sciendo
Proceedings on Privacy Enhancing Technologies ; 2019 (3):289–309

Anja Lehmann

ScrambleDB: Oblivious (Chameleon) Pseudonymization-as-a-Service

Abstract: Pseudonymization is a widely deployed technique to de-sensitize data sets by consistently replacing identifying attributes with non-sensitive surrogates. However, all existing solutions are impractical to deploy in settings where data is accumulated from *distributed* sources: they either require sharing the same secret key with all sources, or rely on a fully trusted service to consistently compute these pseudonyms. Further, the *consistency* of pseudonyms, which is required to maintain the data's utility, comes with inherent and severe privacy limitations. This paper solves the key management and privacy challenges by introducing oblivious pseudonymization-as-a-service. Therein, the pseudonymization is outsourced to a central, yet fully oblivious entity, i.e., the service neither learns the sensitive information nor the pseudonyms it produces. Further, to obtain better privacy we no longer require pseudonyms to be computed consistently and instead introduce a dedicated join procedure. When data is stored at rest, all data is pseudonymized in a fully unlinkable manner. Only when certain subsets of the data are needed, the linkage is established through a controlled and non-transitive join operation. We formally define the desired security properties in the UC framework and propose a generic protocol that provably satisfies them. The core of our scheme is a 3-party oblivious and convertible PRF, which we believe to be of independent interest.

Keywords: pseudonymization, privacy, OPRF

DOI 10.2478/popets-2019-0048
Received 2018-11-30; revised 2019-03-15; accepted 2019-03-16.

1 Introduction

It is a widely held opinion that big data is the "new oil" of the digital economy. In particular, personal data is gaining more and more value and organizations increasingly collect large amounts of such data. When data gets accumulated, these large data bases are often outsourced to third parties, such as cloud or analytic providers. Thereby, legal constraints or security concerns often require the de-sensitization of the data before moving it across borders or into untrusted environments. A crucial technique used for de-sensitization is *pseudonymization* where uniquely identifying information, such as social security numbers, bank account numbers or identifying attribute combinations, e.g. names and addresses, are replaced with a random looking surrogate, the pseudonym. To preserve the utility of the data, all occurrences of the same unique identifier must *consistently* be replaced by the same pseudonym.

Pseudonymization is recommended by NIST for the protection of personal data [21], and even mandated in a number of industry-specific regulations, e.g., by HIPAA [31] or the Payment Card Industry Data Security Standard (PCI DSS) [25]. Furthermore, the General Data Protection Regulation (GDPR) Europe's new privacy regulations that came into effect in 2018, enforces strong rules for the protection of personal data. These strong rules only apply for *personal* data, and the GDPR explicitly recommends pseudonymization as a legitimate way to remove identifying information. For instance, the GDPR permits the processing of pseudonymized data for uses beyond the purpose for which the data was originally collected.

Key Management Challenges. A number of practical pseudonymization solutions have been introduced and are in commercial operation [1, 2, 29, 30, 32]. Typical solutions rely on (keyed) hash functions, encryption schemes, or non-cryptographic methods such as random substitution tables that must be kept secret. What all solutions have in common, is that they assume the pseudonymization to happen in a trusted environment, either directly at the data source itself or by a dedicated entity within the trust domain of the source.

In reality, however, data is often collected from multiple and distributed sources, which poses a number of challenges for the key management. Given that the pseudonymization is a *deterministic and keyed* process, all data sources must share the same secret pseudonymization key or, even worse, keep a shared and consistent version of the substitution table. Clearly, replicating the secret key material across a multitude of data sources, some even being outside of the trust domain of the en-

Anja Lehmann: IBM Research – Zurich, E-mail: anj@zurich.ibm.com

CC BY-NC-ND

tity collecting the data, is not desirable from a security perspective: if any of these data sources gets corrupted, the exposure of the secret key renders the pseudonymization useless.

Reverting to unkeyed pseudonymization is not an option either. As the identifying information has low entropy and the pseudonymization is deterministic, any unkeyed operation is vulnerable to *brute-force attacks* [20]. Services already offer to "reverse" the hashes of email addresses for 4 cents per hash [3].

A more practical and secure approach is to concentrate the pseudonymization task at a central trusted entity (TTP) which holds the secret key. The TTP then provides a service that consistently transforms the sensitive identifiers into their secure pseudonym. Using such a setting relieves the data sources from sharing any secret keys. Current solutions [2, 9, 32] require the full disclosure of the identifiers towards the TTP, though, which makes the TTP a *security and privacy bottleneck* itself. In particular when data is pseudonymized in an online manner and originates from various sources, having a single entity that can recognize and track the activities of the users is clearly not desirable.

The Risk of Linkability. Another – and more fundamental – challenge for pseudonymization is to find a tradeoff between privacy and utility. Pseudonymization is supposed to preserve the core utility of the data, which is realized by re-using the same pseudonym for every occurrence of the same unique identifier. This linkability preserves the crucial correlation among events, but gets also exploited for re-identification attacks, as impressively demonstrated for "anonymized" credit-card transactions [14] and the Netflix challenge [24]. For the former, more than 90 percent of individual consumers could be re-identified from the anonymized information.

To avoid such re-identification attacks that exploit the linkability of pseudonyms, the pseudonymization would have to replace all occurrences of the same identifier by different pseudonyms. While this is preferable from a security and privacy aspect, it clearly diminishes the utility of the pseudonymized data, as the correlation often contains valuable information. A middleground would be to use purpose-specific pseudonyms. This would ensure that data is only linkable within its context, but cannot be linked across them. However, this would require to know the exact purpose of all the data beforehand, as any further linkage *after* the data is collected would no longer be possible. Clearly, this is too restrictive for practical applications and the data collectors will hesitate to choose the option of unlinkability, as they fear to lose too much information by the irreversible decorrelation.

1.1 Our Contributions

In this work we address the aforementioned challenges and realize cryptographically strong pseudonymization in a secure and convenient way. Our solution, ScrambleDB, provides an oblivious service that generates privacy- and utility-preserving pseudonyms.

Oblivious Pseudonymization. Our solution implements the TTP approach but in a privacy-preserving, fully oblivious manner: A central service holds the secret keys and computes the pseudonyms. However, it does not learn any information about the identifiers that should be pseudonymized, nor about the blindly computed pseudonyms. In fact, the service cannot even tell whether two pseudonym requests are for the same identifier or not. Despite performing the pseudonymization in a blind manner, the final pseudonyms are produced in a utility-preserving way.

Chameleon Pseudonyms. To overcome the tension of privacy vs. utility we take a different spin on the problem of pseudonymization and decouple the pseudonymization used when the data is collected from when the data is actually used. Usually, while a large and continuously growing number of data is collected in the data lake, only much smaller and selected subparts are used in the data analysis. Thus when collected, data should be *fully unlinkable* per default. Only when strictly needed or desired, should the required pieces be made linkable in a controlled manner. To pseudonyms that allow for such flexibility and convertability we refer to as *chameleon pseudonyms*.

In our ScrambleDB solution the chameleon pseudonyms are computed in the oblivious-service model, allowing for a convenient deployment. This central service, which we call *converter* is the crucial entity to derive and convert pseudonyms. When a data source wishes to pseudonymize a data set containing of a collection of attributes for a number of users, the converter blindly derives a different pseudonym for every attribute. In the data lake, the data is stored in the form of many unlinkable and scrambled data snippets.

When a data processor is interested in a certain combination of attributes, it can obtain a *joined* version. This correlation can only be done by the converter, which enforces strong usage control of the data, thereby elegantly solving another requirement of the GDPR. To join the requested data, the converter blindly transforms

the unlinkable pseudonyms into a consistent representation, i.e., pseudonyms belonging to the same user will be mapped to the same value. To restrict the re-linked data to a particular purpose, the joins are strictly *non-transitive*, i.e., every conversion is done towards a join-specific representation and cannot be correlated with data received in another join request.

We formally define this concept of oblivious chameleon pseudonymization, discuss its guaranteed security and privacy properties, and finally propose a provably secure protocol. Our protocol realizes the desired security for passive adversaries, which allows for an efficient realization and provides sufficient guarantees in practice: the entities that would use such a system have basic trust in each other, and mainly aim to avoid data breaches. However, we also discuss weaker security notions that are achieved under active corruptions.

3-Party Oblivious & Convertible PRFs. Our ScrambleDB construction is built in a generic way from modular building blocks. The core of our scheme is a new primitive: a 3-party oblivious and convertible PRF. A conventional (2-party) OPRF [19] allows any party to blindly query a PRF. That is, the key holder of the PRF does not learn the values he computed the PRF on, nor the output he is blindly deriving.

We extend this primitive to a 3-party setting, meaning that the requester and receiver of a PRF value can be different parties. This is crucial for pseudonymization, as we do not want the (potentially untrusted) data sources to learn the pseudonyms (which will be PRF values) used in the data lake, and likewise the data lake is not supposed to learn the unique identifiers behind the pseudonyms. We then augment the 3-party OPRF to allow for a blind *conversions* among different PRF keys. Such a coPRF has an efficient algorithm that on input two keys k_i, k_j and a PRF value $y_i = \mathsf{PRF}(k_i, x)$ *blindly* transforms y_i into the corresponding PRF output $y_j = \mathsf{PRF}(k_i, x)$ of x under k_j.

We formalize the desired properties of coPRFs using game-based definitions, which we believe is better suited for building blocks than a UC-functionality: an ideal functionality hardcodes the exact set of security properties, whereas game-based definitions allow to easily omit certain properties when they are not needed.

We show that such coPRFs can be realized from DDH-PRF, re-randomizable and homomorphic encryption and the proxy re-encryption idea by Blaze et al. [6]. We stress that we do not claim the cryptographic techniques behind coPRFs to be particular novel. The contribution of this work is to show how existing techniques can be used to realize a new and versatile tool in provably secure manner. This is similar to key-homomorphic PRFs that have been proposed recently [7] and became useful in many practical applications and inspired several follow-up work, e.g., [5, 8, 16, 28].

1.2 Related Work

In the context of data exchange among *distributed* databases, a similar idea of using convertible pseudonyms was proposed by Camenisch and Lehmann [10, 11]. Therein, the different databases receive seemingly unlinkable pseudonyms which can only be converted from one database to another via a central entity. Within one database, all user data is associated with the same pseudonym though. Further, in [10] the pseudonym generation is non-blind, i.e., the unique identifiers had to be revealed to the converter. This was changed in [11] where pseudonyms get blindly derived by the user himself through an interactive protocol between the user, converter and the server he wants to establish a pseudonym with. Our work does not target such a user-centric setting, but instead aims at generating pseudonyms from untrusted data sources, which are not supposed to learn the pseudonyms (which the user does in [11]). Furthermore, both works aim at transitive transformations between established pseudonyms, whereas we focus on *non-transitive joins*. Thus, both schemes target a considerably different setting and do not provide the functionality we need.

Another line of related work is on adjustable joins over *encrypted databases*, which is an important feature e.g., in the CryptDB system by Popa et al. [27]. Recently, it was shown that CryptDB's join operator is transitive which reveals much more information than expected [22]. Mironov et al. argue that this flaw was due to an imprecise security model [26] and formally define the desired behaviour of non-transitive joins. In contrast to our work which aims at "public" de-sensitization, the setting of encrypted databases assumes that all data is prepared in a trusted environment using symmetric encryption. Thus, their solution is not applicable to our setting where data is accumulated from multiple, and possibly untrusted data sources. Further, the adjustable join tokens are computed by the secret key holder in a non-blind manner and get revealed to the data processors, whereas our joins are blind and performed by the oblivious converter. Overall, while similar in spirit, both settings are incomparable.

However, their work demonstrates the need of clear and formally sound security notions in order to avoid that seemingly privacy-preserving systems reveal more

information than intended. In fact, deploying such systems without rigorous security proofs can be devastating as the claimed privacy properties inspire wrong confidence and might lead users and organizations to put large amounts of personal data at risk.

2 Preliminaries

Here we introduce our notation for databases and the standard building blocks needed in our protocols.

Notation for Database Tables. We write $\mathsf{T}^{m\times n}$ to denote a database table that consists of n rows and m columns for attributes $attr_1, \ldots, attr_m \leftarrow \mathsf{ATTR}(\mathsf{T}^{m\times n})$. Each row is uniquely addressed by a primary key $uid_1, \ldots, uid_n \leftarrow \mathsf{KEY}(\mathsf{T}^{m\times n})$. A table is uniquely identified by the table identifier $tid \leftarrow \mathsf{ID}(\mathsf{T}^{m\times n})$. Each cell $\mathsf{T}^{m\times n}[i,j]$ contains a value $val_{i,j}$ for uid_i and $attr_j$, and for $i = 1, \ldots, n$ and $j = 1, \ldots, m$.

Public-Key Encryption. We need an encryption scheme $(\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ that is chosen-plaintext (CPA) secure. It consists of algorithms for key generation $(epk, esk) \xleftarrow{\$} \mathsf{KGen}(\tau)$, encryption $C \xleftarrow{\$} \mathsf{Enc}(epk, m)$, and decryption $m \leftarrow \mathsf{Dec}(esk, C)$.

We will require that ciphertexts are *re-randomizable*. That is, there must also be an algorithm $C' \xleftarrow{\$} \mathsf{ReRand}(epk, C)$ that returns a re-randomized version of the input ciphertext C. In terms of (additional) security we require that an adversary cannot distinguish a fresh encryption from a re-randomized ciphertext. The formal security definition for this property is given in Appendix A.1. For some of the encryption schemes we further require that the scheme has an appropriate *homomorphic property*, namely that there is an efficient operation $\odot$ on ciphertexts such that, if $C_1 \in \mathsf{Enc}(epk, m_1)$ and $C_2 \in \mathsf{Enc}(epk, m_2)$, then $C_1 \odot C_2 \in \mathsf{Enc}(epk, m_1 \cdot m_2)$. We use exponentiation to denote the repeated application of $\odot$.

We use the ElGamal encryption scheme, both for the instantiation of re-randomizable schemes and the ones that are also homomorphic. It is well known that the ElGamal scheme achieves these notions and is CPA secure. The CPA security is sufficient for our construction, as we consider mostly passive adversaries in this work. Let $(\mathbb{G}, g, q)$ be system parameters available as CRS such that the DDH problem is hard w.r.t. τ, i.e., q is a τ-bit prime. Then the re-randomizable version of ElGamal is defined as follows:

$\mathsf{ElG.KGen}(\tau):\ esk \xleftarrow{\$} \mathbb{Z}_q,\ epk \leftarrow g^{esk}$, output (esk, epk)

$\mathsf{ElG.Enc}(epk, m):\ r \xleftarrow{\$} \mathbb{Z}_q$, output $(epk^r, g^r m)$

$\mathsf{ElG.Dec}(esk, (C_1, C_2)):$ output $m' \leftarrow C_2 \cdot C_1^{-1/esk}$

$\mathsf{ElG.ReRand}(epk, (C_1, C_2)):\ r' \xleftarrow{\$} \mathbb{Z}_q,\ C_1' \leftarrow C_1 \cdot epk^{r'}$, $C_2' \leftarrow C_2 \cdot g^{r'}$, output $C' \leftarrow (C_1', C_2')$

Pseudorandom Functions. We require a standard pseudorandom function, consisting of a key generation $k \xleftarrow{\$} \mathsf{PRF.KGen}(\tau)$ and evaluation function $y \leftarrow \mathsf{PRF.Eval}(k, x)$. We also need a pseudorandom *permutation*, which in addition to $\mathsf{PRP.KGen}$ and $\mathsf{PRP.Eval}$ has an inversion algorithm $x \leftarrow \mathsf{PRP.Invert}(k, y)$.

3 Convertible & Oblivious PRFs

This section introduces our core building block: a 3-party oblivious and convertible PRF (coPRF). We first describe its expected behaviour (Sec. 3.1), then define the required security properties (Sec. 3.2) and present an efficient realization (Sec. 3.3). At the end we discuss the relation of coPRFs to OPRFs, key-homomorphic PRFs and proxy re-encryption (Sec. 3.4).

3.1 Functionality of 3-Party coPRFs

A first crucial difference to existing OPRFs, is that we are focusing on a *3-party* oblivious evaluation of the PRF. In the 3-party setting, a **requester** $\mathcal{R}$ wishes the PRF to be blindly evaluated on some input x towards a **receiver** $\mathcal{V}$. This blinded request, which we denote as $\overline{x}$, is sent to the **evaluator** $\mathcal{E}$ that is the entity holding the secret PRF key. The evaluator blindly transforms $\overline{x}$ into a blinded PRF output $\overline{y}$ of $y = \mathsf{coPRF.Eval}(k, x)$, and sends it to the receiver $\mathcal{V}$ which is the (only) entity that can unblind the response and retrieve y.

