DeCor: Decorrelating unsubscribed users' data from their identities

Anonymous Authors

1 Introduction

1.1 Motivation

Web application companies face increasing legal requirements to protect users' data. These requirements pressure companies to properly delete and anonymize users' data when a user requests to *unsubscribe* from the service (i.e., revoke access to their personal data). For example, the GDPR requires that any user data remaining after a user unsubscribes is *decorrelated*, i.e., cannot be (directly or indirectly) used to identify the user [4].

In this paper, we propose DeCor, a new approach to managing user identities in web applications. DeCor meets the de-identification requirements in the GDPR, and goes beyond: with DeCor, it is possible for users to switch between a privacy-preserving unsubscribed mode and an identity-revealing subscribed mode at any time. This facilitates important new web service paradigms, such as users granting a time-limited "lease" of data to a service instead of having a permanent service account.

1.2 Goals

DeCor's goal is to provide the following properties:

Decorrelation. Informally, decorrelation should guarantee that it is impossible to distinguish between two records formerly associated with the same unsubscribed user and two records from different unsubscribed users.

Resubscription. Users should be able to easily switch between a privacy-preserving unsubscribed mode and an identity-revealing subscribed mode, without permanently losing their application data.

DeCor must implement these properties while ensuring (1) performance comparable to today's widely-used databases, and (2) easy adoption (requiring little to no modification of application schemas or semantics).

1.3 Threat Model

We make the following assumptions of an attacker whose goal is to break decorrelation:

- An attacker can perform only those queries allowed by the application API: an attacker can access the application only via its public interface. [lyt: Alternatively, an attacker could perform arbitrary queries on some public subset of the application schema (e.g., all tables other than the mapping table, or all tables marked with some compliance policy); arbitrary queries over the entirety of the table are out of scope, unless "private" tables are removed and stored by unsubscribing users.]

- An attacker cannot perform application queries to the past or search web archives: information from prior application snapshots may reveal exactly how data records were decorrelated from unsubscribed users.

- An attacker cannot gain identifying information from arbitrary user-generated content (for example, a reposted screenshot, or text in user stories or comments). Decorrelation seeks to remove identifying information from user-generated data that can be enumerated or follows a specific pattern (e.g., a birthday or email address), and application metadata (e.g., date of postings, database ID columns).

1.4 Modeling Decorrelation

Application data is structured as tables, each containing a different *data entity*, e.g., a story, user, or vote. Queries write, read from, and compute over entities.

Decorrelation of a user entity splits the user into a set of *ghost user* entities, with one ghost user for each data entity that is *clustered* by the user. Clusters are determined by foreign key attributes in data tables which correspond to user identifiers, or by attributes that are used to JOIN with the user table. For example, stories cluster by user because they contain a foreign key attribute (`user_id`) into the user table, specifying which user wrote the story.

However, breaking up user-associated clusters of entities may not sufficiently decorrelate these entities from the user. For example, perhaps only stories belonging to the user possess a particular tag. Or perhaps the user's stories are all upvoted only by all of the users' friends. These pieces of information allow an adversary to correlate a story back to a user's identity without using the `user_id` value.

To prevent these information leaks, decorrelation of a user *propagates* recursively through entities associated with the user. To generalize decorrelation, let a of entity type A be the entity being decorrelated. Then decorrelation on a performs the following:

1. **Break Direct Clusters and Recurse.** a splits into ghost entities of type A, one for each entity $b \in B$ which is clustered by a. Each cluster entity b is recursively decorrelated from its data.

   For example, each story (the b) clustered by a user (the a) is reassigned to a ghost user. Then each story is itself decorrelated, splitting into one ghost story per vote on the story. If votes also have clusters, then the votes would also be decorrelated from these entities, and so on. This recursive decorrelation prevents structural information about the story (e.g., friends who liked the story) from leaking identifying information about who the story may belong to.

2. **Decorrelate Identity Substitutes.** If more than one of the b clustered by a also clusters by entity $c \in C$, then we decorrelate c from its data. More formally, let B_a be the set of entities of type B clustered by a, the entity undergoing decorrelation. Then c is an identity substitute for a if $\exists B_c$ such that

   - $B_c \subseteq B_a$
   - $|B_c| > 1$
   - B_c is a set of entities of type B clustered by c

   For example, a tag may act as an identity substitute if many of the stories associated with a `user_id` are also associated with that tag, and the tag is then decorrelated.

1.5 Tradeoff: Ghost Entities vs. Utility

Decorrelation fundamentally decreases the utility of data for the application: introducing ghost entities and ghost associations decrease the amount of accurate information returned to application queries. In essence, ghosts act as a mechanism to add noise to obfuscate a user's presence in a dataset.

