Private Collaborative Data Cleaning via Non-Equi PSI

Erik-Oliver Blass[1] and Florian Kerschbaum[2]

[1]Airbus, Germany
erik-oliver.blass@airbus.com
[2]University of Waterloo, Canada
florian.kerschbaum@uwaterloo.ca

Abstract

We introduce and investigate the privacy-preserving version of collaborative data cleaning. With collaborative data cleaning, two parties want to reconcile their data sets to filter out badly classified, misclassified data items. In the privacy-preserving (private) version of data cleaning, the additional security goal is that parties should only learn their misclassified data items, but nothing else about the other party's data set. The problem of private data cleaning is essentially a variation of private set intersection (PSI), and one could employ recent circuit-PSI techniques to compute misclassifications with privacy. However, we design, analyze, and implement three new protocols tailored to the specifics of private data cleaning that significantly outperform a circuit-PSI-based approach. With the first protocol, we exploit the idea that a small additional leakage (the size of the intersection of data items) allows for runtime and communication improvements of more than one order of magnitude over circuit-PSI. The other two protocols convert the problem of finding a mismatch in data classifications into finding a match, and then follow the standard technique of using oblivious pseudo-random functions (OPRF) for computing PSI. Depending on the number of data classes, this leads to either total runtime or communication improvements of up to two orders of magnitude over circuit-PSI.

1 Introduction

Data cleaning [17] is the most time-consuming task in data science. Current estimates range from 45% to 80% of the total time in data science is spent on data cleaning [29, 44]. Data cleaning is necessary to prepare high-quality data sets that result in high-quality machine learning models. The more data sources can be used to clean data, improve data quality, and reduce errors, the higher the accuracy of the final model.

A standard collaborative data cleaning scenario consists of two data sources, each comprising a set of pairs of data elements and their corresponding classifications (labels). One wants to find those data elements that have been misclassified, i.e., classified differently in each set. More formally, given two sets $\mathcal{S}_A = \{(x_1, u_1), \ldots, (x_n, u_n)\}$ and $\mathcal{S}_B = \{(y_1, v_1), \ldots, (y_n, v_n)\}$ where x_i, y_j are data elements and u_i, v_j their corresponding labels, compute the set of classification errors $\mathcal{S}_E = \{(y_j, v_j) \in B | \exists (x_i, u_i) \in A : x_i = y_j \wedge u_i \neq v_j\}$.

Examples for detecting such classification errors ("misclassifications") are plentiful: malware classifiers, where two Security Operation Centers (SOCs) detect attacks by the same malware but have assigned it to different classes of malware, medical image classifiers, where experts (doctors) have classified shared medical images but have come to different conclusions, and many more. Finding misclassified data elements is crucial, as these cannot be used in their corresponding application, but have to be cleaned (removed, re-classified) instead.

However, often different data sets come from different, untrusted parties, so data cannot be exchanged in clear text. Ideally, different parties would jointly compute misclassified data from their sets, while at the same time not learning anything else about the other party's set. Private set intersection (PSI) [10] is a common tool to link two data sources without revealing anything but the intersection. Yet, for data

PARAMETERS: Input length n

1. Wait for input $< (x_1, u_1), \ldots, (x_n, u_n) >$ from sender S and $< (y_1, v_1), \ldots, (y_n, v_n) >$ from receiver R.
2. Output $\{i | \exists j : (x_j = y_i) \wedge (u_j \neq v_i)\}$ to R.

Figure 1: Ideal misclassification functionality $\mathcal{F}^{\mathsf{MCLASS}}$

cleaning, the intersection may be too revealing. For example, as the intersection is a superset of the data items to be cleaned, leaking the intersection violates privacy regulations such as the GDPR mandate on data minimization [40]. The GDPR dictates that only data necessary to perform the required functionality can be disclosed.

While private data cleaning is a restricted form of PSI, reducing collaborative private data cleaning to private set intersection at scale is non-trivial. As we will see, one might use a general form of circuit-PSI [4, 20, 31, 33, 35, 37] and construct a strawman protocol (Section 1.3) for detecting mismatching labels $u_i \neq v_j$. Yet, the resulting performance is disappointing (Section 5) since it resorts to generic two-party computation. More clever reductions are not straightforward, since turning a label mismatch into a match (as found by PSI) would require comparing a label to the entire set of possible labels.

In this paper, we present two new ideas how to improve the performance of Private Data Cleaning and scale it to big data sizes of 2^{20} inputs. These two ideas use different insights into the problem of data cleaning which makes them interesting to study in parallel. The first (Section 3) results in protocol PDC_1 that has optimal round complexity ($O(1)$, 1.5 rounds) as well as optimal communication, and optimal computation complexity ($O(n)$). On the downside, PDC_1 brings a small additional leakage (the size of the intersection is revealed to one party), and it uses $O(n)$ public key operations. Hence, the protocol's concrete runtime is sub-optimal.

Consequently, we present a second protocol PDC_2 which does not suffer from any leakage and only uses only a small, constant number of public key operations, and then reverts to symmetric key techniques for the bulk of the work. PDC_2 also features optimal round complexity $O(1)$, 1 or 2 rounds depending on the oblivious pseudo-random function (OPRF) used, but has $O(n \log |\mathcal{L}|)$ communication and computation complexity, where $\mathcal{L}$ is the set of all possible labels. Our extensive benchmarks (Section 5) show that PDC_2's concrete runtime and communication cost is always the best choice when label length $\ell = \log |\mathcal{L}|$ is low to medium (up to $\ell \leq 10$ bits). For larger label lengths $\ell \geq 10$ bits and when differentially private leakage is admissible, PDC_1 becomes the best choice still outperforming circuit-PSI by an order of magnitude.

In summary, we make four contributions:

- We formalize the problem of private collaborative data cleaning and show its relation to private set intersection (Section 1.1)
- We present a new protocol with optimal round, communication, and computation complexity, but small additional leakage and non-optimal use of public-key cryptography (Section 3).
- We present a new protocol with optimal round complexity, leakage, and use of public-key cryptography, but with higher communication and computation complexity (Section 4.4). However, this protocol has the best concrete runtime in our experiments.
- We evaluate our open-source implementation of these protocols and the baseline and report runtimes and communication cost (Section 5). Overall, we show an improvement of up to two orders of magnitude over circuit-PSI.

1.1 Problem Definition

We consider two parties, a sender S and a Receiver R. Sender S has a sequence of pairs $\mathcal{S}_S =< (x_1, u_1), \ldots, (x_n, u_n) >$, and R has a sequence of pairs $\mathcal{S}_R =< (y_1, v_1), \ldots, (y_n, v_n) >$. We call the $x_i, y_j \in \mathcal{D}$ data elements, e.g., x-ray images or cryptographic hashes of malware. Here, $\mathcal{D}$ denotes the domain of all possible data elements. Similar to PSI, we assume that all x_i are unique in $\mathcal{S}_S$, and all y_j are unique in $\mathcal{S}_R$. We call the $u_i, v_j \in \mathcal{L}$ labels, e.g., classes in a classification task that come from domain $\mathcal{L}$. Labels do not need to be unique within either $\mathcal{S}_S$ or $\mathcal{S}_R$. Our goal is to allow receiver R to compute all elements (y_j, v_j) representing a classification error with respect to sequence $\mathcal{S}_S$ of sender S. That is, R should be able compute those (y_j, v_i) pairs where y_j matches an x_i, but the corresponding labels u_i and v_i differ: $\mathcal{S}_E = \{(y_j, v_i) \in \mathcal{S}_R | \exists (x_i, v_i) \in \mathcal{S}_S : x_i = y_j \wedge u_i \neq v_j\}$. We require that this computation is secure in the sense that it reveals nothing else to either S or R. In particular, elements (x_i, u_i) which are also in $\mathcal{S}_S$ (and would be revealed by running PSI on $\{x_i\}$ and $\{y_j\}$) are not revealed.

We formalize our intuition of an ideal misclassification functionality in Figure 1. Note that data elements x_i, y_j are typically bit strings of some length L, $x_i, y_j \in \{0,1\}^L$ with $L = \log |\mathcal{D}|$, and similarly $u_i, v_j \in \{0,1\}^\ell$ with $\ell = \log |\mathcal{L}|$. To work with smaller data types and also save communication, a standard trick is to hash arbitrary long inputs into bit strings of length $\log n + \sigma$, where σ is a statistical security parameter. Similar to related work [4] against which we compare, we implicitly use this technique (if not specified otherwise), too.

1.2 Preliminaries

Throughout the paper, λ denotes a computational security parameter, and σ denotes a statistical security parameter.

We write $i \in [n]$ as a shorthand for $i \in \{1, \ldots, n\}$ and $< x_i >_{i \in [n]}$ for sequence $< x_1, \ldots, x_n >$. We will often use sequences instead of sets to represent data in this paper, as sequences allow indexing over individual elements.

For a length ℓ bit string $B = b_1 \ldots b_\ell$, $\mathsf{Prefix}_i(B) = b_1 \ldots b_i$ denotes the length i prefix of B. We write $\mathsf{Prefix}_i(B) \oplus 1$ as a shorthand for the length i prefix of B where the last bit of the prefix is flipped, i.e., $\mathsf{Prefix}_i(B) \oplus 1 = b_1 \ldots (b_i \oplus 1)$.

We write $B[i]$ to denote the i^{th} bit of B.

For a keyed pseudo-random function (PRF) $\mathsf{PRF} : \{0,1\}^\lambda \times \{0,1\}^\lambda \rightarrow \{0,1\}^\lambda$, a cryptographic hash function $H : \{0,1\}^* \rightarrow \{0,1\}^\lambda$ (modeled as a random oracle), and s variable-length bit strings $x_i \in \{0,1\}^{\ell_i}$, we will write $\mathsf{PRF}_K(x_1 || \ldots || x_s)$ as a shorthand for $\mathsf{PRF}_K(H(x_1 || \ldots || x_s))$. Here, "$||$" denotes an unambiguous pairing of inputs, e.g., concatenation with padding.

Similarly, for Elgamal encryption $\mathsf{Enc} : \mathcal{K} \times \mathcal{M} \rightarrow \mathbb{G}$ over key space $\mathcal{K}$, plaintext space $\mathcal{M}$, and message space $\mathbb{G}$, we write $\mathsf{Enc}_K(x_1 || \ldots || x_s)$ as a shorthand for $\mathsf{Enc}_K(H^{\mathbb{G}}(x_1 || \ldots || x_s))$, where $H^{\mathbb{G}} : \{0,1\}^* \rightarrow \mathbb{G}$ is a cryptographic hash function.

If obvious from the context, we will also omit writing key K for a keyed PRF $\mathsf{PRF}_K(\cdot)$ on input x and simply write $\mathsf{PRF}(x)$.

For two families of random variables (probability ensembles) $\{X\}_{\lambda \in \mathbb{N}}$ and $\{Y\}_{\lambda \in \mathbb{N}}$ we write $X \overset{c}{\equiv} Y$ if they are computationally indistinguishable.

1.3 A Strawman Solution using Circuit-PSI

There already exist variations of PSI that one can use in combination with generic two-party computation (2PC) to perform private data cleaning. On input of sequence $\mathcal{S}_S =< x_1, \ldots, x_n >$ by sender S and sequence $\mathcal{S}_R =< y_1, \ldots, y_n >$ by receiver R, a circuit-PSI protocol [4, 20, 31, 33, 35, 37] offers the following functionality. For each $x_i \in \mathcal{S}_S$, S receives a value α_i, and R receives a value β_i such that $\alpha_i = \beta_i$, *iff* $x_i \in \mathcal{S}_S \cap \mathcal{S}_R$. After executing a circuit-PSI protocol, parties can then use 2PC to compute any functionality on top of the intersection, using the α_i, β_i as input to the 2PC.