Further, to capture the conversion property where PRF values can be transformed from one key to another, we need to define the PRF in a *multi-key setting*. Therefore, we model key generation in two steps: the $\mathsf{coPRF.Setup}$ algorithm generates a master key msk from which domain-specific keys k_i for some index i can be derived via $k_i = \mathsf{coPRF.KGen}(msk, i)$. The index allows the requesting party to indicate under which key it wants the PRF to be evaluated on. Finally, we want the PRF to be *convertible*, i.e., there exists an efficient algorithm $\mathsf{coPRF.Convert}(k_i, k_j, y_i)$ that on input two keys k_i, k_j and a PRF value $y_i = \mathsf{coPRF.Eval}(k_i, x)$ transforms this value into the corresponding PRF output $y_j = \mathsf{coPRF.Eval}(k_i, x)$ of x under k_j.

More formally, a 3-party convertible PRF $\mathsf{coPRF} : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$ is defined through the following set of algorithms.

Main Algorithms:
Master Key: $msk \xleftarrow{\$} \mathsf{coPRF.Setup}(\tau)$
Key Generation: $k_i \leftarrow \mathsf{coPRF.KGen}(msk, i)$
Evaluation: $y \leftarrow \mathsf{coPRF.Eval}(k_i, x)$
Conversion: $y_j \leftarrow \mathsf{coPRF.Convert}(k_i, k_j, y_i)$

We require that conversions and fresh PRF evaluations are *consistent*, i.e., for all $msk \xleftarrow{\$} \mathsf{coPRF.Setup}(\tau)$, $x \in \mathcal{X}$, $i, j \in \{0,1\}^*$, $k_i \leftarrow \mathsf{coPRF.KGen}(msk, i)$, $k_j \leftarrow \mathsf{coPRF.KGen}(msk, j)$, we have that

$$\mathsf{coPRF.Eval}(k_j, x) = \mathsf{coPRF.Convert}(k_i, k_j, \mathsf{coPRF.Eval}(k_i, x)).$$

We want the evaluation and conversion to be computable in an oblivious manner. For the blind computation of coPRF.Eval, we require the algorithms stated below. To reflect the 3-party setting we define the blinding and unblinding operations in a public-key setting, i.e., the PRF receiver $\mathcal{V}$ will generate a key pair (bpk, bsk) via coPRF.BKGen and distribute the public blinding key bpk to the PRF requester, and use the secret key bsk for the unblinding of the PRF response.

Blinding Key Generation:
Key Generation $(bpk, bsk) \xleftarrow{\$} \mathsf{coPRF.BKGen}(\tau)$

Blind Evaluation:
Blinding: $\overline{x} \xleftarrow{\$} \mathsf{coPRF.Eval.Blind}(bpk, x)$
Evaluation: $\overline{y} \xleftarrow{\$} \mathsf{coPRF.Eval.Exec}(k_i, bpk, \overline{x})$
Unblinding: $y \leftarrow \mathsf{coPRF.Eval.Unblind}(bsk, \overline{y})$

Similarly, the oblivious, 3-party computation of coPRF.Convert is defined via the three algorithms below. The conversion request $\overline{y}_i$ is computed by a requester $\mathcal{R}$ that wishes to convert the value y_i towards a receiver $\mathcal{V}$ and the secret key with index j. The evaluator blindly computes $\overline{y}_j$ which is the converted PRF value $y_j = \mathsf{coPRF.Convert}(k_i, k_j, y_i) = \mathsf{coPRF.Eval}(k_j, x)$. The blinding and unblinding leverages a key pair (bpk, bsk) of the receiver derived via coPRF.BKGen as defined above.

Blind Conversion:
Blinding: $\overline{y_i} \xleftarrow{\$} \mathsf{coPRF.Convert.Blind}(bpk, y_i)$
Conversion: $\overline{y_j} \xleftarrow{\$} \mathsf{coPRF.Convert.Exec}(k_i, k_j, bpk, \overline{y_i})$
Unblinding: $y_j \leftarrow \mathsf{coPRF.Convert.Unblind}(bsk, \overline{y_j})$

For *correctness* of blind evaluation and conversion we require that any computation through the blinded algorithms performed by honest parties leads to the same value that would be derived using the non-blind counterparts coPRF.Eval and coPRF.Convert.

Whereas most previous work defined OPRFs in form of ideal functionalities in the UC framework, we opted for an algorithm-based definition as it gives direct access to all intermediate values, and thus allows a more flexible use in the protocol design. For instance, it enables *batch evaluation*, i.e., a single evaluation request $\overline{x}$ can be used by the evaluator to derive multiple blinded PRF outputs $\overline{y}_1, \ldots, \overline{y}_n$ under different secret keys, which would not be possible under a UC-based definition.

3.2 Security Notions for coPRFs

We now present our security definitions for convertible and oblivious PRFs. Some core differences to existing notions for oblivious PRFs [17, 19] already stem from our targeted *3-party setting*: The previous works aim at 2-party OPRFs, i.e., they assume that the party querying the OPRF and receiving the final value are always the same entity, whereas our definition is more generic and allows them to be different. This new setting motivates a new security property that is not captured by 2-party OPRFs: if the requester and receiver collude, they should not be able to correlate the PRF's in- and outputs (beyond what is trivially possible given the determinism of the PRF).

The security definitions are defined via games that an adversary runs with a challenger, covering different corruption settings: the notions of *pseudorandomness*, *collusion-resistance* and *one-more unpredictability* provide security against corrupt requesters and receivers, whereas *obliviousness* and *input-hiding* capture the guarantees of coPRFs against corrupt evaluators (and receivers). Our strong notion of collusion-resistance focuses on *passive* adversaries, and one-more unpredictability is a weaker notion that allows *active* adversaries but still enables highly efficient realizations.

Writing Conventions. For brevity we will often write k_i in our definition without making the key generation explicit, i.e., whenever we write k_i we mean $k_i \leftarrow \mathsf{coPRF.KGen}(msk, i)$. Further, when we write *"retrieve a record"* in an oracle but no matching record exists, the oracle call is ignored.

3.2.1 Pseudorandomness

The core property of coPRFs is that, without knowing the secret key, their outputs should be indistinguishable from random — bearing in mind the added conversion functionality. We require this property for the main, i.e., non-blind algorithms coPRF.Eval and coPRF.Convert and extend it to the blind evaluation in the next definition.

The pseudorandomness of convertible PRFs is captured in the following definition where the adversary is given access to oracles $\mathcal{O}^b_{\mathsf{Eval}}$ and $\mathcal{O}^b_{\mathsf{Convert}}$ (Fig. 1). For $b = 0$ the oracles return the real coPRF output whereas

$\mathcal{O}^b_{\mathsf{Eval}}(i,x)$	$\mathcal{O}^b_{\mathsf{Convert}}(i,j,y_i)$

if $b=0$ (Real):
$y_i \leftarrow \mathsf{coPRF.Eval}(k_i,x)$
if $b=1$ (Random):
$y_i \leftarrow f_i(x)$
add (i,x,y_i) to $\mathbf{V}_{\mathsf{PRF}}$
return y_i
	retrieve (i,x,y_i) from $\mathbf{V}_{\mathsf{PRF}}$
	if $b=0$ (Real):
	$y_j \leftarrow \mathsf{coPRF.Convert}(k_i,k_j,y_i)$
	if $b=1$ (Random):
	$y_j \leftarrow f_j(x)$
	add (j,x,y_j) to $\mathbf{V}_{\mathsf{PRF}}$
	return y_j

Fig. 1. Oracles for the pseudorandomness definition of coPRFs. f_i is a random function $f_i : \mathcal{X} \to \mathcal{Y}$ that, for every new i, is chosen at random from the family of such functions, and gets re-used for all recurring calls for the same i.

for $b=1$ the output is derived via a random function f_i. To capture the conversion capability in the ideal random setting we enforce that $\mathcal{O}^b_{\mathsf{Convert}}$ can only be invoked on PRF values that have been obtained through the $\mathcal{O}^b_{\mathsf{Eval}}$ oracle. As we keep records (i,x,y) for all evaluation requests in a list $\mathbf{V}_{\mathsf{PRF}}$ this allows the experiment to look up the underlying value x and return $f_i(x)$ in the conversion call when being in the ideal setting with $b=1$.

Definition 3.1 (Pseudorandomness)**.** *A* coPRF *is called pseudorandom if for all PPT adversaries $\mathcal{A}$ it holds that* $\left|\Pr[\mathsf{Exp}^{\mathsf{coPRF}}_{\mathcal{A},\mathsf{rand}}(\tau)=1]-1/2\right| \leq \mathsf{negl}(\tau)$ *for the following experiment and the oracles defined in Figure 1.*

Experiment $\mathsf{Exp}^{\mathsf{coPRF}}_{\mathcal{A},\mathsf{rand}}(\tau)$:
$msk \xleftarrow{\$} \mathsf{coPRF.Setup}(\tau)$, $\mathbf{V}_{\mathsf{PRF}} \leftarrow \emptyset$, $b \xleftarrow{\$} \{0,1\}$
$b' \xleftarrow{\$} \mathcal{A}^{\mathcal{O}^b_{\mathsf{Eval}},\mathcal{O}^b_{\mathsf{Convert}}}(\tau)$
return *1 if* $b=b'$

3.2.2 Security against Corrupt Requester & Receiver

The following definition covers security against malicious requester and receiver, guaranteeing that the blind computations of coPRF.Eval and coPRF.Convert do not leak more information than their non-blind counterparts. This does not follow from the pseudorandomness and correctness requirement of blind evaluations and conversions, e.g., coPRF.Eval.Exec could output the key k_i as part of $\overline{y}$ without harming either requirement.

Our first definition is targeted to passive attackers, also known as honest-but-curious adversaries. They follow the protocol correctly but try to learn as much information as possible from its execution. A second and weaker notion then captures security against active adversaries.

Passive Security. This security property is phrased by the experiment defined below and for the oracles defined in Figure 2. Here we ask the adversary to distinguish between a real ($b=0$) and a simulated ($b=1$) setting. In the real setting, the oracles return the normal *blind* coPRF outputs. In the simulated world, the outputs are derived from a simulator SIM that gets the corresponding *non-blind* evaluation of coPRF as input. For collusion-resistance we require that their must exist a simulator that makes both worlds indistinguishable.

Note that the simulator has to produce indistinguishable $\overline{y}$ values based purely on the input of y and bpk. As SIM doesn't get the evaluation or conversion request $\overline{x}$ or $\overline{y}$ as input, this definition ensures that the blinded PRF output $\overline{y}$ is *unlinkable* to the blinded input. This is a crucial property for our pseudonym system where a sender pseudonymizes a *batch* of many primary identifiers towards the data lake, and a colluding requester and receiver should not be able to link the identifiers to their pseudonyms.

Definition 3.2 (Collusion Resistance)**.** *A* coPRF *is called collusion-resistant if for all PPT adversaries $\mathcal{A}$ there exists a PPT simulator* SIM *such that* $\left|\Pr[\mathsf{Exp}^{\mathsf{coPRF}}_{\mathcal{A},\mathsf{collres}}(\tau)=1]-1/2\right| \leq \mathsf{negl}(\tau)$ *for the following experiment and the oracles defined in Figure 2.*

Experiment $\mathsf{Exp}^{\mathsf{coPRF}}_{\mathcal{A},\mathsf{collres}}(\tau)$:
$msk \xleftarrow{\$} \mathsf{coPRF.Setup}(\tau)$, $\mathbf{V}_{\overline{x}}, \mathbf{V}_{\overline{y}}, \mathbf{V}_{\mathsf{PRF}} \leftarrow \emptyset$, $b \xleftarrow{\$} \{0,1\}$
$b' \xleftarrow{\$} \mathcal{A}^{\mathcal{O}_{\mathsf{Blind}},\mathcal{O}^b_{\mathsf{Eval.Exec}},\mathcal{O}^b_{\mathsf{Convert.Exec}}}(\tau)$
return *1 if* $b=b'$

As said, this definition is targeted to passive attackers. This is modeled by granting the adversary a blinding oracle $\mathcal{O}_{\mathsf{Blind}}$ which he can use to obtain correctly formed $\overline{x}$ or $\overline{y}$ values, including the used randomness. We then require $\mathcal{A}$ to invoke the $\mathcal{O}^b_{\mathsf{Eval.Exec}}$ and $\mathcal{O}^b_{\mathsf{Convert.Exec}}$ only on blinded values obtained via these oracles.

We stress that for any definition of *consistent* and *blind* coPRFs the definition must enforce that inputs are properly formed and the blinded values are extractable. When allowing active attacks, the instantiation will require heavy tools such as zero-knowledge proofs, privacy-preserving signatures and extractors – which we try to avoid. We believe that passive attacks still capture the main threat in practice, namely the corruption of the secret key and full state of a party. Our definition guarantees security against these attacks and also enables highly efficient instantiations.

In fact, this trade-off resembles the current state in the related area of private set intersection: Therein passive security is accepted as the de-facto standard for practical solutions, as even the most efficient adaptions to malicious security require significant additions to the passively-secure protocols [13].

Active Security. While guaranteeing the pseudorandomness and collusion-resistance properties against ac-

$\mathcal{O}_{\mathsf{Blind}}(\cdot,\cdot,\cdot)$	$\mathcal{O}^b_{\mathsf{Eval.Exec}}(i, bpk, \overline{x})$	$\mathcal{O}^b_{\mathsf{Convert.Exec}}(i, j, bpk, \overline{y}_i)$

upon input (Eval, bpk, x):
get $\overline{x} \xleftarrow{\$}$ coPRF.Eval.Blind(bpk, x)
add $(bpk, x, \overline{x})$ to $\mathbf{V}_{\overline{x}}$
return $(\overline{x}, r_\$)$
upon input (Convert, bpk, y):
get $\overline{y} \xleftarrow{\$}$ coPRF.Convert.Blind(bpk, y)
add $(bpk, y, \overline{y})$ to $\mathbf{V}_{\overline{y}}$
return $(\overline{y}, r_\$)$
	retrieve $(bpk, x, \overline{x})$ from $\mathbf{V}_{\overline{x}}$
	get $y_i \leftarrow$ coPRF.Eval(k_i, x)
	add (i, x, y_i) to $\mathbf{V}_{\mathsf{PRF}}$
	if $b = 0$ (Real):
	get $\overline{y}_i \xleftarrow{\$}$ coPRF.Eval.Exec$(k_i, bpk, \overline{x})$
	if $b = 1$ (Simulated):
	get $\overline{y}_i \xleftarrow{\$}$ SIM(Eval, bpk, y_i)
	return $\overline{y}_i$
		retrieve $(bpk, y_i, \overline{y}_i)$ from $\mathbf{V}_{\overline{y}}$
		and (i, x, y_i) from $\mathbf{V}_{\mathsf{PRF}}$
		get $y_j \leftarrow$ coPRF.Convert(k_i, k_j, y_j)
		add (j, x, y_j) to $\mathbf{V}_{\mathsf{PRF}}$
		if $b = 0$ (Real):
		get $\overline{y}_j \xleftarrow{\$}$ coPRF.Convert.Exec$(k_i, k_j, bpk, \overline{y}_i)$
		if $b = 1$ (Simulated):
		get $\overline{y}_j \xleftarrow{\$}$ SIM(Convert, bpk, y_j)
		return $\overline{y}_j$

Fig. 2. Oracles for the collusion-resistance definition, where $r_\$$ in the $\mathcal{O}_{\mathsf{Blind}}$ oracle is the randomness used by the internal algorithm.

tive adversaries would be very costly to realize, we also propose a weaker property that can be achieved much more efficiently: *one-more unpredictability*. This notion captures that even an actively corrupted requester and receiver cannot learn anything from the evaluator's key that would enable them to compute or convert PRF values on their own. This is defined via a game where the adversary can query $\mathcal{O}_{\mathsf{Eval}}$ and $\mathcal{O}_{\mathsf{Convert}}$ oracles on (maliciously crafted) inputs of his choice and has to come up with more valid PRF tuples than he made queries to the oracles. A similar notion has been used for the (2-Party) OPRF in [15]. Compared with the pseudorandomness and collusion-resistance properties, this definition does not guarantee consistency or committed-input computation, but ensures that the secrecy of the evaluators PRF key is not impacted by any malicious behaviour of the requester and receiver. The formal definition of this property is given in Appendix A.3.

