Multi-key Private Matching for Compute

Prasad Buddhavarapu Benjamin M Case Logan Gore Andrew Knox
Payman Mohassel Shubho Sengupta Erik Taubeneck Min Xue
Facebook Inc.

8th June 2021

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.

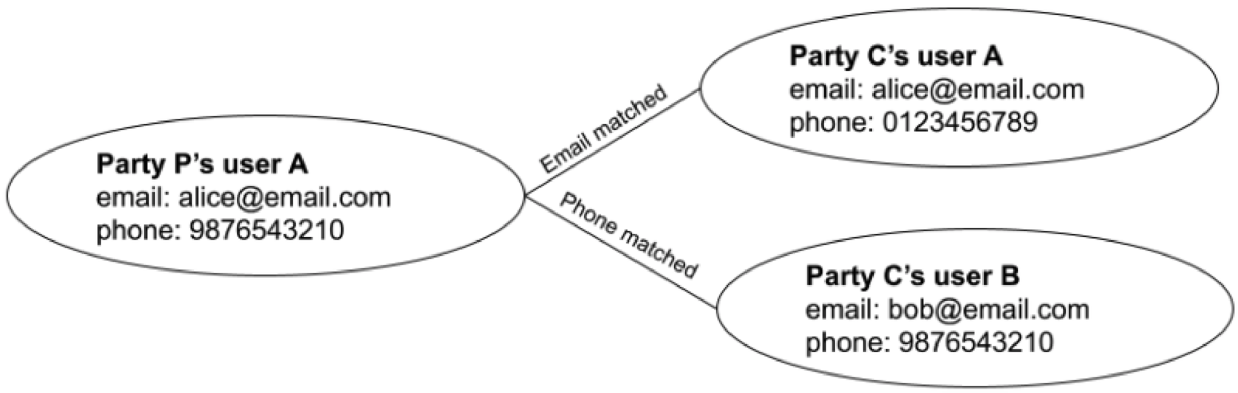


Figure 1: An example of many-to-many connections with multiple identifiers

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

$\mathcal{C}$ and $\mathcal{P}$ 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,

$$\mathcal{C} : \{c_i : (c_i^1, c_i^2, \dots, c_i^{l_i}), i \in (1, 2, \dots n_c)\}$$

$$\mathcal{P} : \{p_i : (p_i^1, p_i^2, \dots, p_i^{m_i}), i \in (1, 2, \dots n_p)\}$$

Each array $(c_i^1, c_i^2, \dots, c_i^{l_i})$ 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_i^j in defining a connection with party P's record containing c_i^j.

The protocol outputs a set of Universal Identifiers (UID), denoted by $UID = \{uid_1, \dots, uid_{|UID|}\}$ 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 $c_i \in \mathcal{C}$ and $M_c[uid_i] = \perp$ otherwise. Similarly, party P learns a map M_p, where $M_p[uid_i] = p_i$ if $p_i \in \mathcal{P}$ 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 $\mathcal{C}$ and $\mathcal{P}$.

4.2 Protocol Overview

Step 1, Exchange records. Our starting point is a DDH based scheme [8, 10, 5, 9, 3].

In Step ❶ of the protocol, (Step 1 in Figures 2 and 3) party C hashes its records componentwise $\mathbf{c}_i = (c_i^1, c_i^2, ..., c_i^{l_i})$ as $H(\mathbf{c}_i) = (H(c_i^1), H(c_i^2), ..., H(c_i^{l_i}))$ and exponentiates each component using a random secret scalar k_c, $H(\mathbf{c}_i)^{k_c}$. Party P also computes $H(\mathbf{p}_j)^{k_p}$ 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 $U_c = \{H(\mathbf{c}_i)^{k_c}\}$ and $U_p = \{H(\mathbf{p}_j)^{k_p}\}$.