Previous works use Oblivious Programmable Pseudo-Random Functions (OPPRFs) to realize circuit-PSI which, interestingly, also allows to include *associate payloads* in the computation of the intersection functionality with 2PC. Here, the associate payload of an element x_i is u_i as in our definition of data cleaning, i.e., parties' inputs are pairs (x_i, u_i) and (y_i, v_i). As we use a circuit-PSI-based strawman construction for data cleaning as our benchmark and to compare our techniques against, we give a brief overview about general circuit-PSI and how our circuit-PSI-based strawman construction works below. For more details on circuit-PSI with associated payload, see Section 6 of Pinkas et al. [33] or Figure 10 of Rindal and Schoppmann [37].

Circuit-PSI. To simply compute shares of the intersection, parties S and R run a circuit-PSI instance using $\mathcal{S}_S =< x_1, \ldots, x_n >$ and $\mathcal{S}_R =< y_1, \ldots, y_n >$ as their input. That is, S chooses n random values t_i and

Table 1: Theoretical communication and computational complexities. n : number of data elements/labels $< x_i, u_i >_{i\in[n]}, < y_i, v_j >_{i\in[n]}$, |DH|: size of element of DDH-group, $\ell = |u_i| = |v_j| \leq \log n + \sigma$ for statistical security parameter σ, λ: computational security parameter, m: size of stash-free cuckoo hash table with three hash functions ($m = 1.27n$), SK: symmetric key operations, PK: public-key operations. For VOLE-PDC_2, we use the VOLE-OPRF communication complexity as specified in Section 7 of Rindal and Schoppmann [37].

Scheme		Communication Complexity	Computational Complexity	Leakage
Circuit-PSI + 2PC		$O(n\ell)$	$O(\lambda)\text{PK} + O(n\ell)\text{SK}$	None
Ours	PDC_1	$13n \cdot \lvert\text{DH}\rvert$	$O(n)\text{PK} + O(n)\text{SK}$	$\lvert X \cap Y\rvert$
	DH-PDC_2	$n\ell \cdot (2 \cdot \lvert\text{DH}\rvert + \lvert H\rvert)$	$O(n\ell)\text{PK} + O(n\ell)\text{SK}$	None
	KKRT-PDC_2	$(5m + 3n)\ell\lambda$	$O(\lambda)\text{PK} + O(n\ell)\text{SK}$	None
	VOLE-PDC_2	$((5 + 2^{17}(m\ell)^{-\frac{19}{20}})m + 3n)\ell\lambda + 2m\ell(\sigma + 2\log(n))$	$O(\lambda)\text{PK} + O(n\ell)\text{SK}$	None

OPPRFs $f_i(y)$ such that $f_i(y) = t_i$ *if* $\exists j : y = x_j$, *else* $f_i(y)$ is random. Sender S and receiver R obliviously evaluate the f_i such that R receives $r_i = f(y_i)$ for y_i. Finally, S and R run a 2PC circuit that outputs a share of bit $(t_i \stackrel{?}{=} r_i)$ to each.

Circuit-PSI-based Strawman. For computing misclassification with circuit-PSI, we follow the associated payloads approach from Rindal and Schoppmann [37] (see also improvements in [35]), but institute a change to the 2PC part as described below.

First, parties use $\mathcal{S}_S =< (x_1, u_1), \ldots, (x_n, u_n) >$ and $\mathcal{S}_R =< (y_1, v_1), \ldots, (y_n, v_n) >$ as their input. Sender S chooses $2n$ random values t_i, t'_i and OPPRFs $f'_i(y)$ such that $f'_i(y) = t_i || t'_i \oplus u_j$, *if* $\exists j : y = x_j$, *else* $f'_i(y)$ is random. Parties obliviously evaluate the f'_i such that R receives $r_i || r'_i = f'(y_i)$ for each y_i.

After the evaluation of the OPPRFs, parties run one circuit in 2PC in our strawman. Namely, parties execute for each (y_i, v_i) a circuit outputting

$$F(r_i, r'_i, v_i, t_i, t'_i) = \begin{cases} 1, & \text{if } r_i = t_i \wedge v_i \neq r'_i \oplus t'_i \\ 0, & \text{otherwise} \end{cases}$$

to R. So, R learns whether (y_i, v_i) is a misclassification if it has received 1 as an output.

In conclusion, one can realize the private data cleaning functionality by essentially running circuit-PSI including the evaluation of a 2PC circuit. While conceptually simple, it turns out that this strawman construction is expensive in practice, as the 2PC computation has high bandwidth requirements, see our evaluation in Section 5. In this paper, we significantly improve both computation and communication overhead for realistic values of $\ell = \log \mathcal{L}$, i.e., the domain of all possible labels.

2 Technical overview of our constructions

We present, analyze, and implement two new ideas to construct more efficient protocols that do not need to resort to generic, expensive 2PC.

2.1 PDC_1

Our first insight is that it suffices if receiver R cannot distinguish between two cases in a PSI protocol: the case $y_j \notin \mathcal{S}_S$, when their element is not in set $\mathcal{S}_S$ of sender S, and the case $x_i = y_j \wedge u_i = v_j$, when their element is in set $\mathcal{S}_S$ of sender S, but has the same label u_i as the corresponding element x_i from S.

Hence, with our first protocol PDC_1 we construct a variant of the basic Diffie-Hellman (DH) key exchange based PSI that evaluates an OPRF [1, 15, 26].

In PDC_1, S and R run OPRFs such that S receives PRF outputs for all data elements from $\mathcal{S}_S$ and $\mathcal{S}_R$ using receiver R's key. At the same time, S also receives Elgamal ciphertexts for all labels in $\mathcal{S}_S$ and $\mathcal{S}_R$, encrypted under R's key. In case there is a PRF output from a data element $\mathcal{S}_R$ matching a PRF output for a data element in $\mathcal{S}_S$, we have $x_i = y_j$. In that case, S sends this PRF output together with the two encrypted labels back to R. Receiver R decrypts these labels, and if they do not match, R has found a misclassification. In case there is no matching PRF output for a data element from $\mathcal{S}_R$ in $\mathcal{S}_S$, S also sends the PRF output to

R, but now together with: 1) the corresponding Elgamal encrypted label, and 2) an Elgamal re-encryption of this encrypted label. Again R decrypts the two encrypted labels and finds matching labels.

Note the security properties that PDC_1 provides. Receiver R obtains, for each y_j, two Elgamal ciphertexts. In case the ciphertexts encrypt different plaintexts, R has found a misclassification. In case they encrypt the same plaintexts, *either* there is a matching x_i, but the labels of x_i and y_i are the same, *or* there is no matching x_i for y_i. Receiver R cannot distinguish the two cases and consequently does not learn any information besides what is specified by ideal functionality $\mathcal{F}^{\mathsf{MCLASS}}$. However, we stress that S learns the intersection cardinality of the data elements, i.e., $|\{x_i\} \cap \{y_j\}|$ which is more than specified by $\mathcal{F}^{\mathsf{MCLASS}}$. As we will see, PDC_1 is still interesting, as it outperforms all other protocols considered in this paper when the length $\ell = |u_i| = |v_i|$ of classification strings increases. Moreover, its additional leakage can be protected by differential privacy and hence be acceptable depending on the scenario.

For the sake of giving an overview, we here omit several additionally required security techniques for PDC_1 like *blinding*, such that, e.g., R cannot compute the actual label. We refer to Section 3 for all technical details.

2.2 PDC_2

Our second insight is that, in order to compare labels for misclassification, we only need to compare their bit representation. In particular, for each mismatching pair of labels there exist unique prefixes of the two label bit strings that differ only in the last bit.

In PDC_2, for each of their labels u_i and v_j, S and R create all possible prefixes. In addition, S flips the last bits of their prefixes. As a result, for a *mismatching* combination $u_i \neq v_j$, there exist exactly one matching combination of prefix from R and modified prefix from S. For a *matching* pair of labels $x_i = v_j$, there exists no matching combination of prefix from R and modified prefix from S.

For each (modified) prefix, PDC_2 then hashes the (modified) prefix together with the corresponding data element x_i or y_i. For length-ℓ classifications $u_i, v_j, |u_i| = |v_j| = \ell$, S computes ℓ hash values $H(x_i||\mathsf{ModifiedPrefix}_1(u_i))$, $\ldots, H(x_i||\mathsf{ModifiedPrefix}_\ell(u_i))$, and R computes ℓ hash values $H(y_j||\mathsf{Prefix}_1(v_j)), \ldots, H(y_j||\mathsf{Prefix}_\ell(v_j))$.

The idea is now to compute private set intersection cardinality (PSI-CA) over two sets of hash values. The resulting cardinality (either 0 or 1) indicates a misclassification.

Again, we omit additional techniques, e.g., how to avoid computing PSI-CA for $O(n^2)$ pairs of sets of size ℓ, but instead compute essentially n PSI-CAs for sets of size ℓ. We refer to Section 4 for all details.

DH-PDC_2. One way to implement a PSI-CA protocol is to permute the OPRF output of a batch of elements. R submits its elements to S in an OPRF protocol. S randomly shuffles the PRFs before returning them to R. Finally, S also sends PRF outputs of their elements, and R computes PSI-CA simply by counting the number of matching PRF outputs while not learning which of their elements match.

In our first protocol, dubbed DH-PDC_2 (Section 4.2), we use the DH-based OPRF as it allows S to shuffle outputs before sending them to R. While computing PSI-CA with a DH-based OPRF approach results in low concrete communication complexity, its concrete computational complexity is high, requiring $3\ell \cdot n$ public key operations.

Vector-PDC_2. Thus, our main construction, dubbed Vector-PDC_2 (Section 4.4), computes PSI-CA with highly efficient OPRFs, such as the KKRT-OPRF [21] and the VOLE-OPRF [37]. These OPRFs require only a number of public key operations that is linear in the security parameter. While they compute over batches of elements, they do not allow to implement shuffling of the output. The inability to shuffle the output turns out to be a major technical problem for implementing PSI-CA.

To remedy, our idea in Vector-PDC_2 is that both S and R use stash-less Cuckoo hashing with three hash functions to hash their data elements into separate Cuckoo hash tables. With Cuckoo hashing and three hash functions, data elements $x_i = y_j$ might end up in three different buckets in their hash tables. For example, let b_1, b_2, b_3 be the three possible bucket indexes that x_i or $y_j = x_i$ can be hashed to. Sender S maps x_i to, e.g., b_2, but R maps y_j to b_3. Our idea is now that R place three "replicas" of $H(y_j||\mathsf{Prefix}_k(v_j))$ into bucket b_3, each blinded with the PRF output of y_i using a PRF key specific to bucket b_1, b_2, and b_3. R sends the resulting blinded table to S.

