Abstract

We extend two-party private set union for secure computation, by considering matching between records having multiple identifiers (or keys), for example email and phone. In the classical setting of this problem, two parties want to perform various downstream computations on the union of two datasets. The union is computed by joining two datasets with the help of a single agreed upon identifier, say email. By extending this to joining records with multiple identifiers, we bring it much closer to real world uses where the match rate and match quality can be greatly improved by considering multiple identifiers.

We introduce an extension to the Private-ID protocol [3] which outputs a full outer join (union) of two datasets by a match logic that can join rows containing multiple identifiers. We also introduce new techniques for privately sharding the protocol across multiple servers. Both constructions are based on Decisional Diffie–Hellman (DDH) assumptions.

Keywords: private set intersection, private identity matching

1 Introduction

Joining records across multiple data owners is an essential precursor to many applications; from gathering aggregate statistics to training machine learning models. For example, computing a test statistic of a randomized controlled trial requires such a join when one party owns a test/control group assignment data, and the other party owns the outcome data [14]. Another application is a model that calculates the risk of a specific health condition, where case-specific health condition labels are known by one party, and the predictive features are known by the other party [3].

Private Set Intersection (PSI) [13, 8, 6, 11] offers a way to join two sets and learn the intersection membership without revealing anything outside the intersection. Private-ID and Private Secret Shared Set Intersection (PS3I) protocols [3, 7], extend this functionality by decoupling the matching phase of these protocols from the computation phase that acts upon the features associated with these records. This allows for a richer set of computation to happen on the associated features in a different privacy computation framework. This also allows for the matching step to be done once while allowing the later addition of features associated with these records.

However, the drawback of these protocols, is they only allowed for one identifier per record to be used in the join logic. In practice, the type and quality of identifiers may vary across data owners, and it is often highly advantageous to match on multiple identifiers to improve the match quality and match rate. For instance, two data owners may wish to match on both email address and phone number, to improve coverage for records with incomplete information. A related work [12] that has considered matching on multiple keys uses different techniques including Garbled Circuits.

In terms of scalability, while the Private-ID protocols are multi-threaded, they run on a single server per party. This is a serious impediment to scaling to hundreds of millions of records. Naive
sharding across multiple servers for improvements in scale and performance is possible in the
single-key case but would reveal the intersection size of each shard; in the multi-key case such naive
sharding solutions applied to the inputs are not even possible.

Thus, we set out to extend the Private-ID union protocol to support joining based on multiple
identifiers per record and to allow for sharding across multiple servers, while preserving its privacy
 guarantees.

2 Our Contribution

We extend the Private-ID protocols with the following functionalities.

- **Deterministic ranked join using multiple identifiers**: We organize multiple identifiers
  as sets and construct pseudorandom Universal Identifiers (UID) corresponding to the records
  created by union of both datasets. Joining on multiple identifiers typically results in many-to-
  many connections, while Private-ID enforces one-to-one mapping. To circumvent this issue, we
  collapse many-to-many connections to one-to-one connections by choosing to match according
  to an ordering of identifiers set by one of the parties.

- **Sharding** We present an extension to the Private-ID protocol to allow sharding across multiple
  servers without leaking any additional information.

We implement the multi-key Private-ID protocol in Rust programming language, and evaluate it’s
performance under multiple settings. Our results indicate that multi-key Private-ID is 3X slower
relative to the single-key protocol (3 min 52 sec vs 1 min 2 sec for a million records with single key
in each variant) but incurred the same communication cost.

3 Multi-key Matching

3.1 Problem setup

Records are often indexed by one or more identifiers (e.g. email address, phone number), and the
notion of an individual defined by a certain combination of identifiers may differ across data owners.
For example, a health care provider may represent patients using a comprehensive set of identifiers
such as social security number, phone number, and email address while a fitness subscription service
may identify subscribers using only an email address.

In both multi-key and single-key based matching a link is established across datasets with exact
matching on identifiers (i.e. no fuzzy matches allowed, although it’s possible to add regex variants
as new identifiers). Two datasets with records indexed by a single identifier may be joined using
exact matching on the common identifier. However, the presence of multiple identifiers allows for
flexible join logic and often yields many-to-many connections. For example, both Party $C$ and Party
$P$ may represent individuals using email and phone number. However, one of Party $P$’s customers
may use a current phone number at the time of purchase, but may not have updated the phone
number on Party $C$’s platform. In this case, the phone numbers may not match, however; matching
on email is still feasible. Another of Party $P$’s customers may utilize a household member’s phone
number to set up a proxy shipment recipient for a purchase albeit using a personal email to track
the shipment. Party $C$’s dataset may then represent two distinct users for the purchaser and the
household member, with respective identifiers. This results in one-to-many connection between
Party $P$ and Party $C$’s datasets when they come together to perform a join (see Figure 1). In
addition, neither parties may be willing to share their notion of an individual with other Party.
3.2 Matching logic

The above examples highlight the need for matching protocols with arbitrary join logic and arbitrary number of identifiers. Private-ID style protocols enforce a one-to-one mapping as the output, hence we leverage a ranked deterministic join logic that collapses many:many connections to one-to-one as described below.

In the case of *many-to-one* connections (i.e. multiple records from the first dataset may be linked to one record in the second dataset if there is at least one common identifier), then we use a predefined identifier ranking to resolve such conflicts by iteratively matching on identifiers. The ordering of identifiers is set by one of the parties performing the matching. In the first round, we match all records on the first identifier (akin to single-key based matching), then proceed to a second round to match on second identifier, but limiting to unmatched records from prior round. The process is continued as many times as the maximum number of identifiers present within the records.

In the case of *one-to-many*, (when one record from the party that chooses the ranking order has identifiers that belong to multiple records in the second party’s dataset), the matching process resolves randomly. At the end, the matching logic only outputs at most one link between the records from both datasets. Note that the protocol may be trivially extended to other similar join logic implementations.

Figure 1 demonstrates an example matching scenario. The aforementioned process maps Party P’s User A to Party C’s user A, if Party C chooses to prefer matching on email followed by phone number in this scenario. In other words, Party C may trust that individual may not share email addresses while its possible to share phone numbers with household members, hence prefer to match on email while falling back to phone number in cases where email is unavailable or unmatched to maximize match cardinality or intersection size. Note that the above examples are not limitations of the proposed protocol rather are a specific embodiment of its usage.

