Flexible Privacy via Disguising and Revealing

by

Lillian Tsai

A.B., Harvard University (2017)
S.M., Harvard University (2017)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2024

Author .....
Department of Electrical Engineering and Computer Science
May 3, 2024

Certified by .....
M. Frans Kaashoek
Charles Piper Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Co-Certified by .....
Malte Schwarzkopf
Assistant Professor of Computer Science, Brown University
Thesis Co-Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Flexible Privacy via Disguising and Revealing

by

Lillian Tsai

Submitted to the Department of Electrical Engineering and Computer Science
on May 3, 2024, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Users have tens to hundreds of accounts with web services that store sensitive data, from social media to tax preparation and e-commerce sites [**tens**, **hundreds**, **password_life_cycle**]. While users have the right to delete their data (via e.g., the GDPR [**eu:gdpr**] or CCPA [**ccpa**]), more nuanced data controls often don't exist. For example, a user might wish to hide and protect their profiles on an e-commerce or dating app when inactive, and to recover their accounts should they return to use the application. However, services often provide only coarse-grained, blunt tools that result in all-or-nothing exposure of users' private data.

This thesis introduces the notion of *disguised data*, a reversible state in which sensitive data is hidden. To demonstrate the feasibility of disguised data, this thesis also presents Edna—the first system for disguised data—which helps database-backed web applications provide new prirvacy features for users, such as removing their data without permanently losing their accounts, anonymizing their old data, and selectively dissociating personal data from public profiles. Edna helps developers support these features while maintaining application functionality and referential integrity in the database via *disguising* and *revealing* transformations. Disguising selectively renders user data inaccessible via encryption, and revealing restores their data to the application. Edna's techniques allow transformations to compose in any order, e.g., deleting a previously anonymized user's account, or restoring an account back to an anonymized state.

With Edna, web applications can enable flexible privacy features with reasonable developer effort and moderate performance impact on application operation throughput. In the Lobsters social media application—a 160k LoC web application with >16k users—adding Edna and its features takes <1k LoC, and decreases throughput 1–7% in the common case and up to 28% when a heavy user who owns 1% of all application data continuously disguises and reveals their account.

Thesis Supervisor: M. Frans Kaashoek
Title: Charles Piper Professor of Electrical Engineering and Computer Science

Thesis Co-Supervisor: Malte Schwarzkopf
Title: Assistant Professor of Computer Science, Brown University

Acknowledgments

This thesis builds upon prior work on Edna published at SOSP 2023 [**edna**]. I am immensely grateful to my collaborators on Edna over the many years: Hannah Gross, Eddie Kohler, Malte Schwarzkopf, and Frans Kaashoek. Without you all, Edna would not exist today. I would like to thank Akshay Narayan, Anish Athalye, Aurojit Panda, Derek Leung, Gohar Irfan Chaudhry, Henry Corrigan-Gibbs, James Mickens, Kevin Liao, Kinan Dak Albab, Matthew Lentz, Nick Young, Nickolai Zeldovich, MIT's PDOS group, Brown's ETOS group, the SystemsResearch@Google Team, and all our anonymous reviewers: you generously donated your time and insights that greatly improved Edna.

While working on Edna, I am grateful to have been supported by an NSF Graduate Research Fellowship, NSF awards CNS-2045170 and CSR-1704376, a Google Research Scholar award, and funding from VMware. CloudLab [**cloudlab**] also provided essential resources to develop the Edna prototype and evaluate its artifact.

Edna and this thesis would not have been possible without the many people whose advice, support, generosity, and kindness have carried me throughout the past six-plus years. I thank you all not just for Edna, but also for encouraging me to be better in all ways: as a thinker, a collaborator, a friend, and a human being. I would especially like to thank:

My Family

- My parents—Phaih-lan Law and Kenwood Tsai—who raised me and still love me despite my many flaws.
- Stephanie ("Stephie"), my sister and forever role model.
- Ah-ma and Ah-gong, who continue to show me how to embrace life to its fullest.
- Aunty Cecilia, Uncle Jacek, Ah-Ee, Aunty Sally, Tua-ku, Jackie, Wei-Jen, Trevor Hohn, and many aunts, uncles, and cousins, for welcoming me home across the country and around the world.
- Benji, for the light you have brought into my life.

My Mentors and Role Models

- Eddie Kohler, Malte Schwarzkopf, Frans Kaashoek, James Mickens, Peter Druschel, Derek Dreyer, Stefan Saroiu, Margo Seltzer, Kim Keeton, Anastasia Ailamaki, Phillip Levis, Joel Emer, Mathilda van Es, Julia Netter, Alexander Kojevnikov, and Ellen Feigenbaum; for catching me when I stumble, and showing me the many different ways to live life well.
- My Advisors—Malte Schwarzkopf, Frans Kaashoek, and Eddie Kohler—for their unwavering belief in me and my research; I will constantly be in admiration of your endless energy and persistent curiosity.

- My PDOS and MIT peers—Alexandra Henzinger, Anish Athalye, Ariel Szekely, Baltasar Dinis, Ben Holmes, Derek Leung, Gohar Irfan Chaudhry, Hannah Gross, Josh Fried, Ryan Lehmkuhl, Sanjit Bhat, Upamanyu Sharma, Yun-Sheng Chang, Zain Ruan, Inho Cho, Akshay Narayan, Tej Chajed, Ataley Mert Ileri, Jon Gjengset, Frank Cangialosi, Vibhaalakshmi Sivaraman, Joseph Tassarotti, Ralf Jung, Adam Belay, Henry Corrigan-Gibbs, Nickolai Zeldovich, and Robert Morris—for providing an inspiring community and home away from home.

My (Other) Families

- My Musical Family: Solon Gordon, Jenn Chang, Valerie Chen, Lynn Chang, Lisa Wong, Ken Allen, Mary Jane, and the many more who have shared their music with me; and Music for Food and Sharing Notes at MIT for enabling me to give back—even just a little—to the MIT and broader Boston communities through music.

- My Climbing/Running Family: Anders Simpson-Wolf, Lekha Jananthan, Whitney Meza, Oliver Thomas, Rick Wainner, Bess and Chris, John Langan, and the others who helped me become the strongest I've ever been.

- My Friends, all of whom I am very lucky to have met: Serena Booth (and Ducki Booth), Valerie Chen, Melissa Wen, Roberta De Viti, Lara Booth, Daniel Rothchild, Ethan Tsai, Hanson Tam, Kevin Liao, Matthew Lentz, Nikhil Benesch, Samyu Yagati, Anitha Gollamudi, Solon Gordon, Jenn Chang, Anders Simpson-Wolf, Lekha Jananthan, Serena Wang, Zachary Munro, James Deng, Wentong Zhang, Ashvin Swaminathan, Karima Ma, James Tompkin, Helena Caminal, Hamish Nicholson, Henry Burnam, Kevin Loughlin, and Yihe Huang—I look forward to the many adventures awaiting us.

Contents

List of Figures

List of TODOs

Chapter 1

Introduction

Users have tens to hundreds of accounts with web services that store sensitive data, from social media to tax preparation and e-commerce sites [**tens**, **hundreds**, **password_life_cycle**]. While users have the right to delete their data (via e.g., the GDPR [**eu:gdpr**] or CCPA [**ccpa**]), more nuanced data controls often don't exist. For example, a user might wish to hide and protect their profiles on an e-commerce or dating app when inactive, and to recover their accounts should they return to use the application. However, services often provide only coarse-grained, blunt tools such as permanent account deletion that result in all-or-nothing exposure of users' private data

This thesis introduces the notion of *disguised data* for flexible data privacy. Disguised data is a reversible state of data in which sensitive data is selectively hidden. To demonstrate the feasibility of disguised data, this thesis also presents Edna—the first system for disguised data—which helps database-backed web applications allow users to remove their data without permanently losing their accounts, anonymize their old data, and selectively dissociate personal data from public profiles. Edna lets developers support these features while maintaining application functionality and referential integrity via *disguising* and *revealing* transformations. Disguising selectively renders user data inaccessible via encryption, and revealing enables the user to restore their data to the application.

This chapter motivates the need for data disguising for flexible privacy; describes the challenges in disguising and revealing data; introduces our approach to achieve data disguising; and summarizes the contributions of this thesis.

1.1 Motivation

User today lack fine-grained controls over their personal data stored by web applications. Consider Twitter: after a change in management [**musk-twitter**], many users wanted to leave the platform and try out alternatives (e.g., Mastodon [**mastadon**]). But each user faced a tricky question: should they keep their Twitter account, or should they delete it? Advice on how to quit Twitter [**quit-twitter-india**, **quit-twitter-mash**] highlight how keeping an inactive account

leaves sensitive information (e.g., private messages) vulnerable on Twitter's servers; but deleting the account prevents the user from changing their mind and coming back, causing them to permanently lose all their followers and content. Hence, many users left Twitter but kept their accounts [**nbc-twitter**, **shondarhimes**, **kenolin**]. A better solution would let users temporarily revoke Twitter's access to their data while having the option to come back.

Similarly, users give dating apps personal data, and frequently deactivate and reactivate their accounts. This sensitive data should be protected from the application and potential data breaches [**tinder**, **okcupid**] when a user deactivates their account, but be readily available when they choose to return.

Users may also prefer old data, such as past purchases in an online store or their passport details with a hotel, to be inaccessible to the service after some time of inactivity, and therefore protected from leaks or service compromises [**retention**, **breach:marriott**]. Or users may prefer to—explicitly or automatically—dissociate their identity from old data, such as teenage social media posts or old reviews on HotCRP [**hotcrp**]. Today, users work around the lack of such support by explicitly maintaining multiple identities (e.g., Reddit throwaway accounts [**reddit:throwaway**] and Instagram "finstas" [**nytimes:finsta**]), an inflexible and laborious solution.

Providing this fine-grained privacy functionality can benefit both the service and the user. It helps the service comply with privacy regulations, reduces its liability on data breaches, and appeals to privacy-conscious users; meanwhile, the user can rest assured that their privacy is protected, but can also get their data back and reveal their association with it if they want.

1.2 Disguised Data: A New Abstraction for Flexible Privacy

This desired flexible privacy functionality describes an in-between state of users data in web applications. The data is not quite gone, because a user can return to the application and restore their data; but the user data is also not quite there, because some or all of it has been removed or made inaccessible to the application.

To capture this new state, this thesis introduces the new abstraction of *disguised data*. Disguised data represents a state of data where (*i*) some or all of the user's original sensitive data is rendered inaccessible to the application; (*ii*) some data may be replaced with placeholders to keep the application structure intact (e.g., placeholder parent comments to maintain comment thread structure); and (*iii*) the data can be restored with the user's authorization.

Systems for disguised data move closer to an Internet where users can leave services and return at any time, where old data on servers is protected by default, and where services provide users with control over their identifying data visible to the service and other users.

1.3 Challenges

Today, disguised data for flexible privacy remains out of reach for users of web applications in part because getting it right is hard for application developers. In particular, developers face four main challenges.

First, real applications store their data in databases with complex schemas and have complex notions of privacy, data ownership, and data sharing. Simple solutions that might try to delete all data associated with a user can create orphaned data or break referential integrity, which is the invariant that if a table row references another row via a foreign key, the referenced row must exist. In order to resolve these problems, developers must change the application to handle these situations and maintain correctness. These solutions also lack support for users to return. To solve this manually, a developer would have to carefully perform application-specific database changes to remove data, store any data removed to be able to later restore it, and correctly revert the database changes on restoring.

Second, to manually realized disguised data, developers would also have to reason about interactions between multiple data-redacting features, and how these features would compose. For example, imagine an application that supports both account deletion and anonymizing old data: if a user wants to delete all their posts after they have been anonymized, a SQL query must somehow determine which anonymized posts belong to the user in order to remove them. And if the user later wants to return and restore their posts, the developer must account for the applied anonymization and restore the user's posts as anonymized.

Third, stored removed data should be protected against data breaches, but accessible if the user chooses to return. The developer also needs to provide user-friendly ways for users to prove their ownership of the stored data.

Fourth, application databases often undergo global updates such as schema migrations and data transformations like content reformatting. However, these updates would fail to affect any data disguised at the time, because the disguised data remains inaccessible to the application and developer. In order to restore disguised data correctly, these updates must be applied to disguise data when a user grants permission to access the disguised data in order to reveal it. For example, a disguised `posts` row might need to be restored as one `posts` and one `postContent` row after a schema migration moves post content into a different table from posts. The developer thus should remember and apply these global updates to disguised data in chronological order during reveal.

1.4 Our Approach

To realize disguised data, we present a general system that helps developers specify and apply two kinds of transformations: *disguising transformations*, which move the user's data into a disguised state; and *revealing transformations*, which restore the original data at a user's request. Disguising transformations aim to protect the confidentiality of users' disguised data (e.g., links

to throwaway accounts or old HotCRP reviews) even if the application is later compromised (e.g., via a SQL injection or a compromised admin's account). Only the user can invoke revealing transformations on their disguise data to authorize the application to restore it.

We demonstrate our approach in Edna, a system that realizes disguising and revealing transformations for database-backed web applications via a small set of expressive primitives that compose cleanly. lily got rid of well-defined. Developers specify the transformations that their application should provide, and Edna takes care of correctly applying, composing, and optionally reverting them, while maintaining application functionality and referential integrity.

Our approach for Edna's approach faces three challenges. First, Edna needs to present a simple, yet versatile interface for developers to specify disguising transformations. Edna addresses this challenge with a restricted programming model centered around three primitives: *remove*, *modify*, and *decorrelate* (which reassigns data to placeholder users). This model limits the potential for developer error, and lets Edna derive the correct disguising and revealing operations, while supporting a wide range of transformations.

Second, to work with existing applications in practice, Edna's disguising transformations should require minimal application modifications. To achieve this, Edna introduces *pseudoprincipals*, anonymous placeholder users that are inserted into the database on disguising and exist solely to own data decorrelated from real users and maintain referential integrity. Pseudoprincipals can also act as built-in "throwaway accounts," as they let the user disown data after-the-fact, as well as potentially later reassociate with it. To correctly reason about ownership when data may be decorrelated multiple times (e.g., by global anonymization after throwaways have been created), Edna maintains an encrypted *speaks-for chain* of pseudoprincipals that only the original user can unlock and modify.

Third, Edna needs to have access to the original data for users to be able to reveal their data and return to the application, but the whole point is to make that data inaccessible to the service. While Edna could ask users to store their own disguised data, this would be burdensome. Instead, Edna stores the disguised data on the server in encrypted form as *diff records*, and unlocks and restores data to the service only when a user provides their *reveal credentials* (e.g., a password or a private key).

1.5 Contributions

The main contribution of this thesis is the identification and exploration of *disguised data* for flexible user data privacy in web applications. As part of this, this thesis contributes:

1. The abstraction of *data disguising via disguising and revealing transformations*, including disguise specifications written with a small, composable set of data-anonymizing primitives (remove, modify, and decorrelate to pseudoprincipals), and reveal credentials to allow users to reveal their data (Ch. 3).
2. A design for Edna that realizes disguised data for practical web applications, based on

techniques such as diff records, speaks-for chains, and reveal-time update specifications. (Ch. 5).

3. A prototype Rust library implementing Edna for MySQL-backed web applications that implements user data control via disguising and revealing (Ch. 6).
4. Case studies that integrate Edna with three real-world web applications and demonstrate Edna's ability to enable composable and reversible transformations (Ch. 7).
5. An evaluation of Edna's effectiveness and performance, including how Edna contrasts with and complements related work (Qapla [**qapla**] and CryptDB [**cryptdb**]) (Ch. 8).

While disguised data can help developers to add more flexible user data controls, the abstraction and its implementation in Edna have some limitations. First, disguised data is a concept scoped for single applications, and does not tackle the problem of data sharing between services. Edna also assumes bug-free disguise specifications, and that applications use Edna correctly. Furthermore, Edna does not aim to protect undisguised data in the database against compromise; combining Edna with an encrypted database can add this protection. Finally, attacks to identify users from Edna's metadata (e.g., the size of stored disguised data) or placeholder data left in the database (e.g., embedded text) are out of scope.

The Edna prototype is open-source at `https://github.com/tslilyai/edna`.

1.6 Related Publications

The work described in this thesis has been previously covered in two peer-reviewed publications:

- **edna-hotos**
- **edna**

This thesis contains additional material on handling global database update operations such as schema migrations and content changes, described in §3.4 and §5.3, and evaluated in §8.6.

Chapter 2

Related Work

Edna addresses the problem of disguised data for flexible data privacy in web applications. Existing systems aim to support data deletion, prevent unauthorized data access, or protect against database server compromise, which are valuable, but complementary goals.

2.1 Compliance Tools

Compliance tools such as K9db [**k9db**] and GDPRizer [**gdprizer**] help correctly extract or delete data in response to compliance-related requests. These tools include systems to support wholesale user data extraction or deletion by tracking data ownership by modifying the data layout [**usershards**, **k9db**], tracking information flow [**schengendb**], or changing the storage hardware [**sdp**].

Tools like DELF at Meta [**delf**] focus on the related problem of data deletion, letting developers specify deletion policies via annotations on social graph edges and object types. DELF ensures correct cascading data deletion, both for compliance-related data deletion requests and for normal application deletions.

While one of the uses of disguising data is to provide GDPR-compliant account deletion, disguising also supports more nuanced use cases beyond simple deletion. For example, Edna allows users to return after deletion, hides old data for inactive users, or hides some but not all data so the user can continue using the application.

