_3, κ_4 w.r.t. different σ values are given. σ is a parameter of the discrete Gaussian distribution (cf. §A). We test 1 000 random values each for σ from $(0, 1)$, $[1, 2]$, $(2, 5]$, $(5, 10]$ and $(10, 100]$. $\kappa_i^*, i \in [4]$ are the maximal values tested. Best values of κ_1, κ_2, κ_3, κ_4 are bold.

σ	κ_1	κ_2	κ_3	κ_4
$(0, 1)$	**0**	**28***	25^*	48^*
$[1, 2]$	28^*	**28***	18^*	36^*
$(2, 5]$	28^*	30^*	15^*	25^*
$(5, 10]$	28^*	30^*	11^*	21^*
$(10, 100]$	28^*	30^*	**9***	**20***

independent, we estimate $p\left(A_{\kappa_3}\right)$ as follows:

$$\begin{aligned} p_{\text{fail}}\left(\text{DiscreteLaplace, Prot. 2}\right) &= p\left(A_{\kappa_3}\right) \\ &= \prod_{i=1}^{\kappa_3} p\left(A_1\right), \end{aligned} \tag{11}$$

where

$$\begin{aligned} p\left(A_1\right) = p\left(f_0 = 1\right) & \\ &= p\left(S_0 = 1\right) \cdot p\left(m_0 = 0 \wedge \text{Prot. 1 succeeds}\right) + p\left(\text{Prot. 1 fails}\right) \\ &= \frac{1}{2} \cdot \left(1 - e^{-\frac{s}{t}}\right) \cdot p\left(\text{Prot. 1 succeeds}\right) + p\left(\text{Prot. 1 fails}\right) \\ &= \frac{1}{2} \cdot \left(1 - e^{-\frac{s}{t}}\right) \cdot \left(1 - p_{\text{fail}}\left(\text{GeometricExp, Prot. 1}\right)\right) \\ &\quad + p_{\text{fail}}\left(\text{GeometricExp, Prot. 1}\right). \end{aligned} \tag{12}$$

Optimization. Prot. 1 is the most expensive step in Prot. 2. The number of iterations κ_1 and κ_2 in Prot. 1 are fixed to ensure obliviousness while they must be large enough that it only fails with negligible probability. Those parameter values will also heavily influence the efficiency of Prot. 2 as already discussed in §D.1. Using Tab. 4 and Eq. 11, we can compute Tab. 5 to determine the optimal parameter values for Prot. 2.

D.3 Discrete Gaussian

For sampling a random value from the discrete Gaussian distribution in Prot. 3, we first derive the protocol's failure probability. It is then used to select the parameter values of Prot. 3 s.t. it fails with negligible probability and minimizes runtime.

Failure Probability. Suppose A_{κ_4} is the event that Prot. 3 fails to output $b = 1$ within κ_4 iterations. Since the iterations are independent, the failure probability can be computed as follows:

$$\begin{aligned} p_{\text{fail}}\left(\text{DiscreteGaussian, Prot. 3}\right) &= p\left(A_{\kappa_4}\right) \\ &= \prod_{i=1}^{\kappa_4} p\left(A_1\right), \end{aligned} \tag{13}$$

where

$$\begin{aligned} p\left(A_1\right) &= \sum_{j=-\infty}^{\infty} p\left(b_0 = 0 \wedge Y_0 = j \wedge \text{Prot. 2 succeeds}\right) \\ &\quad + p\left(\text{Prot. 2 fails}\right) \\ &= \sum_{j=-\infty}^{\infty} p\left(b_0 = 0\right) \cdot p\left(Y_0 = j\right) \cdot p\left(\text{Prot. 2 succeeds}\right) \\ &\quad + p\left(\text{Prot. 2 fails}\right) \\ &= \sum_{j=-\infty}^{\infty} \left(1 - e^{-\frac{\left(|j| - \frac{\sigma^2}{t}\right)^2}{2\sigma^2}}\right) \cdot \frac{\left(e^{\frac{1}{t}} - 1\right) \cdot e^{-\frac{|j|}{t}}}{e^{\frac{1}{t}} + 1} \\ &\quad \cdot p\left(\text{Prot. 2 succeeds}\right) + p\left(\text{Prot. 2 fails}\right) \\ &= \sum_{j=-\infty}^{\infty} \left(1 - e^{-\frac{\left(|j| - \frac{\sigma^2}{t}\right)^2}{2\sigma^2}}\right) \cdot \frac{\left(e^{\frac{1}{t}} - 1\right) \cdot e^{-\frac{|j|}{t}}}{e^{\frac{1}{t}} + 1} \\ &\quad \cdot \left(1 - p_{\text{fail}}\left(\text{DiscreteLaplace, Prot. 2}\right)\right) \\ &\quad + p_{\text{fail}}\left(\text{DiscreteLaplace, Prot. 2}\right). \end{aligned} \tag{14}$$