In full generality, we can think of the many-to-many mapping between the two sets as a bipartite graph. The ranked match logic then defines the weights of the edges in the graph. Since we want to only output a one-to-one matching, the problem we are looking to solve is the globally optimal bipartite matching problem. We could use an algorithm, such as the Hungarian algorithm, to solve for the globally optimal set of matches; however, we currently use a greedy algorithm which selects a random node from one set and then picks its best match. This performs well heuristically but does not guarantee an optimal solution.
4 Protocol

4.1 Setup

\( C \) and \( P \) are sets of \( n_c \) and \( n_p \) records from two different parties, \( C \) and \( P \) respectively, consisting of arrays of identifiers of varying sizes,

\[ C : \{ c_i : (c^1_i, c^2_i, \ldots, c^l_i), i \in \{1, 2, \ldots, n_c\}\} \]

\[ P : \{ p_i : (p^1_i, p^2_i, \ldots, p^m_i), i \in \{1, 2, \ldots, n_p\}\} \]

Each array \((c^1_i, c^2_i, \ldots, c^l_i)\) is pre-ordered based on identifier priority by party \( C \). The ordering varies from row to row and reflects party \( C \)'s belief about the importance of identifier \( c^j_i \) in defining a connection with party \( P \)'s record containing \( c^j_i \).

The protocol outputs a set of Universal Identifiers (UID), denoted by \( UID = \{ uid_1, \ldots, uid_{|UID|}\} \) where \( |UID| = |C \cup P| \) to both parties. In addition to \( UID \), party \( C \) learns a map \( M_c \), where \( M_c[uid_i] = c_i \) if \( c_i \in C \) and \( M_c[uid_i] = \perp \) otherwise. Similarly, party \( P \) learns a map \( M_p \), where \( M_p[uid_i] = p_i \) if \( p_i \in P \) and \( M_p[uid_i] = \perp \) otherwise. \( M_c \) and \( M_p \) enable the usage of UID, for any downstream secure computation using the features associated with records \( C \) and \( P \).

4.2 Protocol Overview

Step 1, Exchange records. Our starting point is a DDH based scheme \cite{8, 10, 5, 9, 3}.

In Step \( 1 \) of the protocol, (Step 1 in Figures 2 and 3) party \( C \) hashes its records componentwise \( c_i = (c^1_i, c^2_i, \ldots, c^l_i) \) as \( H(c_i) = (H(c^1_i), H(c^2_i), \ldots, H(c^l_i)) \) and exponentiates each component using a random secret scalar \( k_c \), \( H(c_i)^{k_c} \). Party \( P \) also computes \( H(p_j)^{k_p} \) for each of its records. These random secret scalars are shown as keys in Figure 2. Both parties shuffle and exchange these encrypted records, denoted as the sets \( U_c = \{ H(c_i)^{k_c} \} \) and \( U_p = \{ H(p_j)^{k_p} \} \).

In Step \( 1 \), party \( C \) also computes \( H(p_j)^{k_p k_c} \) and similarly party \( P \) computes \( H(c_i)^{k_c k_p} \). These double exponentiated Diffie-Hellman (DH) values are denoted by \( E_p \) and \( E_c \) and shown in Figure 2 using two lock symbols. A first natural attempt is to use a component of these DH values as UID for the universe, such that \( UID = E_c \cup E_p \). However this reveals the items in the intersection to party \( C \). A common solution to avoid this leakage is for \( C \) to receive \( E_c \) randomly shuffled, so that it only learns the size of the intersection, but this breaks the linkages between the universal identifiers and their corresponding values in \( C \)'s set. Instead, each party uses one more random secret scalar \( r_c \) and \( r_p \) to calculate the eventual UID’s of the form \( H(c^α_i)^{k_p k_c r_p r_p} \) or \( H(p^α_j)^{k_p k_c r_p r_p} \), where each party will be able to link UIDs to its users, but even the party performing the matching will not learn which records are in the intersection.

Step 2, Matching. In Step \( 2 \) party \( C \) calculates the matching between the sets \( E_c \) and \( E_p \) following a ranked deterministic match process. If many-to-one (party \( C \) to party \( P \)) connections arise, they are resolved using iterative matching leveraging predefined identifier ranking defined by party \( C \). If one-to-many connections arise, they are resolved randomly. The output of this matching process is four sets \( V_c, V_p, S_c, S_p \). The sets \( V_c \) and \( E_c \) are in one-to-one correspondence with the element in \( V_c \) being either the first component of the corresponding array in \( E_c \) or a later component if a match was selected using that later component. The same is true for \( V_p \) and \( E_p \). The sets \( S_c \) and \( S_p \) are made from the first components of the rows of \( E_c \) and \( E_p \) that were never matched. Also, party \( C \) exponentiates the elements in the set \( V_c, V_p \) by its
Step 1

Exponentiate and Shuffle

<table>
<thead>
<tr>
<th>k_c</th>
<th>r_c</th>
<th>s_c</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>81</td>
<td></td>
</tr>
</tbody>
</table>

Exchange and Exponentiate

<table>
<thead>
<tr>
<th>k_p</th>
<th>r_p</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>43</td>
</tr>
<tr>
<td>4</td>
<td>61</td>
</tr>
<tr>
<td>5</td>
<td>67</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

Shuffle and Send to Party C

Step 2

Matching Input

<table>
<thead>
<tr>
<th>E_c</th>
<th>E_p</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>61</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>67</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Matching Output, sent to Party P

Create Mappings

<table>
<thead>
<tr>
<th>M_c</th>
<th>U_c</th>
<th>H_c</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>3</td>
<td>67</td>
</tr>
<tr>
<td>7</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>81</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>null</td>
<td></td>
</tr>
</tbody>
</table>

Unshuffle, Exponentiate

Figure 2: Protocol Diagram
second key $r_c$ and $V_c$ by a third key $s_c$ and exponentiates the elements of the set $S_c$ to form the set $S'_c$. The four sets are then sent to party $P$.

