Beyond the Output: Inference Attacks on Private Set Union and Multi-Key Private Matching

Andrea Raguso
ETH Zurich
Zurich, Switzerland

Francesca Falzon
ETH Zurich
Zurich, Switzerland

Tianxin Tang
University of Glasgow
Glasgow, UK

Kenneth G. Paterson
ETH Zurich
Zurich, Switzerland

Abstract

Recent work (Falzon and Tang USENIX 2025) has shown that a protocol participant who behaves honestly but strategically chooses its inputs can break input privacy in the Private Join and Compute functionality. In this work, we expand our understanding of attacks in this setting by investigating a broader class of functionalities, namely: Private Set Union (PSU), PSU-Cardinality (PSU-CA), and Meta's multi-key private matching (MKPM) functionality.

We begin with a simple yet novel attack on PSU that fully reconstructs the intersection using only two protocol invocations and forms the conceptual foundation for our more complex attacks. We also show that any attack on PSI-Cardinality, such as that of Guo et al. (USENIX 2023), lifts to an attack on PSU-CA that recovers the intersection with only three additional queries. For the MKPM protocol, we distinguish its intended matching functionality from the protocol-specific leakage, give an attack against the intended functionality, and then show that exploiting the additional leakage enables even stronger attacks, including partial reconstruction of the other party's records from a single protocol invocation.

We conclude by discussing possible mitigations for deploying such systems. Our analysis demonstrates limitations of existing secure multi-party computation security definitions and highlights the real-world privacy risks associated with deploying these functionalities in practice.

Keywords

private set union, private set intersection, multi-key private matching, attacks, multi-party computation

1 Introduction

Functionality leakage is an inherent feature of systems, such as secure multi-party computation (MPC), that allow two or more parties to compute over private inputs. Until recently, MPC primitives were largely theoretical, and their security definitions reflected this focus: standard MPC security aims to ensure that the only information leaked is the functionality output.

Today, these primitives are increasingly deployed. For example, Google's Private Join and Compute protocol uses MPC for ad-conversion measurement [27], and Coinbase's MPC wallet employs MPC to split a client's private key into two shares—one held by the client and the other by Coinbase [13]. At the same time, it is well known that given a function and access to one input and the resulting output, an adversary can infer additional information about the other party's input. This information leakage is intrinsic to the functionality and not a failure of the protocol to meet its defined security. The growing adoption of MPC protocols thus makes it critical to understand what information can be inferred by participants and under which threat models such leakage becomes practically significant.

Among the most widely studied and deployed MPC protocols are set-operation functionalities, such as private set intersection (PSI), which allows parties to privately compute the intersection of their datasets. Set-operation functionalities have been extensively explored in both protocol design and attack analysis (see Section 1.2). However, prior cryptanalytic work that considers adversaries who honestly execute the protocol and seek to infer information from the output is narrow in scope, focusing almost exclusively on PSI-like protocols [20, 28, 34]. This leaves a broad range of other functionalities largely unstudied in this adversarial setting.

Analyzing New Functionalities. In this work, we focus on private set operations and more complex join functionalities. Our analysis begins with *private set union (PSU)*, where two or more parties, each holding a set, compute and output the union without revealing anything else [4, 7, 8, 17, 23–25, 32, 33, 37, 38, 41, 47, 48, 50–53]. We also consider a variant called *PSU-Cardinality (PSU-CA)*, which computes and outputs the cardinality of the union, offering aggregate statistics without revealing the contents of the inputs [14, 21].

To enable richer private data analytics, database operations often go beyond basic set operations. For example, a company may wish to group information about the same individual across multiple databases, where each record contains a key (e.g., name) and associated attributes (e.g., email, credit score). If a key appears in the intersection, the corresponding attributes can be *unioned* to complete the individual's information. This functionality, known as a *database join*, can be viewed as a rich generalization of PSI and PSU. In practice, more expressive operations such as multi-key joins are often required. Building on our analysis of PSU and PSU-CA, we further study the multi-key extension of Meta's Private-ID protocol, *Multi-key Private Matching for Compute (MKPM)* [5], to understand the security guarantees provided by such protocols.

Threat Model and Recovery. In this work, we consider a realistic adversary who executes the protocol honestly but may craft its inputs to maximize the information it can infer about the other party's data. A simple illustration of this arises in PSU, which is designed to hide the intersection of the input sets. While the intended security goal is to disclose "nothing beyond the output," even trivial queries can violate this guarantee. For example, an adversary could submit the empty set and immediately learn the other party's entire input. Prior work, such as Frikken [23], addresses this problem by assuming thresholds or authorization mechanisms to prevent such

Proceedings on Privacy Enhancing Technologies 2026(4), 983–1005

https://doi.org/10.56553/popets-2026-0155

pathological queries. In this work, we go further: we ask whether it is possible to circumvent these types of mitigation strategies.

Looking beyond simple, abstract functionalities to concrete protocol instantiations introduces additional subtleties. We explicitly separate the intended functionality (what is supposed to be computed) from additional protocol-specific leakage (e.g., set sizes and structural information). For example, many PSI/PSU schemes leak the parties' input set sizes in addition to the intersection/union. In the case of Meta's multi-key join protocol [5], we also distinguish between the ideal multi-key matching functionality and the actual protocol leakage, which reveals extra information in exchange for efficiency. This distinction motivates our two-part analysis of MKPM: one part considers the intended functionality, and the other considers all leakage available in a real protocol execution. Our analysis shows that attacks can become drastically more efficient when the adversary also learns this additional leakage. This highlights the importance of carefully examining protocol-specific leakages, which is often neglected in the literature, as was the case for MKPM [5].

1.1 Contributions

We present seven attacks that reveal the fundamental limitations in how existing PSU and multi-key join functionalities protect input privacy. Our attacks are summarized in Table 1.

We consider an adversary that participates in the two-party protocol, honestly executes it, and learns the output, but may maliciously craft its input to maximize information gained. As in prior work [20, 28, 34], we assume that the adversarial party, $\tilde{P}_1$, holds a **target set** T it seeks to learn about. The honest party, P_2, holds the **recovery set** Y. The input structure depends on the functionality. For PSU and PSU-CA, $T = \{t_i\}_{i\in[n]}$ and $Y = \{y_k\}_{k\in[m]}$ are sets of values. For MKPM, $T = \{\mathbf{t}_i\}_{i\in[n]}$ and $Y = \{\mathbf{y}_i\}_{i\in[m]}$ are sets of records, where each record is a vector of values. As in prior work [20], we assume the recovery set is static. However, as we will see, many of our attacks do not rely on this assumption, as significant information can be recovered in one protocol invocation.

Attacks against PSU and PSU-CA. To pave the way for the attacks against the more complex functionality of multi-key private matching, we first describe a straghtforward set-based attack against PSU. The attack only requires two queries on disjoint sets that form a partition of the target set T (e.g., $T = X_1 \cup X_2$). Using these two queries, the adversary can fully recover the intersection $T \cap Y$ that PSU is supposed to protect. Next, we consider PSU-CA, which only reveals the cardinality. Even in this setting, we can still recover the intersection $T \cap Y$ through a sequence of adaptively chosen queries. We establish a direct connection to PSI-CA, showing that any attack against PSI-CA (e.g., [20, 28, 34]) can be transformed into an attack against PSU-CA using only three additional queries. Although our results on PSU are simple, they have not been noted in the literature, nor can they be circumvented using, for example, enforcing a threshold on the input set size as is commonly suggested. *This further raises question of whether PSU can be safely used as a standalone protocol or as as a component within more complex systems. Moreover, by establishing the relation between attacks on PSU-CA and PSI-CA, we demonstrate how vulnerabilities in one functionality can be leveraged to attack a broader class of functionalities.*

Attacks against Meta's Multi-Key Private Match. We next turn our attention to Meta's multi-key join functionality [5] and present five concrete attacks. We begin by analyzing the intended functionality (without leakage) and show that, even under the most idealized output specification, an adversary can infer which records in T share attribute values with records in Y. We further establish a close relationship between PSU-CA and MKPM, demonstrating that our attack on PSU-CA can be directly leveraged to recover this information in the multi-key join setting.

The remaining attacks further exploit protocol-specific leakage. In particular, one attack recovers all values shared between T and Y, while the final three produce an exact reconstruction of the records in Y, up to the shared values. These latter attacks are able to do so using a *single* query. *Together, these results underscore the importance of explicitly modeling and cryptanalyzing seemingly benign, protocol-specific leakage that extends beyond the intended functionality.*

Experimental Evaluation. We implemented all attacks to demonstrate their practical efficiency. For example, our PSU attack scales to recovery sets of size $|Y| = 10^6$ elements, with runtimes ranging from 0.74 s to 3.13 s. For adaptive attacks requiring multiple queries (e.g., PSU-CA and MKPM without leakage), the empirical query cost is significantly lower than the theoretical upper bound, demonstrating greater practical efficiency than theory alone suggests.

Broader Impact. Finally, we discuss the broader implications of our work. We re-examine common mitigation strategies, such as rate limiting and input validation, and explain why these mechanisms may be insufficient against the adversaries considered in this work.

1.2 Related Work

The seminal works of Freedman, Nissim, and Pinkas [22] and Agrawal, Evfimievski, and Srikant [1] laid the foundations for modern PSI. Since then, a long line of work has proposed protocols for both PSI (e.g.,[9, 10, 15, 16, 36, 45, 46, 49]) and PSU (e.g., [17, 24, 32, 33, 50–53]). More recent constructions extend PSI to support private computation over payloads on intersecting elements, such as sums [30] and inner products [12, 40].

The systematic study of attacks on set-operation functionalities was initiated by Guo et al. [28], who presented intersection-recovery attacks against PSI-CA, PSI-SUM, and Meta's Private-ID [6]. Jiang et al. [34] strengthened these results by reducing the required number of queries. Falzon and Tang [20] proposed attacks on Google's Private Join and Compute (PJC) [40] using combinatorial, statistical, and signal-processing techniques. In a complementary direction, Zinkus et al. [54] described a generic approach to automatically quantify functionality leakage.

These attack works, however, largely focus on single-key or set-based operations and do not analyze more complex matching and join protocols on multi-key tables. For example, subsequent work extended Private-ID to support multi-key datasets [5]. Other protocols consider different models, such as delegation of computation to an untrusted third party [44], joins across multiple parties [39], or secret-shared outputs to enable downstream computation [11, 43]. Asharov et al. [2] proposed the first maliciously secure protocols that support JOIN and GROUP-BY. However, attacks involving adversarially chosen inputs and repeated executions often fall outside

Ref.	Func.	Goal	#Qry	Approx.	Adapt.
[28]	PSI-CA	IR	$O(n)$		✓
[34]	PSI-CA	IR	$O(n)$		✓
	PSI-Sum	IR	–		✓
[20]	PJC	KVR	$O(k \log n)$		✓
		KVR	–	✓	
		KVR	$2k$		
		KVR	$\Theta(k \log(n/k))$	✓	
§4.1	PSU	IR	2		
§4.2	PSU-CA	IR	$O(n)$		✓
§5	MKPM	MRR	$O(n)$		✓
§6	L-MKPM	VIR	$O(n)$		✓
§7.1		TR	1		
§7.2		TR	1		
§7.3		TR	1		

Table 1: An overview of our attacks and prior work. We report the recovery goal, the number of queries required, whether the attack can reconstruct the values approximately, and whether it is adaptive. We abbreviate the recovery goals as follows: IR=intersection recovery (Def. 3.1), KVR=key-value recovery, MRR=matchable record recovery (Def. 3.2), VIR=value intersection recovery (Def. 3.3), and TR= T-reconstruction (Def. 3.4). Here n denotes the size of the adversary's target set and k is the size of the intersection of the target set and the recovery set.

their explicit threat models, leaving their resilience to the kinds of inference attacks we study as an open question.

2 Preliminaries

Notation. For $n \in \mathbb{N}$, we define $[n] := \{1, \ldots, n\}$. For any $\ell, n \in \mathbb{N}$ such that $\ell \geq \lfloor \log_2(n+1) \rfloor$, $\textsc{Bin}_\ell(n) \in \{0,1\}^\ell$ denotes the ℓ-bit binary representation of n.

For sets X, Y and any function $f : X \to Y$, we denote the image of X under f as $f(X) := \{f(x) | x \in X\} \subseteq Y$. Moreover, $\mathsf{Funcs}[X, Y]$ denotes the space of functions $f : X \to Y$ and $\mathsf{InjectiveFuncs}[X, Y]$ denotes the set of all injective functions of $\mathsf{Funcs}[X, Y]$. $\mathsf{Perms}[X]$ denotes the set of all permutations of X.

2.1 PSU and PSU-CA Functionalities

In its most basic form, the *PSU functionality* is a two-party protocol in which parties P_1 and P_2 hold sets X and Y, respectively. At the end of the protocol, both parties learn the union of their sets, $X \cup Y$.

The *PSU-Cardinality (PSU-CA) functionality* is a variant in which P_1 and P_2, holding sets X and Y as above, learn only the cardinality of the union, i.e., $|X \cup Y|$.

2.2 Multi-key Private Matching Functionality

Indexed Records. The Multi-Key PrivateID (MK-PrivateID) protocol [5] extends the single-key PrivateID protocol [6] to support matching on multiple keys (e.g., names or email addresses). We use terms *identifiers* and *keys* interchangeably, following the terminology of the original paper [5]. Each record contains multiple indexed values under different keys (See Section 2.2.1 for the representation). The keys themselves are implicit.

Matching Procedure. We refer to the functionality implemented by MK-PrivateID as the *Multi-key Private Matching (MKPM)* functionality. In this setting, two parties P_1 and P_2 hold sets of records X and Y, respectively. MKPM first groups records that share at least one common indexed value, and then applies a tailored matching procedure to produce a one-to-one correspondence between the two sets. This design is motivated by the use case in which partially duplicated records correspond to the same individual, and a one-to-one mapping suffices for downstream data analysis.

During matching, one-to-many and many-to-one relationships may arise (i.e., when a record from one party shares indexed values with multiple records from the other.). The protocol resolves many-to-one matches using a ranked, deterministic join logic, and resolves one-to-many matches randomly.

To enable downstream computation over the associated data, the protocol introduces *universal identifiers (UIDs)* to label joined records. It outputs to each party a mapping from UIDs to its records such that matched records are assigned the same UID, while unmatched records receive distinct UIDs.

Figure 2 illustrates this process. It shows an example input featuring a one-to-many match (Fig. 2a) that is resolved into a one-to-one correspondence, along with the resulting output (Fig. 2b).

Security Guarantee. The original work [5] sketches a proof for semi-honest security. However, we note that the specified functionality leaks more information than the intended output. We therefore distinguish the intended functionality, which computes the set of UIDs and the mapping of UIDs to records ($\mathcal{F}_{\mathsf{MKPM}}$), from the real functionality ($\mathcal{F}_{\mathsf{L\text{-}MKPM}}$), which reveals additional information. Returning to our example, given the tables in Figure 2a, $\mathcal{F}_{\mathsf{MKPM}}$ outputs only the maps and UID sets shown in Figure 2b, whereas $\mathcal{F}_{\mathsf{L\text{-}MKPM}}$ additionally leaks to party P_1 a graph isomorphic to that comprising the red and blue edges in Figure 2a.

2.2.1 *Functionality.*

We now present a formal description of the functionality $\mathcal{F}_{\mathsf{MKPM}}$. Let $\mathbb{V}$ denote the universe of record values.

Party P_1 holds

$$X := \{\mathbf{x}_i = (x_{i,1}, \ldots, x_{i,j}, \ldots, x_{i,n_i}) \mid j \in [n_i]\}$$

of size n, and party P_2 holds

$$Y := \{\mathbf{y}_k = (y_{k,1}, \ldots, y_{k,\ell}, \ldots, y_{k,m_k}) \mid \ell \in [m_k]\}$$

of size m. Here, we slightly abuse notation, using n_i, m_k to denote the lengths of records $\mathbf{x}_i$ and $\mathbf{y}_k$, respectively, for all $i \in [n]$ and all $k \in [m]$; all values $x_{i,j}$ and $y_{k,\ell}$ lie in $\mathbb{V}$.

To suppress leakage arising from positional information in the matching protocol, the original work [5] introduces random permutations. Specifically, the records in X are permuted while preserving the order of indexed values within each record, whereas in Y both the records and the values within each record are permuted. We denote the resulting collections by X' and Y', respectively.[1]

The core of the protocol is the match procedure (see Appendix A.1 for detailed pseudocode), which outputs a one-to-one correspondence between the two sets. It takes as input the shuffled

[1] Although sets are unordered, the implementation [42] treats inputs as ordered lists of records; we follow the authors of [5] in adopting this terminology.

Parameters: Two parties, P_1 and P_2, set sizes $n, m \in \mathbb{N}$ and record lengths $n_i, m_k \in \mathbb{N}$ for $i \in [n]$ and $k \in [m]$.
Functionality $\mathcal{F}_{\mathsf{MKPM}}(X;Y)$ [$\mathcal{F}_{\mathsf{L\text{-}MKPM}}(X;Y)$]:

1 Receive the input from party P_1: $X = \{\mathbf{x}_i \mid i \in [n]\}$
2 Receive the input from party P_2: $Y = \{\mathbf{y}_k \mid k \in [m]\}$
3 Sample $\pi_X \leftarrow\!\!\$\ \mathsf{Perms}[[n]]$ and $\pi_Y \leftarrow\!\!\$\ \mathsf{Perms}[[m]]$
4 Let $X' := \{\mathbf{x}'_i \mid i \in [n]\}$, where $\mathbf{x}'_i := \mathbf{x}_{\pi_X(i)}$
5 Let $Y' := \{\mathbf{y}'_k \mid k \in [m]\}$, where $\mathbf{y}'_k := \mathbf{y}_{\pi_Y(k)}$
6 Shuffle the values in $\mathbf{y}'_k$ for all $k \in [m]$
7 Compute set UID, mappings $\mathsf{M}_X, \mathsf{M}_Y \leftarrow \mathsf{match}(X', Y')$ (See Figure 13 in Appendix A.1.)

[
8 Sample $f_1, f_2 \leftarrow\!\!\$\ \mathsf{InjectiveFuncs}[\mathbb{V}, \mathbb{U}]$
9 $\mathcal{L}^1_X := \{(f_1(\mathbf{x}'_i[1]), \ldots, f_1(\mathbf{x}'_i[n_i])) \mid i \in [n]\}$.
10 $\mathcal{L}^1_Y := \{(f_1(\mathbf{y}'_k[1]), \ldots, f_1(\mathbf{y}'_k[m_k])) \mid k \in [m]\}$.
11 $\pi \leftarrow\!\!\$\ \mathsf{Perms}[[n]]$.
12 $X'' := \{\mathbf{x}''_i \mid i \in [n]\}$, where $\mathbf{x}''_i := \mathbf{x}_{\pi(i)}$.
13 $\mathcal{L}^2_X := \{(f_2(\mathbf{x}''_i[1]), \ldots, f_2(\mathbf{x}''_i[n_i])) \mid i \in [n]\}$.
14 $\mathcal{L}^1 \leftarrow (\mathcal{L}^1_X, \mathcal{L}^1_Y)$, $\mathcal{L}^2 \leftarrow \mathcal{L}^2_X$
]

15 Output:
- Send UID, M_X [, $\mathcal{L}^1$] to P_1.
- Send UID, M_Y [, $\mathcal{L}^2$] to P_2.

Figure 1: The Multi-Key Private Matching functionality $\mathcal{F}_{\mathsf{MKPM}}(X;Y)$. The additional steps of the leakage-aware functionality $\mathcal{F}_{\mathsf{L\text{-}MKPM}}(X;Y)$ are shown in dashed boxes.

inputs X' and Y'. Each record in X' (resp. Y') is matched with at most one record in Y' (resp. X'). Both parties receive a common set of universal identifiers, along with a mapping from each UID to their local records or $\perp$ for unmatched records.

The intuition behind match is to process indexed values in an order chosen by Party P_1, allowing it to prioritize values that are less likely to change. For example, matching on a social security number typically yields higher-quality results than matching on a phone number, which is more likely to change.

More formally, for each identifier position j in X', the match procedure scans all $\mathbf{y}_k \in Y'$ and matches $\mathbf{y}_k$ to the first $\mathbf{x}_i \in X'$ whose j-th value appears in $\mathbf{y}_k$, that is, for which there exists an ℓ such that $\mathbf{y}_k[\ell] = \mathbf{x}_i[j]$, where i is minimal.

A detailed description of $\mathcal{F}_{\mathsf{MKPM}}$ is provided in Figure 1.

2.2.2 Leakage. In favour of higher efficiency, MK-PrivateID leaks additional information. We capture this leakage using an extended "leaky" functionality, denoted by $\mathcal{F}_{\mathsf{L\text{-}MKPM}}$. The leakage differs for P_1 and P_2, and we describe each separately.