2.2 Policy Enforcement Systems

Policy enforcement systems aim to prevent unauthorized access to data and protect against leakage via compromised accounts or SQL injections. These systems enforce developer-specified visibility and access control policies via information flow control [**static**, **jeeves**, **jif**, **hails**, **ifdb**], authorized views [**oracle**], per-user views [**multiverse**], or by blocking or rewriting database queries [**blockaid**, **qapla**, **sieve**]. For example, a policy for a relational database-backed appli-

cation may allow the session user *U* to access account information only if the predicate "`WHERE accounts.user_id = U`" returns true; this prevents an attacker who injects a "`SELECT * FROM accounts`" query from reading any user accounts other than their own.

Policy-enforcing systems do not help users anonymize data or maintain application integrity constraints, which is the explicit goal with disguised data. By contrast, data disguising systems modify the database contents of sensitive data—the data under disguise—so the data is no longer available in the database, and thus unavailable even to the service itself. This is unlike policy enforcement systems, which can deny access to sensitive data, but still retains it in the database.

2.3 Encrypted Storage Systems

Encrypted storage systems such as CryptDB [**cryptdb**] and Mylar [**mylar**] protect against database server compromise, with some limitations [**grubbs**]. These systems encrypt data in the database, and ensure that only users with access to the right keys can decrypt the data. Applications must handle keys, and send queries either through trusted proxies that decrypt data [**cryptdb**], or move application functionality client-side [**mylar**]. Encrypted databases have orthogonal goals to systems for disguised data: while they protect data at all times against attackers who do not have the keys, encrypted databases do not help applications protect against data access by service itself, or by legitimate, authenticated users. Any user with legitimate access can view the data in an encrypted database, whereas disguised data is removed from the database and inaccessible to all users and the application without permission from the data's owner.

2.4 Other Related Work

Other work has also focused on protecting user data and giving users more control over their data's exposure. However, these systems differ from Edna in the specific problem they aim to solve, the threat model against which they protect, and how they can be deployed with today's applications.

Some platforms focus on enforcing user-defined policies, instead of the developer-defined policies that Edna supports. They prove that server-side processing respects user-defined data policies via cryptographic means [**zeph**] or systems security mechanisms [**riverbed**]. However, this may restrict feasible application functionality (e.g., to additively homomorphic functions, as in Zeph [**zeph**]), or restrict combining data with different policies, and requires modifying and redeploying existing applications. For example, Riverbed [**riverbed**] requires applications to run proxies on both client and server, allowing the client proxy to attest that the server indeed runs the correct proxy. Furthermore, users in Riverbed who define different policies cannot easily share data with random other users at will, and aggregations over many users' data become difficult.

Systems like Pesos [**pesos**], Ironsafe [**ironsafe**], and Ryoan [**ryoan**] use trusted execution environments to protect all user data and computation on this data against untrusted storage platforms. Ryoan also runs data processing applications as sandboxed modules within TEEs, thus protecting user data against untrusted applications in addition to untrusted platforms. By contrast, Edna protects only disguised data from the application and database server, and trusts developers to be well-intentioned when writing disguise specifications.

Decentralized platforms such as Solid [**solid**], BSTORE [**bstore**], Databox [**databox**], and others [**diy**, **amber**, **oort**, **w5**, **blockstack**] put data directly under user control, since users store their own data. But decentralized platforms burden users with maintaining infrastructure, lack the capacity for server-side compute, and break today's ad-based business model. By contrast, Edna leaves the data and business models unchanged, and stores all data, including disguised data, on the application's servers.

Devices using iOS [**applesecurity**], Android [**applesecurity**], or CleanOS [**cleanos**] revoke data access via encryption, like Edna does. However, these systems operate in settings that store only a single user's data; disguised data applies in settings that include multiple users' data and shared data.

Vanish [**vanish**] provides users with self-destructing data and a proof of data deletion using decentralized infrastructure and cryptographic techniques (with limitations against Sybil attacks [**defeat_vanish**]). Unlike Edna, Vanish cannot restore deleted data and requires substantial application restructuring to deploy (e.g., the application must run a Vanish daemon, be aware of which objects are "vanishing objects," and store and access protected data as key-value pairs instead of objects of other data structures).

Sypse [**sypse**] pseudonymizes user data and partitions personally identifying information (PII) from other data. Instead of partitioning data, Edna modifies the database and stores disguised data encryptedly.

Finally, oblivious object stores like Dory [**dory**], Snoopy [**snoopy**] and Obladi [**obladi**] protect data and search access patterns against powerful adversaries who can, for example, compromise the entire cloud software stack and view metadata like network traffic and access patterns. To provide strong security, these systems rely on complex encryption schemes, oblivious RAM, hardware enclaves, or other cryptographic techniques. However, oblivious object stores support only data retrieval and simple queries (e.g., key/value point queries), and expect clients to perform most computations on data. By contrast, Edna supports arbitrary server-side processing of undisguised data, which most applications today require. Furthermore, Snoopy—the most recent system—achieves performance 39.1× slower than Redis, a commonly-used non-oblivious object store. This slowdown is impractical for many applications.

Chapter 3

Overview

This section introduces the concept of disguised data by describing how developers and users interact with an application that supports data disguising and data revealing.

Edna helps developers realize new options for users to control their data via *disguising transformations* and *revealing transformations*. A developer integrates an application with Edna by writing disguise specifications and adding hooks to disguise or reveal data using Edna's API (Figure 3-1). This proceeds as follows:

(1) An application registers users with a public–private keypair that either the application or the user's client generates. Edna stores the public key in its database, while the user retains the private key for use in future reveal operations.

(2) When the application wants to disguise some data, it invokes Edna with the corresponding developer-provided disguise specification and any necessary parameters (such as a user ID). Disguise specifications can remove data, modify data (replacing some or all of its contents with placeholder values), or decorrelate data, replacing links to users with links to pseudoprincipals (fake users). Edna takes the data it removed or replaced and the connections between the user and any pseudoprincipals it created, encrypts that data with the user's public key, and stores the resulting ciphertext—the *disguised data*—such that it cannot be linked back to the user without the user's private key.

(4) When a user wishes to reveal their disguised data, they pass credentials to the application, which calls into Edna to reveal the data. Credentials are application-specific: users may either provide their private key or other credentials sufficient for Edna to re-derive the private key. Edna reads the disguised data and decrypts it, undoing the changes to the application database that disguising introduced.

Edna provides the developer with sensible default disguising and revealing semantics (e.g., revealing makes sure not to overwrite changes made since disguising).

3.1 Example: Lobsters Topic Anonymization

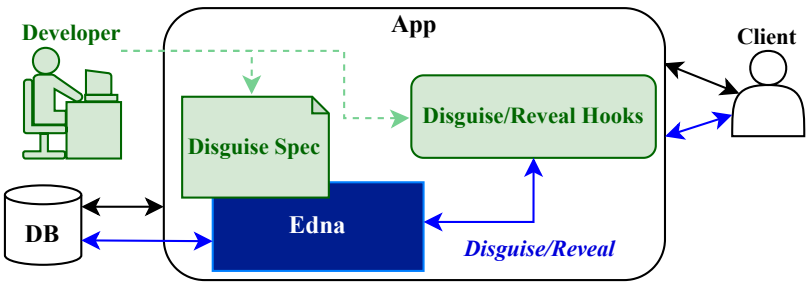
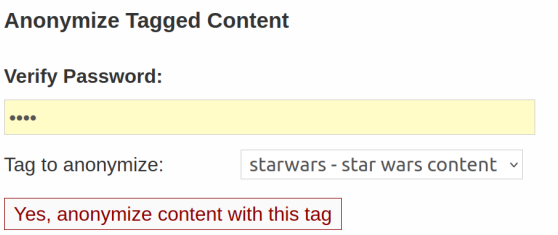


Figure 3-1: Developers write disguise specifications and add hooks to invoke Edna from the application (green); in normal operation, clients use these hooks in the application to disguise and reveal their data in the database (blue).




```
def tag_anon
  # instantiate the disguise spec with the provided tag to anonymize
  disg_spec = edna.instantiate_spec("tag_anon.json",params[:tag])
  # apply the disguising transformation
  disg_id = edna.apply_disguise(@user.id,params[:passwd],disg_spec)
  # email the disguise ID to the user to allow revealing
  SendDisguiseEmail(@user, disg_id)
end
```

Figure 3-2: The Lobsters developer adds a hook in the UI and code to perform topic-based anonymization.

We illustrate how Edna's API and disguise specifications work via a disguising transformation for Lobsters [**lobsters**]. Lobsters is a link-sharing and discussion platform with 15.4k users. Its database schema consists of stories, tags on stories, comments, votes, private messages, user accounts, and other user-associated metadata. Users create accounts, submit URLs as stories, and interact with other users and their posted stories via comment threads and votes.

Consider **topic-based anonymization**, a privacy feature that allows users to hide their interest in a topic (a "tag" in Lobsters) by decorrelating their comments and removing their votes on stories with that tag. For instance, a Lobsters user Bea who posts about their interests—Rust, static analysis, and Star Wars—might want to hide associations with Star Wars before sharing their profile with potential employers. This is currently not possible in Lobsters.

The Lobsters developer can realize topic-based anonymization as a disguising transformation. First, the developer writes a disguise specification that instructs Edna to decorrelate comments and remove votes on "Star Wars" stories (Figure 3-3), and provides it to Edna. They also add frontend code and UI elements that allow authenticated users to trigger the disguising transformation (Figure 3-2). When Bea wants to anonymize their contributions on content

```
// Decorrelate comments on stories w/tag {{TAG}}
"comments": [{
  "type": "Decorrelate",
  "predicate": "tags.tag = {{TAG}}",
  "from": "comments JOIN stories ON comments.story_id = stories.id
           JOIN taggings ON stories.id = taggings.story_id
           JOIN tags ON...",
  // decorrelate-specific fields
  "group_by": "stories.id",
  "principal_fk": "comments.user_id" } ],
// Remove votes on stories w/tag {{TAG}}
"votes": [{
  "type": "Remove",
  "predicate": "tags.tag = {{TAG}}",
  "from": "votes JOIN stories...",
}, ... ]
```

Figure 3-3: Lobsters topic-based anonymization disguise specification (JSON pseudocode), which decorrelates comments and removes votes on stories with the specific topic tag.

tagged "Star Wars," Lobsters invokes Edna with the provided specification.

When Bea wants to deanonymize their "Star Wars" contributions, Lobsters invokes Edna with Bea's reveal credentials and the disguise ID, and Edna reveals Bea's associations with "Star Wars" posts.

3.2 Disguise Specifications

Edna's developer interface ensures that developers need to reason only about the high-level semantics of disguising expressed in the disguise specification. Edna takes care of applying disguise transformations, composing them with other disguising and revealing transformations, and creating and storing disguised data.

Disguise specifications tell Edna what application data objects to disguise and how to disguise them. A disguise specification (Figure 3-3) identifies objects to disguise by database table name and predicate, where a predicate is a SQL `WHERE` clause. Edna by default disguises all objects related to the given principal[1] to disguise, as defined by a foreign-key relationship to the principals table, but predicates can narrow the transformation's scope (e.g., to stories with specific tags). For each selected group of objects, developers choose to *remove*, *modify*, or *decorrelate* them.

The example specification in Figure 3-3 decorrelates all comments and removes all votes on stories with a particular tag, specified by the `TAG` parameter provided at invocation time. The specified foreign key `principal_fk`) for decorrelation of comments tells the decorrelate operation which foreign key to the principals table to rewrite. Decorrelation can use pseudoprincipals at different granularities. In the extreme, the disguise specification may tell Edna

[1]A principal refers to an application object representing a user (e.g., a row in a `users` table). This thesis uses principal to mean either an application object representing a real application client user, or a pseudoprincipal (a fake application user).

to create a unique pseudoprincipal for each decorrelated application object. In our example, however, all comments by the same user on the same story decorrelate to the same pseudoprincipal (`"group_by": "stories.id"`), thus keeping same-story comment threads intact. §3.2 describes the semantics of disguise specifications in more detail.

3.3 User Reveal Credentials

Edna provides developers and users with the abstraction of *reveal credentials*, which users provide to the application to authorize revealing their data. Reveal credentials allow applications that use Edna to support familiar user authentication workflows when users want to reveal their disguised data.

Edna supports two forms of reveal credentials: (*i*) the user's private key itself; or (*ii*) the user's application password or a recovery token (in case they forget their password), either of which Edna can use to rederive the private key. Developers can use either or both of these credentials depending on application needs. In the Lobsters example, Edna rederives the user's private key using their password. Password or keypair changes require an application to re-register the user with Edna, which generates new recovery tokens and re-encrypts the user's disguised data.

3.4 Global Database Updates

Web application databases often undergo global updates initiated by an admin or developer, like schema changes or global content changes (e.g., normalizing URLs of all stories). However, developer updates to the database will fail to apply to any disguised data at the time. Edna may thus incorrectly reveal data that do not reflect the updates. To ensure updates are applied to revealed data, the developer must inform Edna about the update so Edna can apply the update to the disguised data when revealing it.

When a database update is performed, the developer writes a corresponding *reveal-time update specification* that captures the effects of the operation and invokes Edna's API with the update specification. Edna stores these update specifications in an *update replay log* that Edna reads upon reveal.

Edna expects developers to provide a reveal-time update specification that takes a user's disguised data rows and produces updated versions of disguised data rows. Oftentimes, these updates may be a subroutine of the global database update. For example, if a developer wants to normalize all URLs in Lobsters `stories`, they might first read all stories, then normalize the URL in each story individually, and finally `INSERT...UPDATE` the normalized stories. An update specification given to Edna could just capture the normalization step, producing normalized URL stories for the stories in a user's disguised data.

In other situations, no such subroutine of the global update exists because the global update

uses queries that apply to entire database tables instead of rows. In these scenarios, the developer must write a separate update specification for updating disguised data rows. For example, Lobsters has a schema migration that adds a column to the `users` table. The global update performs an `ALTER TABLE` query, while the update specification given to Edna receives user rows without the column as input, and outputs user rows with the column.

Edna assumes that any nondeterminism or side effects from performing the update specification will maintain application correctness. Any pure functions deterministic in their input (placeholder or disguised rows) and do not have side effects are thus supported as update specifications.

To exemplify an unsupported update, take a global update that calculates the current vote count of a story to store as a story's `vote_count` attribute. This update thus depends on other disguised rows, namely `vote` rows. If a disguise has removed both a user's story and its votes, then when Edna reveals the disguise, Edna will first reveal disguised stories before revealing disguised votes in order to maintain referential integrity. But when applying a `vote_count` update specification to stories, Edna will find 0 votes for all of them, because their votes have not yet been restored. Edna supports this update only if this 0-vote state counts as correct application behavior.

3.5 Guarantees and Threat Model

Edna protects the confidentiality of disguised data between the time when a user disguises their data and the time when they reveal it. During this period, Edna ensures that the application cannot learn the contents of disguised data, nor learn what disguised data corresponds to which user, even if the application is compromised and an attacker dumps the database contents (e.g., via SQL injection). Edna encrypts disguised data, so its confidentiality stems from "crypto shredding," a GDPR-compliant data deletion approach based on the fact that ciphertexts are indistinguishable from garbage data if the key material is unavailable [**dnefs**, **townsend:cryptoshredding**, **aws:cryptoshredding**, **gtr:cryptoshredding**]. might depend on private key being separate from password?

We make standard assumptions about the security of cryptographic primitives: attackers cannot break encryption, and keys stored with clients are safe. If a compromised application obtains a user's credentials, either because the user provides them to the application for reveal, or via external means such as phishing, Edna provides no guarantees about the user's current or future disguised data. Edna also expects the application to protect backups created prior to disguising, and external copies of the data (e.g., Internet Archive or screenshots) are out of scope.

While Edna hides the contents of disguised data and relationships between disguised data and users, it does not hide the existence of disguised data. (An attacker can see if a user has disguised some data, but cannot see which disguised data corresponds to this user.) An attacker

can also see any data left in the database, such as pseudoprincipal data or embedded text. Edna puts out of scope attacks that leverage this leftover data and metadata to infer which principal originally owned which objects.

Edna's choice of threat model and its limitations stem from Edna's goal of practicality and usability by existing applications. For example, decorrelation with pseudoprincipals removes explicit user-content links, but leaves placeholder information in the database to avoid application code having to handle dangling references. Similarly, leveraging server-side storage to hold disguised data leaves metadata available to attackers, but avoids burdening users with data storage management. We believe a stronger threat model would require more modification of applications.

Chapter 4

API Semantics

This chapter describes Edna's high-level semantics that a developer can use to reason about the resulting state of data after a disguise and reveal. Following this high-level description comes a deeper dive into Edna's API and examples of how developers can use it.

4.1 High-Level Semantics

4.1.1 Disguising and Revealing

Developers can think about disguising transformations as removing and rewriting data in the application database. The database starts at some original application state, and alters application data depending on the disguise specification (§4.1.2). The resulting state—the disguised data state—is changed only by another disguise, by revealing the applied disguise, or by normal application updates.

Disguising already-disguised data can only decrease the amount of information retained about the original application data state. Removed data cannot be disguised again. If the original data has been rewritten by prior disguises, future disguises of the disguised state have access only to the disguised state, and thus can only further remove aspects of the original data state.

Edna applies disguises on behalf of either a single user or all users. However, only a user can reveal a disguise on their data, and must provide a reveal credential to `RevealData`.

Revealing a disguise D on some data disguised by D reverts the changes to the data applied by D back to the pre-disguised state, except in two cases. First, if a normal application update has since changed the disguised data, then the data remains in its updated state and cannot be revealed. Second, if another disguise has since applied changes to the data D disguised, and that disguise has not yet been revealed, then the data remains in its current state. Only when all subsequent disguises are revealed does the data revert to the state prior to D. Thus, when multiple disguises apply to the same data, the application data state reflects the effects of all yet-unrevealed disguises.