Maintaining aggregate accuracy. Queries that specifically perform aggregations and return statistical measures (e.g., the count of number of users in the system, or the number of stories per user), can return significantly different results. This affects the utility of the data for the application: for example, if the application relies on the number of stories per tag to determine hot topics, these would be heavily changed if ghost tags were created. In addition, the adversary may learn which entities are ghosts: for example, an abnormally low count of stories per tag might indicate to an adversary that these tags are ghost tags. [lyt: But perhaps it's ok if an adversary can tell what's a ghost, as long as it can't tell which user each ghost is correlated with.]

One solution is to analyze the aggregations performed by application queries, and then introduce ghosts entities that lead to the same (or close-enough) aggregation result. For example, if a tag is split into ghost tags, one per story associated with the tag, but the application still would like the count of stories for this tag to be high, one of the ghost tags can be populated with many ghost stories to retain the count of stories per tag. These ghost stories identifiers would be uploaded upon recorrelation, allowing DeCor to remove any ghost stories that were created. In this situation, this creates a large number of ghost tags with only one story, which would only affect application semantics if a query calculates the mean of number of stories per tag (which requires a different way of populating tags with stories). Note that this solution may be impossible for certain combinations of aggregations (e.g., queries that return both the average stories per tag and also the total number of stories).

A second solution would be to store and separately update answers to aggregation queries; these answers would be updated when queries update the data tables.

[lyt: I don't *think* differential privacy really should be applied here, because we'd also face the issue of running out of privacy budget. Adding noise might ensure that the impact of any one (real/ghost) user is very little, but it has its own noise/utility tradeoff. Furthermore, the amount of noise necessary if many ghost users are created might be too large.]

Maintaining application structure. Many applications allow queries to return individual entities (e.g., a particular story) in addition to statistics and aggregations. Ghost entities muddy individual entities' data, and potentially alter the application's structure in ways that render it meaningless. For example, subreddits—similar to tags or categories—on Reddit create communities and social circles that are fundamental to how people interact on Reddit. These subreddits may become nonsensical if many users unsubscribe and stories are decorrelated from a particular subreddit.

To prevent these potentially disastrous scenarios, application developers can annotate which entities do not need to be recursively decorrelated during the first step of decorrelation, which breaks direct clusters. For example, perhaps the cluster of votes grouped by story does not leak any identifying information, and thus stories do not need to be decorrelated. The developer may also indicate if the entity (e.g., the story) does leak identifying information through its clusters but decorrelating it would destroy application integrity, which results in the removal of the entity and its dependencies. However, the entity cannot be restored upon resubscription in this case.

In the second step of decorrelation, we decorrelate identity substitutes only enough to be below a *correlation threshold* specified by the application developer. Again, if the application developer annotates a particular entity as essential to the application structure and the correlation threshold is exceeded, then the entity is entirely removed from the database.

At a high level, the correlation threshold limits the ability of an adversary to use an identity substitute as a proxy for a user's identity. This correlation threshold is defined on a type of entity C as follows. Let B_a be the set of entities of type B clustered by a, the entity undergoing decorrelation. Let $c \in C$ be an entity that acts as an identity substitute for a, with $B_c \subseteq B_a$ clustered by c. Let B_{total} be the set of entities of type B clustered by c. The correlation threshold for the entity c is the maximum value for the proportion B_a/B_{total}. Thus, lowering the threshold increases the number of entities in B_{total} that belong to other users, and lowers the ability of the adversary to use c to correlate the $b \in B_c$ to a single identity.

For entities such as tags or categories, a reasonable correlation threshold might be 0.05: less than 5% of all stories associated with that tag are allowed to cluster by one (unsubscribed) user. Note that user entity types are always associated with a correlation threshold of 0: after a user unsubscribes, no entities should be clustered by that user.

Note that meeting the threshold can be done by breaking up the entity c into ghosts, each associated with a singleton $b \in B_c$, until the threshold is met. Another strategy might pollute B_{total} with *new* ghost entities to lower the proportion, rather than breaking up the cluster.

2 Background and Related Work

2.1 Differentially Private Queries

PINQ, a data analysis platform created by McSherry [2], provides formal differential privacy guarantees for any query allowed by the platform. Data analysts using PINQ can perform transformations (Where, Union, GroupBy, and restricted Join operations) on the underlying data records prior to extracting information via aggregations (e.g., counts, sums, etc.) The aggregation results include added noise to meet the given privacy budget ε, ensuring that analysts only ever receive ε-differentially-private results. PINQ calculates the privacy loss of any given query based on transformations and aggregations to be performed; if a particular query exceeds a predefined privacy budget, PINQ refuses to execute the query.

Differential privacy provides a formal framework that defines the privacy loss a user incurs when the user's data is included in the dataset. However, the setting of differential privacy that of DeCor in several important ways.

