Input Secrecy & Output Privacy: Efficient Secure Computation of Differential Privacy Mechanisms

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Jonas Böhler
aus Bad Säckingen

Tag der mündlichen Prüfung: 10.11.2021

1. Referent: Prof. Dr. Jörn Müller-Quade
2. Referent: Prof. Dr. Florian Kerschbaum

Abstract

Data is the driving force of modern businesses. For example, customer-generated data is collected by companies to improve their products, discover emerging trends, and provide insights to marketers. However, data might contain personal information which allows to identify a person and violate their privacy. Examples of privacy violations are abundant – such as revealing typical whereabout and habits, financial status, or health information, either directly or indirectly by linking the data to other available data sources. To protect personal data and regulate its collection and processing, the general data protection regulation (GDPR) was adopted by all members of the European Union.

Anonymization addresses such regulations and alleviates privacy concerns by altering personal data to hinder identification. *Differential privacy* (DP), a rigorous privacy notion for anonymization mechanisms, is widely deployed in the industry, e.g., by Google, Apple, and Microsoft. Additionally, cryptographic tools, namely, *secure multi-party computation* (MPC), protect the data during processing. MPC allows distributed parties to jointly compute a function over their data such that only the function output is revealed but none of the input data. MPC and DP provide orthogonal protection guarantees. MPC provides *input secrecy*, i.e., MPC protects the inputs of a computation via encrypted processing. DP provides *output privacy*, i.e., DP anonymizes the output of a computation via randomization. In typical deployments of DP the data is randomized locally, i.e., by each client, and aggregated centrally by a server. MPC allows to apply the randomization centrally as well, i.e., only once, which is optimal for accuracy. Overall, MPC and DP augment each other nicely. However, universal MPC is inefficient – requiring large computation and communication overhead – which makes MPC of DP mechanisms challenging for general real-world deployments.

In this thesis, we present efficient MPC protocols for distributed parties to collaboratively compute DP statistics with high accuracy. We support general *rank-based statistics*, e.g., min, max, median, as well as *decomposable aggregate functions*, where local evaluations can be efficiently combined to global ones, e.g., for convex optimizations. Furthermore, we detect *heavy hitters*, i.e., most frequently appearing values, over known as well as unknown data domains. We prove the semi-honest security and differential privacy of our protocols. Also, we theoretically analyse and empirically evaluate their accuracy as well as efficiency. Our protocols provide higher accuracy than comparable solutions based on DP alone. Our protocols are efficient, with running times of seconds to minutes evaluated in real-world WANs between Frankfurt and Ohio (100 ms delay, 100 Mbits/s bandwidth), and have modest hardware requirements compared to related work (mainly, 4 CPU cores at 3.3 GHz and 2 GB RAM per party). Additionally, our protocols can be outsourced, i.e., clients can send encrypted inputs to few servers which run the MPC protocol on their behalf.

Zusammenfassung

Daten sind die Antriebskraft moderner Unternehmen. Von Kunden generierte Daten werden von Firmen gesammelt, um Produkte zu verbessern, aufkommende Trends zu entdecken, und sie an Vermarkter zu verkaufen. Allerdings beinhalten diese Daten potentiell personenbezogene Informationen, die es erlauben eine Person zu identifizieren und den Schutz ihrer Privatsphäre zu verletzen. Datenschutzverletzungen lassen sich im Überfluss finden – versehentliche Bekanntgabe von typischen Aufenthaltsorten und Gewohnheiten, Finanzstatus, oder gesundheitlichen Informationen – entweder direkt oder indirekt, indem Daten mit anderen verfügbaren Datenquellen verbunden werden. Um personenbezogene Daten zu schützen, ihre Sammlung und Verarbeitung zu regulieren, wurde die allgemeine Datenschutzgrundverordnung (DSGVO) von allen Mitgliedern der Europäischen Union verabschiedet. Anonymisierung – die Veränderung von personenbezogenen Daten, um Identifikation zu verhindern – dient dem Zweck, den gesetzlich geforderten Datenschutz zu gewährleisten. *Differential Privacy* (DP) formalisiert den Schutz von Anonymisierungsmechanismen und findet weit verbreiteten Einsatz in der Industrie – unter anderem bei Google, Apple und Microsoft. Zusätzlichen Schutz während der Datenverarbeitung bieten kryptografische Verfahren wie *sichere Mehrparteienberechnung* (im Englischen secure multi-party computation, MPC). MPC ermöglicht es verteilten Parteien gemeinsam eine Funktion über ihren Daten zu berechnen, sodass nur die Funktionsausgabe bekannt wird, aber die Eingaben geheim bleiben. MPC und DP bieten orthogonale Schutzgarantien. MPC bietet *Geheimhaltung der Eingaben* (input secrecy), d.h., MPC schützt die Eingaben, indem deren Verarbeitung nur verschlüsselt erfolgt. DP liefert *Datenschutz für Ausgaben* (output privacy), d.h., DP anonymisiert die Ausgabe einer Berechnung mithilfe von Randomisierung. Die Garantien von MPC und DP ergänzen sich sehr gut. Allerdings ist MPC generell ineffizient und erfordert großen Zusatzaufwand – in Form von Berechnungszeit und Kommunikation – welcher eine Hürde für den praktischen Einsatz von MPC für DP-Mechanismen darstellt.

Diese Dissertation präsentiert effiziente MPC-Protokolle, um DP-Statistiken über vereinten Daten von verteilten Parteien mit hoher Genauigkeit zu berechnen. Wir unterstützen *Rank-basierte Statistiken* wie den Median und *zerlegbare Aggregatsfunktionen* (decomposable aggregate functions), wobei lokale Auswertungen effizient zu globalen Ergebnissen aggregierbar sind, beispielsweise für konvexe Optimierung. Des Weiteren finden wir *häufig vorkommende Werte* (heavy hitters) aus bekannten und unbekannten Wertebereichen. Wir beweisen, dass unsere Protokolle Geheimhaltung während der Berechnung (semi-honest security) und Datenschutz der Ausgabe (differential privacy) bieten. Wir evaluieren die Genauigkeit und Effizienz unserer Protokolle sowohl theoretisch als auch empirisch. Unsere Protokolle bieten eine höhere Genauigkeit als vergleichbare Lösungen, die nur auf DP basieren. Unsere Protokolle sind effizient mit Laufzeiten von Sekunden zu Minuten in einem WAN zwischen Frankfurt und Ohio (100 ms Verzögerung, 100 Mbits/s Bandbreite) und wir haben verhältnismäßig geringe Ansprüche an die Hardware (hauptsächlich 4 CPU-Kerne mit 3.3 GHz, 2 GB RAM pro Partei). Zusätzlich kann die Berechnung unserer Protokolle ausgelagert werden, d.h., Clients können ihre verschlüsselten Eingaben auf wenige Servern verteilen, welche die Protokolle an ihrer Stelle ausführen.

Acknowledgement

First and foremost, I want to thank Jörn Müller-Quade and Florian Kerschbaum for their invaluable advice, academic guidance, and insightful discussions. Jörn's fascinating and joyous cryptography lectures reinforced my interest in security and privacy while studying at KIT, which put me on a path towards research. Florian helped me navigate and illuminate this unmapped and often dark research path with his vast knowledge and political acumen.

I also want to thank my SAP colleagues and friends, who for a while not only shared a room but plenty of inspiring thoughts with me. Namely, applied cryptography researchers Florian[1], Anselme, Andreas, Benny; anonymization researchers Daniel and Benjamin; and my SAP managers Detlef Plümper, Roger Gutbrod, and Mathias Kohler.

On a personal note, I want to thank my family for their endless love, trust, and encouragement. Lastly and most importantly, I'm eternally grateful for my girlfriend Dorothée for her patient support and love during this arduous adventure. I'm looking forward to our next adventure, raising our amazing son Max, who waited long enough to come into this world for me to (almost) finish this thesis.

[1] Different Florian.

Contents

Acronyms

DP	Differential privacy
EM	Exponential mechanism
GM	Gumbel mechanism
GRR	Generalized randomized response
LM	Laplace mechanism
LAN	Local-area network
MPC	Multi-party computation
PPT	Probabilistic polynomial-time
RTT	Round-trip time
WAN	Wide-area network

1 Introduction

In Section 1.1, we motivate our research question, which we present and discuss in Section 1.2. In Section 1.3, we list our contributions, i.e., efficient, secure protocols for distributed, privacy-preserving statistics. We present an overview of our protocols in Section 1.4. Finally, Section 1.5 describes how the remainder of this thesis is structured.

1.1 Motivation

Data is valuable. In most of the 20th century, a limiting factor in data collection was storage space and information retrieval. Information printed on paper documents used to fill entire archive buildings and retrieval required manual labor; nowadays, an exponentially larger amount of data is stored in data centers where indexed databases allow nearly instantaneous retrieval[1]. The digital revolution enabled data collection and processing on a scale that was previously unthinkable. Presently, the business model of some of the most valuable companies in the world is entirely based on collecting and monetizing data of individuals, mainly by selling it to advertisers for targeted advertising. Furthermore, government spending, such as funding of public services and implementation of policy decisions, is informed by demographic studies and census data [And21, Section 11.2].

Data is personal. Already a few collected data points suffice to identify someone from credit card metadata [DMRS+15][2], infer an individual's typical whereabouts and routines from smartphone data [DMHVB13][3], and learn behavioral patterns from smart meter measurements [AF10, MMSF+10][4]. To regulate the rapidly increasing data collection and consistently govern the processing of personal data, the European Union adopted the General Data Protection Regulation (GDPR)[5] for all its members states which became enforceable in May 2018. The GDPR states: "The protection of natural persons in relation to the processing of personal data is a fundamental right."[6]

[1] For example, documents collected by the Ministry for State Security (colloquially called "Stasi") of the German Democratic Republic (1949-1990) fill 111 kilometers of shelf space. Since 1998 the records are indexed in a database. `https://www.bstu.de/en/archives/about-the-archives/`

[2] Montjoye et al. [DMRS+15, Figure 2] identified 90% of individuals, given the date and location of four of their purchases, in a data set with 1.1 million credit card transactions.

[3] Strava, a fitness tracking app, revealed "locations and habits of military bases and personnel, including those of American forces in Iraq and Syria" `https://www.nytimes.com/2018/01/29/world/middleeast/strava-heat-map.html`.

[4] The European Data Protection Supervisor [Eur12, Point 19] warned that smart meters not only reveal, e.g., vacation times and sleep patterns, but potentially also health conditions (e.g., kidney problems) by identifying electrical medical devices (e.g., dialysis machines).

[5] EU regulation 2016/679, `https://eur-lex.europa.eu/eli/reg/2016/679/oj`

[6] GDPR defines *personal data* in Article 4, point (1) as "any information relating to an identified or identifiable natural person (..) who can be identified, directly or indirectly, in particular by reference to an identifier such as a name, an identification number, location data, an online identifier or to one or more factors specific to the physical, physiological, genetic, mental, economic, cultural or social identity of that natural person;" and *processing* in point (4) as "any operation or set of operations which is performed on personal data or on sets of personal data, whether or not by automated means, such as collection, recording, organisation, structuring, storage, adaptation or alteration, retrieval, consultation, use, disclosure by transmission, dissemination or otherwise making available, alignment or combination, restriction, erasure or destruction;"

Cryptography protects data processing. *Secure multi-party computation* (MPC) [Gol09] is a cryptographic protocol to exchange and operate on encrypted data such that only the output of the computation is revealed. As such, MPC protects the data of multiple parties during processing. MPC has been researched for over 30 years, and recently real-world deployments have emerged in industry and government [Lin20, Section 5]: Google and Mastercard reportedly deployed secure computation to track ad conversion by securely linking Google's online ad impressions with offline purchases based on credit card transactions handled by Mastercard [Blo18, IKN+17, IKN+20]. Estonian government institutes applied it to detect tax fraud [BJSV15] and combined income tax records with university records, to analyse if working during university studies negatively impacts earning a degree [BKK+16]. The Boston Women's Workforce Council deployed secure computation to investigate gender-based wage gaps in the greater Boston area with the participation of 114 employers with 166 705 employees [LJA+18, Lin20].

MPC protects the data during processing, however, releasing exact outputs does not limit inference of inputs and allows reconstruction attacks [DN03]. Recently, the US census bureau showed that even high-level aggregate statistics, i.e., demographic counts for residential areas, suffice to identify millions of individuals of the 2010 US census when combined with commercially available data [DKM19, GAM19].

Anonymization controls inference. *Differential privacy* (DP) [Dwo06, DMNS06] is a rigorous anonymization definition. DP requires fine-tuned, computation-dependent randomization which introduces uncertainty hindering reconstruction attacks. In other words, DP limits inference of any input given the output. DP is widely deployed to mitigate privacy risks and regulatory concerns. The US census bureau adopted DP for the 2020 census [Abo18]. Apple deploys DP to privately learn frequently typed words on mobile devices to improve auto-complete suggestions, and to detect websites with large resource consumptions to optimize the browsing experience in iOS and macOS [App16, App17]. Google privately detects popular Chrome browser settings [EPK14, FPE16a] as well as busy times for businesses in Google Maps [Goo19]. Also, Microsoft deploys differentially private telemetry data collection in Windows 10 (Creators Fall Update) across millions of devices [DKY17] and LinkedIn's Audience Engagement API lets marketers perform DP queries to learn, e.g., most frequently shared articles among users with a specific skill set [RSP+20, Rog20]. Real-world deployments [App16, DKY17, EPK14, FPE16a, Goo19] mainly implement the *local model* of DP, i.e., users locally randomize their data and send it to an untrusted aggregator. In the *central model*, e.g., used by LinkedIn [RSP+20, Rog20] and the US Census bureau [Abo18], a trusted party has access to the raw data, which only needs to apply randomization once, on the aggregated result. The local model has fewer assumptions (no trusted party). However, it generally requires exponentially more data samples to achieve the optimal accuracy offered by the central model at the same privacy level [KLN+11]. Small data is the most challenging regime for DP [BEM+17, NRVW20] as the randomization, mostly in the form of additive noise, easily exceeds the signal in the data.

Composing cryptography & anonymization is inefficient. MPC and DP provide orthogonal protections and augment each other. MPC protects input data during processing, and DP limits inference of any inputs given the output. In theory, combining existing solutions for MPC and DP bridges the gap between the local and central model of DP and simultaneously provides strong privacy (no trusted third party) with high accuracy (centralized randomization). In practice, however, general MPC is inefficient as it suffers from prohibitive computation and communication overhead hindering real-world deployments.

1.2 Research Question

The research question investigated in this dissertation is:

Can distributed parties efficiently and accurately compute statistics over their small data sets without revealing secret inputs and ensuring strong privacy guarantees for the output?

In the following, we briefly overview our answer to the research question, and detail aspects of the research question in the remainder of this section. In this thesis, we present secure and efficient protocols for privacy-preserving statistics over distributed parties with high accuracy. Our protocols provide *input secrecy* via MPC, i.e., no input is revealed to others, as well as *output privacy* via DP, i.e., the output limits inference about any input. MPC already ensures high *accuracy* for DP and the main challenge is designing efficient MPC protocols. Our protocols are *efficient* with running times in seconds to minutes over the Internet, and *practical* with modest hardware requirements compared to related work[7]. Furthermore, our protocols support *outsourcing* where input parties (clients) send encrypted inputs to computation parties (servers) who run the protocol on their behalf. While MPC solutions for DP aggregate statistics exist (e.g., sum and mean), DP statistics based on an element's rank or frequency (e.g., median and mode) lack efficient and accurate MPC protocols. We support general *rank-based statistics*, e.g., min, max, median, percentiles, but focus on the median for illustration purposes. We also support *decomposable aggregate functions*, as used in MapReduce-style frameworks, applicable to, e.g., convex optimizations. Additionally, we discover *heavy hitters*, i.e., most frequently appearing values, over known as well as unknown data domains. Next, we further detail aspects of the research question.

Input Secrecy

We consider a distributed setting with two or more parties. To ensure that the parties can jointly compute a function without revealing their secret inputs to the other parties, we employ MPC [Gol09]. Informally, MPC is a cryptographic protocol to exchange and operate on encrypted data (formalized in Section 2.1). While secure computation protects the inputs of a computation, the *exact* output is released, potentially leading to privacy violations [DN03].

Output Privacy

To limit inference from the output, we employ DP [Dwo06, DMNS06], a rigorous privacy notion restricting the privacy loss of any party who participates in a computation. Informally, DP introduces uncertainty by applying fine-tuned, computation-dependent randomization (formalized in Section 2.2).

Accuracy & Small Data

We use MPC to simulate a trusted third party, resulting in highly accurate DP statistics. Our protocols also support large data sizes, however, our focus is on small data. Small data is the most challenging regime for DP [BEM+17, NRVW20], as the noise from the randomization can easily drown the (statistical) signal in the data. Even Google's large-scale data collection [BEM+17, EPK14],

[7] We mainly used Amazon Web Services t2.medium instances with 4 CPU cores and 2 GB RAM for our evaluation. However, for one of our largest evaluations, we used 8 GB RAM, and to evaluate specialized version of one of our protocols, optimized for multi-threading, we required 8 cores and 15 GB RAM. For details, we refer to Section 4.4.1.

with billions of daily user reports in the local model, is insufficient if the statistical value of interest appears infrequently [BEM+17, Section 2.2], e.g., the median. Specifically, an exponential separation exists between the local and central model regarding accuracy and sample complexity [KLN+11].

Efficiency

General-purpose MPC solutions are inefficient for DP, i.e., they suffer from large computation and communication overhead as well as liveness constraints in a wide-area network (WAN). In this thesis, we design efficient, special-purpose MPC protocols for DP – including novel alternatives to secure exponentiations. Our MPC protocols run in seconds to minutes over the Internet on modest hardware and our client communication is in the order of kilobytes.

Statistics

In our protocols, we focus on rank-based statistics, also called order statistics, decomposable aggregate functions, and heavy hitters over distributed data. For distributed aggregate statistics, e.g., mean and sum, various DP solutions exist [DKM+06, GX17, RN10, TKZ16] that basically consist of summing values from each party. Rank-based statistics, however, require knowledge of an element's position in the *sorted* data, posing a challenge for distributed data. Similarly, heavy hitters, i.e., frequent values, can be easily identified for small data domains, e.g., by building a histogram. However, efficient discovery of heavy hitters on *large or even unknown domains* requires additional considerations and clever approximations [CH10, WLJ19].

Formally, *rank-based statistics* are defined over sorted data set $D = \{d_1, \dots, d_n\}$, where $d_1 \leq d_2 \leq \cdots \leq d_n$. Rank-based statistics include

- the minimum d_1 and maximum d_n,
- the range $d_n - d_1$,
- p^{th}-percentile $d_{\lceil n \cdot p/100 \rceil}$, i.e., the value larger than p percent of D,
- interquartile range $d_{\lceil 0.75n \rceil} - d_{\lceil 0.25n \rceil}$,
- and the median $d_{\lceil n/2 \rceil}$, i.e., the 50th-percentile which splits the sorted data roughly in half.

The median is a robust statistic, i.e., few input changes do not lead to large output changes [DL09, Section 1.2]. The median is used to represent a "typical" value from a data set, e.g., median income is more representative than mean income[8], and insurance companies use the median life expectancy to adjust insurance premiums. The median is also useful in the collection of private usage statistics. Reporting the median in addition to the mean allows the collector to detect skew in the distribution, i.e., if outliers exist.

Decomposable aggregate functions are employed in MapReduce-style frameworks to efficiently compute statistics over distributed data. We consider decomposability for utility functions, which score how close any possible output is to a desired evaluation output (formalized in Section 2.2.4). Decomposable utility scores can be in the form of ranks, frequencies, or loss function

[8] For example, consider Medina, Washington, a Seattle suburb near the headquarters of Amazon and Microsoft. With a population of around 3 000 the median income in 2018 was around $192 000 while the average income was about 308 000 [US 18]. The average income could even be in the millions due to outliers that skew the result of the mean, e.g., the billionaires Jeff Bezos and Bill Gates. However, such outliers may be removed from local statistics and only appear in national aggregates [And21, Section 11.2.1.5].

scores, and applications include federated learning with compressed gradients [BWAA18], empirical risk minimization [BST14], and digital goods auctions [MT07] (Section 6.1.1).

Heavy hitters, also known as top-k, are the k most frequently appearing elements in a data set. A special heavy hitter is the *mode*, the most frequent element. As stated before, heavy hitters are often collected to learn common patterns and trends, e.g., frequently typed new words [App16, App17], common user settings [EPK14, FPE16a], and often shared articles [RSP+20].

1.3 Contributions

Previous work on DP median (resp., DP heavy hitters), either require a large number of parties to be accurate [STU17, WGSX20] (resp., [App16, DKY17, EPK14, FPE16a]) or rely on a trusted third party [DL09, McS09, NRS07] (resp., [RSP+20, Rog20]). General MPC solutions for DP statistics cannot scale to large data set or domain sizes [EKM+14, PL15]. Related work is discussed in more detail in Section 3.

Our protocols provide novel alternatives for DP statistics that are efficiently computable even for large data or domain sizes without a trusted party. Our contributions are as follows:

- Our protocols $\mathsf{EM}_{\mathsf{med}}$ and EM^* securely compute the DP median with a running time sublinear in the domain size, and support general order statistics (e.g., min, max, percentiles).

  Protocol EM^* is extensible to decomposable aggregate functions, allowing efficient aggregation over distributed data sets as found in MapReduce-style frameworks.

- Our protocols HH and PEM securely discover DP heavy hitters. HH has a running time linear in the data size and supports unknown domains, PEM is sublinear in the size of the known domain.

- We implement our protocols with secure computation frameworks, prove the security of our protocols against semi-honest (passive) adversaries, and discuss extension for malicious (active) adversaries (Sections 5.2.6, 6.2.8, 7.2.4).

  Protocol $\mathsf{EM}_{\mathsf{med}}$ is implemented in ABY [DSZ15a], EM^* is implemented in SCALE-MAMBA [AKR+20], HH and PEM are implemented in SCALE-MAMBA as well as MP-SPDZ [Kel20].

- Our protocols provide high accuracy even for small data sizes, which is the most challenging regime for DP.

  We analyze the accuracy of our protocols (Sections 5.1.6, 6.1.5, 7.1) and empirically compare them to related work (Sections 5.3.7, 6.3.5, 7.3.5).

- Our protocols achieve efficient running times of seconds to minutes in a WAN with 100 ms delay and 100 Mbits/s on modest hardware with 4 CPU cores at 3.3 GHz and mainly 2 GB RAM (see Section 4.4.1 for details).

  We analyze the running-time complexity of our protocols (Sections 5.2.5, 6.2.7, 7.2.3), and measure running time and communication in real-world networks (Sections 5.3, 6.3, 7.3).

1.4 Our Protocols: $\mathsf{EM}_{\mathsf{med}}$, EM^*, HH, PEM

In this section, we give an overview of our protocols $\mathsf{EM}_{\mathsf{med}}$, EM^*, HH, and PEM. First, we need to informally describe DP mechanisms, which we later formalize in Section 2.2.4. Then, we describe implementation challenges regarding MPC of DP. Finally, we describe our protocols and how they address these challenges.

DP Mechanisms

So far, we said that differential privacy ensures randomized outputs. In more detail, randomized algorithms, called *mechanisms* in DP literature, provide the randomization either via additive noise or probabilistic selection. Additive noise is used in the *Laplace mechanism* LM: Given a data set D and a function f to evaluate, e.g., the median, LM outputs $f(D)+l$, where l is noise from the Laplace distribution parameterized with privacy parameter ϵ. Probabilistic selection is used by the *exponential mechanism* EM: Output r of EM is selected with probability proportional to $\exp(u(D,r)\epsilon)$, where utility function u scores how "close" r is to the desired output $f(D)$. Higher scores translate to higher selection probabilities. Note that scores and probabilities for the entire output domain of f must be computed. The *Gumbel mechanism* GM provides the same output distribution as EM by selecting the element with the largest noisy utility score, where noise is sampled from the Gumbel distribution.

We later show that EM provides better accuracy than LM for the median (Section 3.6.1, Section 6.1.5), hence, we use EM in our protocols. In fact, most of our protocols – namely, $\mathsf{EM}_{\mathsf{med}}$, EM^*, and PEM – are build upon the exponential mechanism EM and only HH implements the Laplace mechanism LM.

Implementation Challenges & Design Considerations

Let $\mathfrak{D}$ denote the data domain and $D = \{d_1, \dots, d_n\} \in \mathfrak{D}^n$ the combined data of all parties. We identify and address the following key challenges for efficient computation of the exponential mechanism, especially with MPC:

(i) *Large domains*: the running time complexity of EM is linear in the domain size $|\mathfrak{D}|$ as probabilities for all possible outputs in $\mathfrak{D}$ are computed.

(ii) *Costly exponentiation*: a straightforward implementation of EM requires $|\mathfrak{D}|$ exponentiations, which is prohibitive for MPC [ABZS13, AS19, DFK+06, Kam15].

Additionally, we investigate trade-offs between running time and privacy as well as accuracy:

(iii) *Balancing trade-offs*: standard EM neither considers a relaxation of DP nor does it permit a parameterized trade-off between running time and accuracy.

Our protocols tackle these challenges by partitioning the domain, eliminating secure exponentiation, and enabling parameterized trade-offs.

Our Protocols

$\mathsf{EM}_{\mathsf{med}}$ (Section 5) is a two-party protocol to securely compute the DP median with EM and can be extended to multiple parties (Section 5.2.7). Our protocol $\mathsf{EM}_{\mathsf{med}}$

(i) handles large domains by operating over *sorted (subset of the) data* with running time sublinear in the domain size,

(ii) avoids costly exponentiations by leveraging a *data-independent utility function* whose exponentiations can be computed locally; the key insight being that an element's utility score corresponds to its position in the *sorted* data,

(iii) allows a trade-off between running time and privacy relaxation by pruning the data if its size is not sublinear in the domain size: Pruning from Aggarwal et al. [AMP10] lets $\mathsf{EM}_{\mathsf{med}}$ efficiently support large data sets as only a small subset of the data must be sorted securely. However, pruning relaxes DP and allows only limited group privacy (Section 5.3.3).

EM* (Section 6) is a multi-party protocol to securely evaluate EM for decomposable utility functions illustrated for the DP median. We also implement GM^*, a variation of EM^* based on the Gumbel mechanism GM. Our protocol EM^*

(i) handles large domains by iteratively partitioning and selecting *domain subranges* of decreasing size with running time sublinear in the domain size,

(ii) avoids costly exponentiations via *decomposable utility functions* whose local partial evaluations can be efficiently combined similar to MapReduce-style frameworks; i.e., given decomposable utility function $u(D,\cdot) = \sum_{i=1}^{n} u(d_i,\cdot)$ local exponentiations $x_i(\cdot) = \exp(u(d_i,\cdot)\epsilon)$ can be combined as $\exp(u(D,\cdot)\epsilon) = \prod_{i=1}^{n} x_i(\cdot)$ (Section 6.1.1),

(iii) allows a parameterized trade-off between running time and accuracy: *many iterations with few subranges* is faster, however, *few iterations with many subranges* improve accuracy (Section 6.3.4).

HH (Section 7) is a multi-party protocol to securely discover DP heavy hitters for small data sizes with unknown domain via LM. HH is based on the sketch by Misra and Gries [MG82], i.e., a space-efficient data structure, and our protocol

(i) handles large domains by operating over the *small data set*, keeping a map of frequent data elements and their approximate counts, with a running time linear in the small data size,

(ii) avoids costly exponentiations by implementing the Laplace mechanism, which does not require exponentiation,

(iii) allows a parameterized trade-off between running time and accuracy: *smaller map size* is faster, however, *larger map size* improves accuracy (Sections 7.1.3, 7.3.5).

PEM (Section 7) is a multi-party protocol to securely discover DP heavy hitters for arbitrary data sizes with known domain via GM. PEM is based on a local-model protocol by Wang et al. [WLJ19], and our protocol

(i) handles large domains by iteratively selecting *domain bit-prefixes* of increasing size from disjoint subsets of input parties with a running time sublinear in the domain size,

(ii) avoids costly exponentiations by implementing the Gumbel mechanism, which does not require exponentiation,

(iii) allows a parameterized trade-off between running time and accuracy: gathering *small prefixes from many small groups* is faster, however, *large prefixes from few large groups* improve accuracy (Sections 7.1.2, 7.3.5). While HH is more accurate for small data sets, PEM is faster for larger data sets.

1.5 Structure

The remainder of this thesis is structured as follows.

Chapter 2 presents preliminaries and definitions for cryptographic primitives and anonymization techniques employed in this thesis. First, we define secure multi-party computation, common security models, namely, semi-honest (passive) and malicious (active) adversaries, and implementation paradigms, namely, garbled circuits and secret sharing. Then, we briefly summarize previous anonymization methods leading to differential privacy. Finally, we formalize differential privacy, list its properties, and detail mechanisms to satisfy this privacy notion.

Chapter 3 discusses related work. First, we compare different privacy models, mainly, the local, central and MPC model of DP. Then, we describe related work combining MPC and DP with a focus on the exponential mechanism. We overview techniques to simplify sampling from the exponential mechanism, describe the influence of limited machine precision on DP, and discuss decomposability in the context of DP. Finally, we survey related work for DP median and DP heavy hitters grouped by the before mentioned privacy models.

Chapter 4 describes our methodology to address the research question stated above. We first detail how we assess security of our protocols via simulation-based arguments, and how we assess privacy by accounting for the worst-case privacy loss. Then, we detail our methodologies to assess accuracy, mainly, absolute error for the median and non-cumulative rank for heavy hitters, and to assess efficiency, i.e., running time and communication of our implementations in a WAN.

Chapter 5 presents our secure two-party protocol $\mathsf{EM}_{\mathsf{med}}$ for rank-based statistics, illustrated for the DP median. First, we describe a high-level overview of $\mathsf{EM}_{\mathsf{med}}$, its building blocks, and how we satisfy DP. To support large data sets, we consider pruning. Pruning requires a privacy relaxation whose influence on accuracy and privacy we discuss and evaluate. Then, we formalize $\mathsf{EM}_{\mathsf{med}}$ and prove its semi-honest security. We detail extensions to $\mathsf{EM}_{\mathsf{med}}$ for multiple parties and malicious adversaries. Finally, we empirically evaluate our privacy relaxation and its accuracy, measure running time and communication of $\mathsf{EM}_{\mathsf{med}}$ in multiple real-world WANs between Ohio and N. Virgina, Canada, and Frankfurt, respectively, and compare $\mathsf{EM}_{\mathsf{med}}$ to our closest related work.

Chapter 6 presents our secure multi-party protocol EM^* for decomposable aggregate functions. First, we define decomposability and list applications with decomposable utility functions, which includes the median. Furthermore, we leverage decomposability to divide the data domain in subranges, and iteratively select increasingly smaller subranges, until we find the DP median. Then, we discuss accuracy for the DP median, and detail multiple alternatives for secure exponentiations to compute selection weights. We proof the semi-honest security of our differentially private protocol EM^* and detail an extension for malicious adversaries. Finally, we evaluate running time and communication of EM^* and its variation GM^* in a real-world WAN (Frankfurt–Ohio) and compare accuracy to related work.

Chapter 7 presents our secure multi-party protocols HH and PEM for DP heavy hitters. First, we present a non-private approach and a local-model protocol to discover heavy hitters on which we base our protocols HH and PEM, respectively. We adapt the existing approaches for differential privacy in the central model with a trusted third party. Then, we replace the trusted third party with MPC, detail optimizations of our MPC protocol, and show that it is semi-honestly secure. Finally, we measure running time as well as communication for HH and PEM in a real-world WAN (Frankfurt–Ohio), and compare accuracy to the state-of-the-art solution in the local model [WLJ19].

Chapter 8 concludes this thesis where we summarize our main insights and contributions per chapter.

2 Preliminaries

In Section 2.1, we present preliminaries for secure multi-party computation (MPC). In Section 2.2, we describe preliminaries for differential privacy (DP). We assume a familiarity with basic mathematical notation and present some notation and general preliminaries next.

Algorithm. An *algorithm* f is a finite sequence of operations applied on an input i to produce an output o, denoted as $o = f(i)$. A *(cryptographic) protocol* is a description of the execution of algorithms run jointly by multiple parties, including their interactions, the structure of exchanged messages (e.g., encryption method, number representation) and how they are processed. Algorithms are modelled as Turing machines which read inputs from an input tape, perform operations on a working tape, and write the output to an output tape. *Probabilistic polynomial-time* (PPT) algorithms run in time that is polynomial in the length of the input and are equipped with an additional random tape initialized with randomness to allow non-deterministic behavior. We use PPT algorithms to model computationally bounded adversaries trying to break secure protocols whose complexity is governed by a security parameter κ. The security parameter is commonly given to the adversary as input in unary encoding, i.e., 1^κ, which we also assume but do not explicitly state. In protocol descriptions, we use upper case letters mainly to denote *arrays*, i.e., ordered, indexed lists, and write the index in square brackets. For example, $A[i]$ refers to the value at index i in array A. We also use $x \leftarrow y$ to denote assignment of value y to variable x.

Big O notation describes asymptotic behavior of functions (and the algorithms implementing them) where $f(n) = O(g(n))$ denotes that $g(n)$ upper bounds $|f(n)|$ up to constant factors. Formally, $|f(n)| \leq c \cdot g(n)$ for some $c > 0$ and all n larger than some threshold.

Data set and domain. We consider a set of parties $\mathcal{P} = \{P_1, \ldots, P_n\}$, where party P_i holds a data element d_i (also called datum), and D denotes their combined *data set* (or *database*). The data set $D = \{d_1, \ldots, d_n\} \in \mathfrak{D}^n$ consists of elements from *data domain* $\mathfrak{D}$. *Duplicates* are non-distinct data elements, i.e., $d_i = d_j$ with $i \neq j$. A *range* $[a, b]$ over domain $\mathfrak{D}$ is the set containing all domain elements $x \in \mathfrak{D}$ satisfying $a \leq x \leq b$ and $(a, b]$ denotes the half-open range excluding a (likewise $[a, b)$ excludes b).

Number Representation. We mainly operate on *integers* ($\mathbb{Z}$) which allow more efficient secure protocols [ABZS13]. The subset of integers from 0 to $p - 1$ are denoted $\mathbb{Z}_p$, which is a field if p is prime. *Rational numbers* ($\mathbb{Q}$) can be expressed as integers via *fixed-point number representation.* A binary number of bit-length b can represent $d \in \mathbb{Q}$ as $d' \in \mathbb{Z}$ if $d = d' \cdot 2^{-f}$ with $-2^{b-1} + 1 \leq d' \leq 2^{b-1} - 1$ and scaling factor 2^{-f} where $f \in \mathbb{N}$ [CDH10, Section 2]. *Real numbers* ($\mathbb{R}$) are approximated via *floating-point number representation.* We adopt the notation from Aliasgari et al. [ABZS13] and represent a floating-point number f as $(1 - 2s)(1 - z) \cdot v \cdot 2^x$ with sign bit s set when the value is negative, zero bit z only set when the value is zero, l_v-bit significand v, and l_x-bit exponent x. Thus, a floating point value f is a 4-tuple (v, x, s, z). To refer to, e.g., the significand v of f we write $f.v$. Note that we sometimes use a fraction with a small denominator in our protocol description (e.g., $n/2$) but implicitly assume fractions (and the values they interact with) to be expressed as a scaled integer and only distinguish between operations on integers and floating-point numbers.

Probability. A *random variable* X can take a value x_i from a sample space Ω with probability p_i, which we write as $p_i = \Pr[X = x_i]$. A *(probability) distribution* is a collection of probabilities for all possible samples. The *probability mass* is the sum (resp., integral) of a subset of samples from a discrete (resp., continuous) distribution. Probabilities are positive and their total mass (for all of Ω) equals 1. Unnormalized probabilities whose total mass is not 1 are called *weights*. The *cumulative distribution function* $F(x_i) = \Pr[X \leq x_i]$ gives the probability mass for all $x_j \in \Omega$ with $x_j \leq x_i$. We let $X \sim P$ denote that random variable X follows probability distribution P and *sampling* refers to the computation of a sample $X \sim P$ given P. The *expected value* of a random variable X over a discrete distribution is $\mathbb{E}[X] = \sum_{i=1}^{k} x_i p_i$ where $x_1, \ldots, x_k$ are all possible values for X and their corresponding probabilities are $p_1, \ldots, p_k$.

Negligible. A function $f : \mathbb{N} \to \mathbb{R}$ is called *negligible* if for every positive polynomial $p(\cdot)$ there exists an N such that for all $n > N$, $f(n) < \frac{1}{p(n)}$. We write $\mathsf{negl}(x)$ to denote a function negligible in parameter x, e.g., $\mathsf{negl}(x) = 2^{-x}$.

2.1 Secure Multi-party Computation

Secure multi-party computation (MPC) allows a set of two or more parties $\mathcal{P} = \{P_1, \ldots, P_n\}$, where party P_i holds sensitive input d_i, to jointly compute a function $y = f(d_1, \ldots, d_n)$ while protecting their inputs [Gol09, HL10]. The secure computation must be *correct*, i.e., the correct y is computed, and *secret*, i.e., only y and nothing else is revealed. The *secrecy* property is typically called *privacy* in the literature, however, we call it (input) secrecy, to distinguish it from (output) privacy. We assume the existence of secure communication channels for each pair of input parties, as commonly provided by secure computation frameworks [AKR⁺20, DSZ15a, Kel20].

In the following, we describe common security models in Section 2.1.1. In Section 2.1.2, we describe cryptographic primitive and briefly summarize implementation paradigms for MPC. Then, we detail the paradigms used in our protocols, namely, garbled circuits in Section 2.1.3 and secret sharing in Section 2.1.4, and conversions between them in Section 2.1.5. Finally, we list basic MPC protocols (e.g., secure comparison) used in our protocols in Section 2.1.6.

2.1.1 Security Models

Security guarantees of MPC are mainly based on the behavior and computational power of an *adversary*, who corrupts a subset of the parties, views their internal state, reads received messages and, possibly, alters the corrupted parties actions during the protocol execution [Gol09, Section 7.1.1], [HL10, Section 1.1], [LK15, Section 10].

Behavior

In the *semi-honest model* a passive adversary behaves honestly and does not deviate from the protocol. However, the adversary tries to infer additional information that should remain secret from the messages received during the protocol execution. Semi-honest adversaries are also called honest-but-curious. In the *malicious model* an active adversary can deviate from the protocol execution, e.g., alter messages. Typically, maliciously-secure protocols are less efficient than protocols in the semi-honest model [HL10, Section 1.1].

Computational Power

An adversary's computational power further specifies the security model, leading to different security notions. *Computational indistinguishability* refers to security against adversaries with bounded computational power (PPT) where the computational complexity is governed by a *security parameter* κ.

Definition 1 (Computational Indistinguishability)**.** *Two sets of indexed distributions* $\mathcal{X} = \{X_i\}_{i \in \mathbb{N}}$, $\mathcal{Y} = \{Y_i\}_{i \in \mathbb{N}}$ *are said to be* computational indistinguishable*, denoted as* $\mathcal{X} \overset{c}{\approx} \mathcal{Y}$*, if for every PPT* $\mathcal{A}$ *and* $\kappa \in \mathbb{N}$

$$|Pr[\mathcal{A}(X \sim X_\kappa) = 1] - Pr[\mathcal{A}(Y \sim Y_\kappa) = 1]| \leq \mathsf{negl}(\kappa).$$

In the presence of adversaries with unbounded computational resources, two notions of security can be distinguished. One notion is *statistical indistinguishability*, where a *statistical security parameter* σ restricts the probability of an adversary to learn a party's input. More formally, the probability (for finite Ω) is bounded by the statistical distance

$$\frac{1}{2} \sum_{r \in \Omega} |\Pr[X \sim X_\kappa = r] - \Pr[Y \sim Y_\kappa = r]|.$$

The other notion is *information-theoretic security*, where the statistical security parameter can be seen as infinite, i.e., an adversary has zero probability to learn any party's input.

Our Model

In this work, we consider the semi-honest model with computationally-bounded parties and we discuss extensions of our protocols to the malicious model. The MPC frameworks, in which we implement our protocols (described in Section 4.5), use security parameter κ as well as an additional statistical security parameter σ [AKR+20, DSZ15a, Kel20]. Utilizing both security parameters simultaneously is interpreted as allowing security violations with probability at most $2^{-\sigma} + \mathsf{negl}(\kappa)$ [EKR+18, Section 2.1].

2.1.2 Primitives & Paradigms

First, we describe basic building blocks for MPC protocols, also called *cryptographic primitives*. Then, we provide a brief overview of *implementation paradigms* for MPC. With these preliminaries, we detail the MPC paradigms we employ, namely, garbled circuits and secret sharing, in the following Sections 2.1.3 and 2.1.4, respectively.

Cryptographic Primitives

Hash Function. A *hash function* $H : \{0,1\}^* \to \{0,1\}^l$ is a deterministic function that compresses an input, represented as binary strings of arbitrary length, to an output of fixed bit-length l. A *cryptographic hash function* satisfies additional properties, namely, pre-image resistance, i.e., given image $H(x)$ it is hard to find pre-image x, and collision-resistance, i.e., finding any x, x' ($x \neq x'$) with collision $H(x) = H(x')$ is hard. Hardness is defined via a PPT adversary who only succeeds in finding inputs as above with probability negligible in l (often set to κ). While it should be efficient to compute $H(x)$ with knowledge of x, the reverse is decidedly not the case – a property known as *one-way*.

Encryption. A *symmetric encryption scheme* consists of three PPT algorithms Gen, Enc, Dec. Algorithm Gen takes 1^κ as input and outputs a key k. Encryption algorithm Enc transforms a plaintext m into a ciphertext c where the transformation is controlled by a secret key k. We write $c \leftarrow \mathsf{Enc}_k(m)$ to denote this transformation where key k and plaintext m are input to Enc which outputs c. Decryption algorithm Dec reverses the transformation, i.e., $m \leftarrow \mathsf{Dec}_k(c)$.

An *asymmetric encryption scheme* lets Gen produce two keys, a public key for encryption Enc (publicly available), and a secret key for decryption Dec (kept secret).

Symmetric schemes are computationally more efficient than asymmetric schemes as the latter require computational hardness assumptions (e.g., factoring, discrete logarithm) [IR89, CO15] and typically uses modular exponentiations [NP01]. Encryption schemes are basic building blocks for cryptographic protocols. We refer to Goldreich [Gol09, Section 5.2] for technical details of encryption schemes.

Oblivious Transfer. A powerful cryptographic primitive is *oblivious transfer* (OT) [Rab81]. In principle, OT is equivalent to MPC [Kil88], i.e., OT suffices to perform any MPC and MPC can provide OT, and several OT protocols exist [BM89, NP01, CO15]. In 1-out-of-k OT a receiver receives one of k possible secrets from a sender, without the sender learning which one. OT protocols require costly computations, i.e., asymmetric cryptography [NP01]. However, OT extensions [Bea96, IKNP03] can efficiently extend few base OTs into many with more efficient symmetric cryptography. For a concrete OT protocol description, we refer to "Simplest OT" from Chou and Orlandi [CO15, Figure 1] which is based on the Diffie-Hellman key exchange [DH76].

Implementation Paradigms

There are two main implementation paradigms for MPC [EKR+18, KPR18]: *garbled circuits* [Yao86][1], where the parties construct a (large, encrypted) Boolean circuit and evaluate it at once, and *secret sharing* [Sha79, Bla79] where the parties interact for each arithmetic circuit gate. In general, the former allows for constant number of rounds but requires larger bandwidth (as fewer, but bigger messages are sent), and the latter has low bandwidth (small messages per gate) and high throughput, where the number of rounds depends on the circuit depth. Alternative paradigms are *(partially) homomorphic encryption* allowing, e.g., secure evaluations of either addition [Pai99] or multiplication [RSA78, ElG85], and *fully homomorphic encryption* [Gen09] supporting both. We focus on schemes based on secret sharing and garbled circuits as they are more efficient for general purpose computations [EKR+18] and are supported by mature frameworks for secure computation (see Section 4.5).

Recent MPC typically operates in two phases [BDOZ11, DPSZ12, KPR18]:

- a slow *offline phase* to pre-compute correlated randomness,
- and a fast *online phase* relying on the material from the offline phase.

Notably, the offline phase does not depend on the inputs or what computation is performed except for an upper bound on the number of required multiplications [BDO14]. The offline phase computes, e.g., base OTs for garbled circuits (Section 2.1.3) or Beaver triples [Bea91] for secret sharing (Section 2.1.4). The online phase is generally more efficient since the offline phase requires asymmetric cryptography [KRSW18]. Our evaluations always consider the *entire* MPC execution and measure the offline and online phase together (Section 5.3, 6.3, 7.3).

[1] Yao described a garbled circuit for two parties in an oral presentation about secure function evaluation [Yao86], the first written description is by Goldreich et al. [GMW87], and the first proof was given by Lindell and Pinkas [LP09].

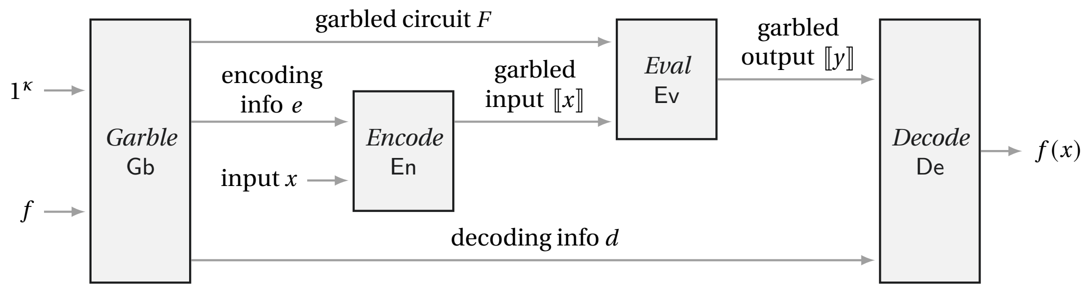


Figure 2.1: Components of a Garbling Scheme $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev})$ [BHR12, Fig. 1]. Gb produces string encodings: garbled function F, encoding function e, and decoding function d. $\mathsf{En}(x, e)$ produces garbled input $[\![x]\!]$. $\mathsf{Ev}([\![x]\!], F)$ outputs the garbled output $[\![y]\!]$, which can be decoded to the actual output y if d is known. Final output $y = \mathsf{De}(d, [\![y]\!])$ must equal $\mathsf{ev}(f, x)$.

2.1.3 Garbled Circuits

Garbled circuits are cryptographic Boolean circuits. Bellare et al. [BHR12] formalize a scheme to create and evaluate garbled circuits, whose components are shown in Figure 2.1 and described in the following.

Definition 2 (Garbling Scheme)**.** *Let* string *refer to a sequence of bits $b \in \{0, 1\}$ of finite length used to describe a function, e.g., as a circuit. A* garbling scheme *is the tuple of algorithms $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev}, \mathsf{ev})$, where* Gb *is probabilistic and all others are deterministic.*

- $(F, e, d) \leftarrow \mathsf{Gb}(1^\kappa, f)$*: Takes as input a security parameter $\kappa \in \mathbb{N}$ and the string f describing the original function to evaluate,* $\mathsf{ev}(f, \cdot)$*, and outputs string F describing the* garbled function, $\mathsf{Ev}(F, \cdot)$*, string e describing an* encoding function, $\mathsf{En}(e, \cdot)$*, and string d describing a* decoding function, $\mathsf{De}(d, \cdot)$*, as defined in the following.*
- $[\![x]\!] \leftarrow \mathsf{En}(e, x)$ *is an* encoding function*, described by string e, that maps an* initial input $x \in \{0, 1\}^l$ *to a* garbled input $[\![x]\!]$.
- $y \leftarrow \mathsf{De}(d, [\![y]\!])$ *is a* decoding function*, described by string d, that maps a* garbled output $[\![y]\!]$ *to a* final output y.
- $[\![y]\!] \leftarrow \mathsf{Ev}(F, [\![x]\!])$ *is an* evaluation function*, described by string F, that maps a garbled input* $[\![x]\!]$ *to a* garbled output $[\![y]\!]$.
- $y \leftarrow \mathsf{ev}(f, x)$ *is an* evaluation function*, described by string f, that maps the input x to the output y, where* $\mathsf{ev}(f, \cdot) : \{0, 1\}^l \rightarrow \{0, 1\}^m$ *is the* original function *we want to garble.*

A garbling scheme fulfills the following properties:

Correctness: *Decoded garbled output* $\mathsf{De}(d, \mathsf{Ev}(F, \mathsf{En}(e, x)))$ *equals actual output* $\mathsf{ev}(f, x)$*,*

Secrecy: $(F, [\![x]\!], d)$ *reveals nothing beyond $f(x)$.*

For further technical details, we refer to Bellare et al. [BHR12]. In our protocols, we let $\mathsf{En}(\cdot)$, $\mathsf{De}(\cdot)$ denote the encoding (garbling) and decoding (de-garbling) operations and omit the corresponding encoding and decoding strings e, d.

To describe Boolean circuits, we require some notation. A binary number $b \in \{0, 1\}^l$ is represented as a sequence of bits $b = b_l \dots b_1$. We say a bit b_i in b is set if it is 1 and unset if it is 0. Boolean circuits consist of a series of *gates* realizing logical operations. Gates are connected by *wires* which feed the output from one gate as input into other gates. A basic logical operation with a single input and output bit is NOT (with operator $\neg$), outputting the inverse of its input (i.e., 0 on input 1, 1 on input 0). Basic logical bit operations with two input bits and one output bit are AND ($\wedge$), returning bit 1 only if both input bits are 1, OR ($\vee$), returning 1 only if both input bits are not 0, and XOR ($\oplus$), returning 1 only if both input bits differ; otherwise 0 is returned by these operations. The operations handle a sequence of bits as inputs by applying the operation bit-wise. Also, we extend a single bit $c \in \{0,1\}$ to a sequence $c \dots c \in \{0^l, 1^l\}$ if c is one of the inputs of an operation on l bits, e.g., $x_l \dots x_1 = \mathsf{XOR}(b, c)$ with $x_i = b_i \oplus c$.

Having defined the basic terms of circuits, we can now describe the original semi-honest garbling scheme from Yao [Yao86].

Yao's Garbled Circuits

Yao's garbled circuit protocol consists of two parties: the *garbler* (or generator) with input $x_1 \in \{0,1\}^l$, which creates the garbled circuit, and the *evaluator* with input $x_2 \in \{0,1\}^l$, which evaluates the garbled circuit without learning intermediate values.

Garbling: The garbler executes $(F, e, d) \leftarrow \mathsf{Gb}(1^\kappa, f)$, where e contains random keys (also called wire labels) for each wire, F is a representation of the garbled circuit for f, and d maps the last output (keys) to actual bits. Then, the garbler garbles his input as $[\![x_1]\!] \leftarrow \mathsf{En}(e, x_1)$.

In more detail, the garbler transforms f into a Boolean circuit and associates two random keys $k_0^w, k_1^w \in \{0,1\}^\kappa$ for each possible bit value $(0,1)$ on each wire w. Assume each gate g has two input wires i, j and one output wire q and $g(\cdot,\cdot)$ denotes the logical operation provided by the gate. For each gate, the garbler uses (input wire) keys $k_{b_i}^i, k_{b_j}^j$, for all $b_i, b_j \in \{0,1\}$, to encrypt (output wire) keys $k_{g(b_i,b_j)}^q$, i.e.,

$$\mathsf{Enc}_{k_{b_i}^i, k_{b_j}^j}\left(k_{g(b_i,b_j)}^q\right),$$

with a suitable symmetric encryption scheme Enc[2]. As an example, an AND-gate results in the following ciphertexts, also called *garbled table*:

$$\begin{bmatrix} \mathsf{Enc}_{k_0^i, k_0^j}\left(k_0^q\right) \\ \mathsf{Enc}_{k_0^i, k_1^j}\left(k_0^q\right) \\ \mathsf{Enc}_{k_1^i, k_0^j}\left(k_0^q\right) \\ \mathsf{Enc}_{k_1^i, k_1^j}\left(k_1^q\right) \end{bmatrix},$$

where the rows are randomly permuted. Overall, the garbled circuit F is the collection of garbled tables required to compute f.

Sending: After the garbling, the garbler sends the garbled circuit F, his garbled input $[\![x_1]\!]$ and decoding information d to the evaluator.

[2] For example, AES [DSZ15a, Section V.A]. Alternatively, with cryptographic hash function H and concatenation denoted as $||$, one can set $\mathsf{Enc}_{k_{b_i}^i, k_{b_j}^j}(x) = H(k_{b_i}^i \,||\, k_{b_j}^j \,||\, q) \oplus x$ [HEKM11, Section 3.4]

Input Retrieval: Evaluator (as receiver) and garbler (as sender) execute an OT protocol such that the evaluator only learns her garbled input $[\![x_2]\!]$, i.e., keys corresponding to the bits in x_2, and the garbler does not learn the evaluator's input bits.

Evaluation: The evaluator evaluates the garbled circuit for $[\![x]\!] = ([\![x_1]\!], [\![x_2]\!])$ and outputs the result $\mathsf{De}(d, \mathsf{Ev}(F, [\![x]\!]))$.

Note that the evaluator cannot learn both keys per wire as this allows evaluation of the circuit (on fixed x_1) with all possible values for x_2 which reveals more than intended [LP09, Section 1]. Many optimizations of Yao's initial protocol have been developed, including (but not limited to) point-and-permute (decrypting one table entry instead of all via a "hint" bit) [BMR90], garbled row reduction (reducing the table size by choosing keys such that one ciphertext is 0) [NPS99], and FreeXOR (removing decryptions for XOR gates by setting their output key to be the XOR of the input keys) [KS08]. Furthermore, this semi-honest two-party scheme can be extended to malicious parties (e.g., *cut-and-choose*: constructing multiple garbled circuits, opening and checking one half randomly, and using majority output of the rest) [HL10, Section 4.1.1] and generalized to multiple parties (e.g., distributed circuit generation with secret-shared wire labels) [BMR90].

2.1.4 Secret Sharing

A (t, n)-secret sharing scheme splits a secret s into n shares s_i such that at least t shares are required to reconstruct the secret. Evans et al. [EKR+18] formally define a secret sharing scheme as follows:

Definition 3 (Secret Sharing Scheme)**.** *Let $\mathcal{S}$ be the domain of secrets and $\mathcal{T}$ the domain of shares. Let $\mathsf{Shr} : \mathcal{S} \to \mathcal{T}^n$ be a (possibly randomized) sharing algorithm, and $\mathsf{Rec} : \mathcal{T}^l \to \mathcal{S}$ be a reconstruction algorithm. A (t, n)-*secret sharing scheme *is a pair of algorithms $(\mathsf{Shr}, \mathsf{Rec})$ that satisfies these two properties:*

Correctness: *Let $(s_1, \dots, s_n) = \mathsf{Shr}(s)$ denote the sharing of s among the n parties. Then,*

$$Pr\Big[\forall l \geq t,\ \mathsf{Rec}(s_{i_1}, \dots, s_{i_l}) = s\Big] = 1,$$

where $\{i_1, \dots, i_l\} \subseteq \{1, \dots, n\}$.

Secrecy: *Any set of shares of size less than t does not reveal anything about the secret (in the information theoretic sense). More formally, for any two secrets $a, b \in \mathcal{S}$ and any possible vector of shares $v = (v_1, \dots, v_l)$, such that $l < t$,*

$$Pr[v = \mathsf{Shr}(a)|_l] = Pr[v = \mathsf{Shr}(b)|_l],$$

where $|_l$ denotes appropriate projection on a subspace of l elements.

A scheme with these properties is secure in the semi-honest model; it is even secure in the malicious model if the reconstruction and secrecy properties hold against malicious adversaries [EKR+18]. We refer to Pullonen et al. [PBS12] for security proofs of secret sharing. We let $\langle s \rangle$ denote the vector $(s_1, \dots, s_n)$, i.e., the sharing of s, and write $\langle s \rangle_i$ instead of s_i if we want to emphasize that it is the share of party i. Secret sharing can be realized, e.g., over integers and polynomials, and we describe these realizations next.

Additive Secret Sharing

Our two-party protocol $\mathsf{EM}_{\mathsf{med}}$ (Section 5) uses *additive secret sharing* [DSZ15a, Section III.A] as well as garbled circuits. For additive secret sharing, we require all values to be in the ring $\mathbb{Z}_{2^{64}}$ and (implicitly) perform operations on secret shares modulo 2^{64}.

Sharing: To construct an additive $(2,2)$-sharing of a secret s the party holding s draws a uniformly random $r \in \mathbb{Z}_{2^{64}}$ and sends $s - r$ to the other party, i.e., $(r, x - r) = (s_1, s_2) = \mathsf{Shr}(s)$.