Optimization. Prot. 2 is the most expensive step in Prot. 3 whose runtime itself is dominated by the number of iterations κ_4. κ_4 has to be set based on the required failure probability of Prot. 2-3, cf. Tab. 6.

E MPC BASED INTEGER-SCALING GAUSSIAN MECHANISM

The integer-scaling Gaussian mechanism [55] is (ε, δ)-differentially private thanks to scaled noise sampled from a symmetric binomial distribution (§A): SymmBino $(n, p = 0.5)$. The scaling factor r and distribution parameter n can be estimated using ε and δ [11, 56]. Prot. 7 presents our MPC protocol for the integer-scaling Gaussian mechanism M_{ISGauss}, an adapted version of the plaintext symmetrical binomial sampling algorithm in [55]. It calls Prot. 8 to generate secret-shares of a symmetrical binomial random value $\langle i \rangle^{\mathsf{B},\mathbb{L}}$. Given the value of n, we first compute the following constant parameters:

$$\begin{aligned}
m &= \left\lfloor \sqrt{2} \cdot \sqrt{n} + 1 \right\rfloor, \\
x_{\text{min}} &= -\frac{\sqrt{n} \cdot \sqrt{\ln \sqrt{n}}}{\sqrt{2}}, \\
x_{\text{max}} &= -x_{\text{min}}, \\
v_n &= \frac{0.4 \cdot 2^{1.5} \cdot \ln^{1.5}(\sqrt{n})}{\sqrt{n}}, \\
\tilde{p}_{\text{coe}} &= \sqrt{\frac{2}{\pi}} \cdot (1 - v_n) \cdot \frac{1}{\sqrt{n}}.
\end{aligned} \tag{15}$$

Failure Probability. For sampling a symmetrical binomial random value from the symmetrical binomial distribution in Prot. 8, we derive the protocol's failure probability based on [22] as follows: Suppose A_κ is the event that Prot. 8 fails to output $f_i == 1$ within κ iterations. Since each iteration is independent, we compute

$$\begin{aligned}
p_{\text{fail}}(\text{SymmetricalBinomial, Prot. 7}) &= \prod_{1}^{\kappa} p\left(A_1\right) \\
&= \left(\frac{15}{16}\right)^{\kappa}.
\end{aligned} \tag{16}$$

To guarantee $p_{\text{fail}}(\text{SymmetricalBinomial, Prot. 7}) < 2^{-40}$, κ should be greater than 430.

```
// Secret-shared exact query result, resolution parameter of
// $M_{\text{ISGauss}}$, parameter of SymmBino $(n, p = 0.5)$
Input  : $\langle f(D)^{\mathbb{L}} \rangle^{\mathsf{B}}, r, \sqrt{n}$
// Secret-shared DP query result
Output : $\langle y^{\mathbb{L}}_{\text{ISGauss}} \rangle^{\mathsf{B}}$
$\left\langle i^{\mathbb{N}}_{\text{SymmBinoNoise}} \right\rangle^{\mathsf{B}} \leftarrow \text{SymmetricalBinomial}(\sqrt{n})$
$\langle Y^{\mathbb{L}}_{\text{GaussNoise}} \rangle^{\mathsf{B}} \leftarrow \text{MulPow2}\left(\text{UINT2FL}\left(\langle i^{\mathbb{N}} \rangle^{\mathsf{B}}\right), k = \log_2 r\right)$
$\langle f_r(D)^{\mathbb{L}} \rangle^{\mathsf{B}} \leftarrow \left\lfloor \langle f(D)^{\mathbb{L}} \rangle^{\mathsf{B}} \right\rceil_r$
$\langle y^{\mathbb{L}}_{\text{ISGauss}} \rangle^{\mathsf{B}} \leftarrow \langle f_r(D)^{\mathbb{L}} \rangle^{\mathsf{B}} + \langle Y^{\mathbb{L}}_{\text{GaussNoise}} \rangle^{\mathsf{B}}$
```