3.2.3 Security against Corrupt Evaluator

We now turn to the properties coPRFs must guarantee against malicious evaluators. The notion of *obliviousness* guarantees the blindness of the values that the evaluator receives and outputs. The second definition of *input hiding* captures the security that remains when the evaluator colludes with the receiver, and thus learns the outputs he computes. Due to the determinism of the PRF, the security in this setting is limited and boils down to brute-force attacks on the input space. We stress that the latter is unavoidable when the secret key and the output of the keyed and deterministic function are known to the adversary. In the protocol this can be strengthened easily though, e.g., by simply applying another PRF on the in- and/or output, which is what we will do in the ScrambleDB scheme.

Obliviousness. This is the core property we want to ensure against a corrupt PRF evaluator $\mathcal{E}$. When the requester and receiver in a blind evaluation or conversion session are honest, then a corrupt $\mathcal{E}$ should not learn anything about the inputs it is receiving or the outputs it is computing. As blinding does not require any secret keys, this definition does not give any oracle access to the adversary. Note that obliviousness as defined below also guarantees *unlinkability*, i.e., a corrupt evaluator cannot tell whether two blinded inputs belong to the same x and y respectively.

Definition 3.3 (Obliviousness). *A* coPRF *is called fully oblivious if for all PPT adversaries $\mathcal{A}$ in the experiments below it holds that* $|\Pr[\mathsf{Exp}^{\mathsf{coPRF}}_{\mathcal{A},\mathsf{blindX}}(\tau) = 1] - 1/2| \leq \mathsf{negl}(\tau)$ *for* $\mathsf{X} \in \{\mathsf{eval}, \mathsf{convert}\}$.

Experiment $\mathsf{Exp}^{\mathsf{coPRF}}_{\mathcal{A},\mathsf{blindeval}}(\tau)$:	**Experiment** $\mathsf{Exp}^{\mathsf{coPRF}}_{\mathcal{A},\mathsf{blindconvert}}(\tau)$:
$(bpk, bsk) \xleftarrow{\$}$ coPRF.BKGen(τ)	$(bpk, bsk) \xleftarrow{\$}$ coPRF.BKGen(τ)
$b \xleftarrow{\$} \{0,1\}$	$b \xleftarrow{\$} \{0,1\}$
$(x_0, x_1, state) \xleftarrow{\$} \mathcal{A}(bpk)$	$(y_0, y_1, state) \xleftarrow{\$} \mathcal{A}(bpk)$
proceed only if $x_0, x_1 \in \mathcal{X}$	proceed only if $y_0, y_1 \in \mathcal{Y}$
$\overline{x}_b \xleftarrow{\$}$ coPRF.Eval.Blind(bpk, x_b)	$\overline{y}_b \xleftarrow{\$}$ coPRF.Convert.Blind(bpk, y_b)
$b' \xleftarrow{\$} \mathcal{A}(state, \overline{x}_b)$	$b' \xleftarrow{\$} \mathcal{A}(state, \overline{y}_b)$
return *1 if* $b = b'$	return *1 if* $b = b'$

Input Hiding. This notion captures the privacy guarantees against a malicious evaluator and receiver. Clearly, when both the PRF evaluator and the PRF receiver are corrupt, we can no longer guarantee the full obliviousness as defined above. However, we still want the blinded PRF input x to be as "hidden" as possible. This is defined by requiring the blinded input message to be simulatable with the input of only some leakage $\mathsf{leak}(x)$ of x. The strength of this notion obviously depends on the leakage function leak: in the extreme case of $\mathsf{leak}(x) = x$ the definition becomes meaningless as the simulator has the full knowledge of the input value. Our construction will realize the input-hiding notion for the leakage being the (one-way) hash function $\mathsf{H}_{\mathbb{G}}$.

The coPRF then inherits the properties of leak, meaning when leak is one-way, this notion models that the coPRF is not reversible (on random, high-entropy inputs) even if an adversary is given the secret key. This is often desired in the context of pseudonymization, as pseudonyms produced by an inherently reversible function would not be considered to be fully de-identifying

and thus require stronger protection than pseudonyms produced via one-way functions. For space reasons, we present the formal definition in Appendix A.2.

3.3 Secure Instantiation of coPRF

The following instantiation securely realizes coPRF as just defined. The construction is based on the PRF by Naor, Pinkas, and Reingold [23] where an input x is first hashed and then raised to an exponent that is the evaluator's secret key as $y = \mathsf{H}_{\mathbb{G}}(x)^{k_i}$. This has been shown to be a secure PRF when the DDH assumption is hard in $\mathbb{G}$ and $\mathsf{H}_{\mathbb{G}}$ is a random oracle. The conversion is realized in a proxy re-encryption manner using the re-encryption idea first proposed by Blaze et al. [6]. That is, conversion of a PRF value $y = \mathsf{H}_{\mathbb{G}}(x)^{k_i}$ towards a key k_j is realized by using the quotient of the source and target key as $y_j = y_i^{k_j/k_i}$. 3-Party obliviousness is realized by encrypting all inputs towards the evaluator with a homomorphic encryption scheme HE under the public key of the PRF receiver. We also require the homomorphic encryption scheme to be re-randomizable, i.e., a ciphertext can be re-randomized such that it becomes indistinguishable from a fresh encryption. The evaluator uses that property to re-randomize every ciphertext it receives, which guarantees collusion-resistance. Finally, we use a standard PRF to derive the individual coPRF keys k_i from a master key msk and an index $i \in \{0,1\}^*$. When used as building block in a protocol, the index will allow to "publicly" steer the key under which the coPRF shall be evaluated.

Construction $\mathsf{coPRF}_{\mathsf{DDH}}$. Our construction assumes the availability of public parameters $(\mathbb{G}, g, q)$ such that the DDH problem is hard w.r.t. the security parameter τ. More precisely, $\mathbb{G}$ is a cyclic group of order q where q is a τ-bit prime. The $\mathsf{coPRF}_{\mathsf{DDH}}$ scheme makes use of a hash function $\mathsf{H}_{\mathbb{G}} : \{0,1\}^* \to \mathbb{G}$ and a standard pseudorandom function $\mathsf{PRF}_q = (\mathsf{PRF}_q.\mathsf{KGen}, \mathsf{PRF}_q.\mathsf{Eval})$ with $\mathsf{PRF}_q : \mathcal{K} \times \{0,1\}^* \to \mathbb{Z}_q$. Note that the choice of these functions is influenced by the group parameters which in turn depend on the security parameter τ. Further let $\mathsf{HE} = (\mathsf{HE.KGen}, \mathsf{HE.Enc}, \mathsf{HE.Dec}, \mathsf{HE.ReRand})$ be a homomorphic and re-randomizable encryption scheme that is compatible with $\mathbb{G}$. Then, $\mathsf{coPRF}_{\mathsf{DDH}} : \{0,1\}^* \times \{0,1\}^* \to \mathbb{G}$ is defined as:

Main Algorithms:

$\mathsf{Setup}(\tau)$:	$msk \xleftarrow{\$} \mathsf{PRF}_q.\mathsf{KGen}(\tau)$
$\mathsf{KGen}(msk, i)$:	$k_i \leftarrow \mathsf{PRF}_q.\mathsf{Eval}(msk, i)$
$\mathsf{Eval}(k_i, x)$:	$y \leftarrow \mathsf{H}_{\mathbb{G}}(x)^{k_i}$
$\mathsf{Convert}(k_i, k_j, y_i)$:	$y_j \leftarrow y_i^{\Delta}$ with $\Delta \leftarrow k_j/k_i$

Blind Evaluation & Conversion:

$\mathsf{BKGen}(\tau)$:	$(bpk, bsk) \xleftarrow{\$} \mathsf{HE.KGen}(\tau)$
$\mathsf{Eval.Blind}(bpk, x)$:	$\overline{x} \xleftarrow{\$} \mathsf{HE.Enc}(bpk, \mathsf{H}_{\mathbb{G}}(x))$
$\mathsf{Eval.Exec}(k_i, bpk, \overline{x})$:	$\overline{y} \xleftarrow{\$} \mathsf{HE.ReRand}(bpk, \overline{x})^{k_i}$
$\mathsf{Eval.Unblind}(bsk, \overline{y})$:	$y \leftarrow \mathsf{HE.Dec}(bsk, \overline{y})$
$\mathsf{Convert.Blind}(bpk, y_i)$:	$\overline{y_i} \xleftarrow{\$} \mathsf{HE.Enc}(bpk, y_i)$
$\mathsf{Convert.Exec}(k_i, k_j, bpk, \overline{y_i})$:	$\overline{y_j} \xleftarrow{\$} \mathsf{HE.ReRand}(bpk, \overline{y}_i)^{k_j/k_i}$
$\mathsf{Convert.Unblind}(bsk, \overline{y_j})$:	$y_j \leftarrow \mathsf{HE.Dec}(bsk, \overline{y}_j)$

It is easy to see that $\mathsf{coPRF}_{\mathsf{DDH}}$ is consistent, and the blind evaluation and conversion procedures are correct as defined in Section 3.1. We have already informally argued how the individual building blocks are used to satisfy all the required properties of coPRFs and give the details of the straightforward proofs for the following theorems in the full version of this paper.

Theorem 3.4 (Pseudorandomness)**.** *The construction* $\mathsf{coPRF}_{\mathsf{DDH}}$ *described above is pseudorandom if* PRF_q *is pseudorandom,* $\mathsf{H}_{\mathbb{G}}$ *is a random oracle, and the DDH assumption holds in* $\mathbb{G}$*.*

Theorem 3.5 (Collusion-resistance)**.** *The construction* $\mathsf{coPRF}_{\mathsf{DDH}}$ *is collusion-resistant if* HE *is re-randomizable (as defined in Def. A.1).*

Theorem 3.6 (Obliviousness)**.** *The construction* $\mathsf{coPRF}_{\mathsf{DDH}}$ *is fully oblivious if* HE *is CPA-secure.*

Theorem 3.7 (Input-Hiding)**.** *The construction* $\mathsf{coPRF}_{\mathsf{DDH}}$ *described above is input-hiding w.r.t. the one-way function* $\mathsf{leak} = \mathsf{H}_{\mathbb{G}}$ *if* $\mathsf{H}_{\mathbb{G}}$ *is one-way.*

Our construction does not achieve collusion-resistance and pseudorandomness as defined in Section 3.2 against *actively* corrupt requester, as this would require additional zero-knowledge proofs and extractability to enforce that inputs are well-formed. Instead, we show that $\mathsf{coPRF}_{\mathsf{DDH}}$ satisfies the notion of one-more unpredictability against active adversaries, which ensures the overall safety of the evaluator's secret key. This property relies on the One-More DH with Inverse Oracle (OMDH-IO) assumption that was introduced recently [18], and is a slight modification of the standard One-More DH assumption that gives the adversary access to an inversion oracle $(\cdot)^{1/x}$ in addition to $(\cdot)^x$.

Theorem 3.8 (Active Security of $\mathsf{coPRF_{DDH}}$). *The construction $\mathsf{coPRF_{DDH}}$ described above is one-more unpredictable if PRF_q is pseudorandom, $\mathsf{H}_{\mathbb{G}}$ is a random oracle, and the OMDH-IO assumption holds in $\mathbb{G}$.*

3.4 Related Work for coPRFs

Our 3-party coPRF can be seen as a generalization and extension of 2-party OPRFs, i.e., coPRFs can be used in the simpler 2-party setting by making the requester and receiver the same party. However, the 3-party setting and the additional collusion-resistance property allow to use coPRFs in applications where no party should be privy of both the in- and output of the PRF, which is exactly what we will need for our pseudonym system: therein batches of identifiers get replaced by pseudonyms and the coPRF hides the relation between them, even when the requester and receiver are corrupt.

In Appendix B we discuss the relation between our coPRF and similar concepts such as key-homomorphic PRFs (KH-PRF) and proxy re-encryption (PRE). In short, convertible PRFs are orthogonal to the both primitives. KH-PRFs require to know the preimage x in order to convert PRF values between keys, whereas coPRF allows conversions based on y only. In PRE, an untrusted proxy can re-encrypt ciphertexts from one key to another, without knowing the secret keys but seeing the ciphertexts it is converting. Whereas in coPRFs the converter *does* know all secret keys but must do the conversion blindly.

4 ScrambleDB

We now present ScrambleDB in which a central and oblivious pseudonymization service is used to desensitize data from *distributed* sources. At the same time, ScrambleDB overcomes the privacy limitations that are inherent whenever globally consistent pseudonyms are used in order to preserve the utility of the data. The main idea is to distinguish between pseudonymization used for data collection and data usage: when collected, data should be *fully unlinkable* per default. Only when strictly needed or desired, should the required pieces be made linkable in a controlled manner, again using the oblivious service for simple deployment. To pseudonyms that allow for such flexibility and convertability we refer to as *chameleon pseudonyms*.

We start with the high-level idea of ScrambleDB (Sec. 4.1), then present our formal security model (Sec. 4.2) and finally propose our protocol (Sec. 4.3).

4.1 High-Level Idea

We have four entities in our setting: a number of **data sources** $\mathcal{S}$, the central pseudonym **converter** $\mathcal{C}$, the **data lake** $\mathcal{L}$, and a number of **data processors** $\mathcal{P}$.

Pseudonymization. The data sources hold tables $\mathsf{T}^{m\times n}$ containing attribute values for n users and m attribute types which they want to upload in pseudonymized form to the data lake. All pseudonymization has to go through the converter $\mathcal{C}$ which breaks the table into m tables each indexed with attribute-specific pseudonyms. The data lake receives and stores such pseudonymized and unlinkable tables.

More precisely, pseudonymization is triggered by some data source $\mathcal{S}$ holding a table $\mathsf{T}^{m\times n}$. The primary key of each row is the unique identifier that is supposed to be pseudonymized, which we write as $(uid_1,\ldots,uid_n) \leftarrow \mathsf{KEY}(\mathsf{T}^{m\times n})$ (see Sec. 2 for more writing-conventions for database tables). The pseudonymization is requested towards the converter $\mathcal{C}$. When $\mathcal{C}$ approves, it derives different pseudonyms for every attribute in the table. That is, $\mathcal{C}$ breaks the $\mathsf{T}^{m\times n}$ table into m tables $\mathsf{T}^{1\times n}$, one table for every attribute, and blindly derives attribute-specific pseudonyms $nym_{i,j}$ for every combination of uid_i and attribute type $attr_j$. At the end, $\mathcal{L}$ receives m shuffled tables $(\mathsf{T}_1^{1\times n},\ldots,\mathsf{T}_m^{1\times n})$, each with pseudonyms $\mathsf{KEY}(\mathsf{T}_j^{1\times n}) \leftarrow (nym_{1,j},\ldots,nym_{n,j})$ as primary keys. Thus, all data stored by $\mathcal{L}$ is broken into unlinkable and scrambled data snippets.

Join. The data processors are entities that wish to receive a *joined* version of some of the tables held by $\mathcal{L}$. For instance, $\mathcal{P}$ could be a data analyst that aims to access data from an intra-enterprise data lake $\mathcal{L}$, where ScrambleDB is used to implement the principle of data minimization within their own premises. The data lake could also be a collection of pseudonymized medical data accumulated from different sources to which external researchers can request selected access to. Depending on the use case, $\mathcal{P}$ might have full access to the pseudonymized (but fully unlinkable) data held by $\mathcal{L}$, only see data excerpts or even only learn the attribute types and quantities. How $\mathcal{P}$ will request the information from $\mathcal{L}$ is very application-specific and will be outside of our model. We simply consider that $\mathcal{L}$ has already agreed on producing a joined output of certain tables $\mathsf{T}_1^{1\times n_1},\ldots \mathsf{T}_l^{1\times n_l}$ towards $\mathcal{P}$.

