Multi-Party Private Join

Anja Lehmann
Hasso Plattner Institute,
University of Potsdam
anja.lehmann@hpi.de

Christian Mouchet
Hasso Plattner Institute,
University of Potsdam
christian.mouchet@hpi.de

Andrey Sidorenko
Hasso Plattner Institute,
University of Potsdam
andrey.sidorenko@hpi.de

Abstract

A multi-party private join (MPPJ) protocol enables multiple *source* parties to provide a *receiver* party with the inner joins over their respective datasets, while revealing as little information as possible. There is currently no protocol that directly and efficiently enables such a MPPJ beyond the two- or three-party setting. The presently known protocols either achieve weaker functionality (e.g., multi-party private set intersection protocols) or more general ones (e.g., private-join-compute and generic secure multi-party computation protocols) and are therefore more costly to run for the sources.

This work formally introduces MPPJ as an explicit goal, and proposes an efficient, helper-assisted protocol that achieves n-party inner joins with small leakage and close-to-optimal overhead for the sources. Specifically, for n databases with m rows, it requires only a single $O(m)$ upload from the sources to the helper, and a single $O(n \cdot m)$ download from the helper to the receiver. Moreover, the helper is entirely oblivious: it enables the efficiency and simplicity goals we are striving for, but it does not learn anything about the computation it facilitates. We formally model and prove the security of our protocol from standard assumptions, in the passive-adversary model. Then, we provide an open-source implementation and an extensive performance evaluation. According to our experiments, our protocol requires 1.02 to 20 times less communication than a current private-join-compute protocol (with no computation over the join) for 2 to 6 parties and input database sizes from 1.5K to 250K records. Finally, we demonstrate the versatility of our approach by extending our protocol to *threshold*-joins.

Keywords

multi-party computation, private join, privacy-preserving compute

1 Introduction

Personal data is often distributed among several databases and entities, e.g., in the social security or healthcare context. In the latter, health information of citizens is typically stored in dozens of databases by their practitioners, medical institutions, and insurance companies. While this data, when associated with the individual person, is highly sensitive and must enjoy strong protection, there is also the increasing desire to leverage the combined information for research purposes. For instance, the European Health Data Space (EHDS) [1] aims to "*enable the secure and trustworthy reuse [of] health data for research, innovation, policy-making, and regulatory activities (secondary use of data)*". By combining previously siloed datasets, these data spaces aim to unlock underutilized health information and reveal insights otherwise hidden due to fragmentation.

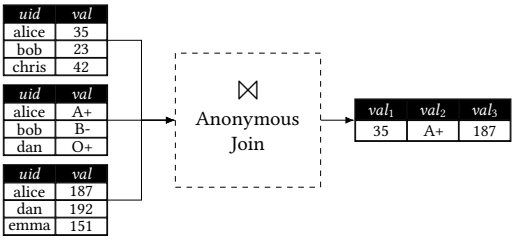

Figure 1: Example of the anonymous join functionality.

Anonymous Join. Given the highly sensitive nature of health information, it is essential for such data spaces to enforce strict data-minimization practices. A crucial primitive in this regard is an *anonymous join*. Such a procedure combines n associative tables, each mapping a set of identifiers to an associated value, into a single table containing the values for which the associated identifier is present in all input tables. Importantly, the output table is aligned, (i.e., values with matching identifiers are associated), but also *anonymous*[1], i.e., it does not contain the identifiers. Figure 1 illustrates an anonymous join functionality.

We are interested in the problem of computing anonymous joins among n distributed *sources* (e.g., medical institutions) that want to provide the anonymous join over their local datasets towards a *receiver*, e.g., a research institute. Therein, the receiver should not learn the values *outside* of the join, as they would typically be unrelated to the study or application. Further, the join should not reveal the unique identifiers, as this would violate the goal of *anonymous* joins. And, finally, the sources should also not learn anything about the other sources' datasets.

Privacy and Simplicity Challenges. Such multi-party anonymous join can be computed through *general-purpose secure multi-party computation* (MPC) protocols. However, while this solution provides optimal privacy, it suffers from impractical costs already for modest database sizes (we demonstrate this in Section 5.3). Further, apart from the protocol costs, it also requires the sources to be able to execute complex and interactive cryptographic protocols. This is an unrealistic assumption for many use cases, in particular in historically less digitalized sectors such as healthcare.

Helper-based settings, where the join is facilitated by a central *helper*, enable easier integration into existing infrastructures. For

Proceedings on Privacy Enhancing Technologies 2026(2), 39–57

https://doi.org/10.56553/popets-2026-0035

[1]While we refer to joined data without primary identifiers as "anonymous", this indeed does not result in completely anonymized data, as the revealed values inherently carry the risk of re-identification through correlations and prior knowledge [30].

instance, the Research Data Center (Forschungsdatenzentrum) [2] in Germany, which will make joined data of distributed health records available to research institutions, follows that design. While a trusted helper can easily integrate the datasets from the sources, naive implementations suffer from poor privacy: the helper typically sees all the data and becomes a privacy risk itself.

The main endeavor of our work is to overcome the seemingly contradicting goals regarding simplicity and data-minimization: we aim to keep the simplicity of such helper-based anonymous joins, but also protect the privacy towards the central helper. Thus, for simplicity, sources should merely have to upload their data for a particular research purpose and a dedicated receiver. For privacy, the helper should not learn any sensitive information about the data for which it performs the join.

State-of-the-Art. Interestingly, the problem of privately computing multi-party joins has not yet seen much attention in the literature, and even less so in its helper-assisted and anonymous formulation. Semantically, this problem is related to *multi-party private set intersection* (MPSI), for which the body of work is extensive [32]. But MPSI alone is not sufficient to realize MPPJ as it lacks the support for associated values. While PSI variants such as *labeled PSI* [10, 12] or *PSI with associated data* (PSI-AD) [7] aim at bridging this gap, they are not applicable to our setting because they are restricted to the two- and three-party settings, and they reveal the intersecting identifiers to the receiver. Moreover, although it might be appealing to use MPSI *as a building block* to construct MPPJ, doing so is not straightforward (we discuss this in Section 3).

Related works closer to MPPJ are so-called *private-join-compute* (PJC) protocols [19, 24, 28, 29], where further secure computation is performed over the join. While performing data analysis within the secure computation seems desirable, this is not possible for many cases that intrinsically require access to the raw values. Further, interestingly, not all of them can be retro-fitted into *plain* private join ones (e.g., when the security is itself based on the additional computation [29]) and, when possible, this transformation always results in additional costs, as the original protocol produces secret-shared outputs (typically, a reconstruction round is then required, e.g., for [25]). We provide a more extensive discussion of the related works on MPSI and PJC in Section 3. To the best of our knowledge, no practical protocol for standalone n-party anonymous join has been proposed so far.

1.1 Our Contributions

In this work, we initiate the study of standalone protocols for *multi-party private join* (MPPJ), and propose a first helper-assisted construction. Our protocol enables n sources $\mathcal{S}_1, ..., \mathcal{S}_n$, each holding a database table, to provide a receiver $\mathcal{R}$ with the anonymous join over their tables, by relying on the support of a helper party $\mathcal{H}$, and while guaranteeing the following properties:

Source Privacy: The sources $\mathcal{S}_i$ must not learn any information about the input data of other sources. In particular, they should not learn any information about the intersection, which is implicitly computed as part of the join.

Helper Obliviousness: The helper $\mathcal{H}$ must not learn any information about the input data sets (beyond benign leakage, such as the size of the input data) nor any information about the join computation it facilitates.

Anonymous Join: The receiver $\mathcal{R}$ only learns the anonymously joined version. Thus, it does not learn the underlying identifiers of the joined values (*uid hiding*), nor any values outside the join, i.e., values associated with identifiers that have not been present in all tables (*value hiding*).

One-way Flow: The protocol must not require the sources to communicate with each other, and only require a simple upload of the sources to the helper. For efficiency, the complexity of the sources computation and communication should be independent of n, the number of sources.

Provably Secure Protocol. We formally define these properties in the form of an ideal functionality in the UC framework and present an efficient n-party protocol. The core idea of the protocol is rather simple: we let the sources only upload an encrypted version of their local data to the helper, using a public key of the receiver. We then rely on the helper to add another layer of encryption, where all data entries are encrypted under a key that is only recoverable by $\mathcal{R}$ for values within the join. The main challenge is adding this second layer of encryption and key recovery without the helper and receiver learning anything about the underlying identifiers. We achieve this through a careful combination of (partially) homomorphic encryption, such as ElGamal, and ideas from (3-party) oblivious pseudorandom functions. Finally, we formally prove the security of our protocol against semi-honest, colluding adversaries, and under standard cryptographic assumptions. While our main protocol targets the *complete* inner join, i.e., only values for identifiers present in *all* data sets should be combined, we also sketch how it extends to the threshold (t, n) case.

Efficient Implementation. Our protocol, which we dub π_{MPPJ}, achieves low leakage, high efficiency, and ease of deployment by relying on the *helper*-assisted setting. Specifically, for n databases with m rows, the protocol requires only two rounds of communication: one upload of size $O(m)$ per source party, and one download of size $O(n \cdot m)$ for the receiver. Furthermore, the protocol's local operations benefit from parallelization and pipelining techniques, which make the protocol scalable to large problem sizes. We demonstrate this by providing a proof-of-concept implementation and evaluating its performance experimentally. Our implementation achieves low latency and reduces by 1.02 to 20 times the communication cost compared to the latest *private-join-compute* protocol [11] (with no computation over the join). Our code is open-source[2].

2 System and Security Model of MPPJ

In this section, we introduce the relevant parties of the multi-party private join (MPPJ) protocol, the design constraints we consider and formally define the desired functionality.

Basic Notation. We use brackets to define sets, e.g., $\{a, b\}$ denotes a set containing elements a and b. For a set X, we write $x \xleftarrow{\$} X$ for the uniform sampling of an element from X and $|X|$ as its cardinality. We use $[n]$ as a shorthand for the set $\{1, \ldots, n\}$.

[2] https://github.com/hpicrypto/mppj-artifacts

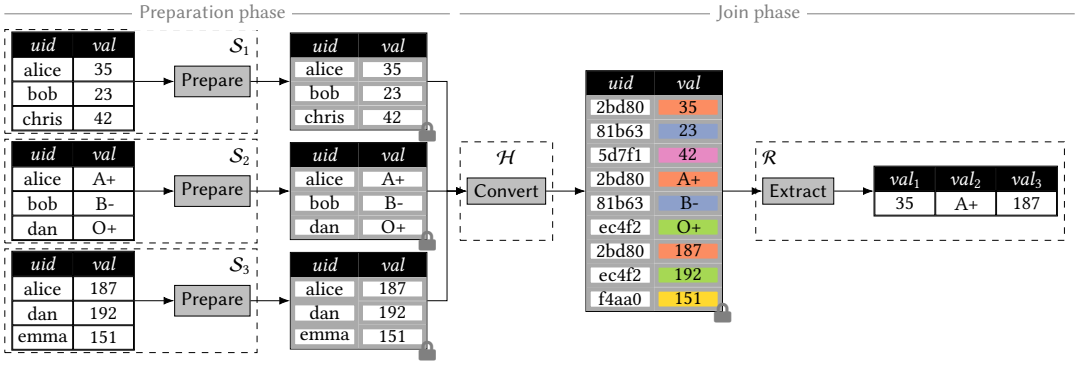

Figure 2: Overview of our MPPJ protocol with two phases, three algorithms, and a one-way communication flow.

Tables. We model an m-row data table as set of tuples $T = \{(uid_j, val_j)\}_{j\in[m]}$, where we refer to $uid_1, \ldots uid_m$ as the *unique identifiers* and to $val_1 \ldots val_m$ as the *values*. The identifiers are pairwise-distinct among T, and we write $\mathsf{UIDs}(T) = \{uid_1, \ldots, uid_m\}$ the set of identifiers in T. Multi-column tables are represented in the same way, by collapsing all column-values into a single value (e.g., as a comma-separated values string). Note that, in the scope of our ideal functionality, this simplification does not entail any loss of generality (we further elaborate on this aspect after introducing this ideal functionality, in Section 2.1). We write $T[uid]$ the value in T associated with uid. We generalize this notation to sets $S = \{(uid_j, val_j)\}_{j\in[m]}$ with non-pairwise-distinct identifiers, where $S[uid]$ becomes the set of values associated with uid.

Parties & Anonymous Join. We consider three different entities in our system: n *data sources* $\mathcal{S}_1, \ldots, \mathcal{S}_n$, a *helper* $\mathcal{H}$, and a *receiver* $\mathcal{R}$. Each source $\mathcal{S}_i$ holds a table T_i with m rows, and the receiver $\mathcal{R}$ aims to obtain an *anonymous join* $J = T_1 \bowtie \ldots \bowtie T_n$ as defined in Definition 1.

Definition 1 (Anonymous Join). *For two tables T_1, T_2 we define the anonymous (inner) join as*

$$T_1 \bowtie T_2 := \{\ (T_1[uid_i], T_2[uid_i]) \mid \forall\ uid_i \in \mathsf{UIDs}(T_1) \cap \mathsf{UIDs}(T_2)\ \}.$$

When joining n tables, $\bowtie_{i=0}^{n} T_i = T_1 \bowtie \ldots \bowtie T_n$ contains the concatenated value tuples for all the uids that are present in all T_i (and the notation extends accordingly).

This operation corresponds to an *inner join* in database terminology. Note that Definition 1 captures a *complete* inner join, i.e., a *uid* must appear in *all* n tables, for the values to be added to J. In Section 6.2 we also discuss how our work can be extended to the case of a *threshold join*, where a value *val* for (*uid*, *val*) is included in J whenever *uid* is present in at least $t < n$ of the tables.

One-way Flow. We aim at a scalable and simple setup, enabled through the central – yet fully oblivious – helper $\mathcal{H}$. The sources and the receiver are referred to as the *endpoints*. Our goal is to achieve optimal efficiency and simplicity, i.e., the protocol must only require endpoint-to-helper communication, and only a single round of interaction from each endpoint. Our *one-way flow* requirement mandates a two-phases protocol:

Preparation Phase: Each $\mathcal{S}_i$ prepares its input for the join by running a Prepare algorithm. It takes the source's table T_i as input, and computes an *obfuscated table*. The output of this procedure is sent to $\mathcal{H}$.

Join Phase: The join phase consists of two algorithms, run by $\mathcal{H}$ and $\mathcal{R}$, respectively. Upon receiving the obfuscated tables from all sources, the helper $\mathcal{H}$ runs a Convert algorithm to compute an *obfuscated join*. The output of this procedure can be retrieved by $\mathcal{R}$, which runs a Extract algorithm to obtain the join result.

The resulting protocol flow is illustrated in Figure 2, and has a considerable advantage in terms of practical deployment: it makes the endpoints independent of each other. Specifically, the sources can go offline after uploading their own obfuscated tables, and the receiver does not need to be online for the whole preparation and conversion process. We observe that this communication flow is the same as for an equivalent plaintext system.

Indeed, the transferred data volumes will be larger due to the cryptographic overhead. In this regard, we aim at *scalable communication*, i.e., that the size of each n obfuscated table is $O(m)$, i.e., independent of the number of sources. This requirement ensures that our protocol scales to a large number of sources. Similarly, the obfuscated join size must be $O(n \cdot m)$ for n tables with m rows each, which is optimal (as any smaller output size would in theory leak information about the actual size of the join to $\mathcal{H}$). Note that our protocol achieves the same asymptotics for the computation time.

2.1 Security Model

In terms of privacy, we aim at a multi-party private computation of the *anonymous join* among the sources $\mathcal{S}_1, ..., \mathcal{S}_n$, assisted by a helper $\mathcal{H}$ and with an external receiver $\mathcal{R}$. As is common, the MPPJ must guarantee the privacy of the inputs: the sources $\mathcal{S}_i$ learn nothing about the input tables T_j of other sources (*source privacy*) and the helper $\mathcal{H}$ must not learn any information about the input tables $T_1, \ldots, T_n$, nor the output join J (*helper obliviousness*). In addition, $\mathcal{R}$ must not learn the underlying identifiers of the joined values (*uid hiding*), nor any values outside the join (*value hiding*).

We now formally define these properties through an ideal functionality in the Universal Composability (UC) [8] framework. The ideal functionality provides the desired computation in a way that

Preparation. On input $(\texttt{PREP}, \text{sid}, T_i)$ by data source $\mathcal{S}_i$

1: Send $(\texttt{PREP}, \text{sid}, \mathcal{S}_i, |T_i|)$ to the adversary, wait for $(\texttt{PREP}, \text{sid}, \mathcal{S}_i, OK)$
2: Create a record $(\texttt{PREP}, \text{sid}, T_i, \mathcal{S}_i)$, output $(\texttt{PREP}, \text{sid}, \mathcal{S}_i)$ to $\mathcal{H}$

Join. On input $(\texttt{JOIN}, \text{sid})$ by helper $\mathcal{H}$

1: Proceed only, if there are records $(\texttt{PREP}, \text{sid}, T_i, \mathcal{S}_i)$ for all $\mathcal{S}_i \in \text{sid}$
2: Compute $J \leftarrow T_1 \bowtie \cdots \bowtie T_n$
3: If $\mathcal{R}$ is corrupt, set $\ell \leftarrow \mathsf{Leak}(T_1, \ldots, T_n)$, otherwise $\ell \leftarrow \perp$
4: Send $(\texttt{JOIN}, \text{sid}, \ell)$ to the adversary, wait for $(\texttt{JOIN}, \text{sid}, OK)$
5: Send $(\texttt{JOINED}, \text{sid}, J)$ to $\mathcal{R}$

Figure 3: The $\mathcal{F}_{\mathsf{MPPJ}}$ ideal functionality for $\text{sid} = (\text{sid}', \mathcal{H}, \mathcal{R}, \mathcal{S}_1, \ldots, \mathcal{S}_n)$**, adversary $\mathcal{A}$, and leakage** Leak**.**

is secure-by-design. Our ideal functionality for the multi-party private join, $\mathcal{F}_{\mathsf{MPPJ}}$, is presented in Figure 3. We now explain how it models the desired properties.