**Step 3, Output mapping.** In Step 3, party $P$ can unshuffle $V_p$ to match the original order of set $P$. Combining this with the elements in $S'_c$ and exponentiating by its second key $r_p$, party $P$ is able to form its copy of the set of UIDs and mapping to its original records. Party $P$ also exponentiates and unshuffles the sets $V_c$ and $S'_p$ (denoted as $W_c$ and $S'_p$) and sends these back to party $C$ so that party $C$ can similarly unshuffle $W_c$ to align with its original records and combine with $S'_p$ to get its copy of UIDs and mapping.
4.3 Protocol Details

\[ \Pi^{\text{PID-multi-key}} \]

**Inputs:**
Party C: \(\{c_i : (c_i^1, c_i^2, \ldots, c_i^l_i), i \in (1, 2, \ldots, n_c)\}\).
Pre-arrange elements of \(c_i\) according to row-level (vary with \(i\)) key-priority (only by Party C)

Party P: \(\{p_i : (p_i^1, p_i^2, \ldots, p_i^{m_i}), i \in (1, 2, \ldots, n_p)\}\)

**Outputs:**
\([C : (\text{UID}, M_c), P : (\text{UID}, M_p)]\)

Let \(G\) be a cyclic group of order \(q\) with generator \(g\) wherein DDH is hard, and \(H(\cdot) : \{0, 1\}^* \rightarrow G\) modeled as a random oracle.

**Step 1 (Exchange records): Party C**
- Let \(k_c, r_c, s_c \overset{R}{\leftarrow} \mathbb{Z}_q\), and \(U_c \leftarrow \emptyset\).
- For each \(c_i \in C\) compute \(u^i_c = H(c_i)^{k_c} = (H(c_i^1)^{k_c}, H(c_i^2)^{k_c}, \ldots, H(c_i^l_i)^{k_c})\), and let \(U_c = U_c \cup \{u^i_c\}\), where \(u^i_c\) will be of size \(l_i\).
- Randomly (outer) shuffle the elements in \(U_c\) using a permutation \(\pi_{U_c}\), however, NOT inner shuffling elements within \(c_i\), and send to P.

**Step 1 (Exchange records): Party P**
- Let \(k_p, r_p \overset{R}{\leftarrow} \mathbb{Z}_q\), and \(U_p, E_c \leftarrow \emptyset\).
- For each \(p_i \in P\) compute \(u^i_p = H(p_i)^{k_p} = (H(p_i^1)^{k_p}, H(p_i^2)^{k_p}, \ldots, H(p_i^{m_i})^{k_p})\), and let \(U_p = U_p \cup \{u^i_p\}\), where \(u^i_p\) will be of size \(m_i\).
- Randomly shuffle the elements in \(U_p\) using a permutation \(\pi_{U_p}\), and inner shuffling elements within \(p_i\) as well (no need to store inner shuffle).
- For each \(u^i_p \in U_p\) received form C:
  - Compute \(e^i_p = (u^i_p)^{k_c}\) and let \(E_c = E_c \cup \{e^i_p\}\)
- Randomly shuffle the elements in \(E_c\) using a permutation \(\pi_{E_c}\), and send the sets \(E_c, U_p\) to C

**Step 2 (Matching): Party C**
- Let \(E_p, V_c, V_p, S_c, S_p^c, S_p \leftarrow \emptyset\)
- For each \(u^i_p \in U_p\):
  - Compute \(e^i_p = (u^i_p)^{k_c}\) and let \(E_p = E_p \cup \{e^i_p\}\)
- Initialize
  \[ S_c = \{c^i_c[1] : i \in (1, 2, \ldots, n_c)\}\]
  \[ S_p = \{e^i_p[1] : i \in (1, 2, \ldots, n_p)\}\]
  \[ V_c = \{(c^i_c[1])^{r_c} : i \in (1, 2, \ldots, n_c)\}\]
  \[ V_p = \{(e^i_p[1])^{r_c} : i \in (1, 2, \ldots, n_p)\}\]

Figure 3: Private-ID multi-key matching with key priority
Step 3 (Output mapping): P

- Let \( W_c, W_p, S_p^\alpha, S_p^\beta = \emptyset \)
- Create \( UID_P: \)
  - Shuffle back the elements of \( V_p \) using \( \pi_{\nu_p}^{\nu_p^{-1}} \). For every \( \nu_p \) in \( V_p \), let \( W_p = W_p \cup \{(\nu_p)^{r_p}\} \), and \( M_p[(\nu_p)^{r_p}] = p_i \)
  - For each \( s_p^p \) in \( S_p^\alpha \), let \( S_p^\alpha = S_p^\alpha \cup \{(s_p^p)^{r_p}\} \) and \( M_p[(s_p^p)^{r_p}] = \bot \)
  - Output \( UID_p = W_p \cup S_p^\alpha \) and \( M_p \)
  - For each \( s_p^p \) in \( S_p^\beta \), let \( S_p^\beta = S_p^\beta \cup \{(s_p^p)^{r_p}\} \)
  - Shuffle back the elements of \( V_c \) using \( \pi_{\nu_c}^{\nu_c^{-1}} \). For every \( \nu_c \) in \( V_c \), let \( W_c = W_c \cup \{(\nu_c)^{r_c}\} \)
  - Send \( S_p^\beta, W_c \) to C

Step 3 (Output mapping): C

- Let \( W_c, S_p^\alpha = \emptyset \)
- Shuffle back the elements of \( W_c \) using \( \pi_{\kappa_c}^{\kappa_c^{-1}} \). For every \( \kappa_c \) in \( W_c \), let \( W_c = W_c \cup \{(\kappa_c)^{r_c}\} \)
  - \( M_c[(\kappa_c)^{r_c}] = c_i \)
  - For every \( s_p^p \) in \( S_p^\beta \), let \( S_p^\beta = S_p^\beta \cup \{(s_p^p)^{r_p}\} \) and \( M_p[(s_p^p)^{r_p}] = \bot \)
  - Output \( UID_c = W_c \cup S_p^\beta \) and \( M_c \)

Figure 4: Private-ID multi-key matching with key priority, continued
5 Security and Privacy

The privacy of a system is in some sense measured by the amount of information that can be gleaned from a secure system. The current design leaks the following information:

- Both parties learn the size of the intersection. It is clear that party $C$ learns the intersection size while computing the matching. Party $P$ learns it through knowing $|\mathcal{P}|$ and seeing $|\mathcal{C}|$ and $|\mathcal{C} \cup \mathcal{P}|$, thus learning $|\mathcal{C} \cap \mathcal{P}| = |\mathcal{C}| + |\mathcal{P}| - |\mathcal{C} \cup \mathcal{P}|$. This leakage is generally benign. However, if the protocol is run multiple times with a single record of identifiers differing, it can reveal membership.