To model this leakage, we introduce a *hiding function*, which is an injective random function from the value space to the UID space, $f_1 : \mathbb{V} \to \mathbb{U}$. This function assigns UIDs to values and allows us to formalize the relationships between records that can be inferred from the output UIDs. By applying the hiding function to the values in the shuffled sets X' and Y', we obtain *hidden copies* of these sets, which are available to party P_1. In particular, repeating values are mapped to the same UIDs consistently in the hidden copies of X' and Y'. See the definition of $\mathcal{L}^1_X$ and $\mathcal{L}^1_Y$ in Figure 1. Also note that the order of indexed values within the records of X is preserved in $\mathcal{L}^1_X$. In contrast, the order within the records of Y is shuffled to obtain Y' and, thus, the same does not hold for $\mathcal{L}^1_Y$.

The leakage observed by P_2 is more limited. Party P_2 learns only a hidden and re-shuffled copy of X. Specifically, the records of X are re-shuffled using fresh randomness, yielding a set X''. A new hiding function $f_2 : \mathbb{V} \to \mathbb{U}$ is then sampled and applied to all values in X'', and the resulting hidden set is available to P_2.

The pseudocode can be found in Figure 1, where the additional leakage is shown dashed boxes.

3 Threat Model and Recovery Goals

Threat Model. Without loss of generality, and in line with prior work [20, 28, 34], we assume that P_1 is adversarial (denoted by $\tilde{P}_1$). The adversary follows the protocol specification but may adaptively choose its inputs and repeatedly invoke the functionality. Party P_2 is honest and holds a fixed set Y of elements, the **recovery set**, about which P_1 aims to learn information. We note that three of our attacks require only a single query and therefore also apply when P_2's recovery set is dynamic.

A **target set** T of values is also introduced, which is held by the adversarial party $\tilde{P}_1$. The set T represents values corresponding to points of interest, and the adversary aims to determine which elements of T also appear in Y, according to the recovery goals defined in the next section.

3.1 Recovery Goals

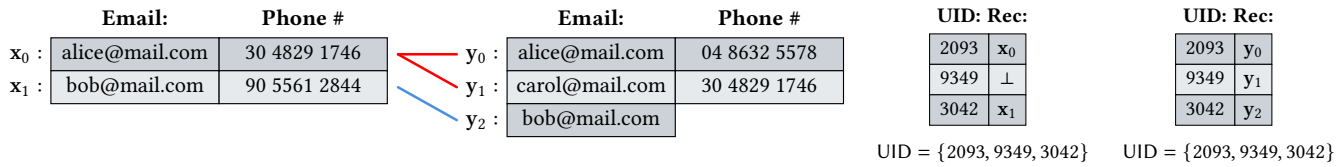
Since we analyze multiple set operations in this work, we specify a recovery goal for each functionality to model attack scenarios. Looking ahead, we focus on intersection recovery in $\mathcal{F}_{\mathsf{PSU}}$ and $\mathcal{F}_{\mathsf{PSU\text{-}CA}}$. For the complex functionality $\mathcal{F}_{\mathsf{MKPM}}$, we introduce tailored recovery notions that capture the information about the target set that an adversary may aim to learn. In the following discussion, we let T be the target set and Y be the recovery set.

3.1.1 Recovery Goal against $\mathcal{F}_{\mathsf{PSU}}$ and $\mathcal{F}_{\mathsf{PSU\text{-}CA}}$. Private Set Union (PSU) aims to hide the intersection of the input sets [33]. Accordingly, the natural attack objective against PSU and PSU-CA is to recover this intersection, i.e., determine which elements of the target set T also appear in Y.

Definition 3.1. An adversary $\mathcal{A}$ achieves **intersection recovery** if it outputs $T \cap Y$.

3.1.2 Recovery Goal against $\mathcal{F}_{\mathsf{MKPM}}$. Given that $\mathcal{F}_{\mathsf{MKPM}}$ operates on sets of records, a natural recovery goal would be to recover the set of records that appear in both input sets. However, the functionality aims at matching records based on shared values (not on the equality of the records). We thus adapt the notion of intersection recovery to the multi-key setting to mirror this logic, aiming instead to identify the records in the target set T that share values with records in Y.

The matching logic of MK-PrivateID (see Figure 13 for a definition and Figure 2 for an example) assigns the same UID to matched pairs of records and distinct UIDs to unmatched ones. One can view match as computing a pseudo-union of records, where a record in the target set and one in the recovery set are considered "equal"



(a) The inputs from Parties P_1 (left) and P_2 (right). **(b) The outputs to Parties P_1 (left) and P_2 (right).**

Figure 2: In (a), we illustrate a one-to-many match: party P_1's record $\mathbf{x}_0$ matches party P_2's records $\mathbf{y}_0$ on email and $\mathbf{y}_1$ on phone (red edges). The conflict is resolved randomly, yielding a one-to-one mapping in which $\mathbf{x}_0$ and $\mathbf{y}_0$ are assigned the same UID (2093), while $\mathbf{y}_1$ is assigned a distinct UID (9349). Record $\mathbf{x}_1$ matches only $\mathbf{y}_2$ on email (blue edge) and thus both are assigned UID 3042. Both parties learn their local mappings and the universe of UIDs.

if they are assigned the same UID. A natural generalization of set intersection is to recover the records in the adversary's target set that are matched with some record in the recovery set.

Since the adversary's goal is to learn as much as information about the target set, which may contain sensitive information, we define the recovery goal as outputting all records in the target set that share at least one value with a record in the recovery set. This captures scenarios in which indexed values, such as social security numbers or email addresses, uniquely identify an individual. Notably, the original paper [5] explicitly emphasizes precautions against leaking this information in the protocol design. We refer to records that could *potentially* be matched by the protocol logic as **matchable records**. We formalize this as follows:

$$T \sqcap Y := \{\mathbf{t} \in T \mid \exists \mathbf{y} \in Y,\ \exists j, \ell \in \mathbb{N} \text{ s.t. } \mathbf{t}[j] = \mathbf{y}[\ell]\}.$$

Definition 3.2. An adversary $\mathcal{A}$ achieves **matchable record recovery (MRR)** if it outputs $T \sqcap Y$.

3.1.3 Recovery Goals against $\mathcal{F}_{\text{L-MKPM}}$. As discussed in Sec. 2.2.2, $\mathcal{F}_{\text{L-MKPM}}$ is the extended functionality that captures the leakage inherent to the protocol. Thus we introduce two additional recovery goals to better understand potential attacks that were not considered in the original paper [5].

The first recovery goal is to recover the set of shared values. This precisely captures the values that the adversary would like to identify in the recovery set. Given a record set X (defined as in Section 2.2.1), we denote by $\mathbb{V}(X)$ the set of values present in the records of X, defined as

$$\mathbb{V}(X) = \{v \in \mathbb{V} \mid \exists \mathbf{x} \in X \text{ such that } v \in \mathbf{x}\}.$$

Here, we slightly abuse notation by writing $v \in \mathbf{x}$ to denote that there exists an index $i \in \mathbb{N}$ such that $\mathbf{x}[i] = v$. The value recovery goal is formally defined as follows.

Definition 3.3. An adversary $\mathcal{A}$ achieves **value intersection recovery (VIR)** if it outputs $\mathbb{V}(T) \cap \mathbb{V}(Y)$.

The final goal is the strongest one: identifying all values shared between the target and recovery sets *and* how these values co-occur across the records of Y. It reveals which shared values appear in the same record and which values are repeated across multiple records—information that is not captured by VIR.

This corresponds to the most informative inference about Y with respect to the target set T that an adversary can obtain from $\tilde{P}_1$'s output. We refer to this notion as *T-reconstruction* of Y, and formalize it as follows. See Figure 3 for an illustration.

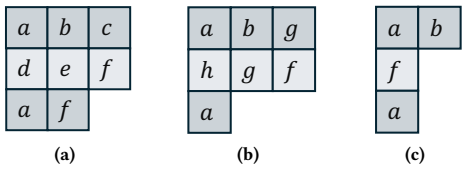


Figure 3: Given a (a) target set T and a (b) recovery set Y, we depict a (c) possible T-reconstruction R_Y. Each matched record in Y corresponds to exactly one record in R_Y, and all values shared between the matched record and the target set T appear in the corresponding record in R_Y.

Definition 3.4. An adversary $\mathcal{A}$ achieves **T-reconstruction (TR)** of Y if it outputs a multiset R_Y for which there exists an injective mapping $\varphi : R_Y \to Y$ such that

1. For all $\mathbf{y}^* \in R_Y$ and all $v \in \mathbf{y}^*$ we have $v \in \varphi(\mathbf{y}^*)$.
2. For all $\mathbf{y}^* \in R_Y$ and all $v \in \varphi(\mathbf{y}^*)$ we have if $v \in \mathbb{V}(T)$ then $v \in \mathbf{y}^*$.
3. For all $\mathbf{y} \in Y$, if there is some $v \in \mathbf{y}$ such that $v \in \mathbb{V}(T)$, then $\mathbf{y} \in \varphi(R_Y)$.

These three conditions formally define correctness. We defer the detailed discussion to Appendix A.2.

4 Analysis of PSU and PSU-CA

In this section, we present the analysis of PSU and PSU-CA. Since they are relatively straightfoward, we use them as a warm-up to the more involved attacks against multi-key matching in the later sections.

While PSU and PSU-CA have received increasing attention, the information leaked by their output remains largely unstudied. Frikken [23] observed that submitting an empty set in PSU fully reveals the other party's input and proposed abort-based mitigations. Jia et al. [33] further analyzed PSU leakage, showing that Guo et al.'s [28] PSI-CA attack extends to certain PSU variants, even under early-abort conditions, and proposed an "enhanced" PSU functionality that delays protocol leakage to the end of protocol executions.

We show that such countermeasures are insufficient by describing an intersection-recovery attack against PSU that uses only two queries and does not require the adversarial input to be empty or disjoint from the recovery set. We then prove that PSU-CA leaks at least as much information as PSI-CA by reducing any PSI-CA attack with q queries to a PSU-CA attack with q+3 queries.

4.1 Analysis of PSU

Recall that the corrupted party $\tilde{P}_1$ holds the target set T and can invoke $\mathcal{F}_{\text{PSU}}(X;Y)$ on arbitrary sets X with a fixed Y. Its goal is to recover $T \cap Y$. Our attack relies on the following observation:

Proposition 4.1. *Let T and Y be sets and $X_1 \cup X_2$ be a partition of T. It holds that $Y = (X_1 \cup Y) \cap (X_2 \cup Y)$.*

Proposition 4.1 follows directly from the fact that X_1 and X_2 are disjoint. The recovery set Y can therefore be reconstructed with only two PSU queries. Since the adversary knows T, the intersection $T \cap Y$, if needed, can then be computed easily.

We refer to this attack as PSU-Diff. Note that it requires no special inputs such as the empty set, but works with any two inputs that are disjoint from each other. Specific mitigation strategies that focus on prohibiting such special inputs are therefore ineffective.

4.2 PSU-CA Attack

The cardinalities of the union and intersection of two sets are tightly related. We give a generic transformation that converts any intersection-recovery attack against PSI-CA (e.g., [28]) into one against PSU-CA. For sets X and Y, our attack uses the relation

$$|X \cap Y| = |X| + |Y| - |X \cup Y|.$$

This equation lets us translate PSU-CA outputs into the PSI-CA setting, thereby lifting any PSI-CA intersection-recovery attack to PSU-CA. The remaining challenge is that $|Y|$ is not guaranteed to be known. We therefore show how to infer $|Y|$ using three PSU-CA evaluations, while avoiding the trivial approach of querying the empty set.

Proposition 4.2. *Let X and Y be two sets and let $X_1 \cup X_2$ be a partition of X. Moreover, let $z := |X \cup Y|$, $z_1 := |X_1 \cup Y|$ and $z_2 := |X_2 \cup Y|$. Then, we have $|Y| = z_2 - (z - z_1)$.*

The proof is provided in Appendix C.1. Thus, with three additional $\mathcal{F}_{\text{PSU-CA}}$ evaluations at the start of the attack, the adversary can simulate the $\mathcal{F}_{\text{PSI-CA}}$ functionality. In particular, for any PSI-CA attack, the adversary can replace each query $z \leftarrow \mathcal{F}_{\text{PSI-CA}}(X;Y)$ with $z' \leftarrow \mathcal{F}_{\text{PSU-CA}}(X;Y)$ and then compute z as $|X| + |Y| - z'$.

We state our result in the following theorem:

Theorem 4.3. *For every adversary $\mathcal{A}$ such that $\mathcal{A}^{\mathcal{F}_{\text{PSI-CA}}(\cdot;Y)}(T) = T \cap Y$ which makes q PSI-CA queries, there is an efficient adversary $\mathcal{B}$ with access to $\mathcal{F}_{\text{PSU-CA}}(\cdot;Y)$ which recovers $T \cap Y$ and makes $q+3$ PSU-CA queries.*

The proof is given in Appendix C.2. In Appendix C.3, we describe and analyze PSU-CA-SearchTree—an attack that uses the transformation described above to turn the PSI-CA attack from Guo et al. [28] to one against PSU-CA.

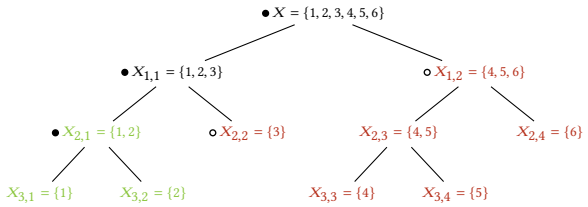


Figure 4: The binary search tree for the target set $X = \{1, 2, 3, 4, 5, 6\}$ in the intersection-recovery attack of [28], shown for recovery set $Y = \{1, 2\}$. Nodes marked with ∘ are queried via PSI-CA, while nodes marked with • have their intersection cardinality inferred from their parent and sibling. Depth first search terminates early at $X_{2,1}$ and $X_{1,2}$. Inferred membership is shown in green for $X \cap Y$ and red for $X \setminus Y$.

5 Matchable Record Recovery with $\mathcal{F}_{\text{MKPM}}$

In this section, we show how the PSU-CA-SearchTree attack can be applied to $\mathcal{F}_{\text{MKPM}}$ with some additional assumptions to achieve *matchable record recovery* (Definition 3.2). In Appendix D we explain how the attack can be applied to the single key private matching functionality $\mathcal{F}_{\text{SKPM}}$ realized by PrivateID [6].

5.1 A Primer on Guo et al.'s PSI-CA Attack

Before describing our attack, we briefly review Guo et al.'s [28] intersection-recovery attack against the PSI-CA functionality,which outputs the intersection cardinality $|X \cap Y|$.

In Guo et al.'s attack, the adversary has a target set X and can query the PSI-CA functionality $\mathcal{F}_{\text{PSI-CA}}(\cdot;Y)$ on adaptively chosen sets $X' \subseteq X$, where Y is a fixed recovery set. The goal is to recover $X \cap Y$. The key observation is that some PSI-CA outputs immediately reveal membership information: (1) if $|X' \cap Y| = 0$, then no element of X' lies in Y and (2) if $|X' \cap Y| = |X'|$, then every element of X' lies in Y.

To exploit this, the adversary builds a balanced binary search tree over X, where each node contains a subset of X and its children form a partition of that subset. The adversary traverses the tree and queries PSI-CA on selected nodes until it finds subsets whose membership can be fully determined by the two termination cases above. Since the children partition their parent, the adversary can infer one child's intersection cardinality from the parent and sibling values, reducing the number of queries needed: if $X_1 \cup X_2$ is a disjoint partition of X, we have $|X_2 \cap Y| = |X \cap Y| - |X_1 \cap Y|$. Repeating this process recursively allows the adversary to label elements as either in $X \cap Y$ or in $X \setminus Y$, thereby recovering the intersection.

We give a pictorial example of the attack in Figure 4 and a more detailed description of the attack in Appendix B.

5.2 From Union to UID Cardinalities

In the context of MKPM, the adversary is given a target set of records $T \subseteq \mathbb{V}^n$, and can evaluate $\mathcal{F}_{\text{MKPM}}(X;Y)$ for arbitrarily many inputs X. Its goal is to determine the set of matchable records, $T \sqcap Y$. The functionality $\mathcal{F}_{\text{MKPM}}$ outputs the set universal identifiers, UID, to both parties. Rather than assigning a common UID to two identical records, it applies a matching logic (see Figure 13) which assigns

a common UID to each pair of matched records, and a distinct UID to every unmatched record in T and Y. Thus, using $|\mathsf{UID}|$ as the cardinality for carrying out PSU-CA-SearchTree appears to be a natural approach for identifying matching records in T.

The PSU-CA-SearchTree attack (Section 4.2) relies on the fact that $|X \cup Y| = |X_1 \cup Y| + |X_2 \cup Y| - |Y|$ where X and Y are arbitrary sets and $X_1 \cup X_2$ form a partition of X. We now translate this to the multi-key setting. Let UID, UID_1, and UID_2 be the UIDs output when evaluating $\mathcal{F}_{\mathsf{MKPM}}(X; Y)$ and $\mathcal{F}_{\mathsf{MKPM}}(X_i; Y)$ for $i \in \{1, 2\}$ respectively. For the attack to apply, we require the following relation to hold:

$$|\mathsf{UID}| = |\mathsf{UID}_1| + |\mathsf{UID}_2| - |Y|. \tag{1}$$

Unfortunately, this condition does not hold for general inputs. The reason is that the MKPM functionality shuffles the inputs before applying the matching logic. Consequently, the matching procedure introduces randomness, leading to inconsistent record matches across different protocol runs, even with the same input sets. Thus, different evaluations of $\mathcal{F}_{\mathsf{MKPM}}$ on identical inputs can yield UID sets with varying cardinalities.

We give an example of such a matching in Figure 5.

To rule out such cases, we impose a restriction on the input sets that, if satisfied, ensures that Eq. 1 holds, thus allowing us to carry out the attack. Luckily, one-to-many relations do not violate Eq. 1. It therefore suffices to exclude many-to-one relations.

Definition 5.1. Let X, Y be sets of records. We say X is Y-**isolated**, if for all $\mathbf{y} \in Y$ we have

$$|\{\mathbf{x} \in X \mid \exists i, j \text{ s.t. } \mathbf{x}[i] = \mathbf{y}[j]\}| \leq 1.$$

This constraint may exclude certain scenarios. For example, if the sets contain IP addresses, many-to-one relationships are likely to occur since multiple users in the same local network share an IP address. However, the adversary's input may be Y-isolated in scenarios where the adversary targets a specific range of identifiers, such as all phone numbers with a specific area code or all email addresses from a particular domain.

Theorem 5.2. *Let $T, V \subseteq \mathbb{V}^n$ be sets of records. If T is Y-isolated, then PSU-CA-SearchTree$^{\mathcal{F}_{\mathsf{MKPM}}(\cdot;Y)}(T)$ recovers $T \sqcap Y$.*

The proof can be found in Appendix E.1.

Time and Query Complexity. The PSU-CA-SearchTree can be applied directly to the output of $\mathcal{F}_{\mathsf{MKPM}}$, as long as the adversary's input sets satisfy Definition 5.1. The attack thus has the same worst-case time complexity and maximum number of protocol invocations as when applied to $\mathcal{F}_{\mathsf{PSU\text{-}CA}}$. Specifically, it requires $O(n \log n)$ time and at most $n + 1$ protocol invocations (although typically fewer queries are needed due to the probability of early termination).

6 Value Intersection Recovery with $\mathcal{F}_{\mathsf{L\text{-}MKPM}}$

We now leverage the additional leakage that $\mathcal{F}_{\mathsf{L\text{-}MKPM}}$ outputs to $\tilde{P}_1$ to mount stronger attacks. We adapt Guo et al.'s search tree approach [28], designed for $\mathcal{F}_{\mathsf{PSI\text{-}CA}}$, to recover the shared values and achieve *value intersection recovery* (Def. 3.3).

For this attack, we restrict the adversary to using only subsets of its target set T as input: it may submit any $T' \subseteq T$ but cannot append, split, or rearrange the values within records (this contrasts with Section 7, where the adversary can modify record contents by appending or reordering values). We assume all records in T have equal length $\ell_{\mathsf{rec}} \in \mathbb{N}$. The goal is to compute the intersection of values which we denote as $\mathsf{pos} := \mathbb{V}(T) \cap \mathbb{V}(Y)$. Throughout the attack, we also construct a second set, $\mathsf{neg} := \mathbb{V}(T) \setminus \mathbb{V}(Y)$, of values in T that are not in Y.