Evaluation: Note that addition with linearly secret-shared values $\langle x\rangle, \langle y\rangle$ is straightforward since $\langle x\rangle+\langle y\rangle = (x_1+y_1, x_2+y_2)$, as is multiplication with a public value z where $z\cdot\langle x\rangle = (z\cdot x_1, z\cdot x_2)$. However, multiplication of secret shared values $\langle x\rangle$ and $\langle y\rangle$ requires additional techniques, e.g., a precomputed *Beaver triple* $\langle a\rangle, \langle b\rangle, \langle c\rangle$ such that $c = a \cdot b$ [Bea91]. Given such a triple, the parties compute $\langle x - a\rangle, \langle y - b\rangle$, and $\alpha = \mathsf{Rec}(\langle x - a\rangle)$ and $\beta = \mathsf{Rec}(\langle y - b\rangle$ to obtain

$$\langle xy\rangle = \langle c\rangle + \alpha\langle b\rangle + \beta\langle a\rangle - \alpha \cdot \beta.$$

Beaver triplets can be constructed, e.g., via additive homomorphic encryption and oblivious transfer [DSZ15a, Section III.A 4) & 5)]. An alternative technique for multiplication uses *replicated secret sharing* à la Maurer [Mau06] where shares are replicated among many parties.

Reconstruction: The reconstruction takes both shares as inputs and adds the shares to produce the output, i.e, $s = \langle s\rangle_1 + \langle s\rangle_2 = \mathsf{Rec}(s_1, s_2)$.

Shamir's Secret Sharing

Our multi-party protocols EM^*, HH, PEM (Sections 6, 7) use *Shamir's secret sharing* [Sha79]. The geometric intuition behind Shamir secret sharing is that a certain number of points suffice to uniquely define a curve, i.e., a polynomial of degree $t - 1$ is uniquely determined by t points.

Sharing: For a (t, n)-secret sharing, a secret s is encoded as point $p(0)$ of a polynomial p with degree $t - 1$ over a finite field $\mathbb{F}$. In more detail, $p(x) = s + c_1x + c_2x^2 + \cdots + c_{t-1}x^{t-1}$ with coefficients $c_i \in \mathbb{F}$ where $\mathbb{F}$ is, e.g., the set of integers modulo a large prime. The sharing of s consists of points of p, i.e., $(p(1), \ldots, p(n)) = (s_1, \ldots, s_n) = \mathsf{Shr}(s)$. Each party P_i receives point $p(i)$ and at least t points are required to reconstruct p and thus s.

Evaluation: Addition of shares and multiplication with a public $c \in \mathbb{F}$ is straightforward as with additive secret sharing and can be directly performed on the underlying polynomials (resp., their points). Multiplication, once again, requires special handling as a naive approach fails: Let f_a, f_b denote the polynomials for secrets a, b then polynomial $h(x) = f_a(x) \cdot f_b(x)$ represents secret $a \cdot b$. However, the degree of h is $2t - 2$ (and will increase for further multiplications) and it is not random (e.g., it is not irreducible as it is the product of two polynomials) [BGW88]. As before, pre-computed Beaver triples can be used to allow multiplication of secret values. Alternatively, the parties share a value of the product polynomial as described by Genaro et al. [GRR98, Figure 2].

Reconstruction: Given $k \geq t$ shares $s_{i_1}, \ldots, s_{i_k}$ where $I = \{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$, the secret s can be reconstructed via polynomial interpolation at point 0, i.e., $s = \sum_{i\in I}\left(s_i \prod_{j\in I, i\neq j} \frac{-j}{i-j}\right)$.

Conversion	*Output*	*#Operations*	*#Messages*	*Communication (bits)*
GC2SS($[\![a]\!]$)	$\langle a \rangle$	$6l$	2	$l\kappa + (l^2 + l)/2$
SS2GC($\langle a \rangle$)	$[\![a]\!]$	$12l$	2	$6l\kappa$

Table 2.1: Complexity of converting between garbled values and secret shares for l-bit integers as implemented in ABY [DSZ15a, Table I], where *#Operations* refers to the number of symmetric cryptographic operations, *#Messages* refers to the number of messages in the online phase, and *Communication* (sent bits) relies on the security parameter κ of the symmetric encryption scheme.

MPC Protocol	*Output*	*#Gates (2-input non-*XOR*)*
XOR($[\![a]\!], [\![b]\!]$)	$[\![a \oplus b]\!]$	0
AND($[\![a]\!], [\![b]\!]$)	$[\![a \wedge b]\!]$	l
OR($[\![a]\!], [\![b]\!]$)	$[\![a \vee b]\!]$	l
LT($[\![a]\!], [\![b]\!]$)	$[\![a < b]\!]$	l
Mux($[\![a]\!], [\![b]\!], [\![c]\!]$)	$[\![a]\!]$ if bit $c = 1$ else $[\![b]\!]$	l
Add($[\![a]\!], [\![b]\!]$)	$[\![a + b]\!]$	l
Sub($[\![a]\!], [\![b]\!]$)	$[\![a - b]\!]$	l

Table 2.2: Basic garbled circuits protocols with number of (2-input non-XOR) gates for l-bit integers [KSS09, Table 2] used in ABY [DSZ15a, Section III.C] with FreeXOR [KS08].

2.1.5 Conversion between Additive Secret Shares & Garbling

Some operations are more efficient with MPC based on secret sharing instead of garbled circuits and vice versa. For example, addition of l-bit integers requires $O(l)$ gates in a Boolean circuit but only one gate in an arithmetic circuits with secret sharing. Comparisons, on the other hand, are more efficient with garbled circuits than secret sharing. To leverage the advantage of both paradigms in a mixed-protocol execution, one requires conversions between them.

Converting an additive secret sharing to a garbled value is straightforward, i.e., one evaluates a garbled circuit to add the shares. Similarly, evaluating a subtraction circuit for garbled value $[\![x]\!]$ and a random value $r \in \mathbb{Z}_{2^{64}}$ (chosen by the garbler) produces $[\![x - r]\!]$, which the evaluator is allowed to decode. Thus, the garbler knows r and the evaluator $x - r$, i.e., an additive $(2, 2)$-secret sharing of x. ABY provides a more efficient conversion based on 1-out-of-κ OT [DSZ15a, Section IV.B], and we list the complexity of ABY's conversions in Table 2.1. We denote a conversion from secret shares to garbled circuits as SS2GC and the reverse as GC2SS.

2.1.6 Basic MPC protocols

In the following, we list the basic MPC protocols required as building blocks for our protocols as well as their complexities. If not otherwise noted, all protocols operate on integers.

Garbled Circuit Protocols

Our protocol $\mathsf{EM}_{\mathsf{med}}$ requires the logical operations for garbled circuits listed in Table 2.2 and conversions between garbled values and secret shares as listed in Table 2.1 in Section 2.1.5. To make our protocol descriptions clearer, we list Sub as a separate operation, although it is expressed

MPC protocol	*Output / Functionality*	*Rounds*	*Interactive Operations*
$\mathsf{Rand}(b)$	$\langle r \rangle$, uniform random b-bit value r	0	0
$\mathsf{Add}(\langle a \rangle, \langle b \rangle)$	$\langle a + b \rangle$	0	0
$\mathsf{Sub}(\langle a \rangle, \langle b \rangle)$	$\langle a - b \rangle$	0	0
$\mathsf{Rec}(\langle a \rangle)$	a	1	1
$\mathsf{Mul}(\langle a \rangle, \langle b \rangle)$	$\langle a \cdot b \rangle$	1	2
$\mathsf{Mux}(\langle a \rangle, \langle b \rangle, \langle c \rangle)$	$\langle a \rangle$ if bit $c = 1$ else $\langle b \rangle$	1	2
$\mathsf{Mod2m}(\langle a \rangle, b)$	$\langle a \mod 2^b \rangle$ with public b	$\log b + 2$	$3b - 1$
$\mathsf{Trunc}(\langle a \rangle, b)$	$\langle \lfloor a/2^b \rfloor \rangle$ with public b	$\log b + 2$	$3b - 1$
$\mathsf{LT}(\langle a \rangle, \langle b \rangle)$	$\langle 1 \rangle$ if $a < b$ else $\langle 0 \rangle$	$\log(l-1) + 2$	$3l - 4$
$\mathsf{LE}(\langle a \rangle, \langle b \rangle)$	$\langle 1 \rangle$ if $a \leq b$ else $\langle 0 \rangle$	$\log(l-1) + 2$	$3l - 4$
$\mathsf{EQ}(\langle a \rangle, \langle b \rangle)$	$\langle 1 \rangle$ if $a = b$ else $\langle 0 \rangle$	$\log l + 2$	$2l$
$\mathsf{Int2FL}(\langle a \rangle)$	$\langle a \rangle_{\mathsf{FL}}$	$\log v + 13$	$\log v(2v-3) - 11$
$\mathsf{Add}_{\mathsf{FL}}(\langle a \rangle_{\mathsf{FL}}, \langle b \rangle_{\mathsf{FL}})$	$\langle a + b \rangle_{\mathsf{FL}}$	$O(\log v)$	$O(v \log v + x)$
$\mathsf{Mul}_{\mathsf{FL}}(\langle a \rangle_{\mathsf{FL}}, \langle b \rangle_{\mathsf{FL}})$	$\langle a \cdot b \rangle_{\mathsf{FL}}$	11	$8v + 10$
$\mathsf{LT}_{\mathsf{FL}}(\langle a \rangle_{\mathsf{FL}}, \langle b \rangle_{\mathsf{FL}})$	$\langle 1 \rangle$ if $a < b$ else $\langle 0 \rangle$	6	$4v + 5x + 4\log x + 13$

Table 2.3: Basic MPC protocols and their complexity for l-bit integers, floats with v-bit significand and x-bit exponent [ABZS13, Table 1] [CDH10, Table 5] [EKM+14, Table 6] [Sec09, Table 10]. Per default we operate on integers and mark secret-shared floats as well as protocols operating on them with subscript FL.

via Add, i.e., addition of negative number. The more complex operations from Table 2.2 consist mainly of XOR gates, which can be evaluated for free [KS08], and AND gates. For example, $\mathsf{OR}(a, b)$ is expressed as $\mathsf{NOT}(\mathsf{AND}(\mathsf{NOT}(a), \mathsf{NOT}(b))$ with $\mathsf{NOT}(a) = \mathsf{XOR}(1, a)$ and $\mathsf{Mux}(a, b, c)$ can be represented as $\mathsf{XOR}(b, \mathsf{AND}(\mathsf{XOR}(a, b), c))$. We refer to the ABY documentation [Eng18, Section 3.2] and Kolesnikov et al. [KSS09] for details about efficient constructions.

Secret Sharing Protocols

Our protocols EM*, HH, and PEM use MPC protocols based on secret sharing listed in Table 2.3. We assume pre-computed Beaver triples for multiplication. MPC complexity is typically measured in the number of *interactive operations* and *rounds* [ABZS13, CDH10, EKM+14]. The number of interactive operations is an abstract measure of *computation complexity*, and interactive operations require exchanging messages with other parties. For example, share reconstruction and multiplication of shares are interactive operations, whereas addition of shares can be computed locally by each party. The number of rounds is a measure of *time complexity*, a round can consist of multiple interactive operations which are assumed to run in parallel and messages are sent in a single batch per round. For example, multiplication with pre-computed Beaver triples requires 2 interactive operations (share reconstructions) in 1 round (shares can be sent in one batch).

Per default we operate on integers, but a version of EM* also requires floating-point numbers, and we use subscript FL to denote protocols operating on floating-point numbers. For example, Add denotes addition on integers while $\mathsf{Add}_{\mathsf{FL}}$ denotes its floating-point equivalent.

As before, we list Sub as a separate operation, based on Add, to make our protocol descriptions clearer. Note that Mux(a, b, c) is implemented with one multiplication as $b + (a - b) \cdot c$, and in the offline-online paradigm Rand is an element of a pre-computed Beaver triple. For some protocols – namely Mod2m, Trunc, LT, LE, EQ – there exist implementations with a constant or a logarithmic number of rounds. We present only the complexity for logarithmic round versions, which provide better performance in practice [AKR+20]. We refer to Catrina and De Hoogh [CDH10, Table 5] for the complexity of constant-round versions and how pre-computation can reduce the complexity of logarithmic round versions.

2.2 Differential Privacy

In Section 2.2.1, we briefly discuss privacy notions leading to differential privacy (DP), and describe its connection to cryptography, before formalizing DP. We describe a notion of DP suited for MPC in Section 2.2.2. After defining DP, we describe its properties in Section 2.2.3, and how additive noise and probabilistic selection satisfy DP in Section 2.2.4. In Section 2.2.5, we discuss noise distributions suitable for distributed deployments of DP mechanisms.

2.2.1 From Syntactic to Semantic Notions of Privacy

Next, we briefly summarize the evolution of privacy notions based on Nissim and Wood [NW18, Section 2.2]. Initial research for anonymization considered *syntactic* privacy notions (e.g., k-anonymity [SS98, Sam01], l-diversity [MKGV07], t-closeness [LLV07]), which place requirements on how the anonymized data should look. Typically, syntactic notions are achieved by suppression of identifiers (e.g., name, government id number), detecting quasi-identifiers (unique combinations of non-identifiers), and coarsening them (e.g., replace values by ranges). For a thorough and formal overview of syntactic privacy notions, we refer to Desfontaines [Des20, Section 2]. Notable privacy incidents are due to incomplete suppression (e.g., "anonymized" search logs contained identifying search terms [BZJ06]), and neglecting auxiliary information (e.g., "anonymized" Netflix movie ratings were linked to public IMdB profiles [NS08]), and plenty of further examples exist [And21, Section 11], [LLSY16, Section 1.1.1], [DKM19, Appendix A], [DMHVB13, DMRS+15, NSR11, SH12]. Recently, the US census bureau showed that high-level aggregate statistics (demographic counts in residential areas) suffice to identify millions of individuals of the 2010 US census by linking it with commercially available data [DKM19, GAM19]. As a consequence of such reconstruction attacks, first formalized by Dinur and Nissim [DN03], the US census adopted differential privacy [Abo18].

Differential privacy, introduced by Dwork et al. [Dwo06, DMNS06], is a *semantic* notion, which places a requirement on the anonymization mechanism $\mathcal{M}$ itself. Informally, when the input data set of $\mathcal{M}$ changes in a single element[3], the effect on the output is bounded. Differential privacy is inspired by the cryptographic notion of *semantic security* [GM84], where an adversary seeing the output of an encryption scheme (ciphertext) has at most a negligible change to infer anything about the input (plaintext)[4]. However, a semantic privacy notion requires non-negligible information leakage to allow any meaningful statistic [DMNS06, Section 1]; if a single

[3] As noted by Kifer and Machanavajjhala [KM11], there are two natural ways to interpret "data sets D, D' that differ in a single entry". A neighboring data set D' can be created by either *replacing* an element in D (e.g., [DMNS06]) or by *adding/removing* an element (e.g., [Dwo06]). Throughout this work, we will use the add/remove interpretation in accordance with Li et al. [LLSY16].

[4] Except the length of the plaintext.

individual cannot (at least slightly) change the outcome of an anonymized statistic, then neither can a population of millions, and useful insights are impossible. Differential privacy bounds this information leakage in general, i.e., independent of the computational strength and auxiliary information an adversary might possess [Dwo08, Section 2]. The formal definition is as follows:

Definition 4 (Differential Privacy (DP))**.** *Data sets D, D', where D' is created from D by adding or removing an element are called* neighbors *and denoted $D \simeq D'$. A mechanism $\mathcal{M}$ satisfies (ϵ, δ)-* differential privacy*, where $\epsilon \geq 0, \delta \geq 0$, if for all neighboring data sets D, D', and all sets $S \subseteq Range(\mathcal{M})$*

$$Pr[\mathcal{M}(D) \in S] \leq \exp(\epsilon) \cdot Pr[\mathcal{M}(D') \in S] + \delta,$$

where $Range(\mathcal{M})$ denotes the set of all possible outputs of mechanism $\mathcal{M}$.

We abbreviate $(\epsilon, 0)$-DP as ϵ-DP. The original definition [Dwo06, DMNS06] with $\delta = 0$ is also called *pure differential privacy* to distinguish it from *approximate differential privacy* with $\delta > 0$. Privacy parameter ϵ, also called *privacy budget*, is a small constant [DR14], where smaller values correspond to a decrease in privacy loss. Typically, δ is assumed to be negligible in the size n of the data set [DR14]. The parameter δ has a similar motivation as the statistical security parameter σ in MPC, i.e., permitting a negligible probability to violate DP[5] to increase accuracy (see also Section 2.2.3). Privacy parameter ϵ, for which there is no equivalent in MPC, allows a finer trade-off between privacy and accuracy than δ. As noted before, some information has to be gained from the output to allow any meaningful statistic. Roth [McS16] points out that $\epsilon = 0$ provides perfect privacy, as all outputs are equally likely and inputs D, D' indistinguishable. But this comes at the price of zero accuracy as we gain no insights from uniformly random outputs. On the other hand, $\epsilon = \infty$ corresponds to perfect accuracy, as the raw data is revealed, but provides zero privacy. Notably, $\exp(\epsilon) \approx 1 + \epsilon$, for small values of ϵ[6], i.e., the probability to receive an output based on D is within multiplicative factor $1 + \epsilon$ of an output based on D'.

Various variations and relaxations of differential privacy exist, see, e.g., [DP20] for an overview, that relax the guarantee (e.g., average-case instead of worst-case) or use different metrics to measure the privacy loss (e.g., Rényi divergence). Note that Definition 4 (implicitly) assumes that mechanism $\mathcal{M}$ has access to the raw data D. In a *distributed* setting, where each party locally randomizes her datum d, the notion of *local* DP (LDP) [KLN+11] is used. Here, for any inputs $d, d' \in D$ the output changes are ϵ-bounded, i.e., $\Pr[\mathcal{M}(d) \in S] \leq \exp(\epsilon) \cdot \Pr[\mathcal{M}(d') \in S]$. Also, Definition 4 implicitly holds against a *computationally unbounded adversary* (i.e., no adversary restriction is defined), and later work considered restrictions on computational power [EKM+14, HMFS17, MPRV09, Vad17]. Due to our use of MPC, we define a computational notion of DP next.

2.2.2 Computational Differential Privacy

We consider semi-honest parties performing a joint secure computation in the presence of a *computationally-bounded adversary* $\mathcal{A}$, who tries to distinguish if the original D or a neighboring data set D' was used in the computation, and outputs a bit $b \in \{0, 1\}$ accordingly. Adversary $\mathcal{A}$ corrupts a subset C of the parties $\mathcal{P}$, and sees the *views* of corrupted parties, i.e., all exchanged messages and internal state. The view is also called the transcript of a protocol. Let $\texttt{VIEW}^{C}_{\Pi}(D) = \{\texttt{VIEW}^{p}_{\Pi}(D)\}_{p \in C}$ denote the *views* of corrupted parties $C \subset \mathcal{P}$ during the execution of protocol Π on input D. The following definition is based on Vadhan [Vad17, Section 10].

[5] For a thorough discussion of the subtleties in interpreting δ, we refer to the technical report from Meiser [Mei18].
[6] For $\epsilon \leq 0.138$, $\exp(\epsilon) - (1 + \epsilon) < 0.01$.

Definition 5 (Computational Differential Privacy)**.** *A randomized protocol Π implemented among a set of parties $\mathcal{P} = \{P_1, \ldots, P_n\}$ achieves* computational differential privacy *with regard to a coalition $C \subset \mathcal{P}$ of semi-honest parties of size at most t, if for all neighbors D, D' and probabilistic polynomial-time adversaries $\mathcal{A}$*

$$Pr\Big[\mathcal{A}\Big(\texttt{VIEW}_{\Pi}^{C}(D)\Big) = 1\Big] \leq \exp(\epsilon) \cdot Pr\Big[\mathcal{A}\Big(\texttt{VIEW}_{\Pi}^{C}(D')\Big) = 1\Big] + \delta(\kappa),$$

where $\delta(\kappa) = \delta + \mathsf{negl}(\kappa)$ with security parameter κ.

Implicitly, this assumes a datum of a party in C does not change between D and D', as otherwise the DP guarantee is trivially broken. The definition can be expanded to the malicious model by replacing the semi-honest parties $\mathcal{P}$ and semi-honestly secure protocol Π with malicious parties and a maliciously secure protocol. Notably, computational indistinguishability (Definition 1) is equivalent to computational differential privacy for $\epsilon = 0, \delta = 0$.

2.2.3 Properties of Differential Privacy

Differential privacy exhibits desirable properties. The privacy guarantee of differential privacy cannot be reduced via *post-processing*. No adversary can increase the privacy loss of a mechanism $\mathcal{M}$ when a data-independent function f is applied on $\mathcal{M}$, denoted as $f \circ \mathcal{M}$ [DR14, Section 2.3]. In other words, auxiliary information cannot reduce the privacy guarantee. For computational differential privacy (Definition 5), post-processing is restricted to computationally bounded adversaries.

Group privacy extends the neighboring definition from changing a single element to k elements which leads the privacy budget to increase linearly in k, i.e., a standard ϵ-DP mechanism satisfies $k\epsilon$-DP with group size $k > 1$ [DR14, Theorem 2.2].

Differential privacy supports the *composition* of k mechanisms $\mathcal{M}_1, \ldots, \mathcal{M}_k$ where each $\mathcal{M}_i$ satisfies (ϵ_i, δ_i)-DP. *Parallel composition* considers the application of each $\mathcal{M}_i$ on disjoint subsets of a data set and satisfies $(\max_{1\leq i\leq k} \epsilon_i, \max_{1\leq i\leq k} \delta_i)$-DP [LLSY16, Section 2.2.2]. *Sequential composition* considers sequential execution of mechanisms, i.e., $\mathcal{M}_k \circ \mathcal{M}_{k-1} \circ \cdots \circ \mathcal{M}_1$. Sequential composition satisfies at least $(\sum_{i=1}^{k} \epsilon_i, \sum_{i=1}^{k} \delta_i)$-DP [DR14, Theorem 3.16]. For (ϵ, δ)-DP mechanisms $\mathcal{M}_i$, i.e., mechanisms with the same privacy parameters, a tighter composition bound is $(\sqrt{2k \log(1/\delta')}\epsilon + k\epsilon(\exp(\epsilon) - 1), k\delta + \delta')$-DP where $\delta, \delta' \geq 0$ [DR14, Theorem 3.20].

Note that approximate DP ($\delta > 0$) requires less privacy budget (ϵ) than pure DP ($\delta = 0$) when k is large enough, as we show next.

Lemma 1. *Sequentially composing k mechanisms satisfying (ϵ, δ)-DP requires a smaller total privacy budget ϵ than $(\epsilon, 0)$-DP mechanisms when $k > \frac{2\log(1/\delta')}{(2-\exp(\epsilon))^2}$ with $\delta' > 0$.*

Proof. Running k (ϵ, δ)-DP mechanisms on the same data leads to a total privacy budget of $k\epsilon$ for pure DP mechanisms ($\delta = 0$), and $(\sqrt{2k \log(1/\delta')}\epsilon + k\epsilon(\exp(\epsilon) - 1), k\delta + \delta')$ for approximate DP mechanisms ($\delta, \delta' > 0$) [DR14, Theorem 3.20]. Therefore, $\sqrt{2k \log(1/\delta')}\epsilon + k\epsilon(\exp(\epsilon) - 1) < k\epsilon \Leftrightarrow \frac{1}{\sqrt{k}}\sqrt{2\log(1/\delta')} + (\exp(\epsilon) - 1) < 1 \Leftrightarrow k > \frac{2\log(1/\delta')}{(2-\exp(\epsilon))^2}$. (See also [MV16, Appendix A]). $\square$

For further details about composition we refer to Murtagh and Vadhan [MV16] and Dong et al. [DDR20].

2.2.4 Mechanisms

Differential privacy requires randomized algorithms called *mechanisms* in the DP literature. So far, we have detailed *what* notion of privacy DP offers. After defining a function's sensitivity, we can formalize *how* mechanisms satisfy DP.

The randomization magnitude depends on the privacy parameter ϵ and the *sensitivity* Δf of the function f evaluated on the data. Sensitivity is an upper bound on any individual's influence on the output of f, i.e., the largest possible difference of neighboring data sets evaluated on f.

Definition 6 (Sensitivity)**.** *The* (global) sensitivity *of a function* $f : \mathfrak{D}^n \to \mathbb{R}$ *is*

$$\Delta f = \max_{\forall D \simeq D'} |f(D) - f(D')| .$$

Various specialized or relaxed sensitivity notions exist [ABCP13, BBK17, DR14, NRS07] and later, in Section 3.6.1, we discuss related work with notions customized for the median.

Having defined the required noise magnitude for a function, we can now present mechanisms that achieve DP. DP mechanisms can be classified by how they randomize, i.e., with additive noise or via probabilistic selection.

Mechanisms with Additive Noise

Noise, added to the function output, is one way to achieve differential privacy, e.g., via the *Laplace mechanism* [DR14].

Definition 7 (Laplace mechanism LM)**.** *Mechanism* LM*, for function* $f : \mathfrak{D}^n \to \mathbb{R}$ *with sensitivity* Δf*, privacy parameter* ϵ*, and a data set* D*, releases*

$$f(D) + \text{Laplace}(\Delta f/\epsilon),$$

where $\text{Laplace}(b)$ *denotes a random variable from the Laplace distribution with scale* b *and density*

$$\text{Laplace}(x; b) = \frac{1}{2b} \exp\left(-\frac{|x|}{b}\right).$$

The Laplace mechanism is ϵ-DP [DR14, Theorem 3.6].

Mechanisms with Probabilistic Selection

The alternative to additive noise is probabilistic selection, which expands the application of differential privacy to functions with non-numerical output domain $\mathcal{R}$, or when the output is not robust to additive noise, e.g., auction bids [MT07] or the median [LLSY16].

The simplest selection mechanism is *randomized response* introduced by Warner [War65] and we describe it according to Dwork and Roth [DR14, Section 3.2]:

Definition 8 (Randomized Response)**.** Randomized response *handles sensitive survey questions with* yes *and* no *answers, i.e.,* $\mathcal{R} = \{yes, no\}$*, as follows:*

1. *Flip a coin.*
2. *If it comes up* tails*, answer truthfully.*
3. *If it comes up* heads*, flip again and answer* yes *if* heads *and* no *if* tails.

Randomized response pre-dates differential privacy but was shown to satisfy $\log_e(3)$-DP [DR14, Section 3.2]. It basically provides a form of *plausible deniability*, i.e., respondents can always claim that they did not answer truthfully ("my first flip showed *heads*"). Randomized response protects each individual respondent and yet allows inference over the population of all respondents as follows. Let f'_y denote the fraction of reported *yes* answers, and let f_y denote the fraction of respondents whose actual answer would be *yes*. The reported fraction of positive answers can be expressed as $f'_y = 1/4 + f_y/2$, i.e., answers due to *heads-heads* (occurs with probability $1/2 \cdot 1/2 = 1/4$) plus truthful answers due to *tails* (actual f_y times probability for *tails*). Thus, the actual fraction of *yes* answers, f_y, can be estimated as $2f'_y - 1/2$.

Randomized response can be generalized to arbitrary domains $\mathcal{R}$ as follows. Respondents report their true value $x \in \mathcal{R}$ with probability p and any other $z \in \mathcal{R}\backslash\{x\}$ with probability $q = 1 - p$.

Definition 9 (Generalized Randomized Response GRR)**.** *Mechanism* $\mathsf{GRR} : \mathcal{R} \to \mathcal{R}$, *applied on* $x \in \mathcal{R}$ *with privacy parameter* ϵ, *outputs* $r \in \mathcal{R}$ *with probability*

$$Pr[\mathsf{GRR}(x) = r] = \begin{cases} p = \frac{\exp(\epsilon)}{\exp(\epsilon)+|\mathcal{R}|-1} & if\, x = r \\ q = \frac{1}{\exp(\epsilon)+|\mathcal{R}|-1} & if\, x \neq r \end{cases}.$$

GRR is ϵ-DP as the quotient p/q is bounded by $\exp(\epsilon)$. Again, let f'_x denote the fraction of reports with value x. The actual fraction f_x is approximated as $f^*_x = \frac{f'_x - q}{p-q}$, which is an unbiased estimator, i.e., $\mathbb{E}[f^*_x] = f_x$ [WBLJ17, Theorem 1].

The *exponential mechanism*, introduced by McSherry and Talwar [MT07], additionally analyzes the data set to provide instance-specific selection probabilities per domain element. The mechanism is exponentially more likely to select "good" results quantified via utility function $u(D, r)$ which takes as input a data set $D \in \mathfrak{D}^n$, and a potential output $r \in \mathcal{R}$ from a fixed set of arbitrary outputs $\mathcal{R}$. Informally, higher utility means the output is more desirable and its selection probability is increased accordingly.

Definition 10 (Exponential Mechanism EM)**.** *The* exponential mechanism $\mathsf{EM}^{\epsilon}_{u}(D)$, *for any utility function* $u : (\mathfrak{D}^n \times \mathcal{R}) \to \mathbb{R}$ *and privacy parameter* ϵ, *outputs* $r \in \mathcal{R}$ *with probability proportional to* $\exp(\frac{\epsilon u(D,r)}{2\Delta u})$, *i.e.,*

$$Pr\left[\mathsf{EM}^{\epsilon}_{u}(D) = r\right] = \frac{\exp\left(\frac{\epsilon u(D,r)}{2\Delta u}\right)}{\sum_{r' \in \mathcal{R}} \exp\left(\frac{\epsilon u(D,r')}{2\Delta u}\right)}, \tag{2.1}$$

where

$$\Delta u = \max_{\forall r \in \mathcal{R}, D \simeq D'} |u(D, r) - u(D', r)|$$

is the sensitivity of the utility function.

The exponential mechanism is ϵ-DP [MT07, Theorem 6][7] and is *universal*, i.e., can implement any DP mechanism $\mathcal{M}$ by defining utility function $u(D, r)$ to be the logarithm of the probability density of $\mathcal{M}(D)$ at r [MT07, Section 3]. We write EM, i.e., omit u, ϵ, D, if they can be derived from the context. In this work, we consider the output domain $\mathcal{R}$ to be the data domain $\mathfrak{D}$ or a partition of it, e.g., subranges of $\mathfrak{D}$.

[7] The original definition [MT07, Definition 2] omits normalization term $2\Delta u$ from Definition 10 and is $2\Delta u\epsilon$-DP, which is equivalent to ϵ-DP with the normalization.

The *Gumbel mechanism* achieves the same output distribution as EM [DR19a, Lemma 4.2] by adding noise from the Gumbel distribution to the utility scores and selecting the output element with the largest noisy utility scores. In other words, by taking the arg max over noisy utility scores.

Definition 11 (Gumbel Mechanism GM)**.** *Mechanism* GM*, for utility function* $u : (\mathfrak{D}^n \times \mathcal{R}) \to \mathbb{R}$ *with sensitivity* $\Delta u = \max_{\forall r \in \mathcal{R}, D \simeq D'} |u(D,r) - u(D',r)|$*, outputs* $r \in \mathcal{R}$ *via*

$$\arg\max_{r \in \mathcal{R}} \{u(D,r) + \mathrm{Gumbel}(2\Delta u/\epsilon)\},$$

where $\mathrm{Gumbel}(b)$ *denotes a random variable from the Gumbel distribution with scale* b *and density*

$$\mathrm{Gumbel}(x; b) = \frac{1}{b}\exp\Big(-\Big(\frac{x}{b} + \exp\Big(-\frac{x}{b}\Big)\Big)\Big).$$

The Gumbel mechanism is also known as the *Gumbel-(soft)max trick* [Gum48, MTM14]. It is originally found in machine learning literature to efficiently compute Equation (2.1) – also called the softmax function which maps arbitrary inputs to probabilities – and only recently applied in DP literature [DR19b].

Report One-Sided Noisy Argmax [DR14, Section 3.4] achieves similar guarantees as EM [DR14, Remark 3.13][8] by selecting outputs based on the argmax over utility scores with additive noise from the exponential distribution[9]. The exponential distribution with scale b is defined as

$$\mathrm{Expon}(x; b) = \frac{1}{b}\exp\Big(-\frac{x}{b}\Big)$$

for $x > 0$ and 0 elsewhere.

Inverse Transform Sampling

After computing the selection probabilities according to the exponential mechanism EM, we need to sample an output based on these probabilities, which we realize with *inverse transform sampling*. Inverse transform sampling uses the uniform distribution, where all values are equally likely, to simulate any distribution based on its cumulative distribution function $F(x)$. Specifically for EM,

$$F(x) = \Pr\big[\mathrm{EM}_u^\epsilon(D) \le x\big] = \frac{\sum_{r \in \mathcal{R}, r \le x} \exp\Big(\frac{\epsilon u(D,r)}{2\Delta u}\Big)}{\sum_{r' \in \mathcal{R}} \exp\Big(\frac{\epsilon u(D,r')}{2\Delta u}\Big)}.$$

Inverse transform sampling, given F and a uniform random $s \in (0,1]$, finds the first output x such that $s < F(x)$ and outputs it [EKM+14, Section 3], [AMFD12, Section 5.2]. To illustrate why this works, consider the following example. Let $\mathfrak{D} = \{a, b\}$ with selection probabilities 0.7 for a and 0.3 for b. Now, we take a large array A and fill 70% of A with a and the rest with b, sample an index i of A at uniform random, and output $A[i]$. The output is a with 70% probability and b otherwise.

[8] Some sources state that report one-sided noisy argmax and EM have the *same* distribution (e.g., [BGG+16, Section 5] cites older version of [DR14, Theorem 3.13]) however, recent work shows the distributions are different [DKS+21].

[9] The name of the mechanism is due to the fact that the exponential distribution is also called one-sided Laplace distribution [BGG+16].

2.2.5 Distributed Noise Generation

Sampling noise via secure computation is inefficient for distributed semi-honest parties, as they have to securely evaluate the inverse cumulative density function requiring complex operations such as the logarithm. For example, sampling Laplace(b) is equivalent to computing $(-1)^s b \log(r)$ given uniform random numbers $r \in (0,1], s \in \{0,1\}$ [JWEG18, Supplementary Material]. Distributed noise generation is more efficient, i.e., each party locally computes partial noises, which are securely combined. Such noise generation is commonly found in the DP literature [ÁC11, DKM⁺06, HLK⁺17, GX17]. Distributed noise generation is straightforward for distributions that are *infinitely divisible*, i.e., samples can be expressed as the sum of independent and identically distributed random variables. Next, we describe distributed noise for the Laplace, exponential and Gumbel distribution, which are infinitely divisible [BS13, GX17].

Distributed Laplace Noise

Random variable Laplace(b) can be expressed via the gamma distribution [KKP12, Table 2.3], i.e.,

$$\sum_{j=1}^{n} \left(Y_j^1 - Y_j^2 \right), \quad Y_j^1, Y_j^2 \sim \text{Gamma}\left(\frac{1}{n}, b \right),$$

where the gamma distribution with shape k and scale b has density

$$\text{Gamma}(x; k, b) = \frac{1}{\Gamma(k) b^k} x^{k-1} \exp\left(-\frac{x}{b} \right).$$

To avoid floating-point numbers, which lead to larger overhead for secure computation compared to integers [ABZS13] and can lead to privacy violations with limited machine precision (see Section 3.4), one can use the discrete Laplace distribution defined over integers. The discrete Laplace distribution is infinitely divisible and samples can be expressed as the difference of two Pólya random variables [GX17].

Distributed Exponential Noise

Random variable Expon(b) can be expressed similarly to distributed Laplace noise[10] as

$$\sum_{j=1}^{n} Y_j, \quad Y_j \sim \text{Gamma}\left(\frac{1}{n}, b \right).$$

Distributed Gumbel Noise

Random variable Gumbel(b) can be expressed via the exponential distribution [BS13], i.e.,

$$b \cdot \lim_{n \to \infty} \left\{ \sum_{j=1}^{n} \frac{Y_j}{j} - \log(n) \right\}, \quad Y_j \sim \text{Expon}(1).$$

[10] The Laplace distribution is also called the double exponential distribution. Laplace-distributed random variables can be written as the difference of two random variables from the exponential distribution [KKP12, Table 2.3]; hence, the similarity to distributed Laplace noise.

While the Laplace distribution can be expressed as a *finite* sum of random variables, the Gumbel distribution requires an *infinite* sum. However, the expected approximation error for the Gumbel distribution can be made arbitrarily small in the number s of summands:

Theorem 1. *For* $\mathrm{Gumbel}_{|s}(b) = b\sum_{j=1}^{s}\frac{Y_j}{j} - b\log(n)$*, where* $Y_j \sim \mathrm{Expon}(1)$*, we have* expected approximation error $|\mathrm{Gumbel}(b) - \mathrm{Gumbel}_{|s}(b)| = O(b/s)$.

Proof. We have $\mathbb{E}[\mathrm{Gumbel}(b)] = \gamma_{\mathrm{EM}} \cdot b$, where $\gamma_{\mathrm{EM}} = \lim_{n\to\infty}\left(\sum_{i=1}^{n} 1/i - \log_e(n)\right) \approx 0.5772$ is the Euler-Mascheroni constant. Furthermore, $\mathbb{E}\left[\mathrm{Gumbel}_{|s}(b)\right] = b\sum_{j=1}^{s}\frac{\mathbb{E}[Y_j]}{j} - b\log(n) \leq b\left(\gamma_{\mathrm{EM}} + 1/(2s) + O\left(1/s^2\right)\right)$, due to $\mathbb{E}[Y_i] = 1$ and an inequality for the difference of the n-th harmonic number and $\log n$ [GK07, Eq. (4.30)]. Altogether,

$$\begin{aligned}\left|\mathbb{E}\left[\mathrm{Gumbel}(b) - \mathrm{Gumbel}_{|s}(b)\right]\right| &= \left|\mathbb{E}[\mathrm{Gumbel}(b)] - \mathbb{E}\left[\mathrm{Gumbel}_{|s}(b)\right]\right| \\ &\leq |\gamma_{\mathrm{EM}} b - b(\gamma_{\mathrm{EM}} + O(1/s))| = bO(1/s).\end{aligned}$$

$\square$

3 Related Work

In Section 3.1, we describe existing privacy models for implementing DP – mainly, the local, central, and MPC model – and focus on the MPC model in Section 3.2. We discuss techniques to prune the data or reduce the domain to improve efficiency in Section 3.3. Finite machine precision, resulting privacy violations, and mitigations are summarized in Section 3.4. Decomposability in the context of DP is discussed in Section 3.5. Finally, we discuss related work grouped by privacy models: DP median in Section 3.6 and DP heavy hitters in Section 3.7.

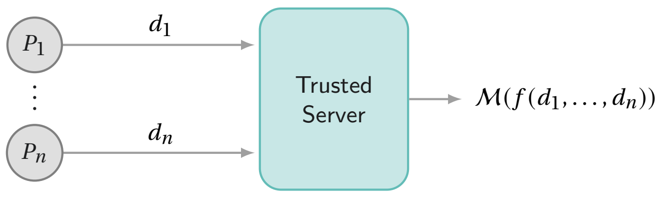
3.1 Privacy Models

Differentially private mechanisms can be implemented in different models, visualized in Figure 3.1 [BK20b, Böh21]. Next, we describe the models, their trade-offs with regards to accuracy and privacy, and how MPC simultaneously supports high accuracy with strong privacy.

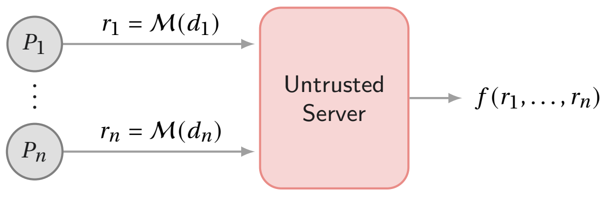
In the *central model* (Figure 3.1a) every client sends their unprotected data to a trusted, central server which runs mechanism $\mathcal{M}$ on the clear data. The central model provides the highest accuracy as the randomization inherent to DP algorithms, is only applied once. In the *local model* (Figure 3.1b), introduced by Kasiviswanathan et al. [KLN⁺11], clients apply $\mathcal{M}$ locally and send anonymized values to an untrusted server for aggregation. The accuracy is limited as the randomization is applied multiple times. Hence, it requires a very large number of users to achieve accuracy comparable to the central model [BEM⁺17, CSU⁺19, HKR12, KLN⁺11, MMP⁺10]. Specifically, an exponential separation exists between the local and central model regarding the accuracy and sample complexity [KLN⁺11]. Recently, an intermediate *shuffle model* (Figure 3.1c) was introduced by Bittau et al. [BEM⁺17][1]: A trusted party is added between client and server in the local model, the shuffler, who does not collude with anyone. The shuffler permutes and forwards the randomized client values. The permutation breaks the mapping between a client and her value, which reduces randomization requirements. The accuracy of the shuffle model lies between the local and central model, however, in general it is strictly weaker than the central model [BC20, CSU⁺19]. To illustrate this separation, consider counting queries with n parties: the central model has accuracy $O(1)$, the shuffle model (with a single message per party) has accuracy $O(\log n)$, and the local model suffers error $O(\sqrt{n})$ [WHMM21]. In addition, Cheu et al. [CU21] showed that in the shuffle model no general analogue exists for the exponential mechanism, which is the basis for most of our protocols.

As our goal is high accuracy without trusted parties even for a small number of users, we implement the *MPC model* (Figure 3.1d). In other words, we simulate the central model in a distributed setting with cryptographic tools, as commonly found in the DP literature [AMFD12, DKM⁺06, EKM⁺14, GX17, PL15, RN10, TKZ16]. General-purpose MPC incurs high computation and communication overhead, reducing efficiency and scalability [CSU⁺19]. However, MPC combines

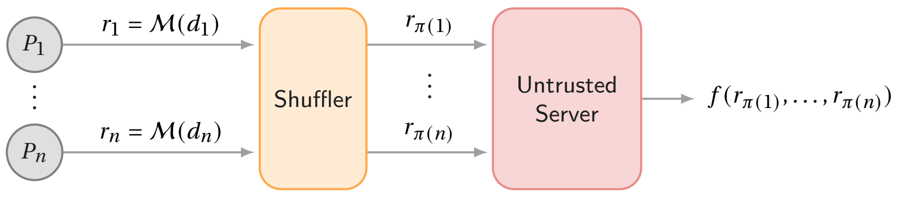
[1] Bittau et al. [BEM⁺17] introduced the shuffle model for differentially private software monitoring, and Cheu et al. [CSU⁺19] initiated the analytical study of the shuffle model. In the context of cryptography, a similar model was introduced by Ishai et al. [IKOS06], utilizing anonymous communication as a building block for MPC.



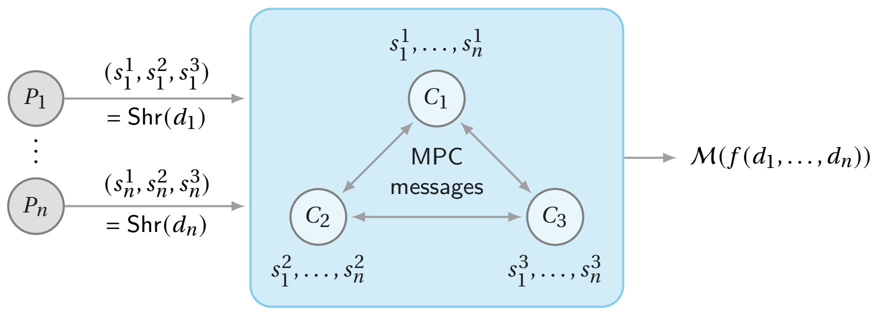
(a) Central Model



(b) Local Model



(c) Shuffle Model with permutation π



(d) MPC Model with 3 Computation Parties C_i

Figure 3.1: Implementation models for DP mechanism $\mathcal{M}$. Party P_i sends a message – raw d_i or randomized data r_i – to a server, who combines all messages with function f. In the MPC model, party P_i distributes secret shares $\mathsf{Shr}(d_i)$ among m computation parties.

the respective benefits of the models, namely, high accuracy and strong privacy, i.e., no disclosure of values to a third party, and we implement efficient and scalable special-purpose MPC protocols.

3.2 MPC and DP

Dwork et al. [DKM+06] first mentioned that differential privacy combines well with secure computation and many works build upon this observation [DKM+06, TKZ16, RN10, PL15, EKM+14, NPR19]. Secure DP summation is easily achieved via additive noise, see, e.g., [GX17] for a survey of such work. Goyal et al. [GKM+16] showed that in general distributed DP protocols can only achieve optimal accuracy when combined with secure computation.

Recently, cryptographic primitives received more attention in the DP literature to improve the shuffle model [GKMP20, BBGN20]. Mainly, these improvements build upon the work of Ishai et al. [IKOS06], which uses anonymous communication as a building block for MPC. Also, DP is used in cryptographic protocols to bound their *leakage* such as revealing some access patterns to improve overall efficiency [CCMS19]. It is also used, e.g., in private record matching to securely compute and release exact matches while protecting the number of non-matching records by adding a certain number of dummy records [HMFS17]. An overview of cryptographic applications of DP is given by Wagh et al. [WHMM21]. In this work, we focus on efficient secure computation of DP statistics with high accuracy, especially on small data sets.

MPC of the Exponential Mechanism

Our protocols are mainly realized as secure computations of the exponential mechanism for the median, decomposable aggregate utility functions, and heavy hitters. Alhadidi et al. [AMFD12] design a secure two-party protocol for the exponential mechanism restricted to "max utility functions", where each party reports their maximum value for a generalized class (e.g., subrange) which are added together [AMFD12, Section 5.1]. They use garbled circuits to compute the maximum and comparisons, and oblivious polynomial evaluation [NP01] as well as secret sharing like Bunn and Ostrovsky [BO07] to approximate the Taylor series for the exponential function (i.e., $\exp(x) = \sum_{n=0}^{\infty} x^n/n!$). They only estimate the running time of their design [AMFD12, Section 7.2], whereas we implement all our protocols and provide evaluations in real-world networks (Section 4.4.1). Also, we avoid complex approximations of the exponential function by leveraging data-independent utility functions ($\mathsf{EM}_{\mathsf{med}}$), decomposable utility functions (EM^*), and the Gumbel mechanism (GM^*, PEM). Furthermore, our protocol EM^* is more general as we support more than two parties and a broader class of utility functions, including but not limited to max utility functions. Eigner et al. [EKM+14] present and implement a carefully designed secure exponential mechanism in the multi-party setting. While their work is more general than ours, i.e., supporting arbitrary utility functions and also malicious parties, they are linear in the size of the domain, and securely compute the exponential function. Our protocols $\mathsf{EM}_{\mathsf{med}}$, EM^*, PEM are sublinear in the domain size without costly secure exponentiation. Secure exponentiation is complex [ABZS13, AS19, DFK+06, Kam15], requiring many interactive rounds. The semi-honest protocol of Eigner et al. requires 42 seconds in a LAN (on an Intel i5 3.20 GHz, 16 GB RAM machine) to select an output from a very small domain of only 5 elements whereas EM^* handles at least 10^5 domain elements in the same time (on AWS t2.medium instances with 4vCPUs, 2 GB RAM), i.e., an improvement of at least 5 orders of magnitude (see Section 6.3.1).

3.3 Data Pruning & Domain Reduction

Efficiently sampling the distribution defined by the exponential mechanism is non-trivial [DR14, Section 3.4], thus, a reduction of the sampling space is considered by related work [BDB16, GLM+10, LLSY16, PL15].

Data Pruning for DP Median

Pettai and Laud [PL15] define MPC protocols for differentially private analytics, including the median (detailed in 3.6.2). In case of filtering (i.e., predicate matching), Pettai and Laud [PL15] apply a form of input pruning and replace half of the excluded values with a small (resp. large) constant. They mention that this does not always preserve the median, unlike the pruning approach by Aggarwal et al. [AMP10] implemented in $\mathsf{EM}_{\mathsf{med}}$.

Domain Reduction for DP Median

Gupta et al. [GLM+10] suggest reducing the output domain for combinatorial problems from exponential to polynomial size and sample from the reduced set with the exponential mechanism. Blocki et al. [BDB16] follow this suggestion and use a relaxed exponential mechanism to sample a DP password frequency list in the central model. They allow a negligible error δ, i.e., they only sample the exponential mechanism correctly with probability $1 - \delta$, which improves sampling from (potentially) exponential time to $O(|D|^{1.5}/\epsilon)$. However, they require full access to the data D in clear. Li et al. [LLSY16] suggest to divide domain $\mathfrak{D}$ into equal-sized ranges, select a range with the exponential mechanism and sample an element in the range at uniform random. However, with this approach any element in the selected range is equally likely to be output independent of its utility. Our two-party protocol $\mathsf{EM}_{\mathsf{med}}$, on the other hand, samples the median only among elements with the *same* utility. Our multi-party protocol EM^* splits the domain as suggested by Li et al. [LLSY16], however, we divide ranges iteratively: We select the best range, divide the selected range into subranges and repeat the selection until the subrange contains only one element. Our multi-party protocol PEM iteratively finds frequent bit-prefixes of increasing size. Furthermore, our protocols provide a parameterized trade-off between accuracy and running time for the exponential mechanism.

Domain Reduction for DP Heavy Hitters

Domain reduction and efficient encoding (e.g., hashing, sketching) are the main challenges of heavy hitter discovery, where the domain is either large or unknown, and discussed in detail in Section 3.7.

3.4 Limited Machine Precision and Privacy Violations

In general, DP mechanisms operate on reals, whereas actual implementations are limited to the precision of physical machines. However, limited precision can lead to privacy violations. Mironov [Mir12] showed that the Laplace mechanism is vulnerable to finite precision, as the set of possible outcomes can differ between neighboring data sets due to irregularities of floating-point implementations. The proposed mitigation is to perform “snapping”, roughly, clamping the noisy result to a fixed range and ensuring evenly spaced outputs. For a detailed implementation description see [Mic20a]. Gazeau et al. [GMP16] consider general finite precision semantics and suggest using fixed precision (via rounding, truncation) for bounded privacy degradation. Recent work by Ilvento [Ilv20] extends this line of study to the exponential mechanism, which is also vulnerable to finite precision. The suggested mitigation is switching from base e to base 2 to perform precise arithmetic, e.g., for integer-valued utility functions one approximates ϵ as $\epsilon' = -\log_2(x/2^y)$ for integers x, y such that $x/2^y \leq 1$.

The investigation of privacy violations due to limited machine precision is outside the scope of this work. However, our protocols do not rely on noise or utility scores represented as floating-point numbers: Our protocol $\mathsf{EM}_{\mathsf{med}}$ (Section 5) uses fixed point numbers, i.e., scaled and truncated integers, instead of floating point numbers. Interestingly, the mitigation techniques from Ilvento [Ilv20] are similar to the optimizations deployed in our protocol EM^* (Section 6), i.e., we utilize base-2 and integer utility functions for efficiency (detailed in Section 6.2.6). Our protocols

PEM, HH (Section 7) use scaled, truncated noise from a continuous distribution (PEM) or can be realized with noise from a discrete distribution defined over the integers (HH).

3.5 Decomposability

Decomposability is often found in the context of MapReduce, which is a programming paradigm for distributed data aggregation: Roughly, a mapper produces intermediary results (e.g., partial sums) that a reducer combines into a result (e.g., total sum). Airavat [RSK+10] is a Hadoop-based MapReduce programming platform for DP statistics based on additive noise (Laplace mechanism) with an untrusted mapper but trusted reducer. We consider decomposable utility functions for probabilistic selection via the exponential mechanism without any trusted parties. Existing secure exponential mechanisms [AMFD12, EKM+14] use decomposable utility functions (max and counts), but neither classify nor provide optimizations for such functions. Blocki et al. [BDB16] minimize cumulative error for DP password frequency lists. They use (decomposability of) frequencies for their dynamic programming, which has access to all the data in the clear. We, on the other hand, use decomposable aggregate functions to efficiently and securely combine distributed inputs in EM^*.

3.6 DP Median

In the context of the DP median, we first detail different sensitivity notions and their accuracy in Section 3.6.1. Then, we discuss related work for the DP median grouped by privacy models in Section 3.6.2.

3.6.1 Sensitivity and Utility Functions for DP Median

Recall Definition 6, i.e., the sensitivity is the maximum difference of a function evaluated on neighboring data sets. Different sensitivity notions exist, and we discuss those relevant for the median next.

Sensitivity of the Median

According to Definition 6, the sensitivity of the median is $\max \mathfrak{D} - \min \mathfrak{D}$[2]. Definition 6 is also called *global sensitivity* as it considers *all* possible data sets and their neighbors. Note that considering only a *fixed* data set instance and its neighbors, known as *local sensitivity*, violates differential privacy [NRS07, Section 2.1][3].

Smooth sensitivity, developed by Nissim et al. [NRS07], satisfies DP as a smooth upper bound on the local sensitivities of all neighbors of a fixed data set instance. Smooth sensitivity is not always computable [NRS07, Section 1] but provides instance-specific randomization, typically

[2] To illustrate, consider a sorted data set $D_1 = \{x, x, y, y\}$ where $x, y \in \mathfrak{D}$ and the median is x. However, y can be the median after a single change (i.e., remove any x). The difference $|x - y|$ is maximized for $x = \min \mathfrak{D}$, $y = \max \mathfrak{D}$, hence, the median sensitivity stated above.

[3] For example, consider $D_2 = \{x, y, y\}$ with $x = \min \mathfrak{D}$, $y = \max \mathfrak{D}$ and median $y = d_{\lceil n/2 \rceil}$. D_2 is a neighbor of D_1 (remove x from D_1), and $D_3 = \{x, y, y, y\}$ (add y to D_2). The local sensitivity of D_2 is the same as the global sensitivity as median y can become x after a single change (add x). However, the local sensitivity of D_3 is 0, i.e., y remains the median after one addition/removal. Recall that sensitivity, alongside the privacy parameter ϵ, governs the noise magnitude for DP mechanisms. Now, D_3 is a neighbor of D_2 but the large difference in the local sensitivity, thus, altered additive noise, suffices to distinguish if D_2 or D_3 was an input to $\mathcal{M}$ which violates differential privacy [NRS07, Section 1.3].

smaller than global sensitivity. For the median, Nissim et al. [NRS07, Proposition 3.4] define the smooth sensitivity as follows.

Definition 12 (Smooth Sensitivity of the Median)**.** *The smooth sensitivity of the median of sorted data set $D = \{d_1, \dots, d_n\} \in \mathfrak{D}^n$ is*

$$\max_{k=0,..,n} e^{-k\epsilon} \max_{t=0,..,k+1} \left(d_{\frac{n}{2}+t} - d_{\frac{n}{2}+t-k-1} \right),$$

where $d_i = \min \mathfrak{D}$ for $i < 1$ and $d_j = \max \mathfrak{D}$ for $j > n$.

Informally, sensitivity of neighboring data sets "further away" (i.e., increasing k) from fixed instance D receive exponentially less weight ($e^{-k\epsilon}$). While smooth sensitivity can be smaller than global sensitivity, computing smooth sensitivity requires access to the entire data set, otherwise the error increases further[4], which prohibits efficient (secure) computation with high accuracy.

Smooth sensitivity is used in DP mechanisms based on additive noise. However, for probabilistic selection via the exponential mechanism a smaller and *data-independent* sensitivity can be defined for the median, as described next.

Utility Function for the Median

Li et al. [LLSY16, Section 2.4.3] note that the Laplace mechanism is ineffective for the median, as (smooth) sensitivity can be high, and present a low-sensitivity utility function for the exponential mechanism. They quantify an element's utility via its *rank* relative to the median. The rank of $x \in \mathfrak{D}$ in a data set D is the number of values in D smaller than x. More formally, $\mathsf{rank}_D(x) = |\{d \mid d \in D : d < x\}|$. Note that for the median, we have $\mathcal{R} = \mathfrak{D}$, which means every domain element is a potential output.

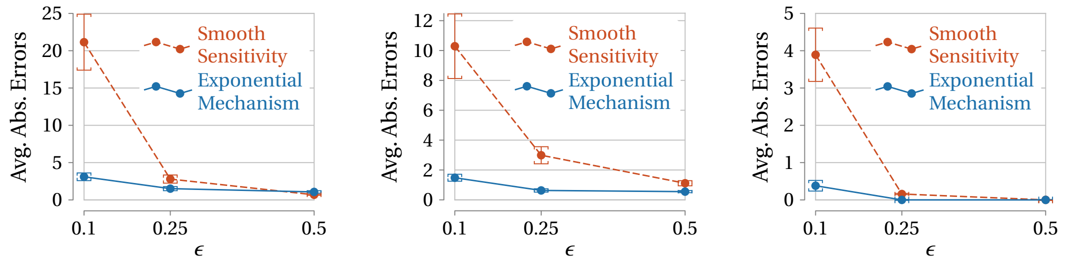
Definition 13 (Median Utility Function)**.** *The* median utility function $u_\mu : (\mathfrak{D}^n \times \mathfrak{D}) \to \mathbb{Z}$ *gives a utility score for each $x \in \mathfrak{D}$ with regard to $D \in \mathfrak{D}^n$ as*

$$u_\mu(D, x) = - \min_{\mathsf{rank}_D(x) \le j \le \mathsf{rank}_D(x+1)} \left| j - \frac{n}{2} \right|.$$

The sensitivity of u_μ is only $1/2$ since adding an element increases $n/2$ by $1/2$ and j either increases by 1 or remains the same [LLSY16, Section 2.4.3][5]. Thus, the denominator $2\Delta u$ in the exponents of Equation (2.1) in Definition 10 equals 1, and we will omit it in the definitions of our DP median protocols. We focus on MPC of the DP median but Definition 13 supports any k^{th}-ranked element by replacing $n/2$ with k and adapting the sensitivity accordingly [Mic20b].