- Party $C$ gets to see the full bipartite graph of matches up to an isomorphism. Since we do not shuffle the identifiers in each record for party $C$, it also sees the number of matches that happen at each location within a record.

This leakage is acceptable as its an aggregated metric.

- Both parties learn the distributions of the number of identifiers per record in the other party’s data. However, this can be avoided by padding dummy identifiers for both parties at the expense of additional compute.

5.1 Security of multi-key Private-ID, $\prod^{\text{PID}}$

We use standard simulation-based definitions of security for secure multiparty computation to prove that the protocol is secure against a semi-honest (honest-but-curious) adversary. In particular, the security argument is split into two pieces, one against a corrupted $C$ and another against a corrupted $P$.

In each case, we describe a simulator $\text{SIM}$ that only takes the corrupted party’s input, the size of the two sets $\mathcal{C}$ and $\mathcal{P}$ (and in case of corrupted $P$ also size of $\mathcal{C} \cap \mathcal{P}$ and in the case of a corrupted $C$ a graph $\mathcal{G} \cong (\tilde{\mathcal{C}}, \tilde{\mathcal{P}})$ which is isomorphic to the bipartite graph of matches between the sets $\mathcal{C}$ and $\mathcal{P}$) as input and indistinguishably simulates the view of that party in the real protocol. In the graph $\mathcal{G}$ the sets $\tilde{\mathcal{C}}$ and $\tilde{\mathcal{P}}$ can be thought of as applying an OPRF to the values in $\mathcal{C}$ and $\mathcal{P}$ and shuffling their rows. The view of a party consists of its inputs, the randomness it uses, as well as messages sent and received throughout the protocol. More formally, let $\text{REAL}^{\text{PID}}_{\prod_{\text{PID}}}^{C,\lambda}(\mathcal{C}, \mathcal{P})$ be a random variable representing the view of party $a$ in a real protocol execution where the random variable ranges over the internal randomness of both parties. Our first theorem captures security against a corrupted $C$ as follows.

**Theorem 1** (Security of $\prod^{\text{PID}}$ against a semi-honest $C$). There exists a PPT simulator $\text{SIM}_C$ such that for all security parameters $\lambda$ and all inputs $\mathcal{C} = \{c_1, \ldots, c_n\}$ and $\mathcal{P} = \{p_1, \ldots, p_m\}$,

$$\text{REAL}_{\prod^{\text{PID}}}^{C,\lambda}(\mathcal{C}, \mathcal{P}) \approx \text{SIM}_C(\mathcal{C}, 1^{\lambda}, m, n, \mathcal{G})$$

where $\mathcal{G} \cong (\tilde{\mathcal{C}}, \tilde{\mathcal{P}})$ is a graph isomorphic to the bipartite graph of matches between the sets $\mathcal{C}$ and $\mathcal{P}$.

**proof sketch.** In Figure 5, we describe the simulator $\text{SIM}_C$ which we claim indistinguishably simulates the real view of party $C$.

Using a sequence of hybrid arguments, we show that the distribution generated by $\text{SIM}_C$ is indeed indistinguishable from the real view of $C$. 


Simulate $C$’s step 1:
- Generate $k_c, r_c, s_c \overset{R}{\leftarrow} \mathbb{Z}_q$
- Honestly generate $U_c$, i.e. for each $c_i \in C$ compute $u^i_c = H(c_i)^{k_c}$ and let $U_c = U_c \cup \{u^i_c\}$.

Simulate $P$’s step 1:
- For each $i \in [n]$ compute $g_i \overset{R}{\leftarrow} G$, and let $E_c = E_c \cup \{g^k_c\}$.
- Construct the set $U_p$ to have the same structure of matches with $E_c$ as $G$, for matches components letting $u^i_p = e^{\alpha_{i,\gamma}}$ and for non-matches letting $u^i_p \overset{R}{\leftarrow} G$.
- Let $V_c = \{v_1, \ldots, v_n\}$ where all $v_i$’s are randomly selected from $G$.
- Randomly shuffle the elements in $E_c, U_p$ and send the sets $E_c, V_c, U_p$ to $C$.

Simulate $C$’s step 2: $\text{SIM}_c$ does this step exactly as the protocol describes and using $r_c, k_c$, and $s_c$ it generated above. So we skip the full details. At the end of this step $\text{SIM}$ outputs $V_p, S'_c, S_p$ for $P$.

Simulate $P$’s step 2:
- Let $J = m - \ell$, where $\ell = |C \cap \mathcal{P}|$. For $i \in [J]$, let $S'_p = S'_p \cup \{s_i\}$ for randomly selected $s_i$ in $G$, and send $S'_p$ to $C$.

Simulate $C$’s step 3: $\text{SIM}_c$ does this step exactly as the protocol describes and using $r_c$ and $s_c$ it generated above.

$\mathcal{H}_0$: This is the view of party $C$ in the real execution of the protocol.

$\mathcal{H}_{1,0}$: Identical to $\mathcal{H}_0$.

$\mathcal{H}_{1,i,\alpha_i}$: Let $(i, \alpha_i)$ range over the individual identifiers in $C$ which are not also in $\mathcal{P}$. $\mathcal{H}_{1,i,\alpha_i-1}$ is the same as $\mathcal{H}_{1,i,\alpha_i}$ except that we replace $H(c_i^{\alpha_i})^{k_c}k_p$ in $E_c$ with $g^k_c$ for random $g_i \in G$.

$\mathcal{H}_{2,0}$: Identical to the last $\mathcal{H}_{1,i,\alpha_i}$.