For example, if a post is disguised twice—once to modify the timestamp, and again to scrub

its content—then revealing only one disguise will restore only that attribute (e.g., a timestamp or the post content). This also applies for disguises that rewrite the same data attributes. For example, if one disguise decorrelates a post from its author to "AnonPig" (i.e., by rewriting its `author` foreign key to the "AnonPig" pseudoprincipal) and another disguise decorrelates the post again to "AnonDog," the post will be correlated with "AnonDog" even if the first disguise decorrelating to "AnonPig" is revealed. And if the second disguise decorrelating to "AnonDog" were revealed first, the data would remain decorrelated to "AnonPig" until the user also reveals that first decorrelating disguise. example figures for both of these.

Revealing transformations also apply a developer's provided reveal-time updates (§4.2.5) to the data to reveal, ensuring that revealed data reflects global application updates since the time of disguise.

4.1.2 Disguise Specifications

A developer describes the effects of a disguise (and undone by a reveal) via disguise specifications. As previously shown in Figure 3-3, each specification operation consists of the disguise operation type, the database table name, and a SQL `WHERE` predicate.

A disguise is invoked either automatically by the application, or by a specific user of the application identified by their user ID. If disguise is user-specific, then only data that has a foreign key to that user's identifier (is "owned" by that user) and matches the disguise specification's predicate will be disguised. Otherwise, the disguise applies to all data matching the disguise specification's predicate.

This section next dives into the semantics of each disguise operation type.

Remove. A remove operation removes the entire row that matches the operation's predicate. Developers should take care to handle referential integrity to removed rows, as Edna will not cascade-delete referencing rows or introduce placeholders. (Developers should instead use decorrelate operations to remove principal rows and replace them with pseudoprincipals).

A reveal of a remove operation will reinsert all removed rows, unless reinserting the row will violate a primary key or uniqueness constraint of the application.

Modify. A modify operation works at per-column granularity, and requires developers to specify a modification policy for each column to modify in addition to the table name and predicate. The modification policy informs Edna how to generate new placeholder values for each column to modify using one of Edna's value generation policies. The current prototype supports constants (e.g., "removed"), random values (e.g., a random email address), and values derived from the prior value (e.g., a redacted phone number with only area code visible).

Revealing a modify operation restores a row's column back to its state prior to the disguise *only if* the current column value matches the value generated by the modify operation during the disguise. This prevents a reveal from overwriting application changes to the column value

since the time of disguise. Some rows may thus end up partially revealed, with only some modified columns restored back to the original pre-disguise state. Developrs can set a (global) flag to disable partial row reveals, to prevent revealing of any column values in a row with at least one conflicting column value.

Decorrelate. A decorrelate operation requires developers to additionally specify (*i*) which foreign keys for the table rows to rewrite, and (*ii*) a `group_by` attribute if rows with the same value for that attribute should refer to the same pseudoprincipal after decorrelation. A decorrelate operation only applies to rows with foreign key relationships to the principals table (if specified on other foreign keys, the operation will do nothing). Developers also provide a global pseudoprincipal-generation policy (using Edna's value generation policies) to tell Edna how to generate per-column placeholder values.

If a user invokes the disguise, then Edna only decorrelates the specified foreign key attributes that refer to that user's identifier, and whose rows match the predicate. Otherwise, if the disguise applies over all users, then Edna decorrelates all specified foreign keys for all rows matching the predicate.

For all rows to decorrelate with the same `group_by` attribute value, Edna rewrites the foreign keys to decorrelate to the same randomly generated pseudoprincipal. If no `group_by` attribute is specified, each row gets a unique pseudoprincipal. Thus, no two disguises share the same pseudoprincipals, and no two tables share the same pseudoprincipals.

Revealing Decorrelate. Pseudoprincipals may acquire new references from application objects inserted after the time of disguise—for example, a decorrelated comment might have new responses. To ensure that revealing a decorrelate operation—which deletes pseudoprincipals—preserves referential integrity, developers also inform Edna how to, during reveal, handle these objects added after disguising that refer pseudoprincipals of the disguise. Developers choose between three options for pseudoprincipal-referring objects: (*i*) change the object's reference to point to the original principal (`RECORRELATE`); (*ii*) delete the object (`REMOVE`); or (*iii*) continue referring to the pseudoprincipal (`RETAIN`). figure for each option

This option is global; another choice of API could support a menu of options, such as per-table checks and fixes (where the developer specifies per-table policies) or per-inserted-object ones (where the developer makes application modifications to log all added references to pseudoprincipals).

4.1.3 Shared Data.

Many applications support shared data; in Lobsters, for example, messages between users are owned by both users. Edna's default semantics for shared data implement an ownership model inspired by a common treatment of shared data as jointly owned. When a user's disguise removes shared data, Edna decorrelates the data from the disguising user, but preserves the data

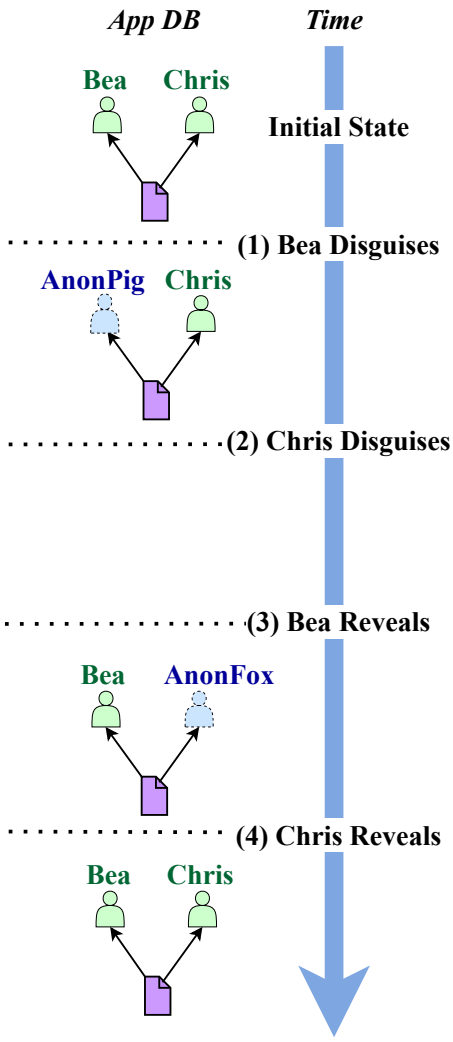


Figure 4-1: Edna supports joint ownership semantics, where shared data is not removed until all users have disguised their accounts. Owners can return in any order, and the message remains decorrelated from unrevealed owners.

and its association with other owners. Edna removes the data once all users have disguised it and all ownership links are to pseudoprincipals. Any owner can reveal the message, which restores the message to the database if it does not exist, and recorrelates only the revealing user; all disguised owners remain decorrelated as pseudoprincipals until they also choose to reveal the message. Regardless of the reveal order, if all owners reveal the message, Edna returns the message to its original state.

For each shared data object, each owner has an associated unique pseudoprincipal. If ony some owners have disguised the row, the row will refer to the disguised owners' unique pseudoprincipals' identifiers instead of the owners'. When a user reveals a shared data row, all users still disguised remain associated with their unique pseudoprincipal (which can be reused over multiple remove disguises).

For instance, consider the scenario in Figure 4-1 where Bea and Chris share a Lobsters message. Bea maps to unique pseudoprincipal AnonPig for this shared message, and Chris to AnonFox.

(1) When Bea disguises the message, the message is owned by Chris and a pseudoprincipal;
(2) If Chris then disguises the message, Edna removes it.
(3) If Bea reveals the message, this restores the message to the database and recorrelates only Bea; Chris remains decorrelated as a pseudoprincipal.
(4) When Chris reveals the message, Edna returns the message to its original state.

§5.4 dives deeper in how Edna's design achieves these shared data semantics.

4.1.4 Composition of Disguises.

Once Edna has decorrelated some user's data to a pseudoprincipal, future disguises applied by the user will fail to further disguise that data. If a user wants to further disguise their pseudoprincipal's data, then the user can grant Edna permission to reveal links to their pseudoprincipal by providing a reveal credential (their password, private key, or backup token). Edna will then treat all of the user's pseudoprincipals' data as if they belong to the user themself, and depending on the disguise specification, will remove, modify, or create a new pseudoprincipal owner for these pseudoprincipals' data. To reveal their data back to its original state, a user must reveal all disguises composed atop the data. §5.5 fleshes out Edna's design for disguise composition.

4.2 API Developer Manual

Developers add hooks to their application to invoke Edna's API, which consists of the functions shown in Figure 4-2. This section describes each function and how developers would use them to support disguising and revealing.

4.2.1 `RegisterPrincipal`

Registers an application user as a principal whose data can be disguised and revealed. Unique users should not be registered multiple times. Once registered, data of an undeleted user that exists in the database can always be disguised and revealed. Users deleted by a disguise do not need to reregister if revealed.

Arguments. Takes the user's associated unique identifier and unique public-private keypair (generated by the user client or application). The provided public key is used to encrypt the user's disguised data. The provided private key and password enable clients to later reveal data using either the private key or password.

```
// Generates keypair for p and returns the user's backup credential
RegisterPrincipal(
        p: UID,
        pw: Password,
        pubkey: PublicKey,
        privkey: PrivKey)
    -> RevealCredential;

// disguises principal p according to the spec,
// optionally over already-disguised data (§5.5)
DisguiseData(
        p: Option<UID>,
        spec: DisguiseSpec,
        principal_gen: PrincipalGenerator,
        schema: Schema,
        disguise_over: Option<RevealCredential>)
    -> disguiseID;

// Reveals data disguised by s for p with p's password.
RevealData(
        p: Option<UID>,
        did: disguiseID,
        pp_ref_policy: PseudoprincipalReferencePolicy,
        allow_partial_row_reveal: bool,
        schema: Schema,
        cred: RevealCredential>)
    -> bool;

// Gets principals that p can speak-for.
CanSpeakFor(p: UID, cred: RevealCredential) -> Vec<UID>;

// Records a reveal-time update spec in the replay log.
RecordUpdate(update_spec: RevealTimeUpdateSpec) -> bool;
```

Figure 4-2: Edna's API (Rust-like syntax).

Return Value. The function returns a backup token that the application should return to the user client. This token enables the user to reveal in the case they lose their password or private key.

4.2.2 `DisguiseData`

example effects of disguises—or add examples in remove/modify/decorrelate sections?

Removes or rewrites application data according to the provided disguise specification (i.e., disguises data). The original data is encrypted and stored within the application database; a user must provide credentials to invoke with a `RevealData` call in order to restore the data.

This function may insert pseudoprincipals (anonymous users) into the application database if a decorrelation—a rewriting of a foreign key to point to an anonymous principal instead of an original principal user—is specified by the disguise. Data disguised within the same disguise and from the same table may refer to (via a foreign key) the same pseudoprincipal. No two objects from different database tables refer to the same pseudoprincipal, and no two disguises'

sets of produced pseudoprincipals overlap.

Arguments. A disguise is invoked either automatically by the application, or a specific user of the application identified by user ID. If the disguise is invoked on behalf of a specific user, then all predicates on specified disguise operations include an "`AND [fk_col] = [uid]`" clause for all foreign keys to the application's principals table.

If a user invokes a disguise through the application and provides a reveal credential (their password, private key, or backup token), the function will disguise data that has already been disguised (§5.5). Depending on the disguise specification, this will remove, modify, or create a new pseudoprincipal owner for data with a foreign key to a pseudoprincipal instead of the user.

The developer provides three arguments:

- A disguise specification that describes how to disguise data by removing, modifying (replacing some or all of its contents with placeholder values), or decorrelating, replacing links to users with links to pseudoprincipals.
- A principal generator, which describes how to create a pseudoprincipal in the application (global across all disguises).
- The database schema, which specifies ownership links from data tables to user tables via foreign key relationships (global across all disguises).

Return Value. The function returns a unique disguise ID for the applied disguise, which should be returned to affected users in case they wish to later reveal the disguise.

Shared Data. repetition? If rows to remove are shared among multiple users, they are only deleted from the database once all users have disguised the row. This occurs if (*i*) all users individually disguise the data, or (*ii*) the disguise is applied over all users (`p = None`).

If the row is not removed (some owning users have not disguised the row), the row will refer to a pseudoprincipal for users who have disguised.

4.2.3 `RevealData`

example effects of reveals—or add examples in remove/modify/decorrelate sections?

Restores data disguised by the disguise corresponding to the provided ID to the database. Revealing the same disguise ID multiple times will do nothing after the first reveal.

If revealing a row violates database constraints specified in the schema (e.g., uniqueness, referential integrity, NULl checks), `RevealData` will not restore that row. `RevealData` may fail to restore some rows, but will still reveal others that successfully pass all constraints checks.

Revealing a row operates on a per-attribute granularity. If an attribute of a row at the time of reveal differs from the attribute value set when `DisguiseData` disguised the row, then that attribute of the row will not be restored to the original value. This prevents `RevealData` from

overwriting an application change since the time of disguise. `RevealData` will still reveal matching attributes for the row unless developers diable partial row removal (`allow_partial_row_reveal`).

Revealed data will reflect all of the updates registered via `RecordUpdate` since the time of disguise `did`, applied in chronological order.

Arguments. A reveal is invoked by a specific user of the application identified by user ID. The user also provides a reveal credential (their password, private key, or backup token) and the identifier for the disguise to reveal.

The developer provides three arguments:

- A pseudoprincipal reference policy (`RECORRELATE`, `REMOVE`, or `RETAIN`) to ensure that reveals of decorrelations preserve referential integrity. `RECORRELATE` ensures that any table rows that have been added since the time of disguise, and which have foreign key references to a pseudoprincipal row that is removed by the reveal, will reference the revealed user replacing the pseudoprincipal. `REMOVE` removes such new references to removed pseudoprincipal rows, and `RETAIN` prevents `RevealData` from the removing pseudoprincipal row and retains any new references to it.
- A flag specifying whether a row can be partially restored if some attributes at the time of reveal differ from the attribute values set when disguising the row.
- The database schema, which specifies ownership links from data tables to user tables via foreign key relationships (global across all disguises).

Return Value. Returns true if all rows to reveal pass database constraint checks, and false otherwise.

Shared Data. repetition here too?

4.2.4 `CanSpeakFor`

example use of canspeakfor for editing data?

Finds all principals in the application identified by user ID that stem from a (potentially recursive) decorrelation of user `p`. User `p` is thus authorized to speak-for any of these principals.

Arguments. `CanSpeakFor` is invoked on behalf of some user with UID `p`, and with the user's password.

Return Value. A list of all user IDs of principals that `p` can speak-for.

4.2.5 `RecordUpdate`

example use of recordupdate (take from design)

Enables a developer to perform updates to disguised data prior to revealing it. Invoking `RecordUpdate` timestamps and logs a developer-provided update specification—a data transformation function—to reflect a global update performed on undisguised data. All updates since the time of disguise will be performed in chronological order on disguised data prior to revealing it.

If the provided reveal-time update specification relies on data external to the disguised data to reveal (e.g., the current time or state of undisguised data), Edna cannot ensure correctness when applying the update to disguised data at reveal time.

Arguments. Takes a reveal-time update specification reflecting the data transformations performed on table rows. The specification maps a set of table rows to a set of updated table rows.

Return Value. Returns true on successful recording in Edna's log, and false on failure.

4.3 Edna Without Encryption

Developers may find Edna's threat model unnecessarily strong: perhaps they do not worry about data breaches, care about regulations like the GDPR, or trust their application code to not expose disguised data. Instead, these developers may want to add disuising and revealing without encryption.

A subset of Edna's API suffices to provide the same semantics for disguises without encryption. Users do not need to register with Edna; applications can use normal user authentication to authorize revealing; disguising already-disguised data no longer requires a user's reveal credential; and reveal-time updates can be applied immediately to disguised data instead of logged and applied at reveal time. §5.9 describes how removing encryption changes would change Edna's design.

Chapter 5

Edna's Design

This chapter describes the design of Edna and the techniques Edna uses to support disguising and revealing. We first look at how Edna performs disguising transformations, and then dive into the details of how Edna reveals disguised data. We then describe the more complicated scenarios possible with Edna, namely disguise composition, shared data, and action as pseudoprincipals. This chapter concludes with an analysis of the security of Edna's design with respect to the threat model described in §3.5.

5.1 Disguising

To apply a disguising transformation, Edna creates a unique *disguise ID* and queries for the data to disguise based on the disguise specification predicates. To preserve referential integrity, Edna constructs a dependency graph between tables based on foreign key relations (assuming no circularity), and first performs removes from tables in topologically-sorted order. Edna then performs decorrelations and modifications in specification order, potentially generating and storing pseudoprincipals.

Next, to record disguised data, Edna generates *diff records* that contain (*i*) the original data row(s), and (*ii*) placeholder data rows the disguise inserted or rewrote in the application (e.g., pseudoprincipals or the value of any modified columns). All types of diff records also contain the disguise ID. The original and placeholder rows contained by a diff record varies by disguise operation as follows:

- Remove diff records contain the removed original row and no placeholder rows;
- Modify diff records contain the unmodified row and the row with the modified value; and
- Decorrelate diff records contain the referencing row with the original foreign key value and—as placeholder data—the referencing row with the placeholder foreign key value pointing to a pseudoprincipal and the pseudoprincipal's row.