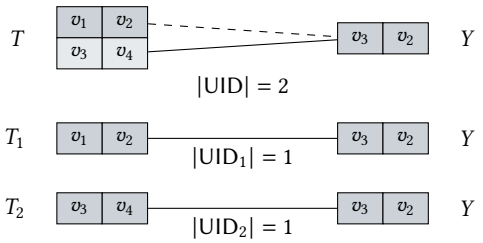


Figure 5: An example that PSU-CA-SearchTree attack cannot be applied to the output of $\mathcal{F}_{\mathsf{MKPM}}(T; Y)$. Lines between records indicate potential matches based on matching values. Solid lines denote the matched records chosen by the matching logic. We see that $|\mathsf{UID}| \neq |\mathsf{UID}_1| + |\mathsf{UID}_2| - |Y|$ and, as such, T is not Y-isolated.

Our adapted attack starts by building a binary search tree $\mathcal{T}$ over T similar to Guo et al.'s attack [28]. The root node corresponds to the entire target set T. Each node v is labeled with a subset $v.\mathsf{set} \subseteq T$. Its children $v_L \leftarrow v.\mathsf{left}$ and $v_R \leftarrow v.\mathsf{right}$ form a disjoint partition of the parent: $v_L.\mathsf{set} \cup v_R.\mathsf{set} = v.\mathsf{set}$ and $v_L.\mathsf{set} \cap v_R.\mathsf{set} = \emptyset$. We use a fixed, deterministic rule to compute the partition (e.g., by index into equal halves) and recurse until leaves are single records.

Starting at the root, the adversary traverses $\mathcal{T}$ in DFS order, selecting a node at each step and querying $\mathcal{F}_{\mathsf{L\text{-}MKPM}}(\cdot; Y)$ on that node's associated subset of records. Let $X \leftarrow v.\mathsf{set}$ for a non-leaf node v, and $X_1 \leftarrow v_L.\mathsf{set}$, $X_2 \leftarrow v_R.\mathsf{set}$ be its children. In Guo et al.'s PSI-CA attack, querying a parent X and one child X_1 suffices to infer the sibling's output: from $|X \cap Y|$ and $|X_1 \cap Y|$ one immediately obtains $|X_2 \cap Y|$ without a third query.

Under $\mathcal{F}_{\mathsf{L\text{-}MKPM}}$, this sibling inference is not possible: each query generates UIDs with fresh randomness, so given $(\mathcal{L}^1_X, \mathcal{L}^1_Y)$ and $(\mathcal{L}^1_{X_1}, \mathcal{L}^1_Y)$, one cannot derive $(\mathcal{L}^1_{X_2}, \mathcal{L}^1_Y)$. Instead of inferring the sibling's full leakage, we switch to a metric for which the sibling's value *can* be inferred without an additional query—namely, the number of matched values *per column* in $\mathcal{L}^1_X$ and $\mathcal{L}^1_{X_1}$—and use this to guide the DFS traversal. Formally, this metric is defined as:

$$\mathrm{ColCA}(\mathcal{L}^1_X, \mathcal{L}^1_Y) := \left(\sum_{\mathbf{t} \in \mathcal{L}^1_X} \mathbb{1}\{\mathbf{t}[i] \in \mathbb{V}(\mathcal{L}^1_Y)\} \right)_{i \in [\ell_{\mathsf{rec}}]}.$$

We make the following important observations about ColCA.

Proposition 6.1. *For any two sets of records X and Y, we have*

$$\mathit{ColCA}(\mathcal{L}^1_X, \mathcal{L}^1_Y) = \mathit{ColCA}(X, Y).$$

In words, because the hiding function used is deterministic, the leakage preserves the per-column counts of matching values between the plaintext inputs X and Y.

Proposition 6.2. *Let X and Y be sets of records, and let $X_1 \cup X_2$ be a partition of X. Then we have*

$$ColCA(\mathcal{L}^1_X, \mathcal{L}^1_Y) = ColCA(\mathcal{L}^1_{X_1}, \mathcal{L}^1_Y) + ColCA(\mathcal{L}^1_{X_2}, \mathcal{L}^1_Y)^2.$$

Proofs are given in Appendix F.1 and F.2, respectively.

From Prop. 6.2 we obtain $\text{ColCA}(\mathcal{L}^1_{X_2}, \mathcal{L}^1_Y) = \text{ColCA}(\mathcal{L}^1_X, \mathcal{L}^1_Y) - \text{ColCA}(\mathcal{L}^1_{X_1}, \mathcal{L}^1_Y)$. Here, $\mathcal{L}^1_{X_2}$ denotes the (hypothetical) leakage from a protocol run on X_2, which is never actually executed. This suffices, since we only need $\text{ColCA}(\mathcal{L}^1_{X_2}, \mathcal{L}^1_Y)$, and this vector can be inferred without knowing $\mathcal{L}^1_{X_2}$ explicitly.

For a leaf node containing a single record, i.e., $X = \{\mathbf{x}\}$, the vector $\text{ColCA}(\mathcal{L}^1_X, \mathcal{L}^1_Y)$ is binary: its i-th entry is 1 if and only if the i-th value of $\mathbf{x}$ appears in Y, and 0 otherwise. More generally, consider any node with record set X. Suppose that after querying this node we obtain

$$\mathbf{c} = \text{ColCA}(\mathcal{L}^1_X, \mathcal{L}^1_Y) \in \{0, |X|\}^{\ell_{\text{rec}}}.$$

By Prop. 6.1, this equals $\text{ColCA}(X, Y)$, so the i-th entry counts how many records in X have their i-th value in $\mathbb{V}(Y)$. If the i-th entry is $|X|$, then *every* record in X has its i-th value in Y; if it is 0, then *no* record in X has its i-th value in Y. Thus, for such a node, each column is either entirely positive (all values occur in Y) or entirely negative (no value occurs in Y). Thus for each $i \in [0, \ldots, \ell_{\text{rec}} - 1]$ if $\mathbf{c}[i] = |X|$ we add $\{\mathbf{x}[i] : \mathbf{x} \in X\}$ to pos, and otherwise we add it to neg. Once we have processed all the nodes in this way, pos must include all values in $\mathbb{V}(T) \cap \mathbb{V}(Y)$.

The remaining component of our adapted attack is a heuristic that (i) decides which child to follow during DFS traversal and (ii) provides the priority for enqueuing the other child. For any set X, this heuristic must be computable from X and $\text{ColCA}(\mathcal{L}^1_X, \mathcal{L}^1_Y)$ alone, since this is all we know about the (hypothetical) leakage for that node. For $\mathbf{c} = \text{ColCA}(\mathcal{L}^1_X, \mathcal{L}^1_Y)$, we define

$$p^+_{MK}(\mathbf{c}, X) := \frac{\sum_{x \in \mathbf{c}} x}{\ell_{\text{rec}}|X|}. \tag{2}$$

As in the original PSI-CA attack [28], our heuristic prioritizes the child node whose subsets contain more matched values, which increases the fraction of the intersection that can be recovered when the adversary is limited to a bounded number of protocol invocations. We additionally propose two further heuristics in F.4 and evaluate our three heuristics in Appendix H.

The pseudocode of MKPM-SearchTree is given in Figure 17 (Appendix F).

Theorem 6.3. *The MKPM-SearchTree$^{\mathcal{F}_{\text{L-MKPM}}(\cdot;Y)}(T)$ attack (Figure 17) achieves value intersection recovery (Def. 3.3).*

The proof can be found in Appendix F.3.

Time and Query Complexity. Let $n = |T|$, $m = |Y|$, and let ℓ_{rec} be the maximum record length in T and Y. Assume n is a power of two and that priority-queue operations push and pop take amortized time $O(\log n)$. Since n nodes are pushed and popped in total, this contributes $O(n \log n)$ time. For inputs X of size a and Y of size b, $\text{ColCA}(X, Y)$ can be computed in expected time $O(\ell_{\text{rec}}(a + b))$. It is called once at the root and for half the nodes at each depth, yielding an expected total cost of $O(\ell_{\text{rec}} n(\log n + m))$. The heuristic p^+_{MK} adds only lower-order terms, so the overall expected runtime is $O(\ell_{\text{rec}} n(\log n + m))$.

Like the original PSI-CA attack, MKPM-SearchTree requires at most n protocol invocations, though this bound is loose due to early termination of DFS traversals (see Section 8 for an empirical evaluation). Even when the number of protocol invocations is restricted, partial intersection recovery remains feasible (see Appendix H).

[2]We use standard vector operations; addition and subtraction are applied element-wise.

7 T-Reconstruction with $\mathcal{F}_{\text{L-MKPM}}$

In this section, we describe three powerful attacks that exploit the "leaky" functionality $\mathcal{F}_{\text{L-MKPM}}$. Buddhavarapu et al. [5] acknowledge this leakage but deem it acceptable as an aggregate metric that would only raise concerns across multiple executions. In contrast, we present three attacks that achieve *T-reconstruction* (Definition 3.4) with a single protocol invocation.

We first present a baseline attack that is highly efficient but relies on a strong assumption on the protocol inputs. We then describe two additional attacks that progressively relax these assumptions.

7.1 The Baseline Attack

Recall that while the records in the adversary's input X are hidden and shuffled, the order of values *within* each record is preserved. We first observe that matching records between X and the leakage $\mathcal{L}^1_X$ is trivial when X contains one record. Exploiting this, we present BaseAttack, which flattens the target set T into an input X consisting of a single record $\mathbf{r}$ that contains all values in $\mathbb{V}(T)$.

The baseline attack constructs a dictionary D that maps hidden values appearing in the leakage $\mathcal{L}^1_X = \{\hat{\mathbf{x}}\}$ back to their original plaintext values in the input $X = \{\mathbf{x}\}$. Specifically, for all $j = 0, \ldots, |\hat{\mathbf{x}}|$, the adversary sets

$$\mathrm{D} : \hat{\mathbf{x}}[j] \mapsto \mathbf{x}[j].$$

It then uses D to recover plaintext values in the recovery set $\mathcal{L}^1_Y$. It initializes an empty multiset R_Y, and for each hidden record $\hat{\mathbf{y}}_i \in \mathcal{L}^1_Y$ it initializes an empty record $\mathbf{s}_i$. For all $j = 0, \ldots, |\hat{\mathbf{y}}_i|$, it checks whether $\hat{\mathbf{y}}_i[j] \in \mathrm{D}$; if so, it appends $\mathrm{D}[\hat{\mathbf{y}}_i[j]]$ to $\mathbf{s}_i$. Finally, it adds $\mathbf{s}_i$ to R_Y. In this way, the adversary recovers the plaintexts of all values in Y that are in common with those in T. This procedure yields a T-reconstruction R_Y of Y.

The pseudocode of BaseAttack is shown in Figure 6 (left). We also define an algorithm Substitute that takes a set of hidden records $\mathcal{L}^1_Y$ and a dictionary $D : G \to \mathbb{V}$ mapping hidden to plaintext values, and replaces the values in the records of $\mathcal{L}^1_Y$ with the corresponding entries in D. Identifiers that do not occur in D are not replaced. This algorithm is used as a subroutine in all attacks in this section and is given in Figure 6 (right).

Correctness of the attack directly follows from the fact that all values in the flattened target record are uniquely identifiable in a single invocation of $\mathcal{F}_{\text{L-MKPM}}$.

Time and Query Complexity. Let ℓ_{rec} denote the maximum record length in T and Y. Let $n = |T|$ and $m = |Y|$. The attack has an asymptotic time complexity of $O(\ell_{\text{rec}}(n + m))$. The attack requires only one query and therefore establishes a tight lower bound on the number of queries needed to achieve TR (Definition 3.4).

7.1.1 A First Optimization. The baseline attack relies on an atypical input structure—one very long record—which is both unnatural

$\textsc{BaseAttack}^{\mathcal{F}_{\text{L-MKPM}}(\cdot,Y)}(T)$

1 : $\mathbf{r} \leftarrow ()$
2 : **for** $\mathbf{t} \in T$ **do**
3 : **for** $i = 1, \ldots, |\mathbf{t}|$ **do**
4 : **if** $\mathbf{t}[i] \notin \mathbf{r}$ **then**
5 : $\mathbf{r} \leftarrow \mathbf{r} \| \mathbf{t}[i]$
6 : $X \leftarrow \{\mathbf{r}\}$
7 : $(\text{UID}, M_C, (\mathcal{L}_X^1, \mathcal{L}_Y^1))$ $\leftrightarrow\$ \mathcal{F}_{\text{L-MKPM}}(X, Y)$
8 : $\{\hat{\mathbf{r}}\} \leftarrow \mathcal{L}_X^1$
9 : Initialize dictionary $D \leftarrow \emptyset$
10 : **for** $i \in [0, |\hat{\mathbf{r}}| - 1]$ **do**
11 : $D[\hat{\mathbf{r}}[i]] \leftarrow r[i]$
12 : $R_Y \leftarrow \textsc{Substitute}(\mathcal{L}^1, D)$
13 : **return** R_Y

$\textsc{Substitute}(\mathcal{L}_Y^1, D)$

1 : Initialize $R \leftarrow \emptyset$
2 : **for** $\hat{\mathbf{y}} \in \mathcal{L}_Y^1$ **do**
3 : $\mathbf{r} \leftarrow ()$
4 : **for** $i \in [0, |\hat{\mathbf{y}}| - 1]$ **do**
5 : **if** $\hat{\mathbf{y}}[i] \in D$ **then**
6 : $\mathbf{r} \leftarrow \mathbf{r} \| D[\hat{\mathbf{y}}[i]]$
7 : $R \leftarrow R \cup \{\mathbf{r}\}$
8 : **return** R

Figure 6: Baseline attack with oracle access to the MKPM functionality with leakage $\mathcal{F}_{\text{L-MKPM}}(\cdot, Y)$. Y is part of the oracle and is unknown to the adversary.

$\mathbf{x}_0$:	v_0	v_0	alice@mail.com	+41 98 765 43 21
$\mathbf{x}_1$:	v_0	v_1	bob@mail.com	+41 12 345 67 89
$\mathbf{x}_2$:	v_1	v_0	eve@mail.com	+41 45 678 91 23
$\mathbf{x}_3$:	v_1	v_2	gus@mail.com	+41 56 789 12 34

(a) Target set T (boxed in blue) and input set X (the entire table).

$\mathbf{x}_2$:	e_0	e_1	e_2	e_3
$\mathbf{x}_1$:	e_1	e_0	e_4	e_5
$\mathbf{x}_3$:	e_0	e_6	e_7	e_8
$\mathbf{x}_0$:	e_1	e_1	e_9	e_{10}

(b) The hidden, shuffled leakage $\mathcal{L}_X^1$.

Figure 7: An example of a target set transformation by EnumAttack. (7a) shows the target set T (blue box) and the input set X, extended with prepended encoding (green box). Note the second value v_2 in the last record. (7b) shows the corresponding hidden, shuffled records $\mathcal{L}_X^1$. The hidden values corresponding to (v_0, v_0) are highlighted in green, and that corresponding to v_2 is highlighted in blue.

and easy to detect in practice. This motivates more refined attacks that achieve TR without such conspicuous inputs.

A first optimization avoids arbitrarily long records by exploiting that record lengths are preserved in the leakage. The values in $\mathbb{V}(T)$ can be partitioned into multiple records, each of a *unique* length. Let X' denote the resulting set of records and $\mathcal{L}_{X'}^1$ the corresponding leakage. Since each record has a distinct length and the order of values within records is preserved, the adversary can uniquely match each record in $\mathcal{L}_{X'}^1$ to the plaintext record of equal length in X', build a dictionary as before, and again achieve TR.

Time and Query Complexity. This optimized attack still uses one query and has time complexity $O(\ell_{\text{rec}}(n+m))$. The maximum record length decreases from $|\mathbb{V}(T)|$ to approximately $\sqrt{2|\mathbb{V}(T)|}$. While less conspicuous than the single-record variant, it remains easily detectable.

7.2 Record Enumeration Attack

We now present an attack whose input set consists of equal-length records. For simplicity, we assume that every record in the target set T has the same length; the attack extends directly to target sets with records of varying lengths.

We refer to this attack as the *Record Enumeration Attack*. It relies only on two properties of the functionality $\mathcal{F}_{\text{L-MKPM}}$: (i) the order of values within each record of $\tilde{P}_1$'s input is preserved in the leakage, and (ii) values are hidden deterministically, so that repeated occurrences of the same value yield the same hidden value within a single protocol execution. The attack proceeds by enumerating the records in T and prepending to each record an encoding of its index using known values. If this enumeration can be recovered from the leakage, then the adversary can compute a dictionary D from hidden to plaintext values and subsequently use D to output a T-reconstruction of the recovery set Y.

Without loss of generality, assume that $|T|$ is not a power of two, and let $v_0, v_1 \in \mathbb{V}$ be two distinct values. Fix an ordering $(\mathbf{t}_0, \ldots, \mathbf{t}_{n-1})$ of the records in T, where $\mathbf{t}_i$ denotes the i-th record. Each index $i \in [0, n-1]$ is encoded as a binary string of length $\ell_{\text{enc}} := \lceil \log_2 n \rceil$, where bits 0 and 1 are represented by v_0 and v_1, respectively. This binary encoding is prepended to $\mathbf{t}_i$, and we denote by X the resulting set of records.

We now analyze the output to $\tilde{P}_1$ when $\mathcal{F}_{\text{L-MKPM}}(\cdot\,; Y)$ is invoked on X. Recall that $\mathcal{F}_{\text{L-MKPM}}$ hides values via an injective random function f_X while preserving the order of values within each record of X. Consequently, for any $\hat{\mathbf{x}} \in \mathcal{L}_X^1$, the first ℓ_{enc} hidden values correspond to the encoded prefix and therefore comprise only two hidden values, i.e., e and e'. Moreover, we have either $e = f_X(v_0)$ and $e' = f_X(v_1)$, or $e = f_X(v_1)$ and $e' = f_X(v_0)$.

By distinguishing between these two cases, the adversary can recover the index encoded in the prefix of each $\hat{\mathbf{x}} \in \mathcal{L}_X^1$ and establish a correspondence between the hidden records in $\mathcal{L}_X^1$ and the plaintext records in X. To determine whether a hidden value e corresponds to v_0 or v_1, observe that since n is not a power of two and $|T| < 2^{\ell_{\text{enc}}}$, no record is assigned index $2^{\ell_{\text{enc}}} - 1$, and the corresponding encoding $(v_1, \ldots, v_1)$ is not used. This serves as a tiebreaker: the encoding of index 0 is the only encoding composed of a single value, namely $(v_0, \ldots, v_0)$.

Without loss of generality, suppose that $e = f_X(v_0)$. The decoding procedure first identifies the hidden record in $\mathcal{L}_X^1$ whose first ℓ_{enc} values are all equal; denote this record by $\hat{\mathbf{x}}_0$. Since values are not permuted within records, we must have $\hat{\mathbf{x}}_0[0] = f_X(v_0) = e$. As before, the adversary can now compute a dictionary that maps hidden values to plaintext values. It initializes an empty dictionary D. Then using the encoded prefix of each $\hat{\mathbf{x}} \in \mathcal{L}_X^1$, the adversary recovers the corresponding index $i \in [0, n-1]$. For each $\hat{\mathbf{x}} \in \mathcal{L}_X^1$ and each $j = 0, \ldots, |\hat{\mathbf{x}}| - 1$, it sets $\mathrm{D} : \hat{\mathbf{x}}[j] \mapsto \mathbf{x}_i[j]$, where $\mathbf{x}_i$ is the

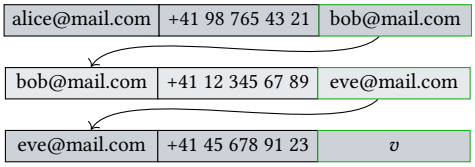


Figure 8: Result of preprocessing a set of records in SnakeAttack. The first attribute is unique across all records. The original records have black borders. Added values are outlined in green. Arrows indicate the (implicit) linked list.

record in X with index i. As before, this dictionary, together with $\mathcal{L}_Y^1$, can be used to compute a T-reconstruction R_Y.

The assumption that n is not a power of two can be dropped. The adversary can introduce a third tie-breaking value $v_2 \notin \{v_0, v_1\}$ and modify the encoding of the index $2^\ell - 1$ by replacing the last symbol in $(v_1, \ldots, v_1)$ with v_2. This guarantees a unique, identifiable prefix and removes the ambiguity between the encodings of 0 and $2^\ell - 1$. We adopt this approach in our full attack. An example of a target set and its augmented input is shown in Figure 7a. Its resulting leakage is shown in Figure 7b.

The pseudocode of EnumAttack is given in Figure 20.

Theorem 7.1. *EnumAttack$^{\mathcal{F}_{\text{L-MKPM}}(\cdot,Y)}(T)$ (Figure 20) achieves T-reconstruction of Y (Def. 3.4).*

The proof of Theorem 7.1 can be found in Appendix G.3.