Protocol 7: M_{ISGauss} — our MPC protocol realizing the integer-scaling Gaussian mechanism [55].

F MPC MICRO-BENCHMARK RESULTS

To instantiate our general MPC protocols in §4 in the most efficient manner, we micro-benchmark all relevant sub-protocols in {Y, B, A}-sharing. Results for 64-bit integer arithmetic and 64-bit floating-point arithmetic are given in Tab. 11. Note that the floating-point arithmetic operations (e.g., exponentiation and natural logarithm) in A that are more than 500× slower than that in {B, Y} are not listed in Tab. 11. Benchmark results for share conversions and other primitive operations (e.g., XOR and AND) in different network environments are given in [21].

G OVERVIEW OF NOISE GENERATION PROCEDURE.

Fig. 1 shows the noise generation procedure for the secure and insecure DP mechanisms (cf. §4 and §5).

```
Input  : $\sqrt{n}$ // Parameter of SymmBino $(n, p = 0.5)$
Output : $\langle i^{\mathbb{Z}} \rangle^{\mathsf{B}}$ // $i \sim$ SymmBino $(n, p = 0.5)$ or $i = 0$
for $j \leftarrow 0$ to $\kappa - 1$ do
    $\langle s_j{}^{\mathbb{Z}} \rangle^{\mathsf{B}} \leftarrow \text{Geometric}(0.5)$
    $\left\langle S^{\mathbb{Z}}_j \right\rangle^{\mathsf{B}} \leftarrow -\left(\left\langle s^{\mathbb{Z}}_j \right\rangle^{\mathsf{B}} + 1\right)$
    $\langle b_j \rangle^{\mathsf{B}} \leftarrow \text{RandBits}(1)$
    $\left\langle k^{\mathbb{Z}}_j \right\rangle^{\mathsf{B}} \leftarrow \text{MUX}\left(\langle b_j \rangle^{\mathsf{B}}, \left\langle s^{\mathbb{Z}}_j \right\rangle^{\mathsf{B}}, \left\langle S^{\mathbb{Z}}_j \right\rangle^{\mathsf{B}}\right)$
    $\langle l_j{}^{\mathbb{Z}} \rangle^{\mathsf{B}} \leftarrow \text{RandInt}(m)$
    $\left\langle x^{\mathbb{Z}}_j \right\rangle^{\mathsf{B}} \leftarrow \left\langle k^{\mathbb{Z}}_j \right\rangle^{\mathsf{B}} \cdot m + \left\langle l^{\mathbb{Z}}_j \right\rangle^{\mathsf{B}}$
    $\left\langle cond_{x_{\text{min}} \leq x_j \leq x_{\text{max}}} \right\rangle^{\mathsf{B}} \leftarrow \left(\left\langle x^{\mathbb{Z}}_j \right\rangle^{\mathsf{B}} \geq x_{\text{min}}\right) \wedge \left(\left\langle x^{\mathbb{Z}}_j \right\rangle^{\mathsf{B}} \leq x_{\text{max}}\right)$
    $\left\langle \tilde{p}^{\mathbb{L}}_j \right\rangle^{\mathsf{B}} \leftarrow \tilde{p}_{\text{coe}} \cdot e^{-\left(\frac{\sqrt{2}}{\sqrt{n}} \cdot \text{INT2FL}\left(\left\langle x^{\mathbb{Z}}_j \right\rangle^{\mathsf{B}}\right)\right)^2}$
    $\left\langle cond_{\tilde{p}_j > 0} \right\rangle^{\mathsf{B}} \leftarrow \left\langle cond_{x_{\text{min}} \leq x_j \leq x_{\text{max}}} \right\rangle^{\mathsf{B}}$
    $\langle p^{\mathbb{L}}_{\text{Bern}} \rangle^{\mathsf{B}} \leftarrow \left\langle \tilde{p}^{\mathbb{L}}_j \right\rangle^{\mathsf{B}} \cdot \left(\left(2^{\text{UINT2FL}\left(\left\langle s^{\mathbb{N}}_j \right\rangle^{\mathsf{B}}\right)}\right) \cdot \frac{m}{4}\right)$
    $\langle c_j \rangle^{\mathsf{B}} \leftarrow \text{Bernoulli}\left(\langle p^{\mathbb{L}}_{\text{Bern}} \rangle^{\mathsf{B}}\right)$
    $\left\langle cond_{c_j == 1} \right\rangle^{\mathsf{B}} \leftarrow \langle c_j \rangle^{\mathsf{B}}$
    $\langle f_j \rangle^{\mathsf{B}} \leftarrow \left\langle cond_{x_{\text{min}} \leq x_j \leq x_{\text{max}}} \right\rangle^{\mathsf{B}} \wedge \left\langle cond_{c_j == 1} \right\rangle^{\mathsf{B}}$
end
$\langle i^{\mathbb{Z}} \rangle^{\mathsf{B}} \leftarrow \text{Sel}\left(\langle x_0{}^{\mathbb{Z}} \rangle^{\mathsf{B}}, \ldots, \langle x^{\mathbb{Z}}_{\kappa-1} \rangle^{\mathsf{B}}, \langle f_0 \rangle^{\mathsf{B}}, \ldots, \langle f_{\kappa-1} \rangle^{\mathsf{B}}\right)$
```