For each new pseudoprincipal, Edna generates a public–private keypair and an encrypted *speaks-for record*, which adds a link to the original principal's *speaks-for chain*. A speaks-for record contains a pair of (original principal, pseudoprincipal) IDs and the pseudoprincipal's private

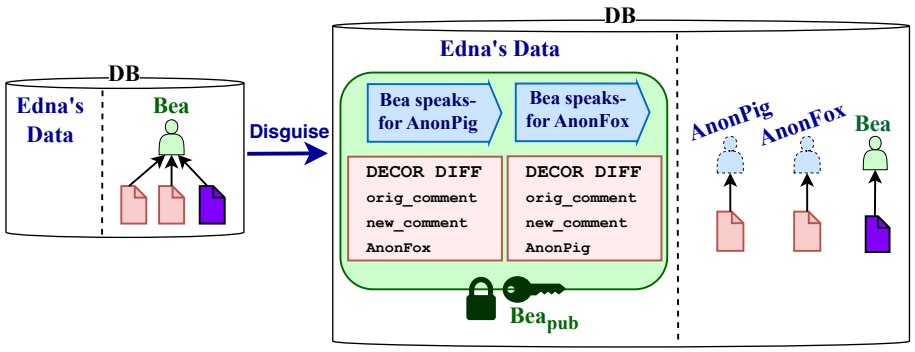


Figure 5-1: When Edna applies topic-based anonymization to Bea's comments on stories tagged "Star Wars" (red), these comments are decorrelated to pseudoprincipals ("AnonPig", "AnonFox"). Edna stores encrypted speaks-for records mapping Bea to their pseudoprincipals, and diff records containing the comments with modified foreign keys.

key. Edna registers the pseudoprincipal with its public key to enable composition of disguises (§5.5). Edna then encrypts the diff and speaks-for records with the principal's key, and stores them in the database. Finally, Edna returns the disguise ID to the application. A client can use the disguise ID and the principal's credentials to reveal the transformation later.

To perform Bea's topic-based anonymization (Figure 5-1), which allows Bea to hide their association with a particular category of content, Edna thus: (*i*) queries the database to fetch comments and votes by Bea affiliated with "Star Wars"; (*ii*) creates a pseudoprincipal (e.g., "AnonFox") for every "Star Wars"-tagged story that Bea commented on, and inserts it as a new user; (*iii*) modifies the database by rewriting comment foreign keys to point to the created pseudoprincipals, and removing Bea's votes on those stories; (*iv*) creates new speaks-for records that map Bea to the created pseudoprincipals and add links to Bea's speaks-for chain; diff records containing Bea's votes on "Star Wars" stories; and diff records that document Bea's original ownership of "Star Wars"-tagged comments and the placeholder pseudoprincipal data; (*v*) encrypts the speaks-for and diff records with Bea's public key and stores them; and (*vi*) returns a unique disguise ID to the application.

Edna adds a *disguise table* and a *principal table* to the application database to store principals' disguised data. The disguise table contains lists of per-principal diff and speaks-for records encrypted with the principal's public key. The principal table is indexed by application user ID; each row contains the principal's public key, and a list of disguise table indexes encrypted with the public key. To store records for principal p, Edna (*i*) encrypts the records with p's public key; (*ii*) stores the ciphertext in the disguise table under index `idx`; (*iii*) encrypts `idx` (salted to prevent rainbow table attacks) with p's public key; and (*iv*) appends the encrypted `idx` to p's list of encrypted disguise tables indexes in the principal table. figure

This allows Edna to store records without needing access to the principal's private key, and to do so securely: the principal table adds a layer of indirection from user ID to encrypted records, so an attacker cannot link principals to their records. At reveal time, Edna can efficiently find

```
Reveal(disgID, uid, privkey):
 encrypted_disg_table_idxs := principal_table[uid]
 decrypted_disg_table_idxs :=
    decrypt(encrypted_disg_table_idxs, privkey)
 for idx in decrypted_disg_table_idxs:
   records = decrypt(disg_table[idx], privkey)
   for rec in records:
     if rec.disgID == disgID:
       // apply rec to application database
       // remove rec from disg_table
     else if rec.type == SPEAKS_FOR:
       // recursively reveal for pseudoprincipal
       // generated by another disguise
       Reveal(disgID, rec.pp_uid, rec.pp_privkey)
```

Figure 5-2: Pseudocode for revealing a disguising transformation while application principal `uid` exists. Recursive revealing (the `else` clause) walks the speaks-for chain to reveal composed records of pseudoprincipals created by other disguising transformations if necessary (§5.5).

disguised data for a given user by decrypting and using disguise table indexes in the principal table.

Disguising transformations may completely remove a principal from the application database. When this happens, Edna moves the corresponding list of encrypted disguise table indexes from the principal table to a *deleted principal table* indexed opaquely, e.g., by the public key. This removes the user ID from the database while allowing future reveal operations by the principal to find their disguise table indexes. Edna also stores a special type of remove diff record that contains the principal's ID and the principal's public key when a disguise removes a principal. This allows Edna to restore the principal to Edna's principal table if a user later reveals the disguise.

5.2 Revealing

To apply a reveal transformation, Edna first locates and decrypts the corresponding diff and speaks-for records using a disguise ID and the user's reveal credentials.

Edna's reveal procedure (Figure 5-2) first looks up all disguise records related to the provided reveal credentials via Edna's principal and disguise tables. Edna then applies diff records created for the `disgID` disguise transformation to the database, thus restoring the relevant application objects to their pre-disguised state.

To preserve referential integrity, Edna restores disguised data that was removed from tables in topologically-sorted order (constructing the dependency graph as in §5.1) Edna then reveals any modifications, and finally performs recorrelations. In general, to reveal a diff record, Edna removes the placeholder row and inserts the original row (both of which are recorded in the diff record). However, for efficiency, Edna first checks if a placeholder row in diff records has the same identifiers (e.g., a primary key) as an original data row in the diff record, and if so, updates the relevant columns of the placeholder row instead.

Finally, Edna de-registers any pseudoprincipal who no longer has any associated disguised data, removing them from the principal table and the application's users table. Developers can configure Edna to also check for references to pseudoprincipals prior to removing them, and depending on the application's needs, configure Edna to delete, rewrite, or leave the references in place. After revealing, the disguised data is no longer needed, so Edna clears the corresponding diff and speaks-for records.

In the example, if Bea wants to reveal their "Star Wars" contributions, Lobsters invokes Edna with the disguise ID and Bea's password as reveal credentials. Edna uses the password to reconstruct Bea's private key, retrieve and decrypt Bea's records, and filter those records for those with the disguise ID. Edna then restores deleted votes and Bea's ownership of decorrelated comments.

5.3 Global Database Updates

Edna's revealing as described thus far may reveal data that ignores database updates applied since the time of disguise, such as global transformations to undisguised data or schema changes. To prevent this, Edna utilizes reveal-time update specifications provided by the developer to apply updates to disguised data prior to revealing it. This ensures that revealed data correctly reflects the current state of the database and any implicit application-level invariants. Prior to revealing the updated data, Edna also performs consistency checks to guarantee adherence to internal database invariants, such as uniqueness constraints. Here, we describe how developers specify reveal-time updates; how Edna applies these updates to data to reveal; and Edna's consistency checks.

5.3.1 Global Data Transformations

In order for revealing to preserve application correctness, Edna must obey global database updates that transform application data. Consider an example: a moderator edits all posts to remove swear words, creating an implicit invariant that all posts created prior to the last moderation pass should contain no swear words. If a user's disguise removes their posts, then a moderation pass occurs, and then the user wants to restore their posts, Edna might incorrectly restore the original post content with swear words present upon reveal. However, Edna could correctly restore the post if it knew to remove the swear words prior to reveal.

Figure 5-3 demonstrates how Edna applies updates to disguised data using Edna's *replay log* with *reveal-time update specifications*:

(1) Edna disguises some user data.

(2) The developer invokes Edna with a reveal-time update specification when performing a global update that must hold over disguised data when it is revealed (e.g., moderations). Edna records the update specification in its replay log. Update specifications map the

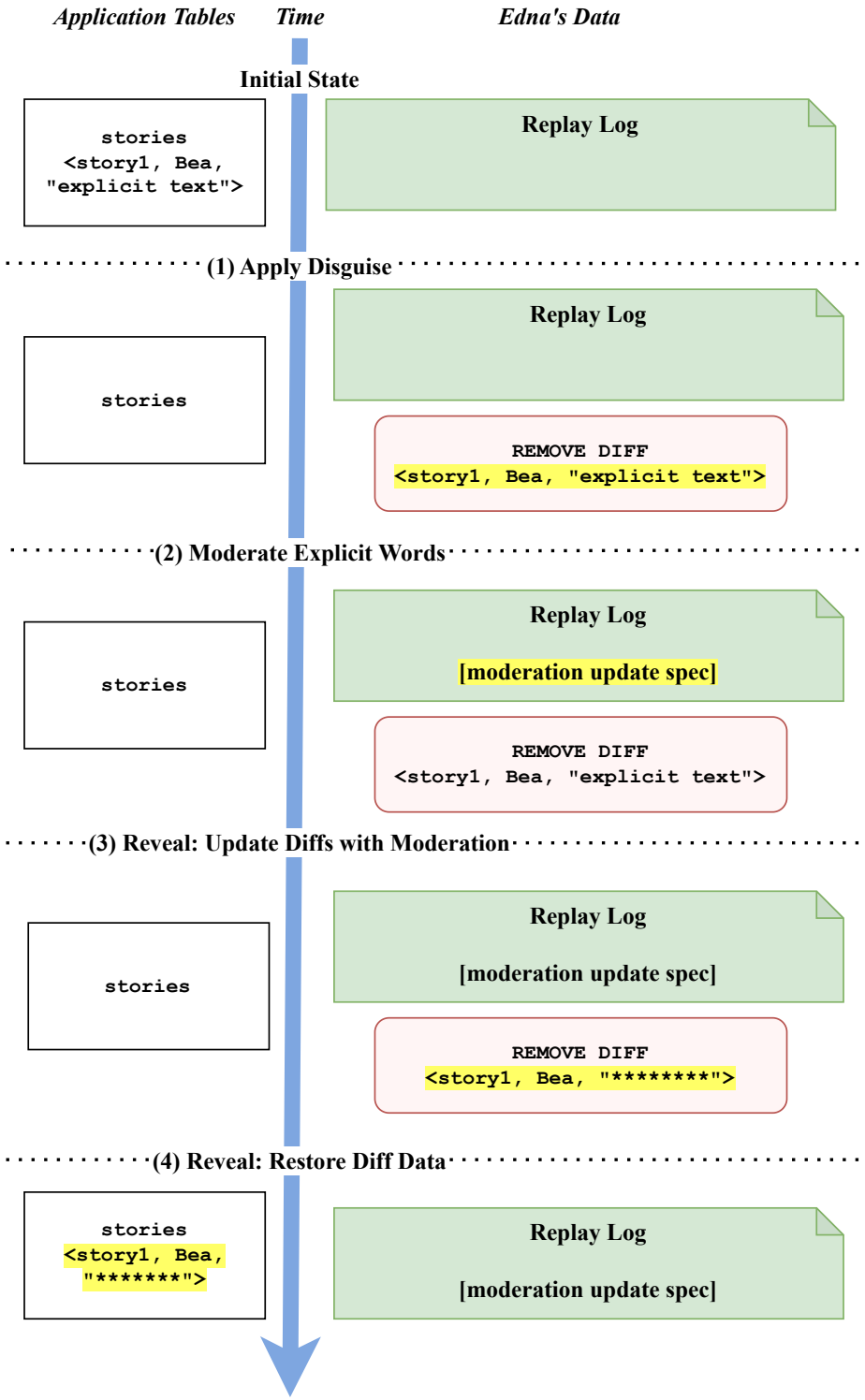


Figure 5-3: When the application applies updates like moderations of explicit words, it invokes Edna to log the corresponding reveal-time update specification. Edna then applies the update to disguised data in diff records prior to revealing them. Yellow highlights changes to the application data or Edna's data.

original data and the placeholder data in diff records to corresponding updated-original and updated-placeholder data.

(3) When revealing data, Edna applies, in order, every specified update in the replay log added since the time of disguise to the disguise's diff records. In this case, Edna moderates explicit story text.

(4) After applying all updates to placeholder data, Edna reveals the updated rows in the diff record like normal.

In the swear words moderation example, Edna queries the replay log, sees the swear words moderation entry, and then applies the swear words moderation to the removed post data in the diff record. Only then does Edna restore the post with no swear words.[1]

5.3.2 Schema Changes

Applications also undergo database updates to migrate their schema in order to reorganize data or add new application features. When these occur, Edna must know how to manipulate any disguised data structured in the old database layout to match that of the current database. At first glance, schema changes might seem like a different class of database update than global data transformations like content moderation, and thus require a different approach. However, the technique described in §5.3.1 for global changes allows Edna to handle schema changes as well.

Take, for example, a developer of an application with a `users` table that contains rows with `username`s, `addr`s, and `email`s. The developer may choose to allow users multiple addresses via a schema change. To do so, they would create a new `addrs` table, with a foreign key to the users table, and populate `addrs` using the address data in `users`. Finally, they would remove the `addr` column from `users`.

Figure 5-3 demonstrates how Edna applies such a schema change to disguised data.

(1) Edna disguises some user data.

(2) The developer invokes Edna with a reveal-time update specification when performing the schema change; Edna records the update specification its replay log.

(3) When revealing data, Edna applies the schema change update to diff record data, which maps an original `users` data row to both a row in `users` with username `uid` and a row in `addrs` that has a foreign key of `uid` to `users`. The update also generates two rows for any placeholder `users` data row in a diff record.

(4) Edna then respectively restores and removes the migrated original data and placeholder data respectively, both of which now consist of one `users` and one `addrs` row.

[1] Note that in this example, the diff record contains no placeholder data that replaced the removed post; in other scenarios such as decorrelation or modification, Edna would apply the logged update to diff placeholder data as well, in order to find the matching version in the database and remove it.

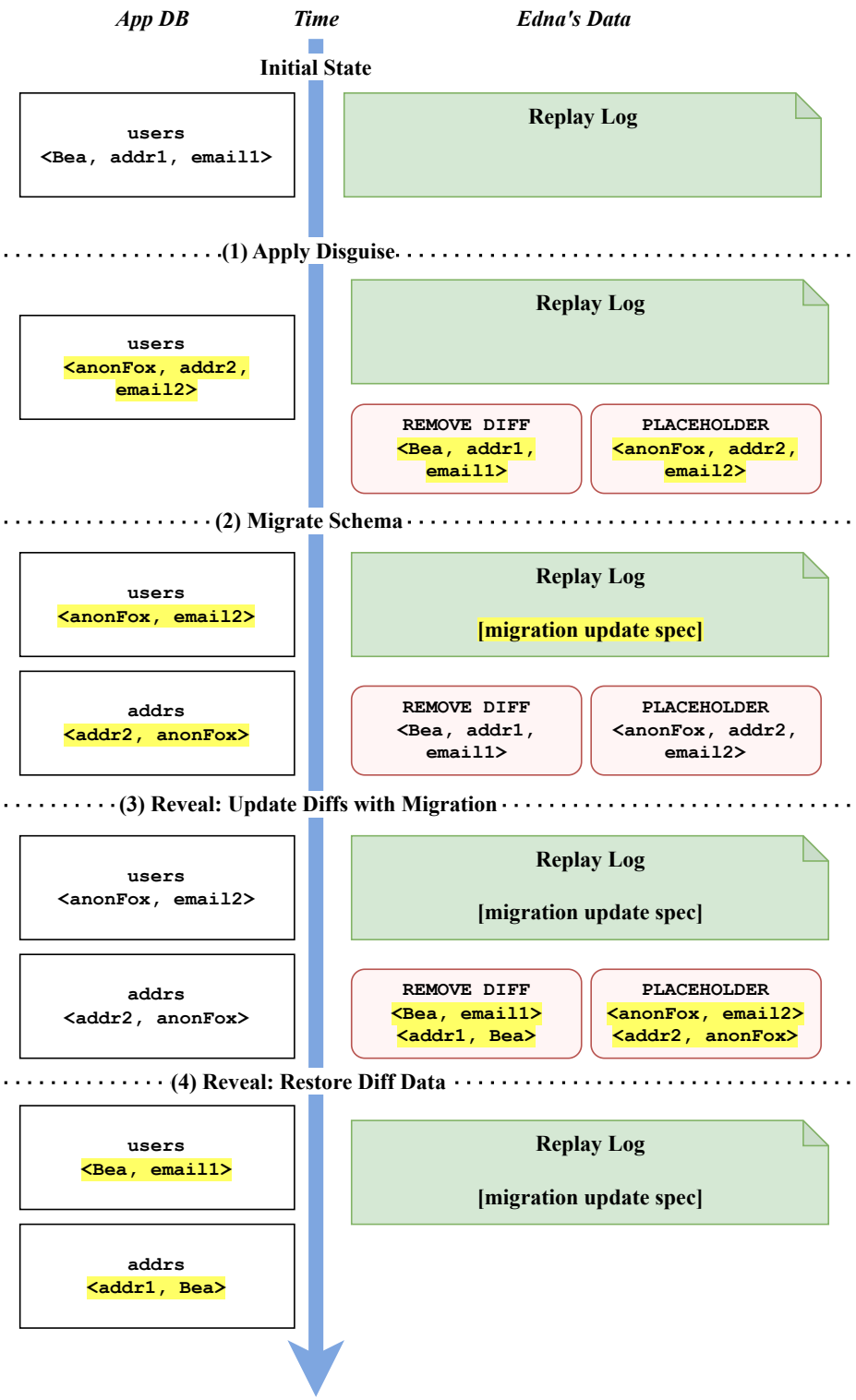


Figure 5-4: Similar to a moderations update, the application notifies Edna of a schema change to apply to diff records prior to reveal. The schema change creates an address table with a foreign key to user rows, and removes the address from the user rows. This allows users to have multiple addresses. Yellow highlights changes to the application data or Edna's data. For simplicity, the figure shows only the placeholder data of a pseudoprincipal row; in reality, placeholder data exists as part of e.g., a decorrelation diff record.