Time and Query Complexity. Assuming that the binary encoding can be computed in time $O(\log n)$, then computing the augmented input set takes $O(n \log n)$. Identifying $\hat{\mathbf{x}}_0$ also incurs time $O(n \log n)$. Constructing the dictionary D takes time $O(n(\log n + \ell_{rec}))$, while computing R_Y using D and $\mathcal{L}_Y^1$ requires time $O(\ell_{\text{rec}} m)$.

Overall, the attack runs in time $O(n \log n + \ell_{\text{rec}}(n + m))$ and requires only a single query.

7.2.1 Generalizing to base-k. We note that while our attack uses as base-2 (binary) encoding, one could also generalize this approach to base-k for any integer $k \geq 2$. This would naturally reduce the length of the encoding needed at the expense of more complex encoding/decoding procedure. Care must be taken to avoid encodings that cannot be uniquely recovered e.g., avoiding use of the $(v_i, \ldots, v_i)$ for $i > 0$. We leave the details for future work.

7.3 Snake Attack

While EnumAttack improves over BaseAttack in terms of detectability, it still requires inputs whose record lengths grow with the target set size. We address this limitation by imposing an order on the records without explicitly assigning a unique index to each one. Concretely, we extend each record by only two values; these values form a chain of pointers that enables the adversary to reconstruct a total order on the records.

The attack starts by checking whether there exists a column (attribute) j^* in T such that every value in that column is unique, i.e., $|\{\mathbf{t}[j^*] \mid \mathbf{t} \in T\}| = |T|$. This is a reasonable assumption, as many values (e.g., email addresses) are typically unique per user. If no such column exists, the adversary can create one by sampling n distinct values in $\mathbb{V}$ and prepending one value to each record in T (in which case $j^* = 0$).

The adversary links the records of T by fixing an arbitrary ordering $(\mathbf{t}_0, \ldots, \mathbf{t}_{n-1})$ and appending the unique value of the $(i+1)$-st record to the i-th record for all $i \in [n-1]$. This yields an input set $X = \{\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}\}$ such that $\mathbf{t}_i$ corresponds to $\mathbf{x}_i$ in an (implicit) linked list. Party $\tilde{P}_1$ then invokes the functionality $\mathcal{F}_{\text{L-MKPM}}(\cdot; Y)$ on X to obtain the output. See Figure 8 for a small example.

Since the hiding function f_X used in $\mathcal{F}_{\text{L-MKPM}}$ is injective and deterministic, the pointers in X are preserved in the corresponding leakage $\mathcal{L}_X^1$. The adversary can thus locate the start of the list as the record whose j^*-th value is not appended to any other record (i.e., the head of the linked list), and traverse the linked list to recover the original ordering of X.

Formally, given the leaked set $\mathcal{L}_X^1$ and the column index j^*, the adversary can reorder the elements of $\mathcal{L}_X^1$ into a list $(\mathbf{s}_0, \ldots, \mathbf{s}_{n-1})$ as follows: $\mathbf{s}_0$ is the unique record whose j^*-th value does not appear as an appended "next" value in any other record, and for each $i \geq 1$, $\mathbf{s}_i$ is the unique record whose j^*-th value equals the appended pointer in $\mathbf{s}_{i-1}$. Thus, $\mathbf{s}_i$ corresponds to the i-th plaintext record in the linked list.

The adversary then initializes an empty dictionary D and, for each $i = 0, \ldots, n-1$ and each $j = 0, \ldots, |\mathbf{s}_i| - 1$, sets $\mathsf{D} : \mathbf{s}_i[j] \mapsto \mathbf{x}_i[j]$. The adversary can once again apply this dictionary to $\mathcal{L}_Y^1$ to recover a T-reconstruction of Y.

The pseudocode of SnakeAttack is given in Figure 21.

Theorem 7.2. *SnakeAttack$^{\mathcal{F}_{\text{L-MKPM}}(\cdot,Y)}(T)$ (Figure 20) achieves T-reconstruction of Y (Def. 3.4).*

The proof of Theorem 7.2 can be found in Appendix G.5.

Time and Query Complexity. Let ℓ_{rec} denote the length of the largest record in T and Y. Determining whether a column in T contains only unique values can be done in $O(\ell_{\text{rec}} n)$ time. Constructing the (implicit) linked list has an average-case time complexity of $O(n)$, as does identifying the linked list in the leakge. Constructing the dictionary involves up to $\ell_{\text{rec}} n$ writes. Computing R_Y using D and $\mathcal{L}_Y^1$ requires time $O(\ell_{\text{rec}} m)$. The overall average-case time complexity for SnakeAttack is $O(\ell_{\text{rec}}(n + m))$. Similar to the previous attacks, it requires only one protocol invocation.

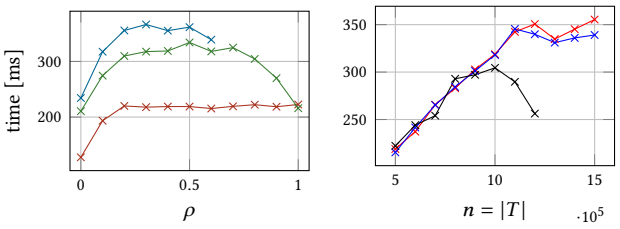
8 Experimental Evaluation

Experimental Setup. We implemented our attacks in Python 3.12.3 and ran the experiments on a 64-core AMD EPYC 7742 2.25 GHz processor with 512 GB memory. Each experiment was executed on a single physical core, and our test suite (which ran 40–50 experiments in parallel) did not exceed 10 GB of memory.

We simulated the protocol on the same node on which we executed the attacks. Our reported time measurements exclude network latencies and functionality evaluation; the reported times thus reflect only the time required to carry out the attack. Each experiment was repeated 50 times, and we report average runtime and query counts across runs.

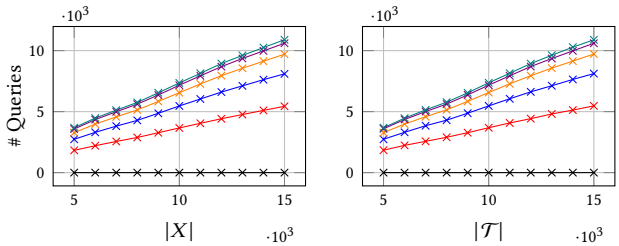
8.1 Data Generation

Our attacks are largely data-independent. For ethical reasons, we evaluated the attacks using synthetic data, which also allowed



(a) PSU-Diff (b) PSU-Diff

Figure 9: Runtime of PSU-Diff for a recovery set of size $|Y| = 10^6$. (a) Runtimes vs. intersection ratio ρ for target set sizes $0.5 \cdot 10^6$, 10^6, $1.5 \cdot 10^6$. (b) Runtimes vs. target set size $|T|$ for ρ values 40%, 60%, 80%.



(a) PSU-CA-SearchTree (b) MKPM-SearchTree

Figure 10: The number of queries required by the (a) PSU-CA-SearchTree and (b) MKPM-SearchTree plotted against the target set size for intersection ratios ρ and match rates η: 0%, 10%, 20%, 30%, 40% and 50%. The experiment was run with $|Y| = 10^4$.

precise control over the test data parameters. We note that Falzon and Tang [20] also used synthetic datasets in their evaluation.

PSU and PSU-CA. Given the high efficiency of our PSU-Diff attack, we tested it with a recovery set of size $m = 10^6$. For the $\mathcal{F}_{\text{PSU-CA}}$ attack, we used $m = 10^4$. The target set size n ranged from 50% to 150% of m in 10% increments.

We varied the intersection size as a fraction of T,

$$\rho := |T \cap Y|/n,$$

from 0% to 100% in 10% increments. When $n > m$, we only measured up to the largest feasible fraction that is a multiple of 10%.

To generate T, we sampled n elements without replacement from the interval $[0, 2n)$. To construct Y, we first sampled ρn elements from T and then sampled the remaining $m - \rho n$ elements without replacement from the interval $[2n, 2(n + m - \rho n))$.

MKPM and L-MKPM. The $\mathcal{F}_{\text{MKPM}}$ and $\mathcal{F}_{\text{L-MKPM}}$ functionalities take as input sets of records. Each record in our datasets consist of an email address, a phone number, an IP address, and an advertising ID (AAID).[3] The first three values were generated using the `Faker` Python package [19], while the AAID was a randomly sampled string of 32 hexadecimal digits.

For a given pair of target and recovery sets T and Y, we define the match rate η as the fraction of all values in T that also appear in Y, including multiple occurrences of the same value. Specifically, for $N := \sum_{\mathbf{t} \in T} |\mathbf{t}|$, we have:

$$\eta := \left(\sum_{\mathbf{t} \in T} \sum_{v \in \mathbf{t}} \mathbb{1}\{v \in \mathbb{V}(Y)\} \right) / N.$$

While η describes the overall match rate of T, the match rates for individual columns of T may vary.

In our experiments with $\mathcal{F}_{\text{MKPM}}$ and $\mathcal{F}_{\text{L-MKPM}}$, the recovery set contained $m = 10^4$ records. We varied η from 0% to 100% in 10% increments. When $n > m$, we measured up to the largest feasible fraction divisible by 10%. The target set size n ranged from 50% to 150% of m, increasing in 10% increments.

[3]This set of values follows Meta's Developer Reference: https://developers.facebook.com/docs/private-computation/reference/

8.2 PSU Attack Evaluation

Overall, the PSU-Diff attack (Section 4.1) is very efficient, as shown in Figure 9. On a large recovery set with $|Y| = 10^6$ elements, runtimes range from 127 ms to 367 ms. The measurements largely follow from Python's set-intersection implementation, which is linear in the size of the smaller operand.

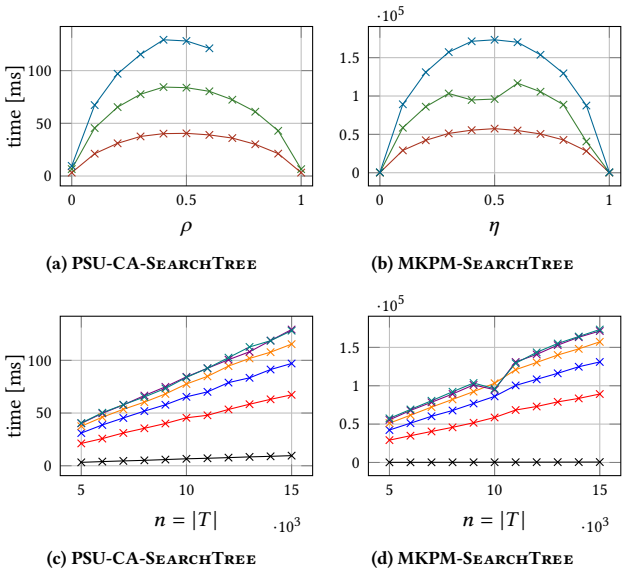
Figure 9a shows the runtimes plotted against the intersection ratio ρ. For large target sets, larger values of ρ lead to a decrease in runtimes , since the increase of the shared elements between T and Y results in smaller sets $Z_i = X_i \cup Y$; this makes the computation of $Y = Z_1 \cap Z_2$ more efficient. For smaller target sets, the runtime seems to be dominated by the computation of the intersection $T \cap Y$, which is constant. The initial increase in runtimes for small values of ρ does not fit this explanation, but may be attributed to a smaller number of memory allocations during the computation of $T \cap Y$. Confirming this would require more extensive testing.

Figure 9b plots runtimes against the target set size $|T|$. The runtime is dominated by computing the intersection $T \cap Y$, which grows linearly in $|T|$ as long as $|T| \leq |Y|$ and stagnates after that. The reason for the decreased runtimes for larger values of ρ and target sets with $|T| > 10^6$ is unclear, but preliminary tests suggest that the affected data sets exhibit a better cache locality.

8.3 Search Tree Attacks Evaluation

We now evaluate our search tree attacks: PSU-CA-SearchTree (Figure 14) and MKPM-SearchTree (Figure 17). We used the same set sizes across experiments for both attacks. We plot the number of queries in Figures 10 and 18, and their runtimes in Figure 11.

The time and query measurements of the two attacks are largely symmetric with respect to the intersection ratio and match rate. Specifically, for any ε, measurements for experiments with $\rho, \eta \in \{\varepsilon, 1 - \varepsilon\}$ are nearly identical. We therefore only report measurements for $\rho, \eta \leq 0.5$. This symmetry follows from the number of queries performed by the attacks. Isolating a small set of matched elements from a large set of unmatched ones (small intersection ratio) is equivalent to isolating a large set of matched elements from a small set of unmatched ones (large intersection ratio). In both cases, the expected number of queries is the same.



(a) PSU-CA-SearchTree (b) MKPM-SearchTree

(c) PSU-CA-SearchTree (d) MKPM-SearchTree

Figure 11: Runtimes of the two search-tree attacks. (a,b) Runtimes plotted against the intersection ratio ρ and match rate η for target set sizes $0.5 \cdot 10^4$, 10^4, $1.5 \cdot 10^4$. (c,d) Runtimes plotted against the target set size $n = |T|$ for intersection ratios ρ and match rates η: 0%, 10%, 20%, 30%, 40%, 50%.

8.3.1 Queries. Figures 10 and 18 (Appendix H) plot the number of queries required by each attack as functions of the target set size, and intersection ratio ρ and match rate η, respectively. Figure 10 shows that the number of queries grows linearly with the target set size, matching our linear theoretical bounds. In Figure 18, we observe that values of ρ and η near 0.5 incur the largest number of queries, and that this is consistent across target set sizes.

Both attacks make almost the same number of queries, despite MKPM-SearchTree having a stricter early-termination criterion (i.e., to avoid pushing a node onto the stack, its record set must not share any values with the recovery set). In all experiments, the attacks' difference never exceeds 0.5% of the theoretical upper bound. The exception is when ρ or η is 0 or 1: MKPM-SearchTree issues only one query, whereas PSU-CA-SearchTree always issues three queries. This is because PSU-CA-SearchTree requires two additional queries to determine $|Y|$.

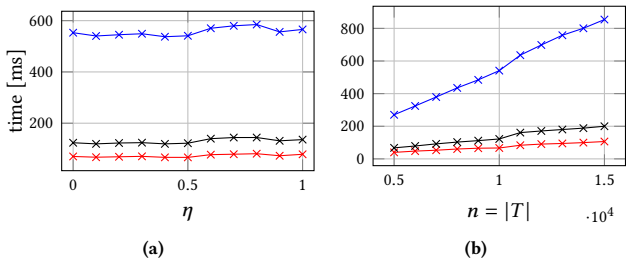
The heuristic used to choose which child to follow during path traversal and which subset to process next after termination, has only a minimal impact on the number of queries. This is expected, as the attacks terminate only once all subsets in the priority queue have been processed. The number of subsets in the queue depends solely on the (random) partitioning, not on the heuristic. In this subsection, we thus only report the measurements for the positive heuristics ($p^+_{\text{PSI-CA}}$ and p^+_{MK}) in this section.

8.3.2 Runtime. Figure 11 shows the runtimes of both attacks as a function of the intersection ratio ρ or match rate η and the target set size. PSU-CA-SearchTree runs in 3 ms to 130 ms, depending on the parameters. MKPM-SearchTree is markedly slower, with runtimes from 145 ms ($\eta = 0$ and $n = 0.5 \cdot 10^4$) up to 177 s ($\eta = 0.5$ and $n = 1.5 \cdot 10^4$).

Although the two attacks issue almost the same number of queries, MKPM-SearchTree is more than three orders of magnitude slower than PSU-CA-SearchTree. We attribute this to the computation of ColCA, which is recomputed after every query and requires scanning all values in $\mathcal{L}^1_Y$ and the current target subset.

Figures 11a and 11b demonstrate that runtimes are significantly influenced by ρ and η, peaking for values near 0.5, which aligns with our prior observation that this regime involves the highest query counts. Meanwhile, Figures 11c and 11d indicate that runtimes for both attacks scale roughly linearly with target set size, despite theoretical complexities of $O(n \log n)$ for PSU-CA-SearchTree and $O(\lambda n(\log n + m))$ for MKPM-SearchTree. We believe the logarithmic overhead from priority-queue operations only becomes noticeable in larger data sets.

In Appendix H, we present additional experiments conducted under limited query budgets. Our findings reveal that adversaries can reconstruct substantial information even with a reduced number of protocol invocations.



(a) (b)

Figure 12: Runtime measurements of BaseAttack, EnumAttack and SnakeAttack plotted against (a) the match rate η and (b) the target set size.

8.4 L-MKPM Attack Evaluation

The T-reconstruction attacks presented in Section 7—BaseAttack, EnumAttack, and SnakeAttack—all require one protocol execution. We thus only discuss the runtime measurements, which are plotted against the match rate η and target set size n in Figure 12.

The attacks are all highly efficient. BaseAttack is the fastest, with runtimes from 38 ms ($n = 0.5 \cdot 10^4$) to 112 ms ($n = 1.5 \cdot 10^4$). SnakeAttack is 1.5–2× slower, ranging from 66 ms to 214 ms, due to simpler pre-processing but costlier post-processing. EnumAttack is the slowest, with runtimes from 266 ms to 884 ms, because its encoding and decoding operations involve prepending and inspecting $\log_2 n$ values per record.

Figure 12a shows that all three attack runtimes are largely independent of the match rate. This is consistent with theory since match rate affects only the reconstruction dictionary size. Figure 12b shows that BaseAttack and SnakeAttack runtimes increase linearly with target set size, matching their theoretical time complexity. We also see that EnumAttack incurs the longest runtimes and steepest growth as n increases.

9 Discussion

Re-thinking Security. Standard security models in the MPC literature do not explicitly account for inference attacks arising from functionality outputs. Such attacks are often handled only indirectly, for example, through set-size thresholds. However, our analysis of $\mathcal{F}_{\text{PSU}}$, $\mathcal{F}_{\text{PSU-CA}}$, and Meta's multi-key private matching functionality $\mathcal{F}_{\text{L-MKPM}}$, together with prior work [20, 28, 34, 54], underscores the need for a more systematic treatment of MPC protocol security. The functionality outputs should be explicitly incorporated into the security model. Failing to do so results in weak input privacy guarantees: in the worst case, a single query may suffice to partially or even fully recover another party's input.

Modified Ideal Functionalities. Since these ideal functionalities inherently reveal some information about the inputs through their outputs, the associated risks are unavoidable. In response, various "mitigations" have been proposed for PSI and PSU, including applying rate limiting, restricting the input domain, and adding noise to outputs. Since these approaches modify the original functionality, we refer to them collectively as modified ideal functionalities.

We discuss the implications of each modification, but it remains unclear to what extent they are truly resilient to inference attacks. As a result, we need better models for understanding and quantifying input leakage, which we view as an important direction for future work.

First, rate limiting helps against attacks that require some lower bound $\omega(1)$ queries (e.g., our search-tree attacks and prior work [20, 28, 34]). However, two concerns arise. First, attacks typically improve over time, so only information-theoretic query lower bounds—and not the current best-known attack—should inform "query budgets." Second, rate limits can be circumvented via Sybil attacks, which are often outside the protocol threat model but represent a very real risk.

Second, one may enforce that inputs lie within a certain input domain. For example, ACORNS [3] use range proofs to ensure that inputs are within the specified input domain. Goel et al. introduced PICS [26], a PSI framework in which protocol participants must first publicly commit to their input sets. The maliciously-secure PSI protocol is augmented with zero-knowledge proofs to ensure that the computation is consistent with the committed inputs across multiple executions. However, such restrictions do not rule out inference attacks using valid inputs, as shown in [20].

Third, one may add noise to the output. For example, Differential Privacy (DP) [18] provides a formal framework for adding noise and enforcing query budgets to guarantee a weak form of input privacy, namely membership privacy. Differentially private set operations [35], as well as a differentially private record-matching functionality [29], have already been proposed. Care must still be taken in selecting suitable parameters and query budgets. Indeed, some of the attacks against PSI-CA in [34] were evaluated against a differentially private PSI-CA variant and were found to remain effective despite the use of DP.

Looking Towards Deployment. Recent works have sought to extend two-party joins to multiple parties [39, 43] and the delegated compute setting [44]; such variants typically rely on a non-collusion assumptions between parties. At the same time, we observe growing adoption of MPC technologies by large organizations (e.g., Google's Private Join and Compute [31]).

This leads to a critical question: if attacks that leverage adversarially chosen inputs are within scope, in which application scenarios do they not arise? The intended scope and trust assumptions (e.g., non-collusion) should be made explicit at the time of proposal; otherwise, security proofs risk having limited practical relevance and may be misapplied.