Session Setup. The ideal functionality is instantiated with a session identifier $\text{sid} = (\text{sid}', \mathcal{H}, \mathcal{R}, \mathcal{S}_1, \ldots, \mathcal{S}_n)$ where sid' is a globally unique string to identify the instance of $\mathcal{F}_{\mathsf{MPPJ}}$, and $\mathcal{H}, \mathcal{R}, \mathcal{S}_1, \ldots, \mathcal{S}_n$ are global identities for the parties. That is, we assume that the protocol participants are fixed before the protocol execution. This is the standard assumption for simplified UC, typically used in MPC protocols [9], and is also natural in our setting: the sources must agree to join their data with a certain set of other sources and then consent to make their data available towards a dedicated receiver $\mathcal{R}$. Within a session sid, the functionality will only accept inputs from parties as specified therein.

Preparation & Join. Each source $\mathcal{S}_i$ provides its input table by invoking the $(\texttt{PREP}, \text{sid}, T_i)$ interface and each table is stored by the functionality. When all sources $\mathcal{S}_i$ in sid have uploaded their tables, the helper $\mathcal{H}$ can trigger the join phase through the $(\texttt{JOIN}, \text{sid})$ interface. $\mathcal{F}_{\mathsf{MPPJ}}$ then computes $J \leftarrow T_1 \bowtie \cdots \bowtie T_n$ as defined in Definition 1 and outputs $(\texttt{JOINED}, \text{sid}, J)$ to $\mathcal{R}$.

Functionality & Multi-column Databases. A protocol implementing $\mathcal{F}_{\mathsf{MPPJ}}$ provides an anonymous multi-party join. Importantly, this functionality does not include any further processing such as deduplication, grouping or aggregation (which would belong to the private-join-compute domain). Hence, $\mathcal{F}_{\mathsf{MPPJ}}$ is independent of the values and their format, which is why collapsing multi-column databases into single-column ones does not reduce the generality of our model. Indeed, the receiver can expand the single value back into multiple columns, which effectively supports multi-column joins. In practice, this assumes that the sources have agreed upon common identifiers, on the data schema, and on which source is going to contribute which column as its input.

Leakage. We distinguish between two types of leakages: the *benign* and *parameterizable* leakages. The benign leakage corresponds to information that is commonly accepted as trivial in the literature. In our functionality, this corresponds to sid and $\mathcal{S}_i, |T_i| \ \forall i \in [n]$ (i.e. the table sizes), which are leaked during the preparation phase. Indeed, sizes-related leakage can be made trivial by padding the tables to a maximum allowed size in the construction. The parameterizable leakage corresponds to non-trivial information $\ell \leftarrow \mathsf{Leak}(T_1, \ldots, T_n)$ that is received by the adversary if $\mathcal{R}$ is corrupt. This Leak parameter introduces flexibility for protocol construction, hence it lets us capture and compare the security properties of partial solutions proposed in the literature (see Section 3). But it requires care in its definition. The ideal leakage function would be $\mathsf{Leak} = \perp$, which provides optimal privacy, but appears challenging to be realized efficiently (see the discussion of related work in Section 3). In the worst case, $\mathsf{Leak}(T_1, \ldots, T_n) = T_1, \ldots, T_n$, the adversary would learn all input data which makes the security model meaningless. Thus, the challenge is to find a reasonable trade-off between acceptable extra leakage, and the efficiency gains empowered by that.

In Section 4, we present our protocol which realizes $\mathcal{F}_{\mathsf{MPPJ}}$ with a minor leakage: it leaks the *sizes* of all *partial joins*. A partial join refers to smaller intersections among the $\mathsf{UIDs}(T_i)$, i.e., where *uid* appears across multiple – but not all– input tables T_i. More specifically, the leakage function of our protocol is defined as follows:

Definition 2 (Partial Join Size Leakage Function). *The leakage function* Leak_{PJS} *for tables* $T_1, \ldots T_n$ *is defined as*

$$\mathsf{Leak}_{PJS}(T_1, \ldots, T_n) \coloneqq \{\ |X| \quad | \quad X \in PartJoins\ \}$$

where $PartJoins$ *defines the set of partial joins as*

$$PartJoins \coloneqq \{\bowtie_{T_i \in X} T_i \quad | \quad X \in \mathsf{P}(\{T_1, \ldots, T_n\}),\ 1 < |X| < n\}$$

and where P *denotes the power set.*

Under this leakage function, the adversary only learns that a partial join of a specific size exists. In particular, no further information is revealed matching rows, including which sources they belong to. Although $\mathsf{Leak}_{\mathsf{PJS}}$ is indeed inferior to an ideal leakage, we still consider it an acceptable privacy trade-off in the light of the efficient protocol it enables.

Corruption & Collusion Model. We consider a *semi-honest* adversary that can corrupt a subset of the parties in the protocol. Hence, we characterize the adversary as a subset $\mathcal{A}$ of the protocol participants, i.e.,

$$\mathcal{A} \subset \{\mathcal{S}_1, \ldots, \mathcal{S}_n, \mathcal{H}, \mathcal{R}\} \quad \text{where}$$
$$(1)\ |\mathcal{A} \cup \{\mathcal{S}_1, \ldots, \mathcal{S}_n\}| \leq n - 1 \text{ and}$$
$$(2)\ |\mathcal{A} \cup \{\mathcal{H}, \mathcal{R}\}| \leq 1.$$

Thus, the adversary can corrupt all but one of the sources, and either the helper or receiver – but never both at the same time. Specifically, Condition 1 excludes the case where the adversary has access to all input data. This is the trivial case, where the adversary is already privy of all data and there is nothing left to protect. Condition 2 excludes a corrupted helper to collude with a corrupted receiver. Although this is a non-trivial assumption, we observe that it is unlikely that a protocol satisfies our efficiency requirement without this condition, as this would imply a single-round MPPJ protocol.

Privacy Properties of $\mathcal{F}_{\mathsf{MPPJ}}$. Our functionality naturally enforces the desired properties outlined in Section 1: The helper, despite being the essential intermediary, does not learn anything beyond which parties participate and the size of the input tables (*helper obliviousness*). Likewise, the sources do not receive any output (*source privacy*). The receiver receives the *anonymous join* J, which only contains the joined values for *uid*s that are in all input tables, and with the *uid*s stripped away (*uid-hiding*). If $\mathcal{R}$ is corrupt, the

adversary additionally receives $\ell \leftarrow \mathsf{Leak}(T_1, \ldots, T_n)$, which defines the additional information $\mathcal{R}$ gets. Our definition of the adversary requires that those properties hold even when both some sources *and* the receiver are corrupted.

It is also crucial to observe that the joined output $J = \bowtie_{i=0}^{n} T_i$ is a *set*. Thus, while the values $(val_1, \ldots, val_n) \in J$ are ordered according to parties' identifiers as per Definition 1 (which is important to maintain the context of the value), there is no ordered structure of these values *within* J. As a consequence, J does not reveal any information about the position of the values in the original tables (which might for example leak information about the corresponding *uid*).

Finally, we observe that our model does not impose any condition on the values. As such, it does not model any reidentification or membership-inference attacks that could be mounted directly from the ideal functionality itself. We nevertheless discuss such attacks and the inherent risk associated with deploying PSI-like applications in Section 7.

3 Related Work

We now discuss the related works on protocols for join-like functionalities. When applicable, we exploit the parameterizable leakage of $\mathcal{F}_{\mathsf{MPPJ}}$ to position these works from a privacy perspective.

Generic MPC. One could, in theory, apply generic secure multi-party computation protocols to compute an MPPJ. This results in optimal leakage of the table and values' sizes, i.e., $\mathsf{Leak} = \perp$. However, directly expressing an n-party join over m rows as an MPC circuit requires at least $n \cdot m$ comparisons, which are particularly costly in MPC (we demonstrate the concrete cost in Section 5.3, as a conservative baseline). Moreover, introducing (oblivious) helpers does not trivially reduce the cost of current generic MPC protocols. Specifically, secret-sharing-based protocols such as those implemented in the MP-SPDZ framework [20] seldom benefit from third-party helpers, because of their reliance on broadcasts for each multiplication gate in the computed circuit. FHE-based MPC [3, 27], such as the Helium framework [26], benefit from the network cost perspective, but would require expressing the join as an FHE circuit. Such a circuit would require $O(n^2m^2)$ homomorphic comparisons over encrypted *uid*, which would be impractical for the helper to run over large databases. For example, the most recent works require computation times in the order of seconds [18, 21] for a single homomorphic comparison over 64-bit messages (note that 64 bits would also be too small for typical *uid*s such as usernames and emails). Finally, generic MPC solutions require at least two communication rounds, or even require parties to be synchronously online; this fails to achieve our *one-way flow* requirement.

MPSI. Although semantically close to our goal, multi-party private set intersection (MPSI) protocols do not consider associated values, hence do not trivially achieve MPPJ. Nevertheless, since joins are defined from the intersection of the identifiers' sets, it seems natural to use MPSI *as a building block* to construct MPPJ. However, this would reveal the intersecting *uid* to the MPSI receiver (either the sources or the receiver), which contradicts our *source-privacy* requirement. Although one could imagine running the MPSI over pseudonyms, this would still require a private information retrieval (PIR) mechanism to retrieve the corresponding values, yet without leaking the number of retrieved values. Current PIR protocols (which retrieve single values) would only achieve this at the cost of $\mathcal{R}$ making $n \cdot m$ PIR queries, which does not scale to large databases. Moreover, this would require the receiver to query the sources directly (PIR protocols assume that the database is known in plaintext to the sender), hence it would not satisfy our *one-way flow* requirement.

Labeled-PSI. Protocols for *labeled PSI* [10, 12] or *PSI with associated data* (PSI-AD) [7] only partially address the lack of support for associated values in PSI solutions: they cover the setting where only one party holds values associated with its identifiers (i.e., the *labels*). These protocols also reveal the identifiers to the receiver, which would be leakage in our setting. This leads to a leakage $\mathsf{Leak}(T_1, \ldots, T_n) = \mathsf{UIDs}(T_1) \cup \ldots \cup \mathsf{UIDs}(T_n)$, which is the same leakage as running a plain PSI over the identifiers, and then running a private information retrieval protocol to retrieve the values.

Private Join and Compute. Several works have addressed a seemingly more general problem of computing a function over the values in the join [11, 19, 24, 28, 29]. For example, computing the sum [19] or inner product [24]. Interestingly, most of these works actually do not provide a standalone MPPJ protocol: They only support a restrained set of aggregation functions (which does not include the identity function), and do not enable running the join part of the protocol alone. Two notable exceptions are the DPMC [28] and IDCloak [11] protocols, for which the parties learn the output of the join as secret-shares and which enables reconstruction of the join in a single additional round. However, the DPMC is restricted to the setting of a left join between a main database (on the left) and n data sources, whereas we are interested in symmetric inner-joins. This makes the IDCloak protocol, which is claimed to be the first n-party private join [11], our closest related work. Although the exact leakage for IDCloak is not specified nor completely clear (the work is a preprint at the time of writing), we nevertheless provide a performance comparison in Section 5.3. While PJC protocols might benefit from introducing a helper, the feasibility of such protocol modifications would need to be assessed on a case-by-case basis (and they would likely constitute contributions on their own).

Joins via Pseudonyms. Another family of works proposes to compute the join by deriving join-specific pseudonyms from the *uid*s, and matching them *in plaintext*. PrivateID [33] provides such a functionality, but is limited to two parties and reveals the matching pseudonyms to both parties. 2-PPJ is straightforward in this case as the parties can simply send the corresponding values to each other. The ScrambleDB [23] protocol targets our setting with n parties assisted by a helper. But it targets a different threat model than ours and does not hide the values outside of the join from the receiver. Its leakage corresponds to $\mathsf{Leak}(T_1, \ldots, T_n) = \widetilde{T_1}, \ldots, \widetilde{T_n}$ where $\widetilde{T_i}$ denotes the complete, yet pseudonymized, table.

One advantage of joining over pseudonyms is that comparisons (between the join-specific pseudonyms) are performed *in plaintext*. Although this seems to inherently imply the leakage of partial joins, this brings tremendous benefit in terms of efficiency and simplicity. Hence, we use this approach as a starting point, and our protocol can be seen as enabling *source privacy* and *value hiding* in pseudonym-based joins.

4 Construction

We now present our protocol; it securely instantiates $\mathcal{F}_{\mathsf{MPPJ}}$ and implements the desired *one-way flow* design as defined in Section 2. We first introduce the required building blocks in Section 4.1. Then, we present our protocol in two steps: We start from a simplified version in Section 4.2 that satisfies our *one-way flow* and *value hiding* requirements. The construction reveals the *uid*s to both the helper and receiver, though, and this lets us sketch the challenges related to hiding them. We then present our full protocol in Section 4.3.

4.1 Building Blocks

We now introduce the primitives required by our protocol and briefly state their security properties here. For completeness, we recall the related standard security definitions in Section A.

Generic Encryption Schemes. We employ a standard *symmetric encryption scheme* $\mathsf{SE} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$. $\mathsf{KGen}(1^\lambda)$ produces a symmetric key $k \in \{0,1\}^\lambda$. $\mathsf{Enc}(k, m) \to c$ encrypts a message m to a ciphertext c, and $\mathsf{Dec}(k, c) \to m$ decrypts c to m. We only require IND-EAV security for SE, as our protocol only SE-encrypts messages with single-use keys.

We also require a *public-key encryption scheme* $\mathsf{PKE} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{ReRnd})$, which supports the rerandomization of its ciphertexts. $\mathsf{KGen}(1^\lambda)$ generates a key pair (sk, pk). $\mathsf{Enc}(pk, m) \to c$ encrypts a message m to a ciphertext c, and $\mathsf{Dec}(sk, c) \to m$ decrypts c to m. $\mathsf{ReRnd}(pk, c) \to c'$ is a public algorithm which, upon input of a ciphertext encrypting some message, outputs a random ciphertext encrypting the same message. We require PKE to be IND-CPA secure, and the rerandomization algorithm to satisfy the ReRand security notion. Intuitively, this notion ensures that a rerandomized ciphertext is indistinguishable from a freshly encrypted one for the same message, even if the adversary knows sk. This security is formally defined in Section A, Definition 3. Finally, we assume that the ciphertexts of PKE can be used as messages for SE.

Groups. We use a group $\mathbb{G}$ of prime order q (meaning that the field of exponents is the integers $\mathbb{Z}$ modulo q) in multiplicative notation. We denote by g an agreed upon generator of $\mathbb{G}$. To describe security of $\mathbb{G}$, we informally recall the DDH assumption: it must be hard to distinguish g^{ab} from a random group element, even when g, g^a, g^b are known (see Section A for a formal definition). We assume that the public parameters $pp = (\mathbb{G}, g, q)$ describe the group.

Hash Functions. Our protocol also requires two hash functions, that can map from or into the group $\mathbb{G}$ from pp, which we denote as $\mathsf{H}_{\mathbb{G}} : \{0,1\}^* \to \mathbb{G}$ and $\mathsf{H} : \mathbb{G} \to \{0,1\}^\ell$.

ElGamal. In addition to PKE, we need an IND-CPA-secure asymmetric encryption scheme with rerandomizable ciphertexts that is also homomorphic over $\mathbb{G}$. We use the ElGamal encryption scheme for that purpose [13], which we denote ElG and recall below:

$\mathsf{ElG.KGen}(pp)$: $x \xleftarrow{\$} \mathbb{Z}_q$, $sk \leftarrow x, pk \leftarrow g^x$, return (sk, pk)
$\mathsf{ElG.Enc}(pk, m)$: $r \xleftarrow{\$} \mathbb{Z}_q$, $c \leftarrow (g^r\ ,\ pk^r \cdot m)$, return c
$\mathsf{ElG.Dec}(sk, c = (c_0, c_1))$: $m \leftarrow c_1 \cdot c_0^{-sk}$, return m
$\mathsf{ElG.ReRnd}(pk, c{=}(c_0, c_1))$: $r \xleftarrow{\$} \mathbb{Z}_q$, $c' \leftarrow (c_0 \cdot g^r, c_1 \cdot pk^r)$ return c'

It is well-known that ElG as formulated above achieves perfect rerandomization (i.e., the indistiguishability is statistical). We also require the following homomorphic group operations:

$\mathsf{ElG.Mul}(pk, c = (c_0, c_1), s)$: $c' \leftarrow (c_0, c_1 \cdot s)$ return c'
$\mathsf{ElG.Exp}(pk, c = (c_0, c_1), e)$: $c' \leftarrow (c_0^e, c_1^e)$, return c'

We have that $\forall c$ such that $\mathsf{Dec}(sk, c) = m$, $\forall s \in \mathbb{G}$, and $\forall e \in \mathbb{Z}_q$: $\mathsf{Dec}(sk, \mathsf{Mul}(pk, c, s)) = m \cdot s$ and $\mathsf{Dec}(sk, \mathsf{Exp}(pk, c, e)) = m^e$. We also define a shortcut operation, RExp, which applies ReRnd to the output of the corresponding operation.

The primary reason for using ElGamal specifically in our protocol is that its plaintext space $\mathbb{G}$ enables the homomorphic evaluation of pseudorandom functions, thereby enabling the construction of an oblivious pseudorandom function (OPRF) protocol.

Three-Party OPRFs. In a three-party OPRF protocol, a sender holds input x, a server holds a key k, and a receiver obtains a pseudorandom output $y = \mathsf{PRF}(k, x)$ without the server learning x or y. We rely on the Hash-DH-based construction by Lehmann [23]. This construction exploits the fact that $\mathsf{PRF}_{\mathsf{DH}}(k, x) = \mathsf{H}_{\mathbb{G}}(x)^k$ is a secure pseudorandom $\{0,1\}^* \to \mathbb{G}$ function, assuming that $\mathsf{H}_{\mathbb{G}}$ is a random oracle and that the DDH assumption holds in $\mathbb{G}$. It can be instantiated from ElG as follows:

Setup: The receiver computes $(sk, pk) \leftarrow \mathsf{ElG.KGen}(pp)$ and sends pk to the sender and server. The server chooses $k \xleftarrow{\$} \mathbb{Z}_q$.
Blind request: The sender computes $c \leftarrow \mathsf{ElG.Enc}(pk, \mathsf{H}_{\mathbb{G}}(x))$, and sends c to the server.
Blind evaluation: The server computes $c' \leftarrow \mathsf{ElG.Exp}(pk, c, k)$ and sends c' to the receiver.
Unblinding: The receiver outputs $y \leftarrow \mathsf{ElG.Dec}(sk, c')$.