$\mathcal{H}_{2,j,\alpha_j}$: Let $(j, \alpha_j)$ range over the individual identifiers in $\mathcal{P}$ but not in $C$. $\mathcal{H}_{2,j,\alpha_j-1}$ is the same as $\mathcal{H}_{2,j,\alpha_j}$ except that we replace $H(p_j^{\alpha_j})^{k_p}$ in $U_p$ with random $h_j \in G$.

$\mathcal{H}_{3,0}$: Identical to the last $\mathcal{H}_{2,j,\alpha_j}$.

$\mathcal{H}_{3,t,\alpha_t}$: Let $(t, \alpha_t)$ range over the individual identifiers in $C$ that also appear in $\mathcal{P}$. $\mathcal{H}_{3,t,\alpha_t-1}$ is the same as $\mathcal{H}_{3,t,\alpha_t}$ except that we replace $H(c_t^{\alpha_t})^{k_c}k_p$ in $E_c$ with $g^k_c$ and $H(p_t^{\alpha_t})^{k_p}$ in $U_p$ with $g_t$ where $(t^*, \alpha_t^*)$ is the index of the element matching $c_t^{\alpha_t}$ in $\mathcal{P}$.

$\mathcal{H}_{4,0}$: Identical to the last $\mathcal{H}_{3,t,\alpha_t}$.

$\mathcal{H}_{4,i}$: for $i \in [n]$, the same as $\mathcal{H}_{4,i-1}$ except that we replace $v_i \in V_c$ with a randomly selected element in $G$.

$\mathcal{H}_{5,0}$: Identical to $\mathcal{H}_{4,n}$.

$\mathcal{H}_{5,i}$: for $i \in [m - \ell]$, the same as $\mathcal{H}_{5,i-1}$ except that we replace $s_i \in S'_c$ with a randomly selected element in $G$.
$H_6$ : The view of $C$ output by $\text{SIM}_c$.

We now need to argue that each consecutive pair of hybrids in the above sequence are indistinguishable by a PPT algorithm. The interesting arguments here are those for $(H_{1.1,\alpha_1-1}, H_{1.1,\alpha_1})$, $(H_{2.1,\alpha_2-1}, H_{2.1,\alpha_2})$, $(H_{3.1,\alpha_3-1}, H_{3.1,\alpha_3})$, $(H_{4.1-1}, H_{4.1})$ and $(H_{5.1-1}, H_{5.1})$. Given that they all follow a similar line of argument that relies on hardness of DDH and the random oracle property of the hash function, we go through the argument for $(H_{1.1,\alpha_1-1}, H_{1.1,\alpha_1})$ as an example. In particular, we argue that for any PPT adversary $A$ who can distinguish the two hybrids, we devise an adversary $B$ who can solve the DDH problem. $B$ is given $(g, g^a, g^b, g^c)$ and needs to decide whether $c$ is random or $c = ab$. First note $B$ can program $H(\cdot)$ to return $g^b$ on input $c_i^{\alpha_i}$. We also let $g^a = g^b$. Then it is easy to observe that since $g_i$ is uniformly random, the tuple $(g, g^a, H(c_i^{\alpha_i}), g^c)$ is identically distributed to $H_{1,1-1}$ if $c = ab$ and is identically distributed to $H_{1,1}$ if $c$ is random (since $g_i$ is uniformly random). If $A$ can decide which hybrid it is interacting with, $B$ can decide which DDH tuple it was given with the same probability.

**Theorem 2** (Security of $\prod_{\text{PID}}^\lambda$ against a semi-honest $P$). There exists a PPT simulator $\text{SIM}_p$ such that for all security parameter $\lambda$ and all inputs $C = \{c_1, \ldots, c_n\}$ and $P = \{p_1, \ldots, p_m\}$,

$$\text{REAL}_{\prod_{\text{PID}}^\lambda}(C, P) \approx \text{SIM}_p(P, 1^\lambda, m, n, \ell)$$

where $\ell = |C \cap P|$.

**proof sketch.** The description of $\text{SIM}_p$ is quite straightforward. It generates $r_p, k_p$ randomly as $P$ would, and performs all computations that $P$ does throughout the protocol using these two values as described. For all group elements to be received from $C$, $\text{SIM}_p$ replaces them with randomly generated elements in $G$. This includes elements in $U_c, V_p, S'_c, S_p$.

We will not go through a detailed sequence of hybrid arguments but note that starting from the first hybrid which is the view of $P$ in the real protocol, we sequentially replace elements sent by $C$ with random group elements until we reach the view generated by $\text{SIM}_p$. The argument we used in the proof of Theorem 1 can be plugged in here to show that each pair of consecutive hybrids are indistinguishable if DDH is hard and $H$ is a random oracle.

6 Private Sharding

To scale multi-key Private-ID to inputs of 500 million records or more, it is necessary to shard the input across multiple servers. But this process of sharding should not leak additional information about the party’s input sets. We present a sharded version of our protocol where each party runs a set of $m$ servers, where each server represents a shard of the data. Our design leaks no additional information between the two parties compared to when it is run using only one server per party. To make the sharding design simpler, we make a simplifying assumption to the match logic that each record has a single identifier per type and they are consistently ordered for both parties, across all records.

In the case of a single-key protocol, one could shard the input values the same way for both parties. Then each pair of servers could independently run the single-key Private-ID protocol between them, for their shards. This, however leaks the intersection size per shard. This method of input sharding cannot be extended to the case of multiple keys since there is no way to send a record to a single shard on which we know the match should happen. Instead, we shard not on the
 identifiers, but on their deterministic encryptions, $x^{k_c,k_p}$. We also perform the shuffles in a way that does not leak anything about the shuffle to the receiving party.

**Notation.** Assume each party has $m$ servers. We will denote these servers as $C_1,...,C_m$ for party $C$ and as $P_1,...,P_m$ for party $P$. We will denote the partitions of a set $S$ sharded across these servers as $S^{(1)},...,S^{(m)}$.

### 6.1 Private Cross Shard Shuffle

A frequent operation in the multi-key Private-ID protocol is the shuffling of records before sending to the other party. When we shuffle and send a set that is held across many servers, we need to ensure a receiving party’s servers cannot discern which records it received from which sending party. We use anonymous routing to enforce this guarantee. To send a record to a receiving server, a sending server will choose a receiving server at random, and then choose another sending server at random as an intermediate node to route it through.