In practice, non-collusion assumptions may fail—for example, under subpoena power, regulatory pressure, or single-administrator access—leading to risks similar to those identified in our analysis. Clearly communicating these assumptions and their implications is therefore essential for responsible deployment and for avoiding a false sense of privacy among users.

Ethical Considerations

This work assumes intended use of the PSU, PSU-CA, and Meta's MKPM protocols. Our analysis leverages intrinsic vulnerabilities in the functionality outputs and does not break the protocols themselves or the underlying cryptographic primitives.

Use of Synthetic Data. Our experiments use only synthetic data, generated either by (a) sampling numerical values from integer intervals or (b) using the Python Faker library [19]. This ensures that no personally identifying or sensitive data is leaked and eliminates any risk of harm to individuals or groups.

Disclosure. We contacted the authors of the MKPM work [5] on 27.02.2026, to share our findings and a preprint of this work. We received an official response on 02.03.2026 from the developers acknowledging the leakage of the protocol. They further stated that MKPM is not currently being used in any production systems.

Our analysis was thus conducted on the functionality of a protocol that, at the time of writing, is not deployed and therefore does not pose any negative impacts on live systems or stakeholders.

Open Science

In accordance with the open science policy, our code base has undergone artifact evaluation. Our artifact contains the code for our attacks and for generating the synthetic data, as well as instructions on how to run the code. The open source respository can be found here: https://github.com/deRaguso/mkpid-attacks-artifact.

Acknowledgments

Francesca Falzon was supported by the Zurich Information Security and Privacy Center (ZISC). Tianxin Tang was partially supported by an NWO VIDI grant (Project No. VI.Vidi.193.066). The authors would also like to thank the reviewers for their helpful feedback, including their suggestion for simplifying the PSU attack. The authors acknowledge the use of generative AI tools solely for basic editing of the text and grammar.

References

[1] Rakesh Agrawal, Alexandre Evfimievski, and Ramakrishnan Srikant. 2003. Information sharing across private databases. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (San Diego, California) *(SIGMOD '03)*. Association for Computing Machinery, New York, NY, USA, 86–97. https://doi.org/10.1145/872757.872771

[2] Gilad Asharov, Koki Hamada, Ryo Kikuchi, Ariel Nof, Benny Pinkas, and Junichi Tomida. 2023. Secure Statistical Analysis on Multiple Datasets: Join and Group-By. In *ACM CCS 2023*, Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda (Eds.). ACM Press, 3298–3312. https://doi.org/10.1145/3576915.3623119

[3] James Bell, Adrià Gascón, Tancrède Lepoint, Baiyu Li, Sarah Meiklejohn, Mariana Raykova, and Cathie Yun. 2023. ACORN: Input Validation for Secure Aggregation. In *USENIX Security 2023*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, 4805–4822. https://www.usenix.org/conference/usenixsecurity23/presentation/bell

[4] Marina Blanton and Everaldo Aguiar. 2012. Private and oblivious set and multiset operations. In *ASIACCS 12*, Heung Youl Youm and Yoojae Won (Eds.). ACM Press, 40–41. https://doi.org/10.1145/2414456.2414479

[5] Prasad Buddhavarapu, Benjamin M Case, Logan Gore, Andrew Knox, Payman Mohassel, Shubho Sengupta, Erik Taubeneck, and Min Xue. 2021. Multi-key Private Matching for Compute. Cryptology ePrint Archive, Paper 2021/770. https://eprint.iacr.org/2021/770

[6] Prasad Buddhavarapu, Andrew Knox, Payman Mohassel, Shubho Sengupta, Erik Taubeneck, and Vlad Vlaskin. 2020. Private Matching for Compute. Cryptology ePrint Archive, Paper 2020/599. https://eprint.iacr.org/2020/599

[7] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas A. Dimitropoulos. 2010. SEPIA: Privacy-Preserving Aggregation of Multi-Domain Network Events and Statistics. In *USENIX Security 2010*. USENIX Association, 223–240. http://www.usenix.org/events/sec10/tech/full_papers/Burkhart.pdf

[8] Gowri R. Chandran, Thomas Schneider, Maximilian Stillger, and Christian Weinert. 2025. Concretely Efficient Private Set Union via Circuit-Based PSI. In *ASIACCS 25*. ACM Press, 149–162. https://doi.org/10.1145/3708821.3710839

[9] Melissa Chase and Peihan Miao. 2020. Private Set Intersection in the Internet Setting from Lightweight Oblivious PRF. In *CRYPTO 2020, Part III (LNCS, Vol. 12172)*, Daniele Micciancio and Thomas Ristenpart (Eds.). Springer, Cham, 34–63. https://doi.org/10.1007/978-3-030-56877-1_2

[10] Hao Chen, Kim Laine, and Peter Rindal. 2017. Fast Private Set Intersection from Homomorphic Encryption. In *ACM CCS 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM Press, 1243–1255. https://doi.org/10.1145/3133956.3134061

[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. *CoRR* abs/2506.01072 (2025). https://doi.org/10.48550/ARXIV.2506.01072 arXiv:2506.01072

[12] Koji Chida, Koki Hamada, Atsunori Ichikawa, Masanobu Kii, and Junichi Tomida. 2023. Communication-Efficient Inner Product Private Join and Compute with Cardinality. In *ASIACCS 23*, Joseph K. Liu, Yang Xiang, Surya Nepal, and Gene Tsudik (Eds.). ACM Press, 678–688. https://doi.org/10.1145/3579856.3582826

[13] Coinbase. 2026. What is a Multi-Party Computation (MPC) wallet? Cryptology ePrint Archive, Paper 2020/599. https://www.coinbase.com/learn/wallet/what-is-a-multi-party-computation-mpc-wallet Accessed: February 24, 2026.

[14] Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. 2012. Fast and Private Computation of Cardinality of Set Intersection and Union. In *CANS 12 (LNCS, Vol. 7712)*, Josef Pieprzyk, Ahmad-Reza Sadeghi, and Mark Manulis (Eds.). Springer, Berlin, Heidelberg, 218–231. https://doi.org/10.1007/978-3-642-35404-5_17

[15] Emiliano De Cristofaro and Gene Tsudik. 2010. Practical Private Set Intersection Protocols with Linear Complexity. In *FC 2010 (LNCS, Vol. 6052)*, Radu Sion (Ed.). Springer, Berlin, Heidelberg, 143–159. https://doi.org/10.1007/978-3-642-14577-3_13

[16] Changyu Dong, Liqun Chen, and Zikai Wen. 2013. When private set intersection meets big data: an efficient and scalable protocol. In *ACM CCS 2013*, Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung (Eds.). ACM Press, 789–800. https://doi.org/10.1145/2508859.2516701

[17] Minglang Dong, Cong Zhang, Yujie Bai, and Yu Chen. 2025. Efficient Multi-Party Private Set Union Without Non-Collusion Assumptions. In *USENIX Security 2025*, Lujo Bauer and Giancarlo Pellegrino (Eds.). USENIX Association, 2005–2024. https://www.usenix.org/conference/usenixsecurity25/presentation/dong-minglang

[18] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. 2006. Calibrating Noise to Sensitivity in Private Data Analysis. In *TCC 2006 (LNCS, Vol. 3876)*, Shai Halevi and Tal Rabin (Eds.). Springer, Berlin, Heidelberg, 265–284. https://doi.org/10.1007/11681878_14

[19] Faker Community. 2026. Faker. https://github.com/joke2k/faker. Accessed: 2026-02-27.

[20] Francesca Falzon and Tianxin Tang. 2025. Learning from Functionality Outputs: Private Join and Compute in the Real World. In *USENIX Security 2025*, Lujo Bauer and Giancarlo Pellegrino (Eds.). USENIX Association, 7603–7622. https://www.usenix.org/conference/usenixsecurity25/presentation/falzon

[21] Ellis Fenske, Akshaya Mani, Aaron Johnson, and Micah Sherr. 2017. Distributed Measurement with Private Set-Union Cardinality. In *ACM CCS 2017*, Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM Press, 2295–2312. https://doi.org/10.1145/3133956.3134034

[22] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. 2004. Efficient Private Matching and Set Intersection. In *EUROCRYPT 2004 (LNCS, Vol. 3027)*, Christian Cachin and Jan Camenisch (Eds.). Springer, Berlin, Heidelberg, 1–19. https://doi.org/10.1007/978-3-540-24676-3_1

[23] Keith B. Frikken. 2007. Privacy-Preserving Set Union. In *ACNS 2007 (LNCS, Vol. 4521)*, Jonathan Katz and Moti Yung (Eds.). Springer, Berlin, Heidelberg, 237–252. https://doi.org/10.1007/978-3-540-72738-5_16

[24] Jiahui Gao, Son Nguyen, Marina Blanton, and Ni Trieu. 2025. PULSE: Parallel Private Set Union for Large-Scale Entities. In *ACM CCS 2025*, Chun-Ying Huang, Jyh-Cheng Chen, Shiuh-Pyng Shieh, David Lie, and Véronique Cortier (Eds.). ACM Press, 1784–1798. https://doi.org/10.1145/3719027.3765108

[25] Jiahui Gao, Son Nguyen, and Ni Trieu. 2024. Toward A Practical Multi-party Private Set Union. *PoPETs* 2024, 4 (Oct. 2024), 622–635. https://doi.org/10.56553/popets-2024-0133

[26] Aarushi Goel, Peihan Miao, Phuoc Van Long Pham, and Satvinder Singh. 2026. PICS: Private Intersection over Committed (and reusable) Sets. In *Proceedings of the 35th USENIX Conference on Security Symposium* (Baltimore, MD, USA). USENIX Association, USA.

[27] Google. 2019. Helping Organizations Do More without Collecting More Data. https://security.googleblog.com/2019/06/helping-organizations-do-more-without-collecting-more-data.html. Accessed: February 21, 2026.

[28] 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*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 1487–1504. https://www.usenix.org/conference/usenixsecurity22/presentation/guo

[29] Ali Inan, Murat Kantarcioglu, Gabriel Ghinita, and Elisa Bertino. 2010. Private record matching using differential privacy. In *Proceedings of the 13th International Conference on Extending Database Technology* (Lausanne, Switzerland) *(EDBT '10)*. Association for Computing Machinery, New York, NY, USA, 123–134. https://doi.org/10.1145/1739041.1739059

[30] 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

[31] 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 (EuroS&P)*. 370–389. https://doi.org/10.1109/EuroSP48549.2020.00031

[32] Yanxue Jia, Shifeng Sun, Hong-Sheng Zhou, Jiajun Du, and Dawu Gu. 2022. Shuffle-based Private Set Union: Faster and More Secure. In *USENIX Security 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 2947–2964. https://www.usenix.org/conference/usenixsecurity22/presentation/jia

[33] Yanxue Jia, Shi-Feng Sun, Hong-Sheng Zhou, and Dawu Gu. 2024. Scalable Private Set Union, with Stronger Security. In *USENIX Security 2024*, Davide Balzarotti and Wenyuan Xu (Eds.). USENIX Association. https://www.usenix.org/conference/usenixsecurity24/presentation/jia-yanxue

[34] Bo Jiang, Jian Du, and Qiang Yan. 2024. AnonPSI: An Anonymity Assessment Framework for PSI. In *NDSS 2024*. The Internet Society.

[35] Bailey Kacsmar, Basit Khurram, Nils Lukas, Alexander Norton, Masoumeh Shafieinejad, Zhiwei Shang, Yaser Baseri, Maryam Sepehri, Simon Oya, and Florian Kerschbaum. 2020. Differentially Private Two-Party Set Operations. In *2020 IEEE European Symposium on Security and Privacy*. IEEE Computer Society Press, 390–404. https://doi.org/10.1109/EuroSP48549.2020.00032

[36] Florian Kerschbaum. 2012. Outsourced private set intersection using homomorphic encryption. In *ASIACCS 12*, Heung Youl Youm and Yoojae Won (Eds.). ACM Press, 85–86. https://doi.org/10.1145/2414456.2414506

[37] Lea Kissner and Dawn Xiaodong Song. 2005. Privacy-Preserving Set Operations. In *CRYPTO 2005 (LNCS, Vol. 3621)*, Victor Shoup (Ed.). Springer, Berlin, Heidelberg, 241–257. https://doi.org/10.1007/11535218_15

[38] Vladimir Kolesnikov, Mike Rosulek, Ni Trieu, and Xiao Wang. 2019. Scalable Private Set Union from Symmetric-Key Techniques. In *ASIACRYPT 2019, Part II (LNCS, Vol. 11922)*, Steven D. Galbraith and Shiho Moriai (Eds.). Springer, Cham, 636–666. https://doi.org/10.1007/978-3-030-34621-8_23

[39] Anja Lehmann, Christian Mouchet, and Andrey Sidorenko. 2026. Multi-Party Private Join. *Proc. Priv. Enhancing Technol.* 2026 (2026).

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

[41] Xiang Liu and Ying Gao. 2023. Scalable Multi-party Private Set Union from Multi-query Secret-Shared Private Membership Test. In *ASIACRYPT 2023, Part I (LNCS, Vol. 14438)*, Jian Guo and Ron Steinfeld (Eds.). Springer, Singapore, 237–271. https://doi.org/10.1007/978-981-99-8721-4_8

[42] Meta. 2026. Private-ID. https://github.com/facebookresearch/Private-ID. Accessed: February 21, 2026.

[43] Payman Mohassel, Peter Rindal, and Mike Rosulek. 2020. Fast Database Joins and PSI for Secret Shared Data. In *ACM CCS 2020*, Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna (Eds.). ACM Press, 1271–1287. https://doi.org/10.1145/3372297.3423358
[44] Dimitris Mouris, Daniel Masny, Ni Trieu, Shubho Sengupta, Prasad Buddhavarapu, and Benjamin M. Case. 2024. Delegated Private Matching For Compute. *PoPETs* 2024, 2 (April 2024), 49–72. https://doi.org/10.56553/popets-2024-0040
[45] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. 2019. SpOT-Light: Lightweight Private Set Intersection from Sparse OT Extension. In *CRYPTO 2019, Part III (LNCS, Vol. 11694)*, Alexandra Boldyreva and Daniele Micciancio (Eds.). Springer, Cham, 401–431. https://doi.org/10.1007/978-3-030-26954-8_13
[46] Benny Pinkas, Thomas Schneider, and Michael Zohner. 2014. Faster Private Set Intersection Based on OT Extension. In *USENIX Security 2014*, Kevin Fu and Jaeyeon Jung (Eds.). USENIX Association, 797–812. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/pinkas
[47] Lucas Piske and Ni Trieu. 2026. Is PSI Really Faster Than PSU? Achieving Efficient PSU with Invertible Bloom Filters. In *Advances in Cryptology – EUROCRYPT 2026*. Springer, Cham, Cham.
[48] Sihang Pu, Jiahui Gao, and Ni Trieu. 2026. Malicious Private Set Union with Two-Sided Output. In *Advances in Cryptology – EUROCRYPT 2026*. Springer, Cham, Cham.
[49] Mike Rosulek and Ni Trieu. 2021. Compact and Malicious Private Set Intersection for Small Sets. In *ACM CCS 2021*, Giovanni Vigna and Elaine Shi (Eds.). ACM Press, 1166–1181. https://doi.org/10.1145/3460120.3484778
[50] Binbin Tu, Yujie Bai, Cong Zhang, Yang Cao, and Yu Chen. 2025. Fast Enhanced Private Set Union in the Balanced and Unbalanced Scenarios. In *USENIX Security 2025*, Lujo Bauer and Giancarlo Pellegrino (Eds.). USENIX Association, 3437–3456. https://www.usenix.org/conference/usenixsecurity25/presentation/tu
[51] Binbin Tu, Yu Chen, Qi Liu, and Cong Zhang. 2023. Fast Unbalanced Private Set Union from Fully Homomorphic Encryption. In *ACM CCS 2023*, Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda (Eds.). ACM Press, 2959–2973. https://doi.org/10.1145/3576915.3623064
[52] Cong Zhang, Yu Chen, Weiran Liu, Liqiang Peng, Meng Hao, Anyu Wang, and Xiaoyun Wang. 2024. Unbalanced Private Set Union with Reduced Computation and Communication. In *ACM CCS 2024*, Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie (Eds.). ACM Press, 1434–1447. https://doi.org/10.1145/3658644.3690308
[53] Cong Zhang, Yu Chen, Weiran Liu, Min Zhang, and Dongdai Lin. 2023. Linear Private Set Union from Multi-Query Reverse Private Membership Test. In *USENIX Security 2023*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, 337–354. https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-cong
[54] Maximilian Zinkus, Yinzhi Cao, and Matthew D. Green. 2023. McFIL: Model Counting Functionality-Inherent Leakage. In *USENIX Security 2023*, Joseph A. Calandrino and Carmela Troncoso (Eds.). USENIX Association, 7001–7018. https://www.usenix.org/conference/usenixsecurity23/presentation/zinkus

A Preliminaries Supplementary Content

A.1 Pseudocode of Match Procedure

The pseudocode for the matching logic of $\mathcal{F}_{\text{MKPM}}$ and $\mathcal{F}_{\text{L-MKPM}}$ (Figure 1) is given in Figure 13.

match(X, Y)

/ Recall that $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}, Y = \{\mathbf{y}_1, \ldots, \mathbf{y}_n\}$
Initialize empty maps M_X and M_Y.
Initialize empty set UID.
$\mathsf{matched}^X \leftarrow (\perp)^n$; $\mathsf{matched}^Y \leftarrow (\perp)^m$
$n_{max} \leftarrow \max_{\mathbf{x} \in X} |\mathbf{x}|$
/ Match Records
for $j = 1, \ldots, n_{max}$ **do**
  for $k = 1, \ldots, m$ **do**
    if $\mathsf{matched}^Y[k] = \top$ **then continue**
    for $i = 1, \ldots, n$ **do**
      if $\mathsf{matched}^X[i] = \top \vee |\mathbf{x}_i| < j$ **then continue**
      for $\ell = 1, \ldots, |\mathbf{y}_i|$ **do**
        if $\mathbf{x}_i[j] = \mathbf{y}_k[\ell]$ **then**
          $\mathsf{uid} \leftarrow\!\!\$\ \mathcal{G} \setminus \mathsf{UID}$
          $\mathsf{UID} \leftarrow \mathsf{UID} \cup \{\mathsf{uid}\}$
          $\mathsf{M}_X[\mathsf{uid}] \leftarrow \mathbf{x}_i$
          $\mathsf{M}_Y[\mathsf{uid}] \leftarrow \mathbf{y}_k$
          $\mathsf{matched}^X[i] \leftarrow \top$; $\mathsf{matched}^Y[k] \leftarrow \top$
/ Assign uids to unmatched records
for $i = 1, \ldots, n$
  if $\mathsf{matched}^X[i] = \perp$ **then**
    $\mathsf{uid} \leftarrow\!\!\$\ \mathcal{G} \setminus \mathsf{UID}$
    $\mathsf{UID} \leftarrow \mathsf{UID} \cup \{\mathsf{uid}\}$
    $\mathsf{M}_X[\mathsf{uid}] \leftarrow \mathbf{x}_i$
    $\mathsf{M}_Y[\mathsf{uid}] \leftarrow \perp$
for $i = 1, \ldots, m$
  if $\mathsf{matched}^Y[i] = \perp$ **then**
    $\mathsf{uid} \leftarrow\!\!\$\ \mathcal{G} \setminus \mathsf{UID}$
    $\mathsf{UID} \leftarrow \mathsf{UID} \cup \{\mathsf{uid}\}$
    $\mathsf{M}_X[\mathsf{uid}] \leftarrow \perp$
    $\mathsf{M}_Y[\mathsf{uid}] \leftarrow \mathbf{y}_i$
return $\mathsf{UID}, \mathsf{M}_X, \mathsf{M}_Y$

Figure 13: Record matching logic used in MK-PrivateID.

A.2 Correctness Conditions of T-Reconstruction

Injectivity of φ ensures that each $\mathbf{y}^* \in R_Y$ corresponds to exactly one $\mathbf{y} \in Y$. Condition (1) enforces per-record soundness, i.e., every value in a reconstructed record appears in the corresponding original record. Condition (2) captures $\mathbb{V}(T)$-restricted per-record completeness i.e., each reconstructed record contains all values from its original record that also appear in $\mathbb{V}(T)$. Finally, Condition (3) enforces $\mathbb{V}(T)$-restricted global completeness i.e., all records containing a value from $\mathbb{V}(T)$ are at least partially reconstructed.

B Guo et al.'s Attack on PSI-CA

The PSI-CA functionality is particularly interesting, since many protocols implementing PSI-related functionalities also leak the cardinality of the intersection, making attacks on PSI-CA applicable to a broad range of protocols. Guo et al. [28] initiated a line of work on the privacy-implications of PSI-CA by presenting an intersection-recovery attack against it. We will show two adaptations of this attack later in this work, which is why we briefly introduce it here.