From a high-level, the PRF evaluation is *oblivious* because the server evaluates $\mathsf{H}_{\mathbb{G}}(x)^k$ homomorphically, under ElG encryption. The output is *pseudorandom* from the DDH assumption (applied over the ElG plaintext-space group). Importantly, we do not use the 3-Party OPRF as a *black-box* (hence we do not define a syntax for it) and, instead, we integrate it (in its ElG-based instantiation described above) directly into our protocol. We do so because our protocol relies on deriving pseudorandom values (obliviously) as part of a larger homomorphic computation. This is the main reason why our protocol is specifically formulated for ElGamal.

Secret Sharing. Finally, we rely on a secret sharing scheme $\mathsf{SS} = (\mathsf{Share}, \mathsf{Reconstruct})$ over $\mathbb{Z}_q$: for any secret $x \in \mathbb{Z}_q$, $\mathsf{Share}(x, t, n) \to x_1, \ldots x_n$ outputs n shares such that x can only be recovered from the knowledge of any t shares, as $x = \mathsf{Reconstruct}(x_1, \ldots x_t)$. Our main protocol requires $t = n$, hence relies on additive secret-sharing where $x = x_1 + \cdots + x_n \mod q$. We nonetheless use the threshold syntax with parameter t because our extension to threshold private join relies on secret sharing scheme of Shamir [31].

4.2 Simplified Protocol w/o UID Hiding

We first sketch a simplified protocol for MPPJ. This protocol does not satisfy all the security properties, and is not practically relevant because simpler techniques could achieve the same (incomplete) privacy. However, it introduces the core idea of our construction

and highlights the challenges towards achieving *uid*-hiding. We address these challenges in our full protocol in Section 4.3.

In the simplified protocol, the receiver $\mathcal{R}$ only learns the values in the *uid*-based join (achieving value hiding), and the helper does not learn anything about the values (helper obliviousness, but for the values only). This solution also satisfies our *one-way flow* requirement. However, it does not provide any privacy of *uid*s; they are instead fully revealed to the helper and receiver.

Double Encrypt & Secret Share. The simplified protocol combines a double-encryption mechanism with secret-sharing to implement the two-phase protocol described in Section 2. The setup consists in the receiver $\mathcal{R}$ generating a key pair $(esk, epk) \leftarrow \mathsf{PKE.KGen}(1^\lambda)$, and making *epk* available to all sources. We briefly describe the Prepare, Convert and Extract procedures, then discuss the achieved properties. For completeness, a pseudocode-description of the algorithms is provided in Section B.

Prepare: Each source $\mathcal{S}_i$ partially encrypts their input T_i, by iterating through each row $(uid_{i,j}, val_{i,j}) \in T_i$ and replacing the value $val_{i,j}$ with its encryption under *epk*. That is, $\mathcal{S}_i$ assembles the obfuscated table as

$$T'_i \leftarrow \left\{(uid_{i,j}, cval_{i,j})\right\} \text{ with } cval_{i,j} \leftarrow \mathsf{PKE.Enc}(epk, val_{i,j})$$

and sends T'_i to the helper $\mathcal{H}$.

Convert: Upon receiving all obfuscated tables $T'_1, \ldots, T'_n$, the helper generates a fresh key $k_l \leftarrow \mathsf{SE.KGen}(1^\lambda)$ for each unique $uid_l \in \mathsf{UIDs}(T'_1) \cup \cdots \cup \mathsf{UIDs}(T'_n)$. It also computes $k_{l,1}, \ldots, k_{l,n} \leftarrow \mathsf{SS.Share}(k_l, n, n)$, and stores these *uid*-specific keys along with their shares as $K \leftarrow \{(uid_l, k_l, k_{l,1}, \ldots, k_{l,n})\}$. Then, for each $i \in [n]$ and each tuple $(uid_{i,j}, cval_{i,j}) \in T'_i$, the helper encrypts $cval_{i,j}$ under the $uid_{i,j}$-specific k_l (which adds a second encryption layer) and attaches the i-th share of that key. That is, the helper computes the obfuscated join as:

$$J' \leftarrow \{(uid_{i,j}, Cval_{i,j}, k_{l,i})\} \text{ where } (k_l, ..., k_{l,i}, ...) \leftarrow K[uid_{i,j}]$$
$$Cval_{i,j} \leftarrow \mathsf{SE.Enc}(k_l, cval_{i,j})$$

As a result, each encryption $Cval_{i,j}$ under k_l is appended with its source-specific share of that *uid*-specific key. Thus, only when $\mathcal{R}$ receives tuples where the $uid_{i,j}$ was present in *all* T_i, it will be able to recover k_l from all n shares and be able to decrypt $Cval_{i,j}$.

Extract: Upon receiving J' the receiver groups the tuples by *uid*, and filters out all groups with less than n entries. This results in a table $J'' = \{(Cval_{l,1}, ..., Cval_{l,n}, k_{l,1}, ..., k_{l,n}),\ l \in [|J|]\}$ which can be decrypted by reconstructing k_l and unwrapping the two layers of encryption:

$$J \leftarrow \{(val_{l,1}, ..., val_{l,n}) \mid val_{l,i} \leftarrow \mathsf{PKE.Dec}(esk, cval_{l,i})$$
$$cval_{l,i} \leftarrow \mathsf{SE.Dec}(k_l, Cval_{l,i})$$
$$k_l \leftarrow \mathsf{SS.Reconstruct}(k_{l,1}, ..., k_{l,n})\}$$

(Un)achieved Properties. This simple protocol satisfies the one-way flow design, and already achieves source privacy, as well as value-hiding towards $\mathcal{H}$ and $\mathcal{R}$: $\mathcal{H}$ does not learn anything about the input values $val_{i,j}$ (based on the security of PKE), and $\mathcal{R}$ does not learn anything about the values *not* in the join (based on the security of SE and SS). However, the protocol does not achieve *uid*-hiding, because it discloses the *uid*s of all input values to both $\mathcal{H}$ and $\mathcal{R}$. To achieve the full set of targeted properties, we must therefore replace the mechanisms that involve the *uid*s with oblivious equivalents. These mechanisms are:

Mechanism 1: *uid-based grouping*: In Extract, the receiver computes the join by performing a grouping of the encrypted values according to their associated *uid*s.
Mechanism 2: *uid-specific value encryption key*: In Convert, the helper encrypts the values under a key k_l specific to the corresponding *uid*, and attaches only a share of that key. This ensures that k_l is only recoverable when *uid* is present in all input tables.

4.3 The π_{MPPJ} Protocol

We now present our protocol, π_{MPPJ}, and how it realizes the two required mechanisms. It uses a symmetric encryption scheme SE and a rerandomizable public-key encryption PKE for the value encryption, and the ElGamal scheme ElG over the group $(\mathbb{G}, p, g)$ for the *uid* encryption. It makes *implicit* use of the 3-party Hash-DH OPRF [23], in its ElG-based formulation (see Section 4.1). For the communication, the parties rely on $\mathcal{F}_{\mathsf{SMT}}$ [8], which is the ideal functionality for secure message transmission. It ensures both confidentiality and integrity, and can be realized, e.g., through a TLS connection. We assume that a party $\mathcal{P}$ only accepts messages from $\mathcal{S}_i, \mathcal{H} \in \mathsf{sid}$ without making it explicit. Finally, we assume that every *val* has a constant-length representation, such as a reversible encoding to a single group element (we provide an extension that supports arbitrary-size values in Section 6.1). The protocol is shown in Figure 4, and we provide a step-by-step explanation below.

4.3.1 Protocol Overview. The **Setup** operation generates the necessary keys: The receiver publishes an ElGamal key pair (bsk, bpk), which is used for the blinding in our implicitly used OPRF. The helper generates two secret keys for the lifetime of the session: a *pseudonym key* $k_{nym} \in \mathbb{Z}_p$, and a *pad key* $(k_{spad_1}, \ldots, k_{spad_n}) \in \mathbb{Z}_p^n$ such that $k_{pad} = k_{spad_1} + \ldots + k_{spad_n}$ (i.e., an additive sharing of a secret-key k_{pad}). Both serve as the helper's (implicit-)OPRF keys. Then, the protocol follows the same high-level idea as the simplified protocol of Section 4.2, and addresses the missing security features. Conceptually, we achieve this by replacing the two aforementioned mechanisms with privacy-preserving equivalents. Note that both mechanisms correspond to **for** loops over the *tuples* in the databases: In Prepare, each source $\mathcal{S}_i$ processes its plaintext tuples $(uid, val) \in T_i$ into T'_i. In Convert, the helper processes each encrypted tuple in $T'_1 \cup \cdots \cup T'_n$. Hence, we describe below how each mechanism processes a single tuple.

***Mechanism 1**: Grouping over Pseudonyms.* Instead of performing the grouping over the *uid*s, our protocol derives fresh *pseudonyms* for each *uid* and lets the receiver $\mathcal{R}$ perform the grouping based on these pseudonyms. Computing these pseudonyms is done by using the 3-Party OPRF with the key k_{nym}, as

$$nym \leftarrow \mathsf{H}_{\mathbb{G}}(uid)^{k_{nym}},$$

which is done *obliviously* by $\mathcal{H}$ (i.e., without revealing the *uid*). Concretely, this requires the sources to encrypt the *uid* of each tuple under the blinding key *bpk* of the receiver as

$$cuid \leftarrow \mathsf{ElG.Enc}(bpk, \mathsf{H}_{\mathbb{G}}(uid)) \qquad \text{(Prepare Step 3)}$$

when uploading their data to the helper. Then, the helper computes the pseudonyms homomorphically, as

$$cnym \leftarrow \mathsf{ElG.RExp}(bpk, cuid, k_{nym}) \quad \text{(Convert Step 6)}$$

from the encrypted identifier *cuid* and the first OPRF key k_{nym}. We rely on the determinism of the (O)PRF to obtain consistent pseudonyms *nym* for the receiver, while ensuring that the helper cannot learn any information about the input or output it computes. The resulting pseudonyms are used by $\mathcal{R}$ in the Extract step to perform the grouping, yet do not enable an adversarial $\mathcal{R}$ to link back to the original *uid*s (as it does not know the OPRF key k_{nym}). Note that this must hold even if the adversary also controls the source from which a given *uid* originates, which requires rerandomization and shuffling operations, that we describe later.

Mechanism 2*: Oblivious Encryption using Key Shares.* Recall that, in the simple protocol of Section 4.2, this mechanism corresponds to the helper adding a second layer of encryption on the (PKE-encrypted) values, under a *uid*-specific encryption key. But, in the full protocol, $\mathcal{H}$ is only provided with the ElG-encrypted identifier *cuid* (see Mechanism 1). Instead of relying on a *per uid* key for the double encryption, $\mathcal{H}$ samples a fresh *per tuple* key $s \leftarrow \mathbb{G}$, and uses $\mathsf{H}(s)$ as the key for the SE layer. Indeed, this seems to only shift the problem to that of computing an encryption of s under a *uid*-specific key. The main insight is that $\mathcal{H}$ can compute such an encryption of s *homomorphically*, by relying on the OPRF to generate *uid*-specific, pseudorandom masks. Specifically, $\mathcal{H}$ evaluates the OPRF a second time on *cuid*, this time with key k_{pad}:

$$cpad \leftarrow \mathsf{ElG.RExp}(bpk, cuid, k_{pad}) \quad \text{(Convert Step 7)}$$

This results in an ElG-encryption of $pad = \mathsf{H}_{\mathbb{G}}(uid)^{k_{pad}}$. Then, it homomorphically multiplies s to the resulting ciphertext:

$$cs \leftarrow \mathsf{ElG.Mul}(bpk, cpad, s) \quad \text{(Convert Step 8)}$$

The resulting cs is an encryption of $s \cdot pad$, i.e., of s masked in $\mathbb{G}$.

Since $\mathcal{R}$ can decrypt the ElGamal layer (hence retrieve $s \cdot pad$), the last challenge is for $\mathcal{H}$ to enable $\mathcal{R}$ to recover *pad*, yet *only when the associated uid is present in all tables*. $\mathcal{H}$ does so by exploiting the fact that the Hash-DH OPRF is *key-homomorphic*, i.e., that

$$\mathsf{PRF}_{\mathsf{DH}}(k_1, x) \cdot \mathsf{PRF}_{\mathsf{DH}}(k_2, x) = \mathsf{H}_{\mathbb{G}}(x)^{k_1+k_2} = \mathsf{PRF}_{\mathsf{DH}}(k_1 + k_2, x)$$

Specifically, for a tuple originating from source S_i, $\mathcal{H}$ attaches an ElGamal encryption of $spad_i = \mathsf{H}_{\mathbb{G}}(uid)^{k_{spad\,i}}$ where k_{spad_i} is the i-th share of the *mask key* of the session (in Step 9). Hence, by collecting and ElG-decrypting n tuples for the same *nym*, the receiver can compute $pad = \prod_{i=1}^{n} spad_i$ (Extract Steps 6+7), can recover s, and can SE+PKE-decrypt the associated value (Steps 10+11).

Shuffling & Rerandomization. Our ideal functionality $\mathcal{F}_{\mathsf{MPPJ}}$ ensures that even colluding $\mathcal{S}_i$ and $\mathcal{R}$ cannot link the joined data back to its origin, in particular to their *uid*s – other than what is possible through the joined values. This means that our cryptographic protocol must not introduce any traces that allow the corrupt $\mathcal{R}$ to link the joined values $(val_{1,j}, \ldots, val_{n,j})$ back to the underlying uid_i. Hence, none of the ciphertexts generated by $\mathcal{S}_i$ can be sent directly to $\mathcal{R}$, as they would constitute such an unwanted trace. This is why we need all encryption schemes used by $\mathcal{S}_i$ to be rerandomizable: In Step 5 of Convert, $\mathcal{H}$ re-randomizes the PKE-encryption before adding the SE layer. In Steps 6,7,8,9, $\mathcal{H}$ uses the re-randomizing version of the ElG homomorphic operations (see Section 4.1) to derive all the values derived from *cuid*. Note that Mul does not need to rerandomize its output because *cpad* is randomized by RExp.

Furthermore, to prevent the data structure from leaking information about the position of the values in the original tables, we have both $\mathcal{H}$ and $\mathcal{R}$ shuffle their datasets before forwarding and outputting them, respectively. However, since our definition of join (Definition 1) preserves the columns across tables, $\mathcal{H}$ includes the

Setup for $\mathsf{sid} = (\mathsf{sid}', n, \mathcal{H}, \mathcal{R}, \mathcal{S}_1, \ldots, \mathcal{S}_n)$ with $\mathsf{sid}' \xleftarrow{\$} \{0,1\}^\lambda$

1: $\mathcal{R}$ runs $(bsk, bpk) \leftarrow \mathsf{ElG.KGen}$, $(esk, epk) \xleftarrow{\$} \mathsf{PKE.KGen}$
2: $\mathcal{R}$ sets $rsk \leftarrow (bsk, esk)$, and publishes $rpk \leftarrow (bpk, epk)$
3: $\mathcal{H}$ sets $k_{nym}, k_{spad_1}, ..., k_{spad_n} \xleftarrow{\$} \mathbb{Z}_q^{n+1}$
4: $\mathcal{H}$ sets $k_{pad} \leftarrow k_{spad_1} + \ldots + k_{spad_n} \mod q$

Preparation:Prepare Run by $\mathcal{S}_i$ on $(\mathtt{PREP}, \mathsf{sid}, T_i, \mathcal{S}_i)$, $i \in [n]$

1: Initialize $T_i' \leftarrow \emptyset$
2: **for** $(uid_{i,j}, val_{i,j}) \in T_i$ **do**
3: $\quad cuid_{i,j} \leftarrow \mathsf{ElG.Enc}(bpk, \mathsf{H}_{\mathbb{G}}(uid_{i,j}))$
4: $\quad cval_{i,j} \leftarrow \mathsf{PKE.Enc}(epk, val_{i,j})$
5: $\quad$ Add $(cuid_{i,j}, cval_{i,j})$ to T_i'
6: Send $(\mathsf{joinprep}, \mathsf{sid}, T_i')$ to $\mathcal{H}$ via $\mathcal{F}_{\mathsf{SMT}}$
7: $\mathcal{H}$ stores (sid, T_i'), and outputs $(\mathtt{PREP}, \mathsf{sid}, \mathcal{S}_i)$

Join:Convert Run by $\mathcal{H}$ on input $(\mathtt{JOIN}, \mathsf{sid}, \mathcal{R})$

1: Retrieve T_i' for all $i \in \{1, \ldots, n\}$ stored for sid
2: Initialize $J' \leftarrow \emptyset$
3: **for** $i \in [n]$ and every $(cuid_{i,j}, cval_{i,j}) \in T_i'$ **do**
4: $\quad s_{i,j} \xleftarrow{\$} \mathbb{G}$
5: $\quad Cval_{i,j} \leftarrow \mathsf{SE.Enc}(\mathsf{H}(s_{i,j}), (i, \mathsf{PKE.ReRnd}(epk, cval_{i,j})))$
6: $\quad cnym_{i,j} \leftarrow \mathsf{ElG.RExp}(bpk, cuid_{i,j}, k_{nym})$
7: $\quad cpad_{i,j} \leftarrow \mathsf{ElG.RExp}(bpk, cuid_{i,j}, k_{pad})$
8: $\quad cs_{i,j} \leftarrow \mathsf{ElG.Mul}(bpk, cpad_{i,j}, s_{i,j})$
9: $\quad cspad_{i,j} \leftarrow \mathsf{ElG.RExp}(bpk, cuid_{i,j}, k_{spad_i})$
10: $\quad$ Add $(cnym_{i,j}, Cval_{i,j}, cs_{i,j}, cspad_{i,j})$ to J'
11: Send (join, J') to $\mathcal{R}$ via $\mathcal{F}_{\mathsf{SMT}}$, with J' in a random order

Join:Extract Run by $\mathcal{R}$ upon receiving (join, J') from $\mathcal{H}$

1: Initialize sets $J \leftarrow \emptyset$ and $J'' \leftarrow \emptyset$
2: **for** $(cnym_{i,j}, Cval_{i,j}, cs_{i,j}, cspad_{i,j}) \in J'$ **do**
3: $\quad nym_{i,j} \leftarrow \mathsf{ElG.Dec}(bsk, cnym_{i,j})$
4: $\quad$ Add $(nym_{i,j}, Cval_{i,j}, cs_{i,j}, cspad_{i,j})$ to J''
5: **for** all identifiers nym_l that appears n times in J'' **do**
6: $\quad$ Collect $(Cval_{i,l}, cs_{i,l}, cspad_{i,l})_{i \in [n]}$ from $J''[nym_l]$
7: $\quad pad_l \leftarrow \prod_{i=1}^{n} \mathsf{ElG.Dec}(bsk, cspad_{i,l})$
8: $\quad$ **for** $i \in [n]$ **do**
9: $\quad\quad s_{i,l} \leftarrow \mathsf{ElG.Dec}(bsk, cs_{i,l}) \cdot pad_l^{-1}$
10: $\quad\quad (i', cval_{i,l}) \leftarrow \mathsf{SE.Dec}(\mathsf{H}(s_{i,l}), Cval_{i,l})$
11: $\quad\quad val_{i',l} \leftarrow \mathsf{PKE.Dec}(esk, cval_{i,l})$
12: $\quad$ Add $(val_{1,l}, \ldots, val_{n,l})$ to J
13: Output $(\mathtt{JOINED}, \mathsf{sid}, J)$

Figure 4: π_{MPPJ} protocol with public parameters $pp = (\mathbb{G}, g, q)$.

index i in $Cval$, meaning that if $\mathcal{R}$ recovers n shares, $\mathcal{R}$ learns both the required value val and its position in the join tuple i.