Protocol 8: SymmetricalBinomial — our MPC protocol realizing symmetrical Binomial sampling [55].

H RUNTIME BREAKDOWN OF MPC-BASED DP MECHANISMS

To show the bottleneck of our MPC-based DP mechanisms, we analyze the runtime breakdown of the *secure* DP mechanisms presented in §4 and §E, in the 2PC setting and LAN10 network environment. Fig. 2a shows the runtime breakdown of the geometric sampling protocol GeometricExp (cf. Prot. 1 in §4.1). We omit the runtime breakdown of the discrete Laplace mechanism M_{DLap}, discrete Gaussian mechanisms M_{DGauss}, and the integer-scaling Laplace M_{ISLap} as GeometricExp (§4.1) is the major overhead (>98.3% in the runtimes). Fig. 2b shows the runtime breakdown of the snapping mechanism M_{Snap} (cf. Prot. 4 in §4.2). Fig. 2c shows the runtime breakdown of the integer-scaling Gaussian mechanism M_{ISGauss} (cf. Prot. 7 in §E).

I COMPLEXITY ANALYSIS AND COMMUNICATION

In this section, we analyze the complexity of our MPC protocols in terms of circuit size and depth. For Y-sharing, the round complexity is constant such that the circuit size, i.e., the number of AND gates, determines performance. In contrast, the circuit depth, which is the longest path of multiplicative gates such as AND and OR, determines the number of communication rounds in B-sharing, which is a dominant factor in its runtime complexity in most cases.

Our analysis results are given in Tab. 8. Overall, our optimized protocols for the Laplace mechanisms save up to 54% of AND gates