First, web applications' utility often derives from visibility into individual data records. PINQ restricts JOIN transformation and exposes information only through differentially-private aggregation mechanisms. While sufficient for many data analyses, this approach severely hinders web application functionality. DeCor supports all application queries, even ones that expose individual records, such as `SELECT story.content, story.tag FROM stories LIMIT 10`. However, this makes formally defining the privacy guarantees of DeCor more complex than simply applying DP. DP provides formal guarantees for results such as sums or averages because such results are computed using well-defined mathematics, allowing us to neatly capture the total knowledge gained by the adversary. However, in the world of web applications, the knowledge gained via queries that may reveal individual data records cannot be so easily defined: adversaries may learn information from the friends who liked the (ghosted) story, or the time and location the story was posted.

Thus, we cannot, in general, use the DP approach of asking, "by how much do answers to adversary queries change with the presence of a users' decorrelated data?" While we can precisely define which data records an adversary sees with and without a user's decorrelated data, we cannot numerically quantify the knowledge gained by the adversary in the same way as we can quantify the percentage change to a sum or average. Instead, DeCor reduces identifying information in the changes induced by a users' (decorrelated) data, rather than reducing the amount of change itself.

This is demonstrated by how PINQ and DeCor differ in masking entities associated with unique keys. PINQ restricts JOINs to group by unique keys. As an (imperfect) example, selecting a JOIN of stories with users via the UID would result in a record, one per UID, which represents the “group” of all stories associated with a user. A normal join would create a separate record per story, each associated with the user of that UID.

Instead of lumping all matching stories together, DeCor explodes the users' UID into individual unique keys (ghosts), one per story. By PINQ's analysis of stable transformations, this leads to unbounded stability (as the number of different results produced by queries is proportional to the potentially very large number of stories for that user). In PINQ's DP framework, such unbounded stability leads to unbounded privacy loss: one users' data could change aggregation results in "significant" ways unless huge amounts of noise were added.

While DeCor's approach has been shown to be problematic in the past (psuedo/anonymization techniques still allow for reidentification/reconstruction attacks), DeCor provides better anonymization than prior approaches, which decorrelate only coarsely by associating remnants of data with a global placeholder for all deleted users, or do not decorrelate at all (e.g., replacing usernames with a pseudonym, as in the AOL dataset case [**?**]). In addition, DeCor and PINQ's approaches can be complementary: for queries that are more analytic in nature (for example, population statistics used by the web application to measure usage or trends), DeCor can utilize PINQ's technique to provide differentially private guarantees (for a particular snapshot of the dataset). DeCor's decorrelation, in fact, adds more noise than necessary in these cases: a count of unique users who posted about cats will be more imprecise because each post by a decorrelated user is now associated with a unique (ghost) user.

2.2 Deletion Privacy

The right to be forgotten has also been formally defined by Garg et al. [1], where correct deletion corresponds to the notion of leave-no-trace: the state of the data collection system after a user requests to be forgotten should be left (nearly) indistinguishable from that where the user never existed to begin with. While DeCor uses a similar comparison, Garg et al.'s formalization assumes that users operate independently, and that the centralized data collector prevents one user's data from influencing another's.

Other related works:

- Problems with anonymization + reidentification later on (see PINQ for references)
- Deceptive Deletions for protecting withdrawn posts: https://arxiv.org/abs/2005.14113
- "My Friend Wanted to Talk About It and I Didn't": Understanding Perceptions of Deletion Privacy in Social Platforms, user survey https://arxiv.org/pdf/2008.11317.pdf; talk about decoy deletion, prescheduled deletion strategies [3]
- Contextual Integrity
- ML Unlearning
- k-anonymization, pseudonymization

3 Design

4 Implementation

5 Evaluation

6 Discussion

Acknowledgments

Availability

[lyt: USENIX program committees give extra points to submissions that are backed by artifacts that are publicly available. If you made your code or data available, it's worth mentioning this fact in a dedicated section.]

References

[1] Sanjam Garg, Shafi Goldwasser, and Prashant Nalini Vasudevan. Formalizing data deletion in the context of the right to be forgotten. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 373–402. Springer, 2020.

[2] Frank McSherry. Privacy integrated queries. *Communications of the ACM*, 53:89–97, September 2010.

[3] M. Minaei, Mainack Mondal, and A. Kate. "my friend wanted to talk about it and i didn't": Understanding perceptions of deletion privacy in social platforms. *ArXiv*, abs/2008.11317, 2020.

[4] E Parliament. Regulation (eu) 2016/679 of the european parliament and of the council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/ec (general data protection regulation). *Official Journal of the European Union*, L119(May 2016):1–88, 2016. http://eur-lex.europa.eu/legal-content/EN/TXT/?uri=OJ:L:2016:119:TOC.