5.3.3 Consistency Checks for Internal Invariants

After applying any developer-specified reveal-time updates that occurred since the time of disguising, Edna performs consistency checks on the data to reveal to ensure that revealing will not violate the database's integrity. These checks allow revealing only if the revealed data: (*i*) will still satisfy uniqueness and primary key constraints; (*ii*) will not overwrite updates that occurred while data was disguised; and (*iii*) will maintain referential integrity.

For (*i*), Edna checks that *removed* disguised data is still removed from the database.

For (*ii*), Edna ensures that *modified* disguised data is in the same modified state and *decorrelated* disguised data is still affiliated with the same pseudoprincipal in the database using the new value stored in the diff record. By default, Edna performs checks at column granularity. For example, a disguised row can have two modified columns, but at the time of reveal, Edna finds that only one column value remains at the modified disguised state that Edna expects. The application thus must have updated the other column value since the time of the disguise. Edna will only reveal the one column that matches the value Edna disguised it to, in order to avoid overwriting application database updates. This results in a partial restoration of the disguised row. Developers can optionally specify that rows must be completely revealed: if any disguised column has been later changed by the application, Edna should not reveal *any* column in the row.

To ensure (*iii*), Edna checks for the existence of all objects referenced by the data to reveal (e.g., a post referenced by a to-be-revealed comment).

Edna is conservative and will never reveal rows for which checks fail; the affected data remains disguised. For example, if a developer chooses not to register conflicting global database updates, Edna's checks may fail, preventing disguised data from being revealed. Edna could log encountered conflicts, giving the application a chance to fix them so a later reveal can pass the checks.

5.4 Shared Data

Edna supports joint ownership semantics for shared data by modifying its design for removal when encountering a shared object. The first time any owner removes the data, Edna would normally remove the data object. However, since the other owners have not since removed the data, Edna must keep the data object around, but dissociated from the removing owner.

Edna ensures that (*i*) if any user removes the data, then the data is decorrelated from their identity; (*ii*) if all users have removed the data, the data is removed from the database; and (*iii*) if any user returns, the data is restored to the database with only revealed users recorrelated.

To support these semantics, Edna creates a *partially-removed* metadata table. The table is indexed by a data row's unique identifier columns (e.g., a primary key id); each index maps to a (plaintext) remove diff record for a shared data object. The diff record contains the shared data row, but where the row has been fullydecorrelated: all foreign keys to owners of the object

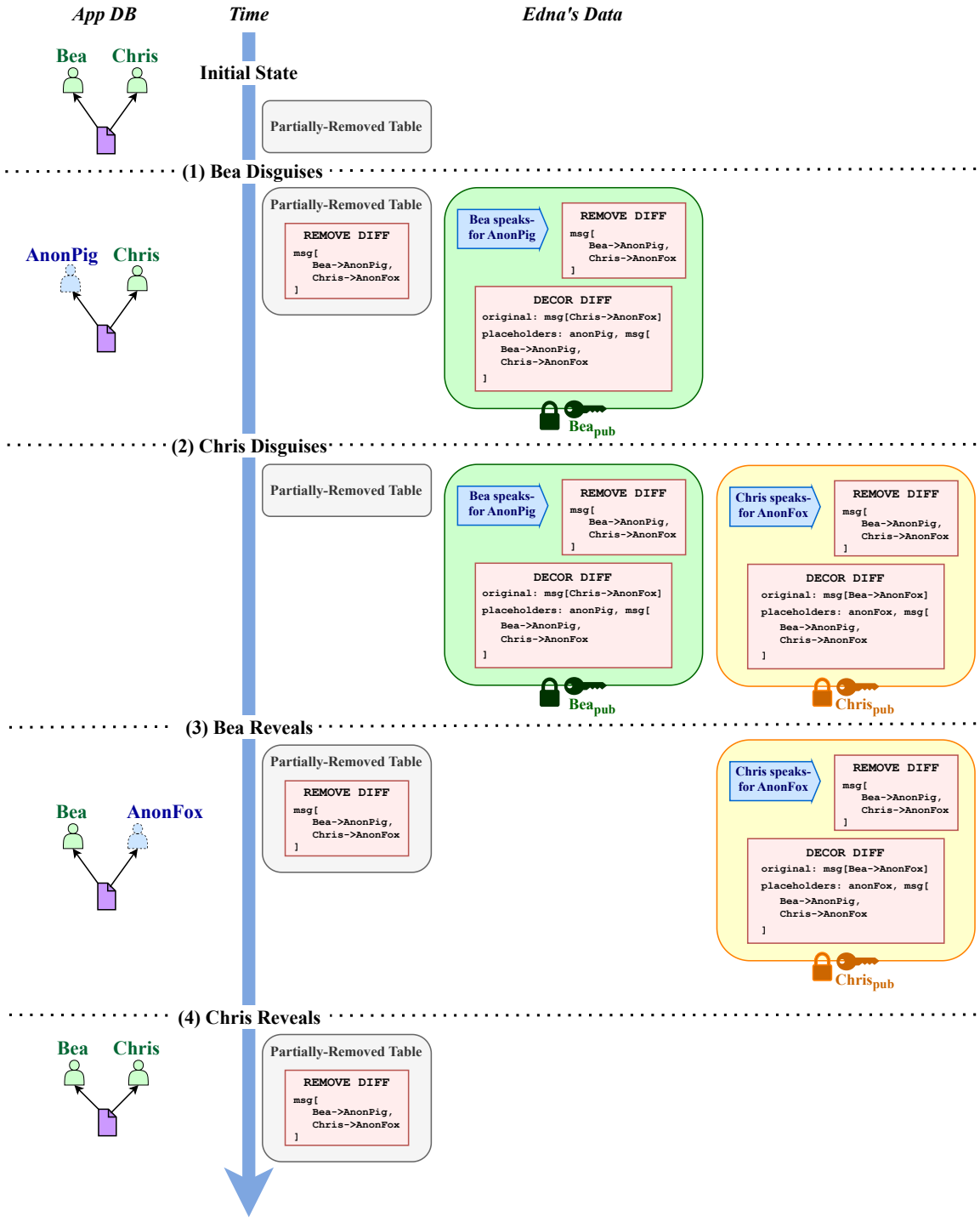


Figure 5-5: Edna implements joint ownership semantics by storing a fully-decorrelated diff record for the removed shared data (modifications to the shared message data are depicted using [owner→pseudoprincipal] notation). Owners can only see pseudoprincipals for the other owners in their disguised data. This allows Edna to disguise and reveal in any order.better caption relating to example.

have been rewritten to refer to a pseudoprincipal. Entries in the partially-removed table ensure that all owners agree on the partially-removed state of shared data (e.g., when some users are gone but others remain, so pseudoprincipals replace the removed users). This allows users to reveal consistently no matter which user reveals the shared data object first.

Figure 5-5 depicts how Edna executes Edna over shared data:

(1) When any owner removes some shared data row, Edna checks the partially-removed metadata table with the data's identifiers.

If Edna does *not* find a matching entry, as in Figure 5-5, then Edna creates a fully-decorrelated version of the data row (with all foreign keys to owner replaced with random pseudoprincipal identifiers). Edna then inserts this into the partially-removed table.

A matching entry must now exist; Edna takes the matching entry and stores a copy of the entry's fully-decorrelated diff record for the disguise. As shown in step 3, this information will let an owner restore the data to the database if all owners have removed, and the data no longer exists in the database.

Edna then creates the pseudoprincipal that matches the corresponding decorrelated foreign key to the owner, and stores diff records recording the pseudoprincipal row and foreign key rewrite for the owner. Finally, Edna stores a speaks-for record recording which pseudoprincipal the owner can speak-for. These records allow the owner to recorrelate back with the shared data after the data is restored to the database in a decorrelated state (steps 3 and 4).

(2) When the last remaining owner removes the data (all other owners are pseudoprincipals, i.e., all other owners have removed the data), Edna removes the shared data row from the application database, and the corresponding entry from Edna's partially-removed table.

(3) When *any* owner chooses to return, Edna first restores the fully-decorrelated row to the database as well as the partially-removed table, using the copy of the fully-decorrelated diff record in the owner's disguised data. Edna then rewrites the foreign key for the pseudoprincipal who currently owns the shared data to recorrelate with the owner, and removes that pseudoprincipal, revealing the other diff records like normal.

(4) If a subsequent owner reveals the shared data, Edna's consistency checks will prevent the reveal of the fully-decorrelated row (inserting the row will cause a duplicate row in the table). However, Edna will still rewrite the foreign key for that owner's pseudoprincipal to the original owner, and remove the pseudoprincipal from the database, thus recorrelating the data to the owner and restoring ownership.

Note that the partially-removed table entry for any shared data object is created only once: the first time any owner's disguise removes the shared data object. Future removes after reveals will reuse the same partially-removed table entry. Edna thus ensures that any user who reveals and then removes the shared data again decorrelates to the *same* pseudoprincipal as before.

With this design, Edna can disguise and reveal shared data no matter the order in which its owners decide to remove or reveal it. If an owner never removes the shared data, they will continue to be correlated and have access to the data even if other owners remove it (and become pseudoprincipals); similarly, if an owner never restores the shared data, the data will remain forever owned by the owner's pseudoprincipal if other owners choose to restore it.

5.5 Composing Disguising Transformations

Edna supports composition of disguising transformations, which occurs when a transformation applies to data that Edna has previously disguised in some other way. Reasoning about composition of transformations can be broken down to reasoning about the composition of primitive operation pairs, e.g., remove after modify, or remove after decorrelation.

Many pairs result in trivial composition: no operation can be composed after a remove (the data is gone), and any operation after a modify updates the data as expected. However, operations after decorrelation result in more complex composition scenarios. For instance, decorrelation after decorrelation could occur if a user decorrelates some posts, after which an administrator decorrelates *all* posts. In this scenario, the administrator's disguising operation applies to pseudoprincipal-owned posts in the same way as it does to unmodified posts. This creates pseudoprincipals that can speak-for other pseudoprincipals. Edna uses the pseudoprincipal's registered public key to encrypt pseudoprincipal diff and speaks-for records, so Edna does not need to know its link to an original principal in order to encrypt and disguise its data.

Removal or modification after decorrelation also require special handling. For instance, a Lobsters user might first decorrelate some of their comments and then request to delete all their comments (e.g., by deleting their account). But the decorrelated comments are no longer linked to the original user; how can the deletion transformation find them? Edna addresses this question by accepting optional reveal credentials as part of the disguise operation. Reveal credentials let Edna recursively decrypt speaks-for records, and create a *speaks-for chain* starting from the disguising user, represented by speaks-for relationships between pseudoprincipals. Edna can then disguise all data of all pseudoprincipals included in the speaks-for chain.

With reveal credentials, Edna decrypts the user's previous diff and speaks-for records. Each speaks-for record includes the identifier for one of the user's pseudoprincipal and the pseudoprincipal's private key, and creates a "link" in the speaks-for chain. A pseudoprincipal's identifier allows Edna to find data referencing that pseudoprincipal and apply disguising transformations on behalf of the this pseudoprincipal as well as the user. A pseudoprincipal's private key—its reveal credential—allows Edna to recursively decrypt its speaks-for records, and disguise any pseudoprincipals of this pseudoprincipal created by multiple decorrelations.

Out-of-Order Reveals. Edna must also handle reveals of transformations in any order. As before, many scenarios are straightforward: revealing removals is trivial (data can only be re-

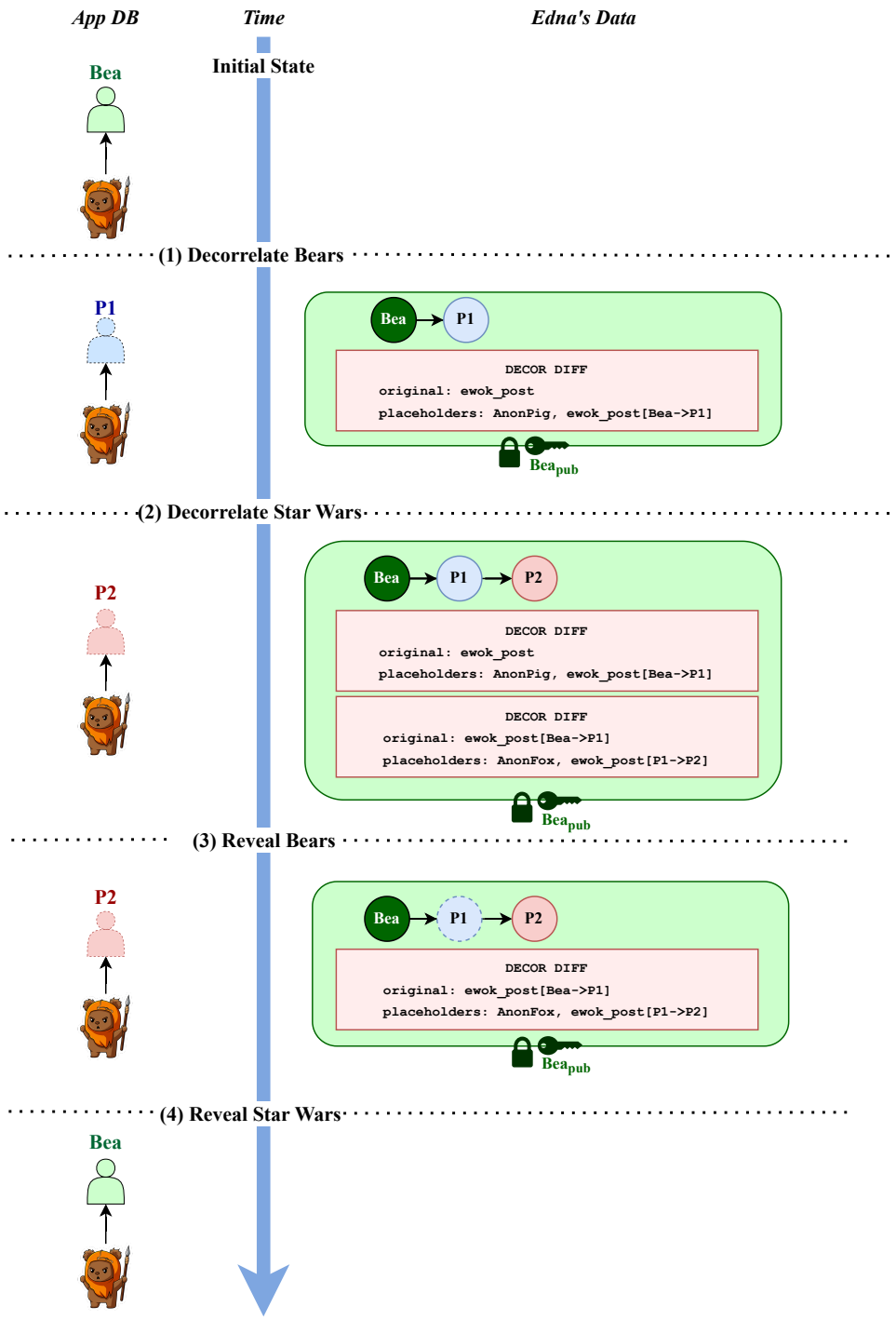


Figure 5-6: Edna allows multiple decorrelations to compose, and can reveal them in any order such that the data remains decorrelated until the user reveals *all* applied decorrelations. The speaks-for chain encoded in speaks-for records (shown in blue) allows Edna to handle recorrelation of intermediate pseudoprincipals. When Bea requests reveal of their "Star Wars" posts (4) after revealing "bears" posts (3), Edna walks the speaks-for chain backwards from P_2 to find the latest principal that can speak-for P_2 and has not yet been recorrelated. Thus, Edna restores ownership back to Bea.

moved and restored once), and revealing modified data simply restores the original (subject to consistency checks). Handling out-of-order reveals of multiple decorrelations presents the greatest challenge. Edna's semantics (§4.1) require that data that is decorrelated multiple times will not be recorrelated until all disguises are removed. For example, as shown in Figure 5-6 (left-hand side), if Bea separately decorrelates their comments on "bears" and "Star Wars" posts, then later reveals the "bears" posts, they might want Ewok-related comments (tagged both "Star Wars" and "bears") to remain disguised, even though they were initially disguised under the "bears" transformation. To support this, Edna again uses the speaks-for chain (§5.5) that represent speaks-for relationships between pseudoprincipals stemming from the revealing user. All reveal operations walk the full speaks-for chain to reveal all necessary records (cf. Figure 5-2).

Furthermore, if reveal operations perform recorrelations out of order, Edna removes an intermediate link in the speaks-for chain. To describe how this works, take the example of two disguises that decorrelate comments on "bears" and "Star Wars" posts respectively (Figure 5-6).

(1) First, the "bears" decorrelation rewrites Ewok posts to pseudoprincipal P_1.

(2) Next, the "Star Wars" decorrelation composed on top decorrelates Ewok posts from P_1 to pseudoprincipal P_2. At this point, Edna has speaks-for records for Bea that encode a speaks-for chain from Bea $\rightarrow P_1 \rightarrow P_2$.

(3) If Bea first reveals the "bears" anonymization—the first applied disguise—on their data, Edna finds all accessible speaks-for records given Bea's reveal credentials (cf. Figure 5-2), and constructs Bea's speaks-for chain as a graph of principal-to-principal edges (namely Bea$\rightarrow P_1 \rightarrow P_2$). Edna finds that P_1's "Ewok" posts no longer exist in the database (they belong to P_2 due to the composed "Star Wars" disguise), and thus does not restore "Ewok" posts to Bea. Edna does still remove pseudoprincipal P_1 in the process of restoring "bear" diff records, and once done, clears all "bear" 'diff records from its encrypted disguise table. Importantly, however, Edna still retains the speaks-for record for Bea$\rightarrow P_1$, since P_1 still has associated disguised data (namely a speaks-for record to P_2)!