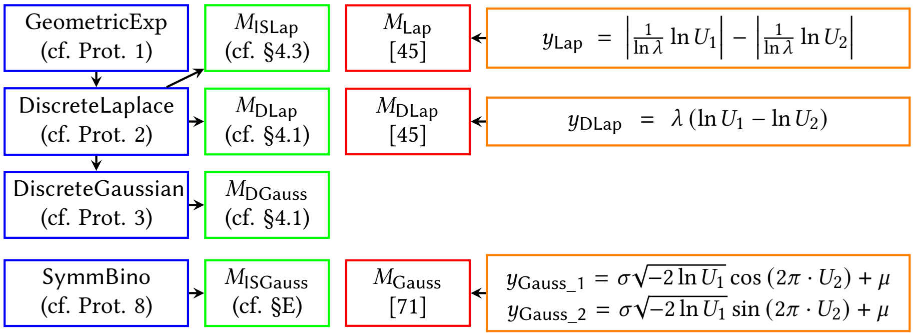
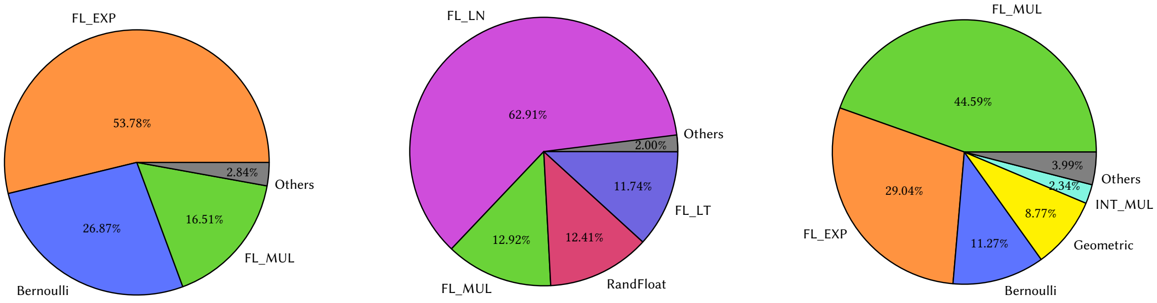

GeometricExp (cf. Prot. 1)
M_{ISLap} (cf. §4.3)
M_{Lap} [45]
$y_{\text{Lap}} = \left|\frac{1}{\ln\lambda}\ln U_1\right| - \left|\frac{1}{\ln\lambda}\ln U_2\right|$
DiscreteLaplace (cf. Prot. 2)
M_{DLap} (cf. §4.1)
M_{DLap} [45]
$y_{\text{DLap}} = \lambda(\ln U_1 - \ln U_2)$
DiscreteGaussian (cf. Prot. 3)
M_{DGauss} (cf. §4.1)
SymmBino (cf. Prot. 8)
M_{ISGauss} (cf. §E)
M_{Gauss} [71]
$y_{\text{Gauss\_1}} = \sigma\sqrt{-2\ln U_1}\cos(2\pi \cdot U_2) + \mu$
$y_{\text{Gauss\_2}} = \sigma\sqrt{-2\ln U_1}\sin(2\pi \cdot U_2) + \mu$

Figure 1: Blocks are the secure and Blocks are the insecure noise sampling algorithms. Blocks and Blocks are the corresponding DP mechanisms. We omit the snapping mechanism M_{Snap} (cf. §4.2) because it is based on M_{Lap} [45] but with additional processing steps to guarantee DP security. All noise sampling algorithms use uniform random bits (cf. §B) or uniform random floating-point numbers (cf. §B), e.g., U_1 and U_2, as the source of randomness. λ, σ, and μ depend on the DP parameters ϵ and δ.



(a) GeometricExp (cf. Prot. 1 in §4.1)

(b) M_{Snap} (cf. Prot. 4 in §4.2)

(c) M_{ISGauss} (cf. Prot. 7 in §E)

Figure 2: Runtime breakdown (in percentage %) of the geometric sampling protocol GeometricExp (cf. Prot. 1 in §4.1) with $\kappa_1 = 25$ and $\kappa_2 = 30$, the snapping mechanism M_{Snap} (cf. Prot. 4 in §4.2), and the integer-scaling Gaussian mechanism M_{ISGauss} (cf. Prot. 7 in §E) in LAN10 (10-Gbit/s with 1ms RTT) averaged over 10 protocols runs using Y-sharing among two parties. Others **are operations that take < 2% of the total runtime.**

in Y-sharing and reduce the depth of the circuit by 36 − 88% in B-sharing compared to the naive protocols.