In Step ❶, party C also computes $H(\mathbf{p}_j)^{k_p k_c}$ and similarly party P computes $H(\mathbf{c}_i)^{k_c k_p}$. 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 $H(c_i^\alpha)^{k_p k_c r_c r_p}$ or $H(p_i^\alpha)^{k_p k_c r_c r_p}$, 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 ❷ 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

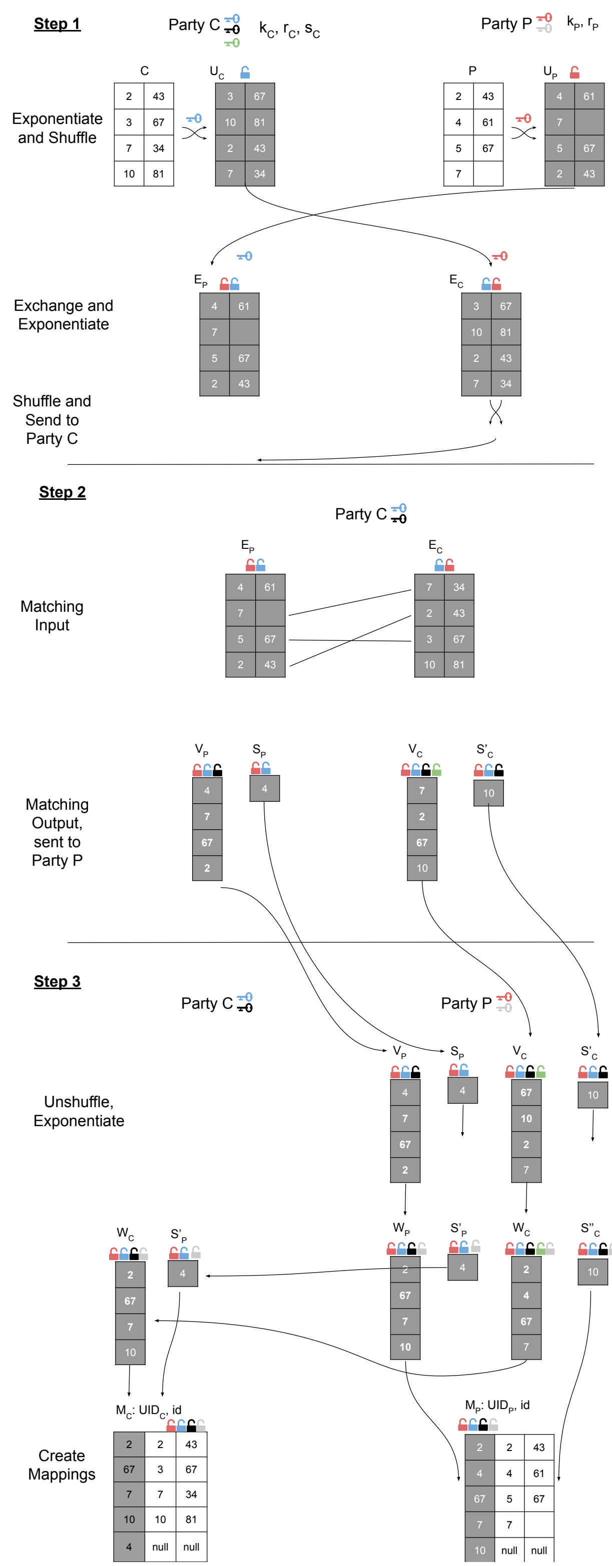


Figure 2: Protocol Diagram

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 ❸, party P can unshuffle V_p to match the original order of set $\mathcal{P}$. 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

$$\Pi^{\mathsf{PID-multi-key}}$$

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

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

Outputs:
$[C : (UID, M_c), P : (UID, M_p)]$

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

Step 1 (Exchange records): Party C

- Let $k_c, r_c, s_c \xleftarrow{R} \mathbb{Z}_q$, and $U_c \leftarrow \emptyset$.
- For each $c_i \in \mathcal{C}$ compute $u_c^i = H(c_i)^{k_c} = (H(c_i^1)^{k_c}, H(c_i^2)^{k_c} \ldots H(c_i^{l_i})^{k_c})$, and let $U_c = U_c \cup \{u_c^i\}$, where u_c^i will be of size l_i.
- Randomly (outer) shuffle the elements in U_c using a permutation $\pi_{U_c}^c$, however, NOT inner shuffling elements within c_i, and send to P.