At the same time, S places $H(x_i||\mathsf{Prefix}_k(u_j))$ into b_2. Then S runs an OPRF with R for each of S' Cuckoo table buckets, using the bucket contents as input and receiving the PRF output with R's bucket

Input of S: n pairs $< (x_1, u_1), \ldots, (x_n, u_n) >, x_i \in \{0,1\}^*, u_i \in \{0,1\}^\ell$, keys $\alpha_1, \alpha_2 \in \mathbb{Z}_p$, public keys g^{β_1}, g^{β_2}
Input of R: n pairs $< (y_1, v_1), \ldots, (y_n, v_n) >, y_i \in \{0,1\}^*, v_i \in \{0,1\}^\ell$, keys $\beta_1, \beta_2 \in \mathbb{Z}_p$, public keys $g^{\alpha_1}, g^{\alpha_2}$
Parameters: Security parameter λ, DDH group $\mathbb{G}$ of prime order p, $|p| = \lambda$, generator g, Elgamal encryption $\mathsf{Enc}, \mathsf{Dec}, \mathsf{UnMask}, \mathsf{UnPeel}, \mathsf{AddEnc}, \mathsf{ReEnc}$, DH-based pseudorandom function family PRF, hash function $H : \{0,1\}^* \to \mathbb{G}$
Protocol:

1. For all $i \in [n]$, S hashes inputs (x_i, u_i) to $X_i = H(x_i)$ and $\Upsilon_i = H(x_i||u_i)$. R hashes inputs (y_i, v_i) to $Y_i = H(y_i)$ and $\Phi_i = H(y_i||v_i)$.
2. S computes the sequence of tuples $< (\mathsf{PRF}_{\alpha_1}(X_i), \mathsf{Enc}_{\alpha_2}(\Upsilon_i)) >_{i\in[n]}$ and sends it to R.
3. R computes the two sequences

$$\mathcal{R}_1 =< (\mathsf{PRF}_{\beta_1\alpha_1}(X_i) = \mathsf{PRF}_{\beta_1}(\mathsf{PRF}_{\alpha_1}(X_i)), \mathsf{Enc}_{\alpha_2\beta_2}(\Upsilon_i) = \mathsf{AddEnc}_{\beta_2}(\mathsf{Enc}_{\alpha_2}(\Upsilon_i))) >_{i\in[n]}$$
$$\mathcal{R}_2 =< (\mathsf{PRF}_{\beta_1}(Y_i), \mathsf{Enc}_{\beta_2}(\Phi_i)) >_{i\in[n]} .$$

   R shuffles $\mathcal{R}_1$ and shuffles $\mathcal{R}_2$ and sends both to S.
4. S computes

$$\mathcal{S} =< (\mathsf{PRF}_{\beta_1}(X_i) = \mathsf{UnMask}_{\alpha_1}(\mathsf{PRF}_{\beta_1\alpha_1}(X_i)), \mathsf{Enc}_{\beta_2}(\Upsilon_i) = \mathsf{UnPeel}_{\alpha_2}(\mathsf{Enc}_{\alpha_2\beta_2}(\Upsilon_i))) >_{i\in[n]} .$$

   S creates an empty sequence Σ. For each pair $(\mathsf{PRF}_{\beta_1}(Y_i), \mathsf{Enc}_{\beta_2}(\Phi_i)) \in \mathcal{R}_2$,
   - If $\exists(\mathsf{PRF}_{\beta_1}(X_j), \mathsf{Enc}_{\beta_2}(\Upsilon_j)) \in \mathcal{S} : \mathsf{PRF}_{\beta_1}(X_j) = \mathsf{PRF}_{\beta_1}(Y_i)$, then S computes tuple $(a_i = \mathsf{PRF}_{\beta_1}(Y_i), b_i = \mathsf{PRF}_{\alpha_1}(\mathsf{Enc}_{\beta_2}(\Phi_i)), c_i = \mathsf{PRF}_{\alpha_1}(\mathsf{Enc}_{\beta_2}(\Upsilon_j))$ and appends it to Σ.
   - If $\nexists(\mathsf{PRF}_{\beta_1}(X_j), \mathsf{Enc}_{\beta_2}(\Upsilon_j)) \in \mathcal{S} : \mathsf{PRF}_{\beta_1}(X_j) = \mathsf{PRF}_{\beta_1}(Y_i)$, then S computes tuple $(a_i = \mathsf{PRF}_{\beta_1}(Y_i), b_i = \mathsf{PRF}_{\alpha_1}(\mathsf{Enc}_{\beta_2}(\Phi_i)), c_i = \mathsf{ReEnc}_{g^{\beta_2}}(b_i)$ and appends it to Σ.

   S sends Σ to R.
5. R computes $< (a_i, b_i' = \mathsf{Dec}_{\beta_2}(b_i), c_i' = \mathsf{Dec}_{\beta_2}(c_i)) >_{i\in[n]}$. For each Y_i from R's input, where $\mathsf{PRF}_{\beta_1}(Y_i) = a_i$, but $b_i' \neq c_i'$, R outputs i.

Figure 2: Linear misclassification protocol PDC_1

specific key. For each data element x_i, S takes the three buckets b_1, b_2, b_3 from R's blinded table, and "unblinds" the replica that corresponds to the bucket R has the PRF output from. To avoid that parties learn at which bit position u_i and v_j differ, there is an additional blinding step (from S) and unblinding (from R) required. For more details, we refer to Section 4.4.

We summarize theoretical communication and computational complexities in Table 1.

3 PDC_1

With protocol PDC_1, we improve over the strawman solution by avoiding secure 2PC and only rely on a (modified) PSI protocol. Our key insight is that the protocol view of receiver R is secure if they cannot distinguish between the two cases when 1) their element y_j is not in the input set of sender S and 2) when y_j is in input set of S ($\exists x_i : x_i = y_j$), but the two corresponding labels are equal, i.e., $u_i = v_j$. The idea behind PDC_1 is based on the standard DH-based PSI protocol for computing the size of the intersection (PSI capacity, PSI-CA [7]) and computes a permuted and blinded set intersection of the data elements. We then augment this set intersection protocol with (Elgamal) encrypted labels, such that S can select R's label if data element x_i is *not* in the intersection, and the label from S if it is.

3.1 Tools: DH-based PRF and Elgamal Encryption

Let $\mathbb{G}$ be a DDH group of prime order $p, |p| = \lambda$, and let key $k \xleftarrow{\$} \mathbb{Z}_p$ be chosen randomly such that $k^{-1} \bmod (p-1)$ exists (k co-prime to $p-1$). For inputs $\chi \in \mathbb{G}$, the output of function

$$\mathsf{PRF}_k(\chi) = \chi^k$$

is indistinguishable from randomly chosen elements of $\mathbb{G}$. This is a simple *DH-based PRF*, essentially masking χ by k [1, 15, 26]. As a side note, one can convert such a DH-based PRF into a regular PRF by a standard application of the leftover hash lemma [13].

There exists an interesting commutativity feature for DH-based PRFs which we will exploit later. For keys $\alpha_1, \beta_1 \in \mathbb{Z}_p$, we have

$$(\mathsf{PRF}_{\alpha_1}(x))^{\beta_1} = (\mathsf{PRF}_{\beta_1}(x))^{\alpha_1} = \mathsf{PRF}_{\alpha_1\beta_1}(x).$$

The masking by raising to a key can also be undone: given $\mathsf{PRF}_{\alpha_1\beta_1}(x)$ and α_1,

$$\mathsf{UnMask}_{\alpha_1}(\mathsf{PRF}_{\alpha_1\beta_1}(x)) = Y^{\alpha_1^{-1}} = \mathsf{PRF}_{\beta_1}(x)$$

and similarly for β_1.

Let $\alpha_2, \beta_2 \xleftarrow{\$} \mathbb{Z}_p$ again be chosen randomly. They serve as secret keys for Elgamal encryptions Enc_{α_2} and Enc_{β_2}. The corresponding public keys g^{α_2} and g^{β_2} are distributed in advance to S and R. For key $k \in \{\alpha_2, \beta_2\}$ and any input $\chi \in \mathbb{G}$, we define Elgamal encryption as

$$\mathsf{Enc}_k(\chi) = < g^r, \chi g^{rk} >,$$

where $r \xleftarrow{\$} \mathbb{Z}_p$. Note that $\mathsf{Enc}_k(\chi)$ is a probability ensemble indexed by the length $p, |p| = \mathsf{poly}(\lambda)$. Consequently, there are many ciphertexts $\mathsf{Enc}_k(\chi)$ for a plaintext χ which we regard as an ensemble. We only need to store one representative of the ensemble, since all other elements in the ensemble can be generated from it.

For this Elgamal encryption, we can add another layer of encryption as follows. With $< A, B > \leftarrow \mathsf{Enc}_{\alpha_2}(\chi)$, we define a double-layer Elgamal ciphertext $\mathsf{Enc}_{\beta_2\alpha_2}(\chi)$ as

$$\begin{aligned}\mathsf{Enc}_{\beta_2\alpha_2}(\chi) &= \mathsf{AddEnc}_{\beta_2}(< A, B >)\\ &= < Ag^{r'}, BA^{\beta_2} \cdot (g^{\alpha_2})^{r'} \cdot (g^{\beta_2})^{r'} >,\end{aligned}$$

with $r' \in \mathbb{Z}_p$ chosen randomly. Note that knowledge of β_2 is required. Adding another layer of encryption is commutative. Let $< A, B > \leftarrow \mathsf{Enc}_{\alpha_2}(\chi)$ and $< A', B' > \leftarrow \mathsf{Enc}_{\beta_2}(\chi)$. We have

$$\mathsf{AddEnc}_{\beta_2}(< A, B >) \stackrel{c}{\equiv} \mathsf{AddEnc}_{\alpha_2}(< A', B' >).$$

To re-encrypt (re-randomize) a given ciphertext $< A, B > \leftarrow \mathsf{Enc}_{\alpha_2}(\chi)$ using public key g^{β_2}, we use

$$\mathsf{ReEnc}_{g^{\beta_2}}(< A, B >) = < Ag^r, B \cdot (g^{\beta_2})^r >,$$

with $r \in \mathbb{Z}_p$ chosen randomly.

Furthermore, we combine our DH-based PRF with Elgamal encryption. For key $k \in \{\alpha_2, \beta_2\}$ and a ciphertext $< A, B >$, we define our DH-based PRF variant for Elgamal ciphertexts as

$$\mathsf{PRF}_k(< A, B >) = < A^k, B^k > .$$

Again, Elgamal encryption and our DH-based PRF offer commutativity. Let $< A, B > \leftarrow \mathsf{Enc}_{\alpha_2}(\chi)$ and $\chi' = \mathsf{PRF}_{\beta_2}(\chi)$. Then,

$$\mathsf{PRF}_{\beta_2}(< A, B >) \stackrel{c}{\equiv} \mathsf{Enc}_{\alpha_2}(\chi').$$

For some double-layer ciphertext $< A, B > \leftarrow \mathsf{Enc}_{\alpha_2\beta_2}$, we can peel-of one layer of encryption from key $k \in \{\alpha_2, \beta_2\}$ by

$$\mathsf{UnPeel}_k(< A, B >) = < A, BA^{-k} > .$$

Finally, we also decrypt any Elgamal ciphertext $< A, B >$ using key $k \in \{\alpha_2, \beta_2\}$ as

$$\mathsf{Dec}_k(< A, B >) = BA^{-k}.$$

Observe that $\mathsf{PRF}_{\beta_2}(\chi) = \mathsf{Dec}_{\alpha_2}(\mathsf{PRF}_{\beta_2}(\mathsf{Enc}_{\alpha_2}(\chi)))$.

3.2 Main Protocol

Our protocol proceeds in the following five main steps.

In Step 1, both parties hash their data elements and labels into elements from an appropriate DDH group. This enables the parties to use the DH-based PRF and Elgamal constructions from the previous section.

In Step 2, sender S sends their input data elements, masked by their PRF, and an Elgamal encryption of the corresponding label to R.

In Step 3, R applies their PRF to the masked data elements and adds an encryption to the labels from S. S's data elements and labels are now protected by two keys, one from S and one from R. R shuffles the result and sends it back to S. Also, R sends PRF outputs of their data elements together with Elgamal encryptions of corresponding labels to S. These are only protected by one key from R.