(4) Now, if Bea reveals the second applied disguise—"Star Wars"—Edna finds diff records for decorrelated "Ewok" posts whose original rows have P_1 as owner. However, Edna cannot reveal the diff record directly and restore the decorrelated posts to P_1, as this will fail referential integrity consistency checks. Furthermore, the reveal of "Star Wars" reveals the last disguise applying to Bea's posts, so Edna should restore posts to their original, undisguised state (instead of decorrelated to P_1).

Instead of immediately revealing the diff record, Edna walks the speaks-for chain (Bea$\rightarrow P_1 \rightarrow P_2$) backwards from P_2 to determine the *next valid* principal in the chain who speaks-for P_2, which in this case is Bea. A valid principal can either be a natural principal or a pseudoprincipal yet to be recorrelated. If a pseudoprincipal appears as placeholder data in any diff record of Bea's disguised data (from any disguise), Edna knows that it has not

yet revealed that pseudoprincipal. Had Edna revealed that pseudoprincipal, then Edna would have removed its corresponding diff record.

After determining that Bea is the next valid user in the speaks-for chain to P_2, Edna rewrites the removed "Ewok" row in the diff record so that it references Bea instead of P_1, and then restores the rewritten row to the database. Finally, Edna deletes the diff records for the "Star Wars" disguise, the speaks-for records for P_2 (which no longer has any associated disguised data), and the speaks-for records for P_1 (which also has no associated disguised data after P_2 is removed). This implicitly truncates the speaks-for chain to simply be Bea.

Edna enforces the invariant that the chain only truncates at link L once all pseudoprincipals recursively generated in the chain beyond L have been recorrelated.

5.6 Authenticating as Pseudoprincipals

As described so far, if Bea wanted to modify a decorrelated "Star Wars" comment, they would have to reveal the comment, edit it using their normal credentials, and then redisguise the comment again. Applications that use Edna can also let users modify decorrelated records without the reveal step. To support this, an application accepts reveal credentials along with a modification request. Edna uses these credentials to validate that the user speaks-for a specific pseudoprincipal by walking the speaks-for chain (cf. Figure 5-2). This ensures that the user can access a speaks-for record linked to the pseudoprincipal. After validating the user's request, Edna updates the database with the modification.

5.7 Design Limitations

Disguises. Edna assumes that all desired disguises are captured with the three primitive operations (remove, modify, and decorrelate). Furthermore, disguising transformations may affect data processing of application data (e.g., aggregates over the number of users), or application side effects dependent on application data (e.g., sending notifications). Edna currently expects the developer to correctly handle these scenarios, and to ensure that any modified aggregations or placeholder data do not violate application correctness.

API. Edna's API assumes that:

1. the application uses a relational database;
2. rows to disguise have direct foreign key relationships to a users table, where each user corresponds to a row of that table;
3. all rows to disguise are owned by one or more principals; and
4. all rows can be uniquely identified (e.g., via primary key).

Applications that fail to satisfy these assumptions—e.g., because they have complex ownership chains or use a NoSQL database—could be supported with extensions to Edna's design. Edna could use techniques from DELF [**delf**] to support multiple data models; and K9DB's data ownership graph [**k9db**] to handle indirect data ownership. If no user owns a data item, Edna could refuse to disguise it and flag the developer to review the disguise specification. Finally, to address the unique identification requirement, Edna could add unique IDs for every data object, so Edna can refind the object when revealing. However, this method requires more invasive changes to application data.

Shared Data. Edna's approach for shared data faces two main limitations. First, the application cannot use foreign keys to the users table as unique identifiers for a table's rows. Edna uses unique identifiers to determine which objects to update upon a reveal, and also modifies foreign keys to the users table when decorrelating and recorrelating users from a shared data object. Thus, if foreign keys to the users table are modified, the unique identifier would change as well, and Edna would incorrectly reveal shared data. Note that this would not be an issue for a single-owner removal, since the object is removed and not dissociated from the owner. If foreign keys to users table act as unique identifiers, one potential solution might introduce a layer of indirection by adding a new unique identifier (e.g., an autoincrementing primary key), at the cost of potential performance impacts.

Second, owners of shared data who remove and reveal the data multiple times decorrelate to the same pseudoprincipal each time. This may allow for more opportunity for observers of the application to determine which pseudoprincipal corresponds to which user (although this falls outside of Edna's threat model). To see why Edna requires this limitation, imagine the following counterexample:

1) Bea and Chris both remove a shared message, so no data is left in the database. Both Bea and Chris store a diff record of the message mapping Bea to P_1, and Chris to P_2.

2) Chris first reveals and then removes the message again, so no data is left in the database. Chris now stores a diff record of the message with Chris mapped some *new* pseudoprincipal P_3.

3) Now Bea reveals, restoring the message with P_1 and P_2 to the database.

4) Chris attempts to reveal the message, but cannot find P_3 in the database, and thus fails to recorrelate with the message.

Chris' reveal's failure arises because Chris and Bea disagree upon what data—in particular, which pseudoprincipal references—to insert into the database, should the shared data object no longer exist. Chris believes it to be P_3, whereas Bea will restore to P_2 for Chris. To remedy this, Edna uses the partially-removed table to ensure that owners agree on the intermediate states of partially removed data.

Reveal-Time Updates. Reveal-time updates in replay logs only apply to diff records, which contain the actual data changes, and not speaks-for records; Edna assumes that the identifiers for principals in speaks-for records that encode the speaks-for chain remain consistent as the database changes, allowing Edna to use its composition techniques with speaks-for chains (§14) to reveal multiple disguises in any order.

Furthermore, as described in §3.4, Edna assumes that any nondeterminism or update side effects will maintain application correctness. One potential extension would be for developers to indicate the data dependencies of each update. To handle the unsupported example in §3.4, Edna can use the knowledge that posts depend on votes to restore disguised votes before posts. However, because votes have a foreign key to posts, this requires Edna to disable foreign key checks and carefully check for dangling votes that violate referential integrity after restoring posts. A perhaps more realistic but more limited solution might require developers to schedule periodic updates to "fix up" any rows that have been revealed since the update. However, this only works for an idempotent update, as it should not incorrectly update already-updated rows again.

5.8 Security Discussion

Edna's design achieves confidentiality of disguised data between the time of disguising and revealing, its key goal. By contrast, some aspects of Edna's design help make Edna practical and deployable without major application modifications, but give up stronger security in exchange for usability.

Under Edna's threat model, Edna achieves:

1. confidentiality of disguised data, via encrypting disguised data using asymmetric encryption, so only the owning user's private key can reveal it;
2. confidentiality of which encrypted disguised data belongs to which user, via opaque, encrypted indexing to reference a user's disguised data; and
3. reduced linkability between parts of a user's data, via splitting data ownership among pseudoprincipals.

Edna provides decorrelation with pseudoprincipals to ease integration with existing applications, even though pseudoprincipals (and their mere existence) can reveal information to the attacker. Pseudoprincipals preserve application data and referential integrity, ensuring that e.g., every post always has an author, or that vote counts on posts remain unchanged, without requiring the developer to handle special cases of deleted users and orphaned data. However, this necessarily leaves information in the database: an attacker with database access could see all application database content and code, and Edna's disguise, principal, and deleted principal tables. Thus, what the attacker learns includes:

1. any undisguised data in the application database;
2. the active principals that have disguised data, via Edna's principal table;

3. the pseudoprincipals currently registered, from Edna's principal table and the application DB;
4. the number of deleted principals, via the size of the deleted principal table;
5. the amount of disguised data in Edna; and
6. the disguise specifications, from application code.

Leveraging the application database (as opposed to separate external storage) to store disguised data increases Edna's practicality as it reuses server-side storage resources ? and avoids burdening users with managing their disguised data. However, this leaves potentially exploitable metadata available to attackers. An attacker could leverage pseudoprincipal groupings (e.g., a pseudoprincipal owning posts in both "CMU 2018" and "BayArea" topics), undisguised data (e.g., comments signed with the user's name), and Edna metadata (e.g., that some anonymous user has more disguised data than another, as Edna stores disguised data without padding for efficiency) to infer the identity of the original owning principal.

Finally, Edna makes no guarantees for users who actively use disguised data after compromise (e.g., by revealing or editing decorrelated data): after an attacker compromises the application at time t, they can harvest private keys that clients provide after t. However, Edna always protects users' disguised data if they remain inactive.

The attacker never has access to a user's private key unless the user actively provides their credentials. The attacker also cannot access the private key of any pseudoprincipal because Edna stores such keys in encrypted speaks-for records. If an application uses password-based reveal credentials, Edna guarantees security equivalent to the security of the user's password.

5.9 Edna without Encryption

Should developers want to support disguised data in a weaker threat model that does not require encryption (as described in §4.3), then certain aspects of Edna's design would be unnecessary. This section describes these aspects and potential alternative designs for disguised data without encryption.

Unnecessary Aspects of Edna. If Edna removes encryption of disguised data, Edna would no longer need the replay log for reveal-time udpates, since disguised data remains accessible to the application and updates can thus apply immediately to disguised data. However, developers may still need to write data transformations corresponding to these updates in order to update disguised data rows stored in the disguised tables.

Removal of encryption also eliminates the need for registration with a user keypair, user reveal credentials, encryption/decryption on the disguise/reveal paths, and encrypted indexes mapping a user to their disguised data. The disguise table would hold plaintext disguised data, and Edna would not need the principal table.

Furthermore, Edna can always disguise already-disguised data, since a user does not need to

provide a reveal credential in order to unlock their already-disguised data. A developer instead can allow a user to specify whether they wish to disguise their disguised data again using a boolean.

This simplified design still helps address the challenges of maintaining referential integrity when a disguise is performed (creating pseudoprincipals as placeholder users); composing disguises and reveals in arbitrary orders; disguising shared data; and applying global updates to disguised data (albeit during update execution, rather than at reveal time).

Alternatives Designs Without Encryption. Instead of moving disguised data into a new database table for disguised data as Edna does, developers could also choose to implement disguised data by logically removing it using database flags and predicates (e.g., `is_deleted`) because disguised data need not be encrypted. In this case, the developer must ensure that all queries for the data predicate on the tag so the application does not return disguised data. Systems like Qapla [**qapla**]—which this thesis compares to Edna in §8.4— can help enforce these flag-based policies.

Chapter 6

Implementation

The current Edna prototype consists of ≈ 6k lines of Rust (broken down by component in Figure 6-1). Edna supports MySQL, and provides an API server that exposes a JSON-HTTP API that applications can use to invoke Edna, if they do not link directly with Edna's Rust library. Edna's API (Figure 4-2) supports principal registration, data disguising and revealing, partial revealing of speaks-for relationships, and recording global database updates as reveal-time update specifications.

Component	LoC
API	0.5k
disguiser	1.0k
revealer	0.3k
records	2.2k
helpers	2.0k
examples and tests	3.1k

Figure 6-1: Code components of Edna and their respective sizes. Much of the logic for reveal is embedded in the record implementation (e.g., a diff record struct implements `reveal`); the revealer handles the reveal order of diff records and modifications of the speaks-for chain.

6.1 Secure Record Storage

When encrypting diff and speaks-for records, Edna appends a random nonce to the record plaintext to prevent known-plaintext attacks. It then generates a new public/private keypair for x25519 elliptic curve key exchange. Using the newly created private key and the principal's public key, Edna performs the x25519 elliptic curve Diffie-Hellman ephemeral key exchange to generate a shared secret. Edna encrypts the record data with the shared secret, and saves the ciphertext along with the freshly generated public key (required to decrypt the data given the principal's private key). This public key algorithm lacks key anonymity, so an attacker can determine which records belong to the same principal, but this is not fundamental [**anonymous-keys**].

6.2 Reveal Credentials

Edna supports two forms of reveal credentials: private keys; or principals' passwords (plus recovery tokens in case they forget their passwords). If an application chooses to use the latter, it provides the principal's password to Edna upon user registration. Edna uses a variant of Shamir's Secret Sharing [**secretsharing**] to generate three shares from the private key, any two of which can reconstruct the private key. Shares are $(x, f(x) \bmod p)$ tuples, where $f(x) = \texttt{privkey} + \texttt{rand} \cdot x$ and $p > \texttt{privkey}$ is a known prime. One share derives x from the user's password using a Password-Based Key Derivation Function (PBKDF) [**pbkdf-rfc**]. Edna stores the resulting $f(x)$ half of the share, allowing Edna to derive one full share from the password. Edna returns the second full share as a recovery token and stores the third full share. Edna can combine this third share with the recovery token or a full share derived from the password to recover the private key.

The PBKDF ensures that Edna cannot guess the password-derived value with dictionary and rainbow table attacks [**pbkdf**], and that Edna cannot brute force the recovery token.

Password-based secret-sharing is only one possible implementation for backup secrets; Edna could also support password-based backup secrets by, for example, storing a version of the private key encrypted with the user's password.

6.3 Reveal-Time Update Specifications

Edna's prototype accepts reveal-time update specifications as Rust functions from database objects (<table, row> pairs) to a new set of database objects. Edna stores the replay log as a vector of updates function pointers with timestamps, and persists the pointers to disk. This approach requires recompiling Edna to add new update functions; an alternative approach could put the functions into separate executables (and Edna would store executable names instead of function pointers). In this alternative, Edna would invoke these executables upon reveal with serialized input rows, and deserialize the output.

6.4 Batching

For efficiency, Edna batches the deletion of rows from the same table, updates to rows of the same table from modifications and decorrelation, and the creation of pseudoprincipals upon disguise. Edna also batches the deletion of pseudoprincipals during reveal, and the reveal of same-table diff records. This allows, for example, all updates for database management operations to be applied to all diff record rows of the same table at the same time; and for all rows of the same table to be inserted, updated, or removed from the database in a single query.

Batching utilizes MySQL's `INSERT... ON DUPLICATE KEY UPDATE`, which allows batch updating different items to different row values. However, this also introduces potential errors

if an item needs to update its primary key as well (e.g., the disguise specification instructs Edna to disguise the user ID with a pseudoprincipal ID, but the user ID column also acts as a primary key). If Edna detects a change to a primary key, Edna individually updates rows' primary keys prior to the batch update, ensuring that rows sent in the `INSERT...UPDATE` query match those in the database.

6.5 Concurrency

Edna runs disguising and revealing transformations in transactions, providing serializable isolation to application users. If a query within a transformation fails, the entire transformation aborts (returning an error to the application). Edna provides an option to run long-running transformations that touch large amounts of data (e.g., anonymization of all users' posts) without a transaction, at the expense of clients potentially observing intermediate states.

Chapter 7

Case Studies

This section evaluates Edna's ability to add new disguising features to several applications; §8 evaluates the effort needed to do so and the resulting performance.

We add disguising and revealing transformations based on the motivating examples in §1 to three applications—Lobsters [**lobsters**], WebSubmit [**websubmit-rs**], and HotCRP [**hotcrp**].

7.1 Lobsters

Lobsters is a Ruby-on-Rails application backed by a MySQL database. Beyond the stories, tags, etc. mentioned in §3.1, Lobsters also contains moderations that mark inappropriate content as removed. We added three disguising transformations: account deletion with return; account decay (i.e., automatic dissociation and protection of old data); and topic-specific throwaway accounts.

GDPR-compliant account deletion (*i*) removes the user account; (*ii*) removes information that's only relevant to the individual user, such as their saved stories; (*iii*) modifies story and comment content to "[deleted content]"; (*iv*) decorrelates private messages; and (*v*) decorrelates votes, stories, comments, and moderations on the user's data. This preserves application semantics for other users—e.g., vote counts remain consistent even after account deletion, and other users' comments remain visible—while protecting the privacy of removed users. Important information such as moderations on user content remains in the database, and Edna recorrelates it if the user restores their account. After Edna applies the disguising transformation, Lobsters emails the user a URL that embeds the disguise ID. The user can visit this URL and provide their credentials to restore their account.

The **account decay** transformation protects user data after a period of user inactivity. We added a cron job that applies account decay to user accounts that have been inactive for over a year. This (*i*) removes the user's account; (*ii*) removes information only relevant to the user, such as saved stories; (*iii*) and decorrelates votes, stories, comments, and moderations on the user's data by associating them with pseudoprincipals. Lobsters sends the user an email which

informs them that their data has decayed, and includes a URL with an embedded disguise ID that can reactivate or completely remove the account if credentials are provided.

Finally, topic-based throwaway accounts via **topic-based anonymization** enable users to decorrelate their content relating to a particular topic. As per §3.1, this disguises contributions associated with the specified tag by (*i*) decorrelating tagged stories and comments associated with tagged stories, and (*ii*) removing votes for tagged stories. Again, Lobsters sends the user an email with links that allow reclaiming or editing these contributions.

With Edna's support for composing disguising transformations, users can delete accounts that have been decayed or dissociated into throwaways, and can later reveal them.

Global Database Changes. We implemented three examples of recent global database updates applied by the Lobsters developers in 2023–2024. The first changes the schema and transforms data contents of stories. It performs URL normalization [**urinorm**] on the `url` column of all rows in the `stories` table, and then stores the normalized URL text in a new `normalized_url` column. The second changes the schema by adding a `show_email` attribute to the `users` table automatically set to `false`. The third creates a new `story_texts` table that stores the `title`, `description`, and `story_cache` attributes of rows from `stories`; and removes the `story_cache` column from `stories`. This enables search over story content.