Step 1 (Exchange records): Party P

- Let $k_p, r_p \xleftarrow{R} \mathbb{Z}_q$, and $U_p, E_c \leftarrow \emptyset$
- For each $p_i \in \mathcal{P}$ compute $u_p^i = H(p_i)^{k_p} = (H(p_i^1)^{k_p}, H(p_i^2)^{k_p} \ldots H(p_i^{l_i})^{k_p})$, and let $U_p = U_p \cup \{u_p^i\}$, where u_p^i will be of size m_i.
- Randomly shuffle the elements in U_p using a permutation $\pi_{U_p}^p$, and inner shuffling elements within p_i as well (no need to store inner shuffle).
- For each $u_c^i \in U_c$ received form C:
  - Compute $e_c^i = (u_c^i)^{k_p}$ and let $E_c = E_c \cup \{e_c^i\}$
- Randomly shuffle the elements in E_c using a permutation $\pi_{E_c}^p$, and send the sets E_c, U_p to C

Step 2 (Matching): Party C

- Let $E_p, V_c, V_p, S_c, S_c', S_p \leftarrow \emptyset$
- For each $u_p^i \in U_p$:
  - Compute $e_p^i = (u_p^i)^{k_c}$ and let $E_p = E_p \cup \{e_p^i\}$
- Initialize
  $S_c = \{e_c^i[1] : i \in (1, 2, \ldots n_c)\}$
  $S_p = \{e_p^i[1] : i \in (1, 2, \ldots n_p)\}$
  $V_c = \{(e_c^i[1])^{r_c s_c} : i \in (1, 2, \ldots n_c)\}$
  $V_p = \{(e_p^i[1])^{r_c} : i \in (1, 2, \ldots n_p)\}$

Figure 3: Private-ID multi-key matching with key priority

$\prod^{\mathsf{PID-multi-key}}$ continued

- Pad each tuple e_c^i of E_c with random unique looking strings such that size of e_c^i, $\forall$ i, is $\max_{i\in(1,2,\ldots n_c)} l_i$
- From padded E_c, prepare sets $E_c[1], E_c[2], \ldots E_c[\max_{i\in(1,2,\ldots n_c)} l_i]$, where $E_c[j] = \{e_c^i[j] : i \in (1,2,\ldots n_c)\}$, where $j \in (1,2,..\max_{i\in(1,2,\ldots n_c)} l_i)$
  - Iterate over each j in $1 : \max_{i\in(1,2,\ldots n_c)} l_i$
    - Iterate over each $e_p^i \in E_p$, which is a vector of size m_i:

      When Matched: If at least one element of e_p^i is within $E_c[j]$, then
      - Denote the matched elements, i.e. intersection of $E_c[j]$ and e_p^i as $match_{cp}^{ij}$ corresponding to the j^{th} batch of matching, and i^{th} element of set E_p.
      - Also denote, the matched indices within set $E_c[j]$ as $match\_loc_{E_c}^{ij}$
      - In the case of many(C):1(P) mapping (i.e. size of $match\_loc_{E_c}^{ij} > 1$), pick first element of $match_{cp}^{ij}$
      - Denote first matched position corresponding to the first matched element, i.e. $match\_loc_{E_c}^{ij}[1]$ as φ.
      - Update the sets
        · $S_c = S_c \setminus e_c^{\varphi}[1]$.
        · $S_p = S_p \setminus e_p^i[1]$
        · Assign the first matched identifier to the φ^{th} element of V_c.
        i.e $V_c[\varphi] = (match_{cp}^{ij}[1])^{r_c s_c}$
        · Assign the first matched identifier to the i^{th} element of V_p.
        i.e $V_p[i] = (match_{cp}^{ij}[1])^{r_c}$
      - Drop the matched elements from E_c and E_p sets, and perform padding using random strings to retain the appropriate index. In practice, padding with random strings may not be necessary, by keeping track of appropriate indices.
        · Replace $e_c^{\varphi}[k]\ \forall\ k > j$ with a random looking unique string.
        · Replace e_p^i with a random looking string (vector of size 1).

      When **unmatched**: i.e. if at least one element of e_p^i is NOT within $E_c[j]$, then
      - Do nothing, and proceed with i++.
- For each $s_c^i \in S_c$, let $S_c' = S_c' \cup \{(s_c^i)^{r_c}\}$, and randomly shuffle elements in S_p.
- Send the sets S_c', S_p, V_c, V_p to P.