In Step 4, S unmasks the doubly protected data elements and unpeels one layer of encryption from the labels of S. These are now protected by the same keys as the data elements and labels from R. Then, S matches the data elements in the two sets. S learns which PRF outputs are in both sets. However, S cannot determine which of their inputs correspond to which PRF output. For each PRF output that is in both sets, S sends to R: the PRF output, a masked version of the corresponding Elgamal encryption of the label as received from R, and a masked version of the Elgamal encryption of the corresponding label from S. For each PRF output in R's set but not in both sets, S sends to R: the PRF output, a masked version of the corresponding Elgamal encryption of the label as received from R, and a masked version of the re-encryption of the corresponding Elgamal encryption of the label a received from R. S can apply the masking to the labels, since encryption and PRF are commutative. Each PRF output has now two masked, encrypted labels where in case both parties have the data element their plaintexts are the labels of the respective party and in case only R has the data element, they are copies of each other.

In Step 5, R decrypts the Elgamal ciphertexts and verifies, for each PRF output, whether the two corresponding Elgamal ciphertexts match (no misclassification) or not (misclassification found).

We formalize our protocol in Figure 2.

3.3 Security Analysis

Semi-Honest Security Model. We operate in the semi-honest, i.e., passive, security model. Security against malicious adversaries is an open problem for circuit-PSI protocols. All current circuit-PSI protocols follow the same construction [4, 20, 33] of a reactive 2PC after computing secret shares of the intersection. Since this construction is composed of reactive functionalities, it is also only secure in the semi-honest model by default. Efficient constructions secure against malicious adversaries that do not resort to 2PC for the entire protocol are still an open problem. Hence, we believe it is justified to also consider the semi-honest model for our protocols.

Let $\mathcal{S}_I$ be the intersection of elements in $< x_i >_{i\in[n]}$ and $< y_i >_{i\in[n]}$. For protocol PDC_1, we assume the leakage of the size of this intersection $\mathcal{S}_I$ given to the sender S. We prove security against semi-honest adversaries by constructing two simulators $\mathsf{Sim}_R(\mathcal{S}_R, \mathcal{S}_E)$ and $\mathsf{Sim}_S(\mathcal{S}_S, |\mathcal{S}_I|)$ or $\mathsf{Sim}_S(\mathcal{S}_S)$, depending on the leakage, for the receiver and sender, respectively. The output of each simulator is computationally indistinguishable from the respective party's view VIEW_R or VIEW_S, i.e., its messages received, during the execution of the real protocol:

$$\begin{aligned} \mathsf{Sim}_R(\mathcal{S}_R, \mathcal{S}_E) &\stackrel{c}{\equiv} \mathsf{VIEW}_R^{\mathsf{PDC}_1} \\ \mathsf{Sim}_S(\mathcal{S}_S, |\mathcal{S}_I|) &\stackrel{c}{\equiv} \mathsf{VIEW}_S^{\mathsf{PDC}_1} \end{aligned}$$

Security Proof.

Theorem 1. *Protocol* PDC_1 *securely implements*[1] *functionality* $\mathcal{F}^{\mathsf{MCLASS}}$ *in the semi-honest security model.*

Proof. We prove the existence of the two simulators by construction.

$\mathsf{Sim}_R(\mathcal{S}_R, \mathcal{S}_E)$: R receives n PRF outputs and Elgamal ciphertexts under S's keys α_1 and α_2. By the definition of the primitives (pseudo-random functions and semantically secure encryption) these can be

[1] With leakage of the set intersection cardinality $|\mathcal{S}_I|$ to S

PARAMETERS: Security parameter λ, batch size b, pseudo-random function family $\mathsf{PRF} : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^\lambda$

1. Wait for input $x_1, \ldots, x_b : x_i \in \{0,1\}^\lambda$ from receiver R.
2. Choose $K \xleftarrow{\$} \{0,1\}^\lambda$ and random permutation $\pi : [b] \to [b]$. Output $< \mathsf{PRF}_K(x_{\pi(i)}) >_{i\in[b]}$ to R and (K, π) to S.

Figure 3: Ideal oblivious pseudo-random function (OPRF) functionality with set semantics $\mathcal{F}^{\mathsf{Set\text{-}OPRF}}$

PARAMETERS: Security parameter λ, batch size b, pseudo-random function family $\mathsf{PRF} : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^\lambda$

1. Wait for input $x_1, \ldots, x_b : x_i \in \{0,1\}^\lambda$ from receiver R.
2. Choose $K_1, \ldots, K_b, K_i \xleftarrow{\$} \{0,1\}^\lambda$. Output $< \mathsf{PRF}_{K_i}(x_i) >_{i\in[b]}$ to R, and output $< K_i >_{i\in[b]}$ to S.

Figure 4: Ideal oblivious pseudo-random function (OPRF) functionality with vector semantics $\mathcal{F}^{\mathsf{Vector\text{-}OPRF}}$

simulated using $3n$ uniformly random numbers from the DDH group $\mathbb{G}$. In the second round, R receives again n PRF outputs and $2n$ Elgamal ciphertexts under R's key's β_1 and β_2. The simulator computes the n PRF outputs from $\mathcal{S}_R$ and R's key β_1. The $2n$ Elgamal ciphertexts decrypt to (and their plaintext can be simulated by) PRF outputs of the element concatenated with the label under S' key α_1. For each element $y_i \in \mathcal{S}_E$, the simulator outputs the same uniformly chosen group element (in $\mathbb{G}$) twice. For each element $y_i \in \mathcal{S}_R \setminus \mathcal{S}_E$, the simulator outputs two independently, uniformly chosen group elements. The message received by R are Elgamal ciphertexts under R's key β_2 of these group elements.

$\mathsf{Sim}_S(\mathcal{S}_S, |\mathcal{S}_I|)$: S receives $2n$ pseudo-random functions and Elgamal ciphertexts under R's keys β_1 and β_2. The Elgamal ciphertexts can be simulated using $4n$ group elements from $\mathbb{G}$. However, the joint distribution of the PRF outputs reveals the set intersection cardinality, while each PRF output remains indistinguishable from a random group element. The simulator uniformly chooses $|\mathcal{S}_I|$ group elements from $\mathbb{G}$ and puts them into set $\mathcal{S}_1$. It then chooses $2(n - |\mathcal{S}_I|)$ group elements and puts them into set $\mathcal{S}_2$. The simulator divides set $\mathcal{S}_2$ into two equal-sized sets $\mathcal{S}_3$ and $\mathcal{S}_4$. Then it forms two new sets $\mathcal{S}_5 = \mathcal{S}_1 \cup \mathcal{S}_3$ and $\mathcal{S}_6 = \mathcal{S}_1 \cup \mathcal{S}_4$. It randomly shuffles $\mathcal{S}_5$ and $\mathcal{S}_6$. These sequences follows the same joint distribution as the sequences received during the real protocol execution. □

Security in IND-CDP-2PC. He et al. [14] define a security model IND-CDP-2PC for two-party computations in the computational differential privacy setting [27]. In this model the leakage of the set intersection cardinality to the sender S can be avoided. This security model allows the view of a party to be differentially private between two neighboring inputs of the other party instead of indistinguishable.

To achieve security in IND-CDP-2PC, we modify the protocol as follows. In addition to $< x_i, u_i >$ S inputs $d = 2\lceil -\log \delta/\epsilon \rceil$ elements $< \chi_1, \upsilon_1, \ldots, \chi_d, \upsilon_d >$. Correspondingly, R chooses a random number d' from a shifted and bounded Laplace distribution $\min(d, \max(0, Lap(-\log \delta/\epsilon, 1/\epsilon)))$. R's additional input is $< \chi_1, \upsilon_1, \ldots, \chi_{d'}, \upsilon_{d'} >$ and $< \psi_1, \upsilon_{d'+1}, \ldots, \psi_{d-d'}, \upsilon_d >$ where $\psi_i \neq \chi_j$. They run the protocol as in Figure 2. R's output is unmodified, since none of the additional inputs has a matching element with mismatching labels. Let $|\mathcal{S}_I|$ be the set intersection cardinality of $< x_i >_{i\in[n]}$ and $< y_i >_{i\in[n]}$. S learns, i.e., its view includes, $|\mathcal{S}_I| + d'$. This satisfies IND-CDP-2PC, since the set intersection cardinality of two neighbouring input databases $\mathcal{S}_R$ and $\mathcal{S}'_R$ (R's input) and the database $\mathcal{S}_S$ may differ by at most 1. The shifted and bounded Laplace mechanism is (ϵ, δ)-differentially private with respect to a change (sensitivity) of 1 (see [41]) and hence the view of S in the protocol is differentially private with respect to neighboring inputs $\mathcal{S}_R$ and $\mathcal{S}'_R$ by R.

4 $\mathsf{PDC_2}$

The high efficiency of OPRF-based PSI [4, 21, 35, 37] stems from the fact that the actual intersection is computed "in the clear". Receiver R receives (O)PRF outputs for their input, and sender S sends PRF values for their input to R. The actual intersection of elements is then computed by R "in the clear" on the two sets of PRF outputs. This avoids reverting to expensive cryptographic techniques such as 2PC or FHE during computation of the intersection and yields high efficiency. The only expensive operation is computing the OPRFs for the parties' inputs. We will stick to this "OPRF-and-comparing-in-the-clear" principle also for PDC_2.

4.1 Overview

Our main idea is based on the following observation. If two bit strings u from S and v from R, $|u| = |v| = \ell$, differ in the bit at position i, then $u[i] \oplus 1 = v[i]$. Using an OPRF, R could receive set

$$\mathcal{R} = \{\mathsf{PRF}_K(v[1]||1), \ldots, \mathsf{PRF}_K(v[\ell]||\ell)\}$$

as output, and S could send set

$$\mathcal{S} = \{\mathsf{PRF}_K(u[1] \oplus 1||1), \ldots, \mathsf{PRF}_K(u[\ell] \oplus 1||\ell)\}$$

in shuffled order to R. Receiver R would now check in the clear whether there is an element in $\mathcal{R}$ that is also in $\mathcal{S}$. If there is such a match, then $u \neq v$. However, with this approach, R would learn additional information, namely how many bits are different between u and v.

To remedy, consider the following two sets

$$\mathcal{R} = \{(\mathsf{Prefix}_1(v)), \ldots, (\mathsf{Prefix}_\ell(v))\} \text{ and}$$
$$\mathcal{S} = \{(\mathsf{Prefix}_1(u) \oplus 1), \ldots, (\mathsf{Prefix}_\ell(u) \oplus 1)\}.$$

These are the sets of all prefixes of u and v, where each time the last bit of the prefixes of u is flipped. It is crucial to observe that, for these two sets, we have set intersection cardinality $|\mathcal{S} \cap \mathcal{R}| = 1$ if and only if $u \neq v$, otherwise $|\mathcal{S} \cap \mathcal{R}| = 0$. That is, either there exists a single match between an element in $\mathcal{R}$ which is also in $\mathcal{S}$ ($u \neq v$), or there is no such match ($u = v$).

Consequently, for a single pair of length ℓ bit strings u from S and v from R, parties run ℓ instances of the OPRF. Here, R's inputs are $\mathsf{Prefix}_1(v)$ to $\mathsf{Prefix}_\ell(v)$ such that R receives set

$$\mathcal{R} = \{\mathsf{PRF}_K(\mathsf{Prefix}_1(v)), \ldots, \mathsf{PRF}_K(\mathsf{Prefix}_\ell(v))\}$$

as output. After the OPRF evaluations, S sends

$$\mathcal{S} = \{\mathsf{PRF}_K(\mathsf{Prefix}_1(u) \oplus 1), \ldots, \mathsf{PRF}_K(\mathsf{Prefix}_\ell(u) \oplus 1)\}$$

to R. Then, R computes $|\mathcal{S} \cap \mathcal{R}|$ in the clear to determine whether $u = v$ and without learning anything else about u. Essentially, S and R compute a private set intersection cardinality (PSI-CA).