[4] Smooth sensitivity approximations exist that provide a factor of 2 approximation in linear-time, or an additive error of $\max(\mathfrak{D})/\mathrm{poly}(|D|)$ in sublinear-time [NRS07, Section 3.1.1]. Note that this error e is with regard to smooth sensitivity s, and the additive noise is even larger as it scales with $(s + e)/\epsilon$.

[5] Here, we point out a technicality, which is a moot point for even data sizes or if the median appears multiple times in the data. Li et al. [LLSY16] approximate the median position as $n/2$ to get low sensitivity $1/2$. We defined the median position as $\lceil n/2 \rceil$, which is the same for even data sizes. However, one cannot use a rounded median position in Definition 13, as it increases the sensitivity to 1, i.e., rounded positions do not change between neighbors but the ranks can change by ± 1. Overall, the median (at position $\lceil n/2 \rceil$) and an adjacent element might receive the same utility score. In this case, one outputs either one of those data elements (resp., domain elements in between) with the same probability. In expectation, one outputs the average of them, similar to the common median definition which returns the average $(d_m + d_{m+1})/2$ for $m = n/2$ and even n. However, this technicality is of little to no consequence when the elements adjacent to the median are very similar or the same – as is to be expected. Recall, the median represents a "typcial" element in the data. The median is considered a robust statistic, i.e., few input changes (resp., small positional shifts) do not lead to large output changes [DL09, Section 1.2].



(a) Credit card transactions, first 100 000 payment records in Cents [ULB18].

(b) Walmart supply chain data, ≈ 175 000 shipment weights as integers [Kag18].

(c) NYC taxi trips, first 75 000 fares in Cents [TLC19].

Figure 3.2: Absolute errors with 95% confidence intervals, averaged for 100 differentially private median computations via Laplace mechanism with smooth sensitivity and the exponential mechanism.

Exponential Mechanism is more accurate than Smooth Sensitivity

To illustrate that additive noise can be high for DP median, we empirically evaluate the absolute error of the Laplace mechanism with smooth sensitivity and the exponential mechanism in Figure 3.2 on real-world data sets [Kag18, ULB18, TLC19]. For low ϵ, corresponding to a stronger privacy protection, the exponential mechanism is more accurate. Note that we evaluated an idealized version of smooth sensitivity by ignoring required constants that further increase the noise magnitude [NRS07, Lemma 2.9] [MG20, Proposition 2], and still see better accuracy for the exponential mechanism. Medina and Gillenwater [MG20] also compared the exponential mechanism to smooth sensitivity for DP median and found the former to be superior as well.

Overall, our protocols $\mathsf{EM}_{\mathsf{med}}$, EM^* achieve better accuracy, i.e., average absolute error, for DP median than approaches without the exponential mechanism for low ϵ with better scalability than the standard exponential mechanism. We discuss accuracy bounds of $\mathsf{EM}_{\mathsf{med}}$ in Section 5.1.6 and provide empirical evaluations in Section 5.3. For EM^*, we discuss accuracy in Section 6.1.5 and detail empirical evaluations with regard to related work in Section 6.3.5.

3.6.2 DP Median and Privacy Models

In the following, we discuss related work for DP median grouped by privacy models, i.e., non-private, local DP, central DP, and MPC of DP.

Non-private Median

The exact median can be computed by general MPC, which offers input secrecy but does not provide any output privacy. Aggarwal et al. [AMP10] present very efficient secure protocols for finding the median of two (resp., multiple) parties requiring only a logarithmic number of secure comparisons in the size of the data (resp., domain). Their protocols iteratively prune the data (resp., domain) until only the median remains and operate similar to binary search. In each iteration, their two-party protocol securely compares local medians and lets each party discard half of their sorted data that cannot contain their mutual median. We formalize their two-party protocol in Section 5.1.5 and detail their multi-party protocol in Section 5.2.7. While Aggarwal et al. compute the exact median we compute the DP median. For large data sets, EM^* employs the pruning from Aggarwal et al. to reduce the input size until it is sublinear in the domain size, so we can efficiently sample the DP median from the pruned input.

Local DP Median

Smith et al. [STU17] and Gaboardi et al. [WGSX20] consider the restrictive non-interactive local model, where at most one message is sent from client to server, and achieve optimal local model error. However, local DP requires more samples to achieve the same accuracy as central DP for the same privacy parameter and no non-interactive LDP protocol [STU17, WGSX20] can achieve asymptotically better sample complexity than $O(\epsilon^{-2}\alpha^{-2})$ for error α [DJW13]. We, on the other hand, are interested in high accuracy as in the central model even for small sample sizes

Central DP Median

Dwork and Lei [DL09] present DP mechanisms for robust statistics where data samples are assumed to be drawn independent and identically distributed. Robust statistics (e.g., median) are not very sensitive to outliers (unlike, e.g., the mean) and small input changes do not drastically alter the result [DL09, Section 1.2]. They describe a *propose-test-release* paradigm, where an analyst (without data access) first proposes a bound on the local sensitivity to the data owner. Then, the data owner performs a differentially private test on the data to check if this bound suffices. Finally, if the test succeeds, the DP statistic is released with the proposed bound. Their DP median approach is the first that does not require a bounded data domain. However, it aborts if the data are not from a “nice” distribution, e.g., the local sensitivity is high. Their DP median approach is based on a private estimation s of the data scale (also called dispersion) via the inter-quartile range. However, their noise magnitude $sn^{-1/3}$ for the median can be prohibitively large, especially for small data sizes n.

Smooth sensitivity, introduced by Nissim et al. [NRS07], is a smooth upper bound on local sensitivity. Smooth sensitivity analyzes the data to provide ideally small instance-specific additive noise. As discussed in Section 3.6.1, the exponential mechanism provides better accuracy for low ϵ and we provide efficient computations over a data subset ($\mathsf{EM}_{\mathsf{med}}$) or domain subranges ($\mathsf{EM}^*$), whereas computing smooth sensitivity for the median requires access to the entire sorted data.

MPC DP Median

As mentioned before, Pettai and Laud [PL15] securely compute DP statistics, including the DP median. Their implementation is based on secret sharing with 3 parties and realizes the sample-and-aggregate mechanism [NRS07, Section 4]. Typically, the sample-and-aggregate mechanism partitions the data in multiple equal-sized subsets, performs a computation per subset, and aggregates the results to provide a noisy approximation. For the median, however, Pettai and Laud [PL15] compute the noisy average of the 100 values closest to the median within a clipping range. This approach provides limited accuracy, especially, if the data contains outliers or large gaps (see discussion in Section 6.1.5 and evaluation in Section 6.3.5). The exponential mechanism, which we securely implement for the median utility function, selects an actual domain element and not a noisy approximation.

Cryptϵ [CWH+20] employs two non-colluding untrusted servers and homomorphic encryption [Pai99] as well as garbled circuits to compute noisy histograms (Laplace mechanism) for SQL queries (e.g., count, distinct count, counts grouped by matching attributes) which can be extended to compute the median. However, computing DP median with probabilistic selection is more accurate than additive noise for low ϵ (Section 3.6.1 and Section 6.3.5). Cryptϵ has a running time linear in the data size. For a data set of one million records, Cryptϵ requires around

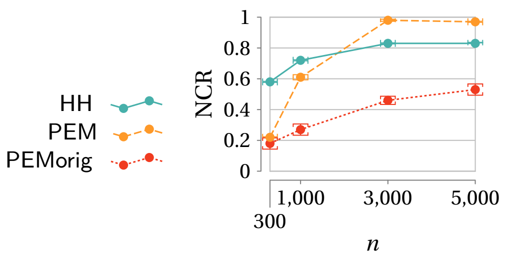
DP median methods	*Error bound α of (α, β)-accuracy*
Nissim et al. [NRS07] – Additive noise with *Smooth Sensitivity*: reduced, instance-specific noise.	$\max_{k=0,..,n} e^{-k\epsilon} \max_{t=0,..,k+1} \left(d_{\frac{n}{2}+t} - d_{\frac{n}{2}+t-k-1}\right)\gamma$
Dwork and Lei [DL09] – Additive noise with *Propose-Test-Release*: propose bound on sensitivity, privately test if it is sufficient, and release noisy result if test succeeded.	$\frac{d_{\lceil 0.75n \rceil} - d_{\lceil 0.25n \rceil}}{n^{1/3}}\gamma$
Pettai and Laud [PL15] – Additive noise with *Sample-and-Aggregate*: average j elements closest to the median in clipping range $[c_l, c_u]$, release noisy average.	$\left(\frac{c_u - c_l}{j} + \frac{\max(\mathfrak{D}) - \min(\mathfrak{D})}{\epsilon \exp(\Omega(\epsilon\sqrt{j}))}\right)\gamma$
This work (Section 6.1.5) – Probabilistic selection with *Exponential Mechanism*: iteratively select subranges containing the median.	$\max_{i \in \{+1,-1\} \cdot \left\lfloor \frac{\ln(\lvert\mathfrak{D}\rvert/\beta)}{\epsilon} \right\rfloor} \left\lvert d_{\frac{n}{2}+i} - d_{\frac{n}{2}} \right\rvert$

Table 3.1: DP median methods in the central model with $\gamma = \ln(1/\beta)/\epsilon$. Data $D \in \mathfrak{D}^n$ is sorted and the error terms are simplified to ease comparison: omitting small constants (mainly δ) [NRS07, DL09], assuming expected sensitivity [DL09], shortened approximation error term [PL15] (see [NRS07, Th. 4.2]), and applying one selection step for this work.

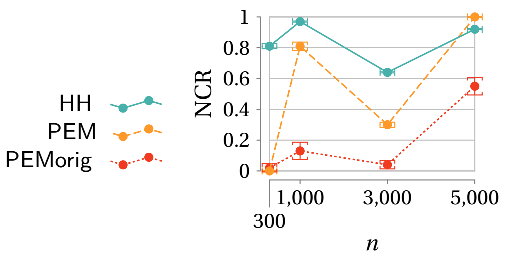
17 minutes for a count with three conditions, e.g., “count of male employees of Mexico having age 30”, with pre-computed DP index for country using Google Cloud c2-standard-8 instances with 8vCPUs and 32 GB RAM [CWH+20, Section 9]. The pre-computed DP index, which consumes part of the privacy budget, approximately shows the location where sorted encrypted values change from one value to another to speed up processing. Without such pre-computation the runtime increases to hours. Similarly, computing a noisy histogram for the attribute age in the form of the cummulative density function over integer domain $[1, 100]$ requires around half an hour for around 32,000 records without pre-computation [CWH+20, Table 3]. Our protocols, on the other hand, can process data in real-time, i.e., without large pre-computation overhead for new data, and our evaluations cover the entire protocol running time. Our DP median protocol $\mathsf{EM}_{\mathsf{med}}$ is sublinear in the data domain with pruning (linear in the data size without pruning) and runs in less than 7 seconds for one million records in a WAN with 100 ms latency and 100 Mbits/s bandwidth on machines with only 2 GB RAM and 4vCPUs (Figure 5.6 in Section 5.3). Our protocol EM^* is independent of the data size and optimized for decomposable functions. GM^* (a variation of EM^*) achieves a running time of less than 90 seconds for millions of domain elements with the same hardware and real-world WAN (Figure 6.4 in Section 6.3).

Theoretical Accuracy Bounds

Table 3.1 lists theoretical accuracy bounds for related work closest to ours, i.e., computation of the DP median in the central or MPC model, omitting any dependence on δ. The table compares the (α, β)-accuracy, i.e., the probability that the absolute error is at most α is at least $1-\beta$ (formalized in Definition 15 in Section 4.3). Note that the table entries, except for this work, are the sensitivity of the method multiplied by factor $\gamma = \ln(1/\beta)/\epsilon$ with an additional error term for Pettai and Laud [PL15]. Related works draw additive noise r from zero-centered Laplace distribution with scale s/ϵ for sensitivity s (Laplace mechanism, Definition 7). Since $\Pr[|r| \leq t \cdot s/\epsilon] = 1 - \exp(-t)$ [DR14, Fact 3.7], we can bound the absolute error $|r|$ as in Table 3.1 by setting $\beta = \exp(-t)$, $\gamma = t/\epsilon = \ln(1/\beta)/\epsilon$. As the theoretical guarantees show strong data dependence, which hinder straightforward comparisons (as discussed in Section 6.1.5), we also provide empirical accuracy comparisons in Section 6.3.5.



(a) Zipf



(b) Retail prices [ULB19]

Figure 3.3: Accuracy (NCR) of our MPC protocols PEM and HH compared to LDP protocol PEMorig [WLJ19] for parameters $k = 8$, $|\mathfrak{D}| = 2^{32}$, $\epsilon = 2$ with $n \in \{300, 1\,000, 3\,000, 5\,000\}$.

3.7 DP Heavy Hitters

First, we briefly illustrate that our protocols provide better accuracy than existing local model approaches. Then, we discuss approaches for heavy hitter discovery without privacy protection, with local DP, central DP, and MPC of DP.

Accuracy for DP Heavy Hitters

We want to illustrate that the MPC model provides better accuracy than the local model for heavy hitters. For this purpose, we compare our MPC protocols PEM and HH with a state-of-the-art local model approach from Wang et al. [WLJ19] which we denote PEMorig. We measure top-k accuracy like Wang et al. via non-cumulative rank (NCR), which is similar to the F1 score weighted by an element's rank, where the most frequent value has rank k, the second most frequent rank $k-1$, etc. (formalized in Definition 16 in Section 4.3). We used synthetic data from the same Zipf distribution as Wang et al.[6] as well as a real-world Online retail data set [ULB19]. Figure 3.3 illustrates that our protocols provide higher accuracy than PEMorig which in turn provides better accuracy compared to other LDP approaches [WLJ19]. The accuracy drop in the real-world data set for $n = 3,000$ (Figure 3.3b) is mainly due to an increase in the number of distinct data elements, which decreases the relative frequency of heavy hitters. We informally describe PEMorig later in this section and formalize it in Section 7.1.2. A more detailed evaluation (with varying k, ϵ) is provided in Section 7.3.

Non-private Heavy Hitters

Algorithms for heavy hitter detection are roughly grouped into three classes [CH10, ABL⁺17]: *Quantile algorithms*, which use estimated quantiles of range endpoints to approximate frequencies of range elements; *hash-based sketches*, which provide a space-efficient frequency estimation, and *counter-based sketches*, where a set of counters are updated when new data arrives. Counter-based sketches are the best with regards to space, speed and accuracy [CH10, ABL⁺17]; thus, we selected one of them, namely, Misra-Gries [MG82], [CH10, Alg. 1] as basis for HH. HH provides differential privacy unlike related work [MG82, CH10, ABL⁺17]. While recent improvements achieve better performance [ABL⁺17] (amortized over the control flow), we cannot leverage them in HH due to our use of MPC (which hides the control flow).

[6] Zipf(1.5), i.e., the j-th most frequent value appears with probability proportional to $1/j^{1.5}$.

Local DP Heavy Hitters

LDP heavy hitter approaches [BS15, BNST17, EPK14, FPE16a, WLJ19, ZKM+20] mainly differ in their client-side encoding and server-side decoding of candidates, for which counts are estimated. Such encoding (in the form of domain reduction, e.g., Bloom filters [EPK14], matrix projection [BS15]) incurs information loss, which can exceed the loss due to DP randomization [WLJ19]. Notably, some encodings already provide some form of DP, e.g., [ZKM+20] (or [CDSKY20] for distinct counts), but only with large ϵ or for large data sizes.

Wang et al. [WLJ19] carefully analyze related work [BS15, BNST17, EPK14, FPE16a], which mainly use non-overlapping segments (e.g., report single bits or sets of bits), present a state-of-the-art protocol by leveraging overlapping prefixes, and show that it provides better accuracy than other LDP approaches.

We build upon the work of Wang et al. [WLJ19], which we denote PEMorig, as basis for our central-model protocol PEM. PEMorig, described in detail in Section 7.1.2, splits the clients into groups which report increasingly larger randomized prefixes. First, the clients encode the prefix of their datum by hashing it to reduce the data domain for generalized randomize response GRR (Definition 9 in Section 2.2.4). Then, GRR is applied on the hash before sending it to server. The server approximates the count for each possible prefix candidate by hashing the candidate and comparing it to all messages. If a hash matches, the candidate count is increased. Roughly, frequent candidate prefixes of, say, length η, reported by the first group, are extended by all possible bit strings of length η, and are used to find matching candidates from the second group, who reports prefixes of length 2η, etc.

We decode and output heavy hitters as our sketches contain the values or bit representation of heavy hitters. Related work, on the other hand, requires costly search to find heavy hitters from their encoded representation (e.g, hash), which has to be mapped to potential candidates from the domain [EPK14, FPE16a, WLJ19]. Note that searching to find count estimates from perturbed reports consumes significant computational resources: PEMorig performs $n2^q$ hash computations to match potential heavy hitters with randomized hashes. Even for small data of size $n = 1000$ around 1 billion hashes are computed with recommended $q = 20$. Likewise, RAPPOR [FPE16a, FPE16b] (follow-up to [EPK14]) detects frequent strings (e.g., browser homepage, installed software) by estimating joint probabilities of randomized n-grams via the expectation maximization algorithm, with complexity $O(|D||L|^{nr})$ for r reported n-grams per party for string alphabet L [FPE16b, Section V.B].

Our MPC protocols have better running time complexity than the above mentioned LDP approaches (Section 7.2.3), provide better accuracy (Section 7.3.5), and the computation can be outsourced to a few computation parties independent of the number of users (Section 7.2.4)

Central DP Heavy Hitters

An alternative to approximate DP with thresholding is probabilistic selection with pure DP, e.g., via exponential mechanism [MT07] or report noisy max [DR14] (which outputs the index of the largest noisy count using Laplace noise). These alternatives can be applied in a *peeling* fashion to find the most frequent value from a known domain, remove it from the domain, and repeat until k values are found. More computationally efficient *one-shot* methods [DSZ15b, Rog20] release k values in one go. We choose thresholding as it is preferable, especially for small data, for two reasons: First, selection requires considering *all elements from a known domain* and sampling an output from the entire domain with probability proportional to an element's utility. With thresholding, on the other hand, focusing on *data elements (from an unknown domain)* suffices – leveraged by our protocol HH. Second, for large domains (e.g., of size 2^{32}) and small data (e.g., few hundred elements) the probability mass of elements with count zero (i.e., not in the data but in the domain) can exceed the selection probability of even the most frequent element, which destroys accuracy (especially using disjoint groups that split the counts among them).

Durfee and Rogers [DR19b] first compute the actual top-k', where $k' > k$, and use $(\epsilon, 0)$-DP noise and δ-based thresholding to release at most k (ϵ, δ)-DP heavy hitters. All central DP approaches assume access to the raw data or a trusted third party. We, on the other hand, securely discover top-k' without such assumptions, and apply thresholding on noisy counts [DR19b] to release at most k DP heavy hitters in PEM. Due to thresholding, we cannot guarantee to find *exactly k* heavy hitters but only *at most k* as values with small counts (unlikely to be heavy hitters) might not exceed the threshold and are dropped. Durfee and Rogers [DR19a, Section 8] note that in such cases, e.g., flat distributions where all counts are very similar, additional output is (almost) uniformly random. Then, no output is preferable as it provides more insights about the data (i.e., flat histogram) than randomness.

MPC DP Heavy Hitters

Melis et al. [MDDC16] consider count-min and count sketches build via secure aggregation, i.e., parties evaluate multiple hash functions on their input, set the counters indexed by the hash functions to 1, and securely aggregate the counters. However, such sketches require search efforts linear in the domain size to find heavy hitters (as each candidate is mapped to sketch entries by evaluating multiple hash functions), whereas our protocols are linear in the data size (HH) or sublinear in the domain size (PEM), and efficiently handle unknown or large domains.

Naor et al. [NPR19] consider DP collection of frequently used passwords with malicious parties. On a very high-level, their hash-then-match approach is similar to PEMorig with $n2^l$ server operations, albeit more efficient ones (no hashing): Each user j receives a random l-bit value r_j from the server, computes l-bit hash h_j of her password and reports one bit, the inner product of r_j and $\mathsf{GRR}(h_j)$ modulo 2. The server keeps 2^l counters, tries to find a matching $x \in 2^l$ for every report and increments the corresponding counters. Hash values are released if their noisy counts exceed a fixed fraction of the user count. It is almost an LDP protocol, with the same accuracy limitations, where secure computation is required as malicious users cannot learn r_j. Their protocol is a series of two-party computations between users and server, whereas our protocol is a multi-party computation, where users can outsource the computation and only need to secret share their inputs.

Boneh et al. [BBC+21] securely compute heavy hitters in a malicious setting with two computation servers. They focus on novel cryptographic primitives, i.e., incremental distributed point functions, allowing secret shares of size $O(m)$ to represent a vector of 2^m values with only one non-zero element. They consider DP only optionally to bound their protocol's information leakage. In contrast, DP with high accuracy is at the heart of our design. They require large noise addition from each server, prohibiting any meaningful DP statistics for small number of clients, and overall provide less accuracy than our DP-focused protocols. They require millions of clients to achieve an absolute error of 16% for $\epsilon < 1$ [BBC+21, Appendix E] and add noise multiple times and not per group. While their server communication is more efficient than ours (requiring only kilobytes), we have similar client communication (kilobytes), however, their computation time is linear in the number of parties. PEM, for semi-honest clients (and potentially malicious servers), is linear in the domain bit-length and asymptotically faster than Boneh et al. [BBC+21]. Adjusted for $k = 256$, $b = 256$, PEM is faster than their approach for more than 6 million clients[7], however, we mainly focus on small data (corresponding to few clients).

[7] Boneh et al. [BBC+21, Table 9] process approximately 120 clients/second with 32 vCPUs, 60 GB RAM, ≈62 ms WAN delay, domain bit-length $b = 256$ and k as 0.1% of number of clients. PEM runs in less than 12 minutes in total (Figure 7.4b in Section 7.3.2) on 4 vCPUs, 8 GB RAM with 100 ms delay, $b = 64$, $k = 16$ and is independent of client count n, but linear in k and b. PEM's time in seconds multiplied by 256/16 (adjusts k), 256/64 (adjusts b), and 120 clients/s, results in $5,529,600$ clients.

4 Methodology

In the following, we describe the assessment methodology of the security, privacy, and accuracy of our protocols, how we measure their efficiency, and describe the used MPC frameworks.

We prove the security of our protocols with a simulation argument as detailed in Section 4.1. We prove the privacy of our protocols by composing known DP mechanisms as described in Section 4.2. We measure accuracy mainly as the average absolute error with 95% confidence intervals as detailed in Section 4.3. The efficiency assessment of our protocols, i.e., average running time and communication with 95% confidence intervals, is detailed in Section 4.4. The MPC frameworks employed in this work are described in Section 4.5.

4.1 Security Assessment

To prove the security of a protocol in the presence of a semi-honest adversary $\mathcal{A}$, we show the existence of a *simulator* Sim according to Goldreich [Gol09]. The simulator operates in an *ideal world* with a trusted third party providing an *ideal functionality* $\mathcal{F}$, i.e., each party P_i sends its input d_i to the trusted party which releases only $\mathcal{F}(d_1, \ldots, d_n)$. A *secure protocol* Π realizing $\mathcal{F}$ operates in the *real world* and replaces the trusted third party with MPC. The goal is to show that distributions of the real-world view and a simulated view constructed in the ideal world are computationally indistinguishable.

Informally, an adversary in the ideal world learns nothing except protocol inputs of corrupted parties and their outputs, hence, if he cannot distinguish simulated views (ideal world) from actual views (real world), he learns nothing in real-world implementations.

Next, we formalize the ideal and real-world executions – denoted ideal and real, respectively – based on Evans et al. [EKR+18, Section 2.3]. Let $\mathtt{VIEW}^p_\Pi$ denote the view of party p during the execution of protocol Π on input D, i.e., all exchanged messages and internal state. The adversary $\mathcal{A}$ corrupts a subset C of the parties $\mathcal{P}$ and has access to their views.

$(\{\mathtt{VIEW}^i_\Pi(d_i)\}_{i \in C}, \{y_i\}_{i \in \mathcal{P}}) \leftarrow \mathsf{real}_\Pi(\kappa, C, \{d_i\}_{i \in \mathcal{P}})$, the real-world execution receives as input security parameter κ, the set $C \subset \mathcal{P}$ of corrupted parties, and each parties input d_i. Then, the real-world execution runs protocol Π, with each party $i \in \mathcal{P}$ behaving honestly using its own input d_i, and outputs the view of all corrupted parties as well as the final output y_i of each party $i \in \mathcal{P}$.

$(\mathcal{S}, \{y_i\}_{i \in \mathcal{P}}) \leftarrow \mathsf{ideal}_{\mathcal{F},\mathsf{Sim}}(\kappa, C, \{d_i\}_{i \in \mathcal{P}})$, the ideal-world execution with the same inputs, relies on ideal functionality $\mathcal{F}$ to compute $\{y_i\}_{i \in \mathcal{P}} \leftarrow \mathcal{F}(\{d_i\}_{i \in \mathcal{P}})$. Then, the simulator receives the set of corrupted parties with their inputs and outputs and creates simulation $\mathcal{S}$, i.e., $\mathcal{S} \leftarrow \mathsf{Sim}(C, \{(d_i, y_i)\}_{i \in C})$. Finally, simulation $\mathcal{S}$ is output along with the protocol outputs of each party, i.e., $\{y_i\}_{i \in \mathcal{P}}$.

A protocol is secure if the output distributions of ideal and real are computationally indistinguishable [EKR+18, Def. 2.2].

Definition 14 (Semi-honest Security)**.** *A protocol* Π *securely realizes* $\mathcal{F}$ *in the presence of semi-honest PPT adversaries if there exists a PPT simulator* Sim *such that for every subset of corrupted parties* $C \subset \mathcal{P}$ *and all possible inputs* $\{d_i\}_{i\in\mathcal{P}}$ *the distributions of the simulated and real-world execution are computationally indistinguishable in security parameter* κ*, i.e.,*

$$\mathsf{ideal}_{\mathcal{F},\mathsf{Sim}}(\kappa, C, \{d_i\}_{i\in\mathcal{P}}) \overset{c}{\approx} \mathsf{real}_{\Pi}(\kappa, C, \{d_i\}_{i\in\mathcal{P}}).$$

Extension to Malicious Model

Malicious security requires additional considerations as corrupted parties can use arbitrary inputs and can alter what output honest parties receive. For further technical details of the malicious model, we refer to Evans et al. [EKR+18, Section 2.3.3] and Goldreich [Gol09, Section 7.5].

We also consider extensions to the malicious model. Note that implementing our protocols with maliciously secure frameworks is not sufficient to achieve malicious security. As malicious users might change their inputs, we also have to ensure that inputs remain consistent between steps ($\mathsf{EM}_{\mathsf{med}}$, EM^*) or are valid (PEM expects bit-vectors with at most one set bit). We describe the consistency checks required for our protocols $\mathsf{EM}_{\mathsf{med}}$, EM^*, PEM in Sections 5.2.7, 6.2.8, 7.2.4, respectively.

Composition

Our protocols consists of multiple subroutines realized with basic MPC protocols listed in Section 2.1.6. To analyze the security of an entire protocol, we apply the well-known *composition theorem* [Gol09, Section 7.3.1]: a protocol calling an ideal functionality (a subroutine provided by a trusted third party) remains secure if the ideal functionality is replaced with an MPC protocol implementing the same functionality.

4.2 Privacy Assessment

We prove that our protocols provide differential privacy as well as semi-honest security; overall, we satisfy computational differential privacy (Definition 5).

To prove privacy, we show that our protocols compose known DP mechanisms and account for the total privacy budget ϵ of our composite mechanism $\mathcal{M}$. In more detail, we rely on privacy proofs of existing DP mechanisms, namely, EM [MT07], GM [DR19b], LM [DR14], and sequential or parallel composition as detailed in Section 2.2.3. To prove security, we show the existence of a simulator Sim that simulates the view of the real-world protocol execution over data D, where Sim only knows the final output (and inputs of corrupted parties). As a consequence of these proofs, we satisfy computational differential privacy: the simulator Sim, given neighbor D' of D, computes the protocol output $\mathcal{M}(D')$ and simulates a *DP view* of the real-world execution over D.

4.3 Accuracy Assessment

In this section, we overview our accuracy assessment and reference the corresponding evaluation sections for our protocols. We theoretically analyze our protocols and also provide empirical evaluations.

We define the median like Aggarwal et al. [AMP10] as the element at position $\lceil n/2 \rceil$ in a sorted data set of size n. If not noted otherwise, we assume the data to be even and omit the ceiling notation $\lceil \cdot \rceil$ for the median position.

Theoretical Accuracy Analyses

For $\mathsf{EM}_{\mathsf{med}}$ with pruning, we analyze the *selection accuracy*, i.e., the probability to select an element from the remaining elements (close to the median) instead of the pruned elements (further away). The formal description is given in Definition 23 in Section 5.1.6 as it requires additional preliminaries. Later, in Section 6.1.5, we expand the theoretical accuracy analysis to the absolute error between actual and DP median for $\mathsf{EM}_{\mathsf{med}}$ and EM^*. In more detail, as DP requires randomization, we consider (α, β)*-accuracy*, where the absolute error is bounded by α with probability at least $1 - \beta$.

Definition 15 ((α, β)-Accuracy). *Given differentially private mechanism $\mathcal{M}_f$ computing function f, bound α, and probability β. We say $\mathcal{M}_f$ is* (α, β)-accurate *with regards to f if*

$$Pr\big[|f(D) - \mathcal{M}_f(D)| < \alpha\big] > 1 - \beta.$$

For heavy hitter discovery, we rely on the accuracy analysis of existing work [CH10, MG82, WLJ19], on which we base our protocols, as described in Section 7.1.

Empirical Accuracy Evaluation

We always compute the average over multiple runs with 95% confidence intervals as DP mechanisms are inherently randomized. We mainly define empirical accuracy as the *absolute error* between the actual result μ and the DP result $\widehat{\mu}$, i.e., $|\widehat{\mu} - \mu|$. In Sections 5.3.5 and 6.3.5, we empirically evaluate accuracy for $\mathsf{EM}_{\mathsf{med}}$ and EM^*, respectively.

Protocols for heavy hitter discovery return a set instead of a single value, requiring a different accuracy notion. For heavy hitters, we define accuracy like Wang et al. [WLJ19] as the *normalized cumulative rank* (NCR). In the following, let C_k denote the set of *actual* top-k values and C the *presumed* top-k as returned by our protocols.

Definition 16 (Normalized Cumulative Rank (NCR)). *The* normalized cumulative rank *of C is*

$$\frac{\sum_{c \in C} r(c)}{\sum_{c' \in C_k} r(c')},$$

where the frequency rank $r(c_i) = k + 1 - i$ for the i-th most frequent element $c_i \in C_k$ and zero otherwise.

Basically, detecting the most frequent element increases the cumulative rank by k, the second most frequent element adds another $k - 1$, etc., and the sum is normalized to $[0, 1]$ by dividing it with maximum score $\sum_{c' \in C_k} r(c') = k(k + 1)/2$. We also evaluated F1 scores and compared them to NCR.

Definition 17 (F1 Score). *The* F1 score

$$\frac{2pr}{p + r}$$

is the harmonic mean of precision $p = \frac{C_k \cap C}{C}$ and recall $r = \frac{C_k \cap C}{C_k}$.

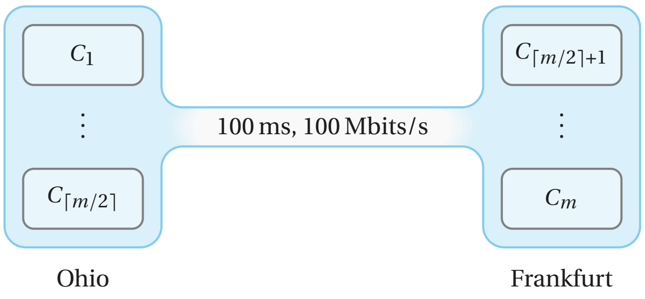


Figure 4.1: Evaluation setup: AWS EC2 instances $C_1, \ldots, C_m$ split between AWS regions Ohio and Frankfurt with an approximately 100 ms RTT, 100 Mbits/s WAN.

In other words, precision is the fraction of detected heavy hitters from all *presumed* heavy hitters, and recall is the fraction of detected heavy hitters from all *actual* heavy hitters. NCR, unlike the F1-score, also considers an element's frequency and gives more weight to more frequent elements. We empirically evaluate accuracy for HH and PEM in Section 7.3.5.

4.4 Efficiency Assessment

First, we describe our evaluation setup in Section 4.4.1. Then, we discuss the theoretically analysed as well empirically measured running time of our protocols in Section 4.4.2. Finally, we discuss the communication for clients as well as servers in Section 4.4.3. We always present the average running time and communication with 95% confidence intervals, however, we omit the intervals if they are not significant (e.g., barely visible in our graphics).

4.4.1 Evaluation Setup

We consider n input parties (clients) with sensitive inputs, and m computation parties (servers). The input parties outsource the computation, i.e., they create and send (t, m)-shares of their inputs to the computation parties, which run the secure computation on their behalf. We measure running time and communication of the computation parties.

Our two-party protocol $\mathsf{EM}_{\mathsf{med}}$ assumes $n = m = 2$ and can be extended to handle multiple input parties. Our multi-party protocols EM^*, HH, PEM assume an honest majority, i.e., at most $t = \lceil m/2 \rceil - 1$ corrupted computation parties. Notably, secret-sharing based outsourcing can be augmented to provide *fault tolerance* and handle up to 1/3 of parties dropping out during the protocol [BIK+16].

Our protocols were deployed on Elastic Compute Cloud (EC2) from Amazon Web Services (AWS). We evaluated all our protocols in real-world wide area networks (WAN) between Ohio (AWS region us-east2) and Frankfurt (eu-central1), with approximately 100 ms delay (round-trip time, RTT) and 100 Mbits/s bandwidth, and split the parties between these locations as shown in Figure 4.1. For $\mathsf{EM}_{\mathsf{med}}$, we additionally measured running time for AWS regions with round-trip times of 12 ms (Ohio–N. Virginia) and 25 ms (Ohio–Canada), with bandwidths of 430 Mbits/s and 160 Mbits/s, respectively. Our evaluation hardware are rather modest t2.medium instances with 4vCPUS and 2 GB RAM, where each vCPU "is a thread of either an Intel Xeon core or an AMD EPYC core" with clock speeds of "up to 3.3GHz" according to the AWS EC2 website [Ama20a]. For our heavy hitter protocols HH and PEM with our largest evaluation parameters (Section 7.3), we used t2.large instances also with 4vCPUs but 8 GB RAM[1]. Also, a multi-threaded version of HH, denoted $\mathsf{HH}_{\mathsf{threads}}$, was evaluated on c4.2xlarge instances with 8vCPUs and 15 GB RAM. Related work typically provides evaluation on machines with similar clock speeds [BBC+21, EKM+14, PL15]. While our machines are equipped with mainly 2 GB, related work uses 16 GB [EKM+14], 32 GB [PL15], or even 60 GB [BBC+21].

4.4.2 Running Time

In the following, we describe how we analyse and measure the running time of our protocols.

Theoretical Running Time

We give the running time complexity of our protocols as the number of basic MPC protocols from Section 2.1.6 called during our protocol execution. We present the theoretical running time complexity for $\mathsf{EM}_{\mathsf{med}}$, EM^*, HH and PEM in Sections 5.2.5, 6.2.7, 7.2.3, respectively. Whereas the exponential mechanism is linear in the domain size, our protocols are sublinear in the domain size ($\mathsf{EM}_{\mathsf{med}}$, EM^*, PEM) or linear in the data size ($\mathsf{EM}_{\mathsf{med}}$, HH).

Empirical Running Time

We measure the entire protocol execution, i.e., offline and online phase, in a real-world WAN with approximately 100 ms delay and 100 Mbits/s bandwidth. The evaluation setup is detailed in Section 4.4.1. Our protocol $\mathsf{EM}_{\mathsf{med}}$ uses $m = 2$ computation parties, for EM^* we evaluate $m \in \{3, 6, 10\}$, and for HH and PEM we set $m = 3$. We present the average running time of 20 runs with 95% confidence intervals. In our graphics, we omit the confidence intervals if they are barely visible (e.g., average deviation below 1% for Section 6.3.1).

Detailed running time measurements are provided in Sections 5.3.1, 6.3.1, 7.3.2. Our protocols $\mathsf{EM}_{\mathsf{med}}$ / GM^* (EM^* variation) / $\mathsf{HH}_{\mathsf{threads}}$ (HH variation) / PEM run in around 7 seconds (Figure 5.6b) / 1.5 minutes (Figure 6.4a, $m = 3$) / 11 minutes (Figure 7.3d) 5.4 minutes (Figure 7.4a) for millions of data values ($\mathsf{EM}_{\mathsf{med}}$) or domain elements ($\mathsf{GM}^*$, HH, PEM).

4.4.3 Communication

Next, we distinguish two types of communication. *Client communication* refers to the communication required by a client to send its (secret shared) input to the servers executing an MPC protocol on the client's behalf. *Server communication* refers to the communication between servers during the execution of an MPC protocol.

Client Communication

The client communication is small, at most in the order of kilobytes (Section 7.3.3) as clients only need to secret share their inputs with the computation servers. Thus, our evaluations focus on server communication as detailed next.

[1] We note that less RAM suffices with the *restart* feature from SCALE-MAMBA [AKR+20]: Instead of executing a large program with s loops, unrolled during MPC compilation, we execute s smaller programs consecutively – while still performing only a single offline phase. However, our exploratory evaluations found no significant improvements of the running time, and memory management with this feature is considered experimental so we did not further test it.

Server Communication

Our protocol $\mathsf{EM}_{\mathsf{med}}$ is realized mainly with garbled circuits, requiring secret sharing only in an intermediate step. Overall, the garbler is responsible for the bulk of the communication i.e., sending the circuit. As the servers have different communication complexities, we present the *total communication* for $\mathsf{EM}_{\mathsf{med}}$. Our protocols EM^*, HH, PEM are based on secret sharing and the communication is roughly evenly divided among the parties. Hence, we present *per-server communication* for these protocols.

The detailed communication evaluations are presented in Sections 5.3.6, 6.3.2, 7.3.3. The server communication is in the order of megabytes. For $m = 3$ servers (resp., 2 for $\mathsf{EM}_{\mathsf{med}}$) and $\mathsf{EM}_{\mathsf{med}}$ / GM^* / HH / PEM (32-bit domain), the evaluated server communication is at most 60 MB (Figure 5.9b, total) / 116 MB (Table 6.3, per server) / 122 MB (Figure 7.5a, per server) / 500 MB (Figure 7.6a, per server), respectively.

4.5 MPC Frameworks

While there are many MPC frameworks – see, e.g., Hastings et al. [HHNZ19] for an overview – we focus on mature frameworks (i.e., having gone through years of research and development) that are still in active development, provide detailed documentation, and support secret sharing.

Our two-party protocol $\mathsf{EM}_{\mathsf{med}}$ employs both implementation paradigms, namely, secret sharing and garbled circuits, for their respective advantages. We implement our two-party protocol in the semi-honest mixed-protocol framework *ABY* [DSZ15a], which supports both paradigms as well as efficient conversions between them (see Section 2.1.5). Code in ABY is written in C/C++ and in our evaluation we deployed the version from October 19, 2019[2].

Our multi-party protocols EM^*, PEM, HH employ MPC based on secret sharing for an efficient implementation in a network with reasonable latency and are implemented in the maliciously secure *SCALE-MAMBA* framework [AKR+20]. Furthermore, we provide some comparison to the *MP-SPDZ* framework [Kel20] which is a fork of SPDZ2, a predecessor of SCALE-MAMBA, supporting semi-honest as well as malicious security. Code in SCALE-MAMBA and MP-SPDZ is written in a Python-like language called MAMBA[3] and can be largely re-used between the frameworks. We deployed SCALE-MAMBA [AKR+20] version 1.3 and MP-SPDZ [Kel20] version 0.1.8 in our evaluation.

[2] `https://github.com/encryptogroup/ABY/tree/08baa853de76a9070cb8ed8d41e96569776e4773`

[3] SCALE-MAMBA is in the process of moving from MAMBA to Rust.

5 $\mathsf{EM}_{\mathsf{med}}$: DP Median

In this chapter, we present $\mathsf{EM}_{\mathsf{med}}$, an efficient MPC protocol for DP rank-based statistics, illustrated for the median, of the union of two confidential data sets. This chapter is based on the following publication:

> Jonas Böhler, Florian Kerschbaum. Secure Sublinear Time Differentially Private Median Computation. In *Network and Distributed Systems Security Symposium*, NDSS, 2020 [BK20b].

The median is an important robust statistical method useful for enterprise benchmarking, e.g., companies compare typical employee salaries and insurance companies can use median life expectancy to adjust insurance premiums [AMP10, Section 1]. Our protocol $\mathsf{EM}_{\mathsf{med}}$ combines the benefits of the local model (no trusted third party) and central model (better accuracy), and has a running time sublinear in the size of the data domain.

The remainder of this chapter is organized as follows. In Section 5.1, we define building blocks of our protocol. Our main insight is that the utility score for rank-based statistics can be locally evaluated on securely sorted data instead of the entire data domain. Thus, the exponentiations for the selection weights can be computed locally, avoiding costly secure exponentiations. We provide differential privacy for small data sets (sublinear in the size of the data domain) and prune large data sets with a relaxed neighboring notion of differential privacy providing limited group privacy. In Section 5.2, we describe our MPC protocol $\mathsf{EM}_{\mathsf{med}}$. We use dynamic programming with a static, i.e., data-independent, access pattern, achieving low complexity of the secure computation circuit. In Section 5.3, we provide a comprehensive evaluation over multiple AWS regions (from Ohio to N. Virgina, Canada, and Frankfurt) with a large real-world data set achieving a practical running time of less than 7 seconds for millions of records. We conclude this chapter in Section 5.4 by summarizing our results.

5.1 Building Blocks for DP Median Selection

In the following, we explain the building blocks of our protocol $\mathsf{EM}_{\mathsf{med}}$, a practically efficient sublinear time dynamic programming, which overcomes the challenges mentioned in Section 1.4, namely, running time linear in the domain size and costly secure exponentiations.

Additional notation for this chapter are given in Section 5.1.1. We give an overview of our approach in Section 5.1.2. To reduce the running time complexity, we simplify the median utility definition by using D instead of $\mathfrak{D}$ as input in Section 5.1.3. We detail how to compute selection probabilities and sample the median in Section 5.1.4. Then, we describe how to prune large data sets D in Section 5.1.5 to further reduce complexity of the secure computation.

5.1.1 Chapter-specific Notation

We consider a two-party setting, where party A and party B hold data sets D_A and D_B, respectively. The data sets D_A and D_B are *multisets* (also called bags) over domain $\mathfrak{D}$ and can contain duplicates. For our proofs, we apply union under multiset semantics, i.e., the combined data set $D = D_A \cup D_B$ is a multiset, containing all elements from D_A and D_B (including duplicates). This interpretation of union is equivalent to the sum function for multisets. We treat the difference of multisets, denoted $D_A \backslash D_B$, as a *set* containing only unique elements from D_A that are not also in D_B. Formally, $D_A \backslash D_B = \{x \in \mathfrak{D} \mid x \in D_A \text{ and } x \notin D_B\}$.

In this chapter, we start counting indices with *zero*, i.e., $D = \{d_0, d_1, \dots, d_{n-1}\} \in \mathfrak{D}^n$, as it simplifies some of our notation[1], and assume data domain $\mathfrak{D}$ to be an integer range, i.e., $\mathfrak{D} = \{x \in \mathbb{Z} \mid a \leq x \leq b\}$ with $a, b \in \mathbb{Z}$. To simplify the description we assume the size n of D to be even which can be ensured by padding. Then, with zero-indexing and padding, the median is the value $d_{n/2-1}$ in sorted D. We denote with $I_D = \{0, \dots, n-1\}$ the set of indices for D.

5.1.2 Ideal Functionality

For now, we focus on a single data set D as we later prune and merge the data sets from the two parties into one data set. For data set D with domain $\mathfrak{D}$ we compute the selection probabilities for the median for all of $\mathfrak{D}$ using only D by utilizing dynamic programming. To compute the probabilities efficiently we first define a simplified utility function $\mathsf{utility}$, which computes utility for all domain elements but only requires D as input, in Section 5.1.3. The simplified $\mathsf{utility}$ provides incorrect utility scores in the presence of duplicates. Thus, we define gap to discard these incorrect scores and compute the selection probabilities, denoted as weight. The sum of these probabilities is the basis for the cumulative distribution function, which we denote with mass. Then, we sample the differentially private median based on mass and gap as detailed in Section 5.1.4. To further reduce complexity of the secure computation complexity we prune the input D in Section 5.1.5.

A high-level overview of our protocol $\mathsf{EM}_{\mathsf{med}}$ with ideal functionalities is visualized in Figure 5.1, and we present our full protocol in Section 5.2. In the first step, the parties prune their input. Then, they securely merge and secret share their pruned data. In the third step they compute selection probabilities and, in the last step, sample the differentially private median.

Note that in the following we define gap, $\mathsf{utility}$, and weight such that direct access to the data D – and therefore the need for secure computation – is minimized: Each party can compute $\mathsf{utility}$ and weight without any access to D. Furthermore, gap has a static access pattern, independent of the elements in (sorted) D, which makes the gap function *data-oblivious*, i.e., an attacker who sees the access pattern cannot learn anything about the sensitive data.

5.1.3 Utility with Static Access Pattern

Recall Definition 13, where the median utility function $u_\mu : (\mathfrak{D}^n \times \mathfrak{D}) \to \mathbb{Z}$ gives a utility score for each $x \in \mathfrak{D}$ w.r.t. $D \in \mathfrak{D}^n$ as $u_\mu(D, x) = -\min_{\mathsf{rank}_D(x) \leq j \leq \mathsf{rank}_D(x+1)} \left|j - \frac{n}{2}\right|$. The exponential mechanism evaluates the utility function u_μ for *all* elements in the data domain $\mathfrak{D}$. However, per definition of u_μ certain outputs have the same utility, namely, duplicates and elements in $\mathfrak{D} \backslash D$. We use this observation to simplify the median utility definition and evaluate it only for elements in data set D instead of the entire domain $\mathfrak{D}$.

[1] The rank of d_i becomes i with zero-based indexing of sorted, distinct data.

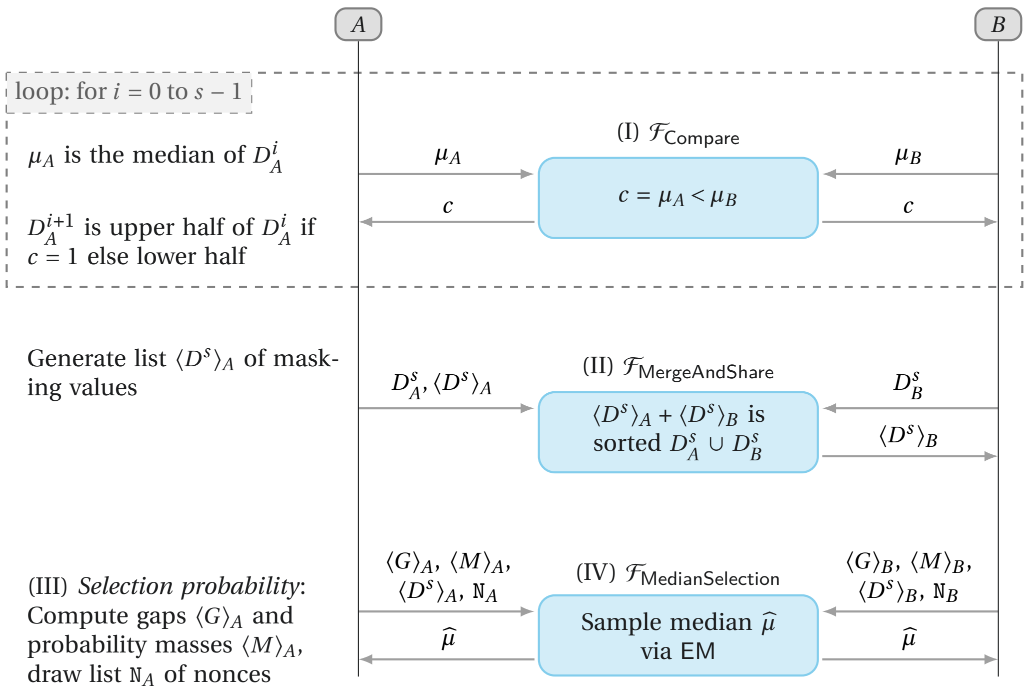

A
B
loop: for $i = 0$ to $s-1$
μ_A is the median of D_A^i
D_A^{i+1} is upper half of D_A^i if $c = 1$ else lower half
(I) $\mathcal{F}_{\text{Compare}}$
μ_A
c
$c = \mu_A < \mu_B$
μ_B
c
Generate list $\langle D^s\rangle_A$ of masking values
$D_A^s, \langle D^s\rangle_A$
(II) $\mathcal{F}_{\text{MergeAndShare}}$
$\langle D^s\rangle_A + \langle D^s\rangle_B$ is sorted $D_A^s \cup D_B^s$
D_B^s
$\langle D^s\rangle_B$
(III) *Selection probability*: Compute gaps $\langle G\rangle_A$ and probability masses $\langle M\rangle_A$, draw list $\mathtt{N}_A$ of nonces
$\langle G\rangle_A$, $\langle M\rangle_A$, $\langle D^s\rangle_A$, $\mathtt{N}_A$
$\widehat{\mu}$
(IV) $\mathcal{F}_{\text{MedianSelection}}$
Sample median $\widehat{\mu}$ via EM
$\langle G\rangle_B$, $\langle M\rangle_B$, $\langle D^s\rangle_B$, $\mathtt{N}_B$
$\widehat{\mu}$

Figure 5.1: High-level overview of EM_{med} in four steps (I)–(IV) where ideal functionalities $\mathcal{F}_{(\cdot)}$ are later realized with secure computation. Commented for A where s is the number of pruning steps, D_A^0 is sorted D_A, and $\langle D^s\rangle_A$, $\langle G\rangle_A$, $\langle M\rangle_A$ are A's shares for all values d_i^s, gaps $\text{gap}(i)$, and masses $\text{mass}(i)$ respectively ($i \in \{0, \ldots, |D^s|-1\}$).

Definition 18 ((Simplified) Median Utility Function)**.** *Let data set $D \in \mathfrak{D}^n$ be sorted. The* median utility function $\text{utility} : I_D \to \mathbb{Z}$ *scores the utility of an element of D at position $i \in I_D$ as*

$$\text{utility}(i) = \begin{cases} i - \frac{n}{2} + 1 & \textit{if } i < \frac{n}{2} \\ \frac{n}{2} - i & \textit{else} \end{cases}.$$

First, we prove the equivalence of utility function utility and u_μ only for distinct data ($D \subseteq \mathfrak{D}$) then we define gap to help with the utility computation for data sets with duplicates.

Theorem 2 (Utility equivalence)**.** *For $D \subseteq \mathfrak{D}$ and index $i \in I_D$ we have*

$$u_\mu(D, x) = \text{utility}(i)$$

for $x \in [d_i, d_{i+1})$ with $i < n/2$ and $x \in (d_{i-1}, d_i]$ with $i \geq n/2$.

Proof. First, we show that all elements in $x \in [d_i, d_{i+1})$ for $i < n/2$ and $x \in (d_{i-1}, d_i]$ for $i \geq n/2$ have the same utility. The utility u_μ of an element $x \in \mathfrak{D}$ is based on a rank from the set $S_x = \{j \mid \text{rank}_D(x) \leq j \leq \text{rank}_D(x+1)\}$ according to Definition 13. For $i < n/2$, $x \geq d_i$ and $x + 1 < d_{i+1}$ we have $\text{rank}_D(x+1) = \text{rank}_D(d_{i+1})$. All elements in the open range (d_i, d_{i+1}) have the same rank set $S = \{\text{rank}_D(x+1)\}$. The rank set for d_i, S_{d_i}, is a superset of S that also includes ranks smaller than $\text{rank}_D(x+1)$. However, $\text{rank}_D(x+1) = S_{d_i} \cap S$ minimizes the term $|\text{rank}_D(x+1) - n/2|$ since it is the value closest to $n/2$. Thus, all elements in the half-open range $[d_i, d_{i+1})$ have the same utility. Analogously, for $i \geq n/2$ elements in $(d_{i-1}, d_i]$ have the same utility.

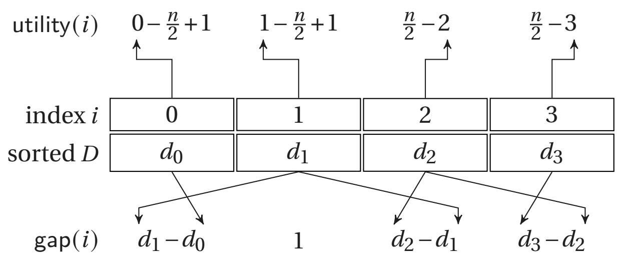


Figure 5.2: utility and gap computed on sorted D with static access pattern.

For sorted $D \subseteq \mathfrak{D}$ and $i \in I_D$, we have $\mathsf{rank}_D(d_i) = i$ and $S_{d_i} = \{\mathsf{rank}_D(d_i), \mathsf{rank}_D(d_i{+}1)\} = \{i, i{+}1\}$. Thus,

$$u_\mu(D, d_i) = -\min_{j\in\{i,i+1\}} \left|j - \frac{n}{2}\right| = \begin{cases} i + 1 - \frac{n}{2} & \text{if } i < \frac{n}{2} \\ \frac{n}{2} - i & \text{else} \end{cases} = \mathsf{utility}(i).$$

□

Thus, the sensitivity of utility is the same as u_μ. We stress that $\mathsf{utility}(i)$ only depends on the *position i* in the sorted data. Basically, we assume all elements in D are distinct, in this case $\mathsf{utility}(i) = u_\mu(D, d_i)$. To only retain the correct utility in the presence of duplicates we define gap next.

Definition 19 (Gap)**.** *The* gap function $\mathsf{gap} : I_D \to \mathbb{N}_0$ *provides the number of consecutive elements in* $\mathfrak{D}$ *with the same utility as* d_i *with*

$$\mathsf{gap}(i) = \begin{cases} d_{i+1} - d_i & \textit{if } i < \frac{n}{2} - 1 \\ 1 & \textit{if } i = \frac{n}{2} - 1 \\ d_i - d_{i-1} & \textit{else} \end{cases}.$$

Each party can compute utility (Definition 18) without any access to D. Furthermore, gap (Definition 19) has a static access pattern, independent of the elements in (sorted) D, which makes the gap function *data-oblivious*, i.e., an attacker who sees the access pattern cannot learn anything about D. Figure 5.2 visualizes how we compute utility and gap with static access pattern over sorted data D. Note that gap is defined for all n indices although there are only $n - 1$ gaps between values in D. We set the median's gap to 1 as it is the only element not contained in the union of all half-open ranges. If D contains duplicates, gap is zero for all except the duplicate closest to the median. Thus, a gap value of zero indicates incorrect utility for a duplicate and we use this to eliminate such utility values in the following.

First, with the help of utility we define the unnormalized selection probability, which we call *weight*.

Definition 20 (Weight)**.** *The* weight function $\mathsf{weight} : I_D \to \mathbb{R}$ *gives the unnormalized selection probability for an element at index* $i \in I_D$ *as*

$$\mathsf{weight}(i) = \exp\left(\epsilon \cdot \mathsf{utility}(i)\right)$$

where ϵ *is the privacy parameter from Definition 4.*

index i	0	1	2	–	3	4	5	6	–	7
sorted D	1	2	2	3,4,5	6	6	7	7	8,9	10
$\mathsf{rank}_D(\cdot)$	0	1	1	3	3	3	5	5	7	7
$u_\mu(D,\cdot)$	−3	−1	−1	−1	0	0	−1	−1	−3	−3
$\mathsf{utility}(i)$	−3	−2	−1	−1	0	0	−1	−2	−3	−3
$\mathsf{gap}(i)$	1	0	4	–	1	0	1	0	–	3

🟦 $\min(\mathfrak{D}), \max(\mathfrak{D})$ ⬜ Missing elements $\mathfrak{D}\backslash D$

Table 5.1: Utility function u_μ compared with u with static access pattern and gap for sorted $D = \{2,2,6,6,7,7\}$ from $\mathfrak{D} = \{1,\ldots,10\}$. To cover utility for all of $\mathfrak{D}$ we add $\min(\mathfrak{D}), \max(\mathfrak{D})$ to D.