Step 3 (Output mapping): P

- Let $W_c, W_p, S_c'', S_p' = \emptyset$
- Create UID_P:
  - Shuffle back the elements of V_p using ${\pi_{U_p}^{p}}^{-1}$. For every $v_p^i \in V_p$, let $W_p = W_p \cup \{(v_p^i)^{r_p}\}$, and $M_p[(v_p^i)^{r_p}] = p_i$
  - For each $s_c^i \in S_c'$, let $S_c'' = S_c'' \cup \{(s_c^i)^{r_p}\}$ and $M_p[(s_c^i)^{r_p}] = \perp$
  - Output $UID_p = W_p \cup S_c''$ and M_p
- For each $s_p^i \in S_p$, let $S_p' = S_p' \cup \{(s_p^i)^{r_p}\}$
- Shuffle back the elements of V_c using ${\pi_{E_c}^{p}}^{-1}$. For each $v_c^i \in V_c$, let $W_c = W_c \cup \{(v_c^i)^{r_p}\}$.
- Send S_p', W_c to C

Step 3 (Output mapping): C

- Let $W_c', S_p'' = \emptyset$
- Shuffle back the elements of W_c using ${\pi_{U_c}^{c}}^{-1}$. For every $w_c^i \in W_c$ let $W_c' = W_c' \cup \{(w_c^i)^{s_c^{-1}}\}$
- $M_c[w_c'^i)] = c_i$
- For every ${s_p'}^i \in S_p'$ let $S_p'' = S_p'' \cup \{({s_p'}^i)^{r_c}\}$ and $M_c[({s_p'}^i)^{r_c}] = \perp$
- Output $UID_c = W_c' \cup S_p''$ 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, $\prod^{\mathsf{PID}}$

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 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 $\mathsf{REAL}^{a,\lambda}_{\prod^{\mathsf{PID}}}(\mathcal{C}, \mathcal{P})$ 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 $\prod^{\mathsf{PID}}$ against a semi-honest C)**.** *There exists a PPT simulator* SIM_c *such that for all security parameters* λ *and all inputs* $\mathcal{C} = \{c_1, \ldots, c_n\}$ *and* $\mathcal{P} = \{p_1, \ldots, p_m\}$*,*

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

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 SIM_c which we claim indistinguisably simulates the real view of party C.

Using a sequence of hybrid arguments, we show that the distribution generated by SIM_c is indeed indistinguishable from the real view of C.

Simulate C's step 1:

- Generate $k_c, r_c, s_c \xleftarrow{R} \mathbb{Z}_q$
- Honestly generate U_c, i.e. for each $\mathbf{c}_i \in \mathcal{C}$ compute $u_c^i = H(c_i)^{k_c}$ and let $U_c = U_c \cup \{u_c^i\}$.

Simulate P's step 1:

- For each $i \in [n]$ compute $g_i \xleftarrow{R} G$, and let $E_c = E_c \cup \{g_i^{k_c}\}$.
- Construct the set U_p to have the same structure of matches with E_c as $\mathcal{G}$, for matches components letting $u_{p,i}^j = e_{c,\gamma}^{\alpha}$ and for non-matches letting $u_{p,i}^j \xleftarrow{R} G$.
- 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: 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 SIM outputs V_p, S_c', S_p for P.

Simulate P's step 2:

- Let $J = m - \ell$, where $\ell = |\mathcal{C} \cap \mathcal{P}|$. 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: SIM_c does this step exactly as the protocol describes and using r_c and s_c it generated above.

Figure 5: Description of SIM_c for Theorem 1

$\mathcal{H}_0$: 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, α_i) range over the individual identifiers in $\mathcal{C}$ 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 $g_i^{k_c}$ 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, α_j) rand over the individual identifiers in $\mathcal{P}$ but not in $\mathcal{C}$. $\mathcal{H}_{2,j,\alpha_j-1}$ 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, α_t) range over the individual identifiers in $\mathcal{C}$ that also appear in $\mathcal{P}$. $\mathcal{H}_{3,t,\alpha_t}$ 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^*, α_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

$\mathcal{H}_6$: The view of C output by SIM_c.