Thus, the join process is triggered by the data lake for a particular data processor $\mathcal{P}$ and requires the explicit approval of the converter. If $\mathcal{C}$ gives such approval, then $\mathcal{P}$ (and only he) will receive transformed tables in

1. Pseudonymization Request. On input $(\textsc{NymReq}, sid, pqid, \mathsf{T}^{m\times n})$ from a data source $\mathcal{S}$:
   - If $\mathcal{C}$ and $\mathcal{L}$ are corrupt: Set $\overline{\mathsf{T}}^{m\times n} \leftarrow \mathsf{T}^{m\times n}$ with $\mathsf{KEY}(\overline{\mathsf{T}}^{m\times n}) \leftarrow (\mathsf{leak}(uid_i), \ldots, \mathsf{leak}(uid_n))$ for $(uid_1, \ldots, uid_n) \leftarrow \mathsf{KEY}(\mathsf{T}^{m\times n})$, and set $\ell \leftarrow \overline{\mathsf{T}}^{m\times n}$. Otherwise set $\ell \leftarrow \mathsf{size}(\mathsf{T}^{m\times n})$.
   - Send $(\textsc{NymReq}, sid, pqid, \mathcal{S}, \ell)$ to $\mathcal{A}$ and wait for $(\textsc{NymReq}, sid, pqid, \mathtt{ok})$ from $\mathcal{A}$.
   - Create a record $(\mathtt{nymreq}, sid, pqid, \mathsf{T}^{m\times n}, \mathcal{S})$ and output $(\textsc{NymReq}, sid, pqid, \mathcal{S}, \mathsf{ATTR}(\mathsf{T}^{m\times n}), n)$ to $\mathcal{C}$.
2. Pseudonymized and Unlinkable Data Output. On input of $(\textsc{NymOut}, sid, pqid)$ from the converter $\mathcal{C}$:
   - Proceed only if a record $(\mathtt{nymreq}, sid, pqid, \mathsf{T}^{m\times n}, \mathcal{S})$ for $pqid$ exists.
   - Send $(\textsc{NymOut}, sid, pqid)$ to $\mathcal{A}$ and wait for $(\textsc{NymOut}, sid, pqid, \mathtt{ok})$.
   - Get $(attr_1, \ldots, attr_m) \leftarrow \mathsf{ATTR}(\mathsf{T}^{m\times n})$, and $(uid_1, \ldots, uid_n) \leftarrow \mathsf{KEY}(\mathsf{T}^{m\times n})$ for the retrieved $\mathsf{T}^{m\times n}$.
   - For $j = 1, \ldots m$:
     - Create $\mathsf{T}_j^{1\times n}$ by using the j-th column of $\mathsf{T}^{m\times n}$, set $\mathsf{ID}(\mathsf{T}_j^{1\times n_j}) \leftarrow (\mathsf{ID}(\mathsf{T}^{m\times n}), attr_j)$, and add $\mathsf{T}_j^{1\times n}$ to **Tables**.
     - Produce the pseudonymized version of $\mathsf{T}_j^{1\times n}$ as follows:
       * For $i = 1, \ldots, n$: Get $nym_{i,j} \leftarrow \mathbf{Nyms}(uid_i, attr_j)$. If $nym_{i,j} = \perp$, choose $nym_{i,j} \xleftarrow{\$} \mathbf{N}$ and set $\mathbf{Nyms}(uid_i, attr_j) \leftarrow nym_{i,j}$.
       * Set $\mathsf{KEY}(\mathsf{T}_j^{1\times n}) \leftarrow (nym_{1,j}, \ldots, nym_{n,j})$ and sort $\mathsf{T}_j^{1\times n}$ by the primary key.
   - Output $(\textsc{NymOut}, sid, pqid, \mathcal{S}, (\mathsf{T}_1^{1\times n}, \ldots, \mathsf{T}_m^{1\times n}))$ to the data lake $\mathcal{L}$.
3. Join Request. On input $(\textsc{JoinReq}, sid, jqid, (tid_1, \ldots, tid_l), \mathcal{P})$ from the data lake $\mathcal{L}$:
   - Proceed only if $\mathsf{T}_1^{1\times n_1}, \ldots \mathsf{T}_l^{1\times n_l} \in \mathbf{Tables}$ for all $(tid_1, \ldots, tid_l)$.
   - If $\mathcal{C}$ and $\mathcal{P}$ are corrupt: For $j = 1, \ldots, l$ set $\overline{\mathsf{T}}_j^{1\times n_j} \leftarrow \mathsf{T}_j^{1\times n_j}$ with $\mathsf{KEY}(\overline{\mathsf{T}}_j^{1\times n_j}) \leftarrow (\mathsf{leak}(uid_1), \ldots, \mathsf{leak}(uid_{n_j}))$ for $(uid_1, \ldots, uid_{n_j}) \leftarrow \mathsf{KEY}(\mathsf{T}_j^{1\times n_j})$, and set $\ell \leftarrow \{\overline{\mathsf{T}}_j^{1\times n_j}\}_{j=1,\ldots,l}$. Otherwise, set $\ell \leftarrow (\mathsf{size}(\mathsf{T}_1^{1\times n_1}), \ldots, \mathsf{size}(\mathsf{T}_l^{1\times n_l}))$.
   - Send $(\textsc{JoinReq}, sid, jqid, \ell, \mathcal{P})$ to $\mathcal{A}$ and wait for $(\textsc{JoinReq}, sid, jqid, \mathtt{ok})$ from $\mathcal{A}$.
   - Create a record $(\mathtt{joinreq}, sid, jqid, (\mathsf{T}_1^{1\times n_1}, \ldots \mathsf{T}_l^{1\times n_l}), \mathcal{P})$ and output $(\textsc{JoinReq}, sid, jqid, (tid_1, \ldots, tid_l), \mathcal{P})$ to $\mathcal{C}$.
4. Join Response. On input of $(\textsc{Join}, sid, jqid)$ from the converter $\mathcal{C}$:
   - Proceed only if $(\mathtt{joinreq}, sid, jqid, (\mathsf{T}_1^{1\times n_1}, \ldots \mathsf{T}_l^{1\times n_l}), \mathcal{P})$ for $jqid$ exists. Set $\mathbf{Join\text{-}IDs} \leftarrow \emptyset$.
   - Send $(\textsc{Join}, sid, jqid)$ to $\mathcal{A}$ and wait for $(\textsc{Join}, sid, jqid, \mathtt{ok})$.
   - For $j = 1, \ldots, l$: Get $(uid_1, \ldots, uid_{n_j}) \leftarrow \mathsf{KEY}(\mathsf{T}_j^{1\times n_j})$ and create a copy $\widetilde{\mathsf{T}}_j^{1\times n_j} \leftarrow \mathsf{T}_j^{1\times n_j}$ as follows
     - For $i = 1, \ldots, n_j$: Get $join\text{-}id_i \leftarrow \mathbf{Join\text{-}IDs}(uid_i)$. If $join\text{-}id_i = \perp$, choose $join\text{-}id_i \xleftarrow{\$} \mathbf{N}$ and set $\mathbf{Join\text{-}IDs}(uid_i) \leftarrow join\text{-}id_i$.
     - Set $\mathsf{KEY}(\widetilde{\mathsf{T}}_j^{1\times n_j}) \leftarrow (join\text{-}id_1, \ldots, join\text{-}id_{n_j})$ and sort $\widetilde{\mathsf{T}}_j^{1\times n_j}$ by the primary key.
   - Output $(\textsc{Joined}, sid, jqid, (\widetilde{\mathsf{T}}_1^{1\times n_1}, \ldots, \widetilde{\mathsf{T}}_l^{1\times n_l}))$ to $\mathcal{P}$ and delete **Join-IDs**.

Fig. 3. Ideal functionality $\mathcal{F}_{\text{ScrambleDB}}$ with $sid = (sid', \mathcal{C}, \mathcal{L}, \mathbf{N})$ parametrized with a leakage function leak and with $sid = (sid', \mathcal{C}, \mathcal{L}, \mathbf{N})$ where $\mathbf{N}$ denotes the pseudonym space. The functionality can be called by multiple data sources $\mathcal{S}$ and processors $\mathcal{P}$.

which all unlinkable pseudonyms are *consistently* converted into an ephemeral and non-transitive pseudonym representation to which we refer to as *join-id*.

Informally, such an oblivious chameleon-pseudonymization-service should satisfy the following:

1) Oblivious Generation & Conversion: The converter $\mathcal{C}$ neither learns the incoming identifiers uid_i, nor the blindly computed pseudonyms $nym_{i,j}$. In fact, the converter $\mathcal{C}$ cannot even tell whether two requests are for the same identifier or not. Furthermore, the data source $\mathcal{S}$ does not learn anything about the triggered pseudonyms, nor does the lake $\mathcal{L}$ learn anything about the identifiers behind the pseudonyms. Likewise $\mathcal{C}$ must also perform the conversion for joins in a blind manner and $\mathcal{L}$ learns nothing about the underlying identifiers or the triggered *join-ids*.

2) Pseudorandom & Unlinkable Pseudonyms: For each user uid_i and attribute type j, a random-looking pseudonym $nym_{i,j}$ is generated. In particular, when given two pseudonyms $nym_{i,j}$ and $nym_{i',j'}$ for two different attribute-types j and j', one cannot tell if $i = i'$, i.e., whether both pseudonyms belong to the same user or not. This property also covers *collusion-resistance*, i.e., even if some data sources and the data lake collude they cannot re-identify the users behind the pseudonyms or correlate their attribute-specific pseudonyms.

3) Controlled Join & Consistency: The only way to link and correlate the chameleon pseudonyms is via requests to the converter. When approved by $\mathcal{C}$, the pseudonyms of the required subsets are consistently converted into ephemeral *join-ids*. That is, if $nym_{i,j}$ and $nym_{i,j'}$ are based on the same unique identifier uid_i, they both will be converted into the same join pseudonym $join\text{-}id_i$.

4) Non-Transitivity of Joins: Two individually joined data sets cannot be correlated any further (via the *join-ids*). That is, the consistent join conversion is strictly non-transitive and prevents corrupt data

processors $\mathcal{P}_i$ to correlate their data beyond what was explicitly approved by the converter.

4.2 Security Model for ScrambleDB

We now formally define the security and functionality of such an oblivious chameleon pseudonymization service by describing an ideal functionality $\mathcal{F}_{\text{ScrambleDB}}$ in the Universal Composability (UC) framework [12].

The UC Framework & Writing Conventions. The ideal functionality works in a way that is secure-by-design. A real-world protocol is said to securely realize a certain functionality $\mathcal{F}$ if an environment cannot distinguish whether it is interacting with the real protocol or with $\mathcal{F}$ and a simulator.

The UC framework allows us to focus our analysis on a single protocol instance with a globally unique session identifier *sid*. Here we use session identifiers of the form $sid = (sid', \mathcal{C}, \mathcal{L}, \mathbf{N})$, i.e., it fixes the identity of the converter $\mathcal{C}$ and the data lake $\mathcal{L}$. Further, sid' is a unique string, and $\mathbf{N}$ denotes the pseudonym space. We also use unique query identifiers *pqid* and *jqid* to distinguish between several pseudonym generation and conversion sessions. We implicitly assume that the functionality checks that all session and query identifiers are well-formed and unique.

Our functionality keeps the following state: The list **Nyms** contains tuples of the form $(uid_i, attr_j, nym_{i,j})$ denoting that $nym_{i,j}$ is the attribute-specific pseudonym for unique id uid_i and $attr_j$. We use the shorthand $nym_{i,j} \leftarrow \mathbf{Nyms}(uid_i, attr_j)$ to denote retrieving the pseudonym $nym_{i,j}$ from the record for uid_i and $attr_j$. The set **Tables** contains all stored tables, which can be selectively retrieved using the table identifier *tid* which we denote as $\mathsf{T}^{m\times n} \leftarrow \mathbf{Tables}(tid)$.

The $\mathcal{F}_{\text{ScrambleDB}}$ Functionality. The definition of our ideal functionality $\mathcal{F}_{\text{ScrambleDB}}$ is given in Figure 3. It has four interfaces, two for the pseudonymization, and two for selective joins. The *pseudonymization request* interface can be invoked by any data source $\mathcal{S}$ that wishes to upload a pseudonymized version of the table $\mathsf{T}^{m\times n}$ to the data lake $\mathcal{L}$. When $\mathcal{C}$ approves the request via an input to the *pseudonymized data output* interface, then $\mathcal{F}_{\text{ScrambleDB}}$ produces *unlinkable* pseudonyms and outputs m shuffled and individually pseudonymized tables $(\mathsf{T}_1^{1\times n}, \dots, \mathsf{T}_m^{1\times n})$ to $\mathcal{L}$.

The *join request* interface is invoked by the data lake $\mathcal{L}$ that wishes to produce a joined (or rather joinable) version of the unlinkable data tables identified by $(tid_1, \dots, tid_l)$ to a certain data processor $\mathcal{P}$. When the converter gives explicit approval via the *join response* interface, $\mathcal{F}_{\text{ScrambleDB}}$ outputs $(\widetilde{\mathsf{T}}_1^{1\times n_1}, \dots, \widetilde{\mathsf{T}}_l^{1\times n_l})$ to $\mathcal{P}$ which are the requested tables that have consistently converted pseudonyms as primary keys. The consistent conversion is done with the help of **Tables**, which contains all the tables of the data lake in fully identifying form and is internally maintained by $\mathcal{F}_{\text{ScrambleDB}}$.

We now discuss how $\mathcal{F}_{\text{ScrambleDB}}$ guarantees the expected security properties described before.

1) Obliviousness: As long as not both the converter and the data lake are corrupt, all the adversary learns for pseudonym generation is the identity of the data source and the size of the table $\mathsf{T}^{m\times n}$ via the leakage ℓ. Likewise, when data is joined, the adversary again only learns the size of the tables $\ell \leftarrow (\mathsf{size}(\mathsf{T}_1^{1\times n_1}), \dots, \mathsf{size}(\mathsf{T}_l^{1\times n_l}))$ for which the join was requested. The leakage function size will reveal the number of rows and columns of a table, and possibly the lengths of the attributes. In our protocol, the latter will be the "leakage" of the underlying encryption scheme that will be used to encrypt the attributes, which usually reveals the (block) length of a message. For simplicity however, we assume that all attributes are bounded by the block size and omit an explicit treatment of their length.
Only when *both* the converter and the receiver of the pseudonyms, i.e., $\mathcal{L}$ for generation and $\mathcal{P}$ for conversion, are corrupt, the adversary learns some leakage of the underlying identifiers as $\mathsf{leak}(uid_i)$. As discussed, this leakage is unavoidable in a setting where $\mathcal{C}$ is handling all the essential keys and colludes with the receiver of its deterministic computations. Our protocol realizes $\mathcal{F}_{\text{ScrambleDB}}$ for leak being a one-way function.

2) Pseudorandomness & Unlinkability: For each user uid_i and attribute type $attr_j$, the functionality assigns a random value $nym_{i,j} \xleftarrow{\$} \mathbf{N}$. This naturally enforces unlinkability of pseudonyms for the same user across different attributes. $\mathcal{F}_{\text{ScrambleDB}}$ reuses the same pseudonym only when multiple tables share the same attributes. Thus, for the best privacy guarantees, all attributes types in the databases should be globally unique. A trivial way to achieve this is to simply prepend the table identifier *tid* to every attribute type. The functionality enforces collusion-resistance by shuffling the table before outputting the pseudonymized version to $\mathcal{L}$ such that no linkage is revealed through the order of rows. The shuffle is done by sorting the table along the primary keys which are random values.

3) Controlled Join & Consistency: As $\mathcal{F}_{\text{ScrambleDB}}$ outputs random values as pseudonyms $nym_{i,j}$, the only way to learn which pseudonyms are correlated is to send a join request to the functionality. When approved by $\mathcal{C}$, $\mathcal{F}_{\text{ScrambleDB}}$ converts these pseudonyms using the internal knowledge of **Tables** which contains all data in identifying form. Thus, the functionality retrieves all requested tables $(\mathsf{T}_1^{1\times n_1}, \ldots \mathsf{T}_l^{1\times n_l})$ from **Tables** with the uid_is as primary keys. It then replaces each occurrence of uid_i with the same random pseudonym $join\text{-}id_i \xleftarrow{\$} \mathbf{N}$, which naturally enforces the desired consistency of joins.

4) Non-Transitivity: The consistent *join-id*s that are used by the functionality are kept in an ephemeral state **Join-IDs**, that is reset to $\emptyset$ for every new request. Thus, consistency is strictly limited to the tables within a request. As the *join-id*s are random values in the pseudonym space $\mathbf{N}$, it follows that *join-id*s for two different requests cannot be linked. This property also holds if $\mathcal{L}$ and $\mathcal{P}$ are colluding: the functionality additionally shuffles the tables $(\widetilde{\mathsf{T}}_1^{1\times n_1}, \ldots, \widetilde{\mathsf{T}}_l^{1\times n_l})$ before outputting them to $\mathcal{P}$ which prevents linkage through the order of the rows. Also, the random generation of the *join-id*s received by $\mathcal{P}$ and the pseudonyms $nym_{i,j}$ known to $\mathcal{L}$, assures that there is no linkage between the pseudonyms and *join-id*s.