Then, we use weight and gap to define the probability mass of elements with the same utility, which we call mass.

Definition 21 (Mass). *The* probability mass function $\mathsf{mass} : I_D \to \mathbb{R}$ *at $i \in I_D$ is*

$$\mathsf{mass}(i) = \sum_{h=0}^{i} \mathsf{weight}(h) \cdot \mathsf{gap}(h).$$

To ensure that mass covers *all* elements in $\mathfrak{D}$ we append the smallest (resp., largest) domain element to the beginning (resp., end) of D before computing mass. Now, we show that mass is the (unnormalized) cumulative density function for the distribution defined by $\mathsf{EM}_u^\epsilon(D)$.

Theorem 3. *Let $\mathcal{R} = \{d_0, \ldots, d_i\} \subseteq \mathfrak{D}$ with D sorted,* $\min(\mathfrak{D}), \max(\mathfrak{D})$ *$\in D$ and $i \in I_D$, then*

$$\frac{\mathsf{mass}(i)}{N} = \sum_{r\in\mathcal{R}} Pr\big[\mathsf{EM}_u^\epsilon(D) = r\big],$$

with $u = u_\mu$ and normalization $N = \sum_{r'\in\mathfrak{D}} Pr\big[\mathsf{EM}_u^\epsilon(D) = r'\big]$.

Proof. Without duplicates $\mathsf{utility} = u_\mu$ (Theorem 2), thus, $\mathsf{weight}(i) = \exp(\epsilon \cdot u_\mu(D, d_i))$ for $i \in I_D$. With duplicates weight can produce incorrect values, however, $\mathsf{weight}(i) \cdot \mathsf{gap}(i) = 0$ as gap is zero for all duplicates except the one closest to the median. In other words, we eliminate weights based on incorrect utility values as they do not alter the sum $\mathsf{mass}[i] = \sum_{h=0}^{i} \mathsf{weight}(h) \cdot \mathsf{gap}(h)$.

On the other hand, $\mathsf{gap} > 0$ indicates the number of consecutive elements in $\mathfrak{D}$ with same utility, and $\mathsf{weight}(i) \cdot \mathsf{gap}(i)$ is their unnormalized probability mass. Thus, $\mathsf{mass}[i]$ equals the sum of unnormalized probabilities for elements in $\mathcal{R} = \{\min(\mathfrak{D}), \ldots, d_i\}$, and $\mathsf{mass}[i]/N$ equals normalized probabilities $\sum_{r\in\mathcal{R}} \Pr\big[\mathsf{EM}_u^\epsilon(D) = r\big]$. $\square$

An example for utility and gap can be found in Table 5.1. It illustrates that utility for sorted D is just a sequence that increases until it reaches the median and decreases afterwards. As mentioned above, we add $\min(\mathfrak{D})$ to the beginning and $\max(\mathfrak{D})$ to the end of D (dark blue columns in Table 5.1). The utility for "missing elements" in $\mathfrak{D}\backslash D$ (light gray columns) is the same as for the preceding or succeeding element in D. Furthermore, gap is zero for the duplicates furthest away from the median and otherwise indicates the number of consecutive elements in $\mathfrak{D}$ with the same utility (e.g., $\mathsf{gap}(2) = 4$ since $2, 3, 4, 5$ have the same utility as $d_2 = 2$).

5.1.4 Median Sampling

We use inverse transform sampling (Section 2.2.4) to sample the differentially private median from the cumulative distribution function mass. Given mass, we first need to find an index $j \in I_D$[2] such that $\mathsf{mass}(j-1) \leq r < \mathsf{mass}(j)$ for a uniform random r. Then, we select an element at uniform random among the $\mathsf{gap}(j)$ consecutive elements with the same utility as the element at index j. Overall, with our simplified utility, we only need to iterate over the (small) data set D, instead of the entire domain $\mathcal{D}$.

5.1.5 Input Pruning & Utility

Our focus is on small data sets whose size is sublinear in the size of their data domain. To also support larger data sets, one can apply pruning and carefully discard data elements that cannot be the median. For this purpose, Aggarwal et al. [AMP10, Protocol 1] developed a very efficient, secure pruning technique which we denote Prune. Informally, Prune compares the local median of each party, and lets the parties discard the lower (resp., upper) half of their sorted data, which cannot contain their median of their joint data. This is repeated $\log n$ times until only one element remains, i.e., their mutual median. For our protocol, it suffices to perform $s < \log n$ steps, until the data is sublinear in the domain size. We will formalize Prune and s shortly, however, first we have to address an issue: Prune is deterministic and a comparison leaks a single bit (whose local median is larger). This leakage can potentially allow to distinguish neighboring data sets and violate differential privacy. A first idea is to randomize the comparison result itself, e.g., via randomized response (Definition 8). However, the probability that the half of the data containing the median is never discarded decreases exponentially in the number of comparisons [HLM17]. Hence, accuracy is significantly impacted and we dismiss randomized pruning in favor of a relaxation of differential privacy. To eliminate distinguishing events, we restrict the neighboring definition. He et al. [HMFS17] introduced a suitable restriction called *f-neighboring*. Informally, f-neighbors are neighbors that also have the same output w.r.t. a function f. The following definition assumes two parties A, B with data sets D_A, D_B of size n_A, n_B, respectively.

Definition 22 (f-Neighbor)**.** *Given function $f : \mathfrak{D}^{n_A} \times \mathfrak{D}^{n_B} \to \mathcal{R}$, $n_A, n_B \in \mathbb{N}$, and $D_A \in \mathfrak{D}^{n_A}$. Data sets D_B and D'_B are f-*neighbors *w.r.t. $f(D_A, \cdot)$ if*

1. *they are neighbors, and*
2. *$f(D_A, D_B) = f(D_A, D'_B)$.*

f-neighboring for D_B is similarly defined.

He et al. apply f-neighboring to record matching, where neighbors differ in at most one *non-matching* record. We, on the other hand, set f to be Prune. To verify that Prune-neighboring is not too restrictive and can be used in real-world applications we evaluated neighbors from real-world data sets [CMS17, Kag18, Soo18, ULB18] and found they are all also Prune-neighbors – albeit with limited group privacy. See Section 5.3 for details of the experiment.

Next, we detail how Aggarwal et al. [AMP10] use pruning to securely find the median of two parties A, B with respective data sets D_A, D_B. Their algorithm Prune is presented in Algorithm 2. To indicate that an operation is performed locally by, e.g., party A, we place "$\underline{\text{Party } A}$:" before an operation. Initially, Prune calls subroutine Pad described in Algorithm 1. In more detail, A

[2] For notational convenience let $j - 1 < 0$ be 0.

Algorithm 1 Pad pads the input of party $P \in \{A, B\}$ such that the element with rank k is at the position of the median [AMP10, Steps 1–3 of Protocol 1] .

```
Input: Data $D_P$, rank $k$, padding $\widehat{p}$
Output: Input padded to place $k^{\text{th}}$-ranked element at median position of the union of $D_A, D_B$
1: Sort $D_P$ and retain only the $k$ smallest values
2: Pad $D_P$ with $+\infty$ until $|D_P| = k$
3: Pad $D_P$ with $\widehat{p}$ until $|D_P| = 2^{\lceil \log_2(k) \rceil}$
4: return $D_P$
```

Algorithm 2 Prune prunes D_A, D_B to D_A^s, D_B^s via [AMP10, Protocol 1].

```
Input: Data $D_A$ from $A$, $D_B$ from $B$, pruning steps $s$, median rank $k = \lceil(|D_A| + |D_B|)/2\rceil$.
Output: $A$ has pruned data $D_A^s$, likewise $B$ has $D_B^s$.
1: Party $A$: $D_A^0 \leftarrow \mathsf{Pad}(D_A, k, +\infty)$  // Algorithm 1
2: Party $B$: $D_B^0 \leftarrow \mathsf{Pad}(D_B, k, -\infty)$
3: for $i \leftarrow 0$ to $s - 1$ do
4:     Party $A$: $\mu_A \leftarrow$ median of $D_A^i$
5:     Party $B$: $\mu_B \leftarrow$ median of $D_B^i$
6:     $[\![\mu_A]\!] \leftarrow \mathsf{En}(\mu_A), [\![\mu_B]\!] \leftarrow \mathsf{En}(\mu_B)$
7:     $c \leftarrow \mathsf{De}(\mathsf{LT}([\![\mu_A]\!], [\![\mu_B]\!]))$
8:     Party $A$: $D_A^{i+1} \leftarrow$ upper half of $D_A^i$ if $c = 1$ else lower half
9:     Party $B$: $D_B^{i+1} \leftarrow$ lower half of $D_B^i$ if $c = 1$ else upper half
10: end for
```

calls Pad$(D_A, k, +\infty)$ and B calls Pad$(D_A, k, -\infty)$ with $k = \lceil(|D_A| + |D_B|)/2\rceil$. Note that we assume the data size of each party, i.e., $|D_A|, |D_B|$, to be known, however, it can be hidden via additional padding [AMP10]. The pre-processing step Pad ensures the parties A, B sort their respective data sets D_A, D_B and only retain the smallest $k = \lceil |D_A| + |D_B| \rceil / 2$ values[3]. Then, they pad the remaining data with $-\infty, +\infty$ to be of size $2^{\lceil \log_2(k) \rceil}$ in a way that preserves the position of the median. In each pruning step, i.e., lines 4–9 in Algorithm 2, the parties compute their respective medians, μ_A, μ_B, perform a secure comparison $\mu_A < \mu_B$, and use the result to discard the halves of their data that cannot contain their mutual median, i.e., A retains the upper half of D_A if $\mu_A < \mu_B$ and the lower half otherwise, B does the opposite. After $\log n$ iterations only their *exact* mutual median remains. As we are interested in the DP median, we perform only s iterations as discussed next.

We denote data sets D_A, D_B after pruning step s as D_A^s, D_B^s and their union as D^s. The median μ of D is also the median of D^s as shown in [AMP10, Lemma 1]. How the data D is distributed among parties changes the intermediary outcome of the pruning, i.e., what elements remain in D_A^s, D_B^s. However, utility depends on an element's closeness to the median which remains or increases if elements in between are removed.

Theorem 4. Prune *does not decrease utility.*

Proof. Let $D_A = \{a_1, \ldots, a_m\}, D_B = \{b_1, \ldots, b_m\}$ with $a_1 < a_2 < \cdots < a_m$ and $b_1 < b_2 < \cdots < b_m$ (otherwise we use padding and uniqueness encoding from [AMP10]). Let $a_i^s = D_A^s[i]$, i.e., the element at index i in the data of A after pruning step s. If some indices i, j, k exist such that $a_i^{s-1} < b_j^{s-1} \leq b_k^{s-1} < a_{i+1}^{s-1}$ where $b_j^{s-1}, \ldots, b_k^{s-1}$ are not in D_B^s but a_i^{s-1} is in D_A^s then pruning step s removed $b_j^{s-1}, \ldots, b_k^{s-1}$ but neither a_i^{s-1} nor a_{i+1}^{s-1}, one of which is further away from the median

[3] If the data contains duplicates, $\lceil \log_2 n \rceil + 1$ bits are added to the element's binary representation to make it unique, which is required for the security proof from Aggarwal et al. [AMP10, Section 3.2]. We implement the uniqueness encoding but omit it in the presented protocol to simplify its description.

Original D:	a_1	b_1	a_2	b_2	a_3	b_3	a_4	b_4
$u_\mu(D,\cdot)$:	-3	-2	-1	0	0	-1	-2	-3
Pruned D^1:	–	b_1	–	b_2	a_3	–	a_4	–
$u_\mu(D^1,\cdot)$:	-2	-1	-1	0	0	-1	-1	-2

Table 5.2: Utility does not decrease before and after one pruning step for sorted $D = D_A \cup D_B$ where $D_A = \{a_1, \ldots, a_4\}$, $D_B = \{b_1, \ldots, b_4\}$. Removed elements are indicated with "–" in pruned data D^1.

than $b_j^{s-1}, \ldots, b_k^{s-1}$. However, the utility of such a removed element either remains the same (it is a duplicate of a remaining element), or increases, i.e., they have the utility of their predecessor (resp., successor) in D^s. Since one of the elements a_i^{s-1}, a_{i+1}^{s-1} is closer to the median after pruning step s than before, its utility increases and so does the utility for all elements between a_i^{s-1} and a_{i+1}^{s-1}. If no such indices i, j, k exist, then we only remove the elements furthest away from the median and the utility for remaining elements is unchanged. The utility for removed element x either remains the same (x is equal to a remaining element) or increases. The latter is due to the fact that removed elements have the same rank-based distance to the median, either $\text{rank}_{D^s}(x) = 0$ or $\text{rank}_{D^s}(x) = |D^s|$. Since $|D^s| < |D^{s-1}|$ we have $u_\mu(D^s, x) > u_\mu(D^{s-1}, x)$. □

An example of non-decreasing utility after pruning is shown in Table 5.2 for unique elements. For example, element a_1 has utility -3 before pruning, after pruning its utility increases to -2, whereas the utility for b_2, a_3 remain as before. Removed elements in the pruned data, indicated with dashes, receive the same utility as the median-closest remaining element next to them. We empirically show that pruning has only a small impact on utility and the output DP median in Section 5.3.5.

5.1.6 Accuracy & Maximum Number of Pruning Steps

On small data sets, EM* selects the same output as the exponential mechanism (Theorem 3). On large data sets, EM* performs s pruning steps where pruned elements receive the lowest selection probability. If s is too large, however, the selected output might differ as the probability mass of pruned elements can exceed the mass of remaining elements. In the following, we first define the selection accuracy, i.e., the probability to select from remaining instead of pruned elements, and then derive the maximum s based on this definition.

We separate the domain $\mathfrak{D}$ in two disjunct sets of *remaining elements* $\mathcal{R}$ and *pruned elements* $\mathcal{P}$ where $\mathcal{R} = \{x \in \mathfrak{D} \mid \min(D^s) \le x \le \max(D^s)\} \subseteq \mathfrak{D}$ and $\mathcal{P} = \{x \in \mathfrak{D} \mid x < \min(D^s) \text{ or } x > \max(D^s)\} = \mathfrak{D} \setminus \mathcal{R}$. Note that $\mathcal{R}$ contains the domain elements closest to the median.

Definition 23 (Selection Accuracy). *Let $u = u_\mu$, then* selection accuracy *is*

$$p_\mathcal{R} = 1 - p_\mathcal{P} = \sum_{x \in \mathcal{R}} Pr\big[\mathsf{EM}_u^\epsilon(D^s) = x\big],$$

i.e., $p_\mathcal{R}$ is the probability mass of all remaining elements.

With accuracy $p_\mathcal{R} > 1/2$ it is more likely to select the differentially private median among $\mathcal{R}$ than among $\mathcal{P}$. In our evaluation, we use accuracy $p_\mathcal{R} = 0.9999$. The number of pruning steps s enables a trade-off between accuracy $p_\mathcal{R}$ and computation complexity: smaller s leads to higher accuracy and larger s translates into smaller input size for the secure computation. We are interested in the maximum number of pruning steps such that it is more likely to select an element from $\mathcal{R}$ instead of $\mathcal{P}$.

Theorem 5 (Upper Bound for Pruning Steps)**.** *Let D be a data set with domain $\mathfrak{D}$, $\epsilon > 0$, and $0 < \alpha < 1$. The upper bound for pruning steps s fulfilling $p_{\mathcal{R}} \geq \alpha$ is*

$$\lfloor \log_2(\epsilon n) - \log_2\Big(\log_e\Big(\frac{\alpha}{1-\alpha}(|\mathfrak{D}|-1)\Big)\Big) - 1 \rfloor.$$

Proof. We find the maximum number of pruning steps s by examining what the maximum probability mass $p_{\mathcal{P}}$ for pruned elements can be.

First, note that the utility for all $x \in \mathcal{P}$ is the same independent of the values in D^s: Half of the values in $\mathcal{P}$ are smaller (resp., larger) than the median μ of D^s, i.e., $\mathsf{rank}_{D^s}(x) = 0$ if $x < \mu$ and $\mathsf{rank}_{D^s}(x) = |D^s|$ otherwise. Thus, $u_\mu(D^s, x) = -\left|0 - \frac{|D^s|}{2}\right| = -\left||D^s| - \frac{|D^s|}{2}\right| = -\frac{n}{2^{s+1}}$ since $|D^s| = \frac{n}{2^s}$. (Recall that D is padded before pruning such that n is a power of two.)

As the utility, and thus selection probability, is the same for all elements in $\mathcal{P}$ the probability mass $p_{\mathcal{P}}$ is maximized if $|\mathcal{P}|$ is maximized. The maximum for $|\mathcal{P}|$ is $|\mathfrak{D}| - 1$ as $\mathcal{R}$ must contain at least one element, the median μ.

Let $p'_{\mathcal{R}}, p'_{\mathcal{P}}$ be the unnormalized probability masses $p_{\mathcal{R}}, p_{\mathcal{P}}$ respectively, then

$$p'_{\mathcal{R}} = \exp(\epsilon u_\mu(D^s, \mu)) = 1$$

since $\mathcal{R} = \{\mu\}$ and $u_\mu(D^s, \mu) = 0$, and

$$p'_{\mathcal{P}} = (|\mathfrak{D}| - 1)\exp\Big(-\epsilon\frac{n}{2^{s+1}}\Big)$$

with normalization term $N = p'_{\mathcal{P}} + p'_{\mathcal{R}}$. Now accuracy $p_{\mathcal{R}}$ of at least α is equivalent to

$$\begin{aligned}\alpha \leq \frac{p'_{\mathcal{R}}}{N} &= \frac{1}{(|\mathfrak{D}|-1)\exp\Big(-\frac{\epsilon n}{2^{s+1}}\Big)+1}\\ \Leftrightarrow \exp\Big(-\frac{\epsilon n}{2^{s+1}}\Big) &\leq \frac{1-\alpha}{\alpha(|\mathfrak{D}|-1)}\\ \Leftrightarrow \log_e\Big(\frac{\alpha(|\mathfrak{D}|-1)}{1-\alpha}\Big) &\leq \frac{\epsilon n}{2^{s+1}}\\ \Leftrightarrow s &\leq \log_2\Bigg(\frac{\epsilon n}{\log_e\big(\frac{\alpha}{1-\alpha}(|\mathfrak{D}|-1)\big)}\Bigg) - 1.\end{aligned}$$

As $s \in \mathbb{N}$ we use $s = \lfloor \log_2\Big(\frac{\epsilon n}{\log_e\left(\frac{\alpha}{1-\alpha}(|\mathfrak{D}|-1)\right)}\Big) - 1 \rfloor$ which concludes the proof. □

This is a *worst-case analysis* and a tighter upper bound can be obtained by using $|\mathcal{P}|$ instead of $|\mathfrak{D}| - 1$. However, the size of $\mathcal{P}$ leaks information about D, hence, we refrain from using the tighter bound. Furthermore, we guarantee an accuracy of *at least* α, the actual accuracy can be even higher.

Lemma 2 (Sublinear Input Size)**.** *If $n \leq \log|\mathfrak{D}|$ our input is already sublinear in the size of the domain. Otherwise, $n > \log|\mathfrak{D}|$, we perform pruning with $s \in O(\log(n) - \log\log|\mathfrak{D}|)$ and the pruned data set's size is sublinear in the size of the data domain, i.e., $|D^s| = n/2^s \in O(\log|\mathfrak{D}|)$.*

Algorithm 3 MergeAndShare merges D^s_A, D^s_B into sorted D^s via [HEK12] and secret shares it.

```
Input: Pruned data $D^s_A$ from $A$ in ascending order, array $\langle D^s \rangle_A$ of $2|D^s_A|$ random values in $\mathbb{Z}_{2^{64}}$
    from $A$, $D^s_B$ from $B$ sorted in descending order.
Output: $A$ has secret shares $\langle D^s \rangle_A$ of sorted union of pruned data, resp. $B$ has $\langle D^s \rangle_B$.
1: $[\![D^s]\!] \leftarrow \mathsf{En}(D^s_A$ appended with $D^s_B)$
2: $\mathsf{Merge}(0, |D^s| - 1, [\![D^s]\!])$ //Algorithm 6 sorts $D^s$ in-place
3: $\langle D^s \rangle \leftarrow \mathsf{GC2SS}([\![D^s]\!])$ // E.g., set $\langle D^s \rangle_B \leftarrow D^s - \langle D^s \rangle_A \mod 2^{64}$
4: return $\langle D^s \rangle_B$ to $B$
```

5.2 Secure Sublinear Time Differentially Private Median Computation

First, we describe our full protocol in Section 5.2.1. Then, we provide more details, i.e., how to sort and sample securely, and describe optimizations in Sections 5.2.2–5.2.4. In Section 5.2.5 we present a running time complexity analysis and in Section 5.2.6 we prove the security of our protocol.

5.2.1 Protocol Description

Our protocol uses pruning developed by Aggarwal et al. [AMP10], which requires padding as a pre-processing step as described in Section 5.1.5. The selection probabilities are computed on securely sorted, pruned data realized via oblivious merging from Huang et al. [HEK12], detailed in Algorithm 6 in Section 5.2.2. The randomness for inverse transform sampling is provided by the parties as described in Algorithm 7 in Section 5.2.4. We build our protocol $\mathsf{EM}_{\mathsf{med}}$ from basic secure protocols for garbled circuits listed in Table 2.2 in Section 2.1.6. Note that our operations on secret shares – addition, subtraction, and multiplication with public values – require no special protocols and can be performed locally (Section 2.1.4).

Our protocol $\mathsf{EM}_{\mathsf{med}}$ has four steps, denoted with (I)–(IV):

(I) Input Pruning (Algorithm 2): Executed if the data size is not sublinear in the size of the domain. Both parties prune their data sets D_A, D_B to D^s_A, D^s_B based secure comparisons [AMP10] realized with garbled circuits.

(II) Oblivious Merge & Secret Sharing (Algorithm 3): The parties merge their pruned data D^s_A, D^s_B into sorted D^s via bitonic mergers [HEK12] implemented with garbled circuits. Note that $D^s = \{d^s_0, \dots, d^s_{|D^s|-1}\}$ is secret shared, i.e., A holds shares $\langle d^s_i \rangle_A$, B holds $\langle d^s_i \rangle_B$ for all $i \in I_{D^s}$.

(III) Selection Probability (Algorithm 4): The parties compute $\mathsf{utility}$, weight, and gap to produce shares of mass. Each party $P \in \{A, B\}$ now holds shares $\langle d^s_i \rangle_P$, $\langle \mathsf{gap}(i) \rangle_P$ and $\langle \mathsf{mass}(i) \rangle_P$ for all $i \in I_{D^s}$,

(IV) Median Selection (Algorithm 5): The parties reconstruct all shares and select the differentially private median via inverse transform sampling realized with garbled circuits. First, they sample $d^s_j \in D^s$ based on mass. Then, they select the differentially private median $\widehat{\mu}$ at uniform random among the $\mathsf{gap}(j)$ consecutive elements with the same utility as d^s_j.

To optimize the performance of the secure computation we utilize garbled circuits as well as secret sharing to use their respective advantages. E.g., multiplication of two b-bit values expressed as a Boolean circuit leads to a large circuit of size $O(b^2)$ and is more efficiently done

Algorithm 4 SelectionProbability computes the probabilities for the median utility.

```
Input: Secret shares $\langle D^s\rangle_A$ from $A$, resp. $\langle D^s\rangle_B$ from $B$, of sorted data $D^s$, and number $q$ of nonces.
Output: $A$ holds secret shares $\langle G_A\rangle$ of gaps and $\langle M_A\rangle$ of probability masses, also nonces $[\![\mathtt{N}_A^1]\!]$,
    $[\![\mathtt{N}_A^2]\!]$; likewise party $B$ has $\langle G_B\rangle$, $\langle M_B\rangle$, $[\![\mathtt{N}_B^1]\!]$, $[\![\mathtt{N}_B^2]\!]$.
 1: Party $A$: $\langle D^s\rangle_A \leftarrow (0, \langle D^s\rangle_A, 0)$
 2: Party $B$: $\langle D^s\rangle_B \leftarrow (\min(\mathfrak{D}), \langle D^s\rangle_B, \max(\mathfrak{D}))$
    //Local computations without interaction
 3: each party $P \in \{A, B\}$ does
 4:     Define arrays $\langle M\rangle_P$, $\langle G\rangle_P$ of size $|D^s|$
 5:     for $i \leftarrow 0$ to $|D^s| - 1$ do
 6:         $utility \leftarrow \begin{cases} i - \frac{|D^s|}{2} + 1 & \textbf{if } i < \frac{|D^s|}{2} \\ \frac{|D^s|}{2} - i & \textbf{else} \end{cases}$
 7:         $weight \leftarrow \exp(\epsilon \cdot utility)$
 8:         $\langle G[i]\rangle_P \leftarrow \begin{cases} \langle d^s_{i+1}\rangle_P - \langle d^s_i\rangle_P & \textbf{if } i < \frac{|D^s|}{2} - 1 \\ \langle 1\rangle_P & \textbf{if } i = \frac{|D^s|}{2} - 1 \\ \langle d^s_i\rangle_P - \langle d^s_{i-1}\rangle_P & \textbf{else} \end{cases}$
 9:         $t \leftarrow \langle M[i-1]\rangle_P$ if $i > 0$ else $0$
10:         $\langle M[i]\rangle_P \leftarrow t + weight \cdot \langle G[i]\rangle_P$
11:     end for
12:     Generate lists $\mathtt{N}_P^1, \mathtt{N}_P^2$ each with $q$ nonces from $[0, \max(\mathfrak{D}) - \min(\mathfrak{D})]$
13:     $[\![\mathtt{N}_P^1]\!] \leftarrow \mathsf{En}(\mathtt{N}_P^1)$, $[\![\mathtt{N}_P^2]\!] \leftarrow \mathsf{En}(\mathtt{N}_P^2)$
14: end each
```

Algorithm 5 MedianSelection selects the median via inverse transform sampling.

```
Input: Secret shares $\langle G\rangle_A$ of gaps, $\langle M\rangle_A$ of probability masses, and $\langle D^s\rangle_A$ of $A$'s (pruned) data,
    also garbled lists of nonces $[\![\mathtt{N}_A^1]\!]$, $[\![\mathtt{N}_A^2]\!]$ from $A$; resp., $\langle G\rangle_B$, $\langle M\rangle_B$, $\langle D^s\rangle_B$, $[\![\mathtt{N}_B^1]\!]$, $[\![\mathtt{N}_B^2]\!]$ from $B$.
Output: Differentially private median $\widehat{\mu}$ of $D_A \cup D_B$.
 1: $[\![N]\!] \leftarrow \mathsf{SS2GC}(\langle M[|D^s|-1]\rangle_A, \langle M[|D^s|-1]\rangle_B)$
 2: $[\![r]\!] \leftarrow \mathsf{RandomDraw}([\![N+1]\!], [\![\mathtt{N}_A^1]\!], [\![\mathtt{N}_B^1]\!])$ //Section 5.2.4
    //Store first index $j$, datum $d \leftarrow d^s_j$, and gap $g \leftarrow G[j]$ where $r < M[j]$
 3: Initialize $[\![j]\!] \leftarrow [\![0]\!]$, $[\![d]\!] \leftarrow \mathsf{SS2GC}(\langle d^s_0\rangle_A, \langle d^s_0\rangle_B)$, $[\![g]\!] \leftarrow \mathsf{SS2GC}(\langle G[0]\rangle_A, \langle G[0]\rangle_B)$
 4: for $i \leftarrow 0$ to $|D^s| - 2$ do
 5:     $[\![m]\!] \leftarrow \mathsf{SS2GC}(\langle M[i]\rangle_A, \langle M[i]\rangle_B)$
 6:     $[\![d_{\text{succ}}]\!] \leftarrow \mathsf{SS2GC}(\langle d^s_{i+1}\rangle_A, \langle d^s_{i+1}\rangle_B)$
 7:     $[\![g_{\text{succ}}]\!] \leftarrow \mathsf{SS2GC}(\langle G[i+1]\rangle_A, \langle G[i+1]\rangle_B)$
 8:     $[\![c]\!] \leftarrow \mathsf{LT}([\![r]\!], [\![m]\!])$
 9:     $[\![j]\!] \leftarrow \mathsf{Mux}([\![j]\!], [\![i+1]\!], [\![c]\!])$ //Set $j, d, g$ to successors if $c$ is 0
10:     $[\![d]\!] \leftarrow \mathsf{Mux}([\![d]\!], [\![d_{\text{succ}}]\!], [\![c]\!])$
11:     $[\![g]\!] \leftarrow \mathsf{Mux}([\![g]\!], [\![g_{\text{succ}}]\!], [\![c]\!])$
12: end for
13: $[\![g_{\text{rnd}}]\!] \leftarrow \mathsf{RandomDraw}([\![g]\!], [\![\mathtt{N}_A^2]\!], [\![\mathtt{N}_B^2]\!])$
14: $[\![c]\!] \leftarrow \mathsf{LT}([\![j]\!], [\![\frac{|D^s|}{2} - 1]\!])$
15: $[\![\widehat{\mu}]\!] \leftarrow \mathsf{Mux}(\mathsf{Add}([\![d]\!], [\![g_{\text{rnd}}]\!]), \mathsf{Sub}([\![d]\!], [\![g_{\text{rnd}}]\!]), [\![c]\!])$
16: return $\mathsf{De}([\![\widehat{\mu}]\!])$ to $A, B$
```

via secret sharing. On the other hand, comparison is more efficient with garbled circuits. Algorithms 3, 4 are implemented with garbled circuits. In Algorithm 2 only line 7 requires garbled circuits, the rest is either data-independent or executed locally. Secret shares, denoted with $\langle\cdot\rangle$, are created in Algorithm 3, used in Algorithm 4, and recombined in Algorithm 5. Furthermore, we compute the required exponentiations in Algorithm 4 line 7 without any secure computation.

Next, we reiterate portions of Section 5.1.3 but in the new context of secure computation.

Algorithm 6 Merge sorts bitonic list $D^s = D^s_A \cup D^s_B$ [HEK12].

```
Input: Left index $l$, right index $r$, bitonic list $[\![D^s]\!]$.
Output: None, $[\![D^s]\!]$ is sorted in-place.
 1: return if $r < l$
 2: $m \leftarrow l + \frac{r-l}{2}$
 3: for $i \leftarrow l$ to $m$ do
 4:     $e \leftarrow i + \lfloor \frac{r-l}{2} + 1 \rfloor$
 5:     $[\![c]\!] \leftarrow \mathsf{LT}([\![d^s_e]\!], [\![d^s_i]\!])$
 6:     $[\![t]\!] \leftarrow \mathsf{AND}(\mathsf{XOR}([\![d^s_i]\!], [\![d^s_e]\!]), [\![c]\!])$
 7:     $([\![D^s[i]]\!], [\![D^s[e]]\!]) \leftarrow (\mathsf{XOR}([\![d^s_i]\!], [\![t]\!]), \mathsf{XOR}([\![d^s_e]\!], [\![t]\!]))$ //Swap $d^s_i$ with $d^s_e$ if $d^s_e < d^s_i$
 8: end for
 9: $\mathsf{Merge}(l, m-1, [\![D^s]\!])$
10: $\mathsf{Merge}(m+1, r, [\![D^s]\!])$
```

5.2.2 Sorting via Garbled Circuits

Our utility definition requires the data to be sorted which inherently relies on comparisons. Comparisons are more efficiently implemented in binary circuits than arithmetic circuits, hence, we use the former. We leverage that D^s_A and D^s_B are already sorted and merge them instead of sorting the union. Oblivious merging of two lists of n sorted b-bit elements only requires $2bn\log(n)$ binary gates whereas oblivious sorting requires $\Theta(n\log(n))$ with a large constant factor [HEK12]. We use *bitonic mergers* from Huang et al. [HEK12], as formalized in Algorithm 6, which requires a bitonic list as input. A bitonic list monotonically increases and then decreases (or vice versa). We can generate a bitonic list by appending D^s_A sorted in ascending order with D^s_B sorted in descending order (Algorithm 3 line 1). Bitonic merging recursively splits the list in halves and compares and swaps elements such that every element of one half is greater than every element of the other half until the list is sorted.

5.2.3 Exponentiation and Arithmetics

To compute the probabilities for $i \in I_{D^s}$ we require *exponentiations* of the form $\exp(\epsilon \cdot \mathsf{utility}(i))$. Note that none of the arguments are secret, since ϵ is a public parameter and we defined $\mathsf{utility}$ to not require data access. Therefore, we are able to compute the required exponentiations without any secure computation. The computation of the probability mass, $\mathsf{weight}(i) \cdot \mathsf{gap}(i)$, requires two arithmetic operations: *subtractions* over secret data D^s to compute gap and *multiplication* of public values (weight), with secret values (gap). Both operations are more efficiently implemented with secret sharing, hence, we implement it that way.

5.2.4 Selection via Garbled Circuits

The median selection is realized with inverse transform sampling which is better suited for garbled circuits as it requires comparisons. First, we provide an overview of the sampling procedure assuming we have a uniform random number. Then, we describe how to securely draw such a number. Given an uniform random number $r \in [0, N]$, we compute the first index $j \in I_{D^s}$ such that the probability mass is larger than r: $\mathsf{mass}(j) > r$ (line 4 in Algorithm 5). Note that we do not sample r from $[0, 1]$ but from $[0, N]$ where $N = \mathsf{mass}(|D^s| - 1)$, i.e., the normalization factor from Equation (2.1). This allows us to use the unnormalized probabilities and eliminates divisions used in normalization. In the final step, we select the differentially private median at

Algorithm 7 RandomDraw returns uniformly random integer in given range.

Input: Upper bound M and lists of q nonces $\texttt{N}_A$,$\texttt{N}_B$ from A, B.
Output: Uniform random integer in $[0, M)$.

```
    //Find most significant 1-bit in $M$, set following bits to 1 in $mask$
 1: Initialize $[\![mask]\!] \leftarrow [\![0]\!]$, $[\![t]\!] \leftarrow [\![0]\!]$,
 2: for $i \leftarrow$ bit-length $b$ to 1 do
 3:     $[\![t]\!] \leftarrow \mathsf{OR}([\![t]\!], [\![i^{\text{th}}$ bit of $M]\!])$
 4:     $[\![i^{\text{th}}$ bit of $mask]\!] \leftarrow [\![t]\!]$
 5: end for
    //Rejection sampling with abort, based on [MM08]
 6: Initialize $[\![success]\!] \leftarrow [\![0]\!]$, define $[\![sample]\!]$
 7: for $i \leftarrow 0$ to $q-1$ do
 8:     $[\![t]\!] \leftarrow \mathsf{XOR}([\![\texttt{N}_A[i]]\!], [\![\texttt{N}_B[i]]\!])$
 9:     $[\![r]\!] \leftarrow \mathsf{AND}([\![t]\!], [\![mask]\!])$
10:     $[\![c]\!] \leftarrow \mathsf{LT}([\![r]\!], [\![M]\!])$
11:     $[\![sample]\!] \leftarrow \mathsf{Mux}([\![r]\!], [\![sample]\!], [\![c]\!])$
12:     $[\![success]\!] \leftarrow \mathsf{OR}([\![success]\!], [\![c]\!])$ //True (1) if at least one sample was accepted
13: end for
14: abort if $\mathsf{De}([\![success]\!])$ is 0
15: return $[\![sample]\!]$
```

uniform random among the $\mathsf{gap}(j)$ consecutive elements with the same utility (and thus probability) as d_j^s (line 15 in Algorithm 5). Note that the range $[0, \max(\mathfrak{D}) - \min(\mathfrak{D})]$ is used for nonces in Algorithm 4 line 12 as it is the maximum possible normalization and gap value (Algorithm 5 lines 2, 13)[4].

A straightforward way for two semi-honest parties to draw a uniform random r is to compute the sum of two nonces modulo $N+1$, where each party provides one nonce. However, this provides slightly biased results (as modulo does not evenly divide the nonce range, slightly preferencing smaller values). We implemented RandomDraw in Algorithm 7 with *rejection sampling* using efficient operations for garbled circuits, namely XOR, OR, AND, and comparison LT.

Rejection sampling is used in, e.g., Apple's macOS [MM08] and is unbiased. For a fixed input size of q nonces rejection sampling might abort. However, the abort probability is at most 2^{-q} as we describe next. We consider the worst-case rejection rate, i.e., comparison $r < M$ in line 10 of Algorithm 7. Recall that r is the XOR of uniform random values, thus, each bit in r is uniform random as well and $mask$ has all bits set after (and including) the most significant set bit in M. Masking ensures that only those bits of r remain set that are also set in $mask$, i.e., $r \leq mask$. The rejection rate is maximized if only one bit in M is set: Masking still leaves undesired values in $[M, mask]$ and range size $mask - M + 1$ is maximized when $mask$ is at its largest compared to M, i.e., when only a single bit is set in M, say at position k. Then, r is rejected with probability $1/2$ as all r with 0 at position k are accepted ($r < M$), while the other half is rejected. Increasing the number of set bits in M decreases the rejection rate (as more r can be smaller than M). Thus, the rejection probability per sample r is at most $1/2$, for an overall rejection probability of 2^{-q} as stated before. An alternative to rejection sampling is a slightly biased sampling algorithm without abort requiring only one nonce per party instead of q: If the masked XOR of nonces (r) is larger than M one uses $r - M$ as the sampled output. However, we use rejection sampling as it is

[4] An upper bound for normalization term N can be obtained by giving all possible unique elements, i.e., $\max(\mathfrak{D}) - \min(\mathfrak{D})$ elements, the highest utility, 0, thus $N = (\max(\mathfrak{D}) - \min(\mathfrak{D}))\exp(\epsilon \cdot 0) = \max(\mathfrak{D}) - \min(\mathfrak{D})$. The maximum gap is the largest possible difference of domain elements in $\mathfrak{D}$, which is $\max(\mathfrak{D}) - \min(\mathfrak{D})$.

unbiased, and only has a small impact on the running time and communication (see evaluation in Section 5.3.4).

We perform a linear scan over the *pruned data* to obliviously find index j (line 4 in Algorithm 5). Later, in Chapter 6 where we present our protocol EM^*, we use binary search to find j over subranges of the *data domain* (Section 6.2.1). However, we cannot apply binary search here as well. Binary search leaks the search pattern, especially j, which we cannot reveal as it allows inference about the data[5]. To avoid such leakage we cannot reveal j and run a linear scan.

5.2.5 Running Time Complexity

We analyse the running time of $\mathsf{EM}_{\mathsf{med}}$ based on the number of secure protocols listed in Table 2.2 in Section 2.1.6. The secure protocols require at most l operations for integers with bit-length l.

Theorem 6. *The running time complexity of* $\mathsf{EM}_{\mathsf{med}}$ *is*

$$O(\max\{\log n - \log\log|\mathfrak{D}|, \log|\mathfrak{D}| \cdot \log\log|\mathfrak{D}|\}),$$

which is sublinear in n for $n > \log|\mathfrak{D}|^{\log|\mathfrak{D}|+1}$*, and sublinear in* $|\mathfrak{D}|$ *otherwise.*

Proof. Step (I), requires $s \in O(\log n - \log\log|\mathfrak{D}|)$ comparisons (see Theorem 5). Step (II) requires $2b|D^s|\log|D^s|$ binary gates [HEK12] for $|D^s|$ elements with bit length b. Steps (III) and (IV) require $O(|D^s|)$ operations each. Since $|D^s| \in O(\log|\mathfrak{D}|)$ (Lemma 2), our overall running time is $O(\max\{\log n - \log\log|\mathfrak{D}|, \log|\mathfrak{D}| \cdot \log\log|\mathfrak{D}|\})$. □

5.2.6 Security

We combine different secure computation techniques in the *semi-honest model* introduced by [Gol09] where corrupted protocol participants do not deviate from the protocol but gather everything created during the run of the protocol. Our protocol consists of multiple subroutines realized with secure computation. To analyze the security of the entire protocol we rely on the well-known *composition theorem* [Gol09, Section 7.3.1]. Basically, a secure protocol that uses an ideal functionality (a subroutine provided by a trusted third party) remains secure if the ideal functionality is replaced with a secure computation implementing the same functionality. We consider Prune-neighboring data sets (Definition 22), i.e., neighboring data sets with the same pruning result.

Theorem 7 (Security)**.** *Our protocol* $\mathsf{EM}_{\mathsf{med}}$ *securely implements the ideal functionality of differentially private median selection via the steps* Prune, MergeAndShare, SelectionProbability *and* MedianSelection *in the semi-honest model.*

Proof. First, we show that Prune is secure based on a simulation proof from Aggarwal et al. [AMP10, Section 3.2]. Then, we define and map ideal functionalities to our real-world implementation.

Aggarwal et al. [AMP10] developed the input pruning we utilize and give a simulation-based security proof only using comparisons as ideal functionality. Note that these comparisons leak

[5] As an example, consider sorted $D^s = \{\min\mathfrak{D}, 4, 5, 5, 6, \max\mathfrak{D}\}$ with $D^s_A = \{5, 6\}$, $D^s_B = \{4, 5\}$, and we stop at median index $j = 2$ and output $\widehat{\mu} = 5$. Then, B can infer that D_A cannot contain only values smaller than 4 via proof by contradiction: Assume that D_A contains only values smaller than 4 with outputs $j, \widehat{\mu}$ as above. Then, $D^s[2] < 4$ (due to sorting) and $\widehat{\mu} = D^s[2]+g$ with $g \in [0, \mathsf{gap}(j))$. Per definition $\mathsf{gap}(j) = 1$ for the median index and so $g = 0$. Thus, $\widehat{\mu} < 4$ which contradicts $\widehat{\mu} = 5$. Note that without knowing j, one cannot rule out, e.g., $j = 0$ with uniformly random $\widehat{\mu} \in [\min\mathfrak{D}, \min(D_A \cup D_B))$, which prohibits such inference.

Algorithm 8 SimulatePruning simulates the k^{th}-ranked element computation [AMP10, Algo. 2].

Input: Parameter element rank k, real execution result μ and iteration count j. Note that D_A is known to A and all items in $D_A \cup D_B$ are distinct.
Output: Simulation of running the protocol for finding the k^{th}-ranked element μ in $D_A \cup D_B$.

```
1: $A$ initializes $D_A^1 \leftarrow \mathsf{Pad}(D_A, k, +\infty)$ //Section 5.1.5
2: for $i \leftarrow 0$ to $j - 1$ do
3:     $A$ computes $\mu_A \leftarrow \mathsf{median}(D_A^i)$
4:     Secure comparison result $c$ is set to 1 if $\mu_A < \mu$ (i.e., $\mu_A < \mu_B$) otherwise it is 0
5:     $A$ sets $D_A^{i+1} \leftarrow$ upper half of $D_A^i$ if $c = 1$ otherwise it is the lower half
6: end for
7: The final secure comparison result $c$ is set to 1 if $\mu_A < \mu$ and else it is 0
```

nothing about Prune-neighboring data sets. Prune, a partial execution of the protocol from Aggarwal et al., allows the same simulation argument. They prove the security of their computation of the k^{th}-ranked element in the semi-honest model by showing that A (similarly B) can simulate the secure protocol given its own input D_A, and the value μ of the k^{th}-ranked element. We reproduce their simulation in the following as we use the same argument with small modifications. The simulation executed by A (similarly B) by Aggarwal et al. [AMP10, Algorithm 2] is detailed in Algorithm 8. If the data D_A contains duplicates, $\lceil \log_2 |D_A| \rceil + 1$ bits are added to the binary representation of each element to make it unique as required for the simulation. E.g., A adds for each element the bit 0 followed by the rank of the element in the least significant bit positions. B follows the same procedure using 1 instead of 0. These bits are removed from the final output. Aggarwal et al. [AMP10] execute the simulation as $\mathsf{SimulatePruning}(k, \mu, \lceil \log_2(k) \rceil)$, i.e., full pruning until only one element remains. [AMP10, Lemma 2] states that the transcript of the real execution and the simulated execution are equivalent. Additionally, the state information, i.e., pruned data D_A^i, that A has at each iteration i is the same as well. Our protocol is a partial execution with s iterations. We do not know the exact value μ, however, A knows its state D_A^s at the final step and we use the median of D_A^s, denoted $\mathsf{median}(D_A^s)$, instead of μ. Altogether, we call the simulation with $\mathsf{SimulatePruning}(k, \mathsf{median}(D_A^s), s)$. We now show by contradiction that our simulation outputs the correct comparison results. Assume $c = 1$, i.e., $\mu_A < \mu_B$, at iteration i in our real execution but our simulation outputs 0, i.e., $\mu_A \geq \mathsf{median}(D_A^s)$. Then D_A^{i+1} is the lower half of D_A^i and only elements smaller than or equal to $\mu_A = \mathsf{median}(D_A^i)$ remain in D_A^{i+1} and thus in D_A^s. In other words, for $x \in D_A^{i+1}$ we have $x \leq \mu_A$ and due to $D_A^s \subseteq D_A^{i+1}$ we have $\mathsf{median}(D_A^s) < \mu_A$. However, this contradicts $\mu_A \geq \mathsf{median}(D_A^s)$, i.e., output 0. Analogously, we find a contradiction if $c = 0$ in our real execution but 1 in the simulation.

Next, we use the composition theorem to analyze the security of our protocol: We define required ideal functionalities, show how they map to our garbled circuit implementation (steps (I), (II), (IV)), and how it combines with secret sharing (step (III)). For the interactive computation, we require the ideal functionalities as shown in Figure 5.1, which we formalize next:

- $c \leftarrow \mathcal{F}_{\mathsf{Compare}}(\mu_A; \mu_B)$.

  In step (I) the ideal functionality on input μ_A, μ_B, i.e., median from A, B respectively, outputs the result of comparison $\mu_A < \mu_B$ as bit c to both parties.

- $\langle D^s \rangle_A, \langle D^s \rangle_B \leftarrow \mathcal{F}_{\mathsf{MergeAndShare}}(D_A^s; D_B^s)$.

  In step (II) the ideal functionality receives as input the pruned data D_A^s, D_B^s from A, B respectively, and outputs the sorted, merged data as secret shares, i.e., $\langle D^s \rangle_A, \langle D^s \rangle_B$ is output to A, B respectively.

- $\widehat{\mu} \leftarrow \mathcal{F}_{\mathsf{MedianSelection}}(\langle G\rangle_A, \langle M\rangle_A, \langle D^s\rangle_A; \langle G\rangle_B, \langle M\rangle_B, \langle D^s\rangle_B)$.
  In step (IV) party A inputs $\langle G\rangle_A, \langle M\rangle_A, \langle D^s\rangle_A$, party B inputs $\langle G\rangle_B, \langle M\rangle_B, \langle D^s\rangle_B$ and the ideal functionality outputs the DP median $\widehat{\mu}$ to both.

Step (III), SelectionProbability, performs local computations without interaction, and does not require any ideal functionality. We realize $\mathcal{F}_{\mathsf{Compare}}$ with garbled circuits in Algorithm 2 line 7. The ideal functionality $\mathcal{F}_{\mathsf{MergeAndShare}}$, from merging step (II), is implemented as MergeAndShare in Algorithm 3 with garbled circuits. Note that A provides the randomness for the secret sharing, i.e., $\langle D^s\rangle_A$, as additional input which is generally not required by the ideal functionality. Garbled circuits are also used in the selection step (IV), where $\mathcal{F}_{\mathsf{MedianSelection}}$ is implemented as MedianSelection in Algorithm 5. Additionally, to the input mentioned for the ideal functionality, the parties also provide nonces as a source of randomness. We rely on the established security proofs for garbled circuits in the semi-honest model provided by Lindell and Pinkas [LP09]. Outputs of (II), (III) are intermediate states of our interactive computation. As noted by Goldreich [Gol09, Section 7.1.2.3] such state can be maintained securely among the computation parties in a secret sharing manner. For security proofs of secret sharing we refer to Pullonen et al. [PBS12] and for security proofs for converting between garbled circuits and secret sharing we refer to Demmler et al. [DSZ15a].

Altogether, the execution of $\mathcal{F}_{\mathsf{Prune}}$, $\mathcal{F}_{\mathsf{MergeAndShare}}$, SelectionProbability, and $\mathcal{F}_{\mathsf{MedianSelection}}$ constitute the ideal functionality for differentially private median. Utilizing the composition theorem and [Gol09, Section 7.1.2.3] we replace the ideal functionality with secure implementations Prune, MergeAndShare, MedianSelection and secret share the intermediate states. $\square$

5.2.7 Extensions: Outsourcing, Multiple Parties, Malicious Model

Outsourcing. The two *input parties*, holding data sets D_A, D_B, can outsource the protocol evaluation to two non-colluding *computation parties*, who run the computation on their behalf. Our protocol can be outsourced as most of the operations are data-independent, i.e., using only indices of the (pre-sorted) data sets or secret-shared values. Instead of secret sharing their sorted data sets with each other, the input parties now secret share it with the computation parties. Outsourcing requires small augmentations to Prune and MergeAndShare as follows: Prune selects intermediate median values based on their position in the sorted, secret-shared data and converts them from secret-shared to garbled values. Then, Prune proceeds as before, i.e., compares the local medians, and the computation parties discard the upper/lower half of their (secret-shared) input data. MergeAndShare now also receives shares, converts them, and proceeds as before. SelectionProbability and MedianSelection already operate on secret-shared input values, output by the previous step, and require no changes. With outsourcing the input parties do not learn the Prune-neighborhood if the computation parties only return the differentially private median and no intermediate computations.

Multi-party extension. Similarly, our two-party protocol can be extended to a multi-party protocol. Consider the case where the number of input parties is sublinear in the size of the data domain and each party P_i holds a single datum d_i. Then, we do not require pruning, as the total input size is already sublinear in the size of the data domain. In this case, the input parties can outsource the computation by secret sharing their inputs with two computation parties, who run MergeAndShare (with additional reconstruction step as above), SelectionProbability, and MedianSelection. If the number of input parties is not sublinear in the domain size (or the parties hold large data sets instead of a single datum), we need to adapt Prune to handle multiple inputs. Prune is a partial execution of a two-party protocol from Aggarwal et al. [AMP10, Section 3], who also provide a secure pruning protocol for more than two parties [AMP10, Section 4]. Next, we describe their protocol augmented with secret-sharing and outsourcing. First, the input parties compute the counts of elements smaller and larger than median candidate $\mu' = \lceil \frac{a+b}{2} \rceil$, where $a = \min \mathfrak{D}$, $b = \max \mathfrak{D}$, and secret-share these counts with the computation parties. The computation parties combine these counts (per input party) into lower and upper bounds on the rank of candidate μ' and compare them against the median's rank: If the lower bound of μ' is larger than the median's rank, the input parties set $b = \mu' - 1$. If the upper bound is smaller, the input parties set $a = \mu' + 1$. These pruning steps are repeated with updated median candidate $\mu' = \lceil \frac{a+b}{2} \rceil$, until the rank fits within the bounds. Note that the two-party protocol for pruning [AMP10, Section 3] is logarithmic in the size of the *data set* (as it discards half of the data per step), whereas the multi-party protocol [AMP10, Section 4] is logarithmic in the size of the *data domain* (as it discards half of the domain range $[a, b]$ per step). After pruning, the computation parties run MergeAndShare (with additional reconstructions), SelectionProbability, and MedianSelection as before. Altogether, these modifications allow our protocol EM^* to support more than two input parties.

From Semi-honest to Malicious. To extend our protocol from semi-honest to malicious parties, we first need to implement our protocol in a maliciously secure framework, i.e., replace semi-honest sub-protocols (e.g., addition) with maliciously secure ones. Maliciously secure sub-protocols ensure that malicious behavior is prevented, e.g., changing inputs *during* computation. In case of initial pruning, which requires adaptive iterations, we also need to ensure that the input remains consistent *between* iterations. Aggarwal et al. [AMP10] define consistency checks for their adaptive pruning, which we can leverage as well. Informally, the checks ensure that inputs from the current iteration fall within bounds from previous iterations. These bounds are not revealed to the parties, and if a check fails, the computation aborts. For the two-party case, Aggarwal et al. [AMP10, Protocol 1] initialize bounds as $l_A = -\infty, l_B = -\infty$, and $u_A = \infty, u_B = \infty$. In each pruning step, the local medians μ_A, μ_B are checked as $l_A < \mu_A < u_A, l_B < \mu_B < u_B$. If $\mu_A \geq \mu_B$, update $u_A = \mu_A, l_B = \mu_B$. Otherwise ($\mu_A < \mu_B$), update $l_A = \mu_A, u_B = \mu_B$. The multi-party case is similar, however, with checks on counts a, b instead of local median values [AMP04, Protocol 3]. As we consider semi-honest parties we omit these checks in our evaluation. However, we discuss the overhead for consistency checks in Section 5.3.2

5.3 Evaluation

Our implementation is written in C/C++ using the mixed-protocol framework *ABY* developed by Demmler et al. [DSZ15a]. We use the default parameters, i.e., security parameter $\kappa = 128$ and statistical security parameter $\sigma = 40$. We chose ABY as it supports secure two-party computation based on arithmetic sharing and Yao's garbled circuits and provides efficient conversion between them (Section 2.1.5). We implemented two versions of our protocol – GC, with garbled circuits, and GC + SS, with garbled circuits as well as secret sharing – to show that using a mixed-protocol, which requires additional conversion between the schemes, is still more efficient than only utilizing garbled circuits.

Setup

We ran the evaluation on AWS t2.medium instances with 2GB RAM and 4 vCPUs (Section 4.4.1). As garbled circuits and pruning are interactive protocols they are influenced by network delay and bandwidth, therefore, we evaluated our protocol in real networks between different AWS regions with round trip times (RTT) of none (LAN), 12 ms (Ohio–N. Virginia), 25 ms (Ohio–Canada), and 100 ms (Ohio–Frankfurt), with bandwidths of 1 Gbits/s, 430 Mbits/s, 160 Mbits/s and 100 Mbits/s respectively.

Data & Parameterization

We evaluated on the Open Payments 2017 data set from the Centers for Medicare & Medicaid Services (CMS) [CMS17]. The CMS collects all payments made to physicians from drug or medical device manufacturers as required by the Physician Payments Sunshine Act. We evaluated different numbers of remaining elements after pruning (i.e., different sizes of D^s) which is inversely proportional to the privacy parameter ϵ as the number of pruning steps depends on it (see Theorem 5). We used an accuracy value of 0.9999 to determine the number of pruning steps.

Precision

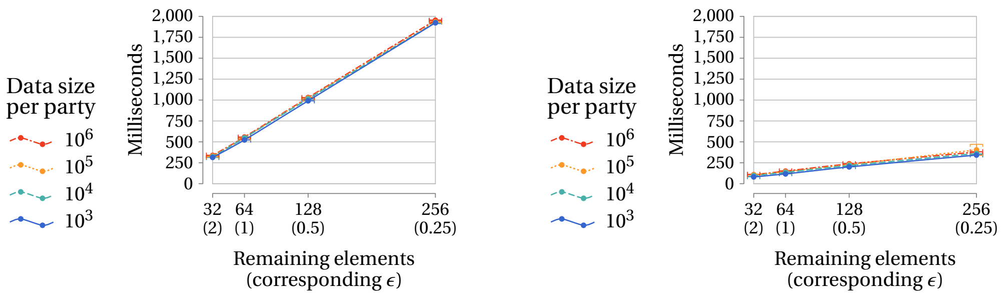
Our implementation uses fixed-point numbers with 64 bits. As probabilities are floating point numbers we evaluated the loss of decimal precision of our secure implementation compared to a floating point operation with access to unprotected data [CMS17]. For the maximum evaluated number of remaining elements, i.e., 256 (corresponding to $\epsilon = 0.25$), the difference for all elements combined was less than $6.5 \cdot 10^{-15}$.

5.3.1 Running Time

We evaluated the running time of GC and GC + SS, which includes setup time (OT extensions, garbling) and online time in seconds (or milliseconds in the LAN setting). The evaluations of running time are presented in Figure 5.3–5.6 with increasing delays and decreasing bandwidths. In each figure we plotted different data set sizes $|D_A| = |D_B| = |D|/2 \in \{10^3, 10^4, 10^5, 10^6\}$ to show that our protocol scales with increasingly larger data sets. The plotted running time is the average of 20 runs and brackets indicate the 95% confidence interval. The running time plots for GC and GC + SS have the same scale (and are grouped side-by-side) to allow for an easier comparison.