Correctness. We provide a correctness analysis in Section C and only briefly highlight the main ideas here. Aside from what follows naturally from the correctness of the encryption schemes, the correctness of π_{MPPJ} hinges upon the correct recovery pad for each value. This follows from (i) the homomorphic properties of the ElGamal, and (ii) from the key-homomorphic property of the Hash-DH-based OPRF. From (i), it follows that $\mathcal{H}$ correctly computes the share of pad, as $\mathsf{H}_{\mathbb{G}}(uid_l)^{k_{spad_i}} = spad_{l,i}$ for a tuple with $nym_l = \mathsf{H}_{\mathbb{G}}(uid_l)^{k_{nym}}$. From (ii), if follows that $\mathcal{R}$ can combine n of these shares to recover the full pad, as

$$\prod_{i=1}^{n} spad_{l,i} = \mathsf{H}_{\mathbb{G}}(uid_l)^{\sum_{i=1}^{n} k_{spad_i}} = \mathsf{H}_{\mathbb{G}}(uid_l)^{k_{pad}} = pad_l.$$

It can then use pad, to recover $s_{i,j}$, which enables the decryption of $Cval_{i,j}$ into $cval_{i,j}$. Finally, val is decrypted from $cval_{i,j}$ using esk.

4.4 Security of π_{MPPJ}

We now show that our protocol π_{MPPJ} securely instantiates $\mathcal{F}_{\mathsf{MPPJ}}$, which guarantees security against passive adversaries under the non-collusion assumption of Section 2.1. At the end of this section, we also explain the impact of colluding adversaries (Section 4.4.2) and active attacks (Section 4.4.3).

4.4.1 Passive Security of π_{MPPJ}. Here we show that π_{MPPJ} is secure in the model introduced in Section 2, and for the leakage $\mathsf{Leak}_{\mathsf{PJS}}$ defined in Definition 2.

Theorem 4.1 (Passive Security of π_{MPPJ}). *The π_{MPPJ} protocol securely realizes the ideal functionality $\mathcal{F}_{\mathsf{MPPJ}}$ for leakage function $\mathsf{Leak}_{PJS}(T_1, \dots T_n)$ in the $\mathcal{F}_{SMT}$-hybrid model against colluding semi-honest adversaries if: (i) the DDH assumption holds in $\mathbb{G}$, (ii) $\mathsf{H}_{\mathbb{G}}$ and H are random oracles, (iii) PKE is IND-CPA-secure and rerandomizable, (iv) SE is IND-EAV-secure, and (v) $\mathcal{H}$ and $\mathcal{R}$ do not collude.*

We provide a proof intuition below and refer for the full proof to Section D. The proof follows a simulation-based paradigm: We must show that for every environment $\mathcal{Z}$ and adversary $\mathcal{A}$, there exists an efficient simulator Sim, such that $\mathcal{Z}$ cannot distinguish whether it is interacting with our π_{MPPJ} protocol or with Sim and $\mathcal{F}_{\mathsf{MPPJ}}$. The simulator Sim mimics all honest parties towards the real-world adversary $\mathcal{A}$ and interacts with $\mathcal{F}_{\mathsf{MPPJ}}$ in the role of the corrupt parties in the ideal world. By $\mathcal{P}_{\mathsf{sim}}$, we denote that the simulator plays the role of the honest party $\mathcal{P}$ towards the adversary in the real-world. As we consider honest-but-curious adversaries, the corrupt parties are also run by the simulator, that executes the original protocol π_{MPPJ} for the respective party, and with $\mathcal{A}$ providing their inputs and receiving their outputs, secret keys as well as all internal values, and randomness. We denote such a passively corrupt party as $\mathcal{P}^*$. We split the proof sketch along two cases, depending on whether the receiver is corrupt or not.

$\underline{\mathcal{R} \notin \mathcal{A}}$. When the receiver is honest, the adversary does not know the receiver's secret key $rsk = (esk, bsk)$. As all values sent by $\mathcal{S}_i$ to $\mathcal{H}$ and from $\mathcal{H}$ to $\mathcal{R}$ are encrypted under epk and bpk respectively, and also protected through $\mathcal{F}_{\mathsf{SMT}}$, the security proof is rather straight-forward. In fact, if the helper is honest too, there is nothing to simulate, as all communication between honest $\mathcal{S}_i$ and $\mathcal{H}$, as well as between $\mathcal{H}$ and $\mathcal{R}$ is secured through $\mathcal{F}_{\mathsf{SMT}}$.

If the helper is corrupt, our simulation must prepare correctly looking ciphertexts stemming from the honest $\mathcal{S}_i$. For each join preparation from an honest source $\mathcal{S}_i$, Sim receives $(\mathsf{PREP}, \mathsf{sid}, \mathcal{S}_i, |T_i|)$ from $\mathcal{F}_{\mathsf{MPPJ}}$, learning only the cardinality of the table T_i. $\mathcal{S}_{\mathsf{sim}}$ generates dummy ciphertexts $\{(cuid_{i,j}, cval_{i,j})\}$ for $j \in [|T_i|]$ which are fresh encryptions under epk and bpk of dummy values of the correct length according to our protocol, and sends them as $(\mathsf{joinprep}, \mathsf{sid}, T_i')$ to $\mathcal{H}^*$. This is indistinguishable to $\mathcal{Z}$ under the IND-CPA security of ElG and PKE.

$\underline{\mathcal{R} \in \mathcal{A}}$. The case where the receiver is corrupt is more challenging, as here $\mathcal{A}$ knows the receiver's secret keys $rsk = (esk, bsk)$, and we must produce correctly looking outputs towards $\mathcal{R}^*$. Note that, in this setting, our adversary model of Section 2.1 implies that $\mathcal{H} \notin \mathcal{A}$. This means that all communication between honest $\mathcal{S}_i$ and $\mathcal{H}$ is fully protected by $\mathcal{F}_{\mathsf{SMT}}$, and there is nothing to simulate for honest sources in the preparation phase. The simulator learns the cardinality of all honest input tables $(\mathsf{PREP}, \mathsf{sid}, \mathcal{S}_i, |T_i|)$ from $\mathcal{F}_{\mathsf{MPPJ}}$ though, which it needs later on.

When the environment initiates a join, the simulator $\mathcal{H}_{\mathsf{sim}}$ must produce a correct looking output of the privately joined J towards the corrupt $\mathcal{R}^*$. In this event, the simulator – in the role of the corrupt $\mathcal{R}$ in the ideal-world – obtains $(\mathtt{JOINED}, \mathsf{sid}, J)$ from $\mathcal{F}_{\mathsf{MPPJ}}$, i.e., it learns the anonymously joined values. Additionally, it gets $(\mathtt{JOIN}, \mathsf{sid}, \ell)$, containing the additional leakage ℓ, which are the sizes of the partial joins from the ideal functionality. $\mathcal{H}_{\mathsf{sim}}$ must simulate the tuples $\{(cnym_{i,j}, Cval_{i,j}, cs_{i,j}, cspad_{i,j})\}$ such that they decrypt to the correct join output. Our simulation must first set the ciphertexts for all values within the join correctly, so that $\mathcal{R}^*$ can detect and decrypt the joined values correctly. The rest are set to dummy values. More precisely, our simulation runs as follows:

Values in J: For each tuple $= \{(val_{1,j}, \dots, val_{n,j})\} \in J$ in J, $\mathcal{H}_{\mathsf{sim}}$ first chooses a random pseudonym tuple $(nym_j, pad_j) \xleftarrow{\$} \mathbb{G}^2$. It computes all $cnym_{i,j}$ as fresh encryptions of nym_j under bpk. It computes $Cval_{i,j}$ as in the protocol, but starts with a fresh encryption of $cval_{i,j} \leftarrow \mathsf{PKE.Enc}(epk, val_{i,j})$. For $cs_{i,j}, cspad_{i,j}$ it deviates and freshly encrypts the expected outcome under bpk, i.e., $cs_{i,j}$ are generated as fresh encryption of $s_{i,j} \cdot pad_j$ and $cspad_{i,j}$ freshly encrypt $spad_{i,j}$. For the latter, $spad_{1,j}, \dots spad_{n,j}$ are multiplicative shares of the padding pseudonym pad_j.

Now, let us argue that these changes are indistinguishable towards $\mathcal{Z}$ and $\mathcal{A}$: The ciphertexts $Cval_{i,j}$ are proper encryptions of $val_{i,j}$, but generated freshly instead from $cval_{i,j}$. This change is indistinguishable based on the ReRand property of PKE (and the fact that the elements of J' are output in a random order).

For $cnym_{i,j}, cs_{i,j}$, and $cspad_{i,j}$ we have now based our computations on random nym_j and pad_j instead of blind evaluations of $cuid_{i,j}$. This is indistinguishable by the DDH and ROM property, that underlies the implicitly performed OPRF computation of $nym_j \leftarrow \mathsf{H}_{\mathbb{G}}(uid_j)^{k_{nym}}$ and $pad_j \leftarrow \mathsf{H}_{\mathbb{G}}(uid_j)^{k_{pad}}$. Further, due to the ReRand property of ElG and the shuffling, the adversary cannot distinguish a homomorphically derived ciphertext from a fresh encryption of the same value.

Values not in J: The simulator must also produce correct looking $(cnym_{i,j}, Cval_{i,j}, cs_{i,j}, cspad_{i,j})$ for J' for all values that are not

in the join. For these tuples, the only information Sim has is the amount of partial joins and the overall size of input tables (the latter was received through (PREP, sid, $\mathcal{S}_i, |T_i|$) from $\mathcal{F}_{\mathsf{MPPJ}}$). We therefore let $\mathcal{H}_{\mathsf{sim}}$ create dummy tuples ($cnym_{i,j}, Cval_{i,j}, cs_{i,j}, cspad_{i,j}$) where $Cval_{i,j}$ merely encrypts a dummy message. For the pseudonyms, it creates random $nym_j \xleftarrow{\$} \mathbb{G}$, which it now uses x times for $x \in \{|X| \mid X \in PartJoins\}$, i.e. x is a partial join size inferred from the leakage ℓ. This means that the same nym_j appears in its encrypted form $cnym_J$ in x tuples ($cnym_j, Cval_{i,j}, cs_{i,j}, cspad_{i,j}$). The ciphertexts $cs_{i,j}, cspad_{i,j}$ are replaced by fresh encryptions of random group elements. In the full proof, we show that this substitution is indistinguishable based on the DDH and ROM assumption (underlying the implicit OPRF computation of pad_j) and the perfect security of additive secret sharing: Recall that here we simulate ciphertexts for values where the *uid* did *not* occur in all n tables, i.e., $\mathcal{R}^*$ will be receiving at most $n-1$ shares of a pseudorandom pad_j value, which we can simulate simply through random shares. Further, the indistinguishability of our simulation of $Cval_{i,j}$ follows from the IND-EAV security of SE. Note that Adv does not learn anything about the underlying key $\mathsf{H}(s_{i,j})$, as we no longer encrypt $s_{i,j}$ in $cs_{i,j}$ anymore.

4.4.2 Notes on Collusion. Our model covers several collusion settings, for which the security follows from the realization of $\mathcal{F}_{\mathsf{MPPJ}}$. Nevertheless, we highlight two specific collusion cases here, which may not be obvious from the above discussion:

$(\mathcal{H}, \mathcal{R}) \in \mathcal{A}$. This collusion setting is excluded from our model, and our proof would therefore not apply to it. From high-level, the issue is that the sources upload their tables in encrypted form, but under a key that is known to the adversary. Concretely, $\mathcal{H}$ could simply forward all the data to $\mathcal{R}$ for it to recover every value (or $\mathcal{R}$ could). To address this setting, one would typically rely on a thresholdized ElG, for which the encryption key would be collectively owned by the sources. But this would require a threshold decryption round for the final result, which would not comply to our *one-way flow* requirement.

$(\mathcal{S}_i, \mathcal{H}) \in \mathcal{A}$. This case is covered by our model and proof. An interesting note is that Leak is not received by the adversary, thus hiding information about partial joins and the adversary cannot perform "sub-joins" by itself, as everything is encrypted towards $\mathcal{R}$.

$(\mathcal{S}_i, \mathcal{R}) \in \mathcal{A}$. This case is included in our model and is thus covered by our proof. Specifically, a colluding $\mathcal{S}_i$ and $\mathcal{R}$ cannot reidentify any *uid* through the execution of the protocol. The helper $\mathcal{H}$ ensures this through the rerandomization and shuffling of all the sources' inputs, which prevent the adversary from linking inputs and output ciphertexts. The OPRF applied by $\mathcal{H}$ also hides all *uid*s from the receiver $\mathcal{R}$. However, we stress that these guarantees are only at the protocol level, and do not exclude any attack on the ideal functionality. Specifically, a reidentification could indeed be possible by exploiting unique combinations of values for a given *uid*. This showcases a general privacy limitation of anonymous private joins w.r.t. distinct values in this collusion setting.

4.4.3 Security against Active Attacks? While our protocol is provably secure only against passive adversaries, we emphasize that it maintains privacy even when the helper $\mathcal{H}$ or receiver $\mathcal{R}$ are *actively* corrupt (still, as long as they don't collude). Actively malicious parties can mainly impact the *correctness* of the computation.

The helper $\mathcal{H}$ operates fully blindly on ciphertexts encrypted under *bpk* and cannot extract any additional information even if it deviates from the protocol. The only advantage the adversary gets through an active corruption of $\mathcal{H}$ is that it can meddle with the join – thus correct outputs are no longer guaranteed. For instance: a corrupt $\mathcal{H}$ can cause $\mathcal{R}$ to recover manipulated, or entirely wrong values and joins. As $\mathcal{H}$ operates on ciphertexts, it cannot target this meddling though. In fact, one can leverage the blind nature of $\mathcal{H}$ to detect such cheating behavior: honest sources and receiver can embed some "test" values in the input tables and check if they are correctly joined towards $\mathcal{R}$.

Active corruption of a receiver $\mathcal{R}$ also has no impact on privacy, as it is purely receiving values. Thus, there is nothing to deviate from that is not already covered by a passive adversary. The only direct impact on privacy can be caused by the active corruption of sources $\mathcal{S}_i$. E.g., they can omit the hash $\mathsf{H}_{\mathbb{G}}$ when computing *cuid*, which no longer enables relying on the pseudorandomness of these values in the security proof. While we do not see an immediate attack based on this, we can no longer argue for privacy, either.

5 Performance

We now discuss a concrete instantiation of π_{MPPJ} (Section 5.1) and analyze its performance from both a theoretical perspective (in Section 5.2) and from an experimental one (in Section 5.3).

5.1 Instantiation

We use the NIST Curve P-256 as the group $\mathbb{G}$, and implement both PKE and ElG over this curve. As ElG is a rerandomizable PKE scheme, we use PKE = ElG. For SE, we use AES128-CTR with an all-zero IV [3]. We use HKDF-SHA256 as the hash function H and the Hash-to-Curve algorithm of RFC9380 as $\mathsf{H}_{\mathbb{G}}$.

Instantiating ElG *for* PKE. We use ElGamal in conjunction with elliptic curves to achieve higher performance and lower bandwidth usage, especially for small values, compared with ElGamal in a finite field. A known challenge when using the ElGamal scheme is finding a message encoding that is both efficiently reversible and is preserved across homomorphic operations, including rerandomization. Specifically, our protocol assumes values to be byte strings which may be too long for encoding them at the exponent (as decoding then involves computing a discrete logarithm). Instead, we use the approach due to Koblitz [22]: we interpret the message concatenated with a 1-byte counter as the x coordinate of a candidate point, and check if $(x, y) \in$ P-256 for some y. If the candidate point is not on the curve, we increase the counter and repeat the above process. To decode a point, we discard the counter byte, then interpret the remaining x-coordinate as the message. On the P-256 curve, this encoding yields a capacity of 30 bytes per group element, as one padding byte is needed in the most significant bits of the coordinate. Note that the above issue does not concern the OPRF part of our protocol (which computes *nym* and *pad*), where the *uid*s are hashed to the group and never decoded.