4.3 The ScrambleDB Protocol

We now present our protocol that securely instantiates $\mathcal{F}_{\text{ScrambleDB}}$. Our ScrambleDB protocol derives the core of the unlinkable pseudonyms via a coPRF, using a different secret key for every attribute-type, and using fresh and ephemeral keys when joining tables. The pseudonyms get hardened by letting the data lake and data processors apply a local and standard PRF (or PRP) on the coPRF output. Further, the associated data gets encrypted using a re-randomizable encryption scheme RE. Finally, we assume that functionalities $\mathcal{F}_{\mathsf{CA}}$ and $\mathcal{F}_{\mathsf{SMT}}$ are available to all parties and that parties call $\mathcal{F}_{\mathsf{CA}}$ to retrieve the necessary key material. The detailed protocol for setup and pseudonym generation is given in Figure 4 and Figure 5 presents the protocol for converting pseudonyms in a join request.

Pseudonymization Request & Output. In the pseudonymization request for a table $\mathsf{T}^{m\times n}$, the data source generates blinded PRF requests $\overline{x}_i$ for all primary keys uid_i in the table. Further, each data field in the table is encrypted with a re-randomizable encryption scheme under the data lake's public key.

When $\mathcal{C}$ gets approval to proceed, it breaks the table $\mathsf{T}^{m\times n}$ into m tables $\mathsf{T}_j^{1\times n_j}$, and assigns each row to the blinded attribute-specific pseudonym $\overline{y}_{i,j}$ for $\overline{x}_i$ under the coPRF key for $attr_j$. Thus, here we use the batch evaluation capability of coPRFs and derive multiple PRF outputs from the same blinded input $\overline{x}_i$. To avoid linkage through the data structure, all encrypted data fields get re-randomized and the individual tables get re-shuffled by $\mathcal{C}$ before they are forwarded to $\mathcal{L}$.

Finally, $\mathcal{L}$ unblinds and decrypts the information it receives from $\mathcal{C}$, and finalizes the pseudonyms by applying a conventional PRP. The latter ensures that a corrupt $\mathcal{C}$ cannot "de-anonymize" or brute-force the derived pseudonyms as long as $\mathcal{L}$ honest.

Join Request & Output. When a certain subset of these tables should be joined towards a processor $\mathcal{P}$, the data lake sends a blinded conversion request to $\mathcal{C}$ and with the data being encrypted under $\mathcal{P}$'s public key.

If the converter approves the join, it blindly converts the pseudonyms towards a random, ephemeral key k^*. Again, to avoid linkage through the structure of the tables, all encrypted data gets re-randomized and shuffled. The data processor finalizes the *join-id*s by unblinding the coPRF output, applying an additional PRF and decrypting the tables. The additional PRF ensures that a corrupt converter $\mathcal{C}$, when seeing the transformed pseudonyms, cannot de-anonymize the underlying users or further join the pseudonyms as long as $\mathcal{P}$ is honest.

Security of ScrambleDB. We have already informally argued how the building blocks contribute to the security of the full protocol. More formally, ScrambleDB achieves the following security against passive adversaries:

Theorem 4.1 (Passive Security of ScrambleDB)**.** *The* ScrambleDB *protocol securely realizes* $\mathcal{F}_{\text{ScrambleDB}}$ *for* leak $=$ $\mathsf{leak}_{\mathsf{coPRF}}$ *against passive adversaries in the* $\mathcal{F}_{\mathsf{CA}}, \mathcal{F}_{\mathsf{SMT}}$*-hybrid model if* coPRF *is a pseudorandom, collusion-resistant, fully oblivious and input-hiding convertible PRF,* RE *is a CPA-secure, re-randomizable encryption scheme,* PRF *is a secure pseudorandom function, and* PRP *is secure pseudorandom permutation.*

To prove that our protocol securely realizes the ideal functionality $\mathcal{F}_{\text{ScrambleDB}}$, we have to show that for any environment $\mathcal{E}$ and any adversary $\mathcal{A}$, there exist a simulator SIM such that $\mathcal{E}$ cannot distinguish whether it's interacting with $\mathcal{A}$ and the protocol in the real world or with SIM and $\mathcal{F}_{\text{ScrambleDB}}$. We refer to Appendix C for the description of this simulator.

Setup. For a session with $sid = (sid', \mathcal{C}, \mathcal{L}, \mathbb{G})$ the parties generate their key material as follows, where coPRF is a convertible PRF (as defined in Sec. 3), RE is a re-randomizable encryption scheme, and PRF, PRP are a standard PRF and PRP.

Converter $\mathcal{C}$: generates $msk \xleftarrow{\$} \mathsf{coPRF.Setup}(\tau)$, and stores (sid, msk).

Data lake $\mathcal{L}$: generates $(bpk_{\mathcal{L}}, bsk_{\mathcal{L}}) \xleftarrow{\$} \mathsf{coPRF.BKGen}(\tau)$, $(epk_{\mathcal{L}}, esk_{\mathcal{L}}) \xleftarrow{\$} \mathsf{RE.KGen}(\tau)$, $k_{\mathcal{L}} \xleftarrow{\$} \mathsf{PRP.KGen}(\tau)$, and $\mathbf{Tables} \leftarrow \emptyset$. It stores $(sid, bsk_{\mathcal{L}}, esk_{\mathcal{L}}, k_{\mathcal{L}})$ and makes $(sid, bpk_{\mathcal{L}}, epk_{\mathcal{L}})$ available via $\mathcal{F}_{\mathsf{CA}}$.

Data processor $\mathcal{P}$: each data processor generates $(bpk_{\mathcal{P}}, bsk_{\mathcal{P}}) \xleftarrow{\$} \mathsf{coPRF.BKGen}(\tau)$, $(epk_{\mathcal{P}}, esk_{\mathcal{P}}) \xleftarrow{\$} \mathsf{RE.KGen}(\tau)$, and $k_{\mathcal{P}} \xleftarrow{\$} \mathsf{PRF.KGen}(\tau)$. It stores $(sid, bsk_{\mathcal{P}}, esk_{\mathcal{P}}, k_{\mathcal{P}})$ and makes $(sid, bpk_{\mathcal{P}}, epk_{\mathcal{P}})$ available via $\mathcal{F}_{\mathsf{CA}}$.

Pseudonymization Request. The data source $\mathcal{S}$ requests pseudonymization of the table $\mathsf{T}^{m\times n}$ via the converter $\mathcal{C}$.

1.a Data source $\mathcal{S}$, on input $(\textsc{NymReq}, sid, pqid, \mathsf{T}^{m\times n})$:
- Compute $\overline{\mathsf{T}}^{m\times n}$, a blinded and encrypted version of $\mathsf{T}^{m\times n}$ with $\mathsf{ID}(\overline{\mathsf{T}}^{m\times n}) \leftarrow \mathsf{ID}(\mathsf{T}^{m\times n})$, $\mathsf{ATTR}(\overline{\mathsf{T}}^{m\times n}) \leftarrow \mathsf{ATTR}(\mathsf{T}^{m\times n})$:
  - Get $(uid_1, \ldots uid_n) \leftarrow \mathsf{KEY}(\mathsf{T}^{m\times n})$
  - For $i = 1, \ldots, n$:
    - Compute $\overline{x}_i \xleftarrow{\$} \mathsf{coPRF.Eval.Blind}(bpk_{\mathcal{L}}, uid_i)$, and set $\overline{\mathsf{T}}^{m\times n}[i,j] \xleftarrow{\$} \mathsf{RE.Enc}(epk_{\mathcal{L}}, \mathsf{T}^{m\times n}[i,j])$ for $j = 1, \ldots, m$
  - Set $\mathsf{KEY}(\overline{\mathsf{T}}^{m\times n}) \leftarrow (\overline{x}_1, \ldots \overline{x}_n)$, and sort $\overline{\mathsf{T}}^{m\times n}$ by the primary key (this implements a random shuffle)
- Send $(\textsc{NymReq}, sid, pqid, \overline{\mathsf{T}}^{m\times n})$ to $\mathcal{C}$ (taken from sid) via $\mathcal{F}_{\mathsf{SMT}}$

1.b Converter $\mathcal{C}$, upon receiving $(\textsc{NymReq}, sid, pqid, \overline{\mathsf{T}}^{m\times n})$ from $\mathcal{S}$ via $\mathcal{F}_{\mathsf{SMT}}$:
- Store $(sid, pqid, \overline{\mathsf{T}}^{m\times n}, \mathcal{S})$ and output $(\textsc{NymReq}, sid, pqid, \mathcal{S}, \mathsf{ATTR}(\overline{\mathsf{T}}^{m\times n}), n)$

Pseudonymization Output. The converter $\mathcal{C}$ and data lake $\mathcal{L}$ jointly produce the pseudonymized and unlinkable output.

2.a Converter $\mathcal{C}$, on input $(\textsc{NymOut}, sid, pqid)$:
- Retrieve $(sid, pqid, \overline{\mathsf{T}}^{m\times n}, \mathcal{S})$ for $pqid$, get $attr_1, \ldots, attr_m \leftarrow \mathsf{ATTR}(\overline{\mathsf{T}}^{m\times n})$ and $(\overline{x}_1, \ldots \overline{x}_n) \leftarrow \mathsf{KEY}(\overline{\mathsf{T}}^{m\times n})$
- For $j = 1, \ldots, m$, create the table $\mathsf{T}_j^{1\times n}$ using the j-th column of $\overline{\mathsf{T}}^{m\times n}$ as follows:
  - Get $k_j \leftarrow \mathsf{coPRF.KGen}(msk, attr_j)$
  - For $i = 1, \ldots, n$:
    - Get $\overline{y}_{i,j} \xleftarrow{\$} \mathsf{coPRF.Eval.Exec}(k_j, bpk_{\mathcal{L}}, \overline{x}_i)$ and set $\mathsf{T}_j^{1\times n}[i] \xleftarrow{\$} \mathsf{RE.ReRand}(epk_{\mathcal{L}}, \overline{\mathsf{T}}^{m\times n}[i,j])$
  - Set $\mathsf{KEY}(\mathsf{T}_j^{1\times n}) \leftarrow (\overline{y}_{1,j}, \ldots \overline{y}_{n,j})$, $\mathsf{ID}(\mathsf{T}_j^{1\times n}) \leftarrow (\mathsf{ID}(\overline{\mathsf{T}}^{m\times n}), attr_j)$ and sort $\mathsf{T}_j^{1\times n}$ by the primary key
- Send $(\textsc{NymOut}, sid, pqid, \mathcal{S}, \mathsf{T}_1^{1\times n}, \ldots, \mathsf{T}_m^{1\times n})$ to $\mathcal{L}$ via $\mathcal{F}_{\mathsf{SMT}}$

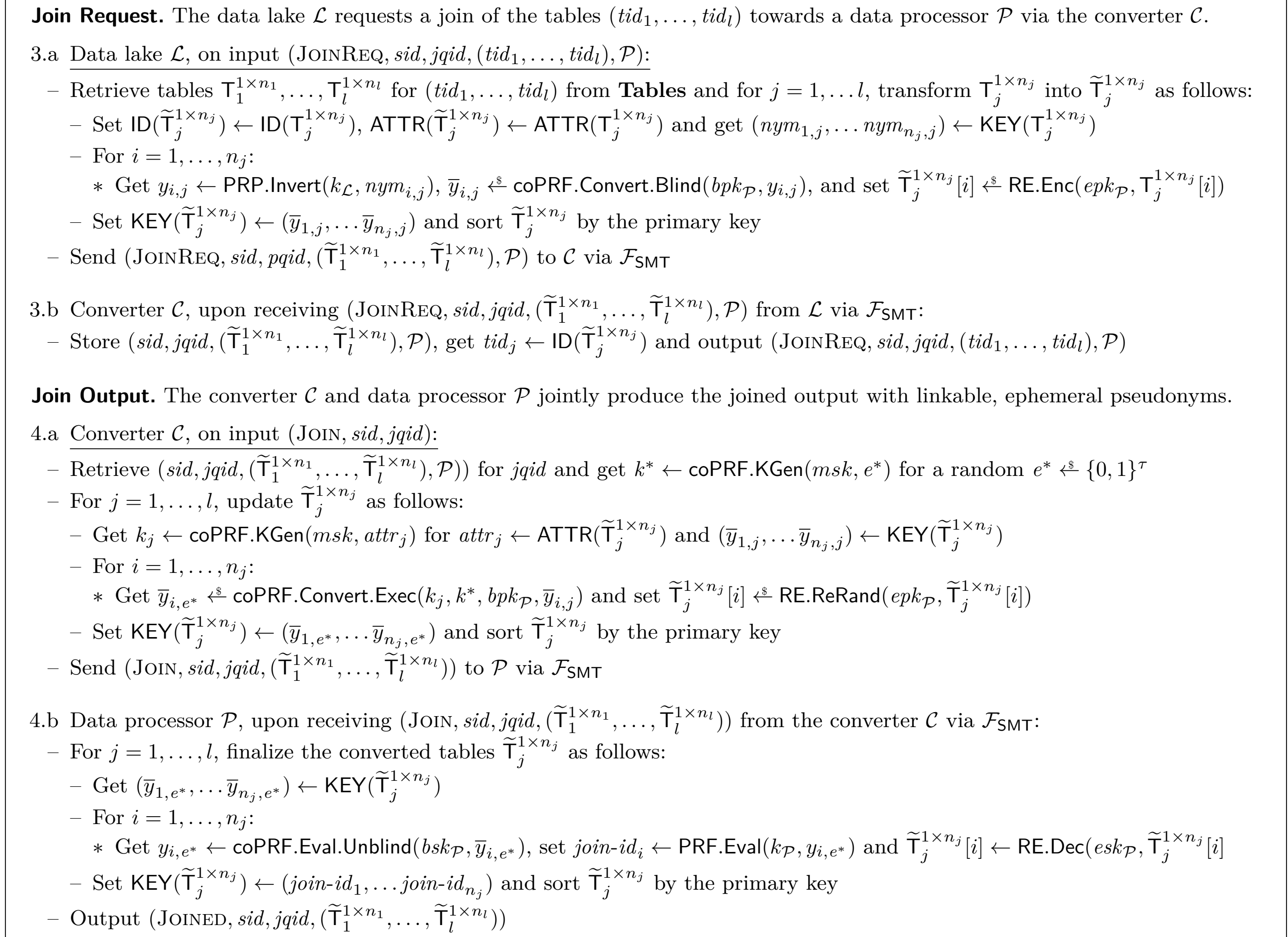
2.b Data lake $\mathcal{L}$, upon receiving $(\textsc{NymOut}, sid, pqid, \mathcal{S}, \mathsf{T}_1^{1\times n}, \ldots, \mathsf{T}_m^{1\times n})$ from the converter $\mathcal{C}$ via $\mathcal{F}_{\mathsf{SMT}}$:
- For $j = 1, \ldots, m$, finalize the unlinkable tables $\mathsf{T}_j^{1\times n}$ as follows:
  - Get $(\overline{y}_{1,j}, \ldots \overline{y}_{n,j}) \leftarrow \mathsf{KEY}(\mathsf{T}_j^{1\times n})$
  - For $i = 1, \ldots, n$:
    - Get $y_{i,j} \leftarrow \mathsf{coPRF.Eval.Unblind}(bsk_{\mathcal{L}}, \overline{y}_{i,j})$, set $nym_{i,j} \leftarrow \mathsf{PRP.Eval}(k_{\mathcal{L}}, y_{i,j})$ and $\mathsf{T}_j^{1\times n}[i] \leftarrow \mathsf{RE.Dec}(esk_{\mathcal{L}}, \mathsf{T}_j^{1\times n}[i])$
  - Set $\mathsf{KEY}(\mathsf{T}_j^{1\times n}) \leftarrow (nym_{1,j}, \ldots nym_{n,j})$ and sort $\mathsf{T}_j^{1\times n}$ by the primary key
- Add $\mathsf{T}_1^{1\times n}, \ldots, \mathsf{T}_m^{1\times n}$ to $\mathbf{Tables}$ and output $(\textsc{NymOut}, sid, pqid, \mathcal{S}, \mathsf{T}_1^{1\times n}, \ldots, \mathsf{T}_m^{1\times n})$

Fig. 4. Setup, pseudonymization request (1.a-b) and unlinkable data output (2.a-b) parts of our ScrambleDB protocol.

Security against active attacks. Similar to coPRFs, our main security guarantees hold against passive adversaries. This is a common trade-off in privacy-preserving protocols in order to avoid heavy machinery where all parties have to prove that their inputs are well-formed. We stress that the pseudonym service still preserves the crucial security properties under active attacks, meaning that malicious parties cannot learn more pseudonyms or more correlations than they make queries to the converter $\mathcal{C}$. This follows immediately from the fact that the converter $\mathcal{C}$ only executes the coPRF evaluations. Thus, if the coPRF has such one-more unpredictability guarantees under active attacks, then this property is inherited by the ScrambleDB scheme as well. For space reasons we omit the formal definition and proof which closely resemble the one-more unpredictability notion of coPRFs introduced in Section 3.1.