Integrating data elements. Our approach above essentially converts the problem of finding a *mismatch* between labels u and v into that of finding a *match*. We expand this technique to also determine equality of the corresponding x and y data elements at the same time. In our situation, a match is a pair of tuples (x, u) and (y, v) with $x = y$ and simultaneously $u \neq v$, so we change the above computation of OPRFs as follows. For R's input (y, v), S and R run ℓ instances of an OPRF where R's input is $\{(y||\mathsf{Prefix}_1(v)), \ldots, (y||\mathsf{Prefix}_\ell(v))\}$, and R receives

$$\mathcal{R} = \{\mathsf{PRF}_K(y||\mathsf{Prefix}_1(v)), \ldots, \mathsf{PRF}_K(y||\mathsf{Prefix}_\ell(v))\}$$

as output. Sender S then sends

$$\mathcal{S} = \{\mathsf{PRF}_K(x||\mathsf{Prefix}_1(u) \oplus 1), \ldots, \mathsf{PRF}_K(x||\mathsf{Prefix}_\ell(u) \oplus 1)\}.$$

Finally, R again computes $|\mathcal{S} \cap \mathcal{R}|$ in the clear, which is either 0 or 1. Observe that now $|\mathcal{S} \cap \mathcal{R}| = 1$ *iff* $x = y \wedge u \neq v$.

To support n inputs $\{(x_1, u_1), \ldots, (x_n, u_n)\}$ from S and $\{(y_1, v_1), \ldots, (y_n, v_n)\}$ from R, parties run $n\ell$ instances of the OPRF, ℓ for each of R's inputs (y_i, v_i). Similarly, S sends ℓ PRF outputs for each input (x_i, u_i).

Input of S: n pairs $< (x_1, u_1), \dots, (x_n, u_n) >, x_i \in \{0,1\}^*, u_i \in \{0,1\}^\ell$
Input of R: n pairs $< (y_1, v_1), \dots, (y_n, v_n) >, y_i \in \{0,1\}^*, v_i \in \{0,1\}^\ell$
Parameters: Security parameter λ, group $\mathbb{G}$ of prime order p where the DDH assumption is believed to be hard, $|p| = \lambda$, g is a generator of $\mathbb{G}$, hash function $H : \{0,1\}^* \to \mathbb{G}$
Protocol:

1. S selects $K \xleftarrow{\$} \mathbb{Z}_p$. R selects $r \xleftarrow{\$} \mathbb{Z}_p$.
2. For each $(y_i, v_i)_{i\in[n]}$, R computes $< h_{i,j} = H(y_i||\mathsf{Prefix}_j(v_i))^r >_{j\in[\ell]}$. R sends all ℓn values $h_{i,j}$ to S.
3. For $i \in [n]$:
   (a) S selects random permutation $\pi_i : [\ell] \to [\ell]$.
   (b) For $j \in [\ell]$:
      i. S sends $h'_{i,j} = (h_{i,\pi_i(j)})^K$ to R, and R computes $\gamma_{i,j} = (h'_{i,j})^{r^{-1}}$.
      ii. S computes $\gamma'_{i,j} = (H(x_i||\mathsf{Prefix}_j(u_i) \oplus 1)^K$.
4. S sends all ℓn values $\gamma'_{i,j}$ in randomly shuffled order to R.
5. R outputs $\{i|\exists(a,b,c) : \gamma_{i,a} = \gamma'_{b,c}\}$.

Figure 5: DH-OPRF-based misclassification protocol DH-PDC_2

Reducing complexity. Conceptually, a PRF and an OPRF for this PRF support arbitrary long inputs using the standard trick of applying a cryptographic hash function to the input before using it as input to the PRF or OPRF. Hashing reduces arbitrary length inputs down to λ bits. As a result, for standard OPRF-based PSI protocols, the length of inputs does not matter for the complexity of the protocol. However for our OPRF-based PSI misclassification protocols, the situation is different. For an input (x, u) (and also (y, v)), the lengths of data elements x and y do not matter, but label length $\ell = |u| = |v|$ does. Now, communication and communication complexity increase by a factor of ℓ. While this is acceptable for smaller values of ℓ, performance will suffer for larger values of $\ell \leq \lambda$. To reduce both asymptotic and concrete complexity, we use a probabilistic variation of the above idea which does not require $\ell = \lambda$ OPRFs for arbitrary long labels, but only $\min(\ell, \log n + \sigma)$, where σ is a statistical security parameter. As the technical features of this variation are not crucial for understanding our OPRF-based constructions in detail, assume for now that for any label length of u or v, we will perform ℓ OPRFs.

Implementing PSI-CA with OPRFs. In our PDC_2 protocols, we will compute and output set intersection cardinality over the set of prefixes for one pair of elements. We hence implement PSI-CA protocols using OPRFs but distinguish two types of OPRFs: Set-OPRF and Vector-OPRF. Both OPRF operate over batches of elements. Set-OPRFs return a permutation of the PRF values in a batch whereas Vector-OPRFs only allow to return the PRFs in the same order as the input elements. Furthermore, Vector-OPRF may choose a different key for each input. The ideal functionalities of both OPRFs are shown in figures 3 and 4. The standard DH-based OPRF is a Set-OPRF, while the more recent and efficient KKRT-OPRF [21] and VOLE-OPRF [35, 37] are Vector-OPRFs. Note that VOLE-OPRF is a special case of a Vector-OPRF: while it does not support random shuffling of outputs, it uses the same key $K_i = K_j$ for all elements. Thus, it is also covered by our definition of $\mathcal{F}^{\mathsf{Vector\text{-}OPRF}}$, and we will use it as a building block in the construction of Vector-PDC_2 below.

We describe the use of the DH-based OPRF as a Set-OPRF for our protocol DH-PDC_2 in Section 4.2 and the use of a Vector-OPRF for our protocol Vector-PDC_2 in Section 4.4.

4.2 DH-PDC_2 Details

Recall from Section 3.1 that the DH-based OPRF is conceptually very simple. For PRF key K from sender S and blinding key r and input y from receiver R, R starts by sending a blinded $y' = y^r$ to S. Sender S replies with $y'' = (y')^K$, and R unblinds with $(y'')^{r^{-1}}$. Informally, this realizes $\mathsf{PRF}_K(y) = y^K$. Applying this protocol to a batch of elements with S shuffling values y'' before returning them to R realizes $\mathcal{F}^{\mathsf{Set\text{-}OPRF}}$.

Figure 5 presents technical details of the DH-based OPRF applied to our PDC_2 idea for PSI misclassification. We dub this protocol DH-PDC_2. The use of the DH-based OPRF in our idea is mostly straightforward, but there are two important peculiarities. First, for each $i \in [n]$, S collects all ℓ blinded inputs $y_i||\mathsf{Prefix}_j(v_i)$ and shuffles them using a random permutation π_i before sending PRF outputs back, see Step (3(b)i) in Figure 5.

It is crucial that S shuffles the ℓ inputs of R for each i. Otherwise, if R later finds a match for one of the ℓ inputs, they would learn the bit position where u and v differ which is more leakage than in the ideal functionality. For the same reason, S also has to shuffle all of their PRF outputs $\gamma'_{i,j}$ before sending them to R, see Step 4. Note that S can shuffle all $\gamma'_{i,j}$ at once while the $h_{i,j}$ have to be shuffled per i such that R can later still determine the index of the element with a misclassification.

There is an optimization possible which we have omitted from Figure 5. As elements from $\mathbb{G}$ are typically larger than λ, we can employ another hash functions $H' : \mathbb{G} \to \{0,1\}^\lambda$ and hash the $\gamma'_{i,j}$ to smaller values. Our implementation in Section 5 uses this technique for increased efficiency.

4.3 Security Analysis of DH-PDC_2

Theorem 2. *Protocol DH-PDC_2 securely implements[2] functionality $\mathcal{F}^{\mathsf{MCLASS}}$ in the semi-honest security model.*

Proof. Since protocol DH-PDC_2 does not leak we prove the existence of the following two simulators:

$$\begin{aligned} \mathsf{Sim}_R(\mathcal{S}_R, \mathcal{S}_E) &\overset{c}{\equiv} \mathsf{VIEW}_R^{\mathrm{DH-PDC_2}} \\ \mathsf{Sim}_S(\mathcal{S}_S) &\overset{c}{\equiv} \mathsf{VIEW}_S^{\mathrm{DHPDC_2}} \end{aligned}$$

$\mathsf{Sim}_R(\mathcal{S}_R, \mathcal{S}_E)$: R receives $2n\ell$ PRF outputs under S's key K. The simulator creates an empty sequence of size $2n\ell$. For each $y_i \in \mathcal{S}_E$, the simulator uniformly chooses a group element in $\mathbb{G}$ and places it at two random positions in the sequence: between $2(i-1)\ell$ and $(2i-1)\ell - 1$ and between $(2i-1)\ell$ and $2i\ell - 1$. All other elements in the sequence are filled with uniformly chosen group elements. This sequence is computationally indistinguishable from R's view in DH-PDC_2.

$\mathsf{Sim}_S(\mathcal{S}_S)$: S receives $n\ell$ PRF outputs under R's random number r. All of these are independently simulatable by uniformly chosen group elements. □

4.4 Vector-PDC_2 Details

While DH-PDC_2 features low communication complexity, its main drawback is the need for $O(n\ell)$ public-key operations during OPRF evaluation. Thus in Vector-PDC_2, we replace the DH-based OPRF by more recent OPRF constructions from the literature which use only $O(\lambda)$ or even no public key operations. The $O(n\ell)$ evaluations of the main OPRF functionality are then performed using only fast symmetric cryptography. In our implementation, we use the KKRT-OPRF from Kolesnikov et al. [21] and the very recent VOLE-OPRF from Rindal and Schoppmann [37] with improvements by Raghuraman and Rindal [35]. Both OPRFs implement the Vector-OPRF functionality $\mathcal{F}^{\mathsf{Vector\text{-}OPRF}}$ from Figure 4.

4.4.1 Technical Challenges

Ideally, we would like to simply exchange the DH-based OPRF building block in DH-OPRF-based protocol DH-PDC_2 by a Vector-OPRF to benefit from its significantly higher efficiency. However, this is obviously not feasible, since a Vector-OPRF does not allow to shuffle the outputs. We hence need a different construction. Furthermore, in contrast to the DH-based Set-OPRF before, a Vector-OPRF's key may depend on the order of inputs. Even if both parties hold the same inputs $x_i = y_j$, PRF outputs of x_i and y_i will match only if they have been computed using the same key. More specifically, if R uses y_j as their j^{th} input, they will obtain $\mathsf{PRF}_{K_j}(y_j)$. Now, S has to somehow send $\mathsf{PRF}_{K_j}(x_i)$, even if x_i is not the j^{th} input of S.

For standard PSI, a well-known trick to enforce that parties use the same order for their inputs is for R to hash their inputs into a (stash-less) Cuckoo hashtable [4, 11, 21, 22, 30, 32–34]. R iterates over all buckets in their Cuckoo table and, for the b^{th} bucket containing element y_j, runs an OPRF with S to receive $\mathsf{PRF}_{K_b}(y_j)$, i.e., the PRF output using the b^{th} key. When using Cuckoo hashing with η hash functions, S

[2] *Without* leakage of the set intersection cardinality $|\mathcal{S}_I|$.