A sending server will randomly shuffle and partition its shard across the $m$ destination nodes. It will then further divide each partition across $m$ intermediate nodes.

- **Round 1** (red in Figure 6): Each sending server sends the partition to the corresponding intermediate node. These intermediate nodes are other sending servers.
- **Round 2** (black in Figure 6): After receiving all data, the sending servers acting as intermediate nodes, will group all data going to a particular receiving node, shuffle the records and then send it.

From the perspective of the recipient nodes, they cannot tell which row originated from which sending node, since the records flow through a random intermediate node. Also since there are independent random shuffles at each step of the way, a row has an equal probability of ending up at any ending index; thus, the protocol samples a random permutation.
6.2 Sharded Protocol Details

We present the details of our sharded protocol in Figures 7, 8, and 9.

\[
\text{Sharded } \prod_{\text{PID--multi--key}}^\text{PID--multi--key}
\]

Inputs:
Party C: \(C = \{c_i : (c_i^1, c_i^2, \ldots, c_i^m), i \in (1, 2, \ldots, n_c)\}\).
Pre-arrange elements of \(c_i\) according to globally aligned key types (ordered by priority).

Party P: \(P = \{p_i : (p_i^1, p_i^2, \ldots, p_i^m), i \in (1, 2, \ldots, n_p)\}\).
Both parties will agree in advance as to what identifiers are allowed in each slot and will agree on
the maximum number of identifiers allowed per user so that \(\max_{\text{num keys}} := \max_{i \in (1, 2, \ldots, n_c)} l_i = \max_{i \in (1, 2, \ldots, n_p)} m_i\).

Outputs:
\([C : (UID, M_c), P : (UID, M_p)]\), but where this output is distributed across the shards with each holding
a part,
\([C_k : (UID^{(k)}, M_c^{(k)}), P_k : (UID^{(k)}, M_p^{(k)})], 1 \leq k \leq m\).

Let \(G\) be a cyclic group of order \(q\) with generator \(g\) wherein DDH is hard, and \(H(\cdot) : \{0, 1\}^* \rightarrow G\) modeled
as a random oracle.

Step 1 (Encryption): Party C
- Let one coordinator for \(C\) generate the secret keys \(k_c, r_c, s_c, \ell_c \leftarrow \mathbb{Z}_q\) and then send these over a secure
  channel to the it’s shards \(C_1, \ldots, C_m\).
- For the input set \(C\), the coordinator will shard it out to the \(C_1, \ldots, C_m\) servers sending partitions
  \(C^{(1)}, \ldots, C^{(m)}\) to each respectively.
- Then on shard \(C_k\), \(1 \leq k \leq m\), let \(U_c^{(k)} \leftarrow \emptyset\) and for each \(c_i \in C^{(k)}\)
  - compute \(u_c^i = H(c_i)^{k_c} = (H(c_i^1)^{k_c}, H(c_i^2)^{k_c} \ldots H(c_i^m)^{k_c})\), and let \(U_c^{(k)} = U_c^{(k)} \cup \{u_c^i\}\), where
    \(u_c^i\) will be of size \(l_i\).
- Then shard \(C_k\) sends \(U_c^{(k)}\) to \(P_k\) (no security gained by shuffling here).

Step 1 (Encryption): Party P
- Let one coordinator for \(P\) generate the secret keys \(k_p, r_p, \ell_p \leftarrow \mathbb{Z}_q\) and sends these over a secure channel
to the it’s shards \(P_1, \ldots, P_m\).
- For the input set \(P\), the coordinator will shard it out to the \(P_1, \ldots, P_m\) servers sending partitions
  \(P^{(1)}, \ldots, P^{(m)}\) to each respectively.
- Then on shard \(P_k\), \(1 \leq k \leq m\), let \(U_p^{(k)}, E_p^{(k)} \leftarrow \emptyset\) and for each \(p_i \in P^{(k)}\)
  - compute \(u_p^i = H(p_i)^{k_p} = (H(p_i^1)^{k_p}, H(p_i^2)^{k_p} \ldots H(p_i^m)^{k_p})\), and let \(U_p^{(k)} = U_p^{(k)} \cup \{u_p^i\}\), where
    \(u_p^i\) will be of size \(m_i\).
- For each \(u_p^i \in U_p^{(k)}\) received form \(C_k\): Compute \(e_p^i = (u_p^i)^{k_p}\) and let \(E_p^{(k)} = E_p^{(k)} \cup \{e_p^i\}\)
- Perform a Cross Shard Shuffle of the sets \(E_c\) and \(U_p\) while sending to the shards of \(C\). The sending
  shards store how to reverse this Cross Shard Shuffle later.

Figure 7: Sharded multi-key Private-ID
Step 2 (Calculate set difference and waterfall matching): Party C

**Pre-Match Processing**
- On shared $C_k$, $1 \leq k \leq m$ which received $E_c^{(k)}$ and $U_p^{(k)}$ from $P_k$ let $E_p^{(k)}, V_c^{(k)}, V_p^{(k)} \leftarrow \emptyset$
  - Compute $E_p^{(k)} = \{(u'_p)^{k_e} : u'_p \in U_p^{(k)}\}$
  - Pad each tuple $c_i^p \in E_c^{(k)}$ with random looking strings such that the size of all $c_i^p$ is $\text{max\_num\_keys}$.
  - Tag each row of $E_c$ and $E_p$ with the original Party C shard number and an index giving the original order of that row on the shard, $(\text{shard\_index}, \text{order\_index})$.