[3] Since our AES keys are used only once, the weaker IND-EAV security suffices here. By not transmitting a new IV per ciphertext, we save 16 bytes per value; this amounts to a network cost reduction of $\approx 4\%$.

Table 1: Total theoretical per-party costs for π_{MPPJ} for n sources with m rows per table

Metric	Computation ($\mathbb{G}$ exp)			Network ($\mathbb{G}$ el.)	
Parties	$\mathcal{S}_i$	$\mathcal{H}$	$\mathcal{R}$	$\mathcal{S} \to \mathcal{H}$	$\mathcal{H} \to \mathcal{R}$
Cost	$m \cdot 4$	$n \cdot m \cdot 14$	$n \cdot m \cdot 4$	$m \cdot 4$	$n \cdot m \cdot 8$

Addressing the limit on the size of values can be done in a straightforward way, by splitting the attribute into chunks of 30 bytes blocks and using ElGamal as a block-cipher. However, this results in a large overhead: each block needs to be rerandomized individually and requires two group elements to be transmitted. Instead, we propose an alternative method in Section 6.1, which makes the cost associated to public-key operations in π_{MPPJ} independent of the attribute size. As this approach reduces the problem of arbitrary-sized attributes to that of a fixed-sized one, we assume for the rest of this analysis that all *vals* fit the 30-bytes constraint.

5.2 Theoretical Analysis

We first analyze the most impactful cost factors of our protocol: the number of exponentiations and transmitted group elements per party. Since the rows are processed independently in our protocol (see Figure 4), we can obtain the total costs by analyzing the cost of processing a single row. Each $\mathcal{S}_i$ performs two ElG.Enc encryptions per table row, which results in 4 exponentiations and 4 transmitted group elements. $\mathcal{H}$ then performs 3 ElG.RExp (each costing 4 exponentiations) and a PKE.ReRnd (costing 2 exponentiation) for a total of 14 exponentiations per row. $\mathcal{R}$ downloads 4 ElG ciphertexts and decrypt them, which costs 4 exponentiations. Table 1 reports on the total cost of the protocol, which corresponds to scaling the per-row costs by the number of processed rows. The theoretical analysis match our efficiency requirements of Section 2: the cost for the input parties is only linear in the database row count m, and independent of the number of parties n.

Latency. However, this total cost does not reflect the latency of our protocol in multi-core architectures. This is because the computation can be conveniently parallelized over the rows. As a result, latency of our protocol decreases linearly with the number of available cores. Moreover, these operations can, for the most part, be pipelined: the source can send each row as soon as it is encrypted, and the receiver can decrypt each incoming row. Although the helper can also pipeline and parallelize most of its computation cost, we observe that it is necessary that the obfuscated join J' is only transmitted (with its entries in random order) once fully computed. As a result of these optimizations, the concrete latency of the protocol on multi-core architectures is likely to be dominated by the network-transmission delays.

5.3 Implementation & Experimental Evaluation

To evaluate the concrete performance of the π_{MPPJ} protocol, we implement it and measure the latency and communication for real-world datasets. We use Go version 1.24 for our implementation. HKDF and AES are part of the Go standard library, and we use the CIRCL library [15] for the curve P-256 implementation. We provide the implementation as an open-source repository[4]. Our experimental evaluation consists of two experiments, presented in Sections 5.3.1 and 5.3.2. In the first experiment, we compare our protocol against a generic-MPC approach for small databases, and ask whether it provides a sufficient performance advantage. In the second experiment, we evaluate run our protocol on larger databases and demonstrate its scalability.

Table 2: Comparison of the π_{MPPJ} protocol to MP-SPDZ (ATLAS model) for 2 sources with m rows per table.

m	MP-SPDZ/ATLAS		π_{MPPJ} (% of MP-SPDZ/ATLAS)	
	Time (s)	Comm (MiB)	Time (s)	Comm (MiB)
10^1	0.12	1.06	0.18 (150.478%)	0.01 (0.744%)
10^2	10.18	90.73	0.21 (2.067%)	0.08 (0.087%)
10^3	1027.97	9072.27	0.49 (0.048%)	0.79 (0.009%)

5.3.1 Comparison with Generic MPC. Our first goal is to establish the relevance of our special-purpose protocol by comparing its performance to a generic MPC framework. Specifically, we compare to the ATLAS protocol [16] in the semi-honest model, implemented in the MP-SPDZ library [20]. We run all the parties on a `c5.4xlarge` AWS instance, running over an Intel Xeon Platinum 8275CL (3 GHz, 16 threads) CPU and 32GB of RAM, over the `localhost` interface.

There are a few caveats to consider in the comparison: First, as observed in Section 3, such a protocol does not satisfy our *one-way flow* requirement, as it requires several rounds of direct, party-to-party communication. Second, although the generic MPC achieves an optimal (empty) leakage in theory, implementing the shuffling results in impractical costs. Hence, our MPC program leaks the intersecting *uids*' indices, whereas our solution does not. Finally, due to the cost of generic MPC, we limit the comparison to two parties and relatively small datasets. The results for two data sources are presented in Table 2, which reports the overall end-to-end latency and the total network traffic for both protocols.

We observe that the main distinguishing factor is the network cost: while the baseline quickly reaches impractical values, the π_{MPPJ} has a low communication cost, which only grows linearly. Although the MPC approach outperforms π_{MPPJ} for the smallest dataset of size 10 in terms of latency, our experiment does not account for real-world transmission delays, which would reduce this advantage. Reaching the relatively small table size of 10^3, π_{MPPJ} outperforms the baseline in both metrics by several orders of magnitude. Since the network cost tends to be the determining factor in real deployments, we conclude that π_{MPPJ} provides an advantage compared to generic MPC.

5.3.2 End-to-End Benchmarks. Our second goal is to ensure that the π_{MPPJ} protocol has practical performance also for large datasets. For this experiment, we account for network-related latencies by running the parties on two distinct `c5.4xlarge` AWS instances, running over an Intel Xeon Platinum 8275CL (3 GHz, 16 threads) CPU and 32GB of RAM. One instance is shared between the sources (each limited to 4 concurrent threads) and the receiver, the other one runs the helper. The parties communicate over the local-area network of the AWS zone, which provides a 4.97 Gbits/sec link.

[4] https://github.com/hpicrypto/mppj-artifacts

Table 3: Benchmarks of the π_{MPPJ} protocol. The intersection is always 80 % of the row count (m).

n	Time in seconds					Total Communication in MiB (Percentage of IDCloak)				
	$m=1353$	$m=1700$	$m=19735$	$m=45211$	$m=253680$	$m=1353$	$m=1700$	$m=19735$	$m=45211$	$m=253680$
2	0.59	0.68	5.36	11.95	66.43	1.07 (59%)	1.34 (15%)	15.55 (60%)	35.61 (98%)	199.83 (83%)
3	0.78	0.92	7.83	17.60	98.24	1.60 (42%)	2.01 (10%)	23.32 (39%)	53.42 (64%)	299.75 (54%)
4	0.95	1.14	10.28	23.25	130.13	2.13 (35%)	2.68 (8%)	31.09 (30%)	71.23 (49%)	399.67 (41%)
5	1.12	1.36	12.76	28.94	162.24	2.66 (31%)	3.35 (6%)	38.86 (24%)	89.04 (40%)	499.58 (33%)
6	1.32	1.57	15.24	34.69	194.65	3.20 (28%)	4.02 (5%)	46.64 (21%)	106.84 (34%)	599.50 (28%)

Baseline. Since this is the first standalone MPPJ protocol, we have no equivalent baseline to directly compare against. However, we can use one of the latest works on *private-join-compute* (PJC) that enables replacing the computation part of the protocol with a simple reconstruction round. To our knowledge, the only work that provide this functionality for n parties is IDCloak [11], so we use it as a baseline. However, there are a number of limitations to this comparison: First, IDCloak requires multiple rounds (contradicting *one-way flow*) and leaks the size of the join to the parties (contradicting *source privacy*). Second, we observe that [11] is a pre-print at the time of writing, so there might be discrepancies with the final version (for both the security claims and performance metric). Lastly, we could not execute the code artifact of IDCloak and reproduce their result on our equipment, making side-by-side latency comparison meaningless. We therefore limit the comparison to the network cost, which is independent of the hardware. We used datasets with the same row count m as IDCloak, and the same test-data generation strategy: the datasets are partitioned *vertically*, among the n parties and by replacing 20% of each original datasets by with fake rows, to simulate some out-of-join rows.

Our performance results are shown in Table 3. In general, we observe that deploying our protocol is practical even for large problem sizes. E.g., even the most extensive dataset, which contains about 250,000 rows per party, when run with 6 sources completes in under 3.5 minutes. In comparison to IDCloak, π_{MPPJ} has a consistently lower communication complexity and scales more favorably to larger numbers of parties. Hence, factoring the transmission-related delays would likely result in an advantage in latency for π_{MPPJ} (unfortunately, it is not possible to execute the IDCloak code over a real network with the artifact available at the time of writing). We conclude that π_{MPPJ} is scalable and provides a realistic solution for large-scale join operations. Although the latency could be too high for interactive use, we do not expect MPPJ to be used in highly interactive settings, because if the sensitive nature of its ideal functionality. Rather, more realistic scenarios would be periodic centralizations of datasets.

6 Extensions

We now provide two extensions of our protocol. The first one relaxes the assumption that the values fit a single group element (Section 6.1), efficiently and without any extra leakage. The second one extends our protocol to *threshold joins*, in which a value is included in the join if its *uid* is present in at least a threshold $t < n$ of the tables (Section 6.2). These extensions illustrate the flexibility of our approach, and its applicability to other type of joins. As another example, we show that it is straightforward to extend our protocol to the *multi-key join* setting of DPMC [28] in Section F.

6.1 Large- and Variable-sized Values

Our π_{MPPJ} instantiation sets requirements on the size of the values, for both correctness and security reasons. For correctness, our instantiation (see Section 5.1) requires an efficiently reversible encoding of values as group elements, which sets a limit of 30 bytes. Although a naive approach could be to chunk the values into an appropriate size and encrypt those chunks separately, this would be suboptimal. First, because each encrypted chunk is expanded to an ElG ciphertext, which results in high costs during rerandomization of individual ciphertexts and during transmission, as each chunk is expanded to two group elements. Second, this cost would be further increased due to the necessity of padding all values to an equal length, to avoid introducing extra leakage. This is because variable-size values would enable a corrupt $\mathcal{R}$ and $\mathcal{S}_i$ to link the encrypted values (outside J) to their source. We now propose an extension of π_{MPPJ} that efficiently addresses this issue.

The core idea is to let the sources provide the values as an SE-encrypted table (which is not subject to the size constraint), in which the encryption key is fresh and value-specific. Then, the parties execute the regular π_{MPPJ} protocol (as a black-box), but over the value-specific key instead of the values themselves. Concretely, the extension adds a pre- and post-processing to the Prepare and Extract algorithms, respectively.

For Prepare, the pre-processing run by each source $\mathcal{S}_i$ is as follows: First, for each *unique* value $val_j \in T_i$ (in a random order), $\mathcal{S}_i$ pads it to the fixed length as val'_j, generates a fresh key k_j, and appends $C_{val_j} \leftarrow \mathsf{SE.Enc}(k_j, val'_j)$ to a ciphertext list $Ctbl_i$. If the table contains duplicate values, it inserts an equivalent number of dummy values at random positions in $Ctbl_i$. Second, the source replaces each $(uid_j, val_j) \in T_i$ by $(uid_j, o_j || k_j)$, where k_j is the unique symmetric key for value val_j and o_j is the offset of C_{val_j} in $Ctbl_i$. Lastly, the source runs the original Prepare on the modified table, and sends the output along with $Ctbl_i$ to the helper. The Convert algorithm is not modified, except for collecting $Ctbl_1, ..., Ctbl_n$ and forwarding them to the receiver.

For Extract, the post-processing corresponds to the receiver looking up the actual value for each element in the join. The index i of the source's list $Ctbl_i$ is obtained from the original protocol, and the offset o_j and decryption key k_j are obtained from parsing the "values" in the join. We provide the concrete algorithms and discuss the security of this extension in Section E.

6.2 Threshold Join

The second extension provides support for threshold joins, in which a tuple is included in the final join if its *uid* appears in at least a set threshold $t < n$ tables (where t is a public parameter). The overall goal is similar to existing works on threshold MPSI protocols, such as [5]. However, as for plain MPSI, this requires different techniques (Section 3). Although this extension can be formulated as a minor modification of the π_{MPPJ} protocol, it comes at the cost of a slightly increased leakage (which we specify after describing the extension). Nonetheless, threshold-joins open a broad variety of applications. For example, consider the case of a federal state willing to ensure that its citizens are adequately taxed even when some of them are taxed across multiple states. Here, a 2-out-of-n MPPJ could be run among the federated states, to gather tax data from such citizens and ensure adequate taxing without disclosing any data for citizens registered in a single state, and without disclosing any identities.

The core idea is to perform the sharing of k_{pad} (in Step 4 of **Setup**) with Shamir sharing instead of additive sharing. Specifically, $\mathcal{H}$ chooses a random degree-t-1 polynomial $f(x)$ over q, so that $f(0) = k_{pad}$, and sets $k_{spad_i} = f(i)$ for $i \in [n]$. Then, the obfuscation Step 3 unfolds as before with one exception: when processing table T'_i, $\mathcal{H}$ appends i to each tuple added to J'. In the Extract algorithm (in Step 5), the receiver reconstructs the $pad_i = \mathsf{H}_{\mathbb{G}}(uid)^{k_{pad}} = \mathsf{H}_{\mathbb{G}}(uid)^{f(0)}$ values by performing the Lagrange interpolation at the exponent. Specifically, for every group of at least t tuples with a matching pseudonym nym_l, $\mathcal{R}$ computes pad_l from t associated $spad_1, ..., spad_t$ as $pad_l \leftarrow \prod_{j=1}^{t} spad_j^{\lambda_j}$ (where λ_j is the Lagrange interpolation coefficient for $spad_j$), and proceeds as before for the decryption of the value. Note that this requires the source-table identifier i (to map each $spad_i$ to the appropriate interpolation coefficient), which is why $\mathcal{H}$ appends it to each tuple in Convert. We provide the amended protocol in Section G.

Additional Leakage. The extension requires the helper to embed a plaintext identifier for the source table in each row of J'. This is different to the non-threshold protocol, where this identifier is encrypted in *Cval* and cannot be decrypted unless it is part of the join. As a result, the protocol now leaks the sources of each row in the *obfuscated join* J'. In terms of leakage, this implies that the adversary now learns which sources are involved in a partial join (instead of only learning the partial joins' cardinalities in the non-threshold protocol). We observe that this leakage is still reasonable in many scenarios, e.g., it is even trivial in settings with $t = 2$ (i.e., there are no partial joins) such as our tax-computation example.

7 Discussion & Conclusion

Our protocol provides a high-efficiency and low-leakage multi-party private join protocol, but also leaves interesting open-questions and potential avenues for future works.

Attacks on the Functionality. This work focuses on the cryptographic aspect of realizing MPPJ and, by definition, does not address the issues related to attacks exploiting the functionality. It is however important that we discuss these attacks in our context. For instance membership inference attacks can be mounted even for simple outputs, such as PSI-cardinality protocols [17]. Recently, these attacks have been extended to the private-join-and-compute setting, for inner product [14].

However, these attacks usually assume a strong adversary model. They typically assume that the adversary manipulates its inputs to the protocol (i.e., is malicious) and is able to adaptively trigger many protocol executions (while often assuming that the honest sources have fixed inputs). As such, these attacks are often thwarted by even simple (yet intrinsically application-specific) countermeasures. For example, the authors of [17] argue that limiting the number of protocol runs can be a sufficient defense. These attacks are nonetheless an important reminder that one should evaluate the functionality-induced risk even when deploying privacy-preserving protocols.

Reidentification Attacks. Specifically for our *anonymous join* setting, it is important to consider the risk of reidentification from the joined values. Indeed, although our protocol does not reveal any information about the *uid*s through its leakage, the values in the inner join might already be sufficient for some colluding $\mathcal{R}$ and $\mathcal{S}_i$s to reidentify the user behind a row. As an extreme example, consider the case of the *uid* being itself included in *val*. We observe that this is actually how one would implement a non-anonymous join from our protocol, while preserving helper-obliviousness. In the malicious setting, a source might even rely on steganography techniques to embed identifiers in the values. Hence, in the applications for which reidentification must be prevented, auxiliary mechanisms and safeguards needs to be implemented, e.g., ensuring that all inputs satisfy k-anonymity before running the join.

7.1 Conclusion

We have introduced a protocol for multi-party private joins and proved it secure in the simple-UC model. To the best of our knowledge, this is the first proposition of a *standalone* n-party private join protocol. Although our protocol leaks slightly more information than generic MPC techniques (i.e., the cardinality of the partial joins), it surpasses their efficiency by several orders of magnitude. Moreover, our protocol significantly reduces the communication overhead compared to a current MPPJ(-and-compute) protocol, and imposes lower integration constraints. Namely, for n databases with m entries, it only requires a single upload of size $O(m)$ per source, and a single download of size $O(n \cdot m)$ for the receiver; this is the same communication complexity as a plaintext system. Furthermore, its local operations greatly benefit from parallelization and pipelining techniques, which enable scaling the protocol to large numbers of parties and database sizes. We demonstrate this experimentally by providing an open-source prototype implementation and executing it over a realistic infrastructure. Finally, one of the main strengths of our protocol is its conceptual simplicity: it only requires minor extensions to implement more advanced types of joins such as *threshold-* or *multi-key*-joins.

Acknowledgments

Andrey Sidorenko and Christian Mouchet received support from the German Federal Ministry of Research, Technology and Space (BMFTR) through funding of the ATLAS project, under reference number 16KISA037. Andrey Sidorenko was further supported by the HPI Research School on Systems Design.

References

[1] 2025. European Health Data Space Regulation (EHDS). https://health.ec.europa.eu/ehealth-digital-health-and-care/european-health-data-space-regulation-ehds_en. Accessed on 31 August 2025.