Input of S: n pairs $<(x_1,u_1),\dots,(x_n,u_n)>, x_i \in \{0,1\}^*, u_i \in \{0,1\}^\ell$
Input of R: n pairs $<(y_1,v_1),\dots,(y_n,v_n)>, y_i \in \{0,1\}^*, v_i \in \{0,1\}^\ell$, key $K_{R,*}$
Parameters: Security parameter λ, Cuckoo table size $m = 1.27n$, hash functions $H : \{0,1\}^* \to \{0,1\}^\lambda$, $H_1, H_2, H_3 : \{0,1\}^* \to [m]$, Vector-OPRF functionality $\mathcal{F}^{\text{Vector-OPRF}}$
Protocol:

1. S creates an m-bucket Cuckoo hash table $\mathcal{T}_S$ using the x_i as keys and hash functions H_1, H_2, and H_3. R creates an m-bucket Cuckoo hash table $\mathcal{T}_R$ using the y_i as keys and hash functions H_1, H_2, and H_3.
2. S and R iterate over the m buckets of their Cuckoo hash tables.
   For $j \in [m]$,
   (a) Let x_i be mapped into bucket $\mathcal{T}_S[j]$. S and R run functionality $\mathcal{F}^{\text{Vector-OPRF}}$ with S being the receiver and R the sender. The inputs of S are $h_{j,k} = H(x_i||\mathsf{Prefix}_k(u_i))$. The outputs of R are keys $K_{R,j,k}$, and the outputs of S are $z_{j,k} = \mathsf{PRF}_{K_{R,j,k}}(h_{j,k})$.
   If no element is mapped into $\mathcal{T}_S[j]$, S inputs ℓ random bit strings as input.
   (b) Let y_i be mapped into $\mathcal{T}_R[j]$. S and R run functionality $\mathcal{F}^{\text{Vector-OPRF}}$ with R being the receiver and S the sender. The inputs of R are $h'_{j,k} = H(y_i||\mathsf{Prefix}_k(v_i) \oplus 1)$. The outputs of S are keys $K_{S,j,k}$, and the outputs of R are $z'_{j,k} = \mathsf{PRF}_{K_{S,j,k}}(h'_{j,k})$.
   If no element is mapped into $\mathcal{T}_R[j]$, R inputs ℓ random bit strings as input.
3. R creates an m-bucket table $\mathcal{T}^*$. Each bucket $\mathcal{T}^*[j]$ comprises ℓ slots $< T^*[j][k] >_{k\in[\ell]}$, $|T^*[j][k]| = 3\lambda$ Bit. R fills $\mathcal{T}^*$ as follows.
   For $j \in [m]$,
   - If $\mathcal{T}_R[j]$ is empty, then R sets $\mathcal{T}^*[j][k] \xleftarrow{\$} \{0,1\}^{3\lambda}$ for all $k \in [\ell]$.
   - If $\mathcal{T}_R[j]$ is not empty, then let $\mathcal{T}_R[j] = y_i$. Let $H_1(y_i), H_2(y_i), H_3(y_i)$ be the three possible positions where y_i can be mapped to in $\mathcal{T}_S$.
     For $k \in [\ell]$, R computes
     $$c_{j,k,1} = z'_{j,k} \oplus \mathsf{PRF}_{K_{R,H_1(y_i),k}}(h'_{j,k}) \oplus \mathsf{PRF}_{K_{R,*}}(y_i)$$
     $$c_{j,k,2} = z'_{j,k} \oplus \mathsf{PRF}_{K_{R,H_2(y_i),k}}(h'_{j,k}) \oplus \mathsf{PRF}_{K_{R,*}}(y_i)$$
     $$c_{j,k,3} = z'_{j,k} \oplus \mathsf{PRF}_{K_{R,H_3(y_i),k}}(h'_{j,k}) \oplus \mathsf{PRF}_{K_{R,*}}(y_i).$$
     R sets $\mathcal{T}^*[j][k] = c_{j,k,1}||c_{j,k,2}||c_{j,k,3}$.
4. R sends $\mathcal{T}^*$ to S.
5. S creates an empty set $\mathcal{S}$ and fills it as follows.
   For $i \in [n]$,
   - Let $H_1(x_i), H_2(x_i), H_3(x_i)$ be the three possible positions where x_i can be mapped to in $\mathcal{T}_R$. Let $\mathsf{IDX}(x_i) \in \{1,2,3\}$ be the index of the hash function that was used to eventually map x_i into $\mathcal{T}_S$ during Step (1). Let $\mathsf{cut} : \{0,1\}^\lambda \times \{1,2,3\} \to \lambda$ be a function with two inputs. The first input to cut is a bit string L of length 3λ, and the second input is either 1, 2 or 3. Function cut outputs either the first, second or third λ bits of L.
   - For $k \in [\ell]$, S computes
     $$d_{i,k,1} = z_{H_{\mathsf{IDX}(x_i)},k} \oplus \mathsf{PRF}_{K_{S,H_1(x_i),k}}(h_{H_{\mathsf{IDX}(x_i)},k}) \oplus \mathsf{cut}(\mathcal{T}^*[H_1(x_i)], \mathsf{IDX}(x_i))$$
     $$d_{i,k,2} = z_{H_{\mathsf{IDX}(x_i)},k} \oplus \mathsf{PRF}_{K_{S,H_2(x_i),k}}(h_{H_{\mathsf{IDX}(x_i)},k}) \oplus \mathsf{cut}(\mathcal{T}^*[H_2(x_i)], \mathsf{IDX}(x_i))$$
     $$d_{i,k,3} = z_{H_{\mathsf{IDX}(x_i)},k} \oplus \mathsf{PRF}_{K_{S,H_3(x_i),k}}(h_{H_{\mathsf{IDX}(x_i)},k}) \oplus \mathsf{cut}(\mathcal{T}^*[H_3(x_i)], \mathsf{IDX}(x_i))$$
     and appends $d_{i,k,1}, d_{i,k,2}, d_{i,k,3}$ to $\mathcal{S}$.
6. S shuffles $\mathcal{S}$ and sends it to R.
7. For $i \in [n]$, if $\mathsf{PRF}_{K_{R,*}}(y_i) \in \mathcal{S}$, then R outputs y_i.

Figure 6: Vector-OPRF-based misclassification protocol Vector-PDC_2

knows all η possible buckets $b_1, \ldots, b_\eta$ where each of their inputs x_i could have been be placed by R. So, S knows all η possible keys and sends $\mathsf{PRF}_{K_{b_1}}(x_i), \ldots, \mathsf{PRF}_{K_{b_\eta}}(x_i)$ to R which can then compare PRF outputs and compute the intersection.

While this is a valid technique to employ a Vector-OPRF in a standard PSI protocol, it does not suffice for our misclassification scenario. R still knows the PRF for each of its elements and can hence determine the matching prefix index of a misclassified element. As already mentioned, a Vector-OPRF does not allow to shuffle the outputs of the PRFs before returning them and hence there is no trivial fix to this problem. We describe our approach of dual use of Vector-OPRFs in the following section.

4.4.2 Main Protocol

Figure 6 formalizes protocol Vector-PDC_2, an application of a Vector-OPRF to the PDC_2 idea. This protocol comprises the following main steps which we explain in detail.

Step 1. First, both S and R hash their input data elements x_i and y_j into stash-less, m-bucket, 3-hash-function Cuckoo tables $\mathcal{T}_S$ and $\mathcal{T}_R$.

Step 2. For each bucket in $\mathcal{T}_S$, S and R run ℓ OPRFs with S being the receiver. The inputs are the $x_i||\mathsf{Prefix}_k(u_i)$ for the data element x_i mapped to that bucket. As a result, S obtains PRF outputs $z_{j,k}$. Then, parties run another ℓ OPRFs for each bucket of $\mathcal{T}_R$ such that R obtains PRF outputs $z'_{j,k}$ for their inputs $y_j||\mathsf{Prefix}_k(v_j)$. Additionally, R obtains PRF keys $K_{R,b,k}$ used for all buckets b in $\mathcal{T}_S$, and S obtains PRF keys $K_{S,b,k}$ used for all buckets b in $\mathcal{T}_R$.

Steps 3 to 6. Let there be two elements $x_i = y_j$ from S and R, and let R have mapped y_j into bucket b_1 in $\mathcal{T}_R$. Due to Cuckoo hashing, R does not know where S has mapped x_i in $\mathcal{T}_S$, but they now the three potential buckets b_1, b_2, b_3 where S could have mapped x_i to.

In Step 3, R creates another m-bucket table $\mathcal{T}^*$. As R has mapped y_j into b_1 of $\mathcal{T}_R$, they put into bucket b_1 of $\mathcal{T}^*$ the corresponding $z'_{b_1,k}$ PRF output they have received for bucket b_1 during Step 2.

Assume R would send such a table $\mathcal{T}^*$ to S. As S also knows possible bucket indexes b_1, b_2, b_3 where R could have mapped x_i in $\mathcal{T}_R$, S would check whether one of the buckets b_1, b_2, b_3 contains $\mathsf{PRF}_{K_{S,b,k}}(x_i||\mathsf{Prefix}_k(u_i) \oplus 1)$. This is bad, because S would learn the index of the bucket which leaks information about R's Cuckoo table. Moreover, R would also learn which of the prefixes is different between u_i and v_j.

To avoid that, R does not put $z'_{b_1,k}$ into bucket b_1 of $\mathcal{T}^*$, but three copies of $z'_{b_1,k}$, each one 1) additionally masked with the PRF output of $y_i||\mathsf{Prefix}_k(v_i) \oplus 1$ using the three possible keys $K_{R,b_1,k}, K_{R,b_2,k}, K_{R,b_3,k}$, and 2) masked by a random ID representing y_i. Now, S can "peel off" the first mask from *one* of these copies only if it has received the correct $z_{b_1,k}$ for input $x_i||\mathsf{Prefix}_k(u_i)$ in bucket b_1 in Step 2.

Thus, for each bucket b_1, b_2, b_3 from $\mathcal{T}^*$, S has computed a candidate bit string $d_{i,j}$. At most one of these bit strings will be equal to R's random ID. Finally, S sends back all candidate bit strings in shuffled order. Again, shuffling is crucial to avoid that R learns at which index the prefixes of u_i and v_j differ.

Step 7. In the shuffled set of candidate strings, R searches for their random IDs. For each random ID found, R knows that the corresponding tuple (y_j, v_j) represents a misclassification.

4.4.3 Discussion

To ease our exposition above, we have omitted several important technicalities.

When parties are performing Cuckoo hashing using the three hash functions H_1, H_2, H_3, we must make sure that no hash collisions for parties' data elements occur. Otherwise, this leads to a security issue in Step 3. There, two (or even all three) values $c_{j,k,1}$ would be the same, telling S something about the input of R. There exist several techniques to avoid collisions among three hash functions for Cuckoo hashing, but we choose the following straightforward one.

Let all hash functions H_i come from a set of possible hash functions $\mathcal{H}$ that all share the property of mapping into bit strings of length λ. Let S and R share a seed s for a PRG. S and R pseudo-randomly choose $H_1, H_2, H_3 \stackrel{\$}{\leftarrow} \mathcal{H}$ using the PRG. Whenever party S (or R) observes a collision for one of their input data elements x_i (or y_j), S (or R) hashes this data element x_i (or y_j) with the next three hash functions $H_4, H_5, H_6 \stackrel{\$}{\leftarrow} \mathcal{H}$ into their Cuckoo table. Again H_4, H_5, H_6 are chosen pseudo-randomly using the PRG. If there is still a collision for x_i (or y_j), H_7, H_8, H_9 are chosen and so on. The next data element x_{i+1} (or

y_{j+1}) will be hashed into the Cuckoo table starting again with hash functions H_1, H_2, H_3. Using this trick, we make sure that the same inputs $x_i = y_j$ will always be hashed with the same three hash functions. Note that the Cuckoo hash table maintains its capacity properties, since the set of hash bins for each element remains independently, identically distributed.