For the discrete Laplace mechanism, our optimized M_{DLap} (cf. §4.1) requires 191× the number of AND gates compared to the *vulnerable* M_{DLap} in the 2PC setting, but its depth of AND and MUX gates is 29% less than M_{DLap} [45] with N ≥ 3 parties. In N-PC settings, the circuit depth of the discrete Gaussian mechanism M_{DGauss} (§4.1) is smaller than the discrete Laplace variants, but our runtime benchmarks in Tab. 1 show it is still slower than those in practice. The reason is the parallelized circuit construction, i.e., M_{DGauss} (cf. Prot. 3 in §4.1) heavily relies on SIMD to guarantee a negligible failure probability such that communication time becomes a dominant factor slowing down the execution. Our optimized M_{Snap} protocol (cf. Prot. 4 in §4.2) reduces the number of AND gates by about 48% compared to the vulnerable M_{Lap} [45] in the 2PC setting. With N ≥ 3 parties, the depth of our optimized M_{ISLap} (cf. Prot. 5 in §4.3) is reduced by 52% compared to the vulnerable M_{Lap} [45]. For the Gaussian mechanism, our private M_{ISGauss} (cf. Prot. 7 in §E) requires 547× the number of AND gates compared to the *vulnerable* M_{Gauss} [71] in the 2PC setting and its depth is 3.78× larger than M_{Gauss} [71] with N ≥ 3 parties.

Communication Cost. The communication benchmark results of our MPC-based DP mechanism protocols are given in Tab. 7. M_{Snap} is most efficient requiring only about 5.3 MB per query in the 2PC setting, while M_{DGauss} requires 1.5-2 GB of communication.

J PERFORMANCE OF OUR MPC PROTOCOLS IN LAN1 AND WAN10

Tab. 9 and Tab. 10 show the runtime of our secure MPC protocols (cf. §4) in LAN1 and WAN100 network environments.

Table 7: Communication costs (MB) and the number of messages per generated noise value averaged over 10 runs of our MPC-based DP mechanisms using B and Y-sharing among N parties. ✓ are protocols not susceptible to the finite precision attacks discussed in §3, while ✗ are vulnerable solutions. The best secure results are marked in bold.

				N = 2			N = 3		
	Mechanisms	Security	Batch	Prot.	Communication	Message	Prot.	Communication	Message
Discrete	M_{DLap} [45]	✗	1	Y	7.57	99112	B	16.76	263620
	M_{DLap} (cf. §4.1, optimized)	✓	1	Y	**492.72**	**27723**	B	**728.37**	**232736**
	M_{DGauss} (cf. §4.1)	✓	1	Y	**1 085.82**	**27687**	B	**858.17**	**104028**
Continuous	M_{Lap} [45]	✗	1	Y	8.27	124163	B	19.90	346736
	M_{Snap} (cf. §4.2, optimized)	✓	1	Y	**5.33**	78543	B	**13.76**	**239700**
	M_{ISLap} (cf. §4.3, optimized)	✓	1	Y	38.73	**33739**	B	66.97	265020
	M_{Gauss} [71]	✗	1	Y	10.33	152591	B	26.44	460824
	M_{ISGauss} (cf. §E)	✓	1	Y	**1 423.88**	**176477**	B	**2 009.53**	**672020**

Table 8: Complexity assessment using circuit size (# AND gates) in Y-sharing and the maximum depth of (longest path of AND and MUX gates) in B-sharing. ✓ are protocols not susceptible to the finite-precision attacks discussed in §3, while ✗ are vulnerable solutions. The best secure results are marked in bold.

				N = 2		N ≥ 3	
	Mechanisms	Security	Batch	Prot.	No. AND	Prot.	Depth AND + MUX
Discrete	M_{DLap} [45]	✗	1	Y	97 651	B	1 920+72
	M_{DLap} (cf. §4.1, naive)	✓	1	Y	40 086 526	B	11 187+55
	M_{DLap} (cf. §4.1, optimized)	✓	1	Y	**18 645 246**	B	**1 321+86**
	M_{DGauss} (cf. §4.1)	✓	1	Y	**13 628 906**	B	**642+73**
Continuous	M_{Lap} [45]	✗	1	Y	73 940	B	2 840+135
	M_{Snap} (cf. §4.2, naive)	✓	1	Y	97 871	B	3 126+141
	M_{Snap} (cf. §4.2, optimized)	✓	1	Y	**53 276**	B	1 993+84
	M_{ISLap} (cf. §4.3, naive)	✓	1	Y	3 057 136	B	12 455+114
	M_{ISLap} (cf. §4.3, optimized)	✓	1	Y	1 404 586	B	**1 469+89**
	M_{Gauss} [71]	✗	1	Y	102 366	B	2 696+137
	M_{ISGauss} (cf. §E)	✓	1	Y	**55 974 725**	B	**10 633+81**