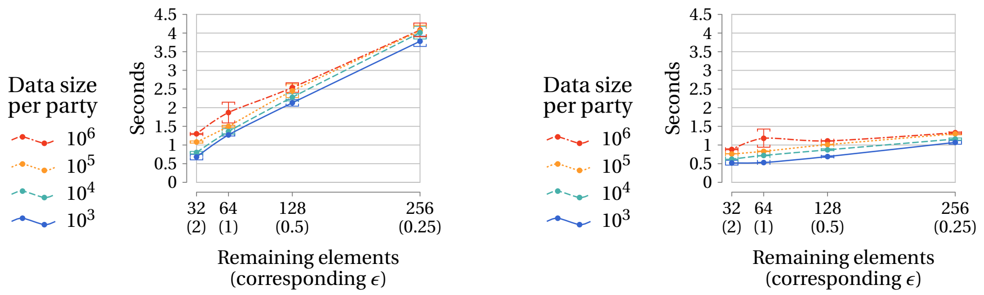
The advantage of GC + SS over GC is most obvious in the LAN setting, where the running time for GC + SS, see Figure 5.3b, is always below that of GC, see Figure 5.3a. The same is true for modest network delay as can be seen by comparing Figure 5.4b with Figure 5.4a.

For network delay of up to 100 ms with 100 Mbits/s bandwidth GC + SS is still faster than GC but less so for 32 remaining elements ($\epsilon = 2$), as shown in Figure 5.5 and 5.6. The reason for GC + SS being not much faster is the increased number of interactive pruning steps required to reach this number of remaining elements. Also, the number of additional garbled circuits to go from GC + SS to GC is smaller for few remaining elements (see Figure 5.9a), so that the pruning has more impact. Even for millions of records GC + SS has a running time of less than 2.6 seconds with 25 ms network delay (Figure 5.5b) and less than 7 seconds for 100 ms delay (Figure 5.6b).



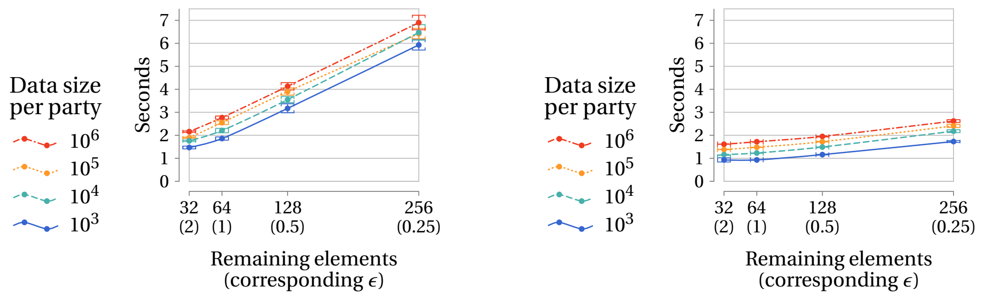
(a) GC (b) GC + SS

Figure 5.3: Running time without network delay and 1 Gbits/s bandwidth (LAN).



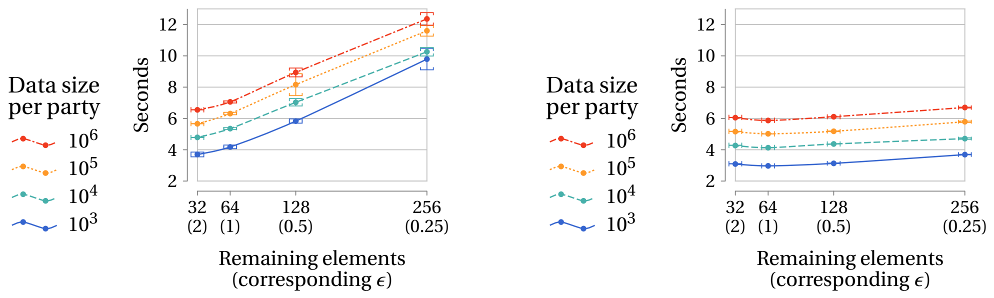
(a) GC (b) GC + SS

Figure 5.4: Running time for ~ 12 ms RTT, ~ 430 Mbits/s (Ohio and N. Virginia).



(a) GC (b) GC + SS

Figure 5.5: Running time for ~ 25 ms RTT, ~ 160 Mbits/s (Ohio and Canada).



(a) GC (b) GC + SS

Figure 5.6: Runtime for ~ 100 ms RTT, ~ 100 Mbits/s (Ohio and Frankfurt).

Index		$j-1$	j	$j+1$	
D_B	$\cdots$	d_{j-1}	d_j	d_{j+1}	$\cdots$
$D_B \setminus \{x\}$	$\cdots$	d_j	d_{j+1}	d_{j+2}	$\cdots$
$D_B \cup \{x\}$	$\cdots$	$[d_{j-3}, d_{j-2}]$	$[d_{j-2}, d_{j-1}]$	d_{j-1}	$\cdots$

Figure 5.7: Neighbors of D_B in relation to comparison index j used by Prune (values highlighted in gray). Neighbors are D_B with a value $x \in D_B$ removed or $x \in \mathfrak{D}$ added, illustrated for $x < d_j$. All data sets are sorted.

5.3.2 Consistency Checking Overhead

Malicious security requires to check input consistency between pruning steps as described in Section 5.2.7. Next, we roughly approximate the computation overhead for these checks.

The consistency checks for the two-party protocol with s pruning steps require $4s$ comparisons (LT) and $4s$ Mux to update the bounds. Note that both operations have the same complexity (Table 2.2 in Section 2.1.6). The additional operations per pruning step can be integrated in the pruning circuit with a small communication overhead in the order of kilobytes. We ignore this small communication overhead and consider a LAN setup to approximate the computation overhead. Evaluating a circuit with a single comparison required at most 5 ms in our evaluations (including its construction in a LAN). Thus, the absolute overhead is at most 640 ms for $8s$ check operations and our largest pruning with $s = 16$, i.e., only 32 remaining elements. Note that this is not a tight upper bound as our entire protocol requires less than 500 ms for the same setup (Figure 5.6). With this upper bound, the relative overhead is at most 10% in a WAN (Figure 5.3) with our largest pruning evaluations. We estimated the overhead in semi-honest ABY, however, for malicious security an implementation with consistency checks in a maliciously secure framework is required.

5.3.3 Prune-neighboring

Our focus is on small data sets (say, a few hundred values), as they are the most challenging for differential privacy. Our protocol EM^* supports such data sets with accuracy as in the central model for standard differential privacy. However, to support also larger data sets, we first prune the data with Prune which requires a relaxation of differential privacy in the form of Prune-neighboring (Definition 22), i.e., neighboring data sets also have the same output w.r.t. Prune. Next, we discuss the influence of Prune on neighboring data sets and how it limits group privacy.

Recall, Prune compares the *sorted, padded* data D_A, D_B at some fixed index j in each pruning step, and a neighbor is D_B with an element x removed or added. As Figure 5.7 illustrates, comparing a neighbor at index j is similar to using the original D at an adjacent index. Thus, neighbors are likely Prune-neighbors when the data contains multiple duplicates or is dense (no large gaps between values) and less so for sparse, unique data. In more detail, we first consider $x < d_j$ where d_j denotes the value of D_B at index j. Let the data be padded to some fixed size. Then, removing x from D_B "shifts" values larger than x to the left whereas adding x can shift smaller values to the right in the sorted data. Removing $x \in D_B$ leads to a single shift left, i.e., Prune uses d_{j+1} instead of d_j. For addition at most two right shifts can occur as we now have to consider $x \in \mathfrak{D}$ instead of $x \in D_B$. Adding $x \in [d_{j-2}, d_{j-1}]$ places it at index j in the sorted neighbor. Thus, in the worst-case for addition, Prune uses d_{j-2} instead of d_j. Note that adding/removing

D_B \ D_A	*Wages [Soo18]*	*Transactions [ULB18]*	*Times [ULB18]*	*Payments [CMS17]*	*Weights [Kag18]*	*Quantities [Kag18]*
Wages [Soo18]	>100 | 18	>100 | 14	12 | 12	22 | 22	>100 | 12	46 | 21
Transactions [ULB18]	65 | 65	8 | 8	>100 | 20	37 | 30	36 | 36	23 | 23
Times [ULB18]	>100 | 22	33 | 18	6 | 6	>100 | 13	>100 | 21	25 | 25
Payments [CMS17]	28 | 28	>100 | 11	>100 | >100	6 | 6	>100 | 41	>100 | 13
Weights [Kag18]	>100 | 43	34 | 33	4 | 4	33 | 33	>100 | 21	48 | 19
Quantities [Kag18]	30 | 30	>100 | 25	>100 | 12	>100 | 9	14 | 14	14 | 14

Table 5.3: **Minimum changes** (worst-case) in D_B to sample a neighbor that is not a Prune-neighbor w.r.t. D_A. Evaluated for 52 000 neighbors (all combinations of up to 50 removals and 50 additions with 20 samples per combination). Each row shows the minimum changes for $\epsilon = 1 \mid \epsilon = 2$ and >100 indicates none were found for up to 100 changes.

$x \geq d_j$ affects only positions larger than j, and all such neighbors are Prune-neighbors for this index. Also, if the original comparison (of D_A, D_B at j) is true, then removing $x < d_j$ produces the same result in Prune (neighbor has an even larger value at j). Likewise if it is false and we add x. To empirically verify that Prune-neighboring (Definition 22) is not too restrictive we evaluated multiple columns from real-world data sets [CMS17, Kag18, Soo18, ULB18], and found that all neighbors are also Prune-neighbors. To illustrate our evaluation methodology, one can imagine the neighboring definition in differential privacy (DP) as a graph. Each database is a vertex and if two data sets are neighbors they are connected by an edge. The common neighboring definition in DP (adding/removing one element) results in a graph. Prune-neighboring is a restriction on that graph in the sense that it removes certain edges, similar constraints on the input databases are considered in [BSRW17, HMFS17]. Any neighboring database must be in a connected component of the neighboring graph where all nodes have the same output of the Prune-function. The result of the Prune steps in our protocol determines the connected component, in which the other party's database is DP in. In that sense DP with Prune-neighboring cannot be violated by any adversary. Any choice of inputs by party A will lead to one (but different) connected component for the DP of B's database, i.e., B's database will always remain differentially private. We empirically showed that Prune-neighboring is not too restrictive, i.e., it does not remove too many edges and make the resulting connected component too small. We sampled edges from the neighboring graph resulting from the common definition on real-world data sets [CMS17, Kag18, Soo18, ULB18] using the following method: Given a real-world database for B, an element to be added or removed chosen by A (note that A must choose before knowing the result), and a step in the protocol does there exist any neighbor for B's database that is excluded by the Prune-neighboring definition. For up to 16 consecutive pruning steps (the maximum according to Theorem 5 for our highest evaluated parameters $\epsilon = 2$, and accuracy of 0.9999), we found none. Given that the connectivity in the neighboring graph is high, this implies that the connected component is expected to remain large.

Group privacy extends the neighboring definition from including (or excluding) a single value to multiple values. Therefore, to quantify group privacy we consider *multiple* changes and provide worst-case and average-case evaluation for Prune-neighboring.

D_A \ D_B	*Wages [Soo18]*	*Trans-actions [ULB18]*	*Times [ULB18]*	*Payments [CMS17]*	*Weights [Kag18]*	*Quantities [Kag18]*
Wages [Soo18]	58.6 ± 0.26	50.7 ± 0.25	49.7 ± 0.13	50.0 ± 0.17	53.9 ± 0.26	50.9 ± 0.24
Transactions [ULB18]	76.6 ± 9.59	50 ± 0.18	50.5 ± 0.26	48.5 ± 0.18	72.3 ± 0.52	55.6 ± 0.16
Times [ULB18]	63.7 ± 0.22	64.9 ± 0.20	50.3 ± 0.18	50 ± 0.25	61.2 ± 0.20	62.5 ± 0.10
Payments [CMS17]	68.9 ± 0.35	59.8 ± 0.19	>100	50 ± 0.15	71.4 ± 1.26	57.9 ± 0.13
Weights [Kag18]	55.0 ± 1.77	49.6 ± 0.15	50.9 ± 0.18	50.7 ± 0.14	61.2 ± 0.20	50.5 ± 0.24
Quantities [Kag18]	68.3 ± 0.63	64.7 ± 0.31	51 ± 0.25	51 ± 0.25	54.5 ± 0.18	59.6 ± 0.13

Table 5.4: **Average changes** in D_B to sample a neighbor that is not a Prune-neighbor w.r.t. D_A. Evaluated for 52 000 neighbors (all combinations of up to 50 removals and 50 additions with 20 samples per combination). Each row shows the average changes for $\epsilon = 2$ with 95% confidence interval and >100 indicates none were found for up to 100 changes.

Worst-case Evaluation

Table 5.3 shows the *minimum* changes required to produce a neighbor that is not also a Prune-neighbor[6]. We evaluated 52 000 neighbors (all combinations of up to 50 removals and 50 additions with 20 samples per combination) for each of the 36 ways to distribute the data between two parties (6 data sets [CMS17, Kag18, Soo18, ULB18] distributed between 2 parties). Prune-neighboring provides only limited group privacy for the largest number of pruning steps ($\epsilon = 2$). However, for our strongest privacy guarantee $\epsilon = 0.25$ we found changes leading to violations in only 2 from 36 data set combinations, requiring at least 12 changes. Furthermore, sequential composition is still supported as the result of our protocol is the median selected by the exponential mechanism which can be used as input for another (DP) mechanism. (Parallel composition, running our protocol on multiple subsets of the data at once, outputs multiple median values of these subsets.)

Average-case Evaluation

Table 5.4 shows the *average* number of changes in a data set D_B to create a neighbor that is not a Prune-neighbor w.r.t. D_A. The number of changes corresponds to the average group privacy we can expect. Each additional pruning step increases the possibility to find a non-Prune-neighbor. Thus, we use $\epsilon = 2$ as it leads to the most number of pruning steps in our evaluation. Overall, at least 49 changes were required on average to violate Prune-neighboring on the evaluated data sets.

Average-case compared to Worst-case

Table 5.5 shows that lower values of ϵ provide higher group privacy. We list the detailed minimum and average number of changes for $\epsilon \in \{0.25, 0.5, 1, 2\}$ where D_A consists of credit card transactions [ULB18]. Note that we list the minimum over all pruning steps (i.e., the value for minimum changes can be the same for different pruning steps and their corresponding ϵ). Overall, the minimum number of changes are 20 (when D_B consists of transaction times [ULB18]) and for $\epsilon < 1$ we found no group privacy violations for up to 100 changes.

[6] Some values are the same for $\epsilon \in \{1, 2\}$ as we only report the minimum number of changes over all pruning steps.

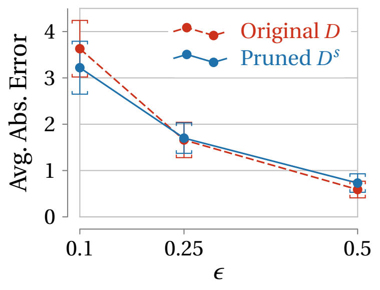
D_B	ϵ	*Avg.*	*Min.*
Open Payments [CMS17] (6M payments)	0.25	>100	>100
	0.5	>100	>100
	1	50.1 ± 0.26	37
	2	48.5 ± 0.18	30
California public salaries [Soo18] (71k wages)	0.25	>100	>100
	0.5	>100	>100
	1	76.6 ± 25.38	65
	2	76.6 ± 9.59	65
Walmart supply chain [Kag18] (175k shipment weights)	0.25	>100	>100
	0.5	>100	>100
	1	72.3 ± 0.73	36
	2	72.3 ± 0.52	36
Walmart supply chain [Kag18] (175k shipment quantities)	0.25	>100	>100
	0.5	>100	>100
	1	55.6 ± 0.23	23
	2	55.6 ± 0.16	23
Credit card [ULB18] (284k transaction times)	0.25	>100	>100
	0.5	>100	>100
	1	>100	>100
	2	50.5 ± 0.26	20

Table 5.5: **Average & minimum changes** in D_B to sample a neighbor that is not a Prune-neighbor w.r.t. D_A, where D_A consits of 284k credit card transactions [ULB18]. Evaluated for 52 000 neighbors (all combinations of up to 50 removals and 50 additions with 20 samples per combination) Evaluated for $\epsilon \in \{0.25, 0.5, 1, 2\}$ (with 95% confidence interval for average), and >100 indicates no violation was found for up to 100 changes.

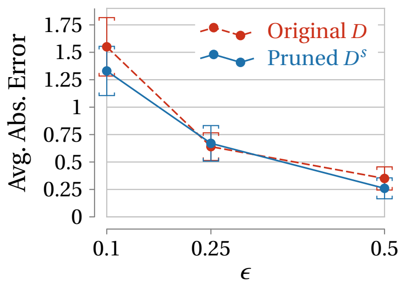
5.3.4 Sampling

For our evaluation in Section 5.3 we used $q = 20$ nonces per rejection sampling. An alternative to rejection sampling is a slightly biased sampling algorithm without abort requiring only one nonce instead of q per party. Biased sampling uses $r - M$ as the sampled output if the masked XOR of nonces (r) is larger than M instead of rejecting the biased sample (i.e., replaces loop line 7 in Algorithm 7). The masking and subtraction results in a simplified modulo operation, and the bias is due to the fact that modulo does not necessarily divide the nonce range evenly.

We compared biased sampling with rejection sampling ($q = 20$) using the median of 20 runs for our largest circuit ($\epsilon = 0.25, |D| = 2 \cdot 10^6$) with approximately 100 ms delay and 100 Mbits/s bandwidth. Biased sampling required around 28k fewer gates and sent 400 KB less than rejection sampling with $q = 20$, which corresponds to a reduction in circuit size and communication of less than 1% for GC and around 3–4% for GC + SS. The running time with biased sampling decreased by 2.2 seconds for GC (18.5% faster) but only by 0.18 seconds for GC + SS (2.6%). (For $q = 30$ an additional 44k gates and 600 KB are required compared to biased sampling, leading to similar running times as for $q = 20$.) Thus, we use rejection sampling as it is unbiased with only small impact on the runtime of GC + SS.



(a) Credit card data [ULB18], first 10^5 payment records in Cents.



(b) Walmart supply chain data [Kag18], 175k shipment weights as integers.

Figure 5.8: Absolute error averaged over 100 runs with and without pruning.

5.3.5 Absolute Error with and without Pruning

Pruning preserves the elements closest to the median and the absolute error compared to the original data is small. We evaluated the absolute error, i.e., actual median versus DP median, for the exponential mechanism on original data and pruned data: Figure 5.8 shows the average over 100 runs, where brackets indicate the 95% confidence interval. Before pruning the data was randomly split between both parties. Our evaluation shows the absolute error decreases by 3% on average over all evaluated $\epsilon \in \{0.1, 0.25, 0.5\}$. However, this is within the margin of error, since the confidence intervals for pruned data overlap with original data's confidence intervals.

5.3.6 Circuit size & Communication

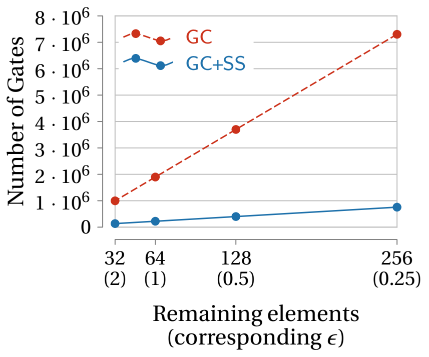
We only report circuit size and communication for 10^6 records as smaller data sizes (i.e., fewer pruning steps) do not noticeably reduce the numbers (recall, a pruning step consists of a single comparison). The number of garbled gates for GC and GC + SS depends on the number of remaining elements and is visualized in Figure 5.9a. GC requires an order of magnitude more gates as GC + SS since GC requires larger circuits for arithmetic operations whereas GC + SS avoids the need for this additional circuit complexity. The communication cost, measured in megabytes per number of remaining elements, can be found in Figure 5.9b. We do not distinguish between (precomputed) setup and online phase and present the total number of megabytes sent. Whereas GC sends about 15 megabytes for 64 remaining elements ($\epsilon = 1$), GC + SS requires less than that even for 256 remaining elements ($\epsilon = 0.25$) as fewer gates have to be garbled and evaluated.

5.3.7 Comparison to Related Work

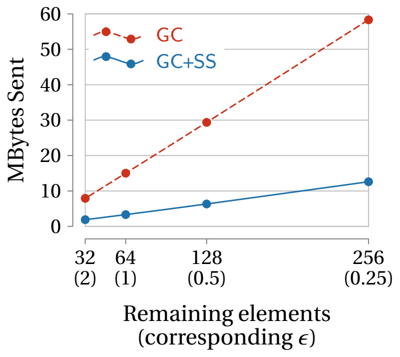
Next, we compare EM* with Pettai and Laud [PL15], our closest related work for the secure computation of differentially private median (i.e., without a trusted third party). Later, in Section 6.3.5, we expand this comparison to related works in the central model of differential privacy (i.e., with a trusted third party).

Pettai and Laud [PL15] compute differentially private analytics on distributed data via secret sharing for three parties, whereas we optimize our protocol for rank-based statistics of two parties and also use garbled circuits.[7] Both parties learn the Prune-neighborhood (for large data sets requiring pruning), but the median output can be shared (or output to only a single party)

[7] Note that 3-party computation on secret shares are usually faster than cryptographic 2-party computations [ABPP16].



(a) Number of garbled circuit gates.



(b) Total megabytes sent.

Figure 5.9: Circuit size and communication for GC vs. GC + SS.

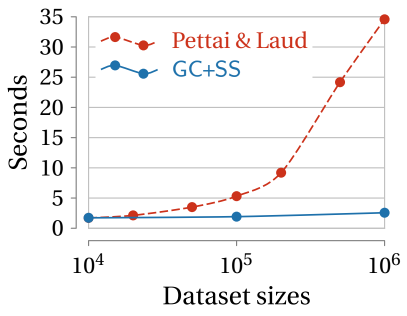


Figure 5.10: Running time of GC + SS (~25 ms RTT and ~160 Mbits/s, 256 remaining elements, $\epsilon = 0.25$) vs. Pettai and Laud [PL15] (LAN).

and processed further. Pettai and Laud [PL15] evaluated their median computation with 48GB RAM and a 12-core 3GHz CPU in a LAN. We, on the other hand, used a comparatively modest setup (t2.medium instances with 2GB RAM, 4vCPUs) and evaluated in multiple WANs. A comparison of our protocol (with ~25 ms delay, ~160 Mbits/s) and [PL15] (in a LAN) is visualized in Figure 5.10. Their median computation requires 34.5 seconds for 10^6 elements in a LAN. Our protocol runs in less than 2.6 seconds with twice as many elements even with network delay and restricted bandwidth. Pettai and Laud [PL15, Section 11] perform integer comparisons with secret sharing, which requires about 6.5 seconds per million elements which is the entire runtime of GC + SS with 100 ms delay and 100 Mbits/s bandwidth.

5.4 Summary

We presented a protocol for secure differentially private median computation on private data sets from two parties with a running time sublinear in the size of the data domain. Our protocol implements the exponential mechanism as in the local model using a distributed, secure computation protocol to achieve accuracy as in the central model without trusting a third party. For the median the exponential mechanism provides the best utility vs. privacy trade-off for low ϵ compared to additive noise (Section 3.6.1). The output is selected with an exponential bias towards elements close to the median while providing differential privacy for the individuals contained in the sensitive data. We note that our protocol can be easily extended to any rank-based DP statistic, e.g., p^{th}-percentile, by replacing the rank of the median with the desired rank in Definition 13 and adjusting sensitivity Δu accordingly. Our experiments evaluate real-world delay and bandwidth, unlike related work [PL15], which we still outperform by at least a factor of 13 (with 25 ms delay and less powerful hardware) by utilizing secret sharing as well as garbled circuits for their respective advantages. We optimize our protocol by computing as little as possible using cryptographic protocols and by applying dynamic programming with a static, i.e., data-independent, access pattern, yielding lower complexity of the secure computation circuit. Our comprehensive evaluation with a large real-world payment data set [CMS17] achieves high accuracy as in the central model and a practical running time of less than 7 seconds for millions of records in real-world WANs.

6 EM*: Decomposable DP Aggregate Functions

In this chapter, we present the MPC protocol EM* to efficiently compute the exponential mechanism for decomposable utility functions, which supports, e.g., general rank-based statistics (e.g., median, p^{th}-percentile, interquartile range) and convex optimizations. We illustrate our approach for the differentially private median. This chapter is based on the following publication:

> Jonas Böhler, Florian Kerschbaum. Secure Multi-party Computation of Differentially Private Median. In *USENIX Security Symposium*, USENIXSec, 2020 [BK20a].

Existing solutions to compute the differentially private median provide good accuracy only for large amounts of users (local model [STU17, WGSX20]), by using a trusted third party (central model [DL09, McS09, NRS07]), or support only small data sizes or domains (MPC [EKM+14, PL15]). Our approach is efficient (practical running time), scalable (sublinear in the data domain size) and accurate, i.e., the absolute error is smaller than comparable methods, and is independent of the number of users, hence, our protocols can be used even for a small number of users.

The remainder of this chapter is organized as follows. In Section 6.1, we define decomposable utility functions with examples and present our ideal functionality. Our main insights are two-fold: First, decomposability, as used in MapReduce-style frameworks for efficient aggregation over distributed data, can be satisfied by a large class of utility functions (see Section 6.1.1). Second, splitting large domains iteratively into subranges and selecting ranges instead of domain elements allows trading off some accuracy for faster running times. In Section 6.2, we describe our MPC protocol EM* (based on the exponential mechanism) as well as a variation GM* (based on the Gumbel mechanism). In Section 6.3, we evaluate our protocols. In our experiments, we were able to compute the DP median for 1 million users in 3 minutes using 3 semi-honest computation parties distributed over the Internet. We conclude this chapter in Section 6.4.

6.1 EM & Decomposability

We implement a multi-party computation of the exponential mechanism EM for decomposable aggregate functions as used in MapReduce-style algorithms. We evaluated our protocol for rank-based statistics to enable distributed parties to learn the differentially private median of their joint data. Next, we restate the main challenges of securely implementing the exponential mechanism from Section 1.4:

(i) *large domain*: the running time complexity is linear in $|\mathfrak{D}|$, i.e., the size of the domain,

(ii) *costly exponentiation*: standard EM is too inefficient, requiring $|\mathfrak{D}|$ exponentiations.

We solve these challenges by (i) recursively dividing the data domain into subranges to achieve sublinear running time in $|\mathfrak{D}|$, and (ii) focusing on utility functions that are efficiently computable in a distributed setting. We call such utility functions *decomposable*, which we formalize in Section 6.1.1, and give example applications.

In the following, we describe an overview of our solution. Our protocol EM* securely implements the exponential mechanism and our protocol GM* securely implements the Gumbel mechanism, a variation of the exponential mechanism. We achieve running time complexity sublinear in the size of the data domain $\mathfrak{D}$ by dividing $\mathfrak{D}$ into k subranges. We select the best subrange and also split it into k subranges for the next iteration, until the last subrange is small enough to directly select the final output from it. After $\lceil \log_k |\mathfrak{D}| \rceil$ iterations the selected subrange contains only one element. Each subrange selection increases the overall privacy loss ϵ, and we enable users to select a trade-off between running time, privacy loss and accuracy. For EM*, we present three different sub-protocols to compute weights (i.e., unnormalized selection probabilities) of the exponential mechanism w.r.t. ϵ:

- Weights$^{\ln(2)}$ fixes $\epsilon = \ln(2)$ to compute $\exp(\epsilon y)$ as 2^y,
- Weights$^{\ln(2)/2^d}$ allows $\epsilon = \frac{\ln(2)}{2^d}$ for some integer $d > 0$,
- Weights* supports arbitrary ϵ.

On a high-level, we have three phases in each iteration of EM*, GM*:

1. *Evaluate*: Each party locally computes the basis for utility scores for each subrange.
2. *Combine*: They combine their results into a global result and compute selection weights.
3. *Select*: Finally, they select an output based on its selection weights.

The results of the evaluation step are computed over sensitive data and might also be sensitive (e.g., utility functions for median and mode leak exact counts [LLSY16]). Therefore, we combine them via MPC to preserve privacy. To ensure efficient implementation of the combination step we require utility functions to have a certain structure as detailed next.

6.1.1 Decomposability & Applications

Recall, each party P_i holds a single value d_i (we can generalize to data sets D_i). To combine local utility scores per party into a global score for all, we require utility functions to be *decomposable*:

Definition 24 (Decomposability)**.** *We call a function* $u : (\mathfrak{D}^n \times \mathcal{R}) \to \mathbb{R}$ decomposable *with regard to function* $u' : (\mathfrak{D} \times \mathcal{R}) \to \mathbb{R}$ *if*

$$u(D, x) = \sum_{i=1}^{n} u'(d_i, x)$$

for $x \in \mathcal{R}$ *and* $D = \{d_1, \ldots, d_n\}$.

We use decomposability to easily combine utility scores in Weights$^{\ln(2)}$, Weights$^{\ln(2)/2^d}$, and to avoid costly secure evaluation of the exponential function [ABZS13, AS19, DFK+06, Kam15] in Weights*. If u is decomposable, users can compute weights locally, and securely combine them via multiplications:

$$\prod_i \exp(u'(d_i, x)\epsilon) = \exp(\sum_i u'(d_i, x)\epsilon) = \exp(u(D, x)\epsilon).$$

Application	*Utility*
Convex optimization: find x that minimizes $\sum_{i=1}^{n} l(x, d_i)$ with convex loss function l defined over D; e.g., empirical risk minimization in machine learning [BST14, STU17], and integer partitions (password frequency lists) [BDB16]	$-\sum_{i=1}^{n} l(x, d_i)$
Unlimited supply auction: find price x maximizing revenue $x \sum_i b_i(x)$, where bidder demand curve b_i indicates how many goods bidder i will buy at price x; e.g., digital goods [MT07]	$x \sum_i b_i(x)$
Frequency: select x based on its frequency in D; e.g., mode [LLSY16], where indicator variable $\mathbb{1}_c$ is 1 if condition c is met and 0 otherwise	$\sum_{i=1}^{n} \mathbb{1}_{x=d_i}$
Rank-based statistics: select x based on its rank in sorted D; e.g., k^{th}-ranked element [LLSY16]	See Section 6.1.2

Table 6.1: Applications with *decomposable* utility functions.

Generalization from a single value, d_i, to multiple values, i.e., data set D_i, per party is straightforward with decomposability: the parties compute sum of decomposable utility per value. Decomposability is satisfied by a wide range of selection problems: counts are clearly decomposable and so are utility functions that can be expressed as a sum of decomposable utility scores; examples of which are listed in Table 6.1. Also, many queries can be represented as counts via *one-hot-encoding*, i.e., a bit-vector where set bits indicate satisfied predicates [EKM+14, CWH+20]. An additional example for decomposable utility is gradient-compressed federated learning (to solve non-convex optimization problems with efficient communication), e.g., with signSGD [BWAA18]: each party only provides the sign of the gradient and the aggregate of all signs is used to perform the update step. The median, whose decomposable utility function we describe shortly, is also used in federated learning to allow robust distributed gradient descent with fault tolerance [YCKB18]. To be sublinear in the size of the domain we consider decomposability w.r.t. ranges instead of elements: parties only report one utility score per range, instead of one score per element. Note that decomposability for elements $x \in \mathfrak{D}$ does not imply decomposability for ranges $R \subset \mathfrak{D}$[1]. Later, in Chapter 7 where we present our DP heavy hitter protocols HH and PEM, we use counts of increasingly longer frequent bit-prefixes to address this issue. Next, we present a decomposable utility function w.r.t. ranges for rank-based statistics.

6.1.2 Decomposable Median Utility Function

First, we extend the median utility function from Definition 13 from elements to ranges. Then, we present a reformulation more convenient for secure implementation and show that it is decomposable. Li et al. [LLSY16, Section 2.4.3] quantify an element's utility via its *rank* relative to the median, where the rank of $x \in \mathfrak{D}$ in a data set D is the number of values in D smaller than x. As $\mathfrak{D}$ can be large, they divide $\mathfrak{D}$ in k equal-sized ranges, and also define utility per range:

Definition 25 (Median Utility Function for Ranges)**.** *The* median utility function $u_\mu : (\mathfrak{D}^n \times \mathfrak{D}) \to \mathbb{Z}$ *gives a utility score for a* range $R = [r_l, r_u)$ *where* $r_l, r_u \in \mathfrak{D}$ *w.r.t.* $D \in \mathfrak{D}^n$ *as*

$$u_\mu(D, R) = -\min_{\mathsf{rank}_D(r_l) \le j \le \mathsf{rank}_D(r_u)} \left| j - \frac{n}{2} \right|.$$

[1] Consider the mode, i.e., the most frequent element in D. Parties cannot simply report the count of their most frequent element per range as their local mode might not be the global mode. For example, for two parties with data sets $D_1 = \{1, 1, 1, 2, 2\}, D_2 = \{2, 2, 3, 3, 3\}$ the mode per data set is 1 resp. 3; however, the mode for the combined data is 2.

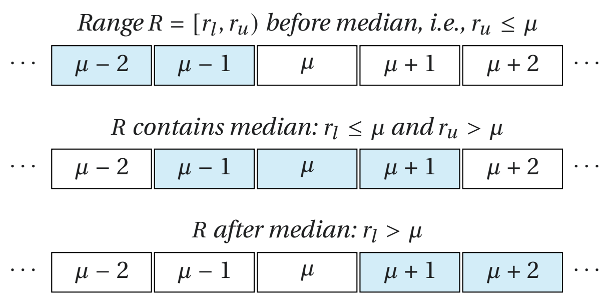

Range $R = [r_l, r_u)$ before median, i.e., $r_u \le \mu$
$\cdots$ $\mu - 2$ $\mu - 1$ μ $\mu + 1$ $\mu + 2$ $\cdots$
R contains median: $r_l \le \mu$ and $r_u > \mu$
$\cdots$ $\mu - 2$ $\mu - 1$ μ $\mu + 1$ $\mu + 2$ $\cdots$
R after median: $r_l > \mu$
$\cdots$ $\mu - 2$ $\mu - 1$ μ $\mu + 1$ $\mu + 2$ $\cdots$

Figure 6.1: Possible positions of range $R = [r_l, r_u)$ relative to median μ.

To compute u_μ one needs to find rank j minimizing the distance between the median and all range elements by iterating over all j where $\mathsf{rank}_D(r_l) \le j \le \mathsf{rank}_D(r_u)$. However, a naive implementation of u_μ leaks information as the iteration count depends on the number of duplicates in the data. We adapt u_μ next to remove this leakage. To avoid iterating over range elements observe that the utility for a range $R = [r_l, r_u)$ is defined by the element in the range closest to the median μ. Thus, it suffices to consider three cases as illustrated in Figure 6.1: The range is either positioned “before” the median ($r_u \le \mu$), contains it, or comes “after” it ($r_l > \mu$). This observation leads to the following definition without iterations[2]:

Definition 26 (Simplified Median Utility Function)**.** *The* median utility function $u^c_\mu : (\mathfrak{D}^n \times \mathfrak{D}) \to \mathbb{Z}$ *gives a utility score for a range* $R = [r_l, r_u)$ *of* $\mathfrak{D}$ *w.r.t.* $D \in \mathfrak{D}^n$ *as*

$$u^c_\mu(D,R) = \begin{cases} \mathsf{rank}_D(r_u) - \frac{n}{2} & \textit{if}\, \mathsf{rank}_D(r_u) < \frac{n}{2} \\ \frac{n}{2} - \mathsf{rank}_D(r_l) & \textit{if}\, \mathsf{rank}_D(r_l) > \frac{n}{2} \\ 0 & \textit{else} \end{cases} .$$

Next, we show equality of Definitions 25 and 26 with proof by cases.

Lemma 3. *Definitions 25 and 26 are equal.*

Proof. Consider range $R = [r_l, r_u)$ and its position relative to the median μ for Definition 25:

i) for $r_u < \mu$ we have $\mathsf{rank}_D(r_u) < n/2$, thus, $u_\mu(D, r_u) = -|\mathsf{rank}_D(r_u) - n/2| = \mathsf{rank}_D(r_u) - n/2$,

ii) for $r_l > \mu$ we have $\mathsf{rank}_D(r_l) > n/2$, thus, $u_\mu(D, r_l) = n/2 - \mathsf{rank}_D(r_l)$,

iii) otherwise, the range contains the median, i.e., u_μ equals 0.

Note that it suffices to look at r_l in case i) (resp., r_u in case ii)), as $\mathsf{rank}_D(r_l) \le \mathsf{rank}_D(r_u)$ and the range endpoint closest to μ defines the utility for the range. Overall, Definition 26 considers the same cases and is an alternative way to express Definition 25. □

In the following, we generalize from a single value per (input) party, d_i, to multiple values, i.e., data set D_i, as computation parties operate on data sets later on. Utility function u^c_μ is decomposable with regard to

$$u'(D_i,R) = \begin{cases} \mathsf{rank}_{D_i}(r_u) - \frac{|D_i|}{2} & \text{if}\, \mathsf{rank}_D(r_u) < \frac{n}{2} \\ \frac{|D_i|}{2} - \mathsf{rank}_{D_i}(r_l) & \text{if}\, \mathsf{rank}_D(r_l) > \frac{n}{2} \\ 0 & \text{else} \end{cases} ,$$

[2] Similar to Aggarwal et al. [AMP10], see also Section 5.2.7.

$\mathcal{F}_{\mathsf{EM}^*}$

1. Set $s = \lceil \log_k |\mathfrak{D}| \rceil$ and split privacy budget ϵ into $\epsilon_1, \dots, \epsilon_s$
2. Initialize $S = \mathfrak{D}$ and repeat below steps s times:
   a) Every party $p \in \mathcal{P}$ divides S into k equal-sized subranges $\{R^i = [r_l^i, r_u^i)\}_{i=1}^k$
      i. if $\epsilon_j = \ln(2)/2^d$ in step j (with integer $d \geq 0$) then party p sends the following input to the functionality
      $$\left\{\mathsf{rank}_{D_p}(r_l^i), \mathsf{rank}_{D_p}(r_u^i)\right\}_{i=1}^k \text{ and } d$$
      ii. else the input is
      $$\left\{e^{\epsilon_j\left(\mathsf{rank}_{D_p}(r_u^i)-|D_p|/2\right)}, e^{\epsilon_j\left(|D_p|/2-\mathsf{rank}_{D_p}(r_l^i)\right)}\right\}_{i=1}^k \text{ and } \epsilon_j$$
   b) The functionality combines the inputs (Section 6.1.2) and outputs $S = R^i$ with probability proportional to $\exp(u_\mu^c(D, R^i)\epsilon_j)$

Figure 6.2: Ideal functionality $\mathcal{F}_{\mathsf{EM}^*}$ for EM^*.

where $\mathsf{rank}_D(r) = \sum_{i=1}^n \mathsf{rank}_{D_i}(r)$ for range endpoints r. We will use both utility definitions interchangeably. Specifically, we use u_μ to simplify notation in our accuracy proofs (Section 6.1.5), and u_μ^c in our implementation (Section 6.2).

For implementations $\mathsf{Weights}^{\ln(2)}$, $\mathsf{Weights}^{\ln(2)/2^d}$ the parties input *ranks* for lower and upper range endpoints (as in u' above), which we combine (as u_μ^c) to efficiently compute weights. For $\mathsf{Weights}^*$ we let the parties input *weights*, i.e., $\exp(\epsilon u')$, which we can efficiently combine via multiplication. In more detail, weights for u' are:

$$e^{\epsilon\cdot u'(D_i,R)} = \begin{cases} e^{\epsilon\left(\mathsf{rank}_{D_i}(r_u)-\frac{|D_i|}{2}\right)} & \text{if } e^{\epsilon\left(\mathsf{rank}_D(r_u)-\frac{n}{2}\right)} < 1 \\ e^{\epsilon\left(\frac{|D_i|}{2}-\mathsf{rank}_{D_i}(r_l)\right)} & \text{if } 1 > e^{\epsilon\left(\frac{n}{2}-\mathsf{rank}_D(r_l)\right)} \\ 1 & \text{else} \end{cases},$$

where, e.g., $e^{\epsilon\left(\mathsf{rank}_D(r)-\frac{n}{2}\right)} = \prod_{i=1}^n e^{\epsilon\left(\mathsf{rank}_{D_i}(r)-\frac{|D_i|}{2}\right)}$ for range endpoints r. Given these inputs, we are ready to describe an idealized version of our protocols next.

6.1.3 Ideal Functionality $\mathcal{F}_{\mathsf{EM}^*}$

The ideal functionality $\mathcal{F}_{\mathsf{EM}^*}$ in Figure 6.2 describes our DP median protocol EM^* as executed by a trusted third party, which we later replace by implementing $\mathcal{F}_{\mathsf{EM}^*}$ with MPC. We iteratively select subranges of domain $\mathfrak{D}$ w.r.t. DP median via the exponential mechanism. After $s = \lceil \log_k |\mathfrak{D}| \rceil$ steps the last selected subrange contains only the DP median. We split ϵ, also called *privacy budget*, into s parts such that $\epsilon = \sum_{j=1}^s \epsilon_j$, and consume ϵ_j for each subrange selection. We describe the budget composition in Section 6.1.5 and provide a heuristic in Section 6.3. Overall, $\mathcal{F}_{\mathsf{EM}^*}$ provides ϵ-differential privacy:

Theorem 8. *$\mathcal{F}_{\mathsf{EM}^*}$, with privacy parameter ϵ_j in step $j \in \{1, \dots, s\}$, is ϵ-differentially private for $\epsilon = \sum_{j=1}^s \epsilon_j$.*

$\mathcal{F}_{\mathsf{GM}^*}$

1. Set $s = \lceil \log_k |\mathfrak{D}| \rceil$ and split privacy budget ϵ into $\epsilon_1, \dots, \epsilon_s$
2. Initialize $S = \mathfrak{D}$ and repeat below steps s times:
   a) Every party $p \in \mathcal{P}$ divides S into k equal-sized subranges $\{R^i = [r_l^i, r_u^i)\}_{i=1}^k$
   b) Party p inputs
   $$\left\{\mathsf{rank}_{D_p}(r_l^i), \mathsf{rank}_{D_p}(r_u^i), \rho_p^i(\epsilon_j)\right\}_{i=1}^k,$$
   where $\rho_p^i(\epsilon_j)$ is distributed Gumbel noise (Section 2.2.5) for subrange i with ϵ_j in step j.
3. The functionality combines the inputs into a utility score (Section 6.1.2) and outputs $S = R^i$, the argmax over the noisy utility scores.

Figure 6.3: Ideal functionality $\mathcal{F}_{\mathsf{GM}^*}$ for GM^*.

Proof. $\mathcal{F}_{\mathsf{EM}^*}$ performs s sequential steps, and each step applies the exponential mechanism $\mathsf{EM}_{u_\mu^c}^{\epsilon_i}$. Since $\mathsf{EM}_{u_\mu^c}^{\epsilon_i}$ is ϵ_i-DP (Section 2.2.4). we have ϵ_i-DP *per step* Thus, according to the composition theorem [DR14], the *total* privacy budget after all steps is $\sum_{j=1}^{s} \epsilon_j$. □

6.1.4 Ideal Functionality $\mathcal{F}_{\mathsf{GM}^*}$

As before, the ideal functionality $\mathcal{F}_{\mathsf{GM}^*}$ in Figure 6.3 assumes the existence of a trusted third party. We later remove this assumption by implementing $\mathcal{F}_{\mathsf{GM}^*}$ with MPC as GM^*. Whereas $\mathcal{F}_{\mathsf{EM}^*}$ realizes DP median selection via the exponential mechanism EM, protocol $\mathcal{F}_{\mathsf{GM}^*}$ implements selection via the Gumbel mechanism GM. The main difference between the ideal functionalities is that $\mathcal{F}_{\mathsf{GM}^*}$ additionally requires distributed noise generation (as detailed in Section 2.2.5) but does not require to input weights. The same DP proof as for $\mathcal{F}_{\mathsf{EM}^*}$ applies as GM and EM have the same output distribution (Section 2.2.4).

6.1.5 Accuracy of DP Median

We recall Definition 15 from Section 4.3, i.e., given DP mechanism $\mathcal{M}_f$ computing function f, (α, β)-*accuracy* is defined as $\Pr\left[|f(D) - \mathcal{M}_f(D)| < \alpha\right] > 1 - \beta$. In other words, the absolute error between actual result and DP result is bounded by α with probability at least $1 - \beta$. In the following, we discuss how the data distribution influences accuracy. Then, we present worst-case bounds on the accuracy of the exponential mechanism for median selection.

Data Distribution

Accuracy depends on the data distribution, specifically, on *gaps* $d_{i+1} - d_i$, and *duplicates* $d_i = d_j$ with $i \neq j$. Recall that a differentially private mechanism bounds the probability that data set D and its neighbor D' can be distinguished from the mechanism output. As neighbor D' may contain values from the gaps of D, these gap values must be output with a non-zero probability. This is why bounds for absolute error depend on such gaps between data elements in this and related work (see Table 3.1 in Section 3.6.2). As a worst-case example, consider a data set with domain $\mathfrak{D} = \{0, 1, \dots, 10^9\}$, and equal number of duplicates for 0 and 10^9. Then, smooth sensitivity is extremely large with 10^9 and the exponential mechanism outputs a value at uniform random. However, for such pathological, worst-case data even the actual median does not provide much insight. On the other hand, the number of duplicates in the data can increase accuracy dramatically. For example, consider a data set where the median has $2c$ duplicates: $d_{n/2\pm i} = d_{n/2}$ for $i \in \{1, \dots, c\}$. Then, the probability that the exponential mechanism outputs the median is $\exp(c\epsilon)$ times higher than for any other element. Such duplicates also fit the intuition that the median is a "typical" value from the data that represents it well. In general, the probability to output a "bad" element x decreases exponentially in $\sum c_i$, where $c_i \geq 1$ are duplicate counts of "good" elements y_i, which are closer to the median than x.

Accuracy Bounds

In the following, we show that the output of $\mathsf{EM}_u^\epsilon(D)$ over domain $\mathcal{R}$ contains an element at most $\left\lfloor \frac{\ln(|\mathcal{R}|/\beta)}{\epsilon} \right\rfloor$ positions away from the median in the sorted data. Note that $|\mathcal{R}|$ is k if we select among k subranges or $|\mathfrak{D}|$ if we output elements directly.

For our accuracy proofs we structure the domain as a tree: we set $\mathfrak{D}$ as the root of a tree of height $\log_b |\mathfrak{D}|$, for some base b, with k child nodes per parent. The child nodes are equal-sized subranges of the parent node and R_i^j denotes the i^{th} subrange in level j.

Theorem 9 (Median Accuracy for Ranges)**.** *Fixing a data set D of size n with a set of k subranges $\mathcal{R} = \{R_1^j, \dots, R_k^j\}$ of data domain $\mathfrak{D}$. Then, output of $\mathsf{EM}_u^\epsilon(D)$ over domain $\mathcal{R}$ contains an element at most $\left\lfloor \frac{\ln(k/\beta)}{\epsilon} \right\rfloor$ positions away from median position $\frac{n}{2}$ with probability at least $1 - \beta$.*

Our proof uses Corollary 3.12 from [DR14], which we restate as the following Lemma:

Lemma 4 (Accuracy of the Exponential Mechanism)**.** *Fixing a data set D, and let the maximum utility score of any element $r \in \mathcal{R}$ be denoted $OPT = \max_{r\in\mathcal{R}} u(D, r)$. Then, we have*

$$Pr\left[u(D, \mathsf{EM}_u^\epsilon(D)) \leq OPT - \frac{2\Delta u}{\epsilon}(\ln|\mathcal{R}| + t)\right] \leq \exp(-t).$$

Proof of Theorem 9. First, we bound the utility difference between optimal and selected output. Then, we translate this to a bound on the output's rank.

The complementary of Lemma 4 with $\Delta u = \frac{1}{2}$ is

$$\Pr\left[\mathrm{OPT} - u(D, \mathsf{EM}_u^\epsilon(D)) < \frac{\ln|\mathcal{R}| + t}{\epsilon}\right] > 1 - \exp(-t).$$

Let $R_i^j = [r_l, r_u)$ be the output of $\mathsf{EM}_u^\epsilon(D)$. Recall, that for median utility OPT = 0, then,

$$\begin{aligned}\mathrm{OPT} - u(D, \mathsf{EM}_u^\epsilon(D)) &= 0 - u(D, R_i^j)\\ &= \min_{\mathsf{rank}_D(r_l) \leq j \leq \mathsf{rank}_D(r_u)} \left| j - \frac{n}{2} \right|.\end{aligned}$$

Next, we consider different cases for R_i^j to bound the rank difference between the selected range and the range that contains the median. Assume median $\mu \notin R_i^j$, as otherwise the bound holds trivially, and let d denote the utility difference $\mathrm{OPT} - u(D, \mathsf{EM}_u^\epsilon(D))$.

For $r_u < \mu$ we have $d = |\mathsf{rank}_D(r_u) - \frac{n}{2}| = \frac{n}{2} - \mathsf{rank}_D(r_u)$ from which we obtain $\mathsf{rank}_D(r_u) > \frac{n}{2} - \frac{\ln|\mathcal{R}|+t}{\epsilon}$ with probability at least $1 - \exp(-t)$. Analog, for $r_l > \mu$ we have $d = \mathsf{rank}_D(r_l) - \frac{n}{2}$, and obtain $\mathsf{rank}_D(r_l) < \frac{n}{2} + \frac{\ln|\mathcal{R}|+t}{\epsilon}$ with the same probability. Altogether, R_i^j is at most $\left\lfloor \frac{\ln|\mathcal{R}|+t}{\epsilon} \right\rfloor$ rank positions away from median rank $n/2$ with probability at least $1 - \exp(-t)$. We have $k = |\mathcal{R}|$ and setting $\beta = \exp(-t)$ concludes the proof. $\square$

To obtain an absolute error with regards to data elements, consider domain elements instead of subranges as the output domain of the exponential mechanism.

Corollary 1 (Median Accuracy). *Fixing a sorted data set D of size n, let μ be the median of D, and $\widehat{\mu}$ the output of $\mathsf{EM}_u^\epsilon(D)$ over domain $\mathfrak{D}$. Then, absolute error $|\mu - \widehat{\mu}|$ is at most*

$$\max_{i \in \{+1,-1\}\cdot\lfloor \ln(|\mathfrak{D}|/\beta)/\epsilon \rfloor} \left| d_{\frac{n}{2}+i} - d_{\frac{n}{2}} \right|$$

with probability at least $1 - \beta$.

The proof follows directly from Theorem 9 by replacing k with $|\mathfrak{D}|$.

The same analysis applies to our protocol $\mathsf{EM}_{\mathsf{med}}$ (Section 5) on small data, i.e., without pruning. With pruning, we only need to replace D with pruned D^s in all statements.

Accuracy for Evenly Spaced Data

Next, we consider a special case, evenly spaced data, to obtain a tighter error bound. Nissim et al. [NRS07, Section 3.1] illustrate that smooth sensitivity of the median is superior to global sensitivity with evenly spaced data. Likewise, we illustrate that the exponential mechanism is even better. Note that evenly spaced data has a constant gap $d_{i+1} - d_i$, which simplifies the accuracy analysis. First, we bound the accuracy of EM for evenly spaced data in Theorem 10; then, we compare the accuracy to smooth sensitivity in Theorem 11.

Theorem 10 (Median Accuracy for Evenly Spaced Data). *Let the n elements in D be evenly spaced in $[0,1]$, i.e., $d_i = \frac{i}{n}$, let n be even (can be ensured with padding), and $\mathcal{R}$ consist of all subranges $[\frac{i}{n}, \frac{i+1}{n})$ for $i \in \{1, \dots, n\}$. Let μ be the median of D, and $\widehat{\mu}$ the output of $\mathsf{EM}_u^\epsilon(D, \mathcal{R})$. Then,*

$$Pr\left[|\mu - \widehat{\mu}| \le \frac{\gamma}{n}\right] \ge 1 - e^{-\epsilon(\gamma+1)}$$

for $\gamma \in \mathbb{N}$.

Proof. Evenly spaced D contains only unique elements and Theorem 2 from Section 5.1.3 applies (we assume even n to conform with Section 5). Thus, we can write the unnormalized probability mass of $\mathsf{EM}_u^\epsilon(D)$, i.e., denominator in Equation (2.1), as

$$N = \sum_{o \in \mathcal{R}} \exp(\epsilon u(D, o)) = 2 \sum_{i=0}^{\frac{n}{2}-1} \exp(-i\epsilon) = 2 \frac{e^\epsilon - e^{-\epsilon(\frac{n}{2}-1)}}{e^\epsilon - 1}.$$

The last equality comes from the fact that N is a geometric series [GKPL94, Eq. (2.25)].

We are interested in the unnormalized probability mass N_γ for outputs with error at most γ/n. This corresponds to rank difference of at most γ for evenly spaced data, and we can write

$$N_\gamma = 2 \sum_{i=0}^{\gamma} \exp(-i\epsilon) = 2 \frac{e^\epsilon - e^{-\epsilon\gamma}}{e^\epsilon - 1}.$$

Altogether, the (normalized) probability for outputs with error at most $\frac{\gamma}{n}$ is

$$\frac{N_\gamma}{N} = \frac{e^{\epsilon} - e^{-\epsilon\gamma}}{e^{\epsilon} - e^{-\epsilon(\frac{n}{2}-1)}} = \frac{1 - e^{-\epsilon(\gamma+1)}}{1 - e^{-\epsilon(\frac{n}{2}-2)}} \geq 1 - e^{-\epsilon(\gamma+1)}.$$

□

Next, we compare the exponential mechanism to the Laplace mechanism with smooth sensitivity. We use an idealized version of smooth sensitivity as in Section 3.6.1, i.e., ignoring constants that increase the noise [NRS07, Lemma 2.9] [MG20, Proposition 2]. We consider the probability that the absolute error is less than $1/n$, i.e., the gap between adjacent data elements.

Theorem 11. *For evenly spaced data and output domain $\mathcal{R}$ as in Theorem 10, absolute error less than $1/n$, and $\epsilon < 2$, the exponential mechanism is more likely to provide better accuracy than the Laplace mechanism with smooth sensitivity for the median.*

Proof. For evenly spaced data, smooth sensitivity s of the median is at most $\frac{1}{\epsilon n}$ [NRS07, Section 3.1] (whereas global sensitivity is 1). The probability that the Laplace mechanism adds noise less than $1/n$ is $p = 1 - \exp(-\epsilon^2)$. Note that for $X \sim \text{Laplace}(s/\epsilon)$ we have $\Pr[|X| < ts/\epsilon] = 1 - \exp(-t)$ [DR14, Fact 3.7]. With $s = \frac{1}{\epsilon n}$ and $ts/\epsilon = 1/n$ we get $t = \epsilon^2$ and arrive at the stated probability p. However, the same result with the exponential mechanism occurs with probability at least $q = 1 - \exp(-2\epsilon)$ according to Theorem 10 ($\gamma = 1$ leads to error less than $1/n$). Thus, for $\epsilon < 2$ we have $q > p$, i.e., the exponential mechanism is more likely to produce higher accuracy. □

Choice of Epsilon

Note that it is more likely to select a "good" subrange as it is to directly select a "good" element from the entire domain (as $k \ll |\mathfrak{D}|$). However, sequential (subrange) selections consumes ϵ_j per selection step j which adds up to a total privacy budget of $\epsilon = \sum_j \epsilon_j$ as described in Section 6.1.3. We now show how to choose ϵ_j to select the subrange containing the median in each iteration step with probability at least $1 - \beta$.