Lemma 4.1 (Active Security of ScrambleDB). *If* coPRF *is one-more unpredictable, then* ScrambleDB *is one-more unpredictable against actively corrupt* $\mathcal{S}, \mathcal{L}, \mathcal{P}$.

The Need for coPRFs. We have identified convertible 3-party OPRFs as the crucial building block for our ScrambleDB solution, which might provoke the question why conventional 2-party OPRFs are not sufficient. First, we need the conversion property to realize the joins in an *oblivious* and *non-transitive* fashion. Second, the 3-party setting is crucial to avoid re-identification attacks by a colluding data source and data lake. Us-



Join Request. The data lake $\mathcal{L}$ requests a join of the tables $(tid_1, \dots, tid_l)$ towards a data processor $\mathcal{P}$ via the converter $\mathcal{C}$.

3.a Data lake $\mathcal{L}$, on input $(\text{JOINREQ}, sid, jqid, (tid_1, \dots, tid_l), \mathcal{P})$:

- Retrieve tables $\mathsf{T}_1^{1\times n_1}, \dots, \mathsf{T}_l^{1\times n_l}$ for $(tid_1, \dots, tid_l)$ from **Tables** and for $j = 1, \dots l$, transform $\mathsf{T}_j^{1\times n_j}$ into $\widetilde{\mathsf{T}}_j^{1\times n_j}$ as follows:
  - Set $\mathsf{ID}(\widetilde{\mathsf{T}}_j^{1\times n_j}) \leftarrow \mathsf{ID}(\mathsf{T}_j^{1\times n_j})$, $\mathsf{ATTR}(\widetilde{\mathsf{T}}_j^{1\times n_j}) \leftarrow \mathsf{ATTR}(\mathsf{T}_j^{1\times n_j})$ and get $(nym_{1,j}, \dots nym_{n_j,j}) \leftarrow \mathsf{KEY}(\mathsf{T}_j^{1\times n_j})$
  - For $i = 1, \dots, n_j$:
    * Get $y_{i,j} \leftarrow \mathsf{PRP.Invert}(k_{\mathcal{L}}, nym_{i,j})$, $\overline{y}_{i,j} \xleftarrow{\$} \mathsf{coPRF.Convert.Blind}(bpk_{\mathcal{P}}, y_{i,j})$, and set $\widetilde{\mathsf{T}}_j^{1\times n_j}[i] \xleftarrow{\$} \mathsf{RE.Enc}(epk_{\mathcal{P}}, \mathsf{T}_j^{1\times n_j}[i])$
  - Set $\mathsf{KEY}(\widetilde{\mathsf{T}}_j^{1\times n_j}) \leftarrow (\overline{y}_{1,j}, \dots \overline{y}_{n_j,j})$ and sort $\widetilde{\mathsf{T}}_j^{1\times n_j}$ by the primary key
- Send $(\text{JOINREQ}, sid, pqid, (\widetilde{\mathsf{T}}_1^{1\times n_1}, \dots, \widetilde{\mathsf{T}}_l^{1\times n_l}), \mathcal{P})$ to $\mathcal{C}$ via $\mathcal{F}_{\mathsf{SMT}}$

3.b Converter $\mathcal{C}$, upon receiving $(\text{JOINREQ}, sid, jqid, (\widetilde{\mathsf{T}}_1^{1\times n_1}, \dots, \widetilde{\mathsf{T}}_l^{1\times n_l}), \mathcal{P})$ from $\mathcal{L}$ via $\mathcal{F}_{\mathsf{SMT}}$:

- Store $(sid, jqid, (\widetilde{\mathsf{T}}_1^{1\times n_1}, \dots, \widetilde{\mathsf{T}}_l^{1\times n_l}), \mathcal{P})$, get $tid_j \leftarrow \mathsf{ID}(\widetilde{\mathsf{T}}_j^{1\times n_j})$ and output $(\text{JOINREQ}, sid, jqid, (tid_1, \dots, tid_l), \mathcal{P})$

Join Output. The converter $\mathcal{C}$ and data processor $\mathcal{P}$ jointly produce the joined output with linkable, ephemeral pseudonyms.

4.a Converter $\mathcal{C}$, on input $(\text{JOIN}, sid, jqid)$:

- Retrieve $(sid, jqid, (\widetilde{\mathsf{T}}_1^{1\times n_1}, \dots, \widetilde{\mathsf{T}}_l^{1\times n_l}), \mathcal{P}))$ for $jqid$ and get $k^* \leftarrow \mathsf{coPRF.KGen}(msk, e^*)$ for a random $e^* \xleftarrow{\$} \{0,1\}^\tau$
- For $j = 1, \dots, l$, update $\widetilde{\mathsf{T}}_j^{1\times n_j}$ as follows:
  - Get $k_j \leftarrow \mathsf{coPRF.KGen}(msk, attr_j)$ for $attr_j \leftarrow \mathsf{ATTR}(\widetilde{\mathsf{T}}_j^{1\times n_j})$ and $(\overline{y}_{1,j}, \dots \overline{y}_{n_j,j}) \leftarrow \mathsf{KEY}(\widetilde{\mathsf{T}}_j^{1\times n_j})$
  - For $i = 1, \dots, n_j$:
    * Get $\overline{y}_{i,e^*} \xleftarrow{\$} \mathsf{coPRF.Convert.Exec}(k_j, k^*, bpk_{\mathcal{P}}, \overline{y}_{i,j})$ and set $\widetilde{\mathsf{T}}_j^{1\times n_j}[i] \xleftarrow{\$} \mathsf{RE.ReRand}(epk_{\mathcal{P}}, \widetilde{\mathsf{T}}_j^{1\times n_j}[i])$
  - Set $\mathsf{KEY}(\widetilde{\mathsf{T}}_j^{1\times n_j}) \leftarrow (\overline{y}_{1,e^*}, \dots \overline{y}_{n_j,e^*})$ and sort $\widetilde{\mathsf{T}}_j^{1\times n_j}$ by the primary key
- Send $(\text{JOIN}, sid, jqid, (\widetilde{\mathsf{T}}_1^{1\times n_1}, \dots, \widetilde{\mathsf{T}}_l^{1\times n_l}))$ to $\mathcal{P}$ via $\mathcal{F}_{\mathsf{SMT}}$

4.b Data processor $\mathcal{P}$, upon receiving $(\text{JOIN}, sid, jqid, (\widetilde{\mathsf{T}}_1^{1\times n_1}, \dots, \widetilde{\mathsf{T}}_l^{1\times n_l}))$ from the converter $\mathcal{C}$ via $\mathcal{F}_{\mathsf{SMT}}$:

- For $j = 1, \dots, l$, finalize the converted tables $\widetilde{\mathsf{T}}_j^{1\times n_j}$ as follows:
  - Get $(\overline{y}_{1,e^*}, \dots \overline{y}_{n_j,e^*}) \leftarrow \mathsf{KEY}(\widetilde{\mathsf{T}}_j^{1\times n_j})$
  - For $i = 1, \dots, n_j$:
    * Get $y_{i,e^*} \leftarrow \mathsf{coPRF.Eval.Unblind}(bsk_{\mathcal{P}}, \overline{y}_{i,e^*})$, set $join\text{-}id_i \leftarrow \mathsf{PRF.Eval}(k_{\mathcal{P}}, y_{i,e^*})$ and $\widetilde{\mathsf{T}}_j^{1\times n_j}[i] \leftarrow \mathsf{RE.Dec}(esk_{\mathcal{P}}, \widetilde{\mathsf{T}}_j^{1\times n_j}[i]$
  - Set $\mathsf{KEY}(\widetilde{\mathsf{T}}_j^{1\times n_j}) \leftarrow (join\text{-}id_1, \dots join\text{-}id_{n_j})$ and sort $\widetilde{\mathsf{T}}_j^{1\times n_j}$ by the primary key
- Output $(\text{JOINED}, sid, jqid, (\widetilde{\mathsf{T}}_1^{1\times n_1}, \dots, \widetilde{\mathsf{T}}_l^{1\times n_l}))$

Fig. 5. Join request (3.a-b) and joined data output (4.a-b) parts of our ScrambleDB protocol.

ing a standard OPRF to derive pseudonyms, the data source would learn the mapping between every primary-identifier and their pseudonyms. This is exactly what we want to avoid, as it can be used by a colluding source and lake to de-anonymize data from other sources. In fact, even if a pseudonymization request is made for a table where all attributes are equivalent, i.e., they do not reveal any information about the originating user, an OPRF would still allow a corrupt $\mathcal{S}$ and $\mathcal{L}$ to know which pseudonymized attribute in the lake belongs to which user. By joining such seemingly "harmless" data with attributes from honest sources, the adversary can de-anonymize the joined information.

Clearly, a malicious data source could always do such an attack by uploading a dataset that contains identifying attributes, and later join this data with information from other honest sources towards a corrupt processor. However, proper audit procedures could recognize and prevent such attempts, whereas a 2-party OPRF would introduce and require unique and traceable identifiers on a low protocol level. Our 3-party OPRF avoids that and ensures that pseudonymization does not *introduce* any handles or subliminal channels that can be exploited for re-identification attacks.

Efficiency. When using our $\mathsf{coPRF}_{\mathsf{DDH}}$ from Section 3.3 and the ElGamal-based scheme from Section 2 for the re-randomizable encryption RE, we obtain the following efficiency figures measured by the number of exponentiations as the most expensive operation:

	$\mathcal{S}$ ($\mathcal{P}$ in Join)	$\mathcal{C}$	$\mathcal{L}$
Pseudonymization of $\mathsf{T}^{m\times n}$	$(m+1)n \cdot 2\mathsf{ex}$	$mn \cdot 5\mathsf{ex}$	$mn \cdot 2\mathsf{ex}$
Join of $\mathsf{T}_1^{1\times n_1}, \dots \mathsf{T}_l^{1\times n_l}$	$\sum_{j=1}^{l}(n_j \cdot 2\mathsf{ex})$	$\sum_{j=1}^{l}(n_j \cdot 5\mathsf{ex})$	$\sum_{j=1}^{l}(n_j \cdot 4\mathsf{ex})$

On efficient curves such as Curve25519 or gls254prot, an exponentiation takes around 0.045ms and 0.013ms respectively on an Intel Xeon E3-1220, 4x3GHz CPU (using benchmarks provided on bench.cr.yp.to). Thus, the conversion of a table with 1000 rows and 10 attribute columns into pseudonymized, unlinkable data snippets will approximately take 286ms for $\mathcal{S}$, 650ms for $\mathcal{C}$ and 260ms for $\mathcal{L}$.

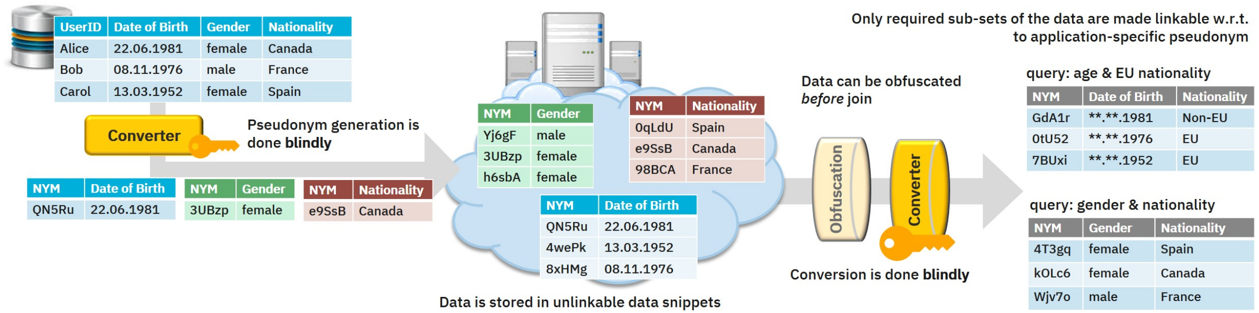


Fig. 6. ScrambleDB : Unlinkable pseudonymization and controlled join of selected data.

4.4 Privacy Limitations & Improvements

We stress that the privacy guarantees such as unlinkability and non-transitivity hold for the chameleon pseudonyms and the process to compute and convert such pseudonyms. Our model and scheme do not – and cannot – exclude that additional linkage or inference is possible via the associated attribute values. In particular when data gets joined towards a data processor, the risk of re-identification increases. Thus, to preserve as much privacy as possible, our chameleon pseudonyms should ideally be combined with additional data obfuscation. We sketch some ideas for such a combination and consider a more detailed exposition and analysis an interesting direction for future work.

Obfuscation by the data lake. When the data lake $\mathcal{L}$ receives a request to join certain tables towards a data processor, it should first analyze the re-identification risk that would occur when bringing these attributes together and then apply appropriate generalization techniques (see Figure 6). For instance, when joining sensitive attribute types, such as date of birth and residence that in combination could de-anonymize users, the data lake should generalize the attributes, e.g., reducing the full date of birth to the year or an age range only, and using more generalized location information instead of concrete addresses. The data lake can also omit rare attribute values in each table, enabling a mild form of k-anonymity, or add controlled noise.

It is crucial that these measures are applied to the individual tables *before* they are send to the converter: this way at no point during the join process a combined version of the sensitive, high-precision data is created.

Dedicated sanitizer. An interesting extension of our work is to introduce a dedicated entity that further obfuscates the *joined* information (in a privacy-preserving way) before it reaches the actual data processor. For instance, to realize k-anonymity, the sanitizer would be the receiver of the converted information (i.e., it would take the role of the data processor in our ScrambleDB protocol), but all attribute values will have an additional inner layer of *deterministic* encryption. This inner encryption is applied on the tables' content by the data lake before running the join protocol and using a dedicated symmetric key for each data table that will be securely communicated to the actual data processor. The sanitizer receives joined but encrypted information, where the determinism of the encryption allows him to recognize and omit all rows with rare attribute combinations before forwarding it to the processor.

Applying obfuscation techniques only at the moment when the data is used and not when it is collected has strong advantages in terms of privacy and utility. When collecting and storing the data, their full quality and utility is preserved. When certain subsets of the data are needed, the usage of the data is often clear and the obfuscation can be tailored to that particular purpose. The latter achieves better privacy and utility preservation than generic obfuscation solutions that must preserve the core utility for *any* subsequent data usage.

5 Conclusion

In this paper we have shown how to realize cryptographically strong pseudonymization when data is collected from a multitude of data sources. Our solution does not require the data sources to share any secret state and outsources the pseudonymization to a central, yet fully oblivious converter. Given the severe privacy limitations that are inherent when consistent pseudonyms are used, ScrambleDB produces unlinkable pseudonyms and introduces a dedicated join procedure. With ScrambleDB we achieve the optimal privacy protection when data is stored at rest, as all data is broken into unlinkable data snippets. Only when certain subsets of the data are needed, the linkage is established in a controlled and non-transitive manner.

Our work complements the recent results for non-transitive joins over encrypted databases [22] and provides a solution for settings where there is no trusted data curator that learns all accumulated data in fully identifying form, which is the typical setting in many real-world applications.

An interesting direction for future research is how to integrate more fine-grained access control into the converter, e.g., enabling users to upload specific join policies which the converter will enforce on their behalf; ideally, without learning the policies of the individual users.

The core of our scheme is a 3-party oblivious and convertible PRF, which we believe to be of independent interest, e.g., in the context of *key-rotation* where a central key holder blindly updates PRF values towards a new key. In particular, coPRFs could be used in password hashing services and realize a more controlled version of the Pythia PRF service proposed in [15], where the central service blindly re-key the hashes itself instead of sending out update tokens. This reduces the risk of token corruptions as re-keying is then handled in a secure environment.

An interesting open question is how to realize such coPRFs against active adversaries or in a quantum-safe way. Such improvements will immediately carry over to our pseudonym scheme which uses these PRFs in a generic manner.

Acknowledgements

This work was supported by the European Union's Horizon 2020 research and innovation program under Grant Agreement No. 768953 (ICT4CART) and 786725 (OLYMPUS).