The formal description of protocol Vector-PDC_2 in Figure 6 is in the $\mathcal{F}^{\mathsf{Vector\text{-}OPRF}}$-hybrid model. Both parties can make ideal calls to a trusted party implementing the Vector-OPRF functionality $\mathcal{F}^{\mathsf{Vector\text{-}OPRF}}$ as defined in Figure 4.

We choose our parameters for stash-less Cuckoo hashing ($m = 1.27n$ buckets, 3 hash functions) following Pinkas et al. [32] who also provide a failure probability analysis.

4.5 Security Analysis

Theorem 3. *Protocol Vector-*PDC_2 *securely implements functionality* $\mathcal{F}^{\mathsf{MCLASS}}$ *in the semi-honest security model.*

Proof. $\mathsf{Sim}_R(\mathcal{S}_R, \mathcal{S}_E)$: In the first round, R receives m PRF outputs under S's keys. These can be simulated by uniformly chosen numbers in $\{0,1\}^\lambda$ according to functionality $\mathcal{F}^{\mathsf{Vector\text{-}OPRF}}$ (see also Kolesnikov et al. [21] for details).

In the second round, R receives $3n$ numbers. For each element $y_i \in \mathcal{S}_E$, the simulator outputs the PRF of y_i under R's key $K_{R,*}$. For the other $3n - |\mathcal{S}_E|$ elements, the simulator outputs uniformly chosen numbers in $\{0,1\}^\lambda$. The simulator outputs these $3n$ numbers in random order. The uniformly chosen numbers in $\{0,1\}^\lambda$ are indistinguishable from the view, because either the corresponding prefix is not in S's set and hence the PRF output under S's key is indistinguishable to R or the prefix is in the set, but R only has access to one PRF output for the corresponding bucket in $\mathcal{T}_R$.

$\mathsf{Sim}_S(\mathcal{S}_S)$: In the first round, S also receives m PRF outputs but under R's keys. These can be simulated by uniformly chosen numbers in $\{0,1\}^\lambda$.

In the second round, S receives $3m$ numbers. All of these can be simulated by independently, uniformly chosen numbers in $\{0,1\}^\lambda$. The uniformly chosen numbers are indistinguishable from the view, because either the corresponding prefix is not in S's set and hence the PRF output under R's key is indistinguishable to R or the prefix is in the set, but S does not have access to the PRF output for the corresponding bucket in $\mathcal{T}_S$ or it is the PRF output of y_i under R's key $K_{r,*}$. □

5 Evaluation

We have implemented our protocol variations PDC_1, DH-PDC_2, KKRT-PDC_2, and VOLE-PDC_2 and evaluated them in different settings, varying input size n, label length ℓ, and network bandwidth. We run Vector-PDC_2 benchmarks with two different Vector-OPRFs, as the KKRT-OPRF promises better computational efficiency and the VOLE-OPRF lower communication efficiency.

The goal of our evaluation is to demonstrate the real-world practicality of our constructions and to point out their individual advantages depending on the setting. We have also compared our construction to an implementation of the strawman circuit-PSI approach from Section 1.3. This strawman protocol serves as a baseline, and we will see that our constructions deliver better overall performance in many realistic settings.

Our implementation is done in C++, and we will publish the source code upon publication of the paper. Both DH-based PRF and Elgamal encryption in PDC_1 as well as the DH-based OPRF in DH-PDC_2 use point operations on curve ristretto255 [8, 38], implemented in libSodium [25]. Our code for protocol KKRT-PDC_2 integrates the KKRT-OPRF implementation by Rindal [36]. VOLE-PDC_2 uses the code by Visa-Research [42] for the underlying VOLE-OPRF.

To benchmark the circuit-PSI strawman approach, we use the VOLE-based implementation of circuit-PSI by Visa-Research [42]. This implements the circuit-PSI by Rindal and Schoppmann [37] (see Figure 10 in [37]) and improvements from follow-up work [35]. To compute the actual misclassification, we change Step 5 from Fig. 10 in [37]. Instead of using 2PC to compute shares of u and v and a bit indicating whether $x = y$, we implement a more complex circuit that computes whether $x = y \wedge u \neq v$. This is more efficient than first computing shares in 2PC and then running an extra 2PC to perform misclassification test on the shares.

Table 2: Total runtime. Times in s, communication in MByte, $*$: arbitrary length labels ($\ell = \log n + \sigma$). For each setting, blue marks lowest total runtime, purple marks lowest communication, gray marks lowest total runtime and communication if weaker security guarantees of PDC_1 are acceptable.

			Circuit-PSI		PDC_1		DH-PDC_2		KKRT-PDC_2		VOLE-PDC_2	
n	ℓ	Bandwidth	Time	Comm	Time	Comm	Time	Comm	Time	Comm	Time	Comm
2^{16}	1	1 GBit/s	1.7		6.8		1.7		0.2		0.2	
		100 MBit/s	12.9	146	8.6	28	2.2	5	1.2	13	0.8	7
		10 MBit/s	126.8		30.1		5.2		11.4		6.3	
	4	1 GBit/s	1.8		6.8		6.9		0.7		0.5	
		100 MBit/s	13.7	154	8.6	28	7.3	18	4.8	53	2.6	26
		10 MBit/s	133.4		30.1		20.2		45.5		23.2	
	10	1 GBit/s	1.9		6.8		18.0		1.6		1.2	
		100 MBit/s	15.0	169	8.6	28	19.1	45	11.8	131	6.1	65
		10 MBit/s	146.5		30.1		50.5		113.8		56.7	
	20	1 GBit/s	2.2		6.8		35.9		3.2		2.1	
		100 MBit/s	17.2	195	8.6	28	39.5	90	23.7	263	12.1	129
		10 MBit/s	168.4		30.1		102.4		227.7		112.6	
	*	1 GBit/s	2.9		6.8		99.7		8.8		5.8	
		100 MBit/s	25.2	286	8.6	28	113.5	252	66.1	736	33.8	361
		10 MBit/s	247.4		30.1		299.3		637.6		314.8	
2^{18}	1	1 GBit/s	6.0		29.3		7.2		0.7		0.6	
		100 MBit/s	53.4	605	35.9	112	7.8	18	5.0	54	2.7	28
		10 MBit/s	523.0		121.0		20.2		47.0		24.6	
	4	1 GBit/s	6.5		29.3		28.8		2.8		1.9	
		100 MBit/s	56.0	636	35.9	112	32.2	72	19.7	217	10.3	111
		10 MBit/s	549.3		121.0		81.2		188.2		96.3	
	10	1 GBit/s	7.4		29.3		71.3		6.6		4.4	
		100 MBit/s	61.4	697	35.9	112	81.5	180	48.8	543	25.7	275
		10 MBit/s	601.9		121.0		210.9		470.3		239.8	
	20	1 GBit/s	8.6		29.3		142.8		13.1		8.9	
		100 MBit/s	70.1	798	35.9	112	165.8	320	97.5	1085	51.4	550
		10 MBit/s	689.7		121.0		432.6		940.5		479.2	
	*	1 GBit/s	12.6		29.3		416.6		37.3		30.7	
		100 MBit/s	103.7	1184	35.9	112	491.9	1044	282.3	3147	151.3	1594
		10 MBit/s	1022.9		121.0		1290.1		2727.0		1390.2	
2^{20}	1	1 GBit/s	28.0		119.3		28.8		3.1		2.4	
		100 MBit/s	220.5	2500	149.0	448	32.3	72	20.0	217	11.2	111
		10 MBit/s	2161.4		489.7		81.1		188.6		97.2	
	4	1 GBit/s	29.0		119.3		114.7		11.5		8.0	
		100 MBit/s	231.1	2622	149.0	448	132.4	288	79.0	868	42.0	440
		10 MBit/s	2266.6		489.7		343.1		753.3		384.2	
	10	1 GBit/s	31.5		119.3		286.4		26.8		20.5	
		100 MBit/s	252.4	2866	149.0	448	336.5	720	195.8	2170	104.6	1100
		10 MBit/s	2477.1		489.7		884.6		1881.8		959.4	
	20	1 GBit/s	35.6		119.3		573.5		52.3		49.6	
		100 MBit/s	287.6	3273	149.0	448	678.9	1440	390.4	4341	211.6	2199
		10 MBit/s	2827.9		489.7		1786.8		3762.5		1920.5	
	*	1 GBit/s	51.8		119.3		1733.4				282.0	
		100 MBit/s	429.2	4898	149.0	448	2065.2	4320	Out of RAM		703.1	6599
		10 MBit/s	4231.1		489.7		5412.2				5832.5	

We evaluate this circuit with 2PC using the EMP-Toolkit [43]. The input bit length of the data elements is set to the optimal $\log n + \sigma$ for 2PC, and we only vary label lengths ℓ.

We have conducted our evaluation on a machine with 3.2 GHz Intel Xeon(R) W-1290 CPU and 64 GByte RAM. To emulate different network scenarios and precisely control network bandwidth, we use `WonderShaper` [16]. The evaluation results are shown in Table 2.

Table 3: Cloud computing costs for 100 runs in US$ on Amazon t3.medium instances in US East data center, assuming communication at 100 MBit/s over the Internet. blue marks lowest total cost, gray marks lowest total cost if weaker security guarantees of PDC_1 are acceptable.

n	ℓ	Circuit-PSI	PDC_1	DH-PDC_2	VOLE-PDC_2
2^{16}	1	1.31	0.26	0.05	0.06
	4	1.35		0.18	0.23
	10	1.52		0.44	0.59
	20	1.75		0.88	1.16
	*	2.57		2.22	3.25
2^{18}	1	5.44	1.07	0.18	0.25
	4	5.72		0.71	1.00
	10	6.27		1.77	2.48
	20	7.17		3.20	4.95
	*	10.64		10.31	14.36
2^{20}	1	22.48	4.28	0.71	1.00
	4	23.57		2.84	3.96
	10	25.77		7.10	9.91
	20	29.43		14.22	19.81
	*	44.03		42.74	59.61

We vary input sizes n from 2^{16} to 2^{20}. For label length ℓ, we focus on practical values 1) $\ell = 1$, e.g., for binary classification, linear regression, medical diagnosis (cancer, no cancer), 2) $\ell = 4$, e.g., for MNIST CIFAR-10 image classification [24], 3) $\ell = 10$, e.g., for ImageNet classification [9] or types of malware, 4) $\ell = 20$, e.g., for supporting a huge number of classes such as with GPT-3 tokens [3]. Yet, we also evaluate arbitrarily long labels, marked $*$ in Table 2, by setting $\ell = \log n + \sigma$.

We vary network bandwidth to emulate the effect of fast LAN networks with 1 GBit/s, inter-continental WANs (100 MBit/s), and even slower cell phone networks (10 MBit/s).

We measure total runtime for all schemes, i.e., from the time receiver R starts until they output misclassifications. The communication complexity comprises all data sent or received by R. For all benchmarks, we set $\lambda = 128$ and $\sigma = 40$. To achieve a failure probability of less than 2^{-40} for Cuckoo hashing, we use $m = 1.27n$ and three hash functions [32].