Theorem 12 (Choice of ϵ)**.** *Let $\mathcal{R} = \{R_1^j, \ldots, R_k^j\}$, where $R_i^j = [r_l, r_u)$ contains the median, and $n_{ij} = \min\{|\mathsf{rank}_D(\mu) - \mathsf{rank}_D(r_l)|, |\mathsf{rank}_D(r_u + 1) - \mathsf{rank}_D(\mu + 1)|\}$ is the minimum count of data elements in R_i^j smaller resp. larger than the median. Then, $\mathsf{EM}_u^\epsilon(D)$ over domain $\mathcal{R}$ selects R_i^j with probability at least $1 - \beta$ if*

$$\epsilon_j \geq \frac{\ln(k/\beta)}{n_{ij}}.$$

Proof. Ranges R_h^j without the median have a rank at least n_{ij} positions away from median rank. More formally,

$$\text{OPT} - u(D, R_h^j) \geq \left|\left(\frac{n}{2} \pm n_{ij}\right) - \frac{n}{2}\right| = n_{ij}.$$

According to Lemma 4 we have $\Pr\left[n_{ij} \geq \frac{\ln|\mathcal{R}|+t}{\epsilon_j}\right] \leq \exp(-t)$. Thus, for $\epsilon_j \geq \frac{\ln|\mathcal{R}|+t}{n_{ij}}$ the probability that any range R_h^j is selected is at most $\exp(-t)$. We have $k = |\mathcal{R}|$ and setting $\beta = \exp(-t)$ concludes the proof. □

Parameter ϵ_j is undefined for $n_{ij} = 0$, i.e., when the median is a range endpoint. However, an undefined ϵ_j can be avoided by using an additional discretization of the domain, with different subrange endpoints, and switching to it if a (differentially private) check suggests $n_{ij} = 0$ [DL09]. Note that the exact value of n_{ij} is data-dependent. E.g., for the uniform distribution $n_{ij} \approx |D|/k^j$. A differentially private n_{ij} can be efficiently computed by distributed sum protocols [DKM+06, GX17, TKZ16, RN10] as it is just a count of data elements. However, a differentially private count also consumes a portion of the privacy parameter. For low epsilon (e.g., $\epsilon = 0.1$) we want to use the entire privacy budget on the actual median selection to achieve high accuracy. Thus, we use a heuristic in our evaluation: larger subranges, that hold exponentially more elements, receive exponentially smaller portions ϵ_j of the privacy budget (see Section 6.3 for details).

6.2 MPC for DP Median

In the following, we describe details of our protocol EM*, which implements ideal functionality $\mathcal{F}_{\mathsf{EM}^*}$, analyse its running time and security.

On a high-level, our protocol recursively selects the best subrange until the DP median is found: First, each party locally *evaluates* a utility score (or weight) for each subrange. They *combine* their results into a global result. Then, they *select* a subrange based on the combined result. We use upper case letters to denote arrays in our protocol, and $A[j]$ denotes the j^{th} element in array A. Our protocol EM* operates on integers as well as floating point numbers whereas GM* operates only on integers. We briefly recall the number representation described in Section 2: a floating-point number f is expressed as $(1-2s)(1-z) \cdot v \cdot 2^x$ with sign bit s set when the value is negative, zero bit z only set when the value is zero, l_v-bit significand v, and l_x-bit exponent x. The sharing $\langle f \rangle_{\mathsf{FL}}$ of a floating-point number f is a 4-tuple $(\langle v \rangle, \langle x \rangle, \langle s \rangle, \langle z \rangle)$ and we use subscripts to refer to parts of a sharing, e.g., $f.v$ refers to the significand v of f.

The basic MPC protocols used in our protocol are detailed in Table 2.3 in Section 2.1.6. Recall, as a default we assume integer operations and use subscript FL to highlight basic MPC protocols operating on floating-point numbers, e.g., Add denotes addition on integers while $\mathsf{Add}_{\mathsf{FL}}$ denotes its floating-point equivalent.

6.2.1 Subrange Selection

On a high level, protocol EM*, implemented in Algorithm 9, computes selection weights for possible outputs (via Algorithm 10) and selects an output according to these weights (via Algorithm 11 or 12). We assume that the domain $\mathfrak{D}$ and combined data size n are known to all parties, however, the latter can be hidden via padding [AMP10]. Recall, that efficient weight computation and selection from a large domain are the main challenges for our secure exponential mechanism. Straightforward selection over all domain elements is linear in the size of $\mathfrak{D}$. To achieve a running time sublinear in the size of $\mathfrak{D}$ we select subranges instead: Algorithm 9 selects one of k subranges based on their median utility. The selected subrange is recursively divided into k subranges until the last subrange, after at most $\lceil \log_k |\mathfrak{D}| \rceil$ iterations, contains only one element: the differentially private median[3]. Alternatively, one can use fewer selection steps s and select an element from the last subrange at uniform random (line 15 in Algorithm 9). We discuss the running time vs. accuracy trade-offs of reduced selection steps in Section 6.3. We implement selection with *inverse transform sampling* (Section 2.2.4) via binary search in

[3] To simplify presentation, assume that $\log_k |\mathfrak{D}|$ is an integer. Otherwise the last subrange might contain less than k elements, and fewer weight computations are needed in the last step.

Algorithm 9 EM* iteratively selects smaller subranges containing DP median via EM.

Input: Number of subranges k, size n of combined data D, number of selection steps $s \in [1, \lceil \log_k |\mathfrak{D}| \rceil]$, and $(\epsilon_1, \ldots, \epsilon_s)$. Data domain $\mathfrak{D}$ is known to all parties.
Output: Differentially private median of D.

```
1: $r_l, r_u \leftarrow 0, |\mathfrak{D}|$
2: for $j \leftarrow 1$ to $s$ do
3:     $r_\# \leftarrow \max\{1, \lfloor \frac{r_u - r_l}{k} \rfloor\}$
4:     $k \leftarrow \min\{k, r_u - r_l\}$
5:     Define array $W$ of size $k$
6:     if $\epsilon_j = \ln(2)/2^d$ for some integer $d$ then
7:         $\langle W \rangle_{\mathsf{FL}} \leftarrow \mathsf{Weights}^{\ln(2)/2^d}(r_l, r_u, r_\#, k, n, d)$  // Algorithm 11
8:     else
9:         $\langle W \rangle_{\mathsf{FL}} \leftarrow \mathsf{Weights}^*(r_l, r_u, r_\#, k, n, \epsilon_j)$  // Algorithm 12
10:    end if
11:    $i \leftarrow \mathsf{Select}(\langle W \rangle_{\mathsf{FL}})$  // Algorithm 10
12:    $r_l \leftarrow r_l + (i-1) \cdot r_\#$
13:    $r_u \leftarrow r_l + r_\#$ if $i < k$
14: end for
15: return Uniform random element in $[\mathfrak{D}[r_l], \mathfrak{D}[r_u])$
```

Algorithm 10 Select samples range index according to its selection weights.

Input: List $\langle W \rangle_{\mathsf{FL}}$ of weights with size k.
Output: Index $j \in [1, k]$ sampled according to $\langle W \rangle_{\mathsf{FL}}$.

```
1: Define array $M$ of size $k$  // Probability mass
2: $\langle M[1] \rangle_{\mathsf{FL}} \leftarrow \langle W[1] \rangle_{\mathsf{FL}}$
3: for $j \leftarrow 2$ to $k$ do
4:     $\langle M[j] \rangle_{\mathsf{FL}} \leftarrow \mathsf{Add}_{\mathsf{FL}}(\langle W[j] \rangle_{\mathsf{FL}}, \langle M[j-1] \rangle_{\mathsf{FL}})$
5: end for
6: $\langle t \rangle \leftarrow \mathsf{Rand}(b)$  // Bit-length $b$
7: $\langle f \rangle_{\mathsf{FL}} \leftarrow \mathsf{Int2FL}(\langle t \rangle)$
8: $\langle x \rangle \leftarrow \mathsf{Sub}(\langle f.x \rangle, \langle b \rangle)$
9: $\langle f \rangle_{\mathsf{FL}} \leftarrow (\langle f.v \rangle, \langle x \rangle, \langle f.z \rangle, \langle f.s \rangle)$
10: $\langle r \rangle_{\mathsf{FL}} \leftarrow \mathsf{Mul}_{\mathsf{FL}}(\langle M[k] \rangle_{\mathsf{FL}}, \langle f \rangle_{\mathsf{FL}})$
11: $i_l \leftarrow 1; i_u \leftarrow k$
12: while $i_l < i_u$ do
13:     $i_m \leftarrow \left\lfloor \frac{i_l + i_u}{2} \right\rfloor$
14:     $\langle c \rangle \leftarrow \mathsf{LT}_{\mathsf{FL}}(\langle M[i_m] \rangle_{\mathsf{FL}}, \langle r \rangle_{\mathsf{FL}})$
15:     $c \leftarrow \mathsf{Rec}(\langle c \rangle)$
16:     $i_l \leftarrow i_m + 1$ if $c = 1$ else $i_u \leftarrow i_m$
17: end while
18: return $i_l$
```

Algorithm 10 similar to Eigner et al. [EKM+14]. Inverse transform sampling (as detailed in Section 2.2.4) uses the uniform distribution to realize any distribution based on its cummulative distribution function. Formally, one draws $r \in (0, 1]$ at uniform random and outputs the first $R_j \in \mathcal{R}$ with $\sum_{i=1}^{j-1} \Pr\left[\mathsf{EM}_u^\epsilon(D, \mathcal{R}) = R_i\right] \leq r < \sum_{i=1}^{j} \Pr\left[\mathsf{EM}_u^\epsilon(D) = R_i\right]$. Recall, we compute unnormalized probabilities (weights), which do not require division for normalization, thus, reducing computation complexity. To use weights instead of probabilities in inverse transform sampling we only need to multiply r with normalization $N = \sum_{o \in \mathcal{R}} \exp(u(D, o)\epsilon)$ (lines 6–10 in Algorithm 10).

We use decomposable utility functions to combine local evaluations over each party's data into a global utility score for the joint data. Next, we present three solutions to efficiently compute weights for decomposable utility functions.

Algorithm 11 $\mathsf{Weights}^{\ln(2)/2^d}$ computes weights based on local ranks.

Input: Range $[r_l, r_u)$, subrange size $r_\#$, number k of subranges, data size n, and parameter $d \in \{0,1\}$. Subrange ranks $\langle \mathsf{rank}_{D_p}(\cdot)\rangle$ are input by each party $p \in \{1,\ldots,m\}$.
Output: List of weights.

```
1: Define array $R$ of size $k+1$, array $W$ of size $k$; initialize $R$ with zeros
2: for $p \leftarrow 1$ to $m$ do  //Get input from each party
3:   for $j \leftarrow 1$ to $k$ do  //Divide range into $k$ subranges
4:     $i_l \leftarrow r_l + (j-1)\cdot r_\#$
5:     $\langle R[j]\rangle \leftarrow \mathsf{Add}(\langle R[j]\rangle, \langle \mathsf{rank}_{D_p}(\mathfrak{D}[i_l])\rangle)$
6:   end for
7:   $\langle R[k+1]\rangle \leftarrow \mathsf{Add}(\langle R[k+1]\rangle, \langle \mathsf{rank}_{D_p}(\mathfrak{D}[r_u])\rangle)$
8: end for
9: for $j \leftarrow 1$ to $k$ do
10:   $\langle u_{\text{upper}}\rangle \leftarrow \mathsf{Sub}(\langle R[j+1]\rangle, \langle \frac{n}{2}\rangle)$
11:   $\langle u_{\text{lower}}\rangle \leftarrow \mathsf{Sub}(\langle \frac{n}{2}\rangle, \langle R[j]\rangle)$
12:   $\langle c_{\text{upper}}\rangle \leftarrow \mathsf{LT}(\langle R[j+1]\rangle, \langle \frac{n}{2}\rangle)$
13:   $\langle c_{\text{lower}}\rangle \leftarrow \mathsf{LT}(\langle \frac{n}{2}\rangle, \langle R[j]\rangle)$
14:   $\langle t\rangle \leftarrow \mathsf{Mux}(\langle u_{\text{upper}}\rangle, \langle 0\rangle, \langle c_{\text{upper}}\rangle)$
15:   $\langle u\rangle \leftarrow \mathsf{Mux}(\langle u_{\text{lower}}\rangle, \langle t\rangle, \langle c_{\text{lower}}\rangle)$
16:   if $d = 0$ then
17:     $\langle W[j]\rangle_{\mathsf{FL}} \leftarrow (\langle 2\rangle, \langle u\rangle, \langle 0\rangle, \langle 0\rangle)$  //float $\langle 2^u\rangle$
18:   else
19:     $\langle t\rangle \leftarrow \mathsf{Trunc}(\langle u\rangle, d)$
20:     $\langle e\rangle_{\mathsf{FL}} \leftarrow (\langle 2\rangle, \langle t\rangle, \langle 0\rangle, \langle 0\rangle)$
21:     $\langle c\rangle \leftarrow \mathsf{Mod2m}(\langle u\rangle, d)$
22:     $\langle s\rangle_{\mathsf{FL}} \leftarrow \mathsf{Mux}_{\mathsf{FL}}(\langle 1\rangle_{\mathsf{FL}}, \langle \sqrt{2}\rangle_{\mathsf{FL}}, \langle c\rangle)$
23:     $\langle W[j]\rangle_{\mathsf{FL}} \leftarrow \mathsf{Mul}_{\mathsf{FL}}(\langle e\rangle_{\mathsf{FL}}, \langle s\rangle_{\mathsf{FL}})$
24:   end if
25: end for
26: return $\langle W\rangle_{\mathsf{FL}}$
```

6.2.2 $\mathsf{Weights}^{\ln(2)}$

We implement $\mathsf{Weights}^{\ln(2)}$ as a special case of our approach $\mathsf{Weights}^{\ln(2)/2^d}$ in Algorithm 11 (with $d = 0$ in line 16). Here, parties locally compute *ranks* which are combined into global utility scores. Weights for these scores use a fixed ϵ of $\ln(2)$ to let us compute 2^u instead of $\exp(\epsilon \cdot u)$. Solutions for secure exponentiation of 2^u exist where u is an integer or a float [DFK+06, AS19, Kam15, ABZS13]. When u is an integer (resp. a float) the result 2^u is an integer (resp. float) as well. The complexity of the integer-based solution is linear in the bit-length of u, however, this is not sufficient for us: Recall, that the utility is based on ranks, i.e., counts of data elements, thus u can be roughly as large as the size of the data. An integer representation of 2^u has bit-length u, which is potentially unbounded. Eigner et al. [EKM+14] use the float-based solution from [ABZS13] but we present a more efficient computation in the following. Although our exponent u is an integer, we do not require the result to be an integer as well. We use the representation of floating point numbers as a 4-tuple to construct a new float to represent 2^u as $(2, u, 0, 0)$, where sign and zero bit are unset, as 2^u cannot be negative or zero. Note that we require no interaction as each party can construct such a float with their share of u. Also, a naive approach requires $2k$ total inputs per party (one per endpoint per k ranges). However, with half-open ranges $[r_l^i, r_u^i)$ in each step i, they overlap for $i > 1$: $r_u^{i-1} = r_l^i$. Thus, the parties only input $k+1$ ranks (Algorithm 11 lines 5, 7).

Algorithm 12 $\mathsf{Weights}^*$ computes (global) weights based on local weights.

```
Input: Range $[r_l, r_u)$, subrange size $r_\#$, number $k$ of subranges, data size $n$, and $\epsilon$. Subrange
    weights $\langle e^{\epsilon(\cdot)} \rangle$ are input by each party $p \in \{1, \dots, m\}$.
Output: List of weights.
 1: Define arrays $W^l, W^u, W$ of size $k$; initialize $W^l, W^u$ with ones
 2: for $p \leftarrow 1$ to $m$ do  // Get input from each party
 3:   for $j \leftarrow 1$ to $k$ do  // Divide range into $k$ subranges
 4:     $i_l \leftarrow r_l + (j-1) \cdot r_\#$
 5:     $i_u \leftarrow r_u$ if $j = k$ else $r_l + j \cdot r_\#$
 6:     $\langle W^l[j] \rangle_{\mathsf{FL}} \leftarrow \mathsf{Mul}_{\mathsf{FL}}(\langle W^l[j] \rangle_{\mathsf{FL}}, \langle e^{\epsilon\left(\frac{|D_p|}{2} - \mathsf{rank}_{D_p}(\mathfrak{D}[i_l])\right)} \rangle_{\mathsf{FL}})$
 7:     $\langle W^u[j] \rangle_{\mathsf{FL}} \leftarrow \mathsf{Mul}_{\mathsf{FL}}(\langle W^u[j] \rangle_{\mathsf{FL}}, \langle e^{\epsilon\left(\mathsf{rank}_{D_p}(\mathfrak{D}[i_u]) - \frac{|D_p|}{2}|\right)} \rangle_{\mathsf{FL}})$
 8:   end for
 9: end for
10: for $j \leftarrow 1$ to $k$ do
11:   $\langle c_u \rangle \leftarrow \mathsf{LT}_{\mathsf{FL}}(\langle W^u[j] \rangle_{\mathsf{FL}}, \langle 1 \rangle_{\mathsf{FL}})$
12:   $\langle c_l \rangle \leftarrow \mathsf{LT}_{\mathsf{FL}}(\langle W^l[j] \rangle_{\mathsf{FL}}, \langle 1 \rangle_{\mathsf{FL}})$
13:   $\langle t \rangle_{\mathsf{FL}} \leftarrow \mathsf{Mux}_{\mathsf{FL}}(\langle W^u[j] \rangle_{\mathsf{FL}}, \langle 1 \rangle_{\mathsf{FL}}, \langle c_u \rangle)$
14:   $\langle W[j] \rangle_{\mathsf{FL}} \leftarrow \mathsf{Mux}_{\mathsf{FL}}(\langle W^l[j] \rangle_{\mathsf{FL}}, \langle t \rangle_{\mathsf{FL}}, \langle c_l \rangle)$
15: end for
16: return $\langle W \rangle_{\mathsf{FL}}$
```

6.2.3 Weights$^{\ln(2)/2^d}$

Next, we generalize the weight computation to support $\epsilon = \ln(2)/2^d$ for integers $d \geq 1$. To illustrate our approach, we implement $\mathsf{Weights}^{\ln(2)/2^d}$ in Algorithm 11 for $d = 1$, and describe the approach for any integer d: Recall, our goal is to compute the weight $\exp(\epsilon u)$ with efficient MPC protocols. As we can efficiently compute $2^{\epsilon u}$ if ϵu is an integer, we approximate the weight by truncating ϵu to an integer before exponentiation with base 2. To avoid a loss of precision we correct this approximation with a multiplicative term based on the truncated remainder. More formally, with ϵ as above the weight for u is

$$2^{u/2^d} = 2^{\lfloor u/2^d \rfloor} \cdot 2^{(u \mod 2^d)/2^d}.$$

First, we compute $2^{\lfloor u/2^d \rfloor}$ (lines 19–21 in Algorithm 12). Then, we multiply this with one of 2^d constants of the form $2^{(u \mod 2^d)/2^d}$. E.g., for $d = 1$, we either use 1, if u is even, or $\sqrt{2}$ otherwise (line 22). The constants themselves are not secret and can be pre-computed. Which constant was selected, leaks the last d bits from u, thus, we choose them securely.

6.2.4 Weights*

We implement $\mathsf{Weights}^*$ in Algorithm 12. To allow arbitrary values for ϵ we avoid costly secure exponentiation for weight computation altogether: Utility u, decomposable w.r.t. u', allows for efficient combination of local weights for D_i, input by the parties, into global weights for D via multiplication as described in Section 6.1.2).

6.2.5 GM*

Algorithm 13, denoted GM^*, iteratively calls the Gumbel mechanism GM which outputs the subrange index with highest noisy utility score. The subrange iteration code is the same as for EM^*

Algorithm 13 GM* iteratively selects smaller subranges containing DP median via GM.

Input: Number of subranges k, size n of combined data D, number of selection steps $s \in [1, \lceil \log_k |\mathfrak{D}| \rceil]$, and privacy budget $(\epsilon_1, \ldots, \epsilon_s)$. Subrange ranks $\langle \mathsf{rank}_{D_p}(\cdot) \rangle$ and distributed noises $\langle \rho_p \rangle$ are input by each party $p \in \{1, \ldots, m\}$. Data domain $\mathfrak{D}$ is known to all parties.
Output: Differentially private median of D.

```
1:  $r_l, r_u \leftarrow 0, |\mathfrak{D}|$
2:  for $i \leftarrow 1$ to $s$ do //Get ranks and distributed noise from each party
3:    $r_\# \leftarrow \max\{1, \lfloor \frac{r_u - r_l}{k} \rfloor\}$
4:    $k \leftarrow \min\{k, r_u - r_l\}$
5:    Define arrays: $R$ of size $k+1$ and $S, N$ of size $k$; initialize $R, N$ with zeros
6:    for $p \leftarrow 1$ to $m$ do //Get ranks and distributed noise from each party
7:      for $j \leftarrow 1$ to $k$ do //Divide range into $k$ subranges
8:        $i_l \leftarrow r_l + (j-1) \cdot r_\#$
9:        $\langle R[j] \rangle \leftarrow \mathsf{Add}(\langle R[j] \rangle, \langle \mathsf{rank}_{D_p}(\mathfrak{D}[i_l]) \rangle)$ //Combine local ranks
10:       $\langle N[j] \rangle \leftarrow \mathsf{Add}(\langle N[j] \rangle, \langle \rho_p^j(\epsilon_i) \rangle)$ //Combine partial noises
11:     end for
12:     $\langle R[k+1] \rangle \leftarrow \mathsf{Add}(\langle R[k+1] \rangle, \langle \mathsf{rank}_{D_p}(\mathfrak{D}[r_u]) \rangle)$
13:   end for
14:   for $j \leftarrow 1$ to $k$ do
15:     $\langle u_u \rangle \leftarrow \mathsf{Sub}(\langle R[j+1] \rangle, \langle \frac{n}{2} \rangle)$
16:     $\langle u_l \rangle \leftarrow \mathsf{Sub}(\langle \frac{n}{2} \rangle, \langle R[j] \rangle)$
17:     $\langle c_u \rangle \leftarrow \mathsf{LT}(\langle R[j+1] \rangle, \langle \frac{n}{2} \rangle)$
18:     $\langle c_l \rangle \leftarrow \mathsf{LT}(\langle \frac{n}{2} \rangle, \langle R[j] \rangle)$
19:     $\langle t \rangle \leftarrow \mathsf{Mux}(\langle u_u \rangle, \langle 0 \rangle, \langle c_u \rangle)$
20:     $\langle S[j] \rangle \leftarrow \mathsf{Mux}(\langle u_l \rangle, \langle t \rangle, \langle c_l \rangle)$ //Utility score
21:   end for
22:   Initialize $\langle u_{\max} \rangle \leftarrow \mathsf{Add}(\langle S[1] \rangle, \langle N[1] \rangle)$ and $\langle j_{\arg\max} \rangle \leftarrow \langle 1 \rangle$
23:   for $j \leftarrow 2$ to $k$ do
24:     $\langle u_{\text{noisy}} \rangle \leftarrow \mathsf{Add}(\langle S[j] \rangle, \langle N[j] \rangle)$
25:     $\langle c \rangle \leftarrow \mathsf{LT}(\langle u_{\text{noisy}} \rangle, \langle u_{\max} \rangle)$
26:     $\langle u_{\max} \rangle \leftarrow \mathsf{Mux}(\langle u_{\max} \rangle, \langle u_{\text{noisy}} \rangle, \langle c \rangle)$
27:     $\langle j_{\arg\max} \rangle \leftarrow \mathsf{Mux}(\langle j_{\arg\max} \rangle, \langle j \rangle, \langle c \rangle)$
28:   end for
29:   $j_{\arg\max} \leftarrow \mathsf{Rec}(\langle j_{\arg\max} \rangle)$
30:   $r_l \leftarrow r_l + (j_{\arg\max} - 1) \cdot r_\#$
31:   $r_u \leftarrow r_l + r_\#$ if $j_{\arg\max} < k$
32: end for
33: return Uniform random element in $[\mathfrak{D}[r_l], \mathfrak{D}[r_u])$
```

implemented in Algorithm 9 (i.e., they share the same first and last four lines), and the utility scores are computed as in Algorithm 11 (compare lines 10–15 in Algorithm 11 with lines 15–20 in Algorithm 13). The main difference is that now each party additionally inputs distributed noise values (Section 2.2.5). We let $\rho_p^j(\epsilon_i)$ denote the distributed noise of party p for subrange j parameterized with ϵ_i in selection step i. These noise values are scaled, truncated integers and subrange ranks are scaled with the same scaling factor.

Furthermore, to implement the exponential mechanism EM, we compute selection weights $\exp(\epsilon \cdot u)$ per utility score u and sample an output via inverse transform sampling. For the Gumbel mechanism GM, on the other hand, we have to find the element whose noisy utility score is the largest. Computing argmax for GM requires k steps compared to the (at most) $\lceil \log_2 k \rceil$ steps to sample from EM (binary search in Algorithm 10). However, the former operations can be implemented with (scaled) integers, whereas the latter always requires floating point numbers (due to potentially large exponents).

6.2.6 Precision and Privacy

As mentioned in Section 3.4, Ilvento [Ilv20] showed that limited machine precision can lead to privacy violations when implementing the exponential mechanism. Interestingly, the suggested mitigations are similar to our efficient secure computation. Our implementation is based on an integer utility function and $\mathsf{Weights}^{\ln(2)}$ uses base 2 for efficiency reasons and is not vulnerable to such attacks. We can strengthen $\mathsf{Weights}^{\ln(2)/2^d}$, with $\epsilon = \ln(2)/2^d$, by using randomized rounding for non-integer utilities [Ilv20, Section 3.2.2] if we omit $1/2^d$ from ϵ and include it as a factor in the utility definition (making the utility non-integers). For $\mathsf{Weights}^*$, which supports arbitrary ϵ, careful choices for ϵ mitigate attacks on limited precision (Section 3.4). Our protocol GM^* operates on integers and does not suffer from privacy issues due to limited precision of floating-point numbers.

6.2.7 Running Time Complexity

Next, we analyse the running time of EM^* w.r.t. MPC protocols from Table 2.3 in Section 2.1.6 (omitting non-interactive addition/subtraction):

Theorem 13. EM^* *with* $\mathsf{Weights}^{\ln(2)}$ *or* $\mathsf{Weights}^{\ln(2)/2^d}$ *requires* $O(k\lceil\log_k|\mathfrak{D}|\rceil)$ *MPC protocol calls, with* $\mathsf{Weights}^*$ *we require* $O(mk\lceil\log_k|\mathfrak{D}|\rceil)$*. Note that complexity of these MPC protocols is at most* $O(l_v \log l_v + l_x)$ *for bit-lengths* l_v, l_x *as detailed in Table 2.3 in Section 2.1.6.*

Proof. EM^* invokes the weight computation and Select at most $\lceil\log_k|\mathfrak{D}|\rceil$ times. An invocation of $\mathsf{Weights}^{\ln(2)}$ or $\mathsf{Weights}^{\ln(2)/2^d}$ performs k truncations Trunc, $2k$ comparisons LT and $2k$ selections Mux. Additionally, $\mathsf{Weights}^{\ln(2)/2^d}$ also requires one truncation Trunc, modulo $\mathsf{Mod2m}$, float selection $\mathsf{Mux}_{\mathsf{FL}}$ and float multiplication $\mathsf{Mul}_{\mathsf{FL}}$. Weight computation via $\mathsf{Weights}^*$ requires $2km$ float multiplications $\mathsf{Mul}_{\mathsf{FL}}$, $2k$ float comparisons $\mathsf{LT}_{\mathsf{FL}}$ and $2k$ float selections $\mathsf{Mux}_{\mathsf{FL}}$. Each invocation of Select requires $k-1$ float additions $\mathsf{Add}_{\mathsf{FL}}$, only one random draw Rand, conversion $\mathsf{Int2FL}$ and float multiplication $\mathsf{Mul}_{\mathsf{FL}}$. Also, Select performs at most $\log_2(k)$ comparisons $\mathsf{LT}_{\mathsf{FL}}$ and share reconstruction steps during binary search. □

Analogously, we analyse the running time of GM^* as the number of (interactive) calls to MPC protocols.

Theorem 14. GM^* *requires* $O(k\lceil\log_k|\mathfrak{D}|\rceil)$ *MPC protocol calls, Note that complexity of these MPC protocols is at most* $O(l)$ *for* l*-bit integers as detailed in Table 2.3 in Section 2.1.6.*

Proof. The first loop performs at most $\lceil\log_k|\mathfrak{D}|\rceil$ iterations with a single reconstruction (Rec) per iteration. Nested in the first loop are two sequential loops which perform at most k iterations. Recall, we omit addition as it is interaction-free. In total, GM^* performs $O(k\lceil\log_k|\mathfrak{D}|\rceil)$ iterations. Each of these iterations requires 3 comparisons LT and 4 selections Mux leading to $O(k\lceil\log_k|\mathfrak{D}|\rceil)$ operations in total. □

6.2.8 Security

Recall, we consider the semi-honest model introduced by Goldreich [Gol09] where corrupted protocol participants do not deviate from the protocol but gather everything created during the run of the protocol. Our protocols EM^*, GM^* consists of multiple subroutines realized with MPC protocols listed in Table 2.3. We rely on the well-known composition theorem [Gol09, Section 7.3.1] for our security analysis: MPC protocols using an ideal functionality remain secure if the ideal functionality is replaced with an MPC protocol implementing the same functionality. We implement such ideal functionality with the maliciously secure SCALE-MAMBA framework [AKR+20]. Our protocol performs multiple subrange selections and each selection round is maliciously secure. Overall, we only provide semi-honest security as malicious adversaries can deviate from inputs provided in previous rounds. We later show how to extend our protocol to malicious adversaries, but first we proof semi-honest security for EM*:

Theorem 15. *Protocol* EM* *realizes* $\mathcal{F}_{\mathsf{EM}^*}$ *in the presence of semi-honest adversaries.*

To prove semi-honest security we show the existence of a simulator Sim such that the distributions of the protocol transcript EM* is computationally indistinguishable from a simulated transcript using $\mathcal{F}_{\mathsf{EM}^*}$ produced in an "ideal world" with a trusted third party (see Section 4.1).

Proof. Simulator Sim produces a transcript for $\mathsf{real}_{\mathsf{EM}^*}$ as follows: As we operate on secret shares, denoted with $\langle\cdot\rangle$, which look random to the parties [EKR+18], Sim replaces all secret shares with random values to create $\mathtt{VIEW}_i$. Likewise, the secret-shared output of the weight computations (Algorithm 11 and 12) are replaced with randomness. Sim can simulate Algorithm 10 by recursively splitting $\mathfrak{D}$ into k subranges, and outputting the subrange containing $\widehat{\mu}$ in each selection step. Finally, Sim outputs a uniform random element from the last subrange (Algorithm 9). Altogether, a semi-honest adversary cannot learn more than the (ideal-world) simulator as this information is sufficient to produce a transcript of our (real-world) protocol. □

Theorem 16. *Protocol* GM* *realizes* $\mathcal{F}_{\mathsf{GM}^*}$ *in the presence of semi-honest adversaries.*

Proof. As before, Sim replaces all secret shares with random values to create $\mathtt{VIEW}_i$ and the simulation proceeds similar to EM* as GM* and EM* share most of their code: GM* and EM* have the same subrange iteration and same utility scoring. The main difference are the additionally provided partial noises, the noise aggregation, and finding the maximum noisy score. However, the in- and outputs of these operations are all secret shared as well, and Sim replaces them with randomness. □

From Semi-honest to Malicious

For malicious adversaries, we need to ensure consistency between rounds similar to Aggarwal et al. [AMP10], who securely compute the (non-DP) median via comparison-based pruning rounds (see Section 5.1.5). Informally, we have two consistency constraints: First, valid rank inputs must be monotone within a step. Second, for consistency between steps, valid inputs are contained in the subrange output in the previous step. Formally, let $\{R_1^i, \ldots, R_k^i\}$ denote the set of subranges in the i^{th} step of EM* and let l_j^i, u_j^i denote the lower resp. upper range endpoint of R_j^i. Then, $\mathsf{rank}_{D_p}(l_1^i) \leq \mathsf{rank}_{D_p}(l_2^i) \leq \cdots \leq \mathsf{rank}_{D_p}(l_k^i) \leq \mathsf{rank}_{D_p}(u_k^i)$ describes monotone input in step i for party p. Consistency between step i and $i+1$, if the j^{th} range was selected, is expressed as $\mathsf{rank}_{D_p}(l_1^{i+1}) = \mathsf{rank}_{D_p}(l_j^i)$ and $\mathsf{rank}_{D_p}(u_k^{i+1}) = \mathsf{rank}_{D_p}(u_j^i)$. In other words, the subrange output in the previous step is used in the current step. Analogously, we can enforce consistency for weights as they are based on rank values. Note that malicious users have limited influence on a rank-based statistic: a collection of t malicious parties can change the output's rank by at most $\pm t$.

6.2.9 Scaling to Many Parties

Recall, we distinguish two sets of parties: *Input parties* send shares of their input to *computation parties* which run the secure computation on their behalf. The computation parties can be a subset of the input parties or some AWS instances executing our protocol. This scales nicely as the number of computation parties is independent of the number of input parties and can be constant, e.g., 3. In our evaluation in Section 6.3, $m \in \{3, 6, 10\}$ computation parties perform the computation for 10^6 input parties, each holding a single datum. Addition suffices for $\mathsf{Weights}^{\ln(2)}$ and $\mathsf{Weights}^{\ln(2)/2^d}$ to combine local rank values into a global rank. Addition is essentially "free" as it requires no interaction between the computation parties. For $\mathsf{Weights}^*$ we require multiplication to combine the local weights, which requires interaction during the preprocessing step. However, $\log n$ rounds suffice to combine the inputs by building a tree of pairwise multiplications with 2^i multiplications at level i [ABZS13].

6.3 Evaluation

We implementation our protocols with SCALE-MAMBA [AKR+20] using Shamir secret sharing with a 128-bit modulus and honest majority ($\lceil m/2 \rceil - 1$ corrupted parties). SCALE-MAMBA's floating point numbers (`sfloat`) are associated with a statistical security parameter κ satisfying $\kappa < b - 2 \cdot l_v$ where b is the bit-length of the modulus and l_v is the bit-length of the significand. Security with $\kappa = 40$ is the default for $b = 128$ and we use $l_v = 40$ in our evaluation, to support large utility values. Next, we evaluate the running time, privacy budget and accuracy of our solution.

6.3.1 Running Time

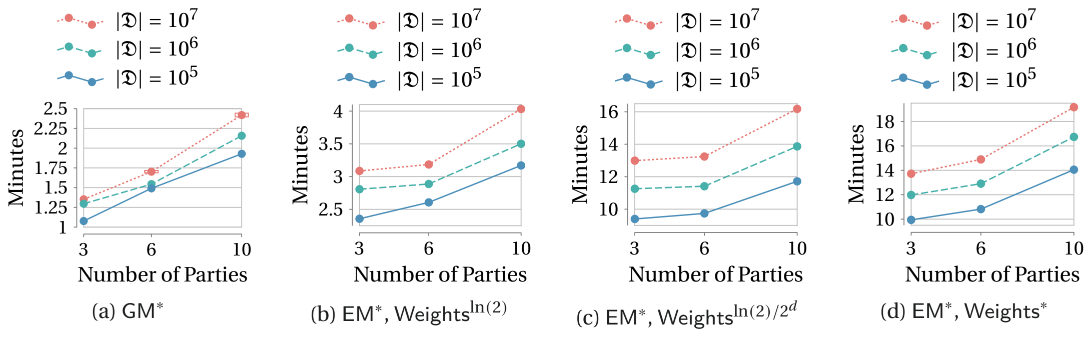
For our evaluation we used t2.medium instances from Amazon Web Services (AWS) with 2GB RAM, 4 vCPUs [Ama20b] and the Open Payments data set from the Centers for Medicare & Medicaid Services (CMS) [CMS17] as in Section 5.3. Our evaluation uses 10^6 records from the Open Payments data set, however, our approach scales to any data set size as we consider domain subranges. We used the maximum number of selection steps, i.e., $s = \lceil \log_k |\mathfrak{D}| \rceil$, with $k = 10$ ranges per step. We evaluated the average running time of 20 runs of the entire protocol EM^*, i.e., offline as well as online phase, and evaluated in a LAN and a WAN.

LAN

We measured running time for 3 parties in a LAN with 1 Gbits/s bandwidth in Table 6.2 to compare our protocols to Eigner et al. [EKM+14] who only report LAN running times. Eigner et al. [EKM+14] evaluated their protocol with a sum utility function on a machine equipped with a 3.20 GHz Intel i5 CPU and 16 GB RAM. They are linear in the size of the domain and compute weights for a very small domain of only 5 elements. We, on the other hand, are sublinear in the size of the domain as we compute weights per subrange and use efficient alternatives to costly secure exponentiation. We evaluated domain sizes at least 5 order of magnitudes larger than [EKM+14] with comparable running times: They compute weights per elements and require around 42 seconds for $|\mathfrak{D}| = 5$, whereas our protocol EM^* with $\mathsf{Weights}^{\ln(2)}$ / $\mathsf{Weights}^{\ln(2)/2^d}$ / $\mathsf{Weights}^*$ runs in approximately 11 / 33 / 64 seconds and GM^* runs in approximately 28 seconds for $|\mathfrak{D}| = 10^5$. Overall, our running time for is below the running time of Eigner et al. on rather modest t2.medium instances (4 vCPUs, 2 GB RAM) for domain size $|\mathfrak{D}| = 10^6$ except for EM^* with $\mathsf{Weights}^*$. Even if we also consider weights per element (i.e., subrange size 1) for any decomposable utility function our protocols compute at least 6 times more weights per second on t2.medium instances. (E.g., for $k = 10, |\mathfrak{D}| = 10^5$ and $\mathsf{Weights}^*$ we compute 50 weights in 64.3

Protocol	$\lvert\mathfrak{D}\rvert$	*Running time*
Eigner et al. [EKM+14]	5	42.3 s
GM*	10^5	28.3 ± 1.9 s (18.5 ± 2.1 s)
	10^6	31.6 ± 2.2 s (22.3 ± 2.2 s)
	10^7	38.4 ± 2.4 s (26.1 ± 2.3 s)
EM* & Weights$^{\ln(2)}$	10^5	11.3 ± 0.8 s (7.7 ± 0.7 s)
	10^6	13.5 ± 2.2 s (9.2 ± 1.1 s)
	10^7	15.4 ± 1.4 s (10.7 ± 1.0 s)
EM* & Weights$^{\ln(2)/2^d}$, $d = 2$	10^5	33.7 ± 3.4 s (23.6 ± 1.3 s)
	10^6	39.8 ± 3.7 s (27.8 ± 1.3 s)
	10^7	46.8 ± 3.5 s (31.4 ± 1.3 s)
EM* & Weights*	10^5	64.3 ± 3.0 s (41.6 ± 1.4 s)
	10^6	77.3 ± 3.0 s (52.4 ± 1.8 s)
	10^7	91.8 ± 4.2 s (61.1 ± 2.7 s)

Table 6.2: LAN: running times for 3 parties in a 1 Gbits/s network for this work and Eigner et al. [EKM+14]. We report the average of 20 runs with 95% confidence intervals on t2.medium instances with 4 vCPUs, 2 GB RAM (and r4.2xlarge instances with 8 vCPUs, 61 GB RAM in parenthesis). Eigner et al. [EKM+14] evaluated on a 3.20 GHz (Intel i5), 16 GB RAM machine.



(a) GM* (b) EM*, Weights$^{\ln(2)}$ (c) EM*, Weights$^{\ln(2)/2^d}$ (d) EM*, Weights*

Figure 6.4: WAN: running times of GM* and EM*– with weight computation subroutines Weights$^{\ln(2)}$, Weights$^{\ln(2)/2^d}$ for $d = 2$, and Weights*– for 20 runs on t2.medium instances in Ohio and Frankfurt (100 ms delay, 100 Mbits/s bandwidth).

seconds, i.e., 0.78 weights per second, compared to 0.12 for [EKM+14].) We also evaluated our protocol on r4.2xlarge instances (8 vCPUs, 61 GB RAM), which we list in parenthesis in Table 6.2. In a LAN the running time compared to t2.medium instances is reduced by at least 30%, however, in a WAN setting the latency plays a more important role than powerful hardware and the running times are much closer. Thus, we only present running times for t2.medium instances in a WAN next.

WAN

We evaluated GM* and EM* with all weight computation subroutines in Figure 6.4 for $m \in \{3, 6, 10\}$ computation parties and $\lvert\mathfrak{D}\rvert \in \{10^5, 10^6, 10^7\}$. We split the m computation parties into two regions, Ohio (us-east-2) and Frankfurt (eu-central-1), and measured an inter-region round time

Protocol	$\vert\mathfrak{D}\vert$	*Communication*		
		$m = 3$	$m = 6$	$m = 10$
GM*	10^5	95 MB	253 MB	852 MB
	10^6	107 MB	273 MB	949 MB
	10^7	116 MB	286 MB	1.07 GB
EM* & Weights$^{\ln(2)}$	10^5	178 MB	402 MB	1.41 GB
	10^6	202 MB	448 MB	1.54 GB
	10^7	222 MB	497 MB	1.75 GB
EM* & Weights$^{\ln(2)/2^d}$, d=2	10^5	634 MB	1.38 GB	4.73 GB
	10^6	748 MB	1.63 GB	5.58 GB
	10^7	866 MB	1.88 GB	6.39 GB
EM* & Weights*	10^5	664 MB	1.56 GB	5.59 GB
	10^6	785 MB	1.83 GB	6.57 GB
	10^7	907 MB	2.11 GB	7.59 GB

Table 6.3: Communication cost: Data sent per party, average of 20 runs for $m \in \{3, 6, 10\}$ parties and $|\mathfrak{D}| \in \{10^5, 10^6, 10^7\}$.

trip (RTT) of approx. 100 ms with 100 Mbits/s bandwidth. The computation parties already received and combined secret-shared inputs from 10^6 users (Section 6.2.9) and we report the average running time of our protocol. Note that the results are very stable and the 95% confidence intervals deviate by less than 1% on average. Thus, we omit all confidence intervals in Figure 6.4 except the largest ones, i.e., for GM* with 3, 6 parties and $|\mathfrak{D}| = 10^7$ in Figure 6.4a.

Our protocol GM* (Figure 6.4a) requires less than 90 seconds for 3 parties and all domain sizes. GM* operates only on (scaled, truncated) integers and is always faster than EM*, which requires some floating-point operations for the weight computation and sampling. Weights$^{\ln(2)}$ (Figure 6.4b) is clearly the fastest weight computation, with running times around 3 minutes for 3 parties, whereas Weights$^{\ln(2)/2^d}$ (Figure 6.4c) and Weights* (Figure 6.4d) requires around 13 and 14 minutes respectively. However, we consider large domain sizes (billions of elements) in a real-world network with large latency and EM*, unlike GM*, requires some floating-point computations. The choice of weight computation enables a trade-off between faster running times, i.e., Weights$^{\ln(2)}$ with fixed ϵ, and smaller privacy loss ϵ, i.e, Weights*, with Weights$^{\ln(2)/2^d}$ positioned in the middle (faster running time than Weights* with smaller ϵ compared to Weights$^{\ln(2)}$). While GM* can be used to avoid such trade-offs, the number k of subranges allows similar adjustments, as discussed next.

6.3.2 Communication

The communication is detailed in Table 6.3. For 3 parties and 10^7 domain elements, the communication for GM* is 116 MB per party. For EM* with Weights$^{\ln(2)}$ each party sends 222 MB in the same setting, with Weights$^{\ln(2)/2^d}$ it is 866 MB, and with Weights* it is 907 MB. We stress that this communication is required for *malicious security* in each round as provided by the SCALE-MAMBA implementation. MP-SPDZ [Kel20], a fork of SCALE-MAMBA's predecessor SPDZ2, also provides semi-honest security. MP-SPDZ with semi-honest security requires much less communication, e.g., only around 25 MB for 3 parties, $|\mathfrak{D}| = 10^5$, and Weights*. However, the running

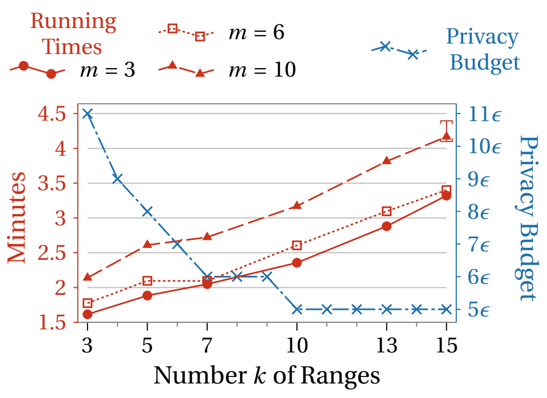


Figure 6.5: Privacy vs. running time trade-off: For increasing number k of subranges the running time (left axis) increases whereas the consumed privacy budget (right axis) decreases. (Illustrated for EM* with Weights$^{\ln(2)}$ and $|\mathfrak{D}| = 10^5$).

time in a WAN is some minutes slower compared to SCALE-MAMBA (presumably due to SCALE-MAMBA's batched communication rounds and integrated online and offline phases, where parallel threads create offline data "just-in-time" [AS19, AKR+20]). Thus, regarding our protocol, one can choose efficiency w.r.t. communication (MP-SPDZ) or running time (SCALE-MAMBA).

6.3.3 Malicious Security

To achieve malicious security, by consistency checks as detailed in Section 6.2.8, we require additional running time and communication. For the maximum number of evaluated steps and domain elements in a WAN (100 Mbits/s with 100 ms latency), GM* and EM* with Weights$^{\ln(2)}$ or Weights$^{\ln(2)/2^d}$ ($d = 2$) ensure consistency of ranks (integers), which additionally needs around 1.3/1.5/2 minutes and 115/260/825 MB for 3/6/10 parties. EM* with Weights* needs to check weights (floats) which additionally requires around 10/10/12 minutes and 0.65/1.4/5 GB for 3/6/10 parties.

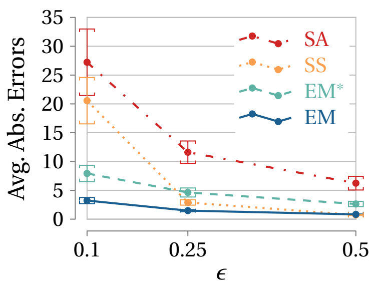
6.3.4 Privacy Budget vs. Running Time

The *privacy budget* is the sum of privacy parameters consumed per step, i.e., the overall privacy loss. Figure 6.5 shows how the privacy budget and the running time are affected by the number k of subranges. Larger k leads to larger running times, as the number of securely computed operations depends on the number of ranges times the number of selection steps (i.e., $k \cdot \lceil \log_k |\mathfrak{D}| \rceil$, see Section 6.2.7), which increases proportionally to k. However, smaller values for k require more selection steps ($\lceil \log_k |\mathfrak{D}| \rceil$), which lead to an increase in the privacy budget. Overall, as evident from Figure 6.5, for $k = 10$ subranges, as used in our evaluation, the consumed privacy budget is small with an acceptable running time.

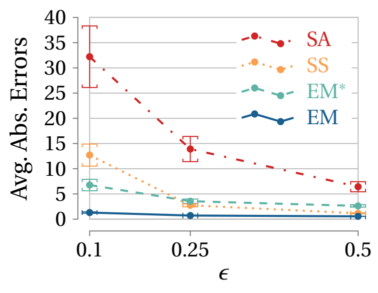
For our protocols supporting arbitrary ϵ per step, the trade-off becomes *accuracy vs. running time*: Larger k means the overall budget is spread among fewer steps, which improves accuracy, whereas smaller k corresponds to faster running time.

6.3.5 Accuracy Comparison to Related Work

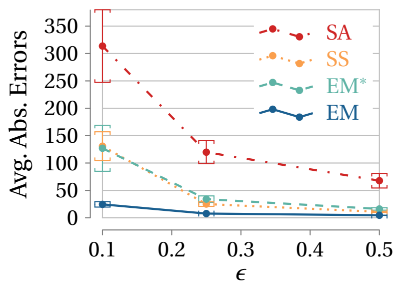
First, we detail how we choose the privacy parameter per selection step. Then, we compare our accuracy to related work.



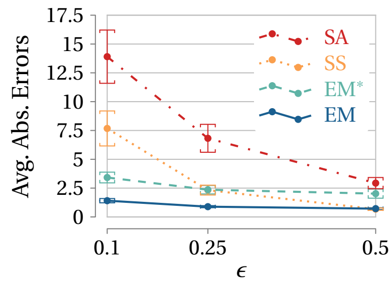
(a) Credit card data [ULB18], first 100 000 payment records in Cents.



(b) Walmart supply chain data [Kag18], ≈ 175 000 shipment weights as integers.



(c) California public salaries [Soo18], ≈ 71 000 records, state department's total wages.



(d) Normal distribution with $\sigma = 3$, 100 000 samples (as integers with scaling factor 1000).

Figure 6.6: Comparing exponential mechanism (EM) as baseline, this work (EM*), smooth sensitivity (SS) [NRS07], sample-and-aggregate (SA) [PL15] on different data, 100 averaged runs.

Composition of the Privacy Budget

Our protocols perform multiple selection steps s, each consume a portion ϵ_i of the overall privacy budget $\epsilon = \sum_{i=1}^{s} \epsilon_i$. How to optimally split ϵ (optimal composition) is #P-complete [MV16]. Thus, we use the following heuristic to divide ϵ among the selection steps: Initial steps cover exponentially larger subranges, and require exponentially less of the privacy budget. After a while an equal split is more advantageous, as the subranges become smaller and contain fewer elements. Altogether, we use $\epsilon_i = \epsilon/2^{s-i+1}$ if $i \leq \lfloor s/2 \rfloor$ and $\epsilon_i = \epsilon'/(s - \lfloor s/2 \rfloor)$ else, where ϵ' is the remaining privacy budget. We used $s = \lceil \log_k |\mathfrak{D}| \rceil - 1$ for our accuracy evaluation. We found in our experiments that performing one selection step less increases accuracy, as the privacy budget can be better divided among the other remaining steps and the last subrange is already small enough (at most k elements).

Accuracy Comparison

We list theoretical accuracy bounds for related work in Table 3.1 in Section 3.6. The theoretical accuracy bounds show that computing DP median exhibits a strong data dependence, which makes straightforward comparison difficult. Therefore, we empirically evaluated the different approaches closest to ours, i.e., supporting more than 2 parties, on real-world data sets [Kag18, ULB18, Soo18] as well as the normal distribution in Figure 6.6 for 100 averaged runs with 95%-confidence intervals. Recall that "small" data is the most challenging regime for DP [NRVW20, BEM+17], thus, we use small data sets to better illustrate the accuracy differences. As EM* and GM* have the same output distribution (Section 2.2.4) we only present the former in the figures.

The evaluation for smooth sensitivity [NRS07] and the exponential mechanism for elements assume a trusted party has access to the entire data set in clear. Note that only our approach and sample-and-aggregate as implemented by [PL15] use MPC instead of a trusted party. Nissim et al. [NRS07] (SS in Figure 6.6) compute instance-specific additive noise, requiring full data access, and achieve good accuracy, however, the exponential mechanism can provide better accuracy for low ϵ. Pettai and Laud [PL15] (SA in Figure 6.6) securely compute the noisy average of the 100 values closest to the median within a clipping range for their running time evaluation. There are three error sources: approximation error, clipping error, and and additive noise. In our accuracy evaluation (using 100 values as well), the approximation error, without any additive noise or clipping, was already larger than the error for the exponential mechanism for data set [ULB18]. Recall, the median is the 50th-percentile. To minimize the error from clipping range $[c_l, c_u]$, we choose c_l = 49th-percentile, c_u = 51th-percentile, i.e., we presume to already know a tight range for the actual median. Nonetheless, in our experiments the absolute error of SA is the largest.

Overall, no solution is optimal for all ϵ and data sets. However, the exponential mechanism EM, and our protocols EM* and GM* provide the best accuracy for low ϵ, i.e., high privacy, compared to approaches with additive noise [NRS07, PL15].

6.4 Summary

We presented a novel alternative for differentially private median computation with high accuracy (even for small number of users), without a trusted party, that is efficiently computable (practical running time) and scalable (sublinear in the size of the domain). Our semi-honest multi-party protocols implement the exponential mechanism (resp., Gumbel mechanism) for decomposable aggregate functions (e.g., rank-based statistics, convex loss functions) as used in MapReduce-style algorithms, and can be extended to malicious parties. For the median, the exponential mechanism provides the best utility vs. privacy trade-off for low ϵ in our evaluations of related work in the central model. We optimize our protocols for decomposable functions (allowing efficient MPC over distributed data), and use efficient alternatives to exponentiations for floating-point numbers. We implemented our protocols in the SCALE-MAMBA framework [AKR+20], and evaluated it for 1 million users using 3 semi-honest computation parties with a running time of seconds in a LAN, and 3 minutes in a WAN with 100 ms network delay, 100 Mbits/s bandwidth.

7 HH & PEM: DP Heavy Hitters

The goal of federated heavy hitters discovery, also called top-k, is to learn the k most frequent values in a distributed data set. We present efficient MPC protocols PEM and HH to discover DP heavy hitters achieving central-model accuracy without a trusted party. This chapter is based on the following publication:

> Jonas Böhler, Florian Kerschbaum. Secure Multi-party computation of Differentially Private Heavy Hitters. In *Computer and Communications Security*, CCS, 2021 [BK21].

DP is widely deployed in the industry for private heavy hitter discovery over user-generated data. As detailed in Section 1, heavy hitters are typically collected to detect trends and patterns, e.g., frequently typed new words [App16, App17], common user settings [EPK14, FPE16a], and often shared articles [RSP+20]. Existing solutions to find DP heavy hitters either require a large number of parties to achieve meaningful accuracy (local model [App16, DKY17, EPK14, FPE16a]), require a trusted third party (central model [RSP+20, Rog20]), or use cryptography but do not achieve high accuracy with efficient protocols (MPC [BBC+21, MDDC16, NPR19]).

The remainder of this chapter is organized as follows. In Section 7.1, we first describe an ideal version of our protocol with a trusted third party, which we later replace with MPC. The straightforward algorithms to accurately detect heavy hitters are inefficient in MPC, and hence we need to employ clever approximate algorithms called *sketches* over streams [CH10] or large domains [WLJ19] to make the secure multi-party computation efficient. Sketches are succinct data structures which typically store counters indexed by multiple hash functions, e.g., count-min sketch [CH10, MDDC16] or Bloom filters [EPK14, FPE16a]. Usually, local-model users apply domain reduction (e.g., hashing) [BS15, BNST17, WLJ19] before randomization. However, hash-based techniques require costly search efforts, e.g., hashing the entire domain to find matching heavy hitters. Searching can consume significant computational resources to, e.g., compute and match billions of hashes [WLJ19] or estimate joint probabilities of perturbed n-grams [FPE16b] (see Section 3.7). Our key insight is that adapting suitable sketches that do not require search allows efficient MPC of DP heavy hitters with high accuracy.

In Section 7.2, we present two MPC protocols to discover the DP heavy hitters on distributed user data: HH and PEM. Our protocols are based on state-of-the-art solutions for heavy hitter detection – HH is build upon *non-private* detection in data streams [CH10], and PEM adapts the *local DP* method from Wang et al. [WLJ19] – realized as efficient MPC implementations of *central DP* randomizations [WZL+20, DR19a, DR19b].

In Section 7.3, we provide a detailed performance evaluation. HH has running time linear in the size of the data and is applicable for very small data sets (hundreds of values). PEM is sublinear in the data domain (i.e., linear in the bit-length of domain size) and provides better accuracy than HH for a large number of users (thousands to millions). In our experiments, we achieved running times of less than 11 minutes using 3 semi-honest computation parties in a WAN with 100 ms network delay, 100 Mbits/s bandwidth. We conclude this chapter in Section 7.4.

```
$\mathcal{F}_{\mathsf{HH}}$

1. Define map $T$ with maximum size $t$ to associate a datum with a count.
2. For each user reported datum $d \in D$:
   a) If $d \in T$, then increment counter $T[d]$.
   b) Else if $|T| < t$, then add $d$ to $T$, and set $T[d]$ to 1.
   c) Else, decrement all counters $T[i]$, and remove $i$ from $T$ if $T[i] = 0$.
3. For each item $i \in T$:
   a) Add noise Laplace$(\Delta/\epsilon)$ to count $T[i]$.
   b) Remove $i$ from $T$ unless
          $T[i] \geq \tau_{\mathsf{HH}}$,
      where $\tau_{\mathsf{HH}} = 1 - \Delta \log\left(2 - 2(1-\delta)^{1/\Delta}\right)/\epsilon$.
4. Output items in $T$ sorted by their noisy count.
```

Figure 7.1: Ideal functionality $\mathcal{F}_{\mathsf{HH}}$ combining heavy hitter detection in streams [MG82],[CH10, Alg. 1] with DP bounded count release [WZL+20, Th. 2].