Table 9: Total runtimes (ms) per generated noise value in LAN1 (1-Gbit/s with 1ms RTT) averaged over 10 protocols runs of our MPC-based DP mechanisms using B and Y-sharing among N parties. ✓ are protocols not susceptible to the finite-precision attacks discussed in §3, while ✗ are vulnerable solutions. The best secure results are marked in bold. - denotes memory overflow.

				N = 2			N = 3		
	Mechanisms	Security	Batch	Prot.	Offline	Online	Prot.	Offline	Online
Discrete	M_{DLap} [45]	✗	10	Y	196.03	5.44	B	823.13	23.73
	M_{DLap} (cf. §4.1, naive)	✓	10	Y	12 221.81	**4.55**	B	—	—
	M_{DLap} (cf. §4.1, optimized)	✓	10	Y	**4 707.46**	4.81	B	**13 474.59**	**22.33**
	M_{DGauss} (cf. §4.1)	✓	5	Y	**11 081.80**	**17.72**	B	**17 333.22**	**42.11**
Continuous	M_{Lap} [45]	✗	30	Y	68.19	6.94	B	380.91	22.65
	M_{Snap} (cf. §4.2, naive)	✓	30	Y	75.46	45.90	B	302.88	140.22
	M_{Snap} (cf. §4.2, optimized)	✓	30	Y	**56.13**	5.63	B	**291.73**	37.79
	M_{ISLap} (cf. §4.3, naive)	✓	30	Y	1 101.77	50.05	B	11 877.52	157.30
	M_{ISLap} (cf. §4.3, optimized)	✓	30	Y	373.54	**5.23**	B	1 114.22	**34.50**
	M_{Gauss} [71]	✗	30	Y	118.08	4.75	B	381.59	16.58
	M_{ISGauss} (cf. §E)	✓	30	Y	**14 219.45**	**6.10**	B	—	—

Table 10: Total runtimes (ms) per generated noise value in WAN100 (100-Mbit/s with 100ms RTT) averaged over 10 protocols runs of our MPC-based DP mechanisms using B and Y-sharing among N parties. ✓ are protocols not susceptible to the finite-precision attacks discussed in §3, while ✗ are vulnerable solutions. The best secure results are marked in bold. - denotes memory overflow.

				N = 2			N = 3		
	Mechanisms	Security	Batch	Prot.	Offline	Online	Prot.	Offline	Online
Discrete	M_{DLap} [45]	✗	10	Y	622.97	45.89	B	23 436.08	150.22
	M_{DLap} (cf. §4.1, naive)	✓	10	Y	88 752.83	**46.96**	B	—	—
	M_{DLap} (cf. §4.1, optimized)	✓	10	Y	**42 352.58**	47.99	B	**82 289.22**	**153.15**
	M_{DGauss} (cf. §4.1)	✓	5	Y	**66 792.65**	**86.63**	B	**89 240.04**	**355.12**
Continuous	M_{Lap} [45]	✗	30	Y	301.17	38.13	B	11 520.13	263.17
	M_{Snap} (cf. §4.2, naive)	✓	30	Y	266.34	127.13	B	7 977.09	4 640.57
	M_{Snap} (cf. §4.2, optimized)	✓	30	Y	**241.53**	42.89	B	**7 656.29**	572.26
	M_{ISLap} (cf. §4.3, naive)	✓	30	Y	6 862.04	123.97	B	71 937.68	4 477.84
	M_{ISLap} (cf. §4.3, optimized)	✓	30	Y	3 284.79	**41.00**	B	10 415.45	**427.07**
	M_{Gauss} [71]	✗	30	Y	370.97	41.90	B	10 360.31	262.90
	M_{ISGauss} (cf. §E)	✓	30	Y	**121 444.31**	**56.75**	B	—	—

Table 11: Total/online runtimes in ms for 64-bit integer/floating-point arithmetic in $\{\mathrm{Y}, \mathrm{B}, \mathrm{A}\}$. Results averaged over 10 runs in LAN10 (10 Gbit/s, 1 ms RTT). Runtime of a single integer operation is amortized over 1 000 SIMD values, resp. 100 SIMD values for one floating-point operation. Best total runtimes are bold.