We now need to argue that each consecutive pair of hybrids in the above sequence are indistinguisable by a PPT algorithm. The interesting arguments here are those for $(\mathcal{H}_{1,i,\alpha_i-1}, \mathcal{H}_{1,i,\alpha_i})$, $(\mathcal{H}_{2,j,\alpha_j-1}, \mathcal{H}_{2,j,\alpha_j})$, $(\mathcal{H}_{3,t,\alpha_t-1}, \mathcal{H}_{3,t,\alpha_t})$, $(\mathcal{H}_{4,i-1}, \mathcal{H}_{4,i})$ and $(\mathcal{H}_{5,i-1}, \mathcal{H}_{5,i})$. 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 $(\mathcal{H}_{1,i,\alpha_i-1}, \mathcal{H}_{1,i,\alpha_i})$ as an example. In particular, we argue that for any PPT adversary $\mathcal{A}$ who can distinguish the two hybrids, we devise an adversary $\mathcal{B}$ who can solve the DDH problem. $\mathcal{B}$ is given (g, g^a, g^b, g^c) and needs to decide whether c is random or $c = ab$. First note $\mathcal{B}$ can program $H(\cdot)$ to return g^b on input $c_i^{\alpha_i}$. We also let $g^a = g^{k_p}$. 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 $\mathcal{H}_{1,i-1}$ if $c = ab$ and is identically distributed to $\mathcal{H}_{1,i}$ if c is random (since g_i is uniformly random). If $\mathcal{A}$ can decide which hybrid it is interacting with, $\mathcal{B}$ can decide which DDH tuple it was given with the same probability. $\square$

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

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

where $\ell = |\mathcal{C} \cap \mathcal{P}|$*.*

proof sketch. The description of 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, 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 SIM_c. 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.

$\square$

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

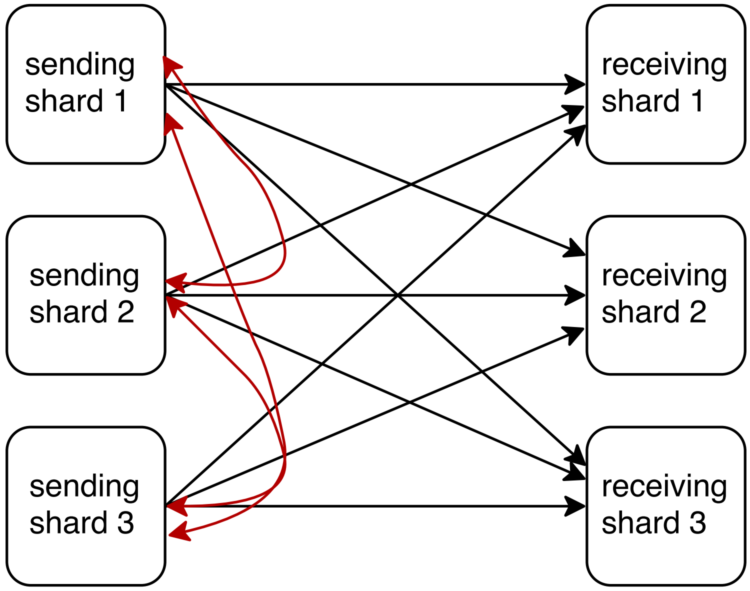


Figure 6: Cross Shard Shuffle with 3 shards

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.

Sharded $\prod^{\mathsf{PID-multi-key}}$

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

Party P: $\mathcal{P} = \{p_i : (p_i^1, p_i^2, \ldots, p_i^{m_i}), i \in (1, 2, \ldots n_p)\}$
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 $max\_num\_keys := \max_{i \in (1,2,\ldots n_c)} l_i = \max_{i \in (1,2,\ldots n_p)} m_i$.
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 $H(\cdot) : \{0, 1\}^* \to G$ modeled as a random oracle.