7.1 Federated Heavy Hitters

The following ideal functionalities $\mathcal{F}_{\mathsf{HH}}$ and $\mathcal{F}_{\mathsf{PEM}}$ describe our protocols as executed by a trusted third party, which we later replace by implementing them with optimized MPC protocols as HH and PEM, respectively. The straight-forward algorithms to accurately detect heavy hitters are inefficient in MPC. Therefore, we employ clever approximate algorithms that work over *streams with unknown domains* [CH10] (non-private) or *support large domains* [WLJ19] (local model) to make the secure multi-party computation efficient. The employed sketches do not require hashing or domain reduction (e.g., Bloom filters [EPK14], or matrix projection [BS15]) and avoid the additional search efforts associated with these techniques. Clients only send a single message – either their value ($\mathcal{F}_{\mathsf{HH}}$) or a bit vector indicating the bit-prefix of their value ($\mathcal{F}_{\mathsf{PEM}}$) – and the server updates a map that associates client messages with a count. We utilize central model thresholds [WZL+20, DR19a, DR19b] and show that $\mathcal{F}_{\mathsf{HH}}, \mathcal{F}_{\mathsf{PEM}}$ are differentially private. In the following, we let Δ denote the maximum number of counts an individual can influence, e.g., $\Delta = 1$ if we query countries of origin, or $\Delta \geq 1$ for current and former employers. Next, we formalize the top-k problem:

Definition 27 (Top-k or Heavy Hitter)**.** *Datum $d \in D$ is a* top-k element *if its frequency f_d in D is among the k most frequent elements, where $f_d = |\{x \mid x \in D$ and $x = d\}|/|D|$.*

7.1.1 Ideal Functionality $\mathcal{F}_{\mathsf{HH}}$

Cormode and Hadjieleftheriou [CH10] surveyed algorithms for (non-private) heavy hitter detection in data streams and found counter-based approaches, to be the best w.r.t. accuracy, speed and space, which was re-confirmed by more recent work [ABL+17]. Next we describe a non-private counter-based approach, which we augment to be privacy-preserving.

Non-private Misra-Gries

The counter-based approach *Misra-Gries* [MG82],[CH10, Alg. 1], is the main part of our ideal functionality $\mathcal{F}_{\mathsf{HH}}$ in Figure 7.1: making up all steps, excluding the DP thresholding in step 3.

Misra-Gries uses counters to track the frequency of already seen elements in a data stream and provides the following guarantee [ABL+17, Lemma 1]:

Lemma 5. *Misra-Gries run on D of size n with t counters provides a frequency estimate $\widehat{f_d}$ for all $d \in D$ satisfying $0 \leq f_d - \widehat{f_d} \leq n/(t+1)$.*

For skewed frequencies, i.e., data sets consisting of mainly a few very frequent items, Berinde et al. [BICS10] show the following *tail guarantee*:

Lemma 6. *Misra-Gries run on D of size n with t counters provides a frequency estimate $\widehat{f_d}$ for all $d \in D$ satisfying $0 \leq f_d - \widehat{f_d} \leq n_{-j}/(t+1-j)$, where $k < j$ and n_{-j} is the data size without the top j most frequent elements.*

Recent improvements [ABL+17] reduce the expected number of times the expensive decrement branch is executed (2c in Figure 7.1), as it requires updating the entire map T. However, as we later implement $\mathcal{F}_{\mathsf{HH}}$ with MPC, which must hide the control flow to prevent leakage, we cannot apply them and focus on the original version. Note that $\mathcal{F}_{\mathsf{HH}}$, due to its use of Misra-Gries, does not require any domain knowledge or distribution assumptions. Also, if the map size is equal to the size of the (small) data set, $\mathcal{F}_{\mathsf{HH}}$ computes an *exact* histogram over an unknown data domain.

Differentially private $\mathcal{F}_{\mathsf{HH}}$

The ideal functionality $\mathcal{F}_{\mathsf{HH}}$ in Figure 7.1 approximates counts for frequent values seen so far via Misra-Gries [MG82] but only releases noisy counts that exceed the δ-based threshold τ_{HH} defined by Wilson et al. [WZL+20, Th. 2].

Theorem 17. *The ideal functionality $\mathcal{F}_{\mathsf{HH}}$ provides $(\Delta\epsilon, \delta)$-differential privacy.*

Proof. Wilson et al. proof in [WZL+20, Th. 2] that the threshold τ_{HH} satisfies $(\Delta\epsilon, \delta)$-DP for counts of unique user contributions in SQL. (I.e., non-empty groups with noisy counts of say column 1 grouped by column 2 are released if they exceed the threshold, and the threshold bounds the probability for releasing differing results between neighbors.) We briefly sketch their proof: A noisy count will be at least τ with probability $p = \frac{1}{2}e^{-\frac{\epsilon(\tau-1)}{\Delta}}$ (property of Laplace distribution). The probability for bad events (e.g., releasing a count for a data set but not its neighbor) is bounded as $p^{\Delta} \leq \delta$ and solving for τ provides τ_{HH}. As we assume a single value per user, each count qualifies as a unique contribution per user, allowing us to use the same threshold τ_{HH}. □

7.1.2 Ideal Functionality $\mathcal{F}_{\mathsf{PEM}}$

Wang et al. [WLJ19] present a "prefix extension method" (PEM) for LDP heavy hitter detection and show that it provides higher accuracy than other LDP approaches [FPE16a, BS15, BNST17]. We adapt their local model protocol, which we denote PEMorig, for our central model protocol $\mathcal{F}_{\mathsf{PEM}}$, and describe them next.

Local model PEMorig

PEMorig leverages overlapping segments by iteratively finding frequent prefixes of increasing lengths. Informally, users are split evenly in disjoint groups. The first group reports perturbed $(\gamma+\eta)$-bit prefixes of their datum to a server, and the server estimates the frequencies of all prefix

```
$\mathcal{F}_{\mathsf{PEM}}$

1. Initialize prefix set $S = \{0,1\}^{\lceil \log k \rceil}$, and split user data $D$
   in $g = \left\lceil \frac{b - \lceil \log k \rceil}{\eta} \right\rceil$ disjoint groups $D_1, \ldots, D_g$.
2. For each group $j \in \{1, \ldots, g\}$:
   a) Initialize empty map $T$ to associate a prefix with a count, and candidate
      prefix set $C = S \times \{0,1\}^{\eta}$.
   b) For each prefix $c \in C$:
         i. Set $T[c] = \sum_{d \in D_j} \zeta_d^c$, where $\zeta_d^c \in \{0,1\}$ is a user report indicating if her
            value $d$ matches prefix candidate $c$.
   c) Set $S = \{\}$ and $z = \min_{c \in C} T[c] + \text{Laplace}(1/\epsilon)$.
   d) For the top-$k$ prefixes $c_k \in C$:
      i. Add noise Laplace$(1/\epsilon)$ to count $T[c_k]$.
      ii. Add $c_k$ to $S$ if
                 $T[c_k] \geq \tau_{\mathsf{PEM}} + z,$
          where $\tau_{\mathsf{PEM}} = 1 + \log(\Delta/\delta)/\epsilon$.
   e) Output items in $S$ sorted by their noisy count.
```

Figure 7.2: Ideal functionality $\mathcal{F}_{\mathsf{PEM}}$ combining distributed heavy hitter detection [WLJ19] with central-DP thresholding [DR19a, DR19b].

candidates (i.e., all binary strings with the same length as the bit prefix). Then, the prefix length is extended by η, the second group reports their perturbed prefixes of length $\gamma + 2\eta$, and the server estimates frequencies of prefixes that extend the top-2^{γ} prefixes of the previous group. This is repeated until the prefix length reaches the domain bit-length b.

To create the reports, a user first reduces the domain size via *optimal local hashing* [WBLJ17], then applies *generalized randomized response* (GRR) on the reduced domain. In more detail, a user in group i selects a hash function $H : \mathfrak{D} \rightarrow \{1, \ldots, u\}$ from a family of hash functions $\mathcal{H}$, where $u = \lceil \exp(\epsilon) + 1 \rceil$. Then, she computes $h = \mathsf{GRR}(H(d'))$ of the $(\gamma + i\eta)$-bit prefix d' of her datum d. Recall Definition 9 from Section 2.2.4, $\mathsf{GRR}(x)$ outputs x with probability $p = \frac{\exp(\epsilon)}{\exp(\epsilon)+u-1}$ and $y \neq x$ with probability $\frac{1}{\exp(\epsilon)+u-1}$ over domain $\{1, \ldots, u\}$ of size u. Finally, she reports (H, h) to the server. Given the reports, the server creates a candidate set C by extending the previous top-2^{γ} prefixes with all possible binary strings of length η. Then, the server estimates the count of each prefix candidate $c \in C$ as $\frac{s_c - n/u}{p - 1/u}$ [WLJ19, Eq. (2)], where s_c is the number of reports with matching hashes, i.e., $s_c = |\{c \mid c \in C \text{ and } H(c) = h\}|$.

The parameter η provides the following trade-off: Smaller values lead to more groups but less (hash) computations, whereas larger values produces fewer groups but requires more computational resources. Note that more groups means fewer counts per prefix candidate which can lead to reduced accuracy. Wang et al. [WLJ19] set $\gamma = \lceil \log_2 k \rceil$ and limit the number of hash computations per report to 2^{20} (i.e., set η to the largest integer satisfying $g2^{\gamma+\eta} < 2^{20}$ for $g = \lceil (b - \gamma)/\eta \rceil$ groups). Overall, PEMorig, with these parameters, requires the server to compute $n2^{20}$ hashes.

Central model $\mathcal{F}_{\mathsf{PEM}}$

Our protocol $\mathcal{F}_{\mathsf{PEM}}$, shown in Figure 7.2, also leverages extending prefixes to find heavy hitters over distributed data. Unlike PEMorig, $\mathcal{F}_{\mathsf{PEM}}$ operates on actual counts instead of estimates from perturbed reports to increase the accuracy. We later implement $\mathcal{F}_{\mathsf{PEM}}$ with MPC to protect the counts and use $\eta \in \{4,5\}$ in our evaluation as this provides a practical trade-off between computational efficiency and accuracy for a small number of users (Section 7.3).

$\mathcal{F}_{\mathsf{PEM}}$ releases *intermediate* results (set of frequent prefix candidates) to improve the frequency estimation in multiple rounds, unlike minimal functionality HH, which only releases the *final* result. However, this does not violate privacy: Differential privacy is enforced in line 2(d)ii of Figure 7.2 by only releasing values whose associated (noisy) frequencies exceed a threshold. The privacy budget of $\mathcal{F}_{\mathsf{PEM}}$ is given next.

Theorem 18. *The ideal functionality $\mathcal{F}_{\mathsf{PEM}}$ provides $\left(\Delta\epsilon, \frac{\delta}{4}(\exp(\Delta\epsilon)+1)(3+\log(\Delta/\delta))\right)$-differential privacy.*

Proof. First, note that the claim holds for a single group, i.e., step 2d, as the thresholding satisfies $\left(\Delta\epsilon, \frac{\delta}{4}(\exp(\Delta\epsilon)+1)(3+\log(\Delta/\delta))\right)$-DP [DR19a, Lemma 6.1]. Now we expand this, without additional privacy cost, to all groups (step 2): Recall, we compute counts on *disjoint* subsets (i.e., D_g of D for group g). Thus, we never count a user contribution more than once. Applying parallel composition [LLSY16, Section 2.2.2] allows us to use the *maximum* (instead of the sum) over the privacy budget for all steps as total budget. As we use the *same* budget per step the maximum is equal to that of a single step, which concludes the proof. □

Unrestricted Sensitivity

In the case of unrestricted sensitivity, i.e., Δ much larger than $|C|$, Durfee and Rogers [DR19a] use Gumbel noise instead of Laplace noise. With Gumbel noise $\mathcal{F}_{\mathsf{PEM}}$ is $(\approx \sqrt{k}\epsilon, \delta)$-DP [DR19a, Th. 1] (i.e., with a dependence on k instead of Δ). To support unrestricted sensitivity in $\mathcal{F}_{\mathsf{PEM}}$ the Laplace noise in lines 2c, 2(d)i of Figure 7.2 changes to Gumbel noise with the same scale with new threshold $\tau_{\mathsf{PEM}} = 1 + \log(|C|/\delta)/\epsilon$ (i.e., Δ replaced by $|C|$). In the following, we focus on the setting with restricted sensitivity, i.e., Laplace noise, but our protocol can be extended to the unrestricted case by using distributed Gumbel noise (Section 7.1.4).

7.1.3 When to use $\mathcal{F}_{\mathsf{HH}}$ or $\mathcal{F}_{\mathsf{PEM}}$

In the following, we discuss the accuracy of $\mathcal{F}_{\mathsf{HH}}$ and $\mathcal{F}_{\mathsf{PEM}}$ in relation to each other.

Theorem 19. *For data set D of size n, $\mathcal{F}_{\mathsf{HH}}$ with fixed map size t provides better accuracy in expectation than $\mathcal{F}_{\mathsf{PEM}}$ if*

$$\tau_{\mathsf{HH}} + \frac{n}{t+1} < f_{k\text{-}th} \leq g \cdot \left(\tau_{\mathsf{PEM}} + f_{|C|\text{-}th}\right)$$

where $f_{k\text{-}th}$ is the frequency of the k-th most frequent element in D, g is the number of groups in $\mathcal{F}_{\mathsf{PEM}}$, $|C|$ is the size of the candidate set in $\mathcal{F}_{\mathsf{PEM}}$, and $\tau_{\mathsf{HH}}, \tau_{\mathsf{PEM}}$ as in Figures 7.1, 7.2 respectively.

Proof. We consider the cases where $\mathcal{F}_{\mathsf{HH}}$ releases a candidate and $\mathcal{F}_{\mathsf{PEM}}$ does not. $\mathcal{F}_{\mathsf{HH}}$ releases candidate c if $T[c] + \text{Laplace}(1/\epsilon) > \tau_{\mathsf{HH}}$. $\mathcal{F}_{\mathsf{HH}}$ uses estimated frequency $T[c] = \widehat{f_c}$, which is at most $n/(t+1)$ below actual frequency f_c (Lemma 5). Thus, $T[c] > f_c - n/(t+1)$ using the fact that Laplace noise is 0 in expectation and replacing f_c with $f_{k\text{-th}}$, we have $\tau_{\mathsf{HH}} + n/(t+1) < f_{k\text{-th}}$.

Analogously, PEM doesn't release candidate c if $T[c] + \text{Laplace}(1/\epsilon) \leq \tau_{\mathsf{PEM}} + \text{Laplace}(1/\epsilon)$. Assuming data is distributed uniformly between groups, we have $T[c] = f_c/g$. Assuming expected noise and $z = f_{|C|\text{-th}}$, replacing f_c by $f_{k\text{-th}}$ as before, we arrive at $f_{k\text{-th}}/g \leq f_{|C|\text{-th}} + \tau_{\mathsf{PEM}}$, which is the right side of the inequality when multiplied with g. □

For fixed η, larger domain bit-length leads to larger group size g in $\mathcal{F}_{\text{PEM}}$. Since $\mathcal{F}_{\text{HH}}$ is independent of the domain size, it provides better accuracy in such cases, as the counts per value are not split among multiple groups. However, we want to keep t small and fixed for our MPC protocol, as $\mathcal{F}_{\text{HH}}$ requires t operations per datum in the worst case (decrement step). Fixed t reduces accuracy for increasing data sizes (Lemma 5); therefore, $\mathcal{F}_{\text{HH}}$ is better suited for small data sets (small n). Also, the candidate set in $\mathcal{F}_{\text{PEM}}$ can be empty if the counts are lower than the threshold, i.e., for very small data sets (a few dozen or hundred values), which provides another argument in favor of $\mathcal{F}_{\text{HH}}$ for small data sets. Our empirical analysis in Section 7.3 confirms these observations.

7.1.4 Distributed Noise Generation

In our ideal functionalities, the noise comes from the trusted parties. In our MPC protocols, the noise is provided by the input parties (resp., computation parties). Distributed noise generation is more efficient than securely sampling from a noise distribution as discussed in Section 2.2.5. In our MPC protocols, we combine partial noise values ρ_p from each party $p \in \mathcal{P}$, to sample from the Laplace and Gumbel distribution as detailed in Section 2.2.5. Recent works consider alternative Laplace noise representations on finite machines, e.g., [BV19, GKMP20, BBGN20], which we can leverage as well. The distributed noise representation does not affect our MPC efficiency as they are based on (integer) addition. Recall, the Gumbel distribution can be expressed as an infinite sum of random variables from the exponential distribution. Note that the input parties can precompute an arbitrary number of such sum terms, and add them to their prefix counts, thus, they only need to provide one input per count for our MPC of $\mathcal{F}_{\text{PEM}}$: a (partially) noisy count $\widehat{\zeta_d^c}$. Alternatively, the computation parties can provide the noise.

7.2 MPC for DP Heavy Hitters

We describe details of our MPC protocols HH, PEM which realize the ideal functionalities $\mathcal{F}_{\text{HH}}$, $\mathcal{F}_{\text{PEM}}$ without a trusted party, and analyse their running time and security.

We use upper case letters to denote arrays in our protocol, and $A[j]$ denotes the j-th element in array A. We indicate Boolean values (in the form of a bit) with b_{state} (e.g., $b_{\text{match}} = 1$ indicates a match). The MPC sub-protocols used in our protocol are listed in Table 2.3. While most of our computation can be represented with integers, our protocol uses fixed-point numbers (scaled, truncated floats) to handle DP noise. Limited machine precision of floating-point numbers can lead to privacy violations in the implementation of the Laplace mechanism [Mir12]. These violations can be mitigated by careful truncation and rounding of floating-point numbers. We do not release noisy counts and do not use floating-point numbers, nonetheless, similar attacks might exist without careful selection of fixed-point numbers.

Secure Sort

We want to release the ordered top-k, i.e., the most frequent values sorted by their counts. Note that we cannot release the noisy counts with their corresponding values to let the parties sort the values locally. While Laplace noise is differentially private, allowing the release of noisy counts, the same is not true for Gumbel noise [DR19a] as used by PEM with unrestricted sensitivity. Furthermore, we consider distributed noise generation, where each party provides partial noise values. Here, each party can remove its partial noise from the noisy count, requiring additional noise or secure noise sampling to prevent a privacy violation or degradation (briefly discussed in Section 7.2.4). Thus, we securely sort the values by their corresponding counts and only release the values like the ideal functionalities $\mathcal{F}_{\mathsf{HH}}$ and $\mathcal{F}_{\mathsf{PEM}}$.

We use the existing secure sorting implemented in SCALE-MAMBA and MP-SPDZ in line 11 of Algorithm 15. The implementations[1] are based on merge sort and conditional swaps. Whenever an array value $A[i]$ is smaller than $A[i+1]$, i.e., $c = \mathsf{LE}(A[i+1], A[i])$ is 1, they are swapped. However, we slightly adapt it, and re-use the comparison result c to sort a second array B in the same way, i.e., for each swap with A we simply perform the same swap with B. A conditional swap of two inputs a, b with a selection bit c can be efficiently realized as $(a+d, b-d)$ with a single multiplication $d = c \cdot (b-a)$. Then, for $c = 1$ the elements are swapped as $(a+(b-a), b-(b-a)) = (b, a)$, and for $c = 0$ they remain as $(a+0, b-0) = (a, b)$. We suggested this approach with a single multiplication (as used by, e.g., Pettai and Laud [PL15]) to the SCALE-MAMBA team[2], and it replaced their previous approach with two multiplications in version 1.9.

7.2.1 HH: MPC of $\mathcal{F}_{\mathsf{HH}}$

Instead of a map T, as in $\mathcal{F}_{\mathsf{HH}}$, we use two arrays V and C which store a value and its corresponding count at the same index. Note that AND and NOT in lines 12, 14 of Algorithm 14 are just aliases for Mul and $\mathsf{Sub}(1, \cdot)$, respectively. We use aliases to improve the readability of our protocol and to highlight that their inputs and outputs are "bits", which we represent as integers 0 and 1 in the following. We ensure that the inputs to AND, NOT are in $\{0, 1\}$, thus, their outputs can only be in $\{0, 1\}$ as well. HH implements the different if-else branches of $\mathcal{F}_{\mathsf{HH}}$ via bits, i.e., b_{found} is set if a value is already in V; $b_{\text{empty},j}$ is set if we had no match ($\mathsf{NOT}(b_{\text{found}})$) but index j is empty; and $b_{\text{decrement}}$ is set if we did not find a match and have no empty spots left. We employ the following optimizations to reduce the number of MPC protocols: Instead of using arithmetic $\mathsf{OR}(a, b) = a + b - a \cdot b$, to combine bits b_{match} into bit b_{found} we add each bit b_{match} (which can be set, i.e., 1, at most once) to form b_{found} (which is 1 only if any match occurred) in line 7. Note that only unique values are in array V. Thus, b_{match} is 1 at most once in the loop and b_{found} is either 0 or 1 and can be input to NOT and AND. Replacing OR by Add is beneficial, since Add can be evaluated locally in secret sharing, i.e., without interaction, whereas arithmetic expression of OR requires multiplication and thus interaction. Similarly, we reduce the number of Mux operations by directly using $b_{\text{decrement}}$ as a decrement value. Furthermore, we do not need to remove values associated with empty counts, saving additional Mux operations. We only use counts to check if a value is empty and if the value is matched (even with empty count), we set the new count to 1 (line 16), i.e., same as if we had not matched and found an empty spot.

We also implement a version more suited for parallelization, denoted as $\mathsf{HH}_{\mathsf{threads}}$ in our evaluation (Section 7.3). The loop bodies in HH can be run in parallel, if we do not set i_{empty} in the first loop (as this requires locking). Thus, the main difference between HH and $\mathsf{HH}_{\mathsf{threads}}$ is that we use an additional (non-parallelized) loop to set i_{empty}.

7.2.2 PEM: MPC of $\mathcal{F}_{\mathsf{PEM}}$

PEM implements $\mathcal{F}_{\mathsf{PEM}}$ by using array C to count candidate prefixes. The users themselves can track which indices correspond to candidate prefixes, simplifying the secure computation com-

[1] `https://github.com/KULeuven-COSIC/SCALE-MAMBA/blob/862ecf547a01883cfbaf81a07c444c0c7cb53010/Compiler/library.py#L424` and `https://github.com/data61/MP-SPDZ/blob/v0.1.8/Compiler/library.py#L464`
[2] `https://groups.google.com/g/spdz/c/urM4Xy46H6I/m/CWJLOjqtAAAJ`

Algorithm 14 HH: MPC of $\mathcal{F}_{\mathsf{HH}}$

Input: User data $\langle D\rangle$, distributed noises $\langle\rho_p\rangle$ per party $p\in\mathcal{P}$, output size k, map size t, and DP threshold τ_{HH}.

Output: DP top-k.

```
1:  Initialize arrays $\langle V\rangle, \langle C\rangle$ of size $t$ with $\langle\perp\rangle, \langle 0\rangle$, resp.
2:  for user datum $\langle d\rangle \in \langle D\rangle$ do  // Update counts $C$ for values $V$
3:    Initialize $\langle b_{\text{found}}\rangle \leftarrow \langle 0\rangle$ and $\langle i_{\text{empty}}\rangle \leftarrow \langle -1\rangle$
4:    for index $j \leftarrow 1$ to $t$ do
5:      $\langle b_{\text{match}}\rangle \leftarrow \mathsf{EQ}(\langle d\rangle, \langle V[j]\rangle)$
6:      $\langle b_{\text{empty}}\rangle \leftarrow \mathsf{LE}(\langle C[j]\rangle, \langle 0\rangle)$
7:      $\langle b_{\text{found}}\rangle \leftarrow \mathsf{Add}(\langle b_{\text{found}}\rangle, \langle b_{\text{match}}\rangle)$
8:      $\langle i_{\text{empty}}\rangle \leftarrow \mathsf{Mux}(\langle j\rangle, \langle i_{\text{empty}}\rangle, \langle b_{\text{empty}}\rangle)$
9:      $\langle C[j]\rangle \leftarrow \mathsf{Add}(\langle C[j]\rangle, \langle b_{\text{match}}\rangle)$
10:   end for
11:   $\langle b_{\neg\text{empty}}\rangle \leftarrow \mathsf{EQ}(\langle i_{\text{empty}}\rangle, \langle -1\rangle)$
12:   $\langle b_{\text{decrement}}\rangle \leftarrow \mathsf{AND}(\langle b_{\neg\text{empty}}\rangle, \langle \mathsf{NOT}(\langle b_{\text{found}}\rangle)\rangle)$  // $\mathsf{AND}, \mathsf{NOT}$ are $\mathsf{Mul}, \mathsf{Sub}(1,\cdot)$, resp.
13:   for index $j \leftarrow 1$ to $t$ do  // Conditional decrement
14:     $\langle b_{\text{empty},j}\rangle \leftarrow \mathsf{AND}(\langle \mathsf{NOT}(\langle b_{\text{match}}\rangle)\rangle, \langle \mathsf{EQ}(\langle i_{\text{empty}}\rangle, \langle j\rangle)\rangle)$
15:     $\langle c\rangle \leftarrow \mathsf{Sub}(\langle C[j]\rangle, \langle b_{\text{decrement}}\rangle)$
16:     $\langle C[j]\rangle \leftarrow \mathsf{Mux}(\langle 1\rangle, \langle c\rangle, \langle b_{\text{empty},j}\rangle)$
17:     $\langle V[j]\rangle \leftarrow \mathsf{Mux}(\langle d\rangle, \langle V[j]\rangle, \langle b_{\text{empty},j}\rangle)$
18:   end for
19: end for
20: for index $j \leftarrow 1$ to $t$ do  // DP thresholding on noisy $C$
21:   for party $p \in \mathcal{P}$ do
22:     $\langle C[j]\rangle \leftarrow \mathsf{Add}(\langle C[j]\rangle, \langle \rho_p^j\rangle)$
23:   end for
24:   $\langle b_{\text{discard}}\rangle \leftarrow \mathsf{LE}(\langle C[j]\rangle, \langle \tau_{\mathsf{HH}}\rangle)$
25:   $\langle V[j]\rangle \leftarrow \mathsf{Mux}(\langle\perp\rangle, \langle V[j]\rangle, \langle b_{\text{discard}}\rangle)$
26: end for
27: Sort values $\langle V\rangle$ by corresponding counts $\langle C\rangle$ descendingly  // Section 7.2
28: return $\mathsf{Rec}(\langle V\rangle)$
```

plexity. For group $i\in\{1,\ldots,g\}$, the prefix bit-length $i\eta+\gamma$ *varies*, however, each group reports the same *fixed* number of counts, i.e., $2^{\lceil\log_2 k\rceil+\eta}$. Hence, the size of array C is fixed. As a toy example, consider $k=2, \eta=1, \gamma=1$. First, each user j from group 1 reports 4 (noisy) counts $\{\widehat{\zeta^c_{d_j}}\}_{c\in X}$ for prefix candidates $X=\{00,01,10,11\}$. The counts are aggregated in array C, where $C[1]$ maps to the first prefix 00 in X, $C[2]$ to the second prefix 01, etc. Let the top-2 prefixes be $Z=\{00,01\}$, i.e., Algorithm 15 outputs $\{1,2\}$ for $i=1$, which corresponds to the first two prefixes in X. Then, group 2 also reports 4 counts but for $X=Z\times\{0,1\}^\eta=\{000,001,010,011\}$. Note that the number of prefixes is the same but their bit-length is extended by $\eta=1$. Let the set of top-2 prefixes be $Z=\{001,011\}$, i.e., Algorithm 15 outputs $\{2,4\}$ for $i=2$. And so on. Note that outputting indices of an *ordered* list of prefix candidates suffices to reconstruct prefixes as above.

In the last round of PEM, less than $2^{\lceil\log k\rceil+\eta}$ iterations are required if $(b-\lceil\log k\rceil)/\eta$ is not an integer. We use this optimization in our implementation but omit it here for readability. If we are not interested in the order, i.e., which value is the i-th most frequent, the sorting step can be replaced by linear scan (to find the minimum count for the threshold), improving the complexity of this step from $O(c\log c)$ to c for $c=2^{\lceil\log k\rceil+\eta}$ (leading to c instead of k iterations for thresholding in line 16 of Algorithm 15).

Algorithm 15 PEM: MPC of $\mathcal{F}_{\mathsf{PEM}}$

```
Input: Noisy user reports $\langle\widehat{\zeta_d^c}\rangle$ indicating if their $d \in D$ has prefix $c$ (with distributed noise, Section 7.1.4), distributed noises $\langle\rho_p\rangle$ per party $p \in \mathcal{P}$, output size $k$, domain bit-length $b$, prefix extension bit-length $\eta$, and DP threshold $\tau_{\mathsf{PEM}}$.
Output: DP top-$k$.
 1: Split users in $g = \left\lceil \frac{b-\lceil\log k\rceil}{\eta} \right\rceil$ disjoint groups where $D = \bigcup_{i=1}^{g} D_i$
 2: for group $i \leftarrow 1$ to $g$ do
 3:   Initialize arrays $\langle S\rangle, \langle C\rangle$ of sizes $k, 2^{\lceil\log k\rceil+\eta}$ with zeros
 4:   Initialize array $\langle I\rangle \leftarrow \{\langle 1\rangle, \dots, \langle 2^{\lceil\log k\rceil+\eta}\rangle\}$
 5:   Initialize $\langle\rho_\tau\rangle \leftarrow \langle 0\rangle$ and $\langle\tau\rangle \leftarrow \langle 0\rangle$
 6:   for candidate $c \leftarrow 1$ to $2^{\lceil\log k\rceil+\eta}$ do
 7:     for user datum $d \in D_i$ do  //Gather candidate counts
 8:       $\langle C[c]\rangle \leftarrow \mathsf{Add}(\langle C[c]\rangle, \langle\widehat{\zeta_d^c}\rangle)$
 9:     end for
10:   end for
11:   Sort candidate indices $\langle I\rangle$ by corresponding counts $\langle C\rangle$ descendingly  //Section 7.2
12:   for party $p \in \mathcal{P}$ do
13:     $\langle\rho_\tau\rangle \leftarrow \mathsf{Add}(\langle\rho_\tau\rangle, \langle\rho_p\rangle)$
14:   end for
15:   $\langle\tau\rangle \leftarrow \mathsf{Add}(\langle\mathsf{Add}(\langle\tau_{\mathsf{PEM}}\rangle, \langle\rho_\tau\rangle)\rangle, \langle C[2^{\lceil\log k\rceil+\eta}]\rangle)$
16:   for candidate $c \leftarrow 1$ to $k$ do  //DP thresholding on noisy $C$
17:     $\langle b_{\text{discard}}\rangle \leftarrow \mathsf{LE}(\langle C[c]\rangle, \langle\tau\rangle)$
18:     $\langle S[c]\rangle \leftarrow \mathsf{Mux}(\langle\bot\rangle, \langle I[c]\rangle, \langle b_{\text{discard}}\rangle)$
19:   end for
20:   return $\mathsf{Rec}(\langle S\rangle)$
21: end for
```

7.2.3 Running Time Complexity

We analyse the running time of our protocols HH, PEM w.r.t. the number of basic MPC protocols – namely, EQ, LE, Mul, Mux, Rec – as detailed in Table 2.3 in Section 2.1.6. Interaction-free protocols, e.g., addition, are omitted, as the parties can compute them locally on secret shares. The complexity for the required basic protocols is at most $O(l)$ for l-bit integers.

Theorem 20. HH *has complexity* $O(nt)$.

Proof. For each of the n values in D protocol HH requires: First, t equality checks (EQ), comparisons (LE), and selections (Mux), to find matching values and look for an empty index. Then, one EQ, AND, and NOT operation to set bit $b_{\text{decrement}}$. For the DP threshold, t LE and Mux operations are used. Finally, we sort the small map, i.e., $O(t \log t)$, and reconstruct the t counts. Note that n is the dominating factor as $t \ll n$, i.e., $nt > t \log t$. Overall, HH performs $O(nt)$ operations. □

Theorem 21. PEM *with sorting has complexity* $O(gc \log c)$*, and* PEM *without sorting has complexity* $O(gc)$*, where* $g = \left\lceil \frac{b-\lceil\log k\rceil}{\eta} \right\rceil$ *and* $c = 2^{\lceil\log k\rceil+\eta}$.

Proof. First, we consider PEM with sorting. For each group PEM sorts all c candidates which requires $O(c \log c)$ operations, and performs k comparisons (LE) and oblivious selections (Mux). Finally, k (sorted) indices are returned. Overall, PEM with sorting requires $O(c \log c)$ operations per group.

PEM, without sorting, requires c comparisons per group to find the lowest candidate count (used in the threshold). Then, PEM iterates over c elements per group (instead of k elements as with sorting). Finally, c indices and counts are returned and the parties can sort them themselves. Altogether, PEM without sorting requires $O(c)$ operations per group. □

Note that the summation of user reports per prefix candidates (line 8 in Algorithm 15) does not require any interaction between the computation parties, as addition can be computed locally.

7.2.4 Security

Recall, we consider the semi-honest model introduced by Goldreich [Gol09] where corrupted protocol participants do not deviate from the protocol but gather everything created during the run of the protocol. Our protocols HH and PEM consists of multiple subroutines realized with MPC protocols listed in Table 2.3 and we apply the composition theorem [Gol09, Section 7.3.1] to analyze the overall security. Basically, protocols based on an ideal functionality remain secure if the ideal functionality is realized with a secure protocol providing the same functionality. We implement the ideal functionalities $\mathcal{F}_{\mathsf{HH}}, \mathcal{F}_{\mathsf{PEM}}$ as HH, PEM with secure computation frameworks MP-SPDZ [Kel20] and SCALE-MAMBA [AKR⁺20] (and compare their performance in Section 7.3).

Now, we show the existence of simulators as defined in Section 4.1 for our protocols.

Theorem 22. *Protocol* HH *realizes* $\mathcal{F}_{\mathsf{HH}}$ *in the presence of semi-honest adversaries.*

Proof. Simulator Sim, given final outputs V, C (i.e., $\{y_i\}_{i\in\mathcal{P}}$) can produce a transcript for $\mathsf{real}_{\mathsf{HH}}$ by replacing all secret shared values with randomness. Note that all values in our protocols are secret shared (marked with $\langle\cdot\rangle$) and computationally indistinguishable from randomness (except with negligible probability in the security parameter for some operations, e.g., integer comparisons [AKR⁺20]). The only values that are not secret shared are publicly known iteration counts (i.e., data size and map size t for HH, and number of groups and number of candidates in PEM). Finally, the simulator ensures the expected reconstruction, i.e, V, C, is produced by $\mathsf{Rec}(V), \mathsf{Rec}(C)$. Here, the corrupted parties, cannot distinguish actual from simulated reconstruction as they cannot see the actual randomness (secret shares) from the other parties. □

Theorem 23. *Protocol* PEM *realizes* $\mathcal{F}_{\mathsf{PEM}}$ *in the presence of semi-honest adversaries.*

Proof. We focus on a transcript for one group of PEM, which can be extended to all groups. Simulator Sim, given S, produces a transcript of $\mathsf{real}_{\mathsf{PEM}}$ as follows: As before, Sim replaces all secret shared values with randomness. Then, in the thresholding step, the index for each candidate c, i.e., $S[c]$ is set such that the reconstruction of S provides the expected result. □

From Semi-honest to Malicious

We consider semi-honest computation parties and design our protocol accordingly. However, SCALE-MAMBA provides malicious security, i.e., consistency within the computation is ensured and malicious tampering can be detected. We employ (t, m)-secret sharing, which prevents up to $t-1$ malicious parties to reconstruct the secret. Still, malicious parties (input parties or computation servers) can provide incorrect initial inputs to skew the results, also known as a *data poisoning attack*. Next, we discuss the affect of poisoning attacks on our protocol as well as potential (but not implemented) mitigations. In general, LDP protocols are vulnerable to data poisoning attacks [CSU21, CJG21]. Cryptographic tools, however, can prevent data poisoning attacks and such attacks have limited impact on our protocols HH and PEM: For HH, each input party provides a single value, which can change a count by at most 1; thus, a coalition of r malicious parties, can alter the count by at most r. For PEM, assuming noise is added by the computation parties, each input party provides a count per prefix, i.e., a single bit indicating if a prefix matches their value's prefix (1) or not (0). Thus, r malicious parties can skew the total count *per prefix* by at most r when inputting bits. Ensuring that only one bit is set per input reduces the skew to r *over all prefixes*. This is the best we can hope for as malicious parties are not required to provide the actual inputs from their corrupted parties and can send any bit-vector with a single set bit.

In more detail, the following consistency checks are required for PEM. First, each prefix count ζ must be in $\{0,1\}$, which we check as we operate on integers and not bits[3]. Second, exactly a single count is 1 and the others 0 per user. As PEM might drop rare prefixes between rounds a user might report only zeros. To ensure that always exactly a single 1 is reported, an additional count can be introduced indicating that a user's datum matches none of the considered prefix candidates. The additional count simplifies checking as it allows to reveal the sum s of counts per user which is always 1 for valid inputs. Without the additional count s cannot be revealed as $s = 0$ leaks that none of the prefixes matched and one must, e.g., securely compare $s \leq 1$ which is less efficient. As straightforward comparisons (e.g., $0 \leq \zeta \leq 1$) are expensive for MPC based on secret sharing, we consider alternatives next. First, for each prefix count $p = \zeta^c_{d_j}$ of prefix candidate c from user j, we compute $p' = p^2 - p$ and reveal p' to check $p' = 0$. Note that $p' = 0$ only if $p \in \{0,1\}$, thus, $p' \neq 0$ identifies cheaters. Second, for each user j, we reveal the sum over all prefix candidates, i.e., $s = \sum_{c \in C} \zeta^c_{d_j}$, and check $s = 1$[4]. Combined with the first check, this tells us that only a single count was 1 and the others 0. For PEM, we evaluated the overhead of consistency checks to detect malicious inputs in Section 7.3.4. Similar checks for HH are not possible, as each party inputs a single value from an unknown domain.

Distributed Noise Generation

Distributed noise generation in the presence of malicious parties is not possible without additional noise or computation overhead. For example, honest parties have to provide more noise as the malicious parties might not provide any noise; see, e.g., Ács et al. [ÁC12, Section 8.3] for a detailed analysis of the required noise increase. To achieve optimal noise magnitudes in the presence of malicious parties the noise can be sampled securely by evaluating the inverse cumulative density function. Thus, Laplace noise Laplace(b) can be computed as $(-1)^s b \log(r)$ given uniform random $s \in \{0,1\}, r \in (0,1]$[5]. Similarly, Gumbel noise Gumbel(b) can be computed as $-b\log(-\log(r))$ for uniform random $r \in (0,1]$. However, securely computing logarithms incurs additional computation costs [AS19], which we do not consider, as we assume semi-honest parties like most LDP protocols [EPK14, FPE16a, BS15, BNST17].

Outsourcing

To outsource the computation the *n input parties* send shares of their input to *m computation parties* which run the secure computation on their behalf. The latter can be a subset of the input parties or non-colluding untrusted servers (e.g., multiple cloud service providers). After sending their secret shared value for HH or candidate counts for PEM the input parties can go offline.

[3] An alternative is to require secret-shared bits as inputs. However, to compute the count per prefix these bits must be converted to secret-shared integers, which was slower than checking if integers are in $\{0,1\}$ in our evaluation.

[4] Alternatively, one can compute and reveal sum s' over all p' and check $s' = 0$, which requires less reconstructions (a single reconstruction for s' instead of one per p') if it is sufficient to learn if there was cheating, i.e., $s' \neq 0$, but not who cheated. Here, identifying cheaters requires, e.g., binary search over reconstructed partial sums to find users with $p' \neq 0$, which requires interaction for the sequential search steps.

[5] Uniform random numbers can be generated in a distributed manner even in a malicious setting, e.g., by XOR-ing random inputs from each parties (which is random as long as a single party provides actual randomness) [JWEG18, Supplementary Material], see also Section 5.2.4, or by using the randomness generated in the offline phase [AKR+20].

7.3 Evaluation

We implement our protocols with SCALE-MAMBA [AKR+20] (malicious security) as well as MP-SPDZ [Kel20] (semi-honest security) using Shamir secret sharing with honest majority, and default settings, i.e., 128-bit modulus and statistical security parameter $\kappa = 40$. Code can be largely re-used between these frameworks as MP-SPDZ [Kel20] is a fork of SCALE-MAMBA's predecessor SPDZ2.

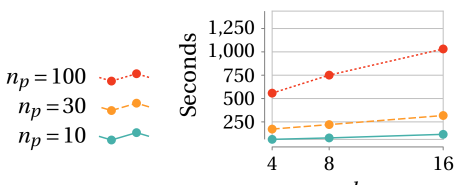
Setup

We briefly recall the evaluation setup from Section 4.4.1. We evaluated the running time and communication of the entire protocol, i.e., offline as well as online phase, in a real-world WAN for $m = 3$ parties. We split the computation parties into two AWS regions, Ohio (us-east-2) and Frankfurt (eu-central-1), and measured an inter-region round time trip (RTT) of approx. 100 ms with 100 Mbits/s bandwidth. The computation parties already received and combined secret-shared inputs from the input users. We present the average of 10 runs for running time and communication (except MP-SPDZ for $\mathsf{HH}_{\mathsf{threads}}$ with 3 runs) and 20 runs for accuracy with 95% confidence intervals, but omit the intervals in most cases, as the results are very stable. We used modest hardware, t2.medium AWS instances (2 GB RAM, 4vCPUs) [Ama20b], to show that the computational overhead of modern MPC is acceptable. More powerful hardware did not provide significant improvements. Recall, $\mathsf{HH}_{\mathsf{threads}}$ is a parallelized version of HH (Section 7.2.1), which required c4.2xlarge instances (15 GB RAM, 8vCPUs) to leverage 8 threads. Also, t2.large (8 GB RAM, 4vCPUs) instances were used for PEM in two settings – MP-SPDZ with $\eta = 5, k = 16$, and SCALE-MAMBA with $\eta = 4, k = 16$ – as more memory was required for these larger programs. To evaluate running time and communication of HH, we set map size $t = k$, and fix it to 16 in our accuracy evaluation (Section 7.3.5). We stress that we evaluated a worst-case scenario for PEM: Each round assumes that the maximum of k prefix candidates are output after thresholding. Fewer outputs decrease computation and communication due to smaller candidate sets for the next round. Sensitivity $\Delta > k$ requires less privacy budget if realized with Gumbel noise, which is not differentially private by itself [DR19a], thus, we cannot release noisy counts and require sorting.

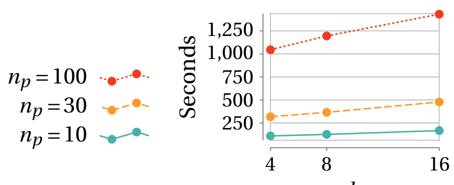
Next, we describe how we compare approaches using different notions of differential privacy. Then, we evaluate the accuracy, running time and communication of our protocols in a real-world WAN.

7.3.1 Comparing different DP notions

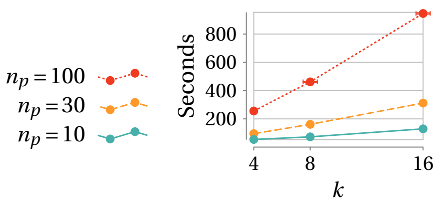
In our evaluation, we use the same value for ϵ to compare our approach to state-of-the-art PEMorig for heavy hitter detection in the local model. Our protocols, however, operate in the central model realized with MPC and approximate differential privacy ($\delta > 0$), whereas PEMorig is a local model protocol with pure differential privacy ($\delta = 0$). The main benefit of approximate DP is improved composition [DR14, Section 3.5], i.e., running g mechanisms on the same data requires a smaller privacy budget of $\approx \sqrt{g}\epsilon$ instead of $g\epsilon$ for large enough g. However, we run PEM once per disjoint subsets of the data and not multiple times on the same data. Thus, we gain no significant advantage over PEMorig from using approximate DP. Furthermore, for an advantage to become noticeable one requires large values of g (see Lemma 1 in Section 2.2.3.)



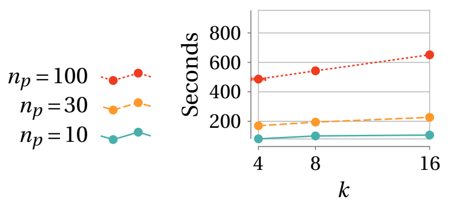
(a) MP-SPDZ: HH



(b) MP-SPDZ: $\text{HH}_{\text{threads}}$

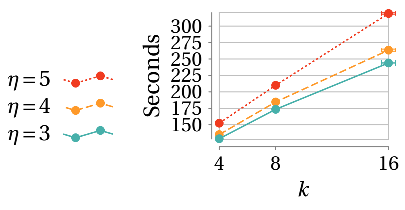


(c) SCALE-MAMBA: HH

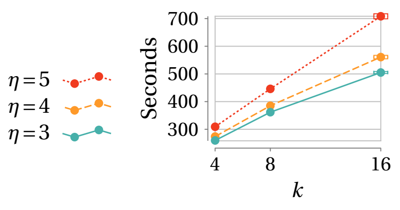


(d) SCALE-MAMBA: $\text{HH}_{\text{threads}}$

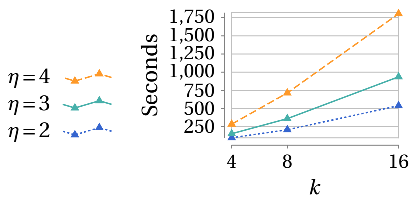
Figure 7.3: Running time of HH, $\text{HH}_{\text{threads}}$.



(a) MP-SPDZ: PEM, $b = 32$



(b) MP-SPDZ: PEM, $b = 64$



(c) SCALE-MAMBA: PEM, $b = 32$

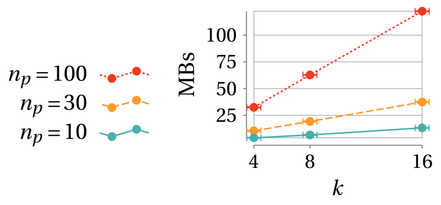
Figure 7.4: Running time of PEM.

7.3.2 Running Time

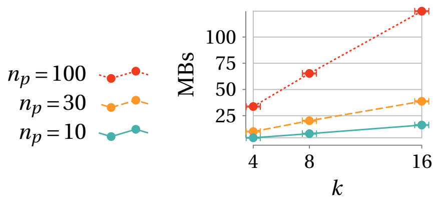
Figures 7.3, 7.4 show the running times for HH, PEM implemented with MP-SPDZ as well as SCALE-MAMBA with data sizes $n_p \in \{10, 30, 100\}$ *per computation party* p i.e., $|D| \in \{30, 90, 300\}$.

To show the difference between HH and $\text{HH}_{\text{threads}}$, we used the same scale for MP-SPDZ (Figures 7.3a, 7.3b) and SCALE-MAMBA (Figures 7.3c, 7.3d). For MP-SPDZ, the running time with 8 threads increases, whereas it decreases with SCALE-MAMBA. Overall, for HH, and especially $\text{HH}_{\text{threads}}$, SCALE-MAMBA is faster than MP-SPDZ, requiring at most 11 minutes for $\text{HH}_{\text{threads}}$, and less than 16 for HH.

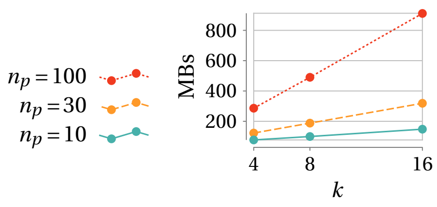
The opposite is the case for PEM: MP-SPDZ is much faster, taking less than 6 minutes for $\eta = 5$, whereas SCALE-MAMBA requires almost half an hour for $\eta = 4$. Note that we used smaller values of η for SCALE-MAMBA (i.e., $\eta \in \{2, 3, 4\}$) since the differences to MP-SPDZ are already sufficiently pronounced here.



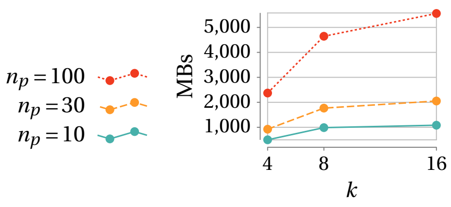
(a) MP-SPDZ: HH



(b) MP-SPDZ: $\mathrm{HH_{threads}}$

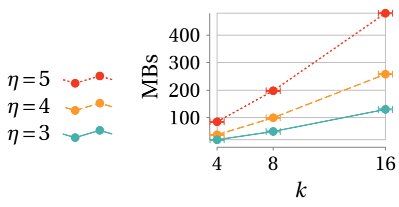


(c) SCALE-MAMBA: HH

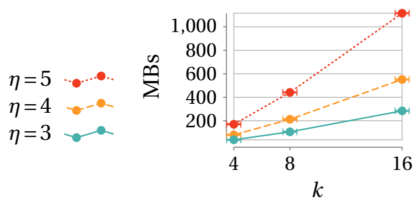


(d) SCALE-MAMBA: $\mathrm{HH_{threads}}$

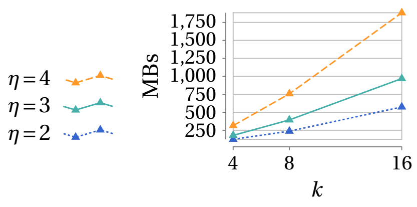
Figure 7.5: Communication per party for HH, $\mathrm{HH_{threads}}$.



(a) MP-SPDZ: PEM, $b = 32$



(b) MP-SPDZ: PEM, $b = 64$



(c) SCALE-MAMBA: PEM, $b = 32$

Figure 7.6: Communication per party for PEM.

7.3.3 Communication

Figure 7.5 shows the communication per computation party for HH and $\mathrm{HH_{threads}}$ and Figure 7.5 shows the communication for PEM.

Client Communication

For HH, a client (input party) sends her secret-shared value to each of the m servers (computation parties). In total, a client sends $m \cdot 128$ bits (our evaluated share size is 128 bits). For PEM, a client sends $2^{\lceil \log k \rceil + \eta}$ secret-shared counts, i.e., at most $m \cdot 8\,KB$ (our largest evaluation with $\eta = 5$, $k = 16$).

Server Communication

As is to be expected, semi-honest MP-SPDZ always sends less than maliciously secure SCALE-MAMBA. We briefly evaluated MP-SPDZ with malicious security (Section 7.3.6), and found it to be still more communication-efficient, albeit slower, than SCALE-MAMBA. Next, we discuss the average communication of HH and PEM per party for $k = 16$. For HH the communication increases linearly with the data size. We consider data size n_p per computation party $p \in \{1, 2, 3\}$, and MP-SPDZ requires ≈13/38/122 MB for n_p 10/30/100. While SCALE-MAMBA provides better running times than MP-SPDZ for $\mathsf{HH}_{\mathsf{threads}}$, MP-SPDZ requires much less communication, e.g., roughly 45 times less for $\mathsf{HH}_{\mathsf{threads}}$ with $k = 16, n_p = 100$ (125 MB vs 5.6 GB), suggesting superior communication batching and parallelization from SCALE-MAMBA compared to MP-SPDZ. For PEM and $b = 32$, MP-SPDZ sends ≈130/258 MB and SCALE-MAMBA sends ≈989/1884 MB for η 3/4. Doubling the domain bit-length to 64 also roughly doubles the communication. Note that PEM, unlike HH, is independent of the data size, as we now consider aggregated candidate counts and not single values.

7.3.4 Malicious Security

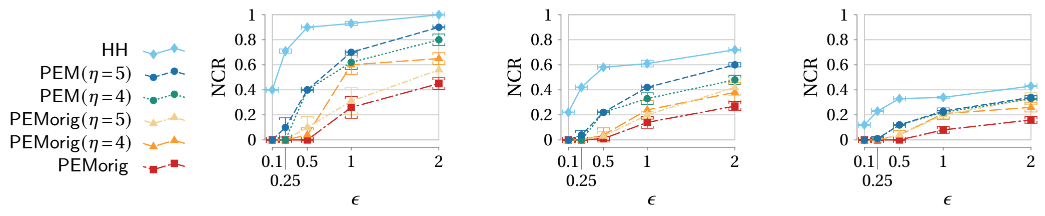
HH is maliciously secure when its implementation is maliciously secure. Unlike PEM, there is no additional check on client inputs for HH. Each HH client inputs a single value from an unknown domain and malicious clients are not required to input their actual value. However, r malicious clients can skew the count of a value by at most r. To prevent malicious client inputs for PEM, however, one can additionally check each input count p by computing $p' = p^2 - p$ as detailed in Section 7.2.4. For 10,000 operations[6] $p^2 - p$, we give the average of 10 runs as before with 95% confidence intervals (omitted if close to zero) on 3 t2.medium instances in a WAN. SCALE-MAMBA with malicious servers requires 32.4 ± 3.9 seconds and 43.5 ± 2.9 MB/party. MP-SPDZ with semi-honest / malicious servers requires only 1.01 / 3.44 ± 0.3 seconds and 0.32 / 2.88 MB/party (mainly due to a leaner offline phase than SCALE-MAMBA). For $n = 1,000$ clients and our largest evaluation with 256 counts per client (i.e., $k = 16, \eta = 5$), the checking overhead is approximately 30 seconds / 90 seconds for MP-SPDZ with semi-honest / malicious servers.

7.3.5 Accuracy

We measure accuracy of k heavy hitters like Wang et al. [WLJ19] via *normalized cumulative rank* (NCR) as in Definition 16 in Section 4.3. Basically, the most frequent element has score k, the next most frequent one $k - 1$, etc., and NCR is computed as the sum of scores for our (at most k) detected heavy hitters divided by the optimal score ($\sum_{i=1}^{k} i$).

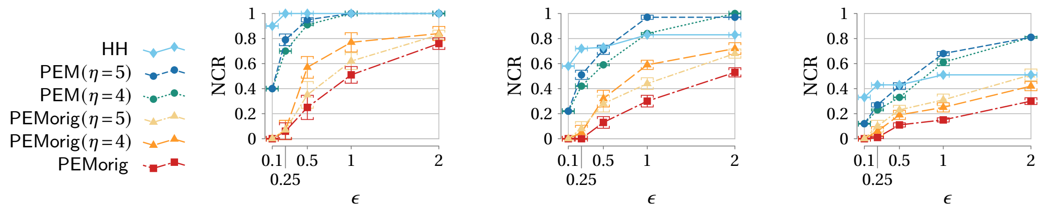
For the accuracy evaluation, we set $\Delta = 1, \delta = 10^{-7}$, assume domain bit-length $b = 32$, and report the average of 20 runs with 95% confidence intervals. Like Wang et al. [WLJ19], we use a synthetic data set sampled from the Zipf distribution with parameter 1.5, i.e., the j-th most frequent value appears with probability proportional to $1/j^{1.5}$. We also used prices from an Online retail data set [ULB19]. Note that we use small data sizes of only a few thousand on purpose, as this is the most challenging regime for DP, as the ratio of "signal" (i.e., actual counts) to noise is small. We compare PEM and PEMorig for different values of $\eta \in \{4, 5\}$, where η is given in brackets (e.g., "PEM($\eta = 4$)"), as well as with PEMorig with query limit count of 2^{20} (denoted as "PEMorig"), where η is set to the largest integer satisfying $g2^{\gamma+\eta} < 2^{20}$ for $g = \lceil (b - \gamma)/\eta \rceil$ groups and $\gamma = \lceil \log_2 k \rceil$ as suggested [WLJ19]. Figures 7.7, 7.9, show NCR for PEM with data size 1 000

[6] Implemented in a loop with annotation `@for_range_parallel()` (provided by MP-SPDZ and SCALE-MAMBA) to process the loop bodies in parallel as they are independent, i.e., result from loop i is not used in loop $i + 1$.



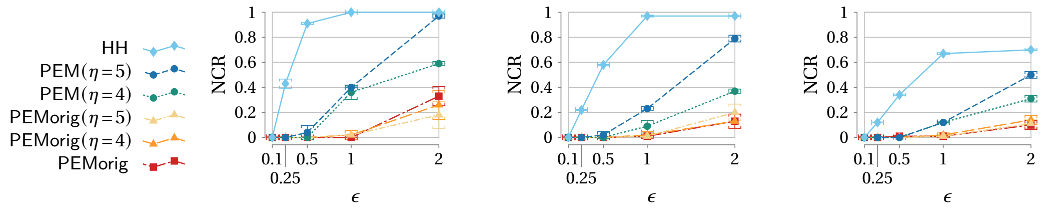
(a) $k = 4$ (b) $k = 8$ (c) $k = 16$

Figure 7.7: NCR of PEM variants and HH for Zipf with fixed $n = 1000$, and varying $k \in \{4, 8, 16\}$, $\epsilon \in \{0.1, 0.25, 0.5, 1, 2\}$.



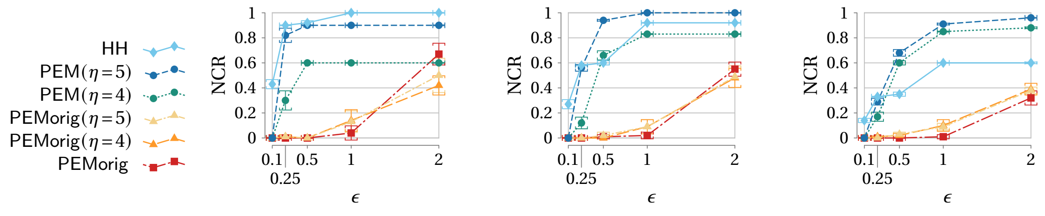
(a) $k = 4$ (b) $k = 8$ (c) $k = 16$

Figure 7.8: NCR of PEM variants and HH for Zipf with fixed $n = 5\,000$, and varying $k \in \{4, 8, 16\}$, $\epsilon \in \{0.1, 0.25, 0.5, 1, 2\}$.