	Total		Online	
Operations	N=2	N=3	N=2	N=3
INT_ADD$^{\mathrm{Y}}$	**0.13**	**0.38**	0.08	0.08
INT_ADD$^{\mathrm{B}}$	0.41	0.44	0.11	0.10
INT_MUL$^{\mathrm{Y}}$	**0.46**	11.63	0.37	0.26
INT_MUL$^{\mathrm{B}}$	1.46	**2.01**	0.13	0.24
INT_DIV$^{\mathrm{Y}}$	**2.72**	**14.48**	1.21	1.53
INT_DIV$^{\mathrm{B}}$	54.40	70.15	52.89	68.41
UINT_MOD$^{\mathrm{Y}}$	**1.49**	**13.63**	0.05	1.56
UINT_MOD$^{\mathrm{B}}$	51.86	64.77	50.50	62.95
INT_LT$^{\mathrm{Y}}$	**0.11**	0.40	0.06	0.08
INT_LT$^{\mathrm{B}}$	0.33	**0.35**	0.15	0.14
INT_EQ$^{\mathrm{Y}}$	**0.09**	0.38	0.06	0.07
INT_EQ$^{\mathrm{B}}$	0.30	**0.31**	0.18	0.18
INT2FL$^{\mathrm{Y}}$	**0.15**	2.75	0.07	0.19
INT2FL$^{\mathrm{B}}$	1.26	**1.73**	0.27	0.40
FL_ADD$^{\mathrm{Y}}$	**2.05**	15.37	0.41	0.72
FL_ADD$^{\mathrm{B}}$	5.89	**6.85**	3.31	3.12
FL_ADD$^{\mathrm{A}}$	472.41	571.64	1.77	3.23
FL_MUL$^{\mathrm{Y}}$	**1.75**	34.86	0.54	1.49
FL_MUL$^{\mathrm{B}}$	6.20	**8.30**	2.36	2.63
FL_MUL$^{\mathrm{A}}$	148.44	181.81	0.56	1.01
FL_DIV$^{\mathrm{Y}}$	**4.97**	67.67	1.46	2.85
FL_DIV$^{\mathrm{B}}$	24.96	**33.49**	17.07	22.91
FL_DIV$^{\mathrm{A}}$	946.76	1 137.44	0.89	2.16
FL_LT$^{\mathrm{Y}}$	**0.53**	2.59	0.08	0.21
FL_LT$^{\mathrm{B}}$	1.91	**1.96**	0.75	0.39
FL_LT$^{\mathrm{A}}$	149.23	182.59	0.45	0.50
FL_FLOOR$^{\mathrm{Y}}$	**0.44**	2.91	0.14	0.34
FL_FLOOR$^{\mathrm{B}}$	1.60	**2.35**	0.40	0.55
FL_FLOOR$^{\mathrm{A}}$	457.41	560.97	4.76	6.77
FL_EXP$^{\mathrm{Y}}$	**5.70**	94.82	2.08	3.40
FL_EXP$^{\mathrm{B}}$	41.49	**59.26**	31.76	44.38
FL2INT$^{\mathrm{Y}}$	**0.76**	5.78	0.17	0.41
FL2INT$^{\mathrm{B}}$	3.29	**4.45**	0.95	1.48
FL2INT$^{\mathrm{A}}$	768.53	939.83	2.44	4.58
MulPow2$^{\mathrm{Y}}$	**0.43**	**1.70**	0.10	0.08
MulPow2$^{\mathrm{B}}$	2.01	2.34	1.10	1.11
RoundToLambda$^{\mathrm{Y}}$	**0.61**	3.85	0.16	0.39
RoundToLambda$^{\mathrm{B}}$	1.78	**2.40**	0.57	0.92