The authors of [28] assume an input-malicious party A with a target set X, which may evaluate PSI-CA$(X'; Y) = |X' \cap Y|$ for arbitrarily many X' and a static recovery set Y. The goal of A is to determine $X \cap Y$. They present two search tree-based attacks against PSI-CA: (1) a generic attack assuming no prior knowledge about the recovery set and (2) a feature-aware attack, where the membership in the recovery set is correlated with a set of features that are known to the adversary. The two attacks differ only in their approach to building the search tree. We focus on the former.

The attack is based on two observations regarding the size of set intersections. First, there are two cases, where the intersection cardinality immediately implies the intersection: (a) if $|X \cap Y| = 0$, then $X \cap Y = \emptyset$ and (b) if $|X \cap Y| = |X|$, then $X \cap Y = X$. Naturally, these cases are unlikely for general X and Y, but note that they generalize to subsets of X:

Proposition B.1. *Let X and Y be arbitrary sets and let $X' \subseteq X$ be a non-empty subset of X.*

(a) If $|X' \cap Y| = 0$, then $X' \nsubseteq X \cap Y$
(b) If $|X' \cap Y| = |X'|$, then $X' \subseteq Y$ and, thus, $X' \subseteq X \cap Y$

An adversary wishing to learn which elements of X are contained in Y can leverage this observation by evaluating PSI-CA on subsets of X, where one of the two cases of B.1 applies. The main challenge of the attack lies in finding such subsets. To this end, the adversary can recursively subdivide the target set X, constructing a balanced binary search tree such that the root node contains X and any two siblings form a partition of their parent. The two subsets comprising the partition are randomly chosen and of roughly equal size. See Figure 4 for an illustration with $|X| = 6$. Recall that for any tree node N, we denote its children with N.left and N.right, and the subset of X stored at N with N.set.

Evaluating PSU-CA on every node in this tree – although posing a valid approach – is inefficient. However, we can save half of the evaluations with the following, second assumption. Consider any set $X' \subseteq X$ and a partition X'_1, X'_2 of X', i.e., $X'_1 \cup X'_2 = X'$ and $X'_1 \cap X'_2 = \emptyset$. Then, $|X' \cap Y| = |X'_1 \cap Y| + |X'_2 \cap Y|$. The adversary can therefore infer $|X'_2 \cap Y| = |X' \cap Y| - |X'_1 \cap Y|$ if it knows $|X' \cap Y|$ and $|X'_1 \cap Y|$. Choosing the right way to traverse the tree at least ensures the former.

We now combine the two observations to obtain the full intersection-recovery attack. The attack starts by initializing an empty set S, to which elements of the intersection are added over the course of the attack. Next, $|X \cap Y|$ is determined by evaluating PSI-CA(X, Y). We then traverse the binary tree in a DFS-style manner, starting at the root. Let N be some node along a DFS path. $|N.\text{set} \cap Y|$ is known, either from evaluating PSI-CA at the tree's root or, by induction, from the previous iteration of the path traversal. Let $L := N$.left and $R := N$.right be the left and right children of N respectively. We can determine $|L.\text{set} \cap Y|$ by evaluating PSI-CA$(L.\text{set}, Y)$ and then compute $|R.\text{set} \cap Y| = |N.\text{set} \cap Y| - |L.\text{set} \cap Y|$, thus leveraging the second observation from above.

To decide with which child to continue the path traversal, we compute the heuristic $p_{\text{PSI-CA}}(|Z \cap Y|, |Z|) := |Z \cap Y|/|Z|$ for both child sets $Z \in \{L.\text{set}, R.\text{set}\}$ and continue with the child that has the larger value. This prioritizes subsets with a larger ratio of matched values. As elaborated below, this aids in the early termination of DFS path traversals. The other node is entered into a priority queue that is sorted by the same heuristic for later treatment. Continuing the example mentioned above, Figure 4 contains markers that indicate at which nodes the intersection size is determined by evaluating the PSI-CA functionality and at which nodes it can be inferred from its sibling and parent nodes.

A DFS path traversal can be aborted at some node N^*, if either case of Proposition B.1 applies, i.e., if $|N^*.\text{set} \cap Y| \in \{0, |N^*.\text{set}|\}$, since either none or all of the elements stored at N^* lie in the intersection. If case (1b) applies, the elements of N^*.set are added to S. In the worst case, this happens if N^* only stores a single element, but is likely to occur earlier, as the authors of [28] point out. In the example in Figure 4, early terminations occur at the nodes $X_{2,1}$ and $X_{1,2}$, as indicated by the missing markers in the respective subtrees.

Upon the completion of a DFS path traversal, the next node to be processed is popped from the aforementioned priority queue. As the probability for an early termination is correlated with the heuristic $p_{\text{PSI-CA}}$, with which subsets are inserted into the queue, paths that are more likely to terminate early are processed first. This increases the recovered fraction of the intersection in case the adversary has a limited query budget, i.e., it can only make a limited number of PSI-CA evaluations.

In the worst case, when no early path terminations occur, the attack evaluates PSI-CA once for the tree root and once for half of the nodes of any given depth in the tree. Since a balanced binary tree with $n := |X|$ leaf nodes contains exactly $2n - 2^{\lceil \log_2 n \rceil}$ nodes of depth $\lceil \log_2 n \rceil$, the attack evaluates PSI-CA at most $1 + \sum_{d=1}^{\lceil \log_2 n \rceil - 1} 2^{d-1} + (2n - 2^{\lceil \log_2 n \rceil})/2 = n$ times. While this is not better than querying every element of X separately, the authors of [28] highlight that we can expect the number of functionality evaluations to be smaller, due to the aforementioned early termination of path traversals.

C Additional Content for PSU-CA Attacks

C.1 Proof of Prop. 4.2

Proof. Let $n = |X|, n_1 = |X_1|$ and $n_2 = |X_2|$. By definition, X_1 contains exactly $n - n_1$ fewer elements than X and $X_1 \cap Y$ exactly $z - z_1 \leq n - n_1$ fewer than $X \cap Y$. Exactly $d := (n - n_1) - (z - z_1)$ elements of $X \setminus X_1$ must therefore be contained in Y, i.e., $|(X \setminus X_1) \cap Y| = d$. Since X_1 and X_2 form a partition of X, we have $X \setminus X_1 = X_2$ and $n - n_1 = n_2$ and, thus, $|X_2 \cap Y| = d$. Then,

$$\begin{aligned} z_2 &= |X_2 \cup Y| \\ &= |X_2| + |Y| - |X_2 \cap Y| \\ &= n_2 + |Y| - ((n - n_1) - (z - z_1)) \\ &= n_2 + |Y| - (n_2 - (z - z_1)) \\ &= |Y| + (z - z_1) \end{aligned}$$

Reordering terms concludes the proof. □

C.2 Proof of Prop. 4.3

Proof. We provide a concrete construction of $\mathcal{B}$ for any $\mathcal{A}$ in Fig. 15. Correctness of $\mathcal{B}$ follows directly from Prop. 4.2 and the fact that we have $|X \cap Y| = |X| + |Y| - |X \cup Y|$ for any sets X and Y. $\mathcal{B}$ makes three PSU-CA evaluations on lines 2 - 4 and one for every PSI-CA query made by $\mathcal{A}$. Since determining m and translating union cardinalities to intersection cardinalities only involves simple integer arithmetic, $\mathcal{B}$ has a similar runtime to $\mathcal{A}$. □

C.3 Instantiating the PSI-CA Attack

We now integrate the generic construction from 15 into the PSI-CA-Search Tree attack, which we introduced in Appendix B.

Recall that the attack starts by building a binary tree $\mathcal{T}$ such that the root $\mathcal{T}$.root holds the target set X and the sets stored at the two children of any node in the tree form a partition of the

PSU-CA-SearchTree$^{\text{PSU-CA}(\cdot,Y)}(X)$

1: Initialize empty sets pos, neg.
2: Initialize max-priority queue Q.
3: $T \leftarrow \text{buildTree}(X)$
4: $T_L = T.\text{root.left};\ T_R = T.\text{root.right}$
5: $X_L = T_L.\text{set};\ X_R = T_R.\text{set}$
6: $z \leftarrow \text{PSU-CA}(X, Y)$
7: $z_L \leftarrow \text{PSU-CA}(X_L, Y)$
8: $z_R \leftarrow \text{PSU-CA}(X_R, Y)$
9: $m \leftarrow z_R - (z - z_L)$ / 4.2
10: $k \leftarrow |X| + m - z$
11: $k_L \leftarrow |X_L| + m - z_L$
12: $k_R \leftarrow |X_R| + m - z_R$
13: $Q.\text{push}(p_{\text{PSI-CA}}(z_L, |X_L|), (T_L, k_L))$
14: $Q.\text{push}(p_{\text{PSI-CA}}(z_R, |X_R|), (T_R, k_R))$
15: **while** Q is not empty **do**
16: $(T_c, k_c) \leftarrow Q.\text{pop}()$
$X_c \leftarrow T_c.\text{set}$
17: / DFS path traversal
18: **while** $0 < k_c < |X_c|$ **do**
19: $T_L \leftarrow T_c.\text{left};\ T_R \leftarrow T_c.\text{right}$
20: $X_L \leftarrow T_L.\text{set};\ X_R \leftarrow T_R.\text{set}$
21: $z_L \leftarrow \text{PSU-CA}(X_L, Y)$
22: $k_L \leftarrow |X_L| + m - z_L$
23: $k_R \leftarrow k_c - k_L$
24: $p_L \leftarrow p_{\text{PSI-CA}}(k_L, |X_L|)$
25: $p_R \leftarrow p_{\text{PSI-CA}}(k_R, |X_R|)$
26: **if** $p_L > p_R$ **then**
/ Continue with left child
27: $T_c \leftarrow T_L;\ k_c \leftarrow k_L$
28: $Q.\text{push}(p_R, (T_R, k_R))$
29: **else**
/ Continue with right child
30: $T_c \leftarrow T_R;\ k_c \leftarrow k_R$
31: $Q.\text{push}(p_L, (T_L, k_L))$
32: **if** $k_c = |X_c|$ **then**
33: $\text{pos} \leftarrow \text{pos} \cup X_c$
34: **else**
35: $\text{neg} \leftarrow \text{neg} \cup X_c$
36: **return** pos, neg

buildTree($\mathcal{T}$)

1: Initialize a tree T with a single node root, which stores $\mathcal{T}$. Build a balanced binary search tree by recursively dividing the set stored at a node into two disjoint subsets of roughly equal size.
2: **return** T

Figure 14: Intersection-recovery attack PSU-CA-SearchTree with oracle access to PSU-CA$(\cdot, Y)$ for a fixed recovery set Y. Note that Y is part of the oracle and unknown to the adversary. Changes to PSI-CA-SearchTree are highlighted in gray.

$\mathcal{B}^{\text{PSU-CA}(\cdot,Y)}_{\mathcal{A}}(T)$

1: Partition T into two disjoint subsets T_1, T_2.
2: $z \leftarrow \text{PSU-CA}(T, Y)$
3: $z_1 \leftarrow \text{PSU-CA}(T_1, Y)$
4: $z_2 \leftarrow \text{PSU-CA}(T_2, Y)$
5: $m \leftarrow z_2 - (z - z_1)$ / Prop. 4.2
6: $Z \leftarrow \mathcal{A}^{\text{PSI-CA-Sim}}(T)$
7: **return** Z

PSI-CA-Sim(X)

1: $z \leftarrow \text{PSU-CA}(X, Y)$
2: $k \leftarrow |X| + m - z$
3: **return** k

Figure 15: Generic construction of an intersection recovery attack $\mathcal{B}$ against PSU-CA given an intersection recovery attack $\mathcal{A}$ against PSI-CA. Note that Y is part of the oracle and unknown to either attack.

set stored at their parent. In particular, the immediate children of T.root store the sets X_1 and X_2, which form a partition of X. The attack then evaluates PSI-CA on X and X_1 (the latter happens in the first loop iteration) and computes $|X_2 \cap Y| = |X \cap Y| - |X_1 \cap Y|$. This inference is not possible if the attacker can only evaluate PSU-CA, as it needs to know m. However, using 4.2 the attacker can determine m by evaluating PSU-CAon X, X_1 and X_2 and then infer $|X \cap Y| = |X| + m - |X \cup Y|$ and $|X_i \cap Y| = |X_i| + m - |X_i \cup Y|$ for $i \in \{1, 2\}$. This is almost equivalent to how PSI-CAattack processes the tree root and its two child nodes,

with the difference that we require an additional evaluation of PSU-CAto determine $|X_2 \cap Y|$. After that, the attacker can carry out the rest of the PSI-CA attack, translating union to intersection cardinalities. The attack is shown in 14. Note that in contrast to PSI-CA, we maintain two sets pos and neg, where pos contains all elements of X that belong to the intersection and neg all elements that do not occur in Y. This allows us to evaluate the performance of the attack in more detail under limited query budgets, see Appendix H. We state the correctness in the following theorem:

Theorem C.1. *Let X and Y be two arbitrary sets. PSU-CA-SearchTree$^{\mathcal{F}_{\text{PSU-CA}}(\cdot,Y)}(X)$ outputs $X \cap Y$.*

Proof. The theorem follows directly from the correctness of PSI-CA-SearchTree and Prop. 4.2 and the identity $|X \cap Y| = |X| + |Y| - |X \cup Y|$. □

For our experimental evaluation in Appendix H, we additionally propose two further heuristics: $p^{-}_{\text{PSI-CA}}$, which prioritizes the reconstruction of the set difference $X \setminus Y$ over that of the intersection, and $p^{*}_{\text{PSI-CA}}$, whose aim is to maximize the total (positive and negative) membership information we can infer. In light of these additional definitions, we define the alias $p^{+}_{\text{PSI-CA}} := p_{\text{PSI-CA}}$ to avoid confusion in the presentation of our experimental results.

$$p^{-}_{\text{PSI-CA}}(k, |X|) := \frac{|X| - k}{|X|}$$

$$p^{*}_{\text{PSI-CA}}(k, |X|) := \frac{\max(k, |X| - k)}{|X|}$$

Time and Query Complexity. Suppose that $n = |X|$ is a power of two (the analysis extends to general n; see the argument in the analysis of PSI-CA-SearchTree in Appendix B). There are $2n - 1$ nodes in $\mathcal{T}$. The two loops in Figure 14 collectively visit each of these nodes at most once. The body of the inner loop requires constant time except for the push operation, which requires amortized $O(\log n)$ time. This yields an overall time complexity of $O(n \log n)$.

PSU-CA-SearchTree evaluates PSU-CA three times in lines 6 - 8. Ignoring early terminations, the attack then makes further evaluations for at most half of the nodes in the rest of the binary search tree (excluding the root and its two direct children). As mentioned above, there are $2n - 1$ nodes in T. The attack therefore evaluates PSU-CA at most $(2n - 4)/2 + 3 = n + 1$ times.

As is the case of PSI-CA-SearchTree, we can expect this number to be lower due to the early termination of DFS path traversals. See Section 8.3.1 for an empirical evaluation.

D Applying the PSU-CA Attack to PrivateID

Meta's PrivateID protocol [6] implements the *Single-Key Private Matching (SKPM)* functionality which we denote as $\mathcal{F}_{\mathsf{SKPM}}$.

The functionality $\mathcal{F}_{\mathsf{SKPM}}$ involves two parties, P_1 and P_2, each with input sets $X \subseteq \mathbb{V}$ and $Y \subseteq \mathbb{V}$, respectively. The functionality computes $Z := X \cup Y$ and assigns new identifiers to the elements of Z by evaluating them under an injective random function, producing the set UID. It then computes two maps $\mathsf{M}_X : \mathsf{UID} \to X \cup \{\perp\}$ and $\mathsf{M}_Y : \mathsf{UID} \to Y \cup \{\perp\}$, which locally map elements in UID to the original elements in X and Y (or $\perp$ if the element only occurs in the other party's set). In the end, party P_1 receives the set of UIDs, UID, and dictionary M_X; party P_2 receives UID and M_Y.

See Figure 16 for the formal definition.

Parameters: Two parties, P_1 and P_2.
Functionality: $\mathcal{F}_{\mathsf{SKPM}}(X; Y)$:

1 Receive the input from party P_1: $X = \{x_i \mid i \in [n]\}$
2 Receive the input from party P_2: $Y = \{y_i \mid i \in [m]\}$
3 Compute $Z := X \cup Y$.
4 Sample $f \leftarrow\!\$$ InjectiveFuncs$[\mathbb{V}, \mathbb{U}]$.
5 Compute dictionaries $\mathsf{M}_X : \mathbb{U} \to X \cup \{\perp\}$ and $\mathsf{M}_Y : \mathbb{U} \to Y \cup \{\perp\}$ s.t. for all $z \in Z$:

$$\mathsf{M}_X[f(z)] = \begin{cases} z & \text{if } z \in X \\ \perp & \text{otw.} \end{cases}$$

$$\mathsf{M}_Y[f(z)] = \begin{cases} z & \text{if } z \in Y \\ \perp & \text{otw.} \end{cases}$$

6 Compute $\mathsf{UID} \leftarrow \{f(z) \mid z \in Z\}$.
7 Output:
- Send UID, M_X to P_1.
- Send UID, M_Y to P_2.

Figure 16: The Single-Key Private Matching functionality $\mathcal{F}_{\mathsf{SKPM}}$, implemented by Meta's PrivateID protocol [6].

Since each element $z \in Z = X \cup Y$ is mapped to its unique identifier under an injective function f, the relationship $|\mathsf{UID}| = |X \cup Y|$ holds for any sets X and Y. Consequently, we can directly apply PSU-CA-SearchTree to $\mathcal{F}_{\mathsf{SKPM}}$ to recover $X \cap Y$ in at most $|X| + 1$ protocol invocations. Guo et al. [28] examined PrivateID in the context of their intersection recovery attack on PSI-CA, where they recover $X \cap Y$ in at most $|X|$ protocol invocations. However, their approach relies on the fact that PrivateID leaks $|X \cap Y|$ during protocol execution, whereas our attack depends solely on the functionality output.

Since the PSU-CA-SearchTree attack applies directly, we omit the pseudocode.

E Additional Content for MRR Attack

E.1 Proof of Theorem 5.2

In order to prove correctness, we must first prove the following two lemmas.:

Lemma E.1. *Let $X, Y \subseteq \mathbb{V}^{\leq\lambda}$ be two sets of records. If X is Y-isolated, the cardinality of* UID *output by $\mathcal{F}_{\mathsf{MKPM}}(X; Y)$ (Figure 1) is consistent across multiple evaluations.*

Proof. For any $\mathbf{x} \in X$, let $Y_{\mathbf{x}} \subseteq Y$ denote the set of records in Y that share at least one identifier with $\mathbf{x}$, i.e.,

$$Y_{\mathbf{x}} := \{\mathbf{y} \in Y \mid \exists i, j \in \mathbb{N} \text{ s.t. } \mathbf{x}[i] = \mathbf{y}[j]\}.$$

If X is Y-isolated, then by definition, we have that $Y_{\mathbf{x}} \cap Y_{\mathbf{x}'} = \emptyset$ for any $\mathbf{x}, \mathbf{x}' \in X$ such that $\mathbf{x} \neq \mathbf{x}'$. Therefore, there exist no two records in X that could be matched with the same record in Y.

We first consider sets $Y_{\mathbf{x}}$ that are not empty. Since $\mathbf{x}$ can only be matched with some $y \in Y_{\mathbf{x}}$ and all $y \in Y_{\mathbf{x}}$ can only be matched with $\mathbf{x}$, match (Figure 13) will match $\mathbf{x}$ with exactly one $\mathbf{y}_{\mathbf{x}} \in Y_{\mathbf{x}}$, which are assigned the same UID (line 11). All other $\mathbf{y} \in Y_{\mathbf{x}} \setminus \{\mathbf{y}_{\mathbf{x}}\}$ will remain unmatched and are assigned a unique UID (line 24). Furthermore, all $\mathbf{y} \in Y$ which do not belong to any $Y_{\mathbf{x}'}$ for some $\mathbf{x}' \in X$ will also remain unmatched and are assigned a unique UID (also line 24). Lastly, all $\mathbf{x} \in X$ for which $Y_{\mathbf{x}} = \emptyset$ will remain unmatched and receive a distinct UID (line 18).

