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 application developers to specify fine-grained privacy policies to meet de-identification requirements, 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.

Developers using DeCor specify privacy-sensitive data and the state of such data after the user leaves. DeCor *decorrelates* a user's data according to this specification when a user unsubscribes; post-decorrelation application state is guaranteed to match the specification. When a user wishes to resubscribe, DeCor *recorrelates* the user's data with the user, restoring the user to her original subscribed state as much as possible .

DeCor makes the key observation that a user's data entities (e.g., posts and upvotes) 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 (e.g., Lobst.rs and Reddit) have come up with solutions to de-identify direct content by anonymizing unique identifiers, leaving arbitrary user-generated content out of scope. Other applications (e.g., Facebook and Twitter) delete (most of the) user's data. The former optimizes the amount of data the application keeps, but also retains all correlations between data entities; the latter can lead to confusing semantics for the application (e.g., nonsensical comment threads) and the loss of useful application data. In both cases, accounts cannot be restored after deletion. [lyt: 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.]

Developers using DeCor can specify policies that both anonymize unique identifiers, and remove identifying information stemming from correlations between data entities. DeCor's support for fine-grained decorrelation policies gives developers options beyond either completely retaining these links or deleting of all of the user's data and their dependees (while still supporting these options). The complexity of implementating such policies is hidden from the developer: 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 What would perfect decorrelation achieve?

Decorrelation ideally 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 should not be able to 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 the content of individual data entities is appropriately anonymized, then perfect decorrelation prevents identifying information from being leaked via correlations.

[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.

1.1.1 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.
- 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).

1.1.2 Modeling perfect 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. 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 belong to the same table, and have 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 users, 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. Decorrelation recursively breaks up any such clusters into singletons by introducing ghost entities (as was done with introducing ghost users), thus removing any identifying information leaked from correlations between user's data entities and other entities in the application.

More generally, decorrelation continues to 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 recursively on each cluster entity b.

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

1.1.3 Why perfect decorrelation doesn't work in practice

Perfect decorrelation takes decorrelation to the extreme, stripping any links to (real) application data from data entities recursively related to the user via clusters. This removes as much correlation-based identifying information as possible while keeping data entities present, but in practice, completely decorrelated data entities may be useless to the application. Applications may lack a meaningful way to generate ghost entities: what does it mean for a story to become many ghost stories? And even if ghosts can be generated, 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.

DeCor offers developers a way to decorrelate as much as possible *while still retaining application semantics*. As the next section describes, DeCor provides decorrelation specification primitives with which developers can specify which (and how) entities can be decorrelated from other entities, and which entities must be kept clustered for application correctness (e.g., votes correlated with a story).

1.2 Specifying Decorrelation Policies

[lyt: FIGURE HERE]

Decorrelation policies consist of two parts:

1. Ghost entity generation annotations on entity types (tables) for types which can be ghosted, and
2. Edge policy annotations on links (foreign-key relations) between two types of entities (e.g., story→user)

1.2.1 Ghost Entity Generation

Developers can choose between the following ghost generation annotations, which apply at per-column granularity.

- **Cloned:** All ghosts have the same value as the original entity for this column. Recorrelation returns an entity with the original value.
- **Generated:** All ghosts have a generated value for this column. Generated values are chosen to be random, a default value, or generated from the original value via a function provided by the developer. If the developer provides a reversible function (e.g., has encrypted the original value with a user-specific key), then the original value is restored upon recorrelation; otherwise, recorrelation returns a generated value in place of the original.
- **Single Clone, Generated Remainder:** One ghost has the same value as the original entity; all other ghosts have generated values. Recorrelation returns an entity with the original value.

1.2.2 Entity Edge Policies

Policy 1: Do not decorrelate. Application developers specify (via an annotation on a foreign key) that edges between entity types cannot be removed, thus preventing clusters from being broken during decorrelation and preventing recursive decorrelation via that edge. For example, tags or categories may play an essential role for the application, and cannot be split into ghosts. Developers thus annotate the foreign key in the stories table to `tag_id` with this policy.

Policy options:

- **Remove user entity**: If decorrelation cannot propagate down an edge from entity a to entity b, then entity a and all of a's dependees are removed.
- **Achieve cluster threshold** t: This indicates that it is acceptable to *add* ghost entities to clusters that cannot be broken up. At a high level, the cluster threshold t limits the ability of an adversary to use an identity proxy as an alias for a user's identity. DeCor calculates what proportion of entities in these clusters belong to the entity being decorrelated. If the proportion is below t, no further actions are needed; otherwise, DeCor generates enough ghosts such that the proportion drops below the threshold.

  For example, a reasonable cluster threshold might be 0.05 for clusters of stories identified by tag: less than 5% of all stories identified by that tag should have been associated with an (unsubscribed) user.

Policy 2: Decorrelate. Application developers specify that edges between entity types can be removed, and decorrelation can propagate down these edges. For example, all edges emanating from a user can be removed: clusters of any type of entity identified by user may be broken, and ghost users created for each entity.

Resubscription (and recorrelation) removes any created ghost entities and restores any broken links.

[lyt: There could also be an option here of "decorrelate the minimum amount possible to reach a cluster threshold"; not sure if it's really that useful.]

1.3 Maintaining aggregate accuracy.

DeCor optionally allows for entities to be decorrelated without affecting queries which specifically return aggregation results.

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.]

DeCor stores and separately updates answers to aggregation queries; these answers are updated when queries update the data tables, and these queries do not read from the application tables (which may contain ghost records).

An alternate solution might analyze the aggregations performed by application queries, and then introduce ghosts 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. DeCor would remove any ghost stories that were created upon recorrelation. Note that this solution 1) requires that generating ghosts is admissible, and 2) 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).

[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.