[2] 2025. Forschungsdatenzentrum Gesundheit. https://www.forschungsdatenzentrum-gesundheit.de/. Accessed on 31. August 2025.

[3] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. 2012. Multiparty Computation with Low Communication, Computation and Interaction via Threshold FHE. In *Advances in Cryptology – EUROCRYPT 2012 (Lecture Notes in Computer Science, Vol. 7237)*, David Pointcheval and Thomas Johansson (Eds.). Springer, Berlin, Heidelberg, 483–501. https://doi.org/10.1007/978-3-642-29011-4_29

[4] Feng Bao, Robert H. Deng, and Huafei Zhu. 2003. Variations of Diffie-Hellman Problem. In *ICICS 03: 5th International Conference on Information and Communication Security (Lecture Notes in Computer Science, Vol. 2836)*, Sihan Qing, Dieter Gollmann, and Jianying Zhou (Eds.). Springer, Berlin, Heidelberg, 301–312. https://doi.org/10.1007/978-3-540-39927-8_28

[5] Aslı Bay, Zekeriya Erkin, Jaap-Henk Hoepman, Simona Samardjiska, and Jelle Vos. 2021. Practical multi-party private set intersection protocols. *IEEE Transactions on Information Forensics and Security* 17 (2021), 1–15.

[6] Mihir Bellare, Rishabh Ranjan, Doreen Riepel, and Ali Aldakheel. 2024. The Concrete Security of Two-Party Computation: Simple Definitions, and Tight Proofs for PSI and OPRFs. In *Advances in Cryptology – ASIACRYPT 2024, Part VI (Lecture Notes in Computer Science, Vol. 15489)*, Kai-Min Chung and Yu Sasaki (Eds.). Springer, Singapore, 328–362. https://doi.org/10.1007/978-981-96-0938-3_11

[7] Abhishek Bhowmick, Dan Boneh, Steve Myers, Kunal Talwar, and Karl Tarbe. 2021. The Apple PSI System. *Apple, Inc., Tech. Rep* (2021).

[8] Ran Canetti. 2001. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *42nd Annual Symposium on Foundations of Computer Science*. IEEE Computer Society Press, 136–145. https://doi.org/10.1109/SFCS.2001.959888

[9] Ran Canetti, Asaf Cohen, and Yehuda Lindell. 2015. A Simpler Variant of Universally Composable Security for Standard Multiparty Computation. In *Advances in Cryptology – CRYPTO 2015, Part II (Lecture Notes in Computer Science, Vol. 9216)*, Rosario Gennaro and Matthew J. B. Robshaw (Eds.). Springer, Berlin, Heidelberg, 3–22. https://doi.org/10.1007/978-3-662-48000-7_1

[10] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. 2018. Labeled PSI from Fully Homomorphic Encryption with Malicious Security. In *ACM CCS 2018: 25th Conference on Computer and Communications Security*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM Press, 1223–1237. https://doi.org/10.1145/3243734.3243836

[11] Shuyu Chen, Guopeng Lin, Haoyu Niu, Lushan Song, Chengxun Hong, and Weili Han. 2025. IDCloak: A Practical Secure Multi-party Dataset Join Framework for Vertical Privacy-preserving Machine Learning. arXiv:2506.01072 [cs.CR] https://arxiv.org/abs/2506.01072

[12] Kelong Cong, Radames Cruz Moreno, Mariana Botelho da Gama, Wei Dai, Ilia Iliashenko, Kim Laine, and Michael Rosenberg. 2021. Labeled PSI from Homomorphic Encryption with Reduced Computation and Communication. In *ACM CCS 2021: 28th Conference on Computer and Communications Security*, Giovanni Vigna and Elaine Shi (Eds.). ACM Press, 1135–1150. https://doi.org/10.1145/3460120.3484760

[13] Taher ElGamal. 1985. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *IEEE Transactions on Information Theory* 31, 4 (1985), 469–472. https://doi.org/10.1109/TIT.1985.1057074

[14] Francesca Falzon and Tianxin Tang. 2025. Learning from Functionality Outputs: Private Join and Compute in the Real World. *Cryptology ePrint Archive* (2025).

[15] Armando Faz-Hernandez and Kris Kwiatkowski. 2019. *Introducing CIRCL: An Advanced Cryptographic Library*. Cloudflare. Available at https://github.com/cloudflare/circl. v1.6.0 Accessed Jan, 2025.

[16] Vipul Goyal, Hanjun Li, Rafail Ostrovsky, Antigoni Polychroniadou, and Yifan Song. 2021. ATLAS: Efficient and Scalable MPC in the Honest Majority Setting. In *Advances in Cryptology – CRYPTO 2021, Part II (Lecture Notes in Computer Science, Vol. 12826)*, Tal Malkin and Chris Peikert (Eds.). Springer, Cham, Virtual Event, 244–274. https://doi.org/10.1007/978-3-030-84245-1_9

[17] Xiaojie Guo, Ye Han, Zheli Liu, Ding Wang, Yan Jia, and Jin Li. 2022. Birds of a Feather Flock Together: How Set Bias Helps to Deanonymize You via Revealed Intersection Sizes. In *USENIX Security 2022: 31st USENIX Security Symposium*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 1487–1504.

[18] Ilia Iliashenko and Vincent Zucca. 2021. Faster homomorphic comparison operations for BGV and BFV. *Proceedings on Privacy Enhancing Technologies* 2021, 3 (July 2021), 246–264. https://doi.org/10.2478/popets-2021-0046

[19] Mihaela Ion, Ben Kreuter, Ahmet Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, Mariana Raykova, David Shanahan, and Moti Yung. 2020. On Deploying Secure Computing: Private Intersection-Sum-with-Cardinality. In *2020 IEEE European Symposium on Security and Privacy*. IEEE Computer Society Press, 370–389. https://doi.org/10.1109/EuroSP48549.2020.00031

[20] Marcel Keller. 2020. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *ACM CCS 2020: 27th Conference on Computer and Communications Security*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM Press, 1575–1590. https://doi.org/10.1145/3372297.3417872

[21] Jaehyung Kim. 2025. Efficient Homomorphic Integer Computer from CKKS. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2025, 4 (Sep. 2025), 873–898. https://doi.org/10.46586/tches.v2025.i4.873-898

[22] Neal Koblitz. 1987. Elliptic curve cryptosystems. *Mathematics of computation* 48, 177 (1987), 203–209.

[23] Anja Lehmann. 2019. ScrambleDB: Oblivious (Chameleon) Pseudonymization-as-a-Service. *Proceedings on Privacy Enhancing Technologies* 2019, 3 (July 2019), 289–309. https://doi.org/10.2478/popets-2019-0048

[24] Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Karn Seth, and Ni Trieu. 2021. Private Join and Compute from PIR with Default. In *Advances in Cryptology – ASIACRYPT 2021, Part II (Lecture Notes in Computer Science, Vol. 13091)*, Mehdi Tibouchi and Huaxiong Wang (Eds.). Springer, Cham, 605–634. https://doi.org/10.1007/978-3-030-92075-3_21

[25] Payman Mohassel, Peter Rindal, and Mike Rosulek. 2020. Fast Database Joins and PSI for Secret Shared Data. In *ACM CCS 2020: 27th Conference on Computer and Communications Security*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM Press, 1271–1287. https://doi.org/10.1145/3372297.3423358

[26] Christian Mouchet, Sylvain Chatel, Apostolos Pyrgelis, and Carmela Troncoso. 2024. Helium: Scalable MPC among Lightweight Participants and under Churn. In *ACM CCS 2024: 31st Conference on Computer and Communications Security*, Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie (Eds.). ACM Press, 3038–3052. https://doi.org/10.1145/3658644.3670346

[27] Christian Mouchet, Juan Ramón Troncoso-Pastoriza, Jean-Philippe Bossuat, and Jean-Pierre Hubaux. 2021. Multiparty Homomorphic Encryption from Ring-Learning-with-Errors. *Proceedings on Privacy Enhancing Technologies* 2021, 4 (Oct. 2021), 291–311. https://doi.org/10.2478/popets-2021-0071

[28] Dimitris Mouris, Daniel Masny, Ni Trieu, Shubho Sengupta, Prasad Buddhavarapu, and Benjamin M. Case. 2024. Delegated Private Matching For Compute. *Proceedings on Privacy Enhancing Technologies* 2024, 2 (April 2024), 49–72. https://doi.org/10.56553/popets-2024-0040

[29] Arjun Narayan and Andreas Haeberlen. 2012. {DJoin}: Differentially private join queries over distributed databases. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 149–162.

[30] Arvind Narayanan and Vitaly Shmatikov. 2007. How To Break Anonymity of the Netflix Prize Dataset. arXiv:cs/0610105 [cs.CR] https://arxiv.org/abs/cs/0610105

[31] Adi Shamir. 1979. How to Share a Secret. *Communications of the Association for Computing Machinery* 22, 11 (Nov. 1979), 612–613. https://doi.org/10.1145/359168.359176

[32] Jelle Vos, Mauro Conti, and Zekeriya Erkin. 2024. SoK: Collusion-resistant Multi-party Private Set Intersections in the Semi-honest Model. In *2024 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 465–483. https://doi.org/10.1109/SP54263.2024.00079

[33] Cong Zhang, Weiran Liu, Bolin Ding, and Dongdai Lin. 2023. Efficient Private Multiset ID Protocols. In *ICICS 23: 25th International Conference on Information and Communication Security (Lecture Notes in Computer Science, Vol. 14252)*, Ding Wang, Moti Yung, Zheli Liu, and Xiaofeng Chen (Eds.). Springer, Singapore, 351–369. https://doi.org/10.1007/978-981-99-7356-9_21

A Security Definitions

This appendix provides complementary definitions of standard notions. The rerandomization definition (Definition 3) uses the experiment in Experiment 1. All the other experiments are defined in Figure 5.

Definition 3. PKE *is ReRand-secure if, for any PPT adversary* $\mathcal{A}$*, there exists a negligible function* $negl(\lambda)$ *such that:*

$$Pr\left[Exp^{ReRand}_{\mathsf{PKE},\mathcal{A}}(\lambda) = 1\right] \leq \frac{1}{2} + negl(\lambda).$$

Experiment 1 $\mathrm{Exp}^{ReRand}_{\mathsf{PKE},\mathcal{A}}(\lambda)$

$(sk, pk) \leftarrow \mathsf{KGen}(1^\lambda), \quad b \xleftarrow{\$} \{0,1\}$
$(c, state) \leftarrow \mathcal{A}(1^\lambda, sk, pk)$
if $\mathsf{Dec}(sk, c) = \perp$ **return** 0
$c_0 \leftarrow \mathsf{Enc}(pk, \mathsf{Dec}(sk, c)), \quad c_1 \leftarrow \mathsf{ReRnd}(pk, c)$
$b' \leftarrow \mathcal{A}(c_b, state)$
return $b' = b$

$\underline{\mathrm{Exp}^{IND-EAV}_{\mathsf{SE},\mathcal{A}}(\lambda)}$
1: $k \leftarrow \mathsf{KGen}(1^\lambda)$
2: $(m_0, m_1, \mathrm{st}) \leftarrow \mathcal{A}(1^\lambda)$
3: $b \xleftarrow{\$} \{0,1\}$
4: $c_b \leftarrow \mathsf{Enc}(k, m_b)$
5: $b' \leftarrow \mathcal{A}(c_b, \mathrm{st})$
6: **if** $b' = b$
7: **return** 1
8: **else return** 0

$\underline{\mathrm{Exp}^{IND-CPA}_{\mathsf{PKE},\mathcal{A}}(\lambda)}$
1: $(sk, pk) \leftarrow \mathsf{KGen}(1^\lambda)$
2: $(m_0, m_1, \mathrm{st}) \leftarrow \mathcal{A}(1^\lambda, pk)$
3: $b \xleftarrow{\$} \{0,1\}$
4: $c_b \leftarrow \mathsf{Enc}(pk, m_b)$
5: $b' \leftarrow \mathcal{A}(c_b, \mathrm{st})$
6: **if** $b' = b$
7: **return** 1
8: **else return** 0

$\underline{\mathrm{Exp}^{DDH}_{\mathbb{G},\mathcal{A}}(\lambda)}$
1: $x, y, r \xleftarrow{\$} \mathbb{Z}_q$
2: $b \xleftarrow{\$} \{0,1\}$
3: **if** $b = 0 : z \leftarrow r$
4: **else** $z = x \cdot y$
5: $b' \leftarrow \mathcal{A}(g, g^x, g^y, g^z)$
6: **if** $b' = b$
7: **return** 1
8: **else return** 0

$\underline{\mathrm{Exp}^{DDDH}_{\mathbb{G},\mathcal{A}}(\lambda)}$
1: $x, y, r \xleftarrow{\$} \mathbb{Z}_q$
2: $b \xleftarrow{\$} \{0,1\}$
3: **if** $b = 0 : z \leftarrow r$
4: **else** $z = \frac{x}{y} \cdot y$
5: $b' \leftarrow \mathcal{A}(g, g^x, g^y, g^z)$
6: **if** $b' = b$
7: **return** 1
8: **else return** 0

$\underline{\mathrm{Exp}^{IND-PRF}_{F,\mathcal{A}}(\lambda)}$
1: $k \xleftarrow{\$} K$
2: $f_0(x) \xleftarrow{\$} \{f | f : X \to Y\}$
3: $f_1(x) \leftarrow F(k, x)$
4: $b \xleftarrow{\$} \{0,1\}$
5: $b' \leftarrow \mathcal{A}^{f_b(\cdot)}(1^\lambda)$
6: **return** $b' = b$

Figure 5: Security experiments for SE IND-EAV, PKE IND-CPA, and $\mathbb{G}$ DDH/DDDH.

Definition 4. *We call a* PKE *scheme* Π *IND-CPA-secure if there exists a negligible function* $negl(\lambda)$ *for any PPT adversary* $\mathcal{A}$ *with:*

$$Pr\left[Exp^{IND-CPA}_{\Pi,\mathcal{A}}(\lambda) = 1\right] \leq \frac{1}{2} + negl(\lambda).$$

Definition 5. *We call a* SE*, scheme* Π *IND-EAV-secure if there exists a negligible function* $negl(\lambda)$ *for any PPT adversary* $\mathcal{A}$ *with:*

$$Pr\left[Exp^{IND-EAV}_{\Pi,\mathcal{A}}(\lambda) = 1\right] \leq \frac{1}{2} + negl(\lambda).$$

Definition 6. *The* $Y \in \{DDH, DDDH\}$ *problem is hard in a group* $\mathbb{G}$ *with prime order* q *and generator* g*, if there exists a negligible function* $negl(\lambda)$ *for any PPT adversary* $\mathcal{A}$ *with:*

$$|Pr\left[Exp^{Y}_{\mathbb{G},\mathcal{A}}(\lambda) = 1\right]| \leq \frac{1}{2} + negl(\lambda).$$

Definition 7. *We call a function* F *pseudorandom, if for there exists a negligible function* $negl(\lambda)$ *for any PPT adversary* $\mathcal{A}$ *with* $Pr\left[Exp^{IND-PRF}_{F,\mathcal{A}}(\lambda) = 1\right] \leq \frac{1}{2} + negl(\lambda)$.

B Simplified Protocol Pseudocode

A pseudocode description of the warm-up protocol of Section 4.2 is provided in Figure 6.

Setup
1: $\mathcal{R}$ generates $(esk, epk) \xleftarrow{\$} \mathsf{PKE.KGen}(1^\lambda)$

Preparation:Prepare Run by $\mathcal{S}_i$ with input T_i
1: $T_i' \leftarrow \left\{(uid_{i,j}, cval_{i,j}) \mid cval_{i,j} \leftarrow \mathsf{PKE.Enc}(epk, val_{i,j})\right\}$
2: Send T_i' to $\mathcal{H}$

Join:Convert Run by $\mathcal{H}$ after receiving all n of T_i'
1: Initialize K as empty table, $J' \leftarrow \emptyset$
2: **for** $i \in [n]$ and $uid_{i,j}, cval_{i,j} \in T_i'$ **do**
3: **if** $uid_{i,j} \notin \mathsf{UIDs}(K)$ **then**
4: $k_l \leftarrow \mathsf{SE.KGen}(1^\lambda)$
5: $k_{l,1}, \ldots, k_{l,n} \leftarrow \mathsf{SS.Share}(k_l, n, n)$
6: $K[uid_{i,j}] \leftarrow (k_l, k_{l,1}, \ldots, k_{l,n})$
7: $(k_l, k_{l,1}, \ldots, k_{l,n}) \leftarrow K[uid_{i,j}]$
8: $Cval_{i,j} \leftarrow \mathsf{SE.Enc}(k_l, cval_{i,j})$
9: Add $(uid_{i,j}, Cval_{i,j}, k_{l,i})$ to J'
10: Send J' to $\mathcal{R}$

Join:Extract Run by $\mathcal{R}$
1: Initialize $J \leftarrow \emptyset$
2: **for** uid_l that appears n times in J' **do**
3: Collect $Cval_{l,1}, \ldots, Cval_{l,n}, k_{l,1}, ..., k_{l,n}$ from $J'[uid_l]$
4: $k_l \leftarrow \mathsf{SS.Reconstruct}(k_{l,1}, ..., k_{l,n})$
5: **for** $i \in [n]$ **do**
6: $cval_{l,i} \leftarrow \mathsf{SE.Dec}(k_l, Cval_{l,i})$
7: $val_{l,i} \leftarrow \mathsf{PKE.Dec}(esk, cval_{l,i})$
8: Append $(val_{l,1}, ..., val_{l,n})$ to J
9: Output J

Figure 6: Simplified protocol for joins based on SE, SS

C Correctness

Correctness of π_{MPPJ} means that for data source inputs $T_1, \ldots, T_n$, π_{MPPJ} performs a join functionally equivalent to the ideal functionality $\mathcal{F}_{\mathsf{MPPJ}}$. It follows from the correctness of the underlying schemes (EIG, SE, PKE): A $nym = \mathsf{EIG.Dec}(bsk, \mathsf{EIG.Exp}(bpk, c, k_{nym})) = \mathsf{H}_{\mathbb{G}}(uid)^{k_{nym}}$, for $c = \mathsf{EIG.Enc}(bpk, \mathsf{H}_{\mathbb{G}}(uid))$, will be the same for each uid due to the correctness and homomorphism of EIG. This also follows analogously for the derived $pad = \mathsf{H}_{\mathbb{G}}(uid)^{k_{pad}}$ and $spad = \mathsf{H}_{\mathbb{G}}(uid)^{k_{spad}}$. pad can be correctly reconstructed from $spad_1, \ldots, spad_n$ using the correctness of additive secret sharing. Each value val can be recovered using $val = \mathsf{PKE.Dec}(esk, \mathsf{SE.Enc}(\mathsf{H}(s), c'))$, where $c' \leftarrow \mathsf{PKE.ReRnd}(epk, \mathsf{PKE.Enc}(epk, val)))$ due to the correctness of SE and PKE.