7.2 WebSubmit

We integrated Edna as a Rust library with WebSubmit [**websubmit-rs**]. WebSubmit is a homework submission application used at Brown University, and its schema consists of tables for lectures, questions, answers, and user accounts. Clients create an account, submit homework answers, and view their submissions; course staff can also view submissions, and add/edit questions and lectures. The original WebSubmit retains all user data forever. We added support for two disguising transformations: GDPR-compliant user account removal with return, and instructor-initiated answer anonymization, which protects data of prior years' students by decorrelating student answers for a given course. These transformations allow instructors to retain FERPA-compliant [**ferpa**] answers after the class has finished. With Edna, students can delete their accounts or access and view their answers even after class anonymization, and can always restore their deleted accounts, including restoring them to anonymized state.

7.3 HotCRP

HotCRP is a conference management application whose users can be reviewers and/or authors. HotCRP's schema contains papers, reviews, comments, tags, and per-user data such as watched papers and review ratings [**hotcrp**]. HotCRP currently retains past conference data forever and requires manual requests for account removal [**hotcrp:privacy**]. We wrote two disguise

specifications for HotCRP: conference anonymization to protect old conference reviews, and GDPR account removal with return.

Conference anonymization is invoked by PC chairs after the conference and decorrelates users from their submissions, reviews, comments, and per-user data such as watched papers. User accounts remain in the database with no associated data. Conference anonymization protects users' data after the conference; with Edna, users can come back to view or edit their anonymized reviews and comments.

Account removal (*i*) removes the user's account; (*ii*) removes information only relevant to the user, such as their review preferences; (*iii*) removes their author relationships to papers; and (*iv*) decorrelates the remainder of their data, such as reviews. Decorrelating a review removes its association with the reviewing user, but importantly keeps the review itself around to preserve utility for others (e.g., the PC and the authors of the reviewed paper). With Edna, users can remove their accounts even after conference anonymization has taken place, and can always restore their accounts.

Chapter 8

Evaluation

This chapter seeks to answer six questions:

1. How much developer effort and application modification does Edna require? (§8.1)
2. How expensive are common application operations, as well as disguising, revealing, and operations over disguised data with Edna? (§8.2)
3. What overheads does Edna impose, and where do they come from? (§8.3)
4. How does the effort required to implement Edna's functionality in a related system (Qapla [**qapla**]), and its performance, compare with using Edna? (§8.4)
5. What is the performance impact of composing Edna's guarantees with those of encrypted databases? (§8.5)
6. Which categories of global database updates can Edna support, and with what overheads? (§8.6)

To measure Edna's expenses (§8.2), we compare Edna to a manual version of each disguising transformation that directly modifies the database (e.g., via SQL queries that remove data), which lacks support for revealing and does not support composition of multiple transformations.

All benchmarks run on a Google Cloud `n1-standard-16` instance with 16 CPUs and 60 GB RAM, running Ubuntu 20.04.5 LTS. Benchmarks run in a closed-loop setting, so throughput and latency are inverses. The benchmarks use MariaDB 10.5 with the InnoDB storage engine atop a local SSD.

8.1 Edna Developer Effort

We evaluate the developer effort required to use Edna by measuring the difficulty of implementing the disguising and revealing transformations in our three case studies. This took one person-day per case study for a developer familiar with Edna but unfamiliar with the applications.

A developer supporting these transformations must first add application infrastructure to allow users to invoke them and notify users when they happen. This is required even if the developer were to implement transformations manually without Edna. These changes add 179

LoC of Ruby to Lobsters (160k LoC), and 312 LoC of Rust to the original WebSubmit (908 LoC). They implement HTTP endpoints, authorization of anonymous users, and email notifications.

A developer using Edna also writes disguise specifications and invokes Edna. Lobsters' disguise specifications are written in 518 LoC, WebSubmit's in 75 LoC, and HotCRP's in 357 LoC (all in JSON). The specification size is proportional to the lines required to specify the database schema in SQL, as well as what data each application disguises. Writing the corresponding updates for the three Lobsters global database updates requires 81 LoC; this adds to the 89 LoC that the developer already writes to implement the global database updates themselves.

Thus, the developer effort required to use Edna—writing Edna specifications, and invoking Edna—requires adding <1k LoC per application (less than 1% of the code of real world applications like Lobsters), even though these applications were not written with Edna in mind.

8.2 Performance of Edna Operations

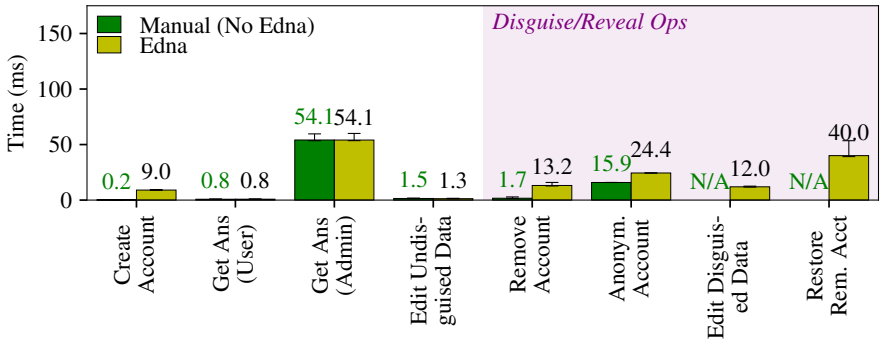
We next evaluate Edna's performance using WebSubmit, HotCRP, and Lobsters. We measure the latency of common operations, disguising transformations, and operations over disguised data enabled by Edna (e.g., account restoration and editing disguised data). The three applications do not create new data that reference pseudoprincipals, but to fully capture any overheads we configure Edna to nevertheless run the checks for lingering pseudoprincipal references on revealing.

A good result for Edna would show no overhead on common operations, since Edna should not be invoked on normal application execution paths. Edna should be competitive with manual disguising, with the caveat that Edna needs to also encrypt and store disguised data. Finally, Edna should have reasonable latencies for revealing operations supported only by Edna (e.g., a few seconds for account restoration), which require both database queries and cryptographic operations.

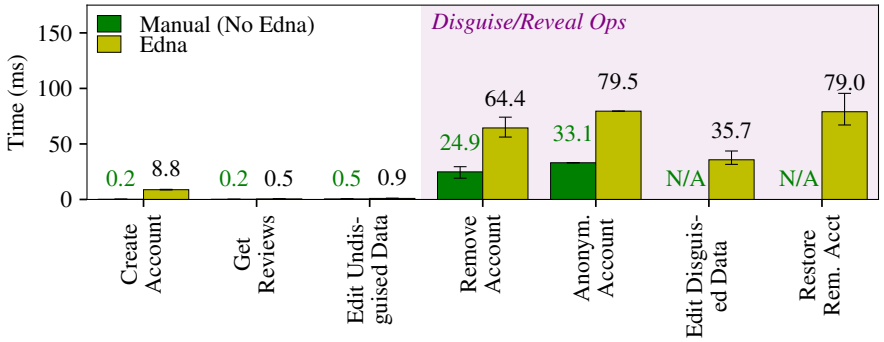
WebSubmit. We run WebSubmit with a database of 2k users, 20 lectures with four questions each, and an answer for each question for each user (160k total answers). We measure end-to-end latency to perform common application operations (which each issue multiple SQL queries), as well as disguising and revealing operations when possible (revealing operations are impossible in the baseline). Figure 8-1a shows that common operations have comparable latencies with and without Edna. Edna adds 9ms to account creation; disguising and revealing operations also take longer in Edna (13.2–40.0ms), but allow users to reveal their data.

HotCRP. We measure server-side HotCRP operation latencies for PC members on a database seeded with 3,080 total users (80 PC members) and 550 papers with eight reviews, three comments, and four conflicts each (distributed evenly among the PC). HotCRP supports the same disguising transformations as WebSubmit, but PC users have more data (200–300 records each), and HotCRP's disguising transformations mix deletions and decorrelations across 12 tables.

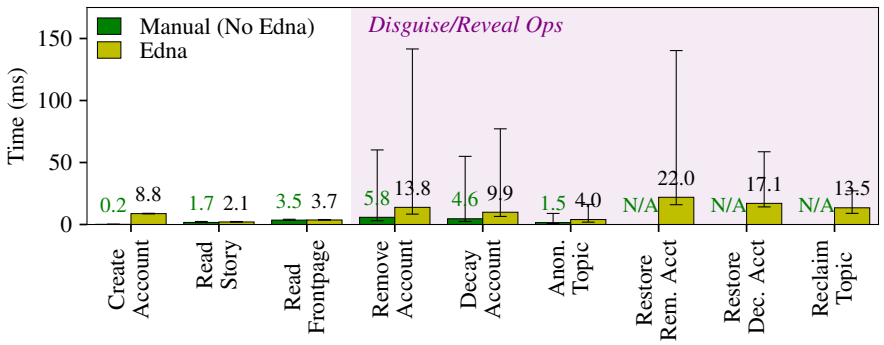
Figure 8-1b shows higher latencies than in WebSubmit in general, even for the manual base-



(a) WebSubmit (2k users, 80 answers/user).



(b) HotCRP (80 reviewers, 3k total users, 200–300 records/reviewer).



(c) Lobsters (16k users, Zipf-distributed data/user).

Figure 8-1: Edna adds no latency overhead to common application operations and modestly increases the latencies of disguising operations compared to a manual implementation that lacks support for revealing or composition. Bars show medians, error bars are $5^{th}/95^{th}$ percentile latencies.

line, which reflects the more complex disguising transformations. Edna takes 64.4–79.5ms to disguise and reveal a PC member's data, again owing to the cryptographic operations necessary. HotCRP's account anonymization is admin-applied and runs for all PC members, so its total latency is proportional to the PC size. With 80 PC members, this transformation takes 6.4s, which is acceptable for a one-off operation. As before, Edna adds small latency to common application operations, and 9ms to account creation.

Lobsters. We run Lobsters benchmarks on a database seeded with 16k users, 120k stories, and 300k comments with votes, comparable to the late-2022 size of production Lobsters [**lobsters**]. Content is distributed among users in a Zipf-like distribution according to statistics from the actual Lobsters deployment [**lobsters-data**], and 20% of each user's contributions are associated with the topic to anonymize. The benchmark measures server-side latency of common operations and disguising/revealing transformations.

The results are in Figure 8-1c. The median latencies for entire-account removal or decay are small (9.9–13.8ms for Edna, and 4.6–5.8ms for the baseline), since the median Lobsters user has little data. Revealing disguised accounts takes 17.1–22.0ms in the median. Highly active users with lots of data raise the 95^{th} percentile latency to ≈150ms for account removal and 150ms for account restoration. Topic anonymization touches less data and is faster than whole-account transformations, taking 4.0ms and 13.5ms for the median user to respectively disguise and reveal.

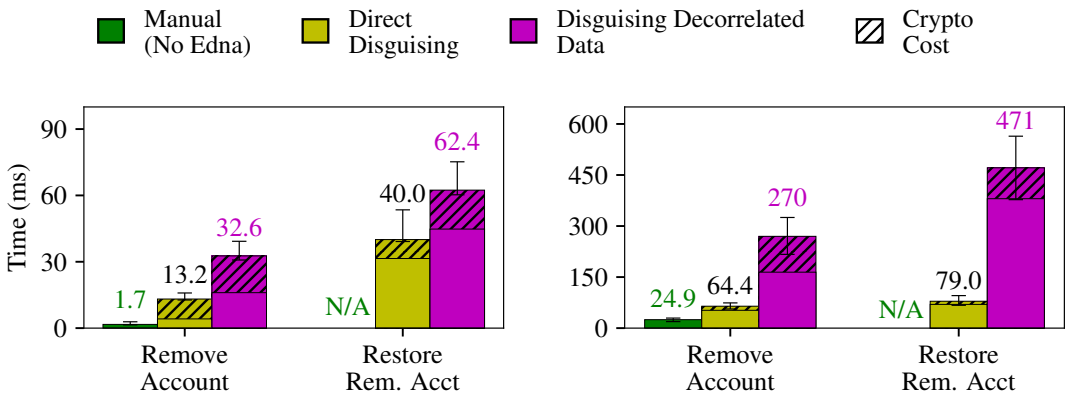
Summary. Edna necessarily adds some latency compared to manual, irreversible data removal, since it encrypts and stores disguised data. However, most disguising transformations are fast enough to run interactively as part of a web request. Some global disguising transformations—e.g., HotCRP's conference anonymization over many users—take several seconds, but an application can apply these incrementally in the background, as in Lobsters account decay.

8.2.1 Edna Performance Drill-Down

This section breaks down the cost of Edna's operations into the cost of database operations and the cost of cryptographic operations. Edna's database operations are fast; in our prototype, they generally take 0.2–0.3ms but vary depending on the amount of data touched. Edna's cryptographic operations are comparatively expensive. PBKDF2 hashing for private key management incurs a 8ms cost and affects account registration and operations on disguised data that reconstruct a user's private key; this accounts for up to 79% of these operations' cost when the operation issues only a few database queries.

Encryption and decryption incur baseline costs of 0.1ms and 0.02ms respectively; their cost grows linearly with data size. In the common case, disguising or revealing data performs two cryptographic operations: one to encrypt/decrypt the diff and speaks-for records, and one to encrypt/decrypt the ID at which they are stored.

Edna also generates a new key for each pseudoprincipal created, which takes 0.2ms. Edna's cryptography accounts for up to 35% of the cost of disguising/revealing operations such as



(a) WebSubmit

(b) HotCRP

Figure 8-2: Applying disguising transformations to previously-decorrelated accounts increases latency linear in the number of pseudoprincipals involved. Hatched lines indicate the proportion of cost attributed to cryptographic operations.

account removal or anonymization; this proportion decreases as the number of database modifications made by a transformation increases. When the application applies multiple disguising transformations and disguises the data of pseudoprincipals, doing so may require several encryptions/decryptions. We evaluate this cost next.

8.2.2 Composing Disguising Transformations

To understand the overhead of composing transformations in Edna (§5.5), we measure the cost of composing account removal on top of a prior disguising transformation to anonymize and decorrelate all users' data. We consider WebSubmit and HotCRP, and compare three setups: (*i*) manual account removal (as before); (*ii*) account removal and restoration *without* a prior anonymization disguising transformation; and (*iii*) account removal and restoration *with* a prior anonymization disguising transformation.

With prior anonymization, a subset of the user's data has already been decorrelated when removal occurs, and removal therefore performs per-pseudoprincipal encryptions of disguised data with pseudoprincipals' public keys. Restoring the removed, anonymized account must then individually decrypt pseudoprincipal records and restore them. Hence, disguising and revealing in the third setup should take time proportional to the number of pseudoprincipals created by anonymization.

Figure 8-2 shows the resulting latencies. WebSubmit account removal and restoration latencies increase by ≈1ms per pseudoprincipal (19.4ms and 22.4ms respectively). 50% of this increased cost comes from the additional, per-pseudoprincipal encryption and decryption of records, the rest comes from database operations.

HotCRP removal latencies also increase by ≈1ms per pseudoprincipal (205.6ms) and restoration latencies increase by ≈2ms per pseudoprincipal (392.0ms). Again, cryptographic opera-

Component	Pre-Delete (MB)	Post-Delete (MB)
App Tables	261	290
Disguise Table (Disk)	0	35.1
Deleted Principals Table (Disk)	0	0.4
Principals Table (Disk)	2.6	14.1
User Shares Table (Disk)	8.9	8.9

Figure 8-3: The space usage of the Lobsters-Edna database increases by 87.5MB on disk (and 44MB in memory) after 10% of 16k users remove their accounts with Edna.

tions add ≈0.5ms per pseudoprincipal, and the remaining cost increase comes from perpseudoprincipal database queries and updates. Restoration requires additional per-pseudoprincipal queries to e.g., perform consistency checks, resulting in a greater latency increase than in removal. Compared to accounts in WebSubmit, accounts in HotCRP have more data and 14–15× more pseudoprincipals after anonymization, which accounts for the larger relative slowdown and the increased effect of per-pseudoprincipal reveals.

WebSubmit and HotCRP do not create new references to pseudoprincipals after data is disguised, but if they did, Edna would need to issue additional per-pseudoprincipal queries to rewrite or remove these references (if configured to do so).

Importantly, disguising latencies stabilize when Edna composes further disguising transformations: since cost is proportional to the number of pseudoprincipals affected, latency does not grow further once the application has maximally decorrelated data (to one pseudoprincipal per record), as done by HotCRP anonymization.

8.3 Edna Overheads

Edna adds both space and compute overheads to the application: Edna stores disguised data and metadata, adding tables to the application DB; and Edna may run disguising and revealing transformations simultaneously with normal application operations, increasing the load on the system and affecting normal application operation throughput. This section measures both these overheads.

8.3.1 Space Used By Edna

To understand Edna's space footprint, we measure the size of all data stored on disk by Edna before and after 10% of users in Lobsters (1.6k users) remove their accounts. Cryptographic material adds overhead and each generated pseudoprincipal adds an additional user to the application database; Edna also stores data for each registered principal (a public key and a list of opaque indexes) as well as encrypted records.