Finally, we also present estimates for monetary costs incurred by these schemes. One could imagine that sender and receiver are running in cloud environments where CPU time and network communication have to be paid for. To estimate monetary costs, we assume pricing from an Amazon US East t3.medium AWS instance [2]. Given current throughputs over the Internet, we chose the evaluation setup that is closest to a cloud setup over different data centers. i.e., 100 MBit/s throughput.

Table 3 presents total costs (CPU and communication) in US$ for 100 executions of each scheme.

Discussion. Our constructions and circuit-PSI outperform each other depending on the choice of ℓ and the available network bandwidth.

For a number of labels of up to thousand (2^{10}) as in current deep neural network classifiers and medium to very fast networks, VOLE-PDC_2 offers the lowest total runtime. It is between 50% and 2000% faster than circuit-PSI, depending on the concrete choice of ℓ and bandwidth. In slow networks (10 MBit/s) and $\ell \leq 20$, DH-PDC_2 is fastest due to it having lower communication requirements. As label lengths grow to arbitrary, circuit-PSI becomes faster than both PDC_2 schemes, due to its lower communication requirements. However, if differentially private leakage is admissible, PDC_1 is the fastest approach with label lengths $\ell \geq 10$. It is between 387% and 764% faster than circuit-PSI, depending on the concrete choice of n, ℓ, and bandwidth.

Surprisingly, despite the computational simplicity of the KKRT-OPRF, the runtime of KKRT-PDC_2 is always worse than the one of VOLE-PDC_2. The savings in communication of the VOLE-OPRF outweigh any computational advantages of the KKRT-OPRF in our specific setting.

There exist many scenarios where the amount of communication, e.g., over the Internet, mobile networks or in a cloud setting, matters most and dominates total cost [18, 19, 39]. In such a setting, DH-PDC_2 is always the cheapest option with no leakage, see Table 3. Depending on the concrete choice of ℓ, circuit-PSI costs between 1% and 3200% more than DH-PDC_2. Again, if differentially private leakage is admissible, and label lengths become long $\ell \geq 10$, PDC_1 is the cheapest approach. It costs between 485% and 929% less than circuit-PSI, depending on the choice of n, ℓ, and bandwidth.

Note that we have omitted KKRT-PDC_2 from Table 3, since it is always outperformed in both computation and communication by VOLE-PDC_2.

6 Related Work

Private data cleaning (PDC) can be implemented using extended PSI, such as circuit-PSI, but we have shown that it also can be reduced to regular PSI. While the literature on PSI is too extensive to summarize it in this paper, we are not aware of related work investigating the connection between data cleaning and PSI or related work that considers mismatch (of labels) in PSI.

Labels associated with their data elements have been considered in the literature as labeled PSI [5, 6]. In labeled PSI, instead of a bit indicating inclusion in the intersection, the output is the label for each element in the intersection.

Circuit-PSI [4, 20, 33] which allows computing arbitrary circuits over the function also operates on "payloads", which are similar to labels and that we use to build our strawman.

Privacy in data cleaning has so far been considered in the single database setting. It has previously been investigated how privacy impacts the querying party and how it can be improved by tailoring data cleaning methods [23]. It has also been previously investigated how privacy can be tailored for the data scientist performing the data cleaning [12].

Collaborative data cleaning without consideration of privacy has also been investigated [28]. However, the obvious privacy implications hinder deployment, and our paper addresses the problem in a systematic and formal manner.

7 Conclusion

In this paper, we have formalized the problem of private collaborative data cleaning (PDC) and investigated its connection to PSI. Private collaborative data cleaning is an important primitive in data science that leads to better data and hence better models with more accurate predictions. Just as PSI, it arises in many data science applications.

While PDC can be solved using circuit-PSI, we show that its efficiency does not scale to large data set sizes. We present a construction that has complexity independent of the number of possible labels, but has a small leakage, and we present a construction that reduces PDC to PSI at the expense of increasing the complexity by the size of the possible labels. However, when the PSI-based construction is implemented using (mostly) symmetric cryptography, its efficiency for current big data sizes is very practical. We combine the currently most efficient oblivious pseudo-random functions by Kolesnikov et al. [21] and Rindal and Schoppmann [37] with a new technique for shuffling its outputs in this second, most efficient protocol. As a result, we achieve total runtime or communication improvements of up to one order of magnitude over circuit-PSI.

References

[1] Rakesh Agrawal, Alexandre V. Evfimievski, and Ramakrishnan Srikant. Information sharing across private databases. In *Proceedings of the ACM Conference on Management of Data, (SIGMOD)*, pages 86–97, 2003.

[2] Amazon. Amazon EC2 On-Demand Pricing, 2022. `https://aws.amazon.com/ec2/pricing/on-demand/`.

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia

Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL `https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html`.

[4] Nishanth Chandran, Divya Gupta, and Akash Shah. Circuit-psi with linear complexity via relaxed batch OPPRF. *Proceedings of Privacy Enhancing Technologies*, 2022(1):353–372, 2022.

[5] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. Labeled PSI from fully homomorphic encryption with malicious security. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 1223–1237, 2018.

[6] Kelong Cong, Radames Cruz Moreno, Mariana Botelho da Gama, Wei Dai, Ilia Iliashenko, Kim Laine, and Michael Rosenberg. Labeled PSI from homomorphic encryption with reduced computation and communication. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 1135–1150, 2021.

[7] Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. Fast and private computation of cardinality of set intersection and union. In Josef Pieprzyk, Ahmad-Reza Sadeghi, and Mark Manulis, editors, *Proceedings of the 11th International Conference on Cryptology and Network Security (CANS)*, pages 218–231, 2012.

[8] Henry de Valence, Jack Grigg, Mike Hamburg, Isis Lovecruft, George Tankersley, and Filippo Valsorda. Internet draft: The ristretto255 and decaf448 Groups. Technical report, 2022. https://datatracker.ietf.org/doc/draft-irtf-cfrg-ristretto255-decaf448/03/.

[9] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 248–255, 2009.

[10] Michael J Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *Proceedings of the 23rd International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 1–19, 2004.

[11] Gayathri Garimella, Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. Oblivious key-value stores and amplification for private set intersection. In *Proceedings of the 41st International Cryptology Conference (CRYPTO)*, pages 395–425, 2021.

[12] Chang Ge, Xi He, Ihab F. Ilyas, and Ashwin Machanavajjhala. Apex: Accuracy-aware differentially private data exploration. In *Proceedings of the ACM Conference on Management of Data, (SIGMOD)*, pages 177–194, 2019.

[13] Johan Håstad, Russell Impagliazzo, Leonid A. Levin, and Michael Luby. A pseudorandom generator from any one-way function. *SIAM Journal on Computing*, 28(4):1364–1396, 1999.

[14] Xi He, Ashwin Machanavajjhala, Cheryl J. Flynn, and Divesh Srivastava. Composing differential privacy and secure computation: A case study on scaling private record linkage. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 1389–1406, 2017.

[15] Bernardo A. Huberman, Matthew K. Franklin, and Tad Hogg. Enhancing privacy and trust in electronic communities. In *Proceedings of the ACM Conference on Electronic Commerce (EC)*, pages 78–86, 1999.

[16] Bert Hubert, Jacco Geul, and Simon Sehier. Wondershaper, a command-line utility for limiting an adapter's bandwidth, 2021. `https://github.com/magnific0/wondershaper`.

[17] Ihab F. Ilyas and Xu Chu. *Data Cleaning*. Association for Computing Machinery, 2019.

[18] 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. *IACR Cryptology ePrint Archive*, page 738, 2017. URL `http://eprint.iacr.org/2017/738`.

[19] 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 *Proceedings of the IEEE European Symposium on Security and Privacy, (EuroS&P)*, pages 370–389, 2020.

[20] Ferhat Karakoç and Alptekin Küpçü. Linear complexity private set intersection for secure two-party protocols. In *Proceedings of the 19th International Conference on Cryptology and Network Security (CANS)*, pages 409–429, 2020.

[21] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient batched oblivious PRF with applications to private set intersection. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 818–829, 2016.

[22] Vladimir Kolesnikov, Naor Matania, Benny Pinkas, Mike Rosulek, and Ni Trieu. Practical multi-party private set intersection from symmetric-key techniques. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 1257–1272, 2017.

[23] Sanjay Krishnan, Jiannan Wang, Michael J. Franklin, Ken Goldberg, and Tim Kraska. Privateclean: Data cleaning and differential privacy. In *Proceedings of the ACM Conference on Management of Data, (SIGMOD)*, pages 937–951, 2016.

[24] A. Krizhevsky. Learning Multiple Layers of Features from Tiny Images. Technical report, Univ. of Toronto, 2009. `https://www.cs.toronto.edu/~kriz/cifar.html`.

[25] libSodium. A modern, portable, easy to use crypto library, 2022. `https://github.com/jedisct1/libsodium`.

[26] Catherine A. Meadows. A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 134–137, 1986.

[27] Ilya Mironov, Omkant Pandey, Omer Reingold, and Salil P. Vadhan. Computational differential privacy. In *Proceedings of the 29th International Cryptology Conference (CRYPTO)*, pages 126–142, 2009.

[28] Mashaal Musleh, Mourad Ouzzani, Nan Tang, and AnHai Doan. Coclean: Collaborative data cleaning. In *Proceedings of the ACM Conference on Management of Data, (SIGMOD)*, pages 2757–2760, 2020.

[29] Eliud Nduati. A data cleaning journey, 2021. `https://medium.com/analytics-vidhya/a-data-cleaning-journey-2b0146407e44`.

[30] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*, pages 515–530, 2015.

[31] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. Efficient circuit-based psi via cuckoo hashing. In *37th International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT))*, pages 125–157, 2018.

[32] Benny Pinkas, Thomas Schneider, and Michael Zohner. Scalable private set intersection based on OT extension. *ACM Transactions on Privacy and Security*, 21(2):7:1–7:35, 2018.

[33] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. Efficient circuit-based PSI with linear communication. In *Proceedings of the 38th International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 122–153, 2019.

[34] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. PSI from paxos: Fast, malicious private set intersection. In *39th International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT))*, pages 739–767, 2020.

[35] Srinivasan Raghuraman and Peter Rindal. Blazing fast psi from improved okvs and subfield vole. Cryptology ePrint Archive, Paper 2022/320, 2022. URL `https://eprint.iacr.org/2022/320`. `https://eprint.iacr.org/2022/320`.

[36] Peter Rindal. libOTe: an efficient, portable, and easy to use Oblivious Transfer Library, 2022. `https://github.com/osu-crypto/libOTe`.

[37] Peter Rindal and Phillipp Schoppmann. VOLE-PSI: fast OPRF and circuit-PSI from vector-OLE. In *Proceedings of the 40th International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pages 901–930, 2021.

[38] Ristretto. The Ristretto Group, 2022. `https://ristretto.group/`.

[39] Mike Rosulek and Ni Trieu. Compact and malicious private set intersection for small sets. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 1166–1181, 2021.

[40] European Union. GDPR, Chapter II, Art. 5(1), Principles relating to processing of personal data, 2022. `https://eur-lex.europa.eu/eli/reg/2016/679/oj`.

[41] Jelle van den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 137–152, 2015.

[42] Visa-Research. Efficient Private Set Intersection base on VOLE, 2022. `https://github.com/Visa-Research/volepsi`, Commit $\#eb788bb$.

[43] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit, 2016. `https://github.com/emp-toolkit`.

[44] Alex Woodie. Data prep still dominates data scientists' time, survey finds, 2020. `https://www.datanami.com/2020/07/06/data-prep-still-dominates-data-scientists-time-survey-finds/`.