**Staged Matching.**
For $j$ in 1 to $\max_{i \in \{1, 2, \ldots, n\}} l_i$. Stage $j$:
- Shard the sets $E_c$ and $E_p$ from all shards $C_1, \ldots, C_m$ to all shards $C_1, \ldots, C_m$ based on the values of the elements in position $j$, $E_c[j]$ and $E_p[j]$ (sending the full rows).
- On Shard $k$
  - Denote the subsets of $E_c$ and $E_p$ that are on this shard as $E_c^{(k)}$ and $E_p^{(k)}$ (note these will contain different partitions of the sets $E_c$ and $E_p$ than was earlier denoted by $E_c^{(k)}$ and $E_p^{(k)}$).
  - For each $c_i^p \in E_p^{(k)}$ check if $c_i^p[j]$ is in $E_c^{(k)}[j]$. (Note that because of the simplified match logic and the restriction of no repeats in the input from either party no many(C):one(P) matches can occur here, which is different from the version without simplified match logic).
  - Denote a match as $(c_i^p, e_i^c, j)$ where $e_i^c[j] = c_i^p[j]$.
  - As each match is found, we will remove the matched rows from $E_c^{(k)}$ and $E_p^{(k)}$ and create the corresponding output rows to add to the sets $V_c, V_p$. For an update $(c_i^p, e_i^c, j)$ compute:
    * $V_c^{(k)} = V_c^{(k)} \cup \{(c_i^c[j])^{r \ast c}, \text{shard\_index}, \text{order\_index}\}$
    * $V_p^{(k)} = V_p^{(k)} \cup \{(e_i^p[j])^{r \ast c}, \text{shard\_index}, \text{order\_index}\}$
    * $E_c^{(k)} = E_c^{(k)} \setminus e_i^c$ (drop row entirely, not just set to false)
    * $E_p^{(k)} = E_p^{(k)} \setminus e_i^p$
  - Then re-shard the sets $E_c^{(k)}$ and $E_p^{(k)}$ sending out according to the value of $E_c^{(k)}[j+1] \mod m$ or $E_p^{(k)}[j] + 1 \mod m$. Leave the rows of $V_c^{(k)}$ and $V_p^{(k)}$ on the shard that created them.

**Important Notes:**
- The key idea in the matching is that we don’t need to create a row of $V_c, V_p$ until the corresponding row of $E_c, E_p$ is set to be dropped. Also, we don’t need to create the sets $S_c, S_p$ until all matches have been found as they just end up being what is left over unmatched of $E_c, E_p$.
- Once a row of $V_c$ has been created and the corresponding row of $E_c$ been dropped, no more updates will come to this row, so it can remain on the shared where it was created until all the stages are finished (no need to re-shard these sets). Similarly for $V_p$ and $E_p$.

**End of stage $j$.**

**Post Match Processing**
- Once all stages have finished, create $S_c, S_p$ from what is left of $E_c$ and $E_p$. On shard $k$ let
  - $S_c^{(k)} = \{(c_i^c[1], \text{shard\_index}, \text{order\_index}) : c_i^c \in E_c^{(k)}\}$
  - $S_p^{(k)} = \{(e_i^p[1], \text{shard\_index}, \text{order\_index}) : e_i^p \in E_p^{(k)}\}$
- Also add what is left of $E_c$ and $E_p$ to $V_c$ and $V_p$. On shard $k$ let
  - $V_c^{(k)} = V_c^{(k)} \cup \{(c_i^c[1])^{r \ast c}, \text{shard\_index}, \text{order\_index}\} : c_i^c \in E_c^{(k)}\}$
  - $V_p^{(k)} = V_p^{(k)} \cup \{(e_i^p[1])^{r \ast c}, \text{shard\_index}, \text{order\_index}\} : e_i^p \in E_p^{(k)}\}$
- Send back the sets $S_c, S_p, V_c, V_p$ to the original shards which received them from $P$.
- Distributed across all the shards compute $S'_{\text{pid\_multi\_key}} = \{ (s_i^c)^{r \ast} : s_i^c \in S_c \}$.
- Shards $C_k$ send back $V_c^{(k)}$ and $V_p^{(k)}$ without shuffling and according to the same order in which the corresponding rows of $E_c$ and $E_p$ were received at the end of Step 1. Party C performs a cross shard shuffle in sending $S_c, S_p$ to $P$. 
Shared $\Pi^{PID-\text{multi-key}}$ continued

Step 3 (Output mapping): Party $P$

- The cross shard shuffle applied at the end of Step 1 when sending the sets $E_c$ and $U_p$ is reversed when Party $C$ receives the sets $V_c$ and $V_p$. Rows get passed back to the the original sending shards.
- On shard $k$ which received $S^{(k)}_c, S^{(k)}_p, V^{(k)}_c, V^{(k)}_p$ from $C_k$ let $W^{(k)}_c, W^{(k)}_p, S''^{(k)}_c, S''^{(k)}_p \leftarrow \emptyset$.
- Create $UID_P$:
  - For every $s^i_p \in V^{(k)}_p$, let $W^{(k)}_p = W^{(k)}_p \cup \{(v^i_p)^r_p\}$, and $M^{(k)}_p[(v^i_p)^r_p] = p_i$
  - For each $s^i_s \in S^{(k)}_c$, let $S''^{(k)}_c = S''^{(k)}_c \cup \{(s^i_s)^r_s\}$ and $M^{(k)}_c[(s^i_s)^r_s] = \perp$
  - Output $UID^{(k)}_p = W^{(k)}_p \cup S''^{(k)}_c$ and $M^{(k)}_c$ where
    \[
    M^{(k)}_c = \{(uid, id) : uid \in UID^{(k)}_p, id \in P \cup \{\perp\}\}.
    \]
    This set of UIDs should either be sorted by or the set $S''^{(k)}_c$ should be shuffled internally before revealing this output publicly or to Party $C$. See (*) for why we need to add this shuffle.
- For each $s^i_p \in S''^{(k)}_p$, let $S^{(k)}_p = S^{(k)}_p \cup \{(s^i_p)^r_p\}$
- For each $v^i_s \in V^{(k)}_s$, let $W^{(k)}_s = W^{(k)}_s \cup \{(v^i_s)^r_s\}$.
- Perform a cross shard shuffle of $S''^{(k)}_p$ and send to $C$; also send $W^{(k)}_s$ without shuffling. See (***) for why we need to add this shuffle of $S''^{(k)}_p$.