Step 1 (Encryption): Party C

- Let one coordinator for C generate the secret keys $k_c, r_c, s_c, \xleftarrow{R} \mathbb{Z}_q$ and then send these over a secure channel to the it's shards $C_1, \ldots, C_m$.
- For the input set $\mathcal{C}$, the coordinator will shard it out to the $C_1, \ldots, C_m$ servers sending partitions $\mathcal{C}^{(1)}, \ldots, \mathcal{C}^{(m)}$ to each respectively.
- Then on shard C_k, $1 \leq k \leq m$, let $U_c^{(k)} \leftarrow \emptyset$ and for each $c_i \in \mathcal{C}^{(k)}$
  - compute $u_c^i = H(c_i)^{k_c} = (H(c_i^1)^{k_c}, H(c_i^2)^{k_c} \ldots H(c_i^{l_i})^{k_c})$, 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 $k_p, r_p \xleftarrow{R} \mathbb{Z}_q$ and sends these over a secure channel to the it's shards $P_1, \ldots, P_m$.
- For the input set $\mathcal{P}$, the coordinator will shard it out to the $P_1, \ldots, P_m$ servers sending partitions $\mathcal{P}^{(1)}, \ldots, \mathcal{P}^{(m)}$ 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 $p_i \in \mathcal{P}^{(k)}$
  - compute $u_p^i = H(p_i)^{k_p} = (H(p_i^1)^{k_p}, H(p_i^2)^{k_p} \ldots H(p_i^{l_i})^{k_p})$, and let $U_p^{(k)} = U_p^{(k)} \cup \{u_p^i\}$, where u_p^i will be of size m_i.
- For each $u_c^i \in U_c^{(k)}$ received form C_k: Compute $e_c^i = (u_c^i)^{k_p}$ and let $E_c^{(k)} = E_c^{(k)} \cup \{e_c^i\}$
- Perform a Cross Shard Shuffle of the sets E_c and U_p while sending to the shards of $\mathcal{C}$. The sending shards store how to reverse this Cross Shard Shuffle later.

Figure 7: Sharded multi-key Private-ID

Sharded $\prod^{\mathsf{PID-multi-key}}$ continued

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 $E_p^{(k)} = \{(u_p^i)^{k_c} : u_p^i \in U_p^{(k)}\}$
  - Pad each tuple $e_c^i \in E_c^{(k)}$ with random looking strings such that the size of all e_c^i is 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, $(shard\_index, order\_index)$.

Staged Matching.

For j in 1 to $\max_{i\in(1,2,\ldots n_c)} l_i$. Stage j:

- Shard the sets E_c and E_p from all shards $C_1, ..., C_m$ to all shards $C_1, ..., C_m$ 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 $e_p^i \in E_p^{(k)}$ check if $e_p^i[j]$ 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 (e_p^i, e_c^γ, j) where $e_p^i[j] = e_c^\gamma[j]$.
  - 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 (e_p^i, e_c^γ, j) compute:
    * $V_c^{(k)} = V_c^{(k)} \cup \{(e_c^\gamma[j])^{r_c s_c}, shard\_index, order\_index)\}$
    * $V_p^{(k)} = V_p^{(k)} \cup \{(e_p^i[j])^{r_c}, shard\_index, order\_index)\}$
    * $E_c^{(k)} = E_c^{(k)} \setminus e_c^i$ (drop row entirely, not just set to false)
    * $E_p^{(k)} = E_p^{(k)} \setminus e_p^i$
  - Then re-shard the sets $E_c^{(k)}$ and $E_p^{(k)}$ sending out according to the value of $E_c^{(k)}[j+1] \pmod m$ or $E_p^{(k)}[j+1] \pmod m$. 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
  - $S_c^{(k)} = \{(e_c^i[1], shard\_index, order\_index), e_c^i \in E_c^{(k)}\}$
  - $S_p^{(k)} = \{(e_p^i[1], shard\_index, order\_index), e_p^i \in E_p^{(k)}\}$
- Also add what is left of E_c and E_p to V_c and V_p. On shard k let
  - $V_c^{(k)} = V_c^{(k)} \cup \{((e_c^i[1])^{r_c s_c}, shard\_index, order\_index) : e_c^i \in E_c^{(k)}\}$
  - $V_p^{(k)} = V_p^{(k)} \cup \{((e_p^i[1])^{r_c}, shard\_index, order\_index) : e_p^i \in E_p^{(k)}\}$
- 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 $S_c' = \{(s_c^i)^{r_c} : s_c^i \in S_c\}$.
- 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 $\prod^{\mathsf{PID-multi-key}}$ 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 $S_c^{(k)}, S_p^{(k)}, V_c^{(k)}, V_p^{(k)}$ from C_k let $W_c^{(k)}$, $W_p^{(k)}$, $S_c^{\prime\prime(k)}$ $S_p^{\prime(k)} \leftarrow \emptyset$.
- Create UID_P:
  - For every $v_p^i \in V_p^{(k)}$, let $W_p^{(k)} = W_p^{(k)} \cup \{(v_p^i)^{r_p}\}$, and $M_p^{(k)}[(v_p^i)^{r_p}] = p_i$
  - For each $s_c^i \in S_c^{\prime(k)}$, let $S_c^{\prime\prime(k)} = S_c^{\prime\prime(k)} \cup \{(s_c^i)^{r_p}\}$ and $M_p^{(k)}[(s_c^i)^{r_p}] = \perp$
  - Output $UID_p^{(k)} = W_p^{(k)} \cup S_c^{\prime\prime(k)}$ and $M_p^{(k)}$ where

$$M_p^{(k)} = \{(uid, id) : uid \in UID_p^{(k)}, id \in \mathcal{P} \cup \{\perp\}\}.$$