Thus, UID contains one UID for each $\mathbf{y} \in Y$ and one UID for each unmatched $\mathbf{x} \in X$, i.e., $|\mathsf{UID}| = |Y| + |\{\mathbf{x} \in X \mid Y_{\mathbf{x}} = \emptyset\}|$. This is independent of the order of the records in X and Y and also independent of the shuffling done before the matching step in MKPM. Since MKPM only uses randomness to shuffle the inputs, the lemma follows. □

Lemma E.2. *Let X and $V \subseteq \mathbb{V}^{\leq\lambda}$ be two sets of records and let X_1 and X_2 be a partition of X. Moreover, let $(\mathsf{UID}, \mathsf{M}_X) \leftarrow\!\$ \mathcal{F}_{\mathsf{MKPM}}(X; Y)$ and $(\mathsf{UID}_i, M_{X,i}) \leftarrow\!\$ \mathcal{F}_{\mathsf{MKPM}}(X_i; Y)$ for $i \in \{1, 2\}$. If X is V-isolated, we have $|\mathsf{UID}| = |\mathsf{UID}_1| + |\mathsf{UID}_2| - |V|$.*

Proof. We show this by induction on $|X_1|$. If $|X_1| = 0$ all records of Y are unmatched and, thus, $|\mathsf{UID}_1| = |Y|$. Note that $X_2 = X$ and thus we have $|\mathsf{UID}| = |\mathsf{UID}_2| = |\mathsf{UID}_2| + |\mathsf{UID}_1| - |Y|$. In the first equality, we use Lemma E.1.

Assume $|X_1| = n_1$ for some $n_1 > 0$ and $|\mathsf{UID}| = |\mathsf{UID}_1| + |\mathsf{UID}_2| - |Y|$. Since X_1 and X_2 form a partition of X, we must move a record from X_2 to X_1 to achieve $|X_1| = n_1 + 1$. Note that $|\mathsf{UID}|$ remains unchanged due to Lemma E.1. Let $\mathbf{x} \in X_2$ and let $X_1' = X_1 \cup \{\mathbf{x}\}$ and $X_2' := X_2 \setminus \{\mathbf{x}\}$. Moreover, let UID_1' and UID_2' be the sets of UIDs resulting from evaluating $\mathcal{F}_{\mathsf{MKPM}}(X_1', Y)$ and $\mathcal{F}_{\mathsf{MKPM}}(X_2', Y)$.

We distinguish two cases.

Case 1: If no identifier of $\mathbf{x}$ occurs in Y, $\mathbf{x}$ was assigned its own UID. Therefore, $|\mathsf{UID}'_1| = |\mathsf{UID}_1| + 1$ and $|\mathsf{UID}'_2| = |\mathsf{UID}_2| - 1$, which implies the claim.

Case 2: For the second case, assume $\mathbf{x}$ shares some identifiers with n^* records of Y. Call this set $Y_{\mathbf{x}}$. Since X is Y-isolated, all records $\mathbf{y} \in Y_{\mathbf{x}}$ only share identifiers with $\mathbf{x}$, but no other $\mathbf{x}' \in X_2$. Thus, $\mathbf{x}$ is matched with some $\mathbf{y}^* \in Y_{\mathbf{x}}$, i.e., $\mathbf{x}$ and $\mathbf{y}^*$ are assigned the same $\mathsf{uid} \in \mathsf{UID}_2$. After removing $\mathbf{x}$ from X_2, $\mathbf{y}^*$ will still be assigned some $\mathsf{uid}' \in \mathsf{UID}'_2$, which it does not share with any $\mathbf{x}' \in X'_2$, again since X is Y-isolated. Therefore, $|\mathsf{UID}_2| = |\mathsf{UID}'_2|$. Similarly, since $\mathbf{x} \notin X_1$, no $\mathbf{y} \in Y_{\mathbf{x}}$ and $\mathbf{x}' \in X_1$ are assigned the same $\mathsf{uid} \in \mathsf{UID}_1$. By the definition of $Y_{\mathbf{x}}$ and since X is Y-isolated, $\mathbf{x}$ will be assigned the same $\mathsf{uid} \in \mathsf{UID}'_1$ as some $\mathbf{y} \in Y_{\mathbf{x}}$ after being added to X_1, implying $|\mathsf{UID}_1| = |\mathsf{UID}'_1|$.

The claim then follows. □

We can now use the above lemmas to finally prove Theorem 5.2.

Proof. Let the output from PSU-CA-SearchTree$^{\mathcal{F}_{\mathsf{MKPM}}(\cdot,V)}(T)$ be the two sets pos and neg. Note all records of T are contained in some subset $T_c \subseteq T$ that will eventually reach line 32, since the attack only terminates once the priority queue is empty. Therefore, every record of T is added to either pos or neg. For any T_c reaching line 32 we have that either (1) $k_c = |T_c|$ or (2) $k_c = 0$ by the loop condition on line 18. Let UID_c result from evaluating $\mathcal{F}_{\mathsf{MKPM}}(T_c, V)$.

By Lemma E.2 we have $k_c = |T_c| + |V| - |\mathsf{UID}_c|$.

Case 1: If $k_c = |T_c|$, then we have $|\mathsf{UID}_c| = |V|$, i.e., all records of T_c are matched. Since records are only added to pos if this case applies, we have $\mathsf{pos} \subseteq T \sqcap V$.

Case 2: If $k_c = 0$, then $|\mathsf{UID}_c| = |T_c| + |V|$, i.e., no records of T_c were matched. Therefore, no record in T_c shares any identifiers with any record in V. Since records are only added to neg if this case applies, we have $\mathsf{neg} \subseteq T \setminus (T \sqcap V)$.

We have therefore shown that pos only contains matchable records, neg only contains non-matchable records and that every record in T is contained in either pos or neg. □

F Additional Content for VIR Attack

F.1 Proof of Prop. 6.1

Proof. Since the value hiding function f is injective, we have that for any value $v \in \mathbb{V}(X)$ and $v' \in \mathbb{V}(Y)$, $v = v'$ if and only if $f(v) = f(v')$. In other words, two values in X and Y are equal if and only if their corresponding hidden values in $\mathcal{L}^1_X$ and $\mathcal{L}^1_Y$ match. Since the order of values in records of X remains unchanged, matched values remain in the same column. The numbers of matched values in the columns of X and $\mathcal{L}^1_X$ are therefore equal. □

F.2 Proof of Prop. 6.2

Proof. Let $\mathbf{c} = \mathrm{ColCA}(\mathcal{L}^1_X, \mathcal{L}^1_Y)$ and fix some $i \in [0, \ell_{\mathrm{rec}} - 1]$. We write $\mathbf{c}[i]$ for the i-th entry of this vector. By Proposition 6.1, we also have $\mathbf{c} = \mathrm{ColCA}(X, Y)$.

Let $\mathbf{c}_1 = \mathrm{ColCA}(\mathcal{L}^1_{X_1}, \mathcal{L}^1_Y)$ and $\mathbf{c}_2 = \mathrm{ColCA}(\mathcal{L}^1_{X_2}, \mathcal{L}^1_Y)$. Again by Proposition 6.1, these equal $\mathrm{ColCA}(X_1, Y)$ and $\mathrm{ColCA}(X_2, Y)$, respectively. Set $\mathbf{c}[i] = x$ and $\mathbf{c}_1[i] = x_1$. By definition of ColCA, exactly x_1 records in X_1 have their i-th value appearing in Y. Since X_1 and X_2 form a partition of X, the remaining $x_2 = x - x_1$ records must lie in X_2, so $\mathbf{c}_2[i] = x_2$.

Thus, for all $i \in [0, \ell_{\mathrm{rec}} - 1]$,

$$\mathbf{c}[i] = x = x_1 + x_2 = \mathbf{c}_1[i] + \mathbf{c}_2[i],$$

which proves the proposition. □

F.3 Proof of Theorem 6.3

Proof. The set pos is updated only by adding values from some column $i \in [0, \ell_{\mathrm{rec}} - 1]$ of a node set $X \subseteq T$ if

$$\mathrm{ColCA}(\mathcal{L}^1_X, \mathcal{L}^1_Y)[i] = |X|.$$

By Prop. 6.1, this is equivalent to $\mathrm{ColCA}(X, Y)[i] = |X|$. By the definition of ColCA, this means that every i-th value in X occurs in Y, so all values added to pos in this case are in $\mathbb{V}(Y)$. Hence $\mathsf{pos} \subseteq \mathbb{V}(T) \cap \mathbb{V}(Y)$, i.e., pos contains no false positives. An analogous argument for updates to neg shows that $\mathsf{neg} \subseteq \mathbb{V}(T) \setminus \mathbb{V}(Y)$.

Next, observe that after each partitioning step (line 9), both child subsets are either processed immediately or inserted into the priority queue. Since the attack terminates only when the queue is empty, every record in T appears in some node that reaches the membership-inference phase (line 20). Thus, every value in $\mathbb{V}(T)$ is eventually added to either pos or neg.

Combining these facts, we obtain

$$\mathsf{pos} = \mathbb{V}(T) \cap \mathbb{V}(Y) \quad \text{and} \quad \mathsf{neg} = \mathbb{V}(T) \setminus \mathbb{V}(Y).$$

Therefore, pos equals the value intersection $\mathbb{V}(T) \cap \mathbb{V}(Y)$, and value intersection recovery (Definition 3.3) is achieved. □

F.4 Additional Heurisitics for MKPM-SearchTree

We propose two additional heuristics for determining which path in the search tree to prioritize exploring: p^-_{MK}, which prioritizes the reconstruction of the set difference $\mathbb{V}(T) \setminus \mathbb{V}(Y)$ and p^*_{MK}, whose aim is to maximize the total (positive and negative) membership information we can infer.

$$p^-_{MK}(\mathbf{c}, X) := \frac{\sum_{x \in \mathbf{c}} |X| - x}{\ell_{\mathrm{rec}} |X|} \tag{3}$$

$$p^*_{MK}(\mathbf{c}, X) := \frac{\sum_{x \in \mathbf{c}} \max(x, |X| - x)}{\ell_{\mathrm{rec}} |X|} \tag{4}$$

F.5 Pseudocode for MKPM-SearchTree

The pseudocode of MKPM-SearchTree is given in Figure 17.

G Proofs for Analysis of $\mathcal{F}_{\mathsf{L}\text{-}\mathsf{MKPM}}$

G.1 Proof of Correctness of Substitute

Lemma G.1. *Let X and Y be two sets of records and let $(\mathcal{L}^1_X, \mathcal{L}^1_Y)$ be the hidden shuffled records received by $\tilde{P}_1$. Moreover, let $f_1 : \mathbb{V} \to \mathbb{U}$ be the hiding function sampled in $\mathcal{F}_{\mathsf{L}\text{-}\mathsf{MKPM}}$ to produce $\mathcal{L}^1_X$ and $\mathcal{L}^1_Y$. If D is a dictionary mapping $f_1(v)$ to v for all $v \in \mathbb{V}(X)$, then Substitute$(D, \mathcal{L}^1_Y)$ (Figure 20) produces an X-reconstruction of Y.*

Proof. Let R be the multiset produced by Substitute. Note that for each $\hat{\mathbf{y}} \in \mathcal{L}^1_Y$ exactly one reconstructed record $\mathbf{r}$ is added to R. Let $d : R \to \mathcal{L}^1_Y$ denote the function mapping these reconstructed records to the corresponding $\hat{\mathbf{y}}$. Clearly, d is a bijection.

MKPM-SearchTree$^{\mathcal{F}_{\text{L-MKPM}}(\cdot,Y)}(T)$

1 : Initialize empty sets pos, neg
2 : Initialize empty max-priority queue Q
3 : $\mathcal{T} \leftarrow$ BuildTree(T)
4 : $(\text{UID}, M_C, (\mathcal{L}_T^1, \mathcal{L}_Y^1)) \leftarrow\$ \mathcal{F}_{\text{L-MKPM}}(T, Y)$
5 : Q.push$(1, (\text{ColCA}(\mathcal{L}_T^1, \mathcal{L}_Y^1), \mathcal{T}.\text{root}))$
6 : **while** Q is not empty **do**
7 : $(\mathbf{c}, v) \leftarrow Q$.pop()
// DFS-style path traversal
8 : **while** $\exists i$ s.t. $0 < \mathbf{c}_v[i] < |v.\text{set}|$ **do**
9 : $v_L \leftarrow v$.left; $v_R \leftarrow v$.right
10 : $(\text{UID}, M, (\mathcal{L}_{X_L}^1, \mathcal{L}_Y^1)) \leftarrow\$ \mathcal{F}_{\text{L-MKPM}}(v_L.\text{set}, Y)$
11 : $\mathbf{c}_L \leftarrow \text{ColCA}(\mathcal{L}_{X_L}^1, \mathcal{L}_Y^1)$
12 : $\mathbf{c}_R \leftarrow \mathbf{c} - \mathbf{c}_L$
13 : $p_L \leftarrow p_{MK}(\mathbf{c}_L, v_L.\text{set})$; $p_R \leftarrow p_{MK}(\mathbf{c}_R, v_R.\text{set})$
14 : **if** $p_R > p_L$ **then**
// Proceed with right child
15 : Q.push$(p_L, (\mathbf{c}_L, v_L))$
16 : $v \leftarrow v_R$; $\mathbf{c} \leftarrow \mathbf{c}_R$
17 : **else**
// Proceed with left child
18 : Q.push$(p_R, (\mathbf{c}_R, v_R))$
19 : $v \leftarrow v_L$; $\mathbf{c} \leftarrow \mathbf{c}_L$
20 : **for** $i = 1, \ldots, \ell_{\text{rec}}$ **do**
21 : **if** $\mathbf{c}[i] > 0$ **then**
22 : pos $\leftarrow$ pos $\cup$ $\{ \mathbf{x}[i] \mid \mathbf{x} \in v.\text{set} \}$
23 : **else**
24 : neg $\leftarrow$ neg $\cup$ $\{ \mathbf{x}[i] \mid \mathbf{x} \in v.\text{set} \}$
25 : **return** (pos, neg)

BuildTree(T)

1 : Initialize a tree $\mathcal{T}$ with a single node root, which stores T. Recursively partition the set at each node into two disjoint subsets of roughly equal size to obtain a balanced binary search tree.
2 : **return** $\mathcal{T}$

Figure 17: MKPM-SearchTree with oracle access to the extended MKPM functionality $\mathcal{F}_{\text{L-MKPM}}(\cdot, Y)$.

We now define an injective function $\varphi : R \to Y$ and prove that it satisfies the three conditions for X-Reconstruction (Definition 3.4). For any record $\mathbf{r} \in R$ let $(d_1, \ldots, d_k) = d(\mathbf{r}) \in \mathcal{L}_Y^1$ and let

$$\varphi(\mathbf{r}) := (f_1^{-1}(d_1), \ldots, f_1^{-1}(d_k))$$

Note that $f_1^{-1}(d_j)$ exists for all $\mathbf{r}$ and j, since $d(\mathbf{r}) \in \mathcal{L}_Y^1$ by definition. Moreover, since f_1 is injective, f_1^{-1} is injective for elements in $f_1(\mathbb{V})$. φ is injective since d and f_1^{-1} are injective. We show the three properties from Definition 3.4 separately:

1 Let $\mathbf{r} \in R$, $k := |\mathbf{r}|$ and $i \le |\mathbf{r}|$. Moreover, let $(d_1, \ldots, d_{k'}) = d(\mathbf{r})$ for some $k' \ge k$. By definition of d and by construction of $\mathbf{r}$ (line 6), there exists some $j^* \in [k']$ such that $r[i] = D[d_{j^*}]$. Suppose for a contradiction that $\mathbf{r}[i] \notin \varphi(\mathbf{r})$, that is, we have $\mathbf{r}[i] \neq f_1^{-1}(d_j)$ for all $j \in [k']$. But since for all $j \in [k']$ with $d_j \in D$ we have $f_1^{-1}(d_j) = D[d_j]$, we also have that $\mathbf{r}[i] \neq D[d_j]$, i.e., there is no j such that $\mathbf{r}[i] = D[d_j]$. This is a direct contradiction to $\mathbf{r}[i] = D[d_{j^*}]$.
2 Let $\mathbf{r} \in R$ and $(d_1, \ldots, d_{k'}) = d(\mathbf{r})$ for some $k' \in \mathbb{N}$. Let $v \in \varphi(\mathbf{r})$, i.e., there is some $i \in [k']$ such that $v = f_1^{-1}(d_i)$. If $v \in \mathbb{V}(X)$, then by assumption we have $f_1(v) \in D$ and $D[f_1(v)] = v$. Since $v = f_1^{-1}(d_i)$, we have $f_1(v) = f_1(f_1^{-1}(d_i)) = d_i$ and thus $d_i \in D$. Therefore, the condition on line 5 is met and $D[d_i] = D[f_1(v)] = v$ is added to $\mathbf{r}$.
3 Holds trivially, since we have $\varphi(R) = Y$ by the definition of d.

The lemma thus follows. □

G.2 Pseudocode of EnumAttack

The pseudocode of EnumAttack can be found in Figure 20.

G.3 Proof of Correctness of EnumAttack

Proof. In order to prove the theorem we prove the following proposition. The theorem follows directly from Proposition G.2 and Lemma G.1.

Proposition G.2. *Let T and Y be two sets of records and let D be the dictionary constructed during an execution of EnumAttack$^{\mathcal{F}_{\text{L-MKPM}}(\cdot,Y)}(T)$ (Figure 20). Moreover, let f_1 be the hiding function sampled during the evaluation of $\mathcal{F}_{\text{L-MKPM}}$ in line 8 in Figure 20. We have $D[f_1(v)] = v$ for all $v \in \mathbb{V}(T)$.*

Proof. **of Proposition G.2** Let $X = \{\mathbf{x}_0, \ldots, \mathbf{x}_{n-1}\}$, where $\mathbf{x}_i$ denotes the altered target record constructed in line 6 for $i \in \{0, \ldots, n-1\}$. We show that $D[f_1(v)] = v$ for all $v \in \mathbb{V}(X)$. Since $\mathbb{V}(T) \subseteq \mathbb{V}(X)$, this implies the claim.

We first show that $u_0 = f_1(v_0)$ for the value u_0 defined on line 10. Note that the transformation of an index's binary representation to a sequence of v_0 and v_1 in lines 23 - 27 of EncodeIndex could produce two encodings that only contain one value, namely, $(v_0)^\ell$ and $(v_1)^\ell$ for $i = 0$ and $i = 2^\ell - 1$ respectively. However, the latter is prevented by the condition on line 29 and therefore EncodeIndex only outputs one sequence where all values are equal. Hence, there is only one record in X whose first ℓ values all coincide, namely, $\mathbf{x}_0$ with encoding $(v_0)^\ell$. Since f_1 is injective and the values in records of X are not shuffled, the latter also holds for $\mathcal{L}_X^1$. Therefore, $\hat{\mathbf{x}}_0 = (f_1(\mathbf{x}_0[1]), \ldots, f_1(\mathbf{x}_0[\lambda + \ell]))$ (line 9). Since $\mathbf{x}_0[1] = v_0$, we have $u_0 = \hat{\mathbf{x}}_0[1] = f_1(\mathbf{x}_0[1]) = f_1(v_0)$ (line 10).

We now show that the attack correctly recovers indices from $\mathcal{L}_X^1$. We have already established that $\hat{\mathbf{x}}_0$ is correctly matched with $\mathbf{x}_0$, as it's encoding is the only one containing only v_0. Any other encoding $\mathbf{s}$ produced by EncodeIndex either (a) is $(v_1)^{\ell-1} \| v_2$ or (b) only contains the values v_0 and v_1.

Case (a): This case is only produced on input $i = 2^\ell - 1$ and the resulting vector of hidden values in the leakage is $(f_1(v_1))^{\ell-1} \| f_1(v_2)$. This satisfies the condition on line 29 of DecodeIndex, which therefore yields the correct index $2^\ell - 1$.

Case (b): In this case, the vector of hidden values corresponding to $\mathbf{s}$ in the leakage, which we denote $\hat{\mathbf{s}} := (f_1(\mathbf{s}[1]), \ldots, f_1(\mathbf{s}[\ell]))$, does not satisfy the condition on line 29, since the only encoding that would do so is $(v_1)^\ell$. But this is never output by EncodeIndex. Therefore, $\hat{\mathbf{s}}$ is interpreted as a binary encoding, where every occurrence of u_0 is interpreted as 0 and every other hidden value as 1. The correctness of this recovery therefore follows from the fact that $u_0 = f_1(v_0)$, which we established above.

By the definition of $\mathcal{F}_{\text{L-MKPM}}$, we have $\hat{\mathbf{x}} = (f_1(\mathbf{x}_i[1]), \ldots, f_1(\mathbf{x}_i[\lambda+\ell]))$ for all $i \in \{0, \ldots, n-1\}$. Since $\mathbb{V}(X) := \bigcup_{l\mathbf{x}\in X v\in\mathbf{x}} v$, the claim holds by the construction of D in lines 12 - 15. □

The theorem follows directly from Proposition G.2 and Lemma G.1. □

G.4 Pseudocode of SnakeAttack

The pseudocode of EnumAttack can be found in Figure 21.