Step 3 (Output mapping): Party $C$

- On shard $k$ which received $S^{(k)}_p$ and $W^{(k)}_s$ from $P_k$, let $W^{(k)}_c, S''^{(k)}_c \leftarrow \emptyset$
- For every $w^i_c \in W^{(k)}_c$ let $W^{(k)}_c = W^{(k)}_c \cup \{(w^i_c)^r_c\}$
- $M^{(k)}_c[w^i_c] = c_i$ for $w^i_c \in W^{(k)}_c$.
- For every $s^{i_1} \in S''^{(k)}_p$ let $S''^{(k)}_c = S''^{(k)}_c \cup \{(s^{i_1})^r_c\}$ and $M^{(k)}_c[(s^{i_1})^r_c] = \perp$
- Output $UID^{(k)}_c = W^{(k)}_c \cup S''^{(k)}_c$ and $M^{(k)}_c$ where
  \[
  M^{(k)}_c = \{(uid, id) : uid \in UID^{(k)}_c, id \in C \cup \{\perp\}\}.
  \]

Notes on new shuffles:

- (*) Since Party $C$ shuffled $S''^{(k)}_c$, if the output is revealed to Party $C$ then they can know from having observed the matching if there was a row of $S''^{(k)}_p$ which as an earlier row of $E_c$ could have been matched with a different set of match rules or random order of matching. Thus, this UID is known to Party $C$ to be in the intersection for some definition of the intersection.
- (***) Without the added shuffle of $S''^{(k)}_p$ Party $C$ can know from having observed the matching if there was a row of $S''^{(k)}_p$ which as an earlier row of $E_p$ could have matched with different match rules or random order of matching. Thus, this resulting UID is known to Party $C$ to be in the intersection for some definition of the intersection.

Figure 9: Sharded multi-key Private-ID, continued

7 Experiments

In this section, we present the details of our implementation of the proposed multi-key Private-ID protocol and report performance in terms of wall clock time, and network traffic volume. We first establish the extra cost of leveraging multi-key Private-ID protocol relative to single-key Private-ID protocol in the case when there is at most one identifier to perform the matching. We also assess the performance of the protocols using additional synthetic datasets created assuming a spectrum
of underlying dataset parameters such as number of input records, intersection size, and number of identifiers per record to demonstrate the scalability.

7.1 Implementation

We implemented the protocol in Rust and it is available at the Private-ID Github repository [4]. We chose the Rust language for its superior memory management and the ease of multi-threading while enforcing safety. We use the Dalek [1] library for Elliptic Curve Cryptography which utilizes Ristretto [2] technique for Curve25519. The performance measurements were carried out on AWS m5.12xlarge (Intel Xeon Scalable Processors with all-core CPU frequency of 3.1GHz, 48 vCPU, 192GB RAM) EC2 instances. To simulate client and server parties we leverage two separate AWS EC2 instances in the same region and availability zone.

7.2 Varying input size

In this test, we vary the number of records in each party’s synthetic dataset to show how the protocol scales with input size. Both parties have the same number of records and each record has one identifier and the intersection size is 50%. We find that the multi-key protocol is roughly three times slower than the single-key protocol irrespective of the input size. Both wall clock run time and network traffic grew roughly linearly with respect to input size (see Figure 10)

<table>
<thead>
<tr>
<th>Input size</th>
<th>Single-key</th>
<th>Multi-key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time(s)</td>
<td>In/Out [MB]</td>
<td>Time(s)</td>
</tr>
<tr>
<td>$10^4$</td>
<td>0.6</td>
<td>1.0/1.2</td>
</tr>
<tr>
<td>$10^5$</td>
<td>4.7</td>
<td>10/12</td>
</tr>
<tr>
<td>$10^6$</td>
<td>47</td>
<td>102/119</td>
</tr>
<tr>
<td>$10^7$</td>
<td>491</td>
<td>1024/1193</td>
</tr>
</tbody>
</table>

* Data obtained from network communication is measured by `/proc/net/dev` stats. In/Out communication is shown for one party $P$, since it is the same for both parties. Input size is the number of records that $C$ and $P$ each have.

Table 1: Performance for varying input sizes

7.3 Varying intersection size

We now fix the input size at 1 million records for both parties and the number of identifiers at 2 per record. We vary the intersection size which is the number of records that will match across parties. We see that wall clock time increases with intersection size, but network traffic decreases. We attribute the reduction in network traffic to the records present in the intersection that need not be transported across parties in the later steps of the protocol. On the other hand, compute grows with intersection size due to the waterfall rounds from the match logic.
Figure 10: Run time with number of records

<table>
<thead>
<tr>
<th>Intersection size</th>
<th>Multi-key</th>
<th>Time(s)</th>
<th>In/Out [MB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1%</td>
<td></td>
<td>69</td>
<td>180/157</td>
</tr>
<tr>
<td>25%</td>
<td></td>
<td>107</td>
<td>164/149</td>
</tr>
<tr>
<td>50%</td>
<td></td>
<td>146</td>
<td>147/140</td>
</tr>
<tr>
<td>75%</td>
<td></td>
<td>185</td>
<td>130/131</td>
</tr>
<tr>
<td>100%</td>
<td></td>
<td>224</td>
<td>113/122</td>
</tr>
</tbody>
</table>

* Data obtained from network communication is measured by /proc/net/dev stats. In/Out communication is shown for one party P, since it is the same for both parties. Input size is the number of records that C and P each have.

Table 2: Performance for varying intersection sizes

7.4 Varying number of identifiers

Finally, with a fixed the input size of 1 million records, intersection size at 50% and identifier size at 10 characters, we increase the number of identifiers per record and we see that both time and network I/O scales with the number of identifiers (Figure 11(b)).
Figure 11: Run time and total network traffic of multi-key PID protocol

<table>
<thead>
<tr>
<th>identifiers</th>
<th>Multi-key Time(s)</th>
<th>In/Out [MB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>146</td>
<td>147/140</td>
</tr>
<tr>
<td>3</td>
<td>217</td>
<td>216/277</td>
</tr>
<tr>
<td>5</td>
<td>299</td>
<td>284/413</td>
</tr>
<tr>
<td>7</td>
<td>410</td>
<td>354/550</td>
</tr>
<tr>
<td>9</td>
<td>516</td>
<td>421/685</td>
</tr>
</tbody>
</table>

* Data obtained from network communication is measured by `/proc/net/dev` stats. In/Out communication is shown for one party $P$, since it is the same for both parties. Input size is the number of records that $C$ and $P$ each have.

Table 3: Performance for varying number of identifiers record

References


