DeCor: Decorrelating unsubscribed users' data from their identities

Anonymous Authors

1 Introduction

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 data remaining after a user unsubscribes cannot be (directly or indirectly) used to identify the user [4].

In this paper, we propose DeCor, which enables applications to meet 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 without permanently losing their data. 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.

DeCor provides strong de-identification properties by maximally *decorrelating* a user's data entities (e.g., posts and upvotes) from each other and from the user's identity upon unsubscription. DeCor makes the key observation that data entities can leak identifying information in two ways: (1) directly via its content, and (2) indirectly via the correlations it has to other data entities in the system. Correlations can range from obviously identifying (e.g., posts correlated with a user clearly belong to that user), to subtly perilous: posts correlated with a particular user-generated tag most likely belong to the tag's author, and posts liked by the same group of users likely belong to a friend of the group.

Many web applications have come up with solutions to address (1) by anonymizing unique identifiers, often leaving arbitrary user-generated content out of scope. However, this only addresses information leakage from single data entities in isolation. DeCor's decorrelation additionally eliminates identifying information stemming from (2) by removing correlations between data entities.

Decorrelation breaks connections between data entities that would otherwise allow an adversary to determine that two entities belong to the same (unsubscribed) user: post-decorrelation, an adversary cannot distinguish between a scenario in which two data entities have been generated by two distinct users, and one in which they were both generated by the same (unsubscribed) user. If two of a user's entities cannot be correlated back to the user, then at most one of these entities may leak identifying information about the user.[1] Thus, if single data entities are appropriately anonymized (via methods to address leakage from a single entity's content), then DeCor prevents identifying information from being leaked via correlations.

Applications using DeCor can easily and correctly decorrelate users' entities when users unsubscribe by providing a high-level specification that captures the desired application semantics for unsubscribed users. Furthermore, DeCor automatically performs decorrelation in a way that allows for recorrelation upon resubscription, while still achieving performance comparable to today's widely-used databases and requiring no modification of application schemas.

1.1 Threat Model

An adversary aims to relink decorrelated entities to unsubscribed users after DeCor performs decorrelation. We make the following assumptions about such an adversary:

- An adversary can perform only those queries allowed by the application API, i.e., can access the application only via its public interface. [lyt: Alternatively, an adversary 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.]

[1] Assume for a contradition that both entities leak identifying information: then both entities are more likely to have been generated by a particular identity than any other, contradicting our assumption.

- An adversary 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 adversary 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).

1.2 Decorrelation Design

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. All entities (e.g., posts) that share a correlation with the same entity (e.g., a user) form a *cluster* identified by that entity. In database terms, entities in the same cluster are identified by having the same foreign key value (e.g., a particular value for column `user_id`).

Decorrelation of a user breaks any clusters around the user into singleton clusters, each identified by a unique ghost user. In essence, the user is exploded into many ghost user, each correlated with only one of the user's data entities.

However, breaking up clusters around the user may not sufficiently decorrelate these entities from the user. For example, stories belonging to the user may also cluster around a particular tag. Or perhaps one of the user's stories is upvoted only by all of the users' friends (these upvotes cluster around the story). These pieces of information allow an adversary to correlate a story back to a single user, even when clusters around the user no longer exist.

The key observation here is that two types of clusters can still leak identifying information about the user: (1) clusters identified by data entities owned by the users, and (2) clusters consisting of the user's data entities, identified by other entities. By breaking any such clusters into singletons by introducing ghost entities (as was done with introducing ghost users), DeCor removes any identifying information leaked from correlations between user's data entities and other entities in the application.

More generally, decorrelation must recursively break up any clusters that may recorrelate a data entity back to the identifier of a broken-up cluster. Let A and B be entity types. Let $a \in A$ be the entity being decorrelated. a identifies at least one cluster $B_a \subseteq B$. Decorrelation on a does the following:

1. **Break direct clusters and recurse.** a splits into ghost entities $A_g \subseteq A$, one for each entity $b \in B_a$. Decorrelation then runs on each cluster entity b.

   Furthermore, if the A_g are in a cluster identified by some entity c, c is decorrelated.

2. **Break clusters around identity proxies and recurse.** If more than one of the $b \in B_a$ is also in a cluster that is identified by an entity $c \in C$ (an identity "proxy"), then we decorrelate c from its data.

For example, each story (the b) posted by a user a belongs in a cluster B_a identified by a. Decorrelation step (1) reassigns each story to a ghost user. Then each story is itself decorrelated. If stories are decorrelated into a ghost story per vote, for example, then the votes would be decorrelated. The ghost stories would cluster by the ghost user, so this ghost user would have to further decorrelate.

Decorrelation step (2) may find that a particular tag c identifies at least two of the users' stories (these stories belong in a cluster B_c identified by c). The tag is then decorrelated to ensure that nothing related to the tag may reassociate these stories with the same tag. This recursive decorrelation prevents any other correlations with the user's stories from leaking identifying information about the stories' author.