References

[1] Striim for gdpr. https://www.striim.com/solutions/gdpr/.
[2] Tokenex cloud security platform saas. https://tokenex.com/gdpr/.
[3] Gunes Acar, Steve Englehardt, and Arvind Narayanan. Four cents to deanonymize: Companies reverse hashed email addresses. https://freedom-to-tinker.com/2018/04/09/four-cents-to-deanonymize-companies-reverse-hashed-email-addresses/, 2018.
[4] Giuseppe Ateniese, Kevin Fu, Matthew Green, and Susan Hohenberger. Improved proxy re-encryption schemes with applications to secure distributed storage. In *NDSS 2005*. The Internet Society, February 2005.
[5] Abhishek Banerjee and Chris Peikert. New and improved key-homomorphic pseudorandom functions. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 353–370. Springer, Heidelberg, August 2014.
[6] Matt Blaze, Gerrit Bleumer, and Martin Strauss. Divertible protocols and atomic proxy cryptography. In Kaisa Nyberg, editor, *EUROCRYPT'98*, volume 1403 of *LNCS*, pages 127–144. Springer, Heidelberg, May / June 1998.
[7] Dan Boneh, Kevin Lewi, Hart William Montgomery, and Ananth Raghunathan. Key homomorphic PRFs and their applications. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 410–428. Springer, Heidelberg, August 2013.
[8] Zvika Brakerski and Vinod Vaikuntanathan. Constrained key-homomorphic PRFs from standard lattice assumptions - or: How to secretly embed a circuit in your PRF. In Yevgeniy Dodis and Jesper Buus Nielsen, editors, *TCC 2015, Part II*, volume 9015 of *LNCS*, pages 1–30. Springer, Heidelberg, March 2015.
[9] Christian Cachin, Jan Camenisch, Eduarda Freire-Stoegbuchner, and Anja Lehmann. Updatable tokenization: Formal definitions and provably secure constructions. Cryptology ePrint Archive, Report 2017/695, 2017. http://eprint.iacr.org/2017/695.
[10] Jan Camenisch and Anja Lehmann. (Un)linkable pseudonyms for governmental databases. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 1467–1479. ACM Press, October 2015.
[11] Jan Camenisch and Anja Lehmann. Privacy-preserving user-auditable pseudonym systems. In *IEEE European Symposium on Security and Privacy, EuroS&P*, pages 269–284, 2017.
[12] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. http://eprint.iacr.org/2000/067.
[13] Emiliano De Cristofaro, Jihye Kim, and Gene Tsudik. Linear-complexity private set intersection protocols secure in malicious model. In Masayuki Abe, editor, *ASIACRYPT 2010*, volume 6477 of *LNCS*, pages 213–231. Springer, Heidelberg, December 2010.
[14] Yves-Alexandre de Montjoye, Laura Radaelli, Vivek Kumar Singh, and Alex Pentland. Unique in the shopping mall: On the reidentifiability of credit card metadata. Science 30: Vol. 347 no. 6221 pp. 536-539, 2015.
[15] Adam Everspaugh, Rahul Chatterjee, Samuel Scott, Ari Juels, and Thomas Ristenpart. The pythia PRF service. In *24th USENIX Security Symposium, USENIX Security 15*, pages 547–562, 2015.
[16] Adam Everspaugh, Kenneth G. Paterson, Thomas Ristenpart, and Samuel Scott. Key rotation for authenticated encryption. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 98–129. Springer, Heidelberg, August 2017.
[17] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. Highly-efficient and composable password-protected secret sharing (or: How to protect your bitcoin wallet online). Cryptology ePrint Archive, Report 2016/144, 2016. http://eprint.iacr.org/2016/144.
[18] Stanislaw Jarecki, Hugo Krawczyk, and Jason Resch. Threshold partially-oblivious prfs with applications to key management. *IACR Cryptology ePrint Archive*, 2018:733,

2018.
[19] Stanislaw Jarecki and Xiaomin Liu. Efficient oblivious pseudorandom function with applications to adaptive OT and secure computation of set intersection. In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 577–594. Springer, Heidelberg, March 2009.
[20] Matthias Marx, Ephraim Zimmer, Tobias Mueller, Maximilian Blochberger, and Hannes Federrath. Hashing of personally identifiable information is not sufficient. In *Sicherheit 2018*, pages 55–68, 2018.
[21] Erika McCallister, Tim Grance, and Karen Scarfone. Guide to protecting the confidentiality of personally identifiable information (PII). NIST special publication 800-122, National Institute of Standards and Technology (NIST), 2010. Available from http://csrc.nist.gov/publications/PubsSPs.html.
[22] Ilya Mironov, Gil Segev, and Ido Shahaf. Strengthening the security of encrypted databases: Non-transitive JOINs. In Yael Kalai and Leonid Reyzin, editors, *TCC 2017, Part II*, volume 10678 of *LNCS*, pages 631–661. Springer, Heidelberg, November 2017.
[23] Moni Naor, Benny Pinkas, and Omer Reingold. Distributed pseudo-random functions and KDCs. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 327–346. Springer, Heidelberg, May 1999.
[24] Arvind Narayanan and Vitaly Shmatikov. Robust de-anonymization of large sparse datasets (how to break anonymity of the netflix prize dataset). arXiv online archive: cs/0610105, 2008. http://arxiv.org/abs/cs/0610105.
[25] PCI Security Standards Council. PCI Data Security Standard (PCI DSS). https://www.pcisecuritystandards.org/document_library?document=pci_dss, 2015.
[26] Raluca Ada Popa, , and Nickolai Zeldovich. Cryptographic treatment of cryptdb's adjustable join. http://people.csail.mit.edu/nickolai/papers/popa-join-tr.pdf, 2012.
[27] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100. ACM, 2011.
[28] Peter Scholl. Extending oblivious transfer with low communication via key-homomorphic prfs. In *Public-Key Cryptography - PKC 2018 - 21st IACR*, volume 10769 of *LNCS*, pages 554–583. Springer, 2018.
[29] Securosis. Tokenization guidance: How to reduce PCI compliance costs. https://securosis.com/assets/library/reports/TokenGuidance-Securosis-Final.pdf.
[30] Smart Card Alliance. Technologies for payment fraud prevention: EMV, encryption and tokenization. http://www.smartcardalliance.org/downloads/EMV-Tokenization-Encryption-WP-FINAL.pdf.
[31] United States Deapartment of Health and Human Services. Summary of the HIPAA Privacy Rule. http://www.hhs.gov/sites/default/files/privacysummary.pdf.
[32] Voltage Security. Voltage secure stateless tokenization. https://www.voltage.com/wp-content/uploads/Voltage_White_Paper_SecureData_SST_Data_Protection_and_PCI_Scope_Reduction_for_Todays_Businesses.pdf.

A Ommitted Definitions

This section contains the formal definitions for re-randomizable encryption and input hiding and one-more unpredictability for coPRFs.

A.1 Re-Randomizable Encryption

The following notion captures that re-randomized ciphertexts are indistinguishable from fresh ones:

Definition A.1 (Re-Randomizable Encryption). *A PKE scheme is called re-randomizable if for all PPT adversaries $\mathcal{A}$ it holds that $\left|\Pr[\mathsf{Exp}^{\mathsf{Enc}}_{\mathcal{A},\mathsf{rerand}}(\tau) = 1] - 1/2\right| \leq \mathsf{negl}(\tau)$ for the following experiment.*

Experiment $\mathsf{Exp}^{\mathsf{Enc}}_{\mathcal{A},\mathsf{rerand}}(\tau)$:
$(epk, esk) \xleftarrow{\$} \mathsf{KGen}(\tau)$, $b \xleftarrow{\$} \{0,1\}$
$(C, state) \xleftarrow{\$} \mathcal{A}(epk, esk)$
proceed only if $\mathsf{Dec}(esk, C) \neq \bot$
$C_0 \xleftarrow{\$} \mathsf{Enc}(epk, \mathsf{Dec}(esk, C))$, $C_1 \xleftarrow{\$} \mathsf{ReRand}(epk, C)$
$b' \xleftarrow{\$} \mathcal{A}(state, C_b)$
return *1* if $b = b'$

All known homomorphic schemes also allow for re-randomization, as they realize $\mathsf{ReRand}(epk, C)$ via $C' \xleftarrow{\$} C \odot \mathsf{Enc}(epk, 1)$ for 1 denoting the neutral element in $\mathbb{G}$. However, in general re-randomizability is not implied by homomorphic encryption which is why we make this property explicit.

A.2 Input Hiding (coPRF)

We now present our definition for the input-hiding property of coPRFs. This property requires the blinded PRF input x to be as "hidden" as possible, even when the evaluator *and* the receiver are corrupt.

Definition A.2 (Input Hiding). *A* coPRF *is called input-hiding w.r.t. to leakage function* leak *if for all PPT adversaries $\mathcal{A}$ there exists a PPT simulator* SIM *such that $\left|\Pr[\mathsf{Exp}^{\mathsf{coPRF,leak}}_{\mathcal{A},\mathsf{hiding}}(\tau) = 1] - 1/2\right| \leq \mathsf{negl}(\tau)$ for the following experiment.*

Experiment $\mathsf{Exp}^{\mathsf{coPRF,leak}}_{\mathcal{A},\mathsf{hiding}}(\tau)$:
$b \xleftarrow{\$} \{0,1\}$
$(x, bpk, state) \xleftarrow{\$} \mathcal{A}(\tau)$
$\overline{x}_0 \xleftarrow{\$} \mathsf{coPRF.Eval.Blind}(bpk, x)$
$\overline{x}_1 \xleftarrow{\$} \mathsf{SIM}(\mathsf{Blind}, bpk, \mathsf{leak}(x))$
$b' \xleftarrow{\$} \mathcal{A}(state, \overline{x}_b)$
return *1 if* $b = b'$

Note that this notion is only needed for blind evaluation, not for conversion as therein the blinded input y can be fully recovered by a corrupt $\mathcal{E}$ and $\mathcal{R}$ by simply running an inverse conversion.

A.3 One-more Unpredictability (coPRF)

Our formal definition for one-more unpredictability of coPRFs that captures security against *active attacks* is as follows.

Definition A.3 (One-More Unpredictability). *A* coPRF *is called one-more unpredictable if for all PPT adversaries $\mathcal{A}$ in the experiment below it holds that* $\Pr[\mathsf{Exp}^{\mathsf{coPRF}}_{\mathcal{A},\mathsf{onemore}}(\tau) = 1] \leq \mathsf{negl}(\tau)$.

Experiment $\mathsf{Exp}^{\mathsf{coPRF}}_{\mathcal{A},\mathsf{onemore}}(\tau)$:
$msk \xleftarrow{\$} \mathsf{coPRF.Setup}(\tau)$
$((i_1, x_1, y_1), ..., (i_n, x_n, y_n)) \xleftarrow{\$} \mathcal{A}^{\mathcal{O}_{\mathsf{Eval}}, \mathcal{O}_{\mathsf{Convert}}}(\tau)$
where $\mathcal{O}_{\mathsf{Eval}}$ *on input* $(i, bpk, \overline{x})$:
returns $\overline{y} \xleftarrow{\$} \mathsf{coPRF.Eval.Exec}(k_i, bpk, \overline{x})$
and $\mathcal{O}_{\mathsf{Convert}}$ *on input* $(i, j, bpk, \overline{y}_i)$:
returns $\overline{y}_j \xleftarrow{\$} \mathsf{coPRF.Convert.Exec}(k_i, k_j, bpk, \overline{y}_i)$
return *1 if all of the following conditions are satisfied:*
1) all tuples $(i_1, x_1, y_1), ..., (i_n, x_n, y_n)$ *are distinct*
2) $y_\ell = \mathsf{coPRF.Eval}(k_{i_\ell}, x_\ell)$ *for* $\ell = 1, \ldots, n$
3) $n > q_{\mathsf{Eval}} + q_{\mathsf{Convert}}$ *where* q_X *denotes the amount of queries to the* $\mathcal{O}_{\mathsf{Eval}}$ *and* $\mathcal{O}_{\mathsf{Convert}}$ *oracles*

B Related Work for coPRFs

In this section we give a detailed comparison between our coPRFs and related concepts.

Oblivious PRFs. We already discussed that our notion of coPRFs can be seen as a generalization of standard 2-party OPRFs to the 3-party setting. Unfortunately, the simple construction by Jarecki et al. [17] that was shown to satisfy their UC-based OPRF definition against active adversaries is not suitable when we assume the requester and receiver to be different parties. Their DH-OPRF scheme is $y = H_1(x, H_0(x)^k)$ where the inner part can be evaluated in a blind way, and the full PRF is completed by the requester when hashing $H_0(x)^k$ together with the input x. This double hashing nicely allows (in the random oracle model) to extract the input x without requiring heavy zero-knowledge proofs.

It is easy to see that this idea does not extend to our setting, as it either requires the receiver to be privy of x, or the requester to learn the PRF output — both should not be possible for 3-party-coPRFs.

Key-Homomorphic PRFs. The construction we use is similar to the key-homomorphic PRF (KH-PRF) by Boneh et al. [7], and thus might raise the question whether KH-PRFs can be used to realize coPRFs. This is not the case though. As the name suggests, KH-PRFs allow for homomorphic operations on the key, namely given $\mathsf{PRF}(k_1, m)$ and $\mathsf{PRF}(k_2, m)$ one can compute $\mathsf{PRF}(k_1, m) \otimes \mathsf{PRF}(k_2, m) = \mathsf{PRF}(k_1 \oplus k_2, m)$. Thus, it can be used to convert two PRF values under keys k_1 and k_2 to the PRF value for key $k_1 \oplus k_2$. Such a conversion requires the knowledge of the preimage x, which makes KH-PRFs not suitable for key conversions where this information is not available. Pseudonymization is such a scenario, as requiring knowledge of the corresponding primary identifiers when converting pseudonyms would render the entire pseudonymization useless.

In general, KH-PRFs cannot be used for re-keying when the goal of the PRF was to deterministically replace a sensitive value with a random surrogate. Our convertible PRFs target exactly such settings as they allow to convert PRF values based only on the knowledge of y_i.

Deterministic Proxy Re-Encryption. The conversion of PRF values from one key to another is similar in spirit to proxy re-encryption (PRE) [4], and in particular to deterministic, symmetric PRE. Again, there are crucial differences though: First, our coPRFs are, and should be, non-reversible. Second, and most importantly, we require the proxy part to be executed in a fully blind manner, i.e, with the proxy learning nothing about its inputs, which is not covered by PRE. In fact, coPRFs can be seen as orthogonal to PRE: In PRE, the proxy is not privy of the secret keys but is allowed to see the ciphertexts it is converting. Whereas in coPRFs the converter does know all secret keys but must do the conversion blindly.

C Proof of Theorem 4.1 (Security of ScrambleDB)

In this section we give the proof that our ScrambleDB construction securely realizes our $\mathcal{F}_{\text{ScrambleDB}}$ functionality against passive adversaries.

Proof. To prove that our protocol securely realizes the ideal functionality $\mathcal{F}_{\text{ScrambleDB}}$, we have to show that for any environment $\mathcal{E}$ and any adversary $\mathcal{A}$, there exist a simulator SIM such that $\mathcal{E}$ cannot distinguish whether it's interacting with $\mathcal{A}$ and the protocol in the real world or with SIM and $\mathcal{F}_{\text{ScrambleDB}}$.

The simulator SIM simulates all honest parties towards the adversary (simulating the real world protocol) and interacts with $\mathcal{F}$ on behalf of all corrupt parties in the ideal world. We denote by "$\mathcal{P}$" that the simulator plays the role of the honest party $\mathcal{P}$ towards the adversary in the real-world.

As we consider only passive adversaries, the corrupt parties are also run by the simulator, but with $\mathcal{A}$ being

privy of the party's secret keys, providing their inputs and receiving their outputs and all internal values including the random coins. We denote such a simulated, passively corrupt party as $\mathcal{P}^*$.

We split the description along the two main cases, depending on whether the converter is honest or (passively) corrupt.

Simulation Case 1: Converter $\mathcal{C}$ is honest

When the converter is honest, we simulate all protocol communication with other honest parties by invoking $\mathcal{F}_{\mathsf{SMT}}$ on dummy messages of zeroes. Recall that the leakage function size will reveal the number of rows and columns of a table, which we will use throughout the proof when mimicking encrypted communication through encrypted dummy messages. For simplicity, we assume that all attributes in the table can be represented as single group element in $\mathbb{G}$, i.e., there is no additional leakage of the (block) length of the attributes.

The interesting parts of the simulation are whenever the honest converter engages with a corrupt data source or data lake, as then it has to provide correctly looking messages which it has to "extract" via $\mathcal{F}_{\text{ScrambleDB}}$. Thus, the proof focuses exclusively on these cases.

Pseudonymization Request. When the **data source is corrupt**, the simulation starts when $\mathcal{S}^*$ receives the input $(\textsc{NymReq}, sid, pqid, \mathsf{T}^{m\times n})$ from the adversary. It then runs the protocol, computing the blinded pseudonym request and sends it to the simulated, honest "$\mathcal{C}$". The simulator also triggers an equivalent process in the ideal world by sending $(\textsc{NymReq}, sid, pqid, \mathsf{T}^{m\times n})$ on behalf of the ideal party $\mathcal{S}$ to $\mathcal{F}_{\text{ScrambleDB}}$. $\mathcal{S}^*$ then hands all randomness needed to create the blinded pseudonyms and ciphertexts to $\mathcal{A}$.