(a) $k = 4$ (b) $k = 8$ (c) $k = 16$

Figure 7.9: NCR of PEM variants and HH for retail data [ULB19] with fixed $n = 1\,000$, varying $k \in \{4, 8, 16\}$, $\epsilon \in \{0.1, 0.25, 0.5, 1, 2\}$.

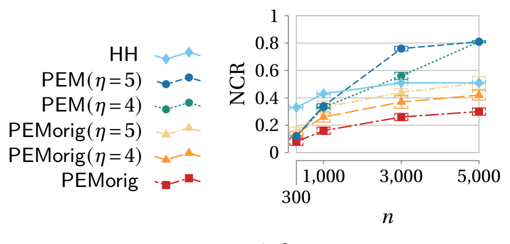


(a) $k = 4$ (b) $k = 8$ (c) $k = 16$

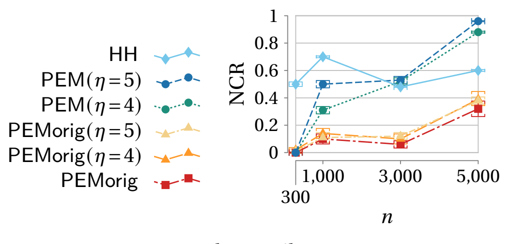
Figure 7.10: NCR of PEM variants and HH for retail data [ULB19] with fixed $n = 5\,000$, varying $k \in \{4, 8, 16\}$, $\epsilon \in \{0.1, 0.25, 0.5, 1, 2\}$.

for $\epsilon \in \{0.1, 0.25, 0.5, 1, 2\}$, where we vary $k \in \{4, 8, 16\}$. Likewise for Figures 7.8, 7.10, however, with larger data size 5000.

First, we focus on comparing PEM with PEMorig. Figure 7.7c shows that for large k (16) and small Zipf data size (1 000), the difference between all approaches is not too strong, still HH, PEM provide better results. However, when we increase the data size (5 000) in Figure 7.8c the accuracy of PEM rises much faster than PEMorig (and its variations with fixed η). We make the same observation, with the real-world data set in Figures 7.9c, 7.10c, i.e., PEM is more accurate and its

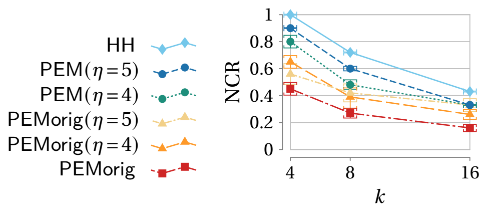


(a) Zipf

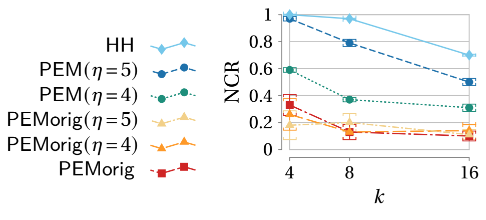


(b) Retail

Figure 7.11: NCR of PEM variants and HH for fixed $\epsilon = 2$, $k = 16$, varying $n \in \{300, 1\,000, 3\,000, 5\,000\}$.

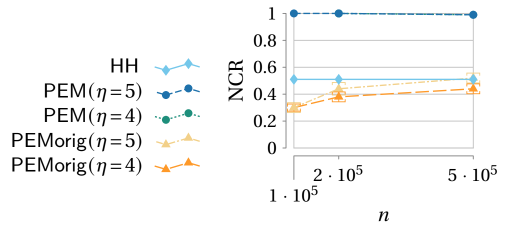


(a) Zipf

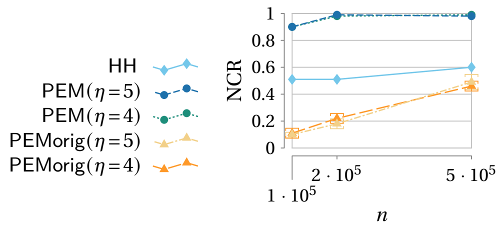


(b) Retail

Figure 7.12: NCR of PEM variants and HH for fixed $\epsilon = 2$, $n = 1\,000$, varying $k \in \{4, 8, 16\}$.



(a) Zipf



(b) Retail

Figure 7.13: NCR of PEM variants and HH for fixed $\epsilon = 0.25$, $k = 16$, varying $n \in \{10^5, 2 \cdot 10^5, 5 \cdot 10^5\}$.

accuracy improves faster when the data size increases. Overall, PEM provides higher accuracy than PEMorig.

Next, we fix $t = 16$ and compare HH to PEM. The choice of map size t provides the following trade-off: keeping t fixed (to a small value) while increasing k decreases accuracy; however, small values for t provide better efficiency for our MPC protocol. With data size 1 000 (Figures 7.7, 7.9) HH provides the best accuracy. For data size 5 000 (Figures 7.8, 7.10) and $k = 4$, HH still provides the best accuracy, however, PEM improves upon HH for $k > 4$. Altogether, the empirical evaluation confirms our analysis in Section 7.1.3: HH provides better accuracy for small data sizes with modest values for t.

In Figure 7.12 we fix $\epsilon = 2, n = 1\,000$ and vary $k \in \{4, 8, 16\}$. As expected, when we increase k while keeping n fixed (and small), the accuracy decreases for all evaluated approaches. However, as shown in Figure 7.11 – where we fix $k = 16, \epsilon = 2$ and vary $n \in \{300, 1000, 3000, 5000\}$ – increasing the data size improves accuracy, as the candidates receive more counts, which can more easily surpass the DP thresholds.

Our protocols, especially PEM, also provide higher accuracy than local-model equivalents for large data sizes, e.g., 10^5, as visualized in Figure 7.13. We omitted the comparison to PEMorig with $\eta > 5$ as the evaluation did not finish after 12 hours on our modest hardware. While HH

Data	k	HH	PEM ($\eta = 4$)	PEM ($\eta = 5$)	PEMorig
Zipf	4	2.6%	1.6%	2.0%	14.8%
	8	4.7%	1.4%	2.3%	16.7%
	16	4.0%	0.6%	2.0%	20.0%
Retail	4	1.1%	-2.1%	-0.5%	6.1%
	8	3.6%	-0.6%	1.5%	9.2%
	16	4.1%	0.0%	1.2%	48.0%

(a) $n = 1,000$.

Data	k	HH	PEM ($\eta = 4$)	PEM ($\eta = 5$)	PEMorig
Zipf	4	0.9%	2.0%	1.4%	25.0%
	8	7.0%	3.4%	5.6%	25.1%
	16	5.7%	5.2%	5.0%	47.1%
Retail	4	1.5%	5.3%	10.9%	3.6%
	8	4.9%	3.5%	1.0%	15.8%
	16	5.0%	7.2%	5.4%	8.1%

(b) $n = 5,000$.

Table 7.1: (NCR−F1)/NCR for fixed n, varying $k \in \{4, 8, 16\}$ averaged over $\epsilon \in \{0.1, 0.25, 0.5, 1, 2\}$.

Data	ϵ	HH	PEM ($\eta = 4$)	PEM ($\eta = 5$)	PEMorig
Zipf	0.1	0.0%	0.0%	0.0%	0.0%
	0.25	4.3%	0.0%	0.0%	0.0%
	0.5	3.0%	0.0%	0.0%	0.0%
	1	5.9%	0.0%	4.3%	50.0%
	2	7.0%	3.0%	5.9%	50.0%
Retail	0.1	0.0%	0.0%	0.0%	0.0%
	0.25	0.0%	0.0%	0.0%	0.0%
	0.5	2.9%	0.0%	0.0%	100.0%
	1	9.0%	0.0%	0.0%	100.0%
	2	8.6%	0.0%	6.0%	40.0%

(a) $n = 1,000$.

Data	ϵ	HH	PEM ($\eta = 4$)	PEM ($\eta = 5$)	PEMorig
Zipf	0.1	3.0%	0.0%	0.0%	0.0%
	0.25	7.0%	4.3%	3.8%	100.0%
	0.5	7.0%	5.7%	5.3%	45.5%
	1	5.9%	8.2%	8.7%	46.7%
	2	5.9%	7.5%	7.4%	43.3%
Retail	0.1	0.0%	0.0%	0.0%	0.0%
	0.25	3.0%	6.3%	3.6%	0.0%
	0.5	5.4%	8.1%	7.2%	0.0%
	1	8.3%	7.1%	7.6%	0.0%
	2	8.2%	14.8%	8.3%	40.6%

(b) $n = 5,000$.

Table 7.2: (NCR−F1)/NCR for fixed n with $k = 16$ and varying $\epsilon \in \{0.1, 0.25, 0.5, 1, 2\}$.

(with fixed $t = 16$) is outperformed by PEMorig for large enough data sets, i.e., around $n = 5 \cdot 10^5$, PEM already finds almost all k heavy hitters for $n = 10^5$.

F1-Score

We also evaluated accuracy via F1 score, i.e., the harmonic mean of precision and recall (Definition 17 in Section 4.3). Next, we compare the relative difference of NCR to F1, i.e., (NCR−F1)/NCR. If NCR is 0, F1 is 0 as well, and we set the relative difference to 0. A positive value means NCR is larger than F1, which is to be expected. Recall, unlike F1, NCR gives more weight to elements that appear more frequently. However, negative values are possible (e.g., mode was not found).

Table 7.1a presents the relative difference of NCR to F1 averaged over $\epsilon \in \{0.1, 0.25, 0.5, 1, 2\}$ for Zipf and retail data with $n = 1,000$. Table 7.1b presents the same for $n = 5,000$. Overall, the averaged scores for F1 and NCR are very close for our protocols (mostly the difference is below 6%) and further apart for PEMorig (mostly above 6% and up to 48% difference), i.e., our protocols provide superior F1 scores.

Table 7.2a gives the detailed comparisons for each ϵ on Zipf and retail data with $n = 1,000$ for fixed $k = 16$. Likewise, Table 7.2b presents the comparison for $n = 5,000$. Large relative differences for PEMorig result from its comparatively low scores. For example, PEMorig has NCR=0.1, F1=0.06 for $k = 16, n = 1,000, \epsilon = 2$ on retail data with a small absolute differences NCR−F1=0.04 leading to a large relative difference of 40% (last row in Table 7.2a).

7.3.6 MPC Frameworks

We deployed SCALE-MAMBA [AKR+20] version 1.3 and MP-SPDZ [Kel20] version 0.1.8 in our evaluation. Here, we evaluated HH, $\text{HH}_{\text{threads}}$ without the final sorting step.

SCALE-MAMBA: Version 1.3 vs. 1.9

Out-of-the-box, i.e., without adjusting options and runtime switches, SCALE-MAMBA version 1.3 was faster than version 1.9[7] for our protocols. Versions 1.4 to 1.9 mainly added features which our protocols do not rely on (e.g., support for garbled circuits, authenticated bits). We used runtime switch `-dOT` from version 1.9, to reduce offline data creation (for features we are not using), for a fairer comparison with 1.3. Still, in our brief evaluation, we found 1.9 to be somewhat slower:

- For PEM with $k = 8, \eta = 2, b = 32$ runtime increased by around 20% from 1.3 to 1.9 (≈206 vs. 248 s). Without `-dOT` communication almost doubled (≈237 vs. 460 MB), with `-dOT` it remained about the same.
- For $\text{HH}_{\text{threads}}$ with $k = 16$ runtime increased by around 10% from 1.3 to 1.9 (≈600 vs. 667 s). Without `-dOT` communication increased by around 30% (≈5.5 vs. 7.2 GB) with `-dOT` it remained about the same.

MP-SPDZ: Semi-honest vs. Malicious

MP-SPDZ supports semi-honest as well as malicious security for multiple secure computation paradigms (e.g,. Shamir secret sharing, BMR) [Kel20], whereas SCALE-MAMBA only supports malicious security. In Section 7.3 we evaluated semi-honest MP-SPDZ. Next, we briefly compare SCALE-MAMBA and MP-SPDZ for *maliciously secure Shamir*:

- For PEM with $k = 16, \eta = 4, b = 32$, MP-SPDZ is more than twice as fast than SCALE-MAMBA (≈14 vs. 30 minutes) with around 400 MB less communication (≈1.47 vs. 1.88 GB).
- For HH with $k = 16$ and $n_p = 30$ per party $p \in \{1, 2, 3\}$, MP-SPDZ is roughly 27% slower than SCALE-MAMBA (≈6 vs. 4.7 minutes), but requires around 60% less communication (≈192 vs. 313 MB).

This suggests that, for malicious security and considering only running time, PEM is more efficient with MP-SPDZ, whereas HH is more efficient with SCALE-MAMBA.

7.3.7 AWS Costs

AWS t2.medium instances cost less than 5 Cents per hour, and communication of 1 GB costs around 2 Cents (per month) [Ama20b]. If one wants to optimize for cost, we suggest to use an MP-SPDZ implementation: All our MP-SPDZ evaluations for HH, PEM run in less than 30 minutes and require less than 1 GB of communication, hence, even our largest MP-SPDZ evaluation cost less

[7] Version 1.9 was the most recent version at the time of our evaluation; version 1.10 was released in October 2020. `https://github.com/KULeuven-COSIC/SCALE-MAMBA/commit/9eda34e6c6205279efa320c7be9e3d615cd6d2da`

than 5 Cents per computation party. (Except for $k = 16, \eta = 5$ which uses t2.large instances that costs less than 10 Cents per hour.) As a comparison, recall that LDP approach PEMorig requires up to 2^{20} hash computations for each user input. Our evaluation of PEMorig – also on t2.medium instances, without parallelization as this requires additional computational resources – showed running times of *hours* compared to the *minutes* required for PEM.

7.4 Summary

We presented protocols for federated, differentially private top-k discovery with secure multi-party computation. Our central DP approaches, HH and PEM, provide higher accuracy than local DP methods for small number of users, without a trusted third party due to our use of cryptography. HH, based on non-private heavy hitter discovery in data streams [CH10], has a running time linear in the data size but supports unknown domains, and provides better accuracy than PEM for very small data sizes, where local DP methods cannot provide meaningful accuracy. PEM, based on Wang et al. [WLJ19], iteratively finds and extends frequent prefixes, is linear in the bit-length of the data domain, and provides better accuracy than HH for larger data sizes. We implemented our protocols with two MPC frameworks [AKR⁺20, Kel20], compared them, and achieved practical running times of less than 11 minutes in a real-world WAN.

8 Conclusion

First, we summarize the chapters of this thesis in Section 8.1. Then, we briefly discuss directions for future research in Section 8.2.

8.1 Summary

In Chapter 1, we first motivated and introduced the research question of this thesis:

Can distributed parties efficiently and accurately compute statistics over their small data sets without revealing secret inputs and ensuring strong privacy guarantees for the output?

Then, we listed our scientific contributions, i.e., efficient protocols securely computing differentially private statistics with high accuracy. Namely, $\mathsf{EM}_{\mathsf{med}}$ for rank-based statistics (e.g., median), EM^* for decomposable aggregate functions (e.g., ranks, convex loss functions), HH for heavy hitters (e.g., mode) from unknown domains and PEM for heavy hitters from known domains.

In Chapter 2, we provided basic notations and preliminaries used in the following chapters. First, cryptographic tools, i.e., garbled circuits and secret sharing; then, anonymization mechanisms, mainly, the Laplace, exponential, and Gumbel mechanism.

In Chapter 3, we described privacy models for DP, mainly, the local, central, and MPC model, and detailed related work, grouped by the privacy models.

In Chapter 4, we detailed our assessment methodology. We ensured semi-honest security of our protocols by combining existing, basic MPC protocols via the MPC composition theorem [Gol09, Section 7.3.1]. Furthermore, we ensured differential privacy of our protocols as they are composed of existing DP mechanisms, and we bounded the total privacy loss via DP composition theorems [LLSY16, Section 2.2.2], [DR14, Theorem 3.20]. We assessed the accuracy of our protocols by comparing them to non-private evaluations. We assessed efficiency of our protocols by measuring their running time and communication in a real-world WAN (Frankfurt–Ohio).

In Chapter 5, we presented our secure two-party protocol $\mathsf{EM}_{\mathsf{med}}$ for rank-based DP statistics. $\mathsf{EM}_{\mathsf{med}}$ is implemented with garbled circuits as well as secret sharing to leverage their respective benefits, i.e., efficient comparisons and arithmetic operations. Our key insight was that sorting simplifies the utility function for rank-based statistics making it almost data independent. Thus, allowing local computation of exponentiations for selection weights, and selection over small data sets instead of the entire data domain. To also support large data sets, we pruned the data. However, pruning required a privacy relaxation as neighboring data sets might be distinguishable, which violates differential privacy. Thus, we employed f-neighboring, a relaxed neighboring notion [HMFS17].

In Chapter 6, we expanded from rank-based statistics to a larger class of functions based on decomposable aggregates without any relaxations. Here, we described our secure multi-party protocol EM^* for decomposable aggregate functions. Our main insight was that decomposability allows local, partial evaluations of utility scores, which can be efficiently combined. We presented multiple alternatives for secure exponentiation used in the exponential mechanism. To handle large domains, we divided the domain in subranges and iteratively selected increasingly smaller subranges with highest utility score.

In Chapter 7, we presented our secure multi-party protocols HH and PEM for DP heavy hitters. Decomposability over subranges is not applicable for heavy hitters (see Section 6.1.1), requiring a new approach. Our main insight was that suitable sketches, i.e., space-efficient data structures, allow efficient MPC for heavy hitters. Related work used sketches which first encode a datum (e.g., by hashing) and then increase an associated counter. However, mapping datums (or prefixes) directly to counters avoids searching for matches (e.g., hashing entire domain). Our protocol HH maps up to t values to their (approximate) frequency in the data and supports unknown domains. Our protocol PEM maps increasingly longer bit-prefixes to their frequency over disjoint subsets of the data, until the bit-prefixes reach the bit-length of the known domain.

In conclusion, we answered the research question in the affirmative by providing efficient MPC protocols for accurate DP statistics ensuring input secrecy as well as output privacy. Our protocols provide high accuracy with efficient running times (seconds to minutes) for distributed parties in real-world networks (100 ms RTT, 100 Mbits/s bandwidth) on modest hardware (mainly, 4 CPU cores at 3.3 GHz and 2 GB RAM per party).

8.2 Directions for Future Research

Recently, MPC has seen more practical applications and the founding of start-ups with MPC as their core business enabler[1] and differential privacy is already widely applied in the industry. However, general-purpose combinations of MPC and DP, especially for the exponential mechanism, are too inefficient for real-world deployments despite their desirable security and privacy guarantees. This thesis presented solutions towards the combined application of MPC and DP in real-world deployments with acceptable overhead.

A general direction for future research is to further increase the efficiency of MPC protocols. Our protocols require basic MPC protocols, listed in Section 2.1.6, as building blocks. Improvements of such building blocks directly improves the efficiency of the MPC protocols relying on them. Alternatively, specialized secure hardware (such as Intel's SGX and AMD's SEV) reduces computational overhead but is vulnerable to various side channel attacks (and, technically, assumes the manufacturer to be a trusted third party). To alleviate side channels based on timing attacks, one can implement constant-time algorithms, which evaluate all conditional branches and prevent leakage of data-dependent information, like our MPC protocols.

A more specific direction is to improve the efficiency of sampling from the distribution induced by the exponential mechanism. To handle large domains, we *iteratively* applied the exponential mechanism to select domain subranges of decreasing size or bit-prefixes of increasing length. An alternative is to find and discard large parts of the domain with negligible probability mass δ and sample from the remaining domain elements with approximate DP. However, this requires a certain problem structure, e.g., combinatorial problems [BDB16, GLM+10]. So far, no efficient, general-purpose equivalent of the exponential mechanism exists for MPC. Hence, specialized MPC protocols are required for efficient solutions, leveraging insights about the problem structure to simplify the required computations.

[1] For example, `https://unboundsecurity.com/` (co-founded by Prof. Y. Lindell and Prof. N. Smart), `https://partisia.com/` (co-founded by Prof. I. Damgård), `https://dualitytech.com/` (co-founded by Prof. S. Goldwasser), `https://zama.ai/` (CTO Dr. P. Paillier).

Author's Publications

[BBK17] Jonas Böhler, Daniel Bernau, and Florian Kerschbaum. Privacy-preserving Outlier Detection for Data Streams. In *IFIP Annual Conference on Data and Applications Security and Privacy*, DBSEC, 2017. http://www.fkerschbaum.org/dbsec17b.pdf.

[BK20a] Jonas Böhler and Florian Kerschbaum. Secure Multi-party Computation of Differentially Private Median. In *USENIX Security Symposium*, USENIXSec, 2020. https://www.usenix.org/system/files/sec20-bohler.pdf.

[BK20b] Jonas Böhler and Florian Kerschbaum. Secure Sublinear Time Differentially Private Median Computation. In *Network and Distributed Systems Security Symposium*, NDSS, 2020. https://www.ndss-symposium.org/wp-content/uploads/2020/02/24150-paper.pdf.

[BK21] Jonas Böhler and Florian Kerschbaum. Secure Multi-party Computation of Differentially Private Heavy Hitters. In *Computer and Communications Security*, CCS, 2021. https://doi.org/10.1145/3460120.3484557.

[Böh21] Jonas Böhler. Secure Computation of Differentially Private Mechanisms. In Sushil Jajodia, Pierangela Samarati, and Moti Yung, editors, *Encyclopedia of Cryptography, Security and Privacy*. Springer Berlin Heidelberg, 2021. https://doi.org/10.1007/978-3-642-27739-9_1714-2.

Bibliography

[ABCP13] Miguel Andrés, Nicolás Bordenabe, Konstantinos Chatzikokolakis, and Catuscia Palamidessi. Geo-Indistinguishability: Differential Privacy for Location-Based Systems. In *Proceedings of the annual ACM Conference on Computer and Communications Security*, CCS, 2013.

[ABL$^+$17] Daniel Anderson, Pryce Bevan, Kevin Lang, Edo Liberty, Lee Rhodes, and Justin Thaler. A high-performance algorithm for identifying frequent items in data streams. In *Proceedings of the Internet Measurement Conference*, ICM, 2017.

[Abo18] John M. Abowd. The U.S. Census Bureau Adopts Differential Privacy. In *Proceedings of the annual ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD, 2018.

[ABPP16] David W. Archer, Dan Bogdanov, Benny Pinkas, and Pille Pullonen. Maturity and performance of programmable secure computation. In *IEEE Symposium on Security and Privacy*, SP, 2016.

[ABZS13] Mehrdad Aliasgari, Marina Blanton, Yihua Zhang, and Aaron Steele. Secure Computation on Floating Point Numbers. In *Network and Distributed Systems Security Symposium*, NDSS, 2013.

[ÁC11] Gergely Ács and Claude Castelluccia. I Have a DREAM! (DiffeRentially privatE smArt Metering). In *International Workshop on Information Hiding*, IH, 2011.

[ÁC12] Gergely Ács and Claude Castelluccia. DREAM: DiffeRentially privatE smArt Metering. *arXiv preprint arXiv:1201.2531*, 2012. `https://arxiv.org/pdf/1201.2531.pdf` (Technical Report Version).

[AF10] Ross Anderson and Shailendra Fuloria. On the Security Economics of Electricity Metering. In *Workshop on the Economics of Information Security*, WEIS, 2010.

[AKR$^+$20] Abdelrahaman Aly, Marcel Keller, Dragos Rotaru, Peter Scholl, Nigel P. Smart, and Tim Wood. SCALE–MAMBA documentation. `https://homes.esat.kuleuven.be/~nsmart/SCALE/`, 2020.

[Ama20a] Amazon.com. Amazon Web Services: Instances Types. `https://aws.amazon.com/ec2/instance-types/`, 2020.

[Ama20b] Amazon.com. Amazon Web Services: Pricing. `https://aws.amazon.com/ec2/pricing/on-demand/`, 2020.

[AMFD12] Dima Alhadidi, Noman Mohammed, Benjamin CM Fung, and Mourad Debbabi. Secure distributed framework for achieving ε-differential privacy. In *International Symposium on Privacy Enhancing Technologies Symposium*, PETS, 2012.

[AMP04] Gagan Aggarwal, Nina Mishra, and Benny Pinkas. Secure Computation of the k^{th}-Ranked Element. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, EUROCRYPT, 2004.

[AMP10] Gagan Aggarwal, Nina Mishra, and Benny Pinkas. Secure computation of the median (and other elements of specified ranks). *Journal of Cryptology*, 2010.

[And21] Ross Anderson. *Security engineering: a guide to building dependable distributed systems*. John Wiley & Sons, 2021.

[App16] Apple. WWDC 2016: Engineering Privacy for Your Users, 2016. `https://developer.apple.com/videos/play/wwdc2016/709/`.

[App17] Apple. Apple's Differential Privacy Team: Learning with Privacy at scale, 2017. `https://machinelearning.apple.com/2017/12/06/learning-with-privacy-at-scale.html`.

[AS19] Abdelrahaman Aly and Nigel P Smart. Benchmarking Privacy Preserving Scientific Operations. In *International Conference on Applied Cryptography and Network Security*, ACNS, 2019.

[BBC$^+$21] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. In *IEEE Symposium on Security and Privacy*, SP, 2021.

[BBGN20] Borja Balle, James Bell, Adria Gascon, and Kobbi Nissim. Private summation in the multi-message shuffle model. In *Proceedings of the annual ACM Conference on Computer and Communications Security*, CCS, 2020.

[BC20] Victor Balcer and Albert Cheu. Separating Local & Shuffled Differential Privacy via Histograms. In *Conference on Information-Theoretic Cryptography*, ITC, 2020.

[BDB16] Jeremiah Blocki, Anupam Datta, and Joseph Bonneau. Differentially Private Password Frequency Lists. In *Network and Distributed Systems Security Symposium*, NDSS, 2016.

[BDO14] Carsten Baum, Ivan Damgård, and Claudio Orlandi. Publicly auditable secure multi-party computation. In *International Conference on Security and Cryptography for Networks*, SCN, 2014.

[BDOZ11] Rikke Bendlin, Ivan Damgård, Claudio Orlandi, and Sarah Zakarias. Semi-homomorphic encryption and multiparty computation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, EUROCRYPT, 2011.

[Bea91] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Annual International Cryptology Conference*, CRYPTO, 1991.

[Bea96] Donald Beaver. Correlated pseudorandomness and the complexity of private computations. In *Proceedings of the annual ACM Symposium on Theory of Computing*, STOC, 1996.

[BEM$^+$17] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnes, and Bernhard Seefeld. Prochlo: Strong privacy for analytics in the crowd. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP, 2017.

[BGG$^+$16] Gilles Barthe, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. Proving differential privacy via probabilistic couplings. In *ACM/IEEE Symposium on Logic in Computer Science*, LICS, 2016.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computing. In *Proceedings of the annual ACM Symposium on Theory of Computing*, STOC, 1988.

[BHR12] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *Proceedings of the annual ACM Conference on Computer and Communications Security*, CCS, 2012.

[BICS10] Radu Berinde, Piotr Indyk, Graham Cormode, and Martin J Strauss. Space-optimal heavy hitters with strong error bounds. *ACM Transactions on Database Systems*, 2010.

[BIK$^+$16] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for federated learning on user-held data. In *NeurIPS Workshop on Private Multi-Party Machine Learning*, 2016.

[BJSV15] Dan Bogdanov, Marko Jõemets, Sander Siim, and Meril Vaht. How the estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation. In *International Conference on Financial Cryptography and Data Security*, FC, 2015.

[BKK$^+$16] Dan Bogdanov, Liina Kamm, Baldur Kubo, Reimo Rebane, Ville Sokk, and Riivo Talviste. Students and taxes: a privacy-preserving study using secure computation. In *International Symposium on Privacy Enhancing Technologies Symposium*, PETS, 2016.

[Bla79] George Robert Blakley. Safeguarding cryptographic keys. In *International Workshop on Managing Requirements Knowledge*, MARK, 1979.

[Blo18] Bloomberg News. Google and Mastercard Cut a Secret Ad Deal to Track Retail Sales, 2018.

[BM89] Mihir Bellare and Silvio Micali. Non-interactive oblivious transfer and applications. In *Annual International Cryptology Conference*, CRYPTO, 1989.

[BMR90] Donald Beaver, Silvio Micali, and Phillip Rogaway. The round complexity of secure protocols. In *Proceedings of the annual ACM Symposium on Theory of Computing*, STOC, 1990.

[BNST17] Raef Bassily, Kobbi Nissim, Uri Stemmer, and Abhradeep Guha Thakurta. Practical locally private heavy hitters. In *Advances in Neural Information Processing Systems*, NeurIPS, 2017.

[BO07] Paul Bunn and Rafail Ostrovsky. Secure two-party k-means clustering. In *Proceedings of the annual ACM Conference on Computer and Communications Security*, CCS, 2007.

[BS13] Pierre Bosch and Thomas Simon. On the self-decomposability of the Fréchet distribution. *Indagationes Mathematicae*, 2013.

[BS15] Raef Bassily and Adam Smith. Local, private, efficient protocols for succinct histograms. In *Proceedings of the annual ACM Symposium on Theory of Computing*, STOC, 2015.

[BSRW17] Asia J. Biega, Rishiraj Saha Roy, and Gerhard Weikum. Privacy through solidarity: A user-utility-preserving framework to counter profiling. In *Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR, 2017.

[BST14] Raef Bassily, Adam Smith, and Abhradeep Thakurta. Private empirical risk minimization: Efficient algorithms and tight error bounds. In *Annual IEEE Symposium on Foundations of Computer Science*, FOCS, 2014.

[BV19] Victor Balcer and Salil Vadhan. Differential Privacy on Finite Computers. *Journal of Privacy and Confidentiality*, 2019. `https://doi.org/10.29012/jpc.679`.

[BWAA18] Jeremy Bernstein, Yu-Xiang Wang, Kamyar Azizzadenesheli, and Anima Anandkumar. signSGD: Compressed optimisation for non-convex problems. In *Proceedings of the International Conference on Machine Learning*, PMLR, 2018. `http://proceedings.mlr.press/v80/bernstein18a/bernstein18a.pdf`.

[BZJ06] Michael Barbaro and Tom Zeller Jr. A face is exposed for AOL searcher no. 4417749, 2006. New York Times.

[CCMS19] TH Hubert Chan, Kai-Min Chung, Bruce M Maggs, and Elaine Shi. Foundations of differentially oblivious algorithms. In *Proceedings of the annual ACM SIAM symposium on Discrete Algorithms*, SODA, 2019.

[CDH10] Octavian Catrina and Sebastiaan De Hoogh. Improved primitives for secure multiparty integer computation. In *International Conference on Security and Cryptography for Networks*, SCN, 2010.

[CDSKY20] Seung Geol Choi, Dana Dachman-Soled, Mukul Kulkarni, and Arkady Yerukhimovich. Differentially-Private Multi-Party Sketching for Large-Scale Statistics. In *Proceedings on Privacy Enhancing Technologies*, PETS, 2020.

[CH10] Graham Cormode and Marios Hadjieleftheriou. Methods for finding frequent items in data streams. *The VLDB Journal*, 2010.

[CJG21] Xiaoyu Cao, Jinyuan Jia, and Neil Zhenqiang Gong. Data poisoning attacks to local differential privacy protocols. In *USENIX Security Symposium*, USENIXSec, 2021.

[CMS17] CMS Centers for Medicare & Medicaid Services. Complete 2017 Program Year Open Payments Dataset, 2017. `https://www.cms.gov/OpenPayments/Explore-the-Data/Dataset-Downloads.html`.

[CO15] Tung Chou and Claudio Orlandi. The simplest protocol for oblivious transfer. In *International Conference on Cryptology and Information Security in Latin America*, LATINCRYPT, 2015.

[CSU+19] Albert Cheu, Adam Smith, Jonathan Ullman, David Zeber, and Maxim Zhilyaev. Distributed differential privacy via shuffling. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, EUROCRYPT, 2019.

[CSU21] Albert Cheu, Adam Smith, and Jonathan Ullman. Manipulation Attacks in Local Differential Privacy. In *IEEE Symposium on Security and Privacy*, SP, 2021.

[CU21] Albert Cheu and Jonathan Ullman. The limits of pan privacy and shuffle privacy for learning and estimation. In *Proceedings of the annual ACM Symposium on Theory of Computing*, STOC, 2021.

[CWH+20] Amrita Roy Chowdhury, Chenghong Wang, Xi He, Ashwin Machanavajjhala, and Somesh Jha. Cryptϵ: Crypto-assisted differential privacy on untrusted servers. In *Proceedings of the annual ACM SIGMOD International Conference on Management of data*, SIGMOD, 2020.

[DDR20] Jinshuo Dong, David Durfee, and Ryan Rogers. Optimal Differential Privacy Composition for Exponential Mechanisms. In *Proceedings of the International Conference on Machine Learning*, PMLR, 2020. `http://proceedings.mlr.press/v119/dong20a/dong20a.pdf`.

[Des20] Damien Desfontaines. *Lowering the cost of anonymization.* PhD thesis, ETH Zürich, 2020.

[DFK+06] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Theory of Cryptography Conference*, TCC, 2006.

[DH76] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 1976.

[DJW13] John C. Duchi, Michael I. Jordan, and Martin J. Wainwright. Local privacy and statistical minimax rates. In *Annual IEEE Symposium on Foundations of Computer Science*, FOCS, 2013.

[DKM+06] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. Our Data, Ourselves: Privacy Via Distributed Noise Generation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, EUROCRYPT, 2006.

[DKM19] Cynthia Dwork, Nitin Kohli, and Deirdre Mulligan. Differential Privacy in Practice: Expose Your Epsilons! *Journal of Privacy and Confidentiality*, 2019. `https://doi.org/10.29012/jpc.689`.

[DKS+21] Zeyu Ding, Daniel Kifer, Thomas Steinke, Yuxin Wang, Yingtai Xiao, Danfeng Zhang, et al. The permute-and-flip mechanism is identical to report-noisy-max with exponential noise. *arXiv preprint arXiv:2105.07260*, 2021. `https://arxiv.org/pdf/2105.07260.pdf`.

[DKY17] Bolin Ding, Janardhan Kulkarni, and Sergey Yekhanin. Collecting Telemetry Data Privately. In *Advances in Neural Information Processing Systems*, NeurIPS, 2017.

[DL09] Cynthia Dwork and Jing Lei. Differential privacy and robust statistics. In *Proceedings of the annual ACM Symposium on Theory of Computing*, STOC, 2009.

[DMHVB13] Yves-Alexandre De Montjoye, César A Hidalgo, Michel Verleysen, and Vincent D Blondel. Unique in the crowd: The privacy bounds of human mobility. *Scientific reports*, 2013.

[DMNS06] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography Conference*, TCC, 2006.

[DMRS$^+$15] Yves-Alexandre De Montjoye, Laura Radaelli, Vivek Kumar Singh, et al. Unique in the shopping mall: On the reidentifiability of credit card metadata. *Science*, 2015.

[DN03] Irit Dinur and Kobbi Nissim. Revealing information while preserving privacy. In *Proceedings of the ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS, 2003.

[DP20] Damien Desfontaines and Balázs Pejó. SoK: Differential Privacies. In *Privacy Enhancing Technologies Symposium*, PETS, 2020.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Annual International Cryptology Conference*, CRYPTO, 2012.

[DR14] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 2014.

[DR19a] David Durfee and Ryan Rogers. Practical Differentially Private Top-k Selection with Pay-what-you-get Composition. *arXiv preprint arXiv:1905.04273*, 2019. (Extended version). `https://arxiv.org/abs/1905.04273`.

[DR19b] David Durfee and Ryan Rogers. Practical Differentially Private Top-k Selection with Pay-what-you-get Composition. In *Advances in Neural Information Processing Systems*, NeurIPS, 2019.

[DSZ15a] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *Network and Distributed Systems Security Symposium*, NDSS, 2015.

[DSZ15b] Cynthia Dwork, Weijie Su, and Li Zhang. Private false discovery rate control. *arXiv preprint arXiv:1511.03803*, 2015. `https://arxiv.org/abs/1511.03803`.

[Dwo06] Cynthia Dwork. Differential Privacy. In *International Colloquium on Automata, Languages, and Programming*, ICALP, 2006.

[Dwo08] Cynthia Dwork. Differential privacy: A survey of results. In *International conference on theory and applications of models of computation*, TAMC, 2008.

[EKM$^+$14] Fabienne Eigner, Aniket Kate, Matteo Maffei, Francesca Pampaloni, and Ivan Pryvalov. Differentially private data aggregation with optimal utility. In *Proceedings of the Annual Computer Security Applications Conference*, ACSAC, 2014.

[EKR$^+$18] David Evans, Vladimir Kolesnikov, Mike Rosulek, et al. A pragmatic introduction to secure multi-party computation. *Foundations and Trends® in Privacy and Security*, 2018.

[ElG85] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 1985.

[Eng18] Engineering Cryptographic Protocols Group (Encrypto, TU Darmstadt). ABY documentation. `https://www.informatik.tu-darmstadt.de/media/encrypto/encrypto_code/abydevguide.pdf`, 2018.

[EPK14] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the annual ACM Conference on Computer and Communications Security*, CCS, 2014.

[Eur12] European Data Protection Supervisor. Opinion of the European Data Protection Supervisor on the Commission Recommendation on preparations for the roll-out of smart metering systems, 2012. `https://edps.europa.eu/data-protection/our-work/publications/opinions/smart-metering-systems_en`.

[FPE16a] Giulia Fanti, Vasyl Pihur, and Úlfar Erlingsson. Building a RAPPOR with the unknown: Privacy-preserving learning of associations and data dictionaries. In *Proceedings on Privacy Enhancing Technologies*, PETS, 2016.

[FPE16b] Giulia Fanti, Vasyl Pihur, and Úlfar Erlingsson. Building a RAPPOR with the unknown: Privacy-preserving learning of associations and data dictionaries. *arXiv preprint arXiv:1503.01214*, 2016. (Extended Version). `https://arxiv.org/pdf/1503.01214.pdf`.

[GAM19] Simson Garfinkel, John M Abowd, and Christian Martindale. Understanding database reconstruction attacks on public data. *Communications of the ACM*, 2019.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the annual ACM Symposium on Theory of Computing*, STOC, 2009.

[GK07] Daniel H Greene and Donald E Knuth. *Mathematics for the Analysis of Algorithms*. Springer Science & Business Media, 2007.

[GKM$^+$16] Vipul Goyal, Dakshita Khurana, Ilya Mironov, Omkant Pandey, and Amit Sahai. Do distributed differentially-private protocols require oblivious transfer? In *International Colloquium on Automata, Languages, and Programming*, ICALP, 2016.

[GKMP20] Badih Ghazi, Ravi Kumar, Pasin Manurangsi, and Rasmus Pagh. Private counting from anonymous messages: Near-optimal accuracy with vanishing communication overhead. In *International Conference on Machine Learning*, ICML, 2020.

[GKPL94] Ronald L Graham, Donald E Knuth, Oren Patashnik, and Stanley Liu. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, 1994.

[GLM$^+$10] Anupam Gupta, Katrina Ligett, Frank McSherry, Aaron Roth, and Kunal Talwar. Differentially private combinatorial optimization. In *Proceedings of the annual ACM SIAM symposium on Discrete Algorithms*, SODA, 2010.

[GM84] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of computer and system sciences*, 1984.

[GMP16] Ivan Gazeau, Dale Miller, and Catuscia Palamidessi. Preserving differential privacy under finite precision semantics. In *Theoretical Computer Science*, TCS, 2016.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *Proceedings of the annual ACM Symposium on Theory of Computing*, STOC, 1987.

[Gol09] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge university press, 2009.

[Goo19] Google Developers Blog. Enabling developers and organizations to use differential privacy, 2019. `https://developers.googleblog.com/2019/09/enabling-developers-and-organizations.html`.

[GRR98] Rosario Gennaro, Michael O Rabin, and Tal Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In *Proceedings of the annual ACM Symposium on Principles of distributed computing*, PODC, 1998.

[Gum48] Emil Julius Gumbel. *Statistical theory of extreme values and some practical applications: a series of lectures.* US Government Printing Office, 1948.

[GX17] Slawomir Goryczka and Li Xiong. A comprehensive comparison of multiparty secure additions with differential privacy. In *IEEE Transactions on Dependable and Secure Computing*, TDSC, 2017.

[HEK12] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *Network and Distributed Systems Security Symposium*, NDSS, 2012.

[HEKM11] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*, USENIXSec, 2011.

[HHNZ19] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. SoK: General purpose compilers for secure multi-party computation. In *IEEE Symposium on Security and Privacy*, SP, 2019.

[HKR12] Justin Hsu, Sanjeev Khanna, and Aaron Roth. Distributed private heavy hitters. In *International Colloquium on Automata, Languages, and Programming*, ICALP, 2012.

[HL10] Carmit Hazay and Yehuda Lindell. *Efficient secure two-party protocols: Techniques and constructions.* Springer Science & Business Media, 2010.

[HLK+17] Mikko Heikkilä, Eemil Lagerspetz, Samuel Kaski, Kana Shimizu, Sasu Tarkoma, and Antti Honkela. Differentially private Bayesian learning on distributed data. In *Advances in Neural Information Processing Systems*, NeurIPS, 2017.

[HLM17] Naoise Holohan, Douglas J Leith, and Oliver Mason. Optimal differentially private mechanisms for randomised response. *IEEE Transactions on Information Forensics and Security*, 2017.

[HMFS17] Xi He, Ashwin Machanavajjhala, Cheryl Flynn, and Divesh Srivastava. Composing Differential Privacy and Secure Computation: A case study on scaling private record linkage. In *Proceedings of the annual ACM Conference on Computer and Communications Security*, CCS, 2017.

[IKN+17] Mihaela Ion, Ben Kreuter, Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, David Shanahan, and Moti Yung. Private Intersection-Sum Protocol with Applications to Attributing Aggregate Ad Conversions. Cryptology ePrint Archive, Report 2017/738, 2017. `https://eprint.iacr.org/2017/738`.

[IKN+20] Mihaela Ion, Ben Kreuter, Ahmet Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, Mariana Raykova, David Shanahan, and Moti Yung. On Deploying Secure Computing: Private Intersection-Sum-with-Cardinality. In *IEEE European Symposium on Security and Privacy*, EuroS&P, 2020.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In *Annual International Cryptology Conference*, CRYPTO, 2003.

[IKOS06] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Cryptography from anonymity. In *Annual IEEE Symposium on Foundations of Computer Science*, FOCS, 2006.

[Ilv20] Christina Ilvento. Implementing the Exponential Mechanism with Base-2 Differential Privacy. In *Proceedings of the annual ACM Conference on Computer and Communications Security*, CCS, 2020.

[IR89] Russell Impagliazzo and Steven Rudich. Limits on the provable consequences of one-way permutations. In *Proceedings of the annual ACM Symposium on Theory of Computing*, STOC, 1989.

[JWEG18] Bargav Jayaraman, Lingxiao Wang, David Evans, and Quanquan Gu. Distributed learning without distress: Privacy-preserving empirical risk minimization. In *Advances in Neural Information Processing Systems*, NeurIPS, 2018.

[Kag18] Kaggle.com. Walmart Supply Chain: Import and Shipment. `https://www.kaggle.com/sunilp/walmart-supply-chain-data/data`, 2018. Retrieved: October, 2019, `https://www.kaggle.com/sunilp/walmart-supply-chain-data/data`.

[Kam15] Liina Kamm. *Privacy-preserving statistical analysis using secure multi-party computation.* PhD thesis, University of Tartu, 2015.

[Kel20] Marcel Keller. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *Proceedings of the annual ACM Conference on Computer and Communications Security*, CCS, 2020.

[Kil88] Joe Kilian. Founding cryptography on oblivious transfer. In *Proceedings of the annual ACM Symposium on Theory of Computing*, STOC, 1988.

[KKP12] Samuel Kotz, Tomasz Kozubowski, and Krzystof Podgorski. *The Laplace distribution and generalizations: a revisit with applications to communications, economics, engineering, and finance.* Springer Science & Business Media, 2012.

[KLN$^+$11] Shiva Prasad Kasiviswanathan, Homin K Lee, Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. What can we learn privately? *SIAM Journal on Computing*, 2011.

[KM11] Daniel Kifer and Ashwin Machanavajjhala. No free lunch in data privacy. In *Proceedings of the annual ACM SIGMOD International Conference on Management of data*, SIGMOD, 2011.

[KPR18] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: making SPDZ great again. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, EUROCRYPT, 2018.

[KRSW18] Marcel Keller, Dragos Rotaru, Nigel P Smart, and Tim Wood. Reducing communication channels in MPC. In *International Conference on Security and Cryptography for Networks*, SCN, 2018.

[KS08] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In *International Colloquium on Automata, Languages, and Programming*, ICALP, 2008.

[KSS09] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In *International Conference on Cryptology and Network Security*, ACNS, 2009. (Full version).

[Lin20] Yehuda Lindell. Secure Multiparty Computation (MPC). *Communications of the ACM*, 2020.

[LJA$^+$18] Andrei Lapets, Frederick Jansen, Kinan Dak Albab, Rawane Issa, Lucy Qin, Mayank Varia, and Azer Bestavros. Accessible privacy-preserving web-based data analysis for assessing and addressing economic inequalities. In *Proceedings of the ACM SIGCAS Conference on Computing and Sustainable Societies*, COMPASS, 2018.

[LK15] Peeter Laud and Liina Kamm. *Applications of secure multiparty computation.* Ios Press, 2015.

[LLSY16] Ninghui Li, Min Lyu, Dong Su, and Weining Yang. Differential Privacy: From Theory to Practice. *Synthesis Lectures on Information Security, Privacy, & Trust*, 2016.

[LLV07] Ninghui Li, Tiancheng Li, and Suresh Venkatasubramanian. t-closeness: Privacy beyond k-anonymity and l-diversity. In *International Conference on Data Engineering*, ICDE, 2007.

[LP09] Yehuda Lindell and Benny Pinkas. A Proof of Security of Yao's Protocol for Two-Party Computation. *Journal of Cryptology*, 2009.

[Mau06] Ueli Maurer. Secure multi-party computation made simple. *Discrete Applied Mathematics*, 2006.

[McS09] Frank McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *Proceedings of the annual ACM SIGMOD International Conference on Management of data*, SIGMOD, 2009.

[McS16] Frank McSherry. Differential privacy and Demographics. `https://github.com/frankmcsherry/blog/blob/master/posts/2016-02-06.md`, 2016.

[MDDC16] Luca Melis, George Danezis, and Emiliano De Cristofaro. Efficient Private Statistics with Succinct Sketches. In *Network and Distributed Systems Security Symposium*, NDSS, 2016.

[Mei18] Sebastian Meiser. Approximate and Probabilistic Differential Privacy Definitions. Cryptology ePrint Archive, Report 2018/227, 2018. (Technical Report) `https://eprint.iacr.org/2018/277`.

[MG82] Jayadev Misra and David Gries. Finding repeated elements. *Science of computer programming*, 1982.

[MG20] Andrés Muñoz Medina and Jenny Gillenwater. Duff: A Dataset-Distance-Based Utility Function Family for the Exponential Mechanism. *arXiv preprint arXiv:2010.04235*, 2020. `https://arxiv.org/abs/2010.04235`.

[Mic20a] Microsoft & Harvard University's Privacy Tools and Privacy Insights projects. Snapping Mechanism Notes. `https://github.com/opendp/smartnoise-core/tree/develop/whitepapers/mechanisms/snapping`, 2020.

[Mic20b] Microsoft & Harvard University's Privacy Tools and Privacy Insights projects. The Exponential Mechanism for Medians. `https://github.com/opendp/smartnoise-core/blob/develop/whitepapers/mechanisms/exponential_median/`, 2020.

[Mir12] Ilya Mironov. On significance of the least significant bits for differential privacy. In *Proceedings of the annual ACM Conference on Computer and Communications Security*, CCS, 2012.

[MKGV07] Ashwin Machanavajjhala, Daniel Kifer, Johannes Gehrke, and Muthuramakrishnan Venkitasubramaniam. l-diversity: Privacy beyond k-anonymity. *ACM Transactions on Knowledge Discovery from Data*, 2007.

[MM08] David Mazieres and Damien Miller. Source of arc4random.c, 2008.

[MMP$^+$10] Andrew McGregor, Ilya Mironov, Toniann Pitassi, Omer Reingold, Kunal Talwar, and Salil Vadhan. The limits of two-party differential privacy. In *Annual IEEE Symposium on Foundations of Computer Science*, FOCS, 2010.

[MMSF$^+$10] Andrés Molina-Markham, Prashant Shenoy, Kevin Fu, Emmanuel Cecchet, and David Irwin. Private memoirs of a smart meter. In *Proceedings of the ACM workshop on embedded sensing systems for energy-efficiency in building*, BuildSys, 2010.

[MPRV09] Ilya Mironov, Omkant Pandey, Omer Reingold, and Salil Vadhan. Computational differential privacy. In *Annual International Cryptology Conference*, CRYPTO, 2009.

[MT07] Frank McSherry and Kunal Talwar. Mechanism design via differential privacy. In *Annual IEEE Symposium on Foundations of Computer Science*, FOCS, 2007.

[MTM14] Chris J Maddison, Daniel Tarlow, and Tom Minka. A* sampling. In *Advances in Neural Information Processing Systems*, NeurIPS, 2014.

[MV16] Jack Murtagh and Salil Vadhan. The complexity of computing the optimal composition of differential privacy. In *Theory of Cryptography Conference*, TCC, 2016.

[NP01] Moni Naor and Benny Pinkas. Efficient oblivious transfer protocols. In *Proceedings of the annual ACM SIAM symposium on Discrete Algorithms*, SODA, 2001.

[NPR19] Moni Naor, Benny Pinkas, and Eyal Ronen. How to (not) share a password: Privacy preserving protocols for finding heavy hitters with adversarial behavior. In *Proceedings of the annual ACM Conference on Computer and Communications Security*, CCS, 2019.

[NPS99] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *Proceedings of the ACM conference on Electronic commerce*, EC, 1999.

[NRS07] Kobbi Nissim, Sofya Raskhodnikova, and Adam Smith. Smooth sensitivity and sampling in private data analysis. In *Proceedings of the annual ACM Symposium on Theory of Computing*, STOC, 2007.

[NRVW20] Seth Neel, Aaron Roth, Giuseppe Vietri, and Zhiwei Steven Wu. Oracle Efficient Private Non-Convex Optimization. In *Proceedings of the International Conference on Machine Learning*, PMLR, 2020. `https://proceedings.icml.cc/static/paper_files/icml/2020/354-Paper.pdf`.

[NS08] Arvind Narayanan and Vitaly Shmatikov. Robust de-anonymization of large sparse datasets. In *IEEE Symposium on Security and Privacy*, SP, 2008.

[NSR11] Arvind Narayanan, Elaine Shi, and Benjamin IP Rubinstein. Link prediction by de-anonymization: How we won the kaggle social network challenge. In *The International Joint Conference on Neural Networks*, IJCNN, 2011.

[NW18] Kobbi Nissim and Alexandra Wood. Is privacy privacy? *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 2018.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, EUROCRYPT, 1999.

[PBS12] Pille Pullonen, Dan Bogdanov, and Thomas Schneider. The design and implementation of a two-party protocol suite for Sharemind 3. *CYBERNETICA Institute of Information Security, Tech. Rep.*, 2012.

[PL15] Martin Pettai and Peeter Laud. Combining differential privacy and secure multiparty computation. In *Proceedings of the Annual Computer Security Applications Conference*, ACSAC, 2015.

[Rab81] Mo Rabin. How to Exchange Secrets by Oblivious Transfer. *Technical Memo TR-81*, 1981.

[RN10] Vibhor Rastogi and Suman Nath. Differentially private aggregation of distributed time-series with transformation and encryption. In *Proceedings of the annual ACM SIGMOD International Conference on Management of data*, SIGMOD, 2010.

[Rog20] Ryan Rogers. A Differentially Private Data Analytics API at Scale. In *USENIX Conference on Privacy Engineering Practice and Respect*, PEPR, 2020.

[RSA78] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 1978.

[RSK+10] Indrajit Roy, Srinath TV Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: Security and privacy for MapReduce. In *USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2010.

[RSP+20] Ryan Rogers, Subbu Subramaniam, Sean Peng, David Durfee, Seunghyun Lee, Santosh Kumar Kancha, Shraddha Sahay, and Parvez Ahammad. LinkedIn's Audience Engagements API: A Privacy Preserving Data Analytics System at Scale. *arXiv preprint arXiv:2002.05839*, 2020. https://arxiv.org/abs/2002.05839.

[Sam01] Pierangela Samarati. Protecting respondents identities in microdata release. *IEEE Transactions on Knowledge and Data Engineering*, 2001.

[Sec09] SecureSCM. Security Analysis, 2009. Deliverable D9.2, EU FP7 Project Secure Supply Chain Management (SecureSCM).

[SH12] Mudhakar Srivatsa and Mike Hicks. Deanonymizing mobility traces: Using social network as a side-channel. In *Proceedings of the annual ACM Conference on Computer and Communications Security*, CCS, 2012.

[Sha79] Adi Shamir. How to share a secret. *Communications of the ACM*, 1979.

[Soo18] Gaurav Sood. California Public Salaries Data, 2018. https://doi.org/10.7910/DVN/KA3TS8.

[SS98] Pierangela Samarati and Latanya Sweeney. Protecting privacy when disclosing information: k-anonymity and its enforcement through generalization and suppression. *Technical Report, SRI International*, 1998.

[STU17] Adam Smith, Abhradeep Thakurta, and Jalaj Upadhyay. Is interaction necessary for distributed private learning? In *IEEE Symposium on Security and Privacy*, SP, 2017.

[TKZ16] Hassan Takabi, Samir Koppikar, and Saman Taghavi Zargar. Differentially Private Distributed Data Analysis. In *IEEE International Conference on Collaboration and Internet Computing*, CIC, 2016.

[TLC19] TLC: NYC Taxi and Limousine Commision. Trip Record Data, January 1-5, 2019, 2019. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page.

[ULB18] ULB Machine Learning Group. Credit Card Fraud Detection, 2018. https://www.kaggle.com/mlg-ulb/creditcardfraud/data.

[ULB19] ULB Machine Learning Group. Online Retail, 2019. `https://archive.ics.uci.edu/ml/datasets/Online+Retail+II`.

[US 18] US Census Bureau. American Community Survey 2018, 2018. `https://data.census.gov`.

[Vad17] Salil Vadhan. The complexity of differential privacy. In *Tutorials on the Foundations of Cryptography*. Springer, 2017.

[War65] Stanley L Warner. Randomized response: A survey technique for eliminating evasive answer bias. *Journal of the American Statistical Association*, 1965.

[WBLJ17] Tianhao Wang, Jeremiah Blocki, Ninghui Li, and Somesh Jha. Locally differentially private protocols for frequency estimation. In *USENIX Security Symposium*, USENIXSec, 2017.

[WGSX20] Di Wang, Marco Gaboardi, Adam Smith, and Jinhui Xu. Empirical Risk Minimization in the Non-interactive Local Model of Differential Privacy. *Journal of Machine Learning Research*, 2020. `http://jmlr.org/papers/v21/19-253.html`.

[WHMM21] Sameer Wagh, Xi He, Ashwin Machanavajjhala, and Prateek Mittal. DP-cryptography: Marrying differential privacy and cryptography in emerging applications. *Communications of the ACM*, 2021.

[WLJ19] Tianhao Wang, Ninghui Li, and Somesh Jha. Locally differentially private heavy hitter identification. In *IEEE Transactions on Dependable and Secure Computing*, TDSC, 2019.

[WZL$^+$20] Royce J Wilson, Celia Yuxin Zhang, William Lam, Damien Desfontaines, Daniel Simmons-Marengo, and Bryant Gipson. Differentially private SQL with bounded user contribution. In *International Symposium on Privacy Enhancing Technologies Symposium*, PETS, 2020.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Annual IEEE Symposium on Foundations of Computer Science*, FOCS, 1986.

[YCKB18] Dong Yin, Yudong Chen, Ramchandran Kannan, and Peter Bartlett. Byzantine-robust distributed learning: Towards optimal statistical rates. In *Proceedings of the International Conference on Machine Learning*, PMLR, 2018. `http://proceedings.mlr.press/v80/yin18a/yin18a.pdf`.

[ZKM$^+$20] Wennan Zhu, Peter Kairouz, Brendan McMahan, Haicheng Sun, and Wei Li. Federated Heavy Hitters Discovery with Differential Privacy. In *International Conference on Artificial Intelligence and Statistics*, AISTATS, 2020. `http://proceedings.mlr.press/v108/zhu20a/zhu20a.pdf`.