Ghost entities are generated by replacing foreign key attributes with a unique identifier drawn from a random distribution; all other attributes can be replaced by default values, or using application-specific generators (for example, a randomly generated username or phone number). If foreign keys values are kept and consequently cluster the generated ghost entities, then these foreign entities must be decorrelated to ensure that ghosts cannot be grouped back together.

[lyt: Given this ghost generation scheme, decorrelation of a story would then destroy story content (essentially deleting the story... the user would not get this content back). A smarter way to ghost stories may be to split up the content into "pieces", one per ghost story, and encrypt these pieces so that ghost stories cannot be relinked together, but the user could decrypt the pieces and reform the original story.]

1.3 The Impact of Decorrelation

Taken to the extreme, decorrelation strips any links to (real) application data from data entities recursively related to the user via clusters. While this removes as much correlation-based identifying information as possible while keeping data entities present, completely decorrelated data entities may be useless to the application. Worse yet, the noise and data pollution from ghost entities may instead affect the accuracy and semantics of the application: users may see meaningless content, comment threads may be disjoint and scattered, and highly-ranked content may suddenly lose votes. On the other extreme, however, performing no decorrelation at all fails to adequately de-identify users.

DeCor balances between the two extremes by decorrelating only to the extent allowed by the application developer. The developer provides schema annotations on relationships (foreign keys) that cannot be broken (e.g., votes related to stories) because they would create meaningless ghost content or break application semantics. When possible, DeCor measures the potential amount of identifying information leaked by the retained correlations, and, depending on what the developer specifies, either removes the entity entirely if a *cluster threshold* is exceeded, or adds noise until the amount of leaked information is below the threshold. Furthermore, DeCor allows for entities to be decorrelated without affecting queries which specifically return aggregation results.

Specifying Decorrelation Behavior and Correlation Thresholds. Ghost entities muddy individual entities' data, and potentially alter the application's structure in ways that render it meaningless. For example, perhaps it is meaningless to create ghost stories in order to break up clusters of votes identified by story; or perhaps it would ruin the applications' recommendation-by-tag semantics to create ghost tags. To prevent these potentially disastrous scenarios, application developers provide annotations on which entity types (database tables) that cannot be exploded into ghost entities. The application developer also indicates whether it is acceptable to *add* ghost entities to clusters if clusters cannot be broken, and if so, the mechanism to generate these ghosts.

DeCor determines whether a still-correlated entity needs to be removed—its correlations expose too much identifying information—by computing the proportion of entities in a cluster that are related to the entity being decorrelated. This proportion is compared to a *cluster threshold* [lyt: (developer specified? set by user?)].

If DeCor cannot decorrelate clusters identified by an entity (step 1), then DeCor checks whether it is possible to add ghost entities to the cluster. If yes, then DeCor generates enough ghosts such that the original cluster entities make up less than the cluster threshold proportion of all entities in the cluster. If this correlation threshold is exceeded, or the developer does not allow for ghosts to be added to the cluster, DeCor removes the entity and its dependencies. However, the entity cannot then be restored upon resubscription.

If DeCor cannot decorrelate an identity proxy (step 2), then DeCor calculates what proportion of entities in proxy-identified clusters belong to the entity being decorrelated. If the proportion is below the cluster threshold, no further actions are needed; otherwise, if the developer allows ghosts to be added to the cluster, then DeCor generates enough ghosts such that the proportion drops below the threshold. Again, if the developer does not allow for ghosts to be added to the cluster, DeCor removes the proxy entity and its dependencies.

For entities such as tags or categories, a reasonable cluster threshold might be 0.05: less than 5% of all stories identified by that tag should have been associated with an (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 even for entities for which decorrelation is acceptable, they may only need to be decorrelated until their clusters meet the cluster threshold.

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.]

2 Background and Related Work

2.1 Anonymization in Web Applications

Today's web applications are incentivized to retain as much user data on their platform as possible, both to sell and to process for the application's targeting, and to increase the utility of their platform for other users (e.g., keeping reviews or votes for other users of the application to use to judge a product).

When users on web applications delete their accounts, some applications (e.g., Lobst.rs and Reddit) choose to de-identify user data upon account deletion by replacing unique user identifiers such as usernames with global placeholders ("anonymous") or a pseudonym ("anonymous_mouse"). Other applications (e.g., Facebook and Twitter) delete (most of the) user's data. The first choice optimizes the amount of data the application retains, the second can lead to confusing semantics for the application (e.g., nonsensical comment threads). In both cases, accounts cannot be restored after deletion.

Applications such as Facebook or Twitter do support user account deactivation by hiding (most) of the user's data, but retain all user data with their identifiers in their databases. [lyt: Facebook keeps a "log" of data versions, but this wouldn't be a problem if everything in the database is ghosted!]

DeCor's decorrelation provides stronger de-identification properties than a global placeholder, and keeps as much data as possible in the application while still guaranteeing de-identification. DeCor also supports resubscription without allowing the application to retain identifying user data in its database.

2.2 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.

2.3 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.