D Security Proof for Theorem 4.1

Now, to prove security, we must show that for every environment $\mathcal{Z}$ and adversary $\mathcal{A}$, there exists an efficient simulator Sim, such that $\mathcal{Z}$ cannot distinguish whether it is interacting with our π_{MPPJ} protocol or with Sim and $\mathcal{F}_{\mathsf{MPPJ}}$. The simulator Sim mimics all honest parties towards the real-world adversary $\mathcal{A}$ and interacts with $\mathcal{F}_{\mathsf{MPPJ}}$ in the role of the corrupt parties in the ideal world. By $\mathcal{P}_{\mathsf{sim}}$, we denote that the simulator plays the role of the honest party $\mathcal{P}$ towards the adversary in the real world. As we consider honest-but-curious adversaries, the corrupt parties are also run by

the simulator, but with $\mathcal{A}$ providing their inputs and receiving their outputs, secret keys as well as all internal values, and randomness. We denote such a simulated, passively corrupt party as $\mathcal{P}^*$.

We recall, that due to Sim playing the role of the adversary, it gets access to the following information: The table sizes $|T_i|$, the leakage about the sizes of the partial joins $\mathsf{Leak}_{\mathsf{PJS}}(T_1, \ldots, T_n) = \{ |X| \mid X \in PartJoins \}$, both from $\mathcal{F}_{\mathsf{MPPJ}}$. The ideal functionality $\mathcal{F}_{\mathsf{SMT}}$ is a secure communication channel, which leaks the message sizes of all messages sent though it, as well as the sender and recipient identifiers to the adversary. Lastly, the adversary always receives the public key *rpk*.

We now recap communication in the real protocol π_{MPPJ}: A data source $\mathcal{S}_i$ has input T_i, no protocol output and the view T_i', the prepared table that it sends. The helper $\mathcal{H}$ has no protocol input or output and has the view $T_1', \ldots, T_n'$ and J'. The receiver has no protocol input, its protocol output is J and its view is J'. The task of the simulator Sim is to create a plausible view for each party, out of that party's protocol inputs and outputs. In case of data source corruption, we consider all but one data source to be corrupt. W.l.o.g let $\mathcal{S}_n$ be the data source that is always honest.

Lastly, we assume for simplicity that all tables T_i have the same number of rows and that each *uid*, *nym*, and *val* can be represented by a single group element, and thus all have the same size. We distinguish the protocol steps and their cases based on the combinations of corrupt parties. We start with the relatively simple cases ($\mathcal{S}_i$ or $\mathcal{H}$ corrupt) first, and discuss the more challenging cases (where $\mathcal{R}$ is corrupt) later on.

D.1 Case $\mathcal{A} = \emptyset$ (Network Adversary)

For all protocol messages, Sim learns the lengths of T_i' or J' (which is the sum of all lengths $|T_i'|$) using the $\mathcal{F}_{\mathsf{SMT}}$ leakage. For the case of all $\mathcal{S}$, $\mathcal{H}$, and $\mathcal{R}$ being honest, Sim can rely on the ideal security of $\mathcal{F}_{\mathsf{SMT}}$ and does not need to simulate anything.

D.2 Case $\mathcal{A} = \{S_1, \ldots S_{n-1}\}$

For **Preparation**, the data source $\mathcal{S}_i^*$ receives $(\mathsf{PREP}, \mathsf{sid}, T_i, \mathcal{S}_i)$ from the environment $\mathcal{Z}$. It correctly blinds the hashed *uid*s using *bpk* and encrypts all *val*s using *epk*. $\mathcal{S}_i^*$ then discloses all the randomness used for this step to the $\mathcal{A}$ and then sends the blinded and encrypted table T_i' towards $\mathcal{H}$ using $(\mathsf{joinprep}, \mathsf{sid}, T_i')$ over $\mathcal{F}_{\mathsf{SMT}}$. Sim can simulate this process by submitting $(\mathsf{PREP}, sid, T_i, \mathcal{S}_i)$ to $\mathcal{F}_{\mathsf{MPPJ}}$. Since the corrupt party follows the protocol to create its view (T_i') based solely on its input (T_i), these views are indistinguishable.

As soon as $\mathcal{H}$ receives the prepared table in the real world, Sim sends $(\mathsf{PREP}, \mathsf{sid}, \mathcal{S}_i, OK)$ to $\mathcal{F}_{\mathsf{MPPJ}}$. This prompts the creation of the appropriate record in the ideal world.

D.3 Case $\mathcal{A} = \{\mathcal{H}\}$

The helper has no protocol inputs or outputs. It is only involved in **Join: Convert** and at the end of **Convert**. In the case of only the helper being corrupt Sim simulates $\mathcal{S}_i$ using a table of length $|T_i|$ consisting only of $(uid, val) = (1, 1)$ tuples, blinding *uid* using ElGamal and encrypting *val* using PKE with the receiver's public key for each row. "$\mathcal{S}_i$" sends this prepared table to $\mathcal{H}^*$, who proceeds as in the protocol. The received values are indistinguishable from real ones due to IND-CPA of ElGamal and PKE.

D.4 Case $\mathcal{A} = \{S_1, \ldots S_{n-1}, \mathcal{H}\}$

For the execution of $\mathcal{S}_i^*$ in **Preparation**, see the above case for $\mathcal{A} = \{S_1, \ldots S_{n-1}\}$. Since $\mathcal{H}^*$ operates on indistinguishable (due to IND-CPA) ciphertexts for honest data sources (see $\mathcal{A} = \{\mathcal{H}\}$), it is oblivious to the contents of the honest parties' T_i.

In **Join: Convert**, we need to compare the distributions of J in the ideal world and the real protocol, where $\mathcal{Z}$ knows the original *uid*s and the keys k_{nym}, k_{pad} used by $\mathcal{H}$. In the ideal world, the rows are shuffled, since there is no inherent ordering in the set. This is mimicked by the real world, where there is also an implicit shuffle. Thus, the adversary cannot gain an advantage here, even though it can compute *nym* and *pad* for the values it controls. The contents of J are also identical due to the correctness of the utilized schemes.

D.5 Case $\mathcal{A} = \{\mathcal{R}\}$

This is the most intricate case, since the converter has to construct a plausible J' sent by $\mathcal{H}_{\mathsf{sim}}$ from the real join J. To simulate **Preparation**, Sim blinds/encrypts $(uid, val) = (1, 1)$ tuples, as in $\mathcal{A} = \{\mathcal{H}\}$.

In the ideal world, Sim sends $(\mathtt{JOIN}, \mathsf{sid})$ to $\mathcal{F}_{\mathsf{MPPJ}}$, imitating the helper. The functionality replies with $(\mathtt{JOIN}, \mathsf{sid}, \ell, \mathcal{R}, OK)$, where $\ell = \mathsf{Leak}_{\mathsf{PJS}}(T_1, \ldots T_n)$. Sim now answers with $(\mathtt{JOIN}, \mathsf{sid}, OK)$. Since Sim plays the role of the corrupt $\mathcal{R}$ in the ideal world, it then receives the full join J as part of $(\mathtt{JOIN}, \mathsf{sid}, J)$. The simulator's goal is now to simulate the protocol message J' of **Join: Convert** based on the leakage ℓ, the table sizes $|T_1|, \ldots |T_n|$, and the final output J. To construct J', Sim proceeds differently for values in J and all other values.

Values in J: For each tuple = $\{(val_{1,j}, \ldots, val_{n,j})\} \in J$ in J, $\mathcal{H}_{\mathsf{sim}}$ first chooses a random pseudonym tuple $(nym_j, pad_j) \xleftarrow{\$} \mathbb{G}^2$. It computes all $cnym_{i,j}$ as fresh encryptions of nym_j under *bpk*. It computes $Cval_{i,j}$ as in the protocol, but starts with a fresh encryption of $cval_{i,j} \leftarrow \mathsf{PKE.Enc}(epk, val_{i,j})$. For $cs_{i,j}, cspad_{i,j}$ it deviates and freshly encrypts the expected outcome under *bpk*, i.e., $cs_{i,j}$ are generated as fresh encryption of $s_{i,j} \cdot pad_j$ and $cspad_{i,j}$ freshly encrypt $spad_{i,j}$. For the latter, $spad_{1,j}, \ldots spad_{n,j}$ are multiplicative shares of the padding pseudonym pad_j.
The ciphertexts $Cval_{i,j}$ are proper encryptions of $val_{i,j}$, but generated freshly instead from $cval_{i,j}$.
In summary, the only difference from a simulated J' tuple $(cnym_{i,j}, Cval_{i,j}, cs_{i,j}, cspad_{i,j})$ are the truly random nym_j, pad_j.

Values not in J: The simulator must also produce correct looking $(cnym_{i,j}, Cval_{i,j}, cs_{i,j}, cspad_{i,j})$ for J' for all values that are not in the join. For these tuples, the only information Sim has is the amount of partial joins and the overall size of input tables (the latter was received through $(\mathsf{PREP}, \mathsf{sid}, \mathcal{S}_i, |T_i|)$ from $\mathcal{F}_{\mathsf{MPPJ}}$). We therefore let $\mathcal{H}_{\mathsf{sim}}$ create dummy tuples $(cnym_{i,j}, Cval_{i,j}, cs_{i,j}, cspad_{i,j})$ where $Cval_{i,j}$ merely encrypts a dummy message. For the pseudonyms, it creates random $nym_j \xleftarrow{\$} \mathbb{G}$, which it now uses x times for $x \in \{|X| \mid X \in PartJoins\}$, i.e. x is a partial join size inferred from the leakage ℓ. This means that the same nym_j appears in its encrypted form $cnym_J$ in x tuples $(cnym_j, Cval_{i,j}, cs_{i,j}, cspad_{i,j})$. The ciphertexts $cs_{i,j}, cspad_{i,j}$ are replaced by fresh encryptions of random

group elements. Thus, the differences from a simulated J' tuple $(cnym_{i,j}, Cval_{i,j}, cs_{i,j}, cspad_{i,j})$ are the following: $Cval_{i,j}$ encrypts a random bitstring, each $nym_j, pad_j, spad_{i,j}$ are random group elements instead of being computed by a PRF, and $cs_{i,j}$ encrypts a random group element, instead of $pad \cdot s_j$.

We now have to argue that this simulated $J' = \{(cnym_{i,j}, Cval_{i,j}, cs_{i,j}, cspad_{i,j}), \ldots\}$ cannot be distinguished from a real one. First, let us denote an adversary's advantage of distinguishing the fresh encryptions using bpk from rerandomized ones as $Pr[\text{Exp}^{Game1}_{\pi_{\text{MPPJ}},\mathcal{A}}(\lambda) = 1]$. From now on, we consider the rerandomized ElG ciphertexts indistinguishable from the simulated ones. This means that we only need to analyze the far simpler items. We now distinguish the cases of whether a nym appears n times in J':

Case nym appears n times The decryption of an item in $tup \in J'$ in the real world yields:

$$tup_{real} = (\mathsf{H}_{\mathbb{G}}(uid_{i,j})^{k_{nym}}, Cval_{i,j}, \mathsf{H}_{\mathbb{G}}(uid_{i,j})^{k_{pad}} \cdot s, \mathsf{H}_{\mathbb{G}}(uid_{i,j})^{k_{spad_i}})$$

whereas the simulated one in this case contains

$$tup_{sim} = (nym_{i,j}, Cval'_{i,j}, pad_{i,j} \cdot s_{i,j}, spad_{i,j})$$

for random $nym, pad, spad_i \xleftarrow{\$} \mathbb{G}$, with $\Pi_{I=1}^{n}(spad_I) = pad$, since the nym appears n times. We denote an adversary's advantage of distinguishing the real PRF values nym and pad from random group elements as $Pr[\text{Exp}^{Game2}_{\pi_{\text{MPPJ}},\mathcal{A}}(\lambda) = 1]$, and thus replace the PRF values in the real world with random group elements, too. When $\mathcal{R}$ decrypts the real $Cval_{i,j}$ according to the honest protocol, it receives the rerandomized ciphertext $cval_{i,j}$. In the simulated J', $Cval'_{i,j}$ instead contains a fresh $cval'_{i,j}$. $Pr[\text{Exp}^{Game3}_{\pi_{\text{MPPJ}},\mathcal{A}}(\lambda) = 1]$ denotes the advantage of distinguishing these. We assume from now on that they are indistinguishable.

Case value not in J: In this case, a simulated tuple $tup \in J'$ contains

$$tup_{sim} = (nym_{i,j}, Cval'_{i,j}, a, b)$$

for random $a, b \xleftarrow{\$} \mathbb{G}$. We have already established, that the PRF value $\mathsf{H}_{\mathbb{G}}(uid_{i,j})^{k_{nym}}$ is indistinguishable to the simulated, random $nym_{i,j}$. To argue that b is indistinguishable from a group element, recall that in this case, at least one share is missing. Due to the perfect security of additive secret sharing, the missing share is this indistinguishable from a random group element. We now have to argue that a random $a \xleftarrow{\$} \mathbb{G}$ is indistinguishable from the real-world $\mathsf{H}_{\mathbb{G}}(uid_{i,j})^{k_{pad}} \cdot s$. For this, consider these two rows in the real world, where $\mathsf{H}_{\mathbb{G}}(uid_I) = h_I$ and $\mathsf{H}_{\mathbb{G}}(uid_I I) = h_I I$:

$$h_I^{k_{nym}}, s_I \cdot h_I^{k_{pad}}, h_I^{k_{spad_1}}, \ldots, h_I^{k_{spad_n}}$$
$$h_{II}^{k_{nym}}, s_{II} \cdot h_{II}^{k_{pad}}, h_{II}^{k_{spad_1}}, \ldots, h_{II}^{k_{spad_{n-1}}}$$

This represents the "best case scenario", as the adversary has all n share for row I and is missing a single share for row II, which, w.l.o.g. is the n-th share. Since $\mathsf{H}_{\mathbb{G}}(uid_x)^{k_{pad}} = \mathsf{H}_{\mathbb{G}}(uid_x)^{k_{spad_1}} \cdot \ldots \cdot \mathsf{H}_{\mathbb{G}}(uid_x)^{kspad_n}$, the adversary can divide all the known values $\mathsf{H}_{\mathbb{G}}(uid_x)^{k_{spad_i}}$ from $\mathsf{H}_{\mathbb{G}}(uid_x)^{k_{pad}}$ to obtain

$$h_I, s_I, h_I^{kspad_n}$$
$$h_{II}, s_{II} \cdot h_{II}^{k_{spad_n}}$$

In our simulation, we replace $h_{II}^{k_{pad}} \cdot s_{II}$ with a random a. s_I is completely independent of this, and can thus be discarded. We now rewrite $h_I = g^x$ and $h_{II} = g^y$ for x, y uniformly chosen in $\mathbb{G}$, as $\mathsf{H}_{\mathbb{G}}$ is a random oracle. We also replace k_{spad_n} with the uniform scalar $y \in \mathbb{Z}_q$ for notational consistency. The adversary has now the following information, apart from the generator $g \in pp$:

$$g^x, (g^x)^z$$
$$g^y, s_{II} \cdot (g^y)^z$$

We now define the probability of distinguishing $(g^y)^z$ from a uniform value, given $g^x, (g^x)^z, g^y$ as $Pr[\text{Exp}^{Game4}_{\pi_{\text{MPPJ}},\mathcal{A}}(\lambda) = 1]$. $(g^y)^z$ can then be replaced by a random value, and s_{II} is also chosen uniformly. This corresponds to a one-time pad, rendering their product indistinguishable form a random element $a \xleftarrow{\$} \mathbb{G}$. Note that when considering multiple rows with the same uid, it is the different s values that act as the one-time pad keys to protect a recurring $(g^y)^z$. Thus, given $Cval'_{i,j}$, a does not give the adversary any advantage. Distinguishing $Cval'_{i,j}$ that encrypts dummy data under a dummy key from $Cval_{i,j}$, which encrypts a value under an unknown key results in $Pr[\text{Exp}^{Game5}_{\pi_{\text{MPPJ}},\mathcal{A}}(\lambda) = 1]$ Note that the adversary cannot infer anything about $s_{t,j}$ in $\mathsf{H}_{\mathbb{G}}(uid_{i,j})^{k_{pad}} \cdot s_{t,j}$ from the encryption of $Cval_{i,j}$ due to the real SE key being the ROM hash of $s_{t,j}$. Thus, in the ideal or real world, the adversary has to attain the preimage, which is covered by Game 4.