  - This set of UIDs should either be sorted by or the set $S_c^{\prime\prime(k)}$ should be shuffled internally before revealing this output publicly or to Party C. See (*) for why we need to add this shuffle.
- For each $s_p^i \in S_p^{(k)}$, let $S_p^{\prime(k)} = S_p^{\prime(k)} \cup \{(s_p^i)^{r_p}\}$
- For each $v_c^i \in V_c^{(k)}$, let $W_c^{(k)} = W_c^{(k)} \cup \{(v_c^i)^{r_p}\}$.
- Perform a cross shard shuffle of S_p' and send to C; also send W_c without shuffling. See (**) for why we need to add this shuffle of S_p'.

Step 3 (Output mapping): Party C

- On shard k which received $S_p^{\prime(k)}$ and $W_c^{(k)}$ from P_k, let $W_c^{\prime(k)}, S_p^{\prime\prime(k)} \leftarrow \emptyset$
- For every $w_c^i \in W_c^{(k)}$ let $W_c^{\prime(k)} = W_c^{\prime(k)} \cup \{(w_c^i)^{s_c^{-1}}\}$
- $M_c^{(k)}[w_c^{\prime i}] = c_i$ for $w_c^{\prime i} \in W_c^{\prime(k)}$.
- For every ${s_p'}^i \in S_p^{\prime(k)}$ let $S_p^{\prime\prime(k)} = S_p^{\prime\prime(k)} \cup \{({s_p'}^i)^{r_c}\}$ and $M_c^{(k)}[({s_p'}^i)^{r_c}] = \perp$
- Output $UID_c^{(k)} = W_c^{\prime(k)} \cup S_p^{\prime\prime(k)}$ and $M_c^{(k)}$ where

$$M_c^{(k)} = \{(uid, id) : uid \in UID_c^{(k)}, id \in \mathcal{C} \cup \{\perp\}\}.$$

Notes on new shuffles:

- (*) Since Party C shuffled $S_c^{\prime\prime(k)}$, if the output is revealed to Party C then they can know from having observed the matching if there was a row of S_c'' 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 $S_p^{\prime(k)}$ Party C can know from having observed the matching if there was a row of $S_p^{\prime(k)}$ 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 of input size. Both wall clock run time and network traffic grew roughly linearly with respect to input size (see Figure 10)

Input size	**Single-key**		**Multi-key**	
	Time(s)	**In/Out [MB]**	**Time(s)**	**In/Out[MB]**
10^4	0.6	1.0/1.2	1.8	1.5/1.4
10^5	4.7	10/12	14	15/14
10^6	47	102/119	146	147/140
10^7	491	1024/1193	1501	1462/1395

* 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.