When, "$\mathcal{C}$" receives the message from "$\mathcal{S}$", SIM mimics the completed transmission in the ideal world by sending $(\textsc{NymReq}, sid, pqid, \mathtt{ok})$ to $\mathcal{F}_{\text{ScrambleDB}}$.

Pseudonymized & Unlinkable Data Output. When the **data lake is corrupt**, "$\mathcal{C}$" must produce correct looking outputs towards $\mathcal{L}^*$ which it creates with the help of $\mathcal{F}_{\text{ScrambleDB}}$. Thus, when SIM receives $(\textsc{NymOut}, sid, pqid)$ from $\mathcal{F}_{\text{ScrambleDB}}$ it immediately returns $(\textsc{NymOut}, sid, pqid, \mathtt{ok})$, upon which it will receive $(\textsc{NymOut}, sid, pqid, \mathcal{S}, (\mathsf{T}_1^{1\times n}, \ldots, \mathsf{T}_m^{1\times n}))$ from the functionality in the role of the corrupt, ideal-world $\mathcal{L}$. The simulated converter "$\mathcal{C}$" then uses this ideal-world output and transforms it into a real-world output towards the corrupt data lake. To this end, "$\mathcal{C}$" prepares correct looking blinded pseudonyms and encrypted data values using the simulator $\mathsf{SIM}_{\mathsf{coPRF}}$ of the coPRF scheme and encrypting each data item from scratch.

Finally, "$\mathcal{C}$" sends these blinded and encrypted tables as $(\textsc{NymOut}, sid, qid, \mathcal{S}, \mathsf{T}_1^{1\times n}, \ldots, \mathsf{T}_m^{1\times n})$ to $\mathcal{L}^*$.

When $\mathcal{L}^*$ receives "$\mathcal{C}$"'s message, it correctly unblinds and decrypts the response from "$\mathcal{C}$" and hands the output to $\mathcal{A}$.

Overall, this simulation is indistinguishable to $\mathcal{A}$ by the pseudorandomness and collusion-resistance of coPRF, the re-randomizability of RE and the authenticity of messages provided by $\mathcal{F}_{\mathsf{SMT}}$.

Join Request. When the **data lake is corrupt**, $\mathcal{L}^*$ receives the input $(\textsc{JoinReq}, sid, jqid, (tid_1, \ldots, tid_l), \mathcal{P})$ from $\mathcal{A}$ and correctly follows the protocol, sending blinded and encrypted versions of the requested tables to "$\mathcal{C}$" (as the corrupt data lake knows all the tables for $(tid_1, \ldots, tid_l)$). The simulator hands all randomness used by coPRF.Convert.Blind and RE.Enc to $\mathcal{A}$ and mimics the request in the ideal world by sending $(\textsc{JoinReq}, sid, jqid, (tid_1, \ldots, tid_l), \mathcal{P})$ to $\mathcal{F}_{\text{ScrambleDB}}$.

When "$\mathcal{C}$" receives the join request message sent by $\mathcal{L}^*$, it mimics the completion in the ideal world by sending $(\textsc{JoinReq}, sid, jqid, \mathtt{ok})$ to $\mathcal{F}_{\text{ScrambleDB}}$.

Join Output. When the **data processor is corrupt**, "$\mathcal{C}$" prepares a correctly looking output using the information the simulator receives as corrupt data processor in the ideal world. That is, when the simulator receives $(\textsc{JoinReq}, sid, jqid, \ell, \mathcal{P})$ from $\mathcal{F}_{\text{ScrambleDB}}$ it immediately responds with $(\textsc{JoinReq}, sid, jqid, \mathtt{ok})$ upon which it will receive $(\textsc{Joined}, sid, jqid, (\widetilde{\mathsf{T}}_1^{1\times n_1}, \ldots, \widetilde{\mathsf{T}}_l^{1\times n_l}))$ from the functionality in the role of the ideal-world $\mathcal{P}$. The tables $(\widetilde{\mathsf{T}}_1^{1\times n_1}, \ldots, \widetilde{\mathsf{T}}_l^{1\times n_l})$ are indexed with ephemeral, random, but consistent *join-ids* and contain the real attribute values. The simulated "$\mathcal{C}$" now uses the *join-ids* provided by the functionality as coPRF output towards the corrupt real-world $\mathcal{P}^*$. "$\mathcal{C}$" also encrypts the received table fields using RE.Enc. Finally, "$\mathcal{C}$" sends $(\textsc{Join}, sid, jqid, (\widetilde{\mathsf{T}}_1^{1\times n_1}, \ldots, \widetilde{\mathsf{T}}_l^{1\times n_l}))$ to $\mathcal{P}^*$ via $\mathcal{F}_{\mathsf{SMT}}$ and ends. These simulated tables $(\widetilde{\mathsf{T}}_1^{1\times n_1}, \ldots, \widetilde{\mathsf{T}}_l^{1\times n_l})$ are indistinguishable from correctly derived ones in the ScrambleDB protocol by the pseudorandomness and collusion-resistance of coPRF and the re-randomizability of RE.

When $\mathcal{P}^*$ receives "$\mathcal{C}$"'s, where $\mathcal{F}_{\mathsf{SMT}}$ guarantees that $\mathcal{A}$ cannot alter the messages being sent by the simulated converter, it follows the ScrambleDB protocol and correctly unblinds and decrypts the response from "$\mathcal{C}$" (which in this case contains the tables received via $\mathcal{F}_{\text{ScrambleDB}}$). $\mathcal{P}^*$ also outputs all received and intermediate values to $\mathcal{A}$.

Simulation Case 2: Converter $\mathcal{C}$ is corrupt

When the converter is corrupt, the adversary is privy of the coPRF key and can compute the PRF-core of the pseudonyms himself. Thus, SIM must ensure that even with the knowledge of this secret key, the adversary cannot distinguish the simulated from the real protocol. We again describe the simulator along the protocol steps and branch depending on the corruption status of the involved parties. As the converter is corrupt, the simulation now needs to "look ahead", meaning that the simulation of both the pseudonymization and join *request* will be handled differently for honest and corrupt *receivers*. As before, the corrupt parties are run by the simulator, but with $\mathcal{A}$ being privy of the party's secret keys, providing their inputs and receiving all internal values including the random coins.

Pseudonymization Request. When the **data source is honest** and the **data lake is honest** too, SIM starts when it receives $(\text{NYMREQ}, sid, pqid, \ell, \mathcal{S})$ with $\ell = \mathsf{size}(\mathsf{T}^{m\times n})$ from $\mathcal{F}_{\text{ScrambleDB}}$. The simulator immediately returns $(\text{NYMREQ}, sid, pqid, \texttt{ok})$ upon which it will receive $(\text{NYMREQ}, sid, pqid, \mathcal{S}, \mathsf{ATTR}(\mathsf{T}^{m\times n}), n)$ in the role of the corrupt, ideal-world $\mathcal{C}$. "$\mathcal{S}$" then mimics a real request by sending dummy values for all blinded coPRF requests and the encrypted data items. More precisely, "$\mathcal{S}$" assembles the message as described in ScrambleDB but uses the following $\overline{x}_i$ and $\overline{\mathsf{T}}^{m\times n}[i,j]$ values instead of the real ones[1]: For $i = 1,\dots,n$: $\overline{x}_i \xleftarrow{\$} \mathsf{coPRF.Eval.Blind}(bpk_{\mathcal{L}}, 0)$; For $j = 1,\dots,m$: $\overline{\mathsf{T}}^{m\times n}[i,j] \xleftarrow{\$} \mathsf{RE.Enc}(epk_{\mathcal{L}}, 0)$. Finally, "$\mathcal{S}$" sets $\mathsf{KEY}(\overline{\mathsf{T}}^{m\times n}) \leftarrow (\overline{x}_1,\dots\overline{x}_n)$, sorts $\overline{\mathsf{T}}^{m\times n}$ by the primary key and sends $(\text{NYMREQ}, sid, pqid, \overline{\mathsf{T}}^{m\times n})$ to $\mathcal{C}^*$.

This simulation is indistinguishable to the adversary (having corrupted $\mathcal{C}$) by the obliviousness of coPRF, and the CPA-security of RE.

When the **data source is honest** and the **data lake is corrupt**, then SIM must prepare a request message that contains the actual data values and correctly formed pseudonym requests as expected by the corrupt data lake. In this corruption setting, the simulator gains more information from the functionality as it receives $(\text{NYMREQ}, sid, pqid, \ell, \mathcal{S})$ with ℓ being the table $\mathsf{T}^{m\times n}$ sent by the honest (ideal-world) source $\mathcal{S}$ but where the unique identifiers $uid_i,\dots,uid_n$ are replaced by $\mathsf{leak}(uid_1),\dots,\mathsf{leak}(uid_n)$ and leak being the leakage inherited from the coPRF. The simulated "$\mathcal{S}$" uses these values and the simulator $\mathsf{SIM}_{\mathsf{coPRF}}(\mathsf{Blind},\cdot,\cdot)$ from the coPRF scheme to simulate the blinded PRF requests $\overline{x}_1,\dots,\overline{x}_n$. The encrypted data values are done as in the real protocol, leveraging the full knowledge of $\mathsf{T}^{m\times n}$ in this setting.

When the **data source is corrupt**, the simulation is the same as in the case of an honest converter described above (i.e., here the simulation does not depend on the corruption status of $\mathcal{L}$).

In all cases, when $\mathcal{C}^*$ receives the message sent by "$\mathcal{S}$" (or $\mathcal{S}^*$), it outputs the message to $\mathcal{A}$ and mimics the completed transmission in the ideal world by sending $(\text{NYMREQ}, sid, pqid, \texttt{ok})$ to $\mathcal{F}_{\text{ScrambleDB}}$.

Pseudonymized & Unlinkable Data Output. When $\mathcal{C}^*$ receives $(\text{NYMOUT}, sid, pqid)$ from the adversary, the simulated corrupt converter performs the protocol steps of ScrambleDB. If the **data lake is honest**, $\mathcal{C}^*$ blindly converts and re-randomizes the dummy (but correctly formed) messages assembled by "$\mathcal{S}$" (or $\mathcal{S}^*$). If the **data lake is corrupt**, $\mathcal{C}^*$ has received a proper protocol message from "$\mathcal{S}$" (or $\mathcal{S}^*$) and thus behaves exactly as in the real protocol.

When the data lake receives $\mathcal{C}^*$'s message, it continues as in the first case, where $\mathcal{C}$ is honest. However, to argue indistinguishability of the simulation when $\mathcal{L}$ is honest, we now rely on the pseudorandomness of the PRP. This property guarantees that the pseudonyms output by an honest "$\mathcal{L}$" towards the environment are still indistinguishable from random, despite the adversary knowing the coPRF key (and possibly controlling $\mathcal{S}$ too).

Join Request. When the **data lake is honest** and the **data processor is honest** too, the simulation starts when SIM receives the message $(\text{JOINREQ}, sid, jqid, \ell, \mathcal{P})$ with $\ell = (\mathsf{size}(\mathsf{T}_1^{1\times n_1}),\dots,\mathsf{size}(\mathsf{T}_l^{1\times n_l}))$ from $\mathcal{F}_{\text{ScrambleDB}}$. "$\mathcal{L}$" then creates l simulated tables of the correct size, i.e., with dummy blinded pseudonyms $\overline{y}_{i,j} \leftarrow \mathsf{coPRF.Convert.Blind}(bpk_{\mathcal{P}}, 0)$ as keys and filled with dummy ciphertexts, being encryptions of 0 under $epk_{\mathcal{P}}$. Recall that $\mathsf{size}(\mathsf{T}_j^{1\times n_j})$ reveals the number of rows n_j of each table which "$\mathcal{L}$" uses to create the correct amount of dummy values. This is indistinguishable to $\mathcal{A}$ by the conversion-obliviousness of coPRF and the CPA security of RE.

If the **data lake is honest** and the **data processor is corrupt**, then SIM receives $(\text{JOINREQ}, sid, jqid, \ell, \mathcal{P})$ from the functionality where $\ell =$

[1] As stated before, for simplicity we assume that all attribute values can be encoded as one group element, and thus there is no additional leakage of the length of the attributes.

$(\mathsf{T}_1^{1\times n_1},\ldots,\mathsf{T}_l^{1\times n_l})$ are the tables to be joined, each with primary keys $\mathsf{leak}(uid_1),\ldots,\mathsf{leak}(uid_{n_j}))$ and leak being the leakage inherited from the coPRF. The simulator uses this information to create a join request with encrypted and blinded tables $(\widetilde{\mathsf{T}}_1^{1\times n_1},\ldots,\widetilde{\mathsf{T}}_l^{1\times n_l})$ that are indistinguishable from ones derived via the ScrambleDB protocol. First, "$\mathcal{L}$" uses the simulator $\mathsf{SIM}_{\mathsf{coPRF}}$ from the coPRF scheme to obtain the simulated blinded PRF requests $\overline{x}_1,\ldots,\overline{x}_{n_j}$ with $\overline{x}_i \xleftarrow{\$} \mathsf{SIM}_{\mathsf{coPRF}}(\mathsf{Blind}, bpk_{\mathcal{P}}, \mathsf{leak}(uid_i))$. Using the converter's secret coPRF key, "$\mathcal{L}$" then transforms the simulated $\overline{x}$ values into the expected $\overline{y}$ values which are used as primary keys for $(\widetilde{\mathsf{T}}_1^{1\times n_1},\ldots,\widetilde{\mathsf{T}}_l^{1\times n_l})$. It also sets all data entries in $(\widetilde{\mathsf{T}}_1^{1\times n_1},\ldots,\widetilde{\mathsf{T}}_l^{1\times n_l})$ to be the freshly encrypted values from $(\mathsf{T}_1^{1\times n_1},\ldots,\mathsf{T}_l^{1\times n_l})$ using $\mathcal{P}^*$'s public key. Finally, "$\mathcal{L}$" sends $(\textsc{JoinReq}, sid, jqid, (\widetilde{\mathsf{T}}_1^{1\times n_1},\ldots,\widetilde{\mathsf{T}}_l^{1\times n_l}), \mathcal{P})$ to $\mathcal{C}^*$. Overall, this simulation is indistinguishable to $\mathcal{A}$ by the input-hiding property of coPRF and the re-randomizability of RE.

If the **data lake is corrupt**, $\mathcal{L}^*$ receives the input $(\textsc{JoinReq}, sid, jqid, (tid_1,\ldots,tid_l), \mathcal{P})$ from $\mathcal{A}$ and proceeds according to the ScrambleDB protocol, handing all randomness to $\mathcal{A}$.

Join Output. When $\mathcal{C}^*$ receives the join request message sent by "$\mathcal{L}$" (or $\mathcal{L}^*$) via $\mathcal{A}$, it proceeds according to the $\mathcal{F}_{\text{ScrambleDB}}$ protocol and hands all messages and randomness to the adversary. Again, $\mathcal{F}_{\mathsf{SMT}}$ guarantees that $\mathcal{A}$ cannot alter the messages being sent by the simulated converter. Note that we have already ensured in the join request simulation that the request messages are well-formed. In particular, if $\mathcal{L}$ or $\mathcal{P}$ are corrupt, then the message sent to $\mathcal{C}$ contains properly blinded PRF conversion requests and encrypted data values. Only if $\mathcal{L}$ and $\mathcal{P}$ are both honest, the input contains dummy values of the correct format that are indistinguishable from real ones.

When the **data processor is honest**, and "$\mathcal{P}$" receives the simulated message from $\mathcal{C}^*$, then SIM finalizes the request in the ideal world by sending $(\textsc{Join}, sid, jqid, \texttt{ok})$ to $\mathcal{F}_{\text{ScrambleDB}}$ and ends. The environment, "being" the honest $\mathcal{P}$ in the ideal world will receive tables indexed with random *join-id*. That is, here we rely on the pseudorandomness of the PRF applied by "$\mathcal{P}$" in the real-world protocol to ensure that $\mathcal{A}$ cannot determine "$\mathcal{P}$"'s output despite knowing the coPRF key.

When the **data processor is corrupt**, $\mathcal{P}^*$ receives $\mathcal{C}^*$'s message via $\mathcal{A}$ and follows the ScrambleDB protocol: all the input values have the proper format due to the simulation described for $\mathcal{L}$ above. $\mathcal{P}^*$ also hands all received and intermediate values to $\mathcal{A}$. ■