Let Adv_{GameX} be the advantage of an adversary of winning the respective game (i.e. how much better than guessing it is). ElG is perfectly rerandomizable, $\text{Adv}_{Game1} = 0$. Game 2 has a setting close to DDH, but here, the adversary is tasked with distinguishing the following two distributions *simultaneously*, i.e. either the left-hand side is true at once, or the right-hand side:

$$(g, g^{k_{nym}}, \mathsf{H}_{\mathbb{G}}(uid), \mathsf{H}_{\mathbb{G}}(uid)^{k_{nym}}) \approx (g, g^{k_{nym}}, \mathsf{H}_{\mathbb{G}}(uid), nym)$$
$$(g, g^{k_{pad}}, \mathsf{H}_{\mathbb{G}}(uid), \mathsf{H}_{\mathbb{G}}(uid)^{k_{pad}}) \approx (g, g^{k_{pad}}, \mathsf{H}_{\mathbb{G}}(uid), pad)$$

With $k_{nym}, k_{pad} \xleftarrow{\$} \mathbb{Z}_q$ and $nym, k_{pad} \xleftarrow{\$} \mathbb{G}$. In the real protocol, the adversary does not receive $g^{k_{nym}}, g^{k_{pad}}$, which is easily fixed by omitting these values in a reduction. Since $\mathsf{H}_{\mathbb{G}}$ is a ROM, we can assume that $\mathsf{H}_{\mathbb{G}}(uid) = g^x$ for some $x \xleftarrow{\$} \mathbb{Z}_q$. Each of these equations can be phrased as a DDH tuple by itself, however, they are dependent on $\mathsf{H}_{\mathbb{G}}(uid)$ being the same. Fortunately, a fitting game-based definition, called DDH-MU, is described in [6]: The adversary may query oracles multiple times; one returns random bases, in our case $\mathsf{H}_{\mathbb{G}}(uid) \xleftarrow{\$} \mathbb{G}$, the other returns the generator with a random, unknown key, which in our case is $g^{k_{nym}}$ and $g^{k_{pad}}$, and the challenge is distinguishing, whether querying a base/key combination results in the real values $(\mathsf{H}_{\mathbb{G}}(uid)^{k_{nym}}, \mathsf{H}_{\mathbb{G}}(uid)^{k_{pad}})$, or randomly and independently picked group elements (nym, pad). As stated by Bellare et al. [6], DDH-MU is equivalent to standard DDH, implying that $\text{Adv}_{Game2} = \text{Adv}_{DDH}$.

Game 3 is equivalent to breaking ReRand for PKE, $\mathsf{Adv}_{Game2} = \mathsf{Adv}_{ReRand}$. Solving Game 4 is equivalent to the DDDH problem, which is formally defined in Figure 5. As proven in [4], DDDH is equivalent to solving DDH, so $\mathsf{Adv}_{Game4} = \mathsf{Adv}_{DDH}$. Lastly, solving Game 5, i.e. distinguishing one-time SE ciphertexts for a random key is equivalent to IND-EAV, $\mathsf{Adv}_{Game4} = \mathsf{Adv}_{IND-EAV}$. The sum of all these probabilities, Game 1 appearing thrice, since there are 3 ciphertexts, is thus:

$$
\begin{aligned}
&(3 \cdot \mathsf{Adv}_{Game1} + \mathsf{Adv}_{Game2} + \mathsf{Adv}_{Game3} + \mathsf{Adv}_{Game4} + \mathsf{Adv}_{Game5}) \\
&= \leq (3 \cdot 0 + \mathsf{Adv}_{DDH} + \mathsf{Adv}_{ReRand} + \mathsf{Adv}_{DDH} + \mathsf{Adv}_{IND-EAV}) \\
&= \leq (2 \cdot \mathsf{Adv}_{DDH} + \mathsf{Adv}_{ReRand} + \mathsf{Adv}_{IND-EAV})
\end{aligned}
$$

This is the advantage of $\mathcal{A}$ distinguishing the simulation from the real world for a given tuple in J'. The total distinguishing advantage $Pr[\mathrm{Exp}^{Total}_{\pi_{\mathsf{MPPJ}},\mathcal{A}}(\lambda) = 1]$, for n sources with m values each, is thus less than or equal to:

$$\frac{1}{2} + n \cdot m \cdot (2 \cdot \mathsf{Adv}_{DDH} + \mathsf{Adv}_{ReRand} + \mathsf{Adv}_{IND-EAV})$$

Due to IND-EAV of SE, ReRand of PKE and the DDH assumption, $\mathsf{Adv}_{IND-EAV}$, Adv_{ReRand}, and Adv_{DDH} are negligible. Thus, the final term is $Pr[\mathrm{Exp}^{Total}_{\pi_{\mathsf{MPPJ}},\mathcal{A}}(\lambda) = 1] \leq \frac{1}{2} + n \cdot m \cdot (2 \cdot 3 \cdot \mathrm{negl}(\lambda)) = \frac{1}{2} + \mathrm{negl}(\lambda)'$, for negligible functions $\mathrm{negl}(\lambda)$ and $\mathrm{negl}(\lambda)'$.

Thus, the adversary's advantage of distinguishing the simulated J' from a real one is negligible, due to IND-EAV of SE, the DDH assumption, $\mathsf{H}_{\mathbb{G}}/\mathsf{H}$ being random oracles and the rerandomization of ElG/PKE.

D.6 Case $\mathcal{A} = \{S_1, \ldots S_{n-1}, \mathcal{R}\}$

Here, the simulation is as in the previous case, but now the adversary has extra information on the values (uid, val) from $\mathcal{S}_i$, hence it can try to distinguish simulated pseudonyms nym and ciphertexts $cval$ based on its input. However, the val not in the intersection are protected by SE. The adversary also cannot distinguish a real nym of a uid from freely "invented" nyms, based on the pseudorandomness of $\mathsf{PRF}_{\mathsf{DH}}$. The fresh and real ciphertexts are also unlinkable, due to the rerandomization of PKE. Thus, the rest of the argumentation from $\mathcal{A} = \{\mathcal{R}\}$ applies.

E Large- and Variable-sized Values

Algorithms and Protocol. The preprocessing step that each source $\mathcal{S}_i$ has to perform for the extension is shown in Algorithm 1. The tables with the fixed-size values $\widehat{T}_i$ are used as inputs to π_{MPPJ}. The helper simply passes the every $Ctbl$ along to $\mathcal{R}$. $\mathcal{R}$ has to transform the join on fixed-size values, $\widehat{J}$, back to the real inputs using the algorithm in Algorithm 2. The full protocol is shown in Figure 7.

Dealing with duplicate values. As discussed in Sections 4.3.1 and 4.4, if a corrupt data source sends an identical value multiple times, it can collude with $\mathcal{R}$ to deanonymize entries based on $cval$ ciphertexts, if they are not rerandomized. val_{fixed} is rerandomized properly, so this attack does not apply to it. We highlight why C_{var} is not susceptible either. Since the source reuses an identifier val_{fixed} for duplicate values, $\mathcal{R}$ similarly learns only what the functionality guarantees: The same value (and its respective ciphertext) occur in the join. By placing a random "decoy" ciphertext per duplicate, each data source preserves a consistent view of its table length $|T_i|$, since for each loop, a C_{val} is added. The separate shuffle of the input table and the output list remove the linkability of an original value and its duplicates based on its location in the list.

Algorithm 1 Preprocessing for table T_i and value length ℓ

procedure PREPROCESS(T_i)
  Initialize $T_{used}, \widehat{T}_i, Ctbl_i$ as empty tables
  $o_i \leftarrow 0$
  Shuffle T_i
  for $uid_{i,j}, val_{i,j} \in T_i$ **do**
    $k_{i,j} \leftarrow \mathsf{SE.KGen}(1^\lambda)$
    $val'_{i,j} \leftarrow \mathsf{padToLenght}(val_{i,j}, \ell)$
    $C_{val_{i,j}} \leftarrow \mathsf{SE.Enc}(k_{i,j}, val'_{i,j})$
    if $val_{i,j} \in \mathsf{UIDs}(T_{used})$ **then**
      $k_{i,j}||o_i \leftarrow T_{used}[val_{i,j}]$
    else
      $T_{used}[val_{i,j}] \leftarrow k_{i,j}||o_i$
    $\widehat{T}_i[uid_{i,j}] \leftarrow k_{i,j}||o_i$
    $Ctbl_i[o_i] \leftarrow C_{val_{i,j}}$
    Increment the offset $o_i \leftarrow o_i + 1$
  Shuffle $\widehat{T}_i$
  Output $(\widehat{T}_i, Ctbl_i)$

Algorithm 2 Postprocessing for $\widehat{J}, Ctbl_1, \ldots, Ctbl_n$

procedure POSTPROCESS($\widehat{J}, Ctbl_1, \ldots, Ctbl_n$)
  Initialize $J \leftarrow \emptyset$
  for $(k_{1,j}||o_1, \ldots, k_{n,j}||o_n) \in \widehat{J}$ **do**
    for $i \in [n]$ **do**
      Parse $k_{i,j}, o_i \leftarrow k_{i,j}||o_i$
      Retrieve $C_{val_{i,j}} \leftarrow Ctbl_i[o_i]$
      $val'_{i,j} \leftarrow \mathsf{SE.Dec}(k_{i,j}, C_{val_{i,j}})$
      $val_{i,j} \leftarrow \mathsf{unpad}(val'_{i,j})$
    Apppend $(val_{1,j}, \ldots val_{1,j})$ to J
  Output J

Large Variable-size Values. This extension also makes it practical to completely hide the values' length from the network adversary and from the receiver (for the values outside the join). This is achieved by padding the values to a fixed maximum size. The aforementioned operation is also highly efficient, even for large values, since it relies on symmetric encryption, and can be parallelized easily.

Concrete Parameters. For our protocol, choosing a 16-byte random key k for (all-zero IV) AES-CTR and using an 12-byte offset o enables a $k||o$ to fit into a single coordinate of a P-256 point, minimizing the required rerandomizations. This also permits a maximum table size of 2^{96} per source.

F Multi-key Joins

One straightforward modification is the support for multi-label maps, i.e., maps of the form $\{(uid_0, \ldots, uid_z, val)\}$. This enables referring to data through multiple identifiers, e.g. e-mail addresses,

Setup for sid = $(\text{sid}', n, \mathcal{H}, \mathcal{R}, \mathcal{S}_1, \ldots, \mathcal{S}_n)$ with $\text{sid}' \xleftarrow{\$} \{0,1\}^\lambda$

1: Same as π_{MPPJ}.Setup.

Preparation:Prepare Run by $\mathcal{S}_i$ on $(\text{PREP}, \text{sid}, T_i, \mathcal{S}_i)$, $i \in [n]$

1: $\widehat{T}_i, Ctbl_i \leftarrow \text{Preprocess}(T_i)$
2: $T'_i \leftarrow \pi_{\text{MPPJ}}.\text{Prepare}(\text{sid}, \widehat{T}_i)$
3: Send $(\text{joinprep}, \text{sid}, T'_i, Ctbl_i)$ to $\mathcal{H}$ via $\mathcal{F}_{\text{SMT}}$
4: $\mathcal{H}$ stores $(\text{sid}, T'_i, Ctbl_i)$, and outputs $(\text{PREP}, \text{sid}, \mathcal{S}_i)$

Join:Convert Run by $\mathcal{H}$ on input $(\text{JOIN}, \text{sid}, \mathcal{R})$

1: Retrieve $T'_i, Ctbl_i$ for all $i \in [n]$ stored for sid
2: $J' \leftarrow \pi_{\text{MPPJ}}.\text{Convert}(\text{sid}, T'_1, ..., T'_n)$
3: Send $(\text{join}, J', Ctbl_1, ..., Ctbl_n)$ to $\mathcal{R}$ via $\mathcal{F}_{\text{SMT}}$, with J' in a random order

Join:Extract Run by $\mathcal{R}$ on $(\text{join}, J', Ctbl_1, \ldots, Ctbl_n)$ from $\mathcal{H}$

1: $\widehat{J} \leftarrow \pi_{\text{MPPJ}}.\text{Extract}(J')$
2: $J \leftarrow \text{Postprocess}(\widehat{J}, Ctbl_1, \ldots, Ctbl_n)$
3: Output $(\text{JOINED}, \text{sid}, J)$

Figure 7: MPPJ protocol with support for variable-sized values. The public parameters for all parties are $pp = (\mathbb{G}, g, q)$.

names, and phone numbers, simultaneously. The intended functionality is to reveal *val* to $\mathcal{R}$ if at least one of the identifiers matches across all n data sources. DPMC [28] also supports multi-label matching for their n-to-1 join protocol, highlighting that multi-key joining is a desirable feature.

For π_{MPPJ}, this can be achieved by performing the same processing steps (Prepare, Convert) for each *uid* in a parallel fashion, so that $\mathcal{R}$ can recover $\mathsf{H}_{\mathbb{G}}(uid_x)^{k_{pad}}$ for any nym_x during Join extraction. This method scales linearly in the number of additional *uid*s, however, the value *Cval* only needs to be generated once. The seed s is then encrypted z times using $\mathsf{H}_{\mathbb{G}}(uid_x)^{k_{pad}}$.

G Threshold Join

The main protocol changes for this extension are at the helper and receiver. The helper uses Shamir secret sharing, instead of additive sharing, to split k_{pad}. This means that a share is now a point $(i, f(i)) \in \mathbb{Z}_q^2$ for a random polynomial f over $\mathbb{Z}_q$, where the degree of f is $t-1$ and $f(0) = k_{pad}$.

This means that $\mathcal{H}$ can compute $spad_i$ as previously. However, $\mathcal{R}$ now needs to perform Lagrange interpolation at the exponent. For this, it needs to know the Lagrange basis for the t shares it receives. In order to facilitate this reconstruction, $\mathcal{H}$ now needs to attach the table index i in plain, as having it encrypted precludes its retrieval – $\mathcal{R}$ needs to interpolate based on i to decrypt *Cval*. The reconstruction itself is performed as described in the main body Section 6.2.

Setup for sid = $(\text{sid}', n, \mathcal{H}, \mathcal{R}, \mathcal{S}_1, \ldots, \mathcal{S}_n)$ with $\text{sid}' \xleftarrow{\$} \{0,1\}^\lambda$

1: Same as in π_{MPPJ}.Setup

Preparation:Prepare Run by $\mathcal{S}_i$ on $(\text{PREP}, \text{sid}, T_i, \mathcal{S}_i)$, $i \in [n]$

1: Same as in π_{MPPJ}.Prepare

Join:Convert Run by $\mathcal{H}$ on input $(\text{JOIN}, \text{sid}, \mathcal{R})$

1: Retrieve T'_i for all $i \in \{1, \ldots, n\}$ stored for sid
2: Initialize $J' \leftarrow \emptyset$
3: $k_{nym}, k_{pad} \xleftarrow{\$} \mathbb{Z}_q^2$
4: $(1, k_{spad\,1}), \ldots, (n, k_{spad\,n}) = SSS.Share(k_{pad}, t, n)$
▷ *Use Shamir secret sharing here, so that $f(i) = k_{spad_i}$*
5: **for** $i \in [n]$ and $(cuid_{i,j}, cval_{i,j}) \in T'_i$ **do**
6: $s_{i,j} \xleftarrow{\$} \mathbb{G}$
7: $Cval_{i,j} \leftarrow (i, \text{SE.Enc}(\mathsf{H}(s_j), \text{PKE.ReRnd}(epk, cval_{i,j})))$
▷ *Add table index **outside** the ciphertext*
8: $cnym_{i,j} \leftarrow \text{ElG.RExp}(bpk, cuid_{i,j}, k_{nym})$
9: $cpad_{i,j} \leftarrow \text{ElG.RExp}(bpk, cuid_{i,j}, k_{pad})$
10: $cs_{i,j} \leftarrow \text{ElG.Mul}(bpk, cpad_{i,j}, s_{i,j})$
11: $cspad_{i,j} \leftarrow \text{ElG.RExp}(bpk, cuid_{i,j}, k_{spad_i})$
▷ *$cspad_{i,j}$ encrypts $uid_{i,j}$-specific share $spad_{i,j}$ with $spad_{i,j} = \mathsf{H}_{\mathbb{G}}(uid_{i,j})^{k_{spad\,i}} = \mathsf{H}_{\mathbb{G}}(uid_{i,j})^{f(i)}$*
12: Add $(cnym_{i,j}, Cval_{i,j}, cs_{i,j}, cspad_{i,j})$ to J'
13: Send (join, J') to $\mathcal{R}$ via $\mathcal{F}_{\text{SMT}}$, with the entries of J' in a random order

Join:Extract Run by $\mathcal{R}$ upon receiving (join, J') from $\mathcal{H}$

1: Initialize $J \leftarrow \emptyset$ and $J'' \leftarrow \emptyset$
2: **for** $(cnym_{i,j}, Cval_{i,j}, cs_{i,j}, cspad_{i,j}) \in J'$ **do**
3: Parse $i, Cval'_{i,j} \leftarrow Cval_{i,j}$
4: $nym_{i,j} \leftarrow \text{ElG.Dec}(bsk, cnym_{i,j})$
5: Add $(nym_{i,j}, i, Cval'_{i,j}, cs_{i,j}, cspad_{i,j})$ to J''
▷ *tuples are directly associated with their origin i*
6: **for** all identifiers nym_l that appears $n' \geq t$ times in J'' **do**
7: Collect $(i, Cval'_{i,l}, cs_{i,l}, cspad_{i,l})_{i \in [n']}$ from $J''[nym_l]$
8: Set $(spad_{i,l})_{i \in [t]} \leftarrow (\text{ElG.Dec}(bsk, cspad_{i,l}))_{i \in [t]}$
9: $pad_l \leftarrow \prod_{i=1}^{t} spad_{i,l}^{\prod_{m=1, m \neq i}^{t} \frac{m}{m-i}}$
▷ *SSS reconstruction at the exponent from t of the n' shares.*
10: **for** $i \in [n']$ **do**
11: $s_{i,l} \leftarrow \text{ElG.Dec}(bsk, cs_{i,l}) \cdot pad_l^{-1}$
12: $cval_{i,l} \leftarrow \text{SE.Dec}(\mathsf{H}(s_{i,l}), Cval'_{i,l})$
13: $val_{i,l} \leftarrow \text{PKE.Dec}(esk, cval_{i,l})$
14: Add $(val_{1,l}, \ldots val_{n',l})$ to J
15: Output $(\text{JOINED}, \text{sid}, J)$

Figure 8: MPPJ protocol with support for threshold t-out-of-n joins. The public parameters for all parties are $pp = (\mathbb{G}, g, q)$. For the sake of notation, assumes (w.l.o.g.) that parties $1, ..., n'$ have matching identifiers.