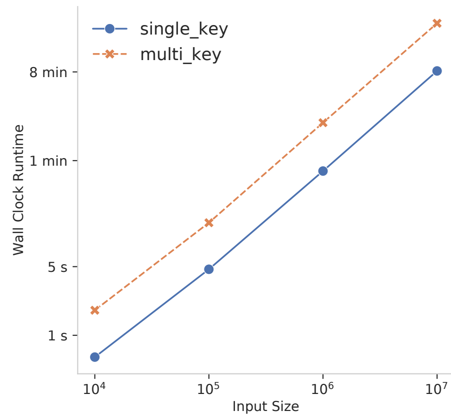


Figure 10: Run time with number of records

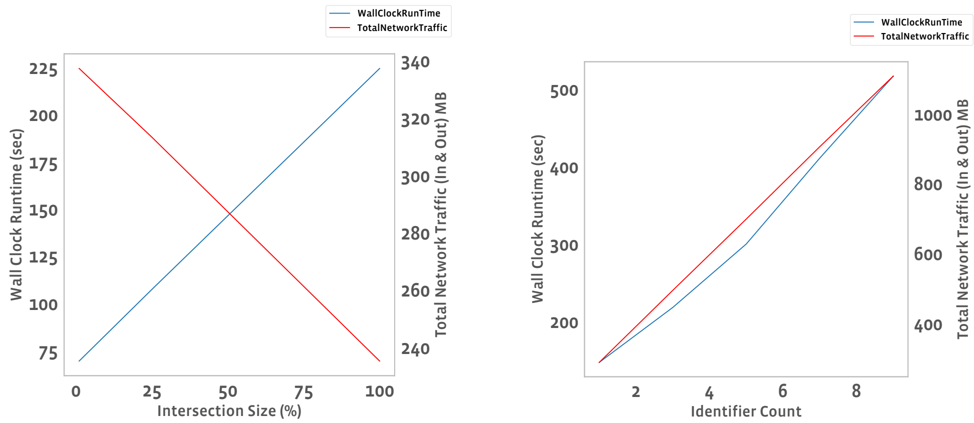
Intersection size	**Multi-key**	
	Time(s)	**In/Out** [MB]
1%	69	180/157
25%	107	164/149
50%	146	147/140
75%	185	130/131
100%	224	113/122

* 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)).



(a) Varying intersection size of datasets (b) Varying per record identifier count

Figure 11: Run time and total network traffic of multi-key PID protocol

identifiers	Multi-key	
	Time(s)	**In/Out [MB]**
1	146	147/140
3	217	216/277
5	299	284/413
7	410	354/550
9	516	421/685

* 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

[1] Dalek library for elliptic curve cryptography. `https://github.com/dalek-cryptography/curve25519-dalek`, May 2020.

[2] Ristretto. `https://ristretto.group/`, May 2020.

[3] Prasad Buddhavarapu, Andrew Knox, Payman Mohassel, Shubho Sengupta, Erik Taubeneck, and Vlad Vlaskin. Private matching for compute. Cryptology ePrint Archive, Report 2020/599, 2020. `https://eprint.iacr.org/2020/599`.

[4] Prasad Buddhavarapu, Andrew Knox, Payman Mohassel, Shubho Sengupta, Erik Taubeneck, and Vlad Vlaskin. Privateid. `https://github.com/facebookresearch/Private-ID`, 2020.

[5] Emiliano De Cristofaro, Jihye Kim, and Gene Tsudik. Linear-complexity private set intersection protocols secure in malicious model. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 213–231. Springer, 2010.

[6] Michael J Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *International conference on the theory and applications of cryptographic techniques*, pages 1–19. Springer, 2004.

[7] Gayathri Garimella, Payman Mohassel, Mike Rosulek, Saeed Sadeghian, and Jaspal Singh. Private set operations from oblivious switching. Cryptology ePrint Archive, Report 2021/243, 2021. https://eprint.iacr.org/2021/243.

[8] Bernardo A Huberman, Matt Franklin, and Tad Hogg. Enhancing privacy and trust in electronic communities. In *Proceedings of the 1st ACM conference on Electronic commerce*, pages 78–86. ACM, 1999.

[9] Mihaela Ion, Ben Kreuter, Ahmet Erhan Nergiz, Sarvar Patel, Mariana Raykova, Shobhit Saxena, Karn Seth, David Shanahan, and Moti Yung. On deploying secure computing commercially: Private intersection-sum protocols and their business applications. Cryptology ePrint Archive, Report 2019/723, 2019. https://eprint.iacr.org/2019/723.

[10] Stanisław Jarecki and Xiaomin Liu. Fast secure computation of set intersection. In *International Conference on Security and Cryptography for Networks*, pages 418–435. Springer, 2010.

[11] Lea Kissner and Dawn Song. Privacy-preserving set operations. In *Annual International Cryptology Conference*, pages 241–257. Springer, 2005.

[12] Ben Kreuter, Sarvar Patel, and Ben Terner. Private identity agreement for private set functionalities. Cryptology ePrint Archive, Report 2020/620, 2020. https://eprint.iacr.org/2020/620.

[13] Catherine Meadows. A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In *1986 IEEE Symposium on Security and Privacy*, pages 134–134. IEEE, 1986.

[14] Mahnush Movahedi, Benjamin M. Case, Andrew Knox, Li Li, Yiming Paul Li, Sanjay Saravanan, Shubho Sengupta, and Erik Taubeneck. Private randomized controlled trials: A protocol for industry scale deployment. *CoRR*, abs/2101.04766, 2021.