G.5 Proof of Correctness of SnakeAttack

Proof. To prove the theorem, we prove the following proposition. The theorem follows directly from Proposition G.3 and Lemma G.1.

Proposition G.3. *Let T and Y be two sets of records and let D be the substitution dictionary constructed during an invocation of $\textsc{SnakeAttack}^{\mathcal{F}_{\text{L-MKPM}}(\cdot,Y)}(T)$. Moreover, let f_1 be the hiding function sampled during the invocation of $\mathcal{F}_{\text{L-MKPM}}$ in line 11. Then, we have $D[f_1(v)] = v$ for all $v \in \mathbb{V}(T)$.*

Proof. **of Proposition G.3** Let $X := \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ where $\mathbf{x}_i$ is produced by MakeSnake in line 10 and let $k' := |\mathbf{x}_i|$ for any $i \in [n]$[4]. We show that $D[f_1(v)] = v$ for all $v \in \mathbb{V}(X)$. Since SnakeAttack potentially adds fresh values to records of T, but does not remove any, we have $\mathbb{V}(T) \subseteq \mathbb{V}(X)$, which implies the claim.

Similar to the proof of Proposition G.2, we show that $\hat{\mathbf{x}}_i[j] = f_1(\mathbf{x}_i[j])$ for all $i \in [n]$ and all $j \in [k']$. The result then follows from the construction of D in lines 14 - 16. In particular, we will show that the linked list we create in EncodeIndex is preserved in the leakage, and that this linked list allows us to uniquely identify each record. We show this by induction on i. Let j^* denote the index of a column whose values are all unique (note that if such a column does not exist, it is created on line 3).

For $i = 1$, observe that the value $t_1[j^*]$ is not appended to any record in MakeSnake; in other words, it is the only item in the linked list that doesn't have anything pointing to it. In contrast, for all $i' > 1$, the value $t_{i'}[j^*]$ is appended to $t_{i'+1}$. $\hat{\mathbf{x}}_1$ is therefore the only record in X whose j^*-th value is not contained in the last column, i.e., $\hat{\mathbf{x}}_1[j^*] \notin \{\hat{\mathbf{x}}_{i'}[k'] \mid i' \in [n]\}$. By the definition of $\mathcal{F}_{\text{L-MKPM}}$, this property is preserved in the leakage $\mathcal{L}_X^1$. That is, the corresponding record in $h = (f_1(\hat{\mathbf{x}}_1[1]), \ldots, f_1(\hat{\mathbf{x}}_1[k']))$ is the only record in $\mathcal{L}_X^1$ such that $h[j^*] \notin \{h_{i'}[k'] \mid i' \in [n]\}$. In line 31 in UnknotSnake we pick $h_{\text{ind}} := h$. Thus, for all $j \in [k']$ we have that $h_{\text{ind}}[j] = f_1(\mathbf{x}_1[j])$ and, therefore $\hat{\mathbf{x}}_1[j] = h_{\text{ind}}[j] = f_C(\mathbf{x}_1[j])$, where the first equality follows from line 34.

[4]We assume that all records have the same length λ. Thus, $k' \in \{\lambda+1, \lambda+2\}$

Now assume that for some $i < n$ we have $\hat{\mathbf{x}}_i[j] = f_1(\mathbf{x}_i[j])$ for all $j \in [k']$. We show that there is a unique record which $\mathbf{x}_i[k']$ is pointing to (that has not been traversed by the previous items in the linked list), and that, once again this property is preserved in the leakage allowing for unique identification of the record. By construction in MakeSnake, we have $\mathbf{x}_{i+1}[j^*] = \mathbf{x}_i[k']$. Since all values in the j^*-th column are distinct, $\mathbf{x}_{i+1}$ is the only record in X that satisfies this condition. As before, since f_1 is injective, the corresponding record in the leakage $h = (f_1(\mathbf{x}_{i+1}[1]), \ldots, f_1(\mathbf{x}_{i+1}[k']))$ is the only record that satisfies $h[j^*] = \hat{\mathbf{x}}_i[k']$. Moreover, all hidden values in the j^*-th column of $\mathcal{L}_X^1$ are distinct. Thus, the dictionary Loc in DecodeIndex contains exactly the n keys $\{f_1(\mathbf{x}_{i'}[j^*]) \mid i' \in [n]\} = \{f_1(\hat{\mathbf{x}}_{i'}[j^*]) \mid i' \in [n]\}$, which are mapped as $\text{Loc}[f_1(\mathbf{x}_{i'}[j^*])] = i'$. In line 34, we pick $\hat{\mathbf{x}}_{i+1} := h_{\text{ind}}$ for $\text{ind} = \text{Loc}[\hat{\mathbf{x}}_i[k']]$, implying $\hat{\mathbf{x}}_{i+1}[j^*] = \hat{\mathbf{x}}_i[k']$ and, thus, $\hat{\mathbf{x}}_{i+1} = h$. We therefore have $\hat{\mathbf{x}}_{i+1}[j] = h[j] = f_1(\mathbf{x}_{i+1}[j])$ for all $j \in [k']$. □

The statement follows from Proposition G.3 and Lemma G.1. □

H Additional Evaluation Results

H.1 SearchTree Attack Queries Experiments

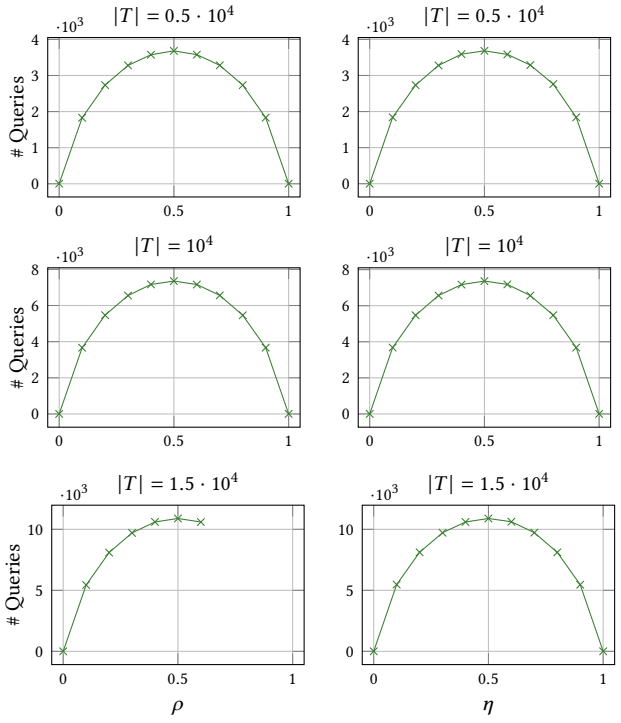
In Figure 18 we plot the number of protocol invocations required by the (a) PSU-CA-SearchTree and (b) MKPM-SearchTree attacks, plotted against the intersection ratio ρ and match rate η for a fixed $|Y| = 10^4$.

H.2 Performance under Limited Query Budgets

We also evaluated the performance of our search tree-based attacks under constrained query budgets. We allocated query budgets in the range of $10 - 100\%$ percent of the theoretical maximum in 10% increments. Once an attack reached a given query budget, path traversal was terminated and the priority queue was cleared to process remaining subsets without further queries.

Our goal was to measure how much of the original recovery goal the attacks managed to achieve. Recall that with enough queries, both PSU-CA-SearchTree and MKPM-SearchTree output two sets $\text{pos} = A \cap B$ and $\text{neg} = A \setminus B$, where $A = X$, $B = Y$ for the PSU-CA-SearchTree attack and $A = \mathbb{V}(T)$, $B = \mathbb{V}(Y)$ for the MKPM-SearchTree attack. Limiting the number of queries, we now have $\text{pos} \subseteq A \cap B$ and $\text{neg} \subseteq A \setminus B$. We discuss three metrics: (1) the recovered fraction of the intersection $|\text{pos}|/|A \cap B|$, (2) the recovered fraction of the set difference $|\text{neg}|/|A \setminus B|$, and (3) the fraction of elements whose membership status was decided $(|\text{pos}| + |\text{neg}|)/|A|$. Figure 19 shows these tree metrics plotted against the allocated query budget for the PSU-CA-SearchTree attack. Measurements for MKPM-SearchTree yielded similar results.

Figure 19 shows that PSU-CA-SearchTree can partially recover the intersection with a limited number of queries. The recovered fraction depends not only on the query budget but also on the intersection ratio ρ, as expected, given its relationship to the number of queries. Our experiments indicate that, depending on ρ, a query budget of 20 % to 40 % of the theoretical upper bound $|X| + 1$ suffices to determine the membership status of at least half of the elements in X on average. These results show that PSU-CA-SearchTree

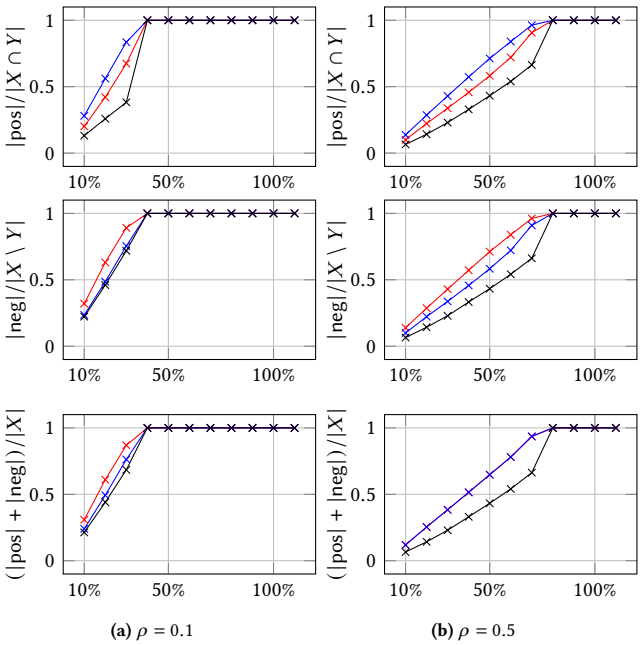


(a) PSU-CA-SearchTree (b) MKPM-SearchTree

Figure 18: Number of protocol queries required by the (a) PSU-CA-SearchTree and (b) MKPM-SearchTree attacks, plotted against the match rate ρ and η, respectively, for a fixed $|Y| = 10^4$.

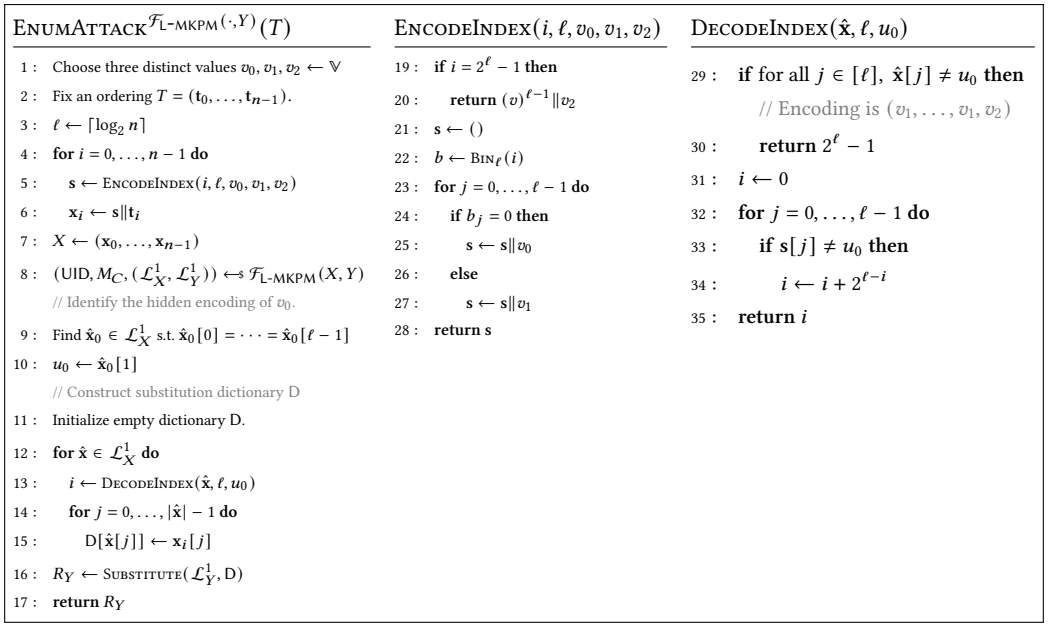
performs effectively under constrained budgets, making the worst-case need for $|X| + 1$ queries unlikely. Similar observations apply to MKPM-SearchTree.

In contrast to Section 8.3.1 (where we ran the attacks to completion), the impact of the chosen heuristic for partial recovery is evident. Interestingly, the heuristic $p^{+}_{\text{PSI-CA}}$, intended to maximize the recovery of $X \cap Y$, appears more efficient at increasing the recovery of $X \setminus Y$. Conversely, $p^{-}_{\text{PSI-CA}}$ is more effective at optimizing the recovered intersection than the set difference. Morevoer, $p^{*}_{\text{PSI-CA}}$ underperforms across all metrics, including total membership inference, which it was meant to optimize.



(a) $\rho = 0.1$ (b) $\rho = 0.5$

Figure 19: Recovery rate of PSU-CA-SearchTree under limited query budgets. (a) and (b) show recovered fractions of $T \cap Y$, $T \setminus Y$, and fraction of T elements whose membership status was determined, using heuristics $p^{+}_{\text{PSI-CA}}$, $p^{-}_{\text{PSI-CA}}$, and $p^{*}_{\text{PSI-CA}}$, against allocated query budget as a percentage of theoretical upper bound $|T| + 1 = 10^4 + 1$.



EnumAttack $^{\mathcal{F}_{\text{L-MKPM}}(\cdot,Y)}(T)$

Choose three distinct values $v_0, v_1, v_2 \leftarrow \mathbb{V}$
Fix an ordering $T = (\mathbf{t}_0, \ldots, \mathbf{t}_{n-1})$.
$\ell \leftarrow \lceil \log_2 n \rceil$
for $i = 0, \ldots, n-1$ **do**
  $\mathbf{s} \leftarrow$ EncodeIndex(i, ℓ, v_0, v_1, v_2)
  $\mathbf{x}_i \leftarrow \mathbf{s} \| \mathbf{t}_i$
$X \leftarrow (\mathbf{x}_0, \ldots, \mathbf{x}_{n-1})$
$(\mathsf{UID}, M_C, (\mathcal{L}_X^1, \mathcal{L}_Y^1)) \leftarrow\$ \mathcal{F}_{\text{L-MKPM}}(X, Y)$
// Identify the hidden encoding of v_0.
Find $\hat{\mathbf{x}}_0 \in \mathcal{L}_X^1$ s.t. $\hat{\mathbf{x}}_0[0] = \cdots = \hat{\mathbf{x}}_0[\ell-1]$
$u_0 \leftarrow \hat{\mathbf{x}}_0[1]$
// Construct substitution dictionary D
Initialize empty dictionary D.
for $\hat{\mathbf{x}} \in \mathcal{L}_X^1$ **do**
  $i \leftarrow$ DecodeIndex$(\hat{\mathbf{x}}, \ell, u_0)$
  for $j = 0, \ldots, |\hat{\mathbf{x}}| - 1$ **do**
    $\mathrm{D}[\hat{\mathbf{x}}[j]] \leftarrow \mathbf{x}_i[j]$
$R_Y \leftarrow$ Substitute$(\mathcal{L}_Y^1, \mathrm{D})$
return R_Y

EncodeIndex(i, ℓ, v_0, v_1, v_2)

if $i = 2^\ell - 1$ **then**
  return $(v)^{\ell-1} \| v_2$
$\mathbf{s} \leftarrow ()$
$b \leftarrow \text{Bin}_\ell(i)$
for $j = 0, \ldots, \ell - 1$ **do**
  if $b_j = 0$ **then**
    $\mathbf{s} \leftarrow \mathbf{s} \| v_0$
  else
    $\mathbf{s} \leftarrow \mathbf{s} \| v_1$
return $\mathbf{s}$

DecodeIndex$(\hat{\mathbf{x}}, \ell, u_0)$

if for all $j \in [\ell]$, $\hat{\mathbf{x}}[j] \neq u_0$ **then**
  // Encoding is $(v_1, \ldots, v_1, v_2)$
  return $2^\ell - 1$
$i \leftarrow 0$
for $j = 0, \ldots, \ell - 1$ **do**
  if $\mathbf{s}[j] \neq u_0$ **then**
    $i \leftarrow i + 2^{\ell-i}$
return i

Figure 20: Record enumeration attack EnumAttack with oracle access to the extended MK-PrivateID functionality $\mathcal{F}_{LMKPM}(\cdot; Y)$. Note that Y is part of the oracle and not known to the adversary.

SnakeAttack $^{\mathcal{F}_{\text{L-MKPM}}(\cdot,Y)}(T)$

$n \leftarrow |T|$
Fix an ordering $T = (\mathbf{t}_0, \ldots, \mathbf{t}_{n-1})$.
if $\nexists j^*$ s.t. $|\{\mathbf{t}_i[j^*] \mid \mathbf{t}_i \in T\}| = n$ **then**
  Pick n distinct $v_0, \ldots v_{n-1} \leftarrow \mathbb{V}$
  foreach $i \in [0, n-1]$ **do**
    $\mathbf{t}_i \leftarrow v_i \| \mathbf{t}_i$
    $j^* \leftarrow 0$
else
  Pick $j^* \leq n-1$ s.t. $|\{\mathbf{t}_i[j^*] \mid \mathbf{t}_i \in T\}| = n$
$X \leftarrow$ MakeSnake$(j^*, (\mathbf{t}_0, \ldots, \mathbf{t}_{n-1}))$
$(\mathsf{UID}, M_C, (\mathcal{L}_X^1, \mathcal{L}_Y^1)) \leftarrow\$ \mathcal{F}_{\text{L-MKPM}}(X, Y)$
$(\hat{\mathbf{x}}_0, \ldots, \hat{\mathbf{x}}_{n-1}) \leftarrow$ UnknotSnake$(j^*, \mathcal{L}_X^1)$
// Construct substitution map D.
Initialize empty dictionary D
for $i = 0, \ldots, n-1$ **do**
  for $j = 0, \ldots |\hat{\mathbf{x}}_i| - 1$ **do**
    $\mathrm{D}[\hat{\mathbf{x}}_i[j]] \leftarrow \mathbf{x}_i[j]$
$R_Y \leftarrow$ Substitute$(\mathcal{L}_Y^1, \mathrm{D})$
return R_Y

MakeSnake(j, T)

$(\mathbf{t}_0, \ldots, \mathbf{t}_{n-1}) \leftarrow T$
Initialize empty dictionary E
for $i = 0, \ldots, n-2$ **do**
  $\mathbf{x}_i \leftarrow \mathbf{t}_i \| \mathbf{t}_{i+1}[j]$
$v \leftarrow\$ \mathcal{I} \setminus \{\mathbf{t}_i[j] \mid i \in [0, n-1]\}$
$\mathbf{x}_{n-1} \leftarrow \mathbf{t}_{n-1} \| v$
return $(\mathbf{x}_0, \ldots, \mathbf{x}_{n-1})$

UnknotSnake$(j^*, \mathcal{L}_X^1)$

Initialize empty dictionary Loc
Fix an ordering $\mathcal{L}_X^1 = (\hat{\mathbf{x}}_0, \ldots, \hat{\mathbf{x}}_{n-1})$
$k \leftarrow |\hat{\mathbf{x}}_0|$ // $= |\hat{\mathbf{x}}_1| = \cdots = |\hat{\mathbf{x}}_{n-1}|$
// Store locations of linked values.
for $i = 0, \ldots, n-1$ **do**
  $\mathsf{Loc}[\hat{\mathbf{x}}_i[j^*]] \leftarrow i$
// Find start index ind of the linked list.
Find ind $\leq n-1$ s.t. $\hat{\mathbf{x}}_{\mathsf{ind}}[j^*] \notin \{\hat{\mathbf{x}}_i[k] \mid i \in [n]\}$
// Recover original ordering.
$i \leftarrow 0$
while $\hat{\mathbf{x}}_{\mathsf{ind}}[k] \in \mathsf{Loc}$ **do**
  $\mathbf{s}_i \leftarrow \hat{\mathbf{x}}_{\mathsf{ind}}$
  $\mathsf{ind} \leftarrow \mathsf{Loc}[\hat{\mathbf{x}}_{\mathsf{ind}}[k]]$
  $i \leftarrow i + 1$
return $(\mathbf{s}_0, \ldots, \mathbf{s}_{n-1})$

Figure 21: Snake attack with oracle access to the extended MKPM functionality $\mathcal{F}_{\text{L-MKPM}}(\cdot, Y)$. Y is part of the oracle and unknown to the adversary.