Figure 8-3 shows the increase in space used by Edna and the application after Edna disguises 10% of Lobsters users. Edna's storage initially consumes 12 MB, which consists of entries in the principals table and user shares table from the registration of all 16k users. Edna's storage

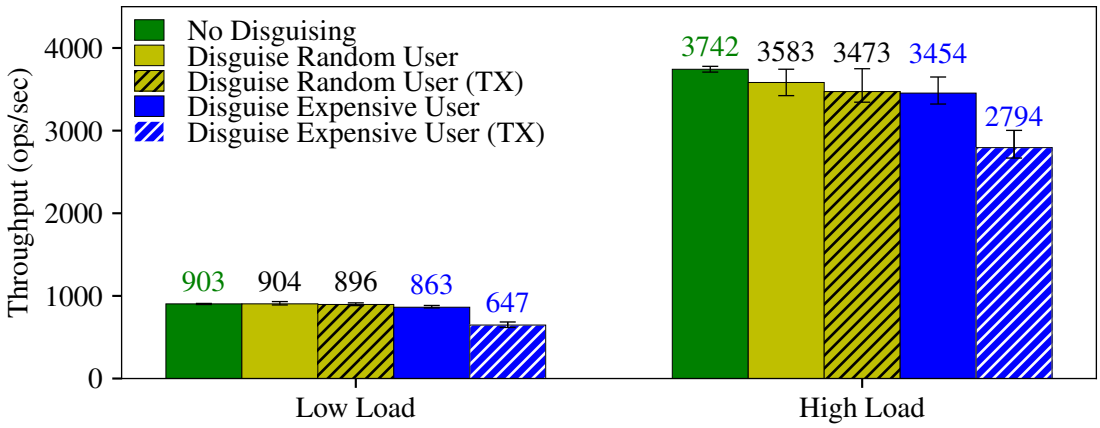


Figure 8-4: Continuous disguising/revealing operations in Lobsters have <7% impact on application request throughput when disguising a random user; an extreme case of a heavy-hitter user with lots of data repeatedly disguising and revealing causes up to 8% drop in throughput without transactions, and up to 28.3% drop in throughput with transactions. Higher bars are better.

space grows to 58.5 MB after the users remove their accounts, and the application database size increases from 261 MB to 290 MB (+11%). (Edna also caches this data using 44MB in memory.) The space used is primarily proportional to the number of pseudoprincipals produced: each pseudoprincipal requires storing an application database record, a speaks-for record, and row in the principal table. In this experiment, Lobsters produces 78.1k pseudoprincipals. Edna removes the public keys for the 1.6k removed principals—subtracting from the principals table, but adding to the deleted principals table—and removes the principals' database data to store as encrypted diff records in the disguise table, which uses 2.2 MB.

is this good/ok/acceptable?

8.3.2 Impact On Concurrent Application Use

For Edna to be practical, the throughput and latency of normal application requests by other users should be largely unaffected by Edna's disguising and revealing operations, even though Edna may modify the same tables touched by normal application requests.

font?

This section measures the impact of Edna's operations on other concurrent requests in Lobsters. In the experiment, a set of users make continuous requests to the application that simulate normal use, while another distinct set of users continuously remove and restore their accounts. Edna applies disguising transformations sequentially, so only one transformation happens at a time. We measure the throughput of "normal" users' application operations, both without Edna operations (the baseline) and with the application continuously invoking Edna. The Lobsters workload is based on request distributions in the real Lobsters deployment [**lobsters-data**].

Since users' disguising/revealing costs vary in Lobsters, we measure the impact of (*i*) randomly chosen users invoking account removal/restoration, and (*ii*) the user with the most data continuously removing and restoring their account (an expensive scenario). The results illustrate throughput under low load (≈20% CPU load), and high load (≈95% CPU load). Finally, we measure settings with and without a transaction for Edna transformations. A good result for Edna would show little impact on normal operation throughput when concurrent disguising transformations occur.

Figure 8-4 shows the results. If a random user disguises and reveals their data (the common case), normal operations are mostly unaffected by concurrent disguising and revealing: throughput drops ≤4.2% without transactions and ≤7.2% with transactions (at high load). This shows that Edna's disguising and revealing transformations have acceptable impact on other users' application experience in the common case.

Constantly disguising and revealing the user with the most data (an expensive scenario) has a larger effect, with throughput reduced by up to 7.7% without transactions and up to 28.3% with transactions at both low and high load. This scenario disguises and reveals a user owning 1% of all the data in a database of 16k users, and demonstrates the upper bound on Edna's performance impacts when Edna runs large disguising and revealing transactions continuously. Because the queries within disguise and reveal transactions touch 16 tables, all which are commonly read by normal application operations, this impact is expected: transactions in MariaDB's InnoDB storage engine lock any written tables, preventing application operations from reading them until the transaction has completed. With transactions, the benchmark completes hundreds more disguising and revealing transformations within the same time period than when run without transactions. This illustrates how Edna's transactions, when touching large amounts of data, can block the execution of normal application operations. malte's comment about speed

The latency of disguising operations depends on load: on average, disguising and revealing transformations take <100ms. The expensive user's account removal and revealing take 2.7 and 5.4 seconds under high load. not comparable to facebook! This is acceptable: 50% of data deletions at Facebook take five minutes or longer to complete [**delf**].

8.4 Comparison to Qapla

This section compares Edna's performance and the effort to use Edna to an implementation of the same disguising and revealing functionality for WebSubmit using Qapla's query rewriting and access control policies. Unlike Edna, Qapla's goal is not to create abstractions for disguised data, but rather to enforce developer-specified data access policies. This section explores the benefits of Edna—a design specifically tailored for disguising and revealing—compared to a design that modifies an existing system (i.e., Qapla) to support disguised data.

8.4.1 Effort

Specifying disguising transformations as Qapla policies requires more explicit reasoning about transformations' implementations and their compositions. In Qapla, a developer would realize disguising transformations via metadata flags that they add to the schema (e.g., `is_deleted` for removed data) and toggles in application code. They then provision Qapla with a predicate that checks if this metadata flag is `true` before returning a row. Qapla's predicates grow in complexity with the number of disguising transformations that can compose. For example, an application supporting both account removal and account anonymization must combine predicates such that removal always takes precedence. Each additional transformation increases the number of predicates whose combinations the developer must reason about. This constrasts with Edna because developers working with Edna can add a transformation without thinking about how this transformation composes with others. Developers must also optimize Qapla predicates (e.g., reducing joins, adding schema indexes and index hints) to achieve reasonable performance (§8.4.2).

To modify data, the application developer can use Qapla's "cell blinding" mode, which dynamically changes column values (to fixed values) based on a predicate before returning query results. The developer must manually implement more complex modifications and decorrelation (i.e., creating pseudoprincipals and rewriting foreign keys).

Realizing WebSubmit transformations in Qapla required 576 lines of C/C++, and 110 lines of Rust to add pseudoprincipal, modification, and decorrelation support.

Overall, we found that Qapla requires more developer effort than Edna, particularly in writing composable and performant predicates, and manually implementing modifications and decorrelations. However, Qapla's approach does make some things easier. Because data remains in the database, revealing simply requires toggling metadata flags, and data to reveal can adapt to database changes (e.g., schema updates). But keeping the data in the database also means that developers cannot use Qapla to achieve GDPR-compliant data removal.

8.4.2 Performance

We measure Qapla's performance (Figure 8-5) on the WebSubmit operations evaluated in §8.2 (Figure 8-1a). Qapla performs well on operations that require only writes, since Qapla does not rewrite write queries. Removing and restoring accounts requires only a single metadata flag update in Qapla, whereas Edna encrypts/decrypts user data and actually deletes it from the database. However, Qapla rewrites all read queries, so Qapla performs poorly on operations that require reads, such as listing answers and editing (disguised or undisguised) data. Qapla's query rewriting takes ≈1ms, and rewrites `SELECT` queries in ways that affect performance (e.g., adding joins to evaluate predicates). Overall, Edna achieves better performance on read operations.

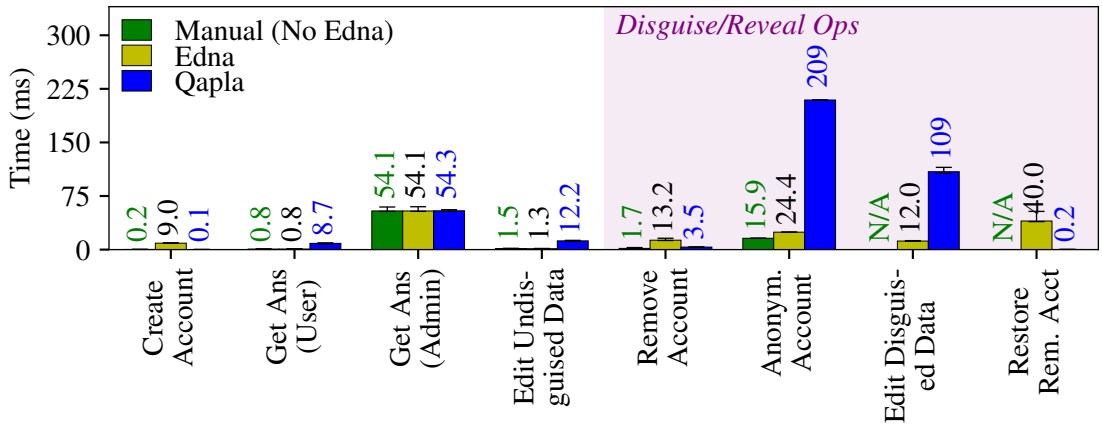


Figure 8-5: Edna achieves competitive performance with a manual baseline and outperforms Qapla on nearly all common WebSubmit operations (2k users, 80 answers/user). Bars show medians, error bars are 5^{th}/95^{th} percentile latencies.

8.5 Edna+CryptDB

We combine Edna with CryptDB to evaluate the cost of composing Edna's guarantees with those of encrypted databases. CryptDB protects undisguised database contents against attackers who compromise the database server itself (with some limitations [**grubbs**]), and therefore provides complementary guarantees to Edna-like protections for disguised data.

Edna+CryptDB operates in CryptDB's threat model 2 (database server and proxy can be compromised). A developer using Edna+CryptDB deploys the application (and Edna) atop a proxy that encrypts and decrypts database rows. Queries from Edna and the application operate unchanged atop the proxy, but to ensure proper access to user data, the application and Edna must handle user sessions. Edna+CryptDB handles keys in the same way as CryptDB: Edna+CryptDB encrypts database rows with per-object keys, and object keys are themselves encrypted with the public keys of the users who can access the object.

Edna+CryptDB exposes an API to log users in and out using their credentials. After a user logs in, the application gives the proxy their private key, thus allowing decryption of their accessible objects. Logging out deletes the user's private key from the proxy and reencrypts their object keys, thus preventing the proxy from accessing those objects. Prior to applying transformations to a user's data, Edna+CryptDB performs a login to ensure that Edna+CryptDB has legitimate access to their data (e.g., the user, an admin, or someone sharing the data is logged in).

The prototype supports only the CryptDB deterministic encryption scheme (AES-CMC encryption), which limits it to equality comparison predicates. It also does not support joins, a limitation shared with multi-principal CryptDB.

Performance. We measure the latency of WebSubmit operations like before, and compare a manual baseline, Edna, and Edna+CryptDB. Edna+CryptDB is necessarily more expensive than

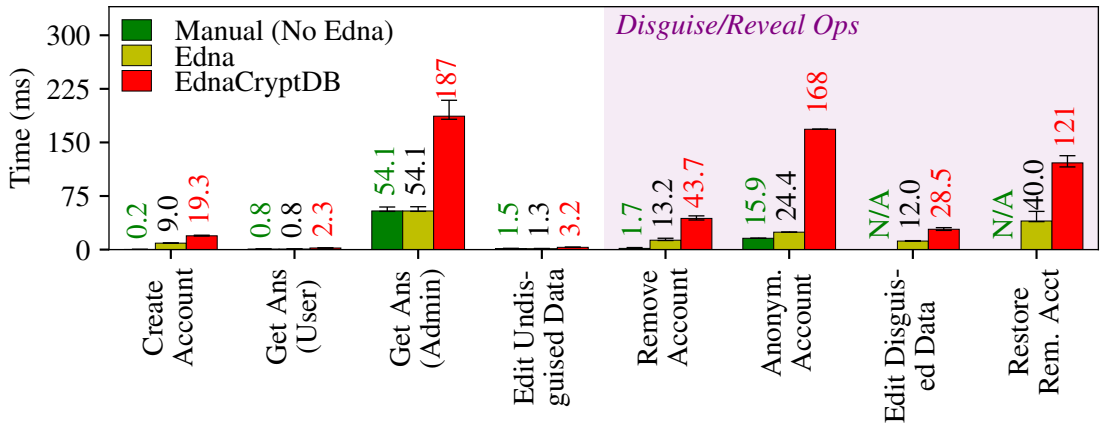


Figure 8-6: Latencies of WebSubmit (2k users, 80 answers/user) operations when implemented with Edna+CryptDB (adding encrypted database support). Bars show median latency; error bars are $5^{th}/95^{th}$ percentile latencies.

Edna, and a good result for Edna+CryptDB would therefore show moderate overheads over Edna, and acceptable absolute latencies.

Figure 8-6 shows the results. Normal application operations are 2–3× slower with Edna+CryptDB than in Edna, with the largest overheads on operations that access many rows, such as the admin viewing all answers. Disguising and revealing operations are also 2–6× slower than Edna.

These overheads result from the cryptographic operations and additional indirection in Edna+CryptDB. Edna+CryptDB relies on a MySQL proxy, which adds latency: a no-op version of this proxy makes operations 1.03-1.5× slower. Cryptographic operations themselves are cheap (< 0.2ms), but every object inserted, updated, or read also requires lookups to find out which keys to use, query rewriting to fetch the right encrypted rows, and execution of more complex queries.

This is particularly expensive when the user owns many keys (e.g., the WebSubmit admin). Admin-applied anonymization incurs the highest overhead (+143.6ms) as it issues many queries to read user data and execute decorrelations. Among the common operations, an admin getting all the answers for a lecture suffers similar overheads (+132.9ms).

Like CryptDB, Edna+CryptDB increases the database size (4–5× for our WebSubmit prototype). Edna+CryptDB also stores an encrypted object key and its metadata (1KB per key) for each user with access to that object.

8.6 Global Database Updates

Adding support for global database updates to disguised data during reveal may both increase developer effort and affect Edna's performance. This section answers the following two ques-

Attribute	Normalize URL	Add Story Texts	Add Show Email
Global Update	68 LoC	19 LoC	2 LoC
Update Spec	∅	54 LoC	27 LoC

Figure 8-7: Schema migrations that change the schema require writing separate reveal-time update specifications; however, updates like normalizing URLs that transform database table data can be used as update specifications.

tions:

1. What categories of global database update can Edna support, with how much extra developer effort?
2. What are the performance impacts of invoking Edna's API to record an update, and to apply updates during reveal?

8.6.1 Supporting Updates

Developers must already write their global updates today (e.g., Lobsters implements these updates as Ruby Active Record Migrations [**ruby_arm**]). However, depending on the type of update, developers may need to additionally write a reveal-time update specification for Edna to capture that operation. Schema changes (e.g., an `ALTER TABLE` command) require writing separate update specifications for Edna, which take as input a set of rows instead of the table. However, global updates may already operate on row data, and thus can be reused as an update specification to Edna.

The Lobsters case study implements the selected three global updates (§**??**) in Rust instead of Ruby Active Record Migration. Figure 8-7 shows the amount of code required per operation, as well as the additional code required to add a corresponding reveal-time update specification to give Edna. The reveal-time update specification to Edna reuses the global URL normalization update code, since the global update changes rows one by one, and thus requires no additional code. However, the two schema changes that respectively add `show_email` and add `story_texts` require the developer to write separate reveal-time updates that operate on individual rows, taking 54 and 27 LoC respectively. Invoking Edna with the three different global database updates for Lobsters required adding one line of code per update that the application calls when performing the update.

In addition to the updates described and implemented in §7.1, we inspected the 20 most recent global database updates in Lobsters from the past three years, and classified them as:

1. a data transformation (which can be reused as an update specification to Edna), such as URL normalization;
2. a schema change (which requires writing a new update specification for Edna), such as generating tables from other tables (e.g., `story_texts` from `stories`) or altering columns; or
3. an unsupported operation, such as updating the `vote_count` of comments based on the

Update Type	%	Examples	Example SQL
Schema Change	75	Create tables Alter table columns Add table indexes	`ALTER TABLE users ADD COLUMN show_email`
Data Transformation	15	Moderate text Normalize URLs	`UPDATE stories SET url= NORMALIZE(url)`
Unsupported	10	Update cached story vote count	`UPDATE stories SET vote_count=(SELECT COUNT(*) FROM votes WHERE votes.story_id = stories.id)`

Figure 8-8: Out of the most recent 20 Lobsters global database updates, the majority (75%) are schema changes which require developers to write separate update specifications for Edna. 15% are data transformations which can be reused as updates without additional effort, and 10% are unsupported because they rely on data external to the disguised data (values of external tables).

number of votes), which relies on data external to the data disguised.

Figure 8-8 shows the distribution of 20 global updates among these three categories. Edna supports all updates except those that update one table's row based on another row's contents. Rows outside of the row to reveal may have changed since the developer applied the update, and thus Edna cannot guarantee the correctness of these potentially nondeterministic updates at the time of reveal. In the investigated Lobsters updates, two updates out of the 20 were unsupported, and both updated the cached `vote_count` value of stories or comments. These types of updates might occur in applications that optimize for performance by denormalizing their database schema (as the Lobsters developers did); we hypothesize that this category of update would not exist with a normalized schema. However, these particular updates also happen to be idempotent for any rows affected, and thus could be reapplied via a cron job regularly to update revealed data.

Thus, for the large majority of global database updates, users can still reveal their data disguised prior to these migrations, as if the migration had occurred with their data present in the database.

8.6.2 Performance of Reveals with Updates

This section looks at how global updates affect application performance, and how logging reveal-time update specs affects Edna's reveal performance. Table 8-9 shows the results. Overall, the effects of performing updates increases with the amount of disguised data a user wants to reveal. Users with little data (e.g., fewer than 5 stories) experience no visible increase in reveal latency with updates, but users with lots of data can experience a noticeable effect (e.g., revealing a user with 673 stories, 1971 comments, and 2000 messages takes on average 4.67s instead of 4.50s, an increase of 3.8%). The rest of this section breaks down these costs.

Attribute	Normalize URL	Add Story Texts	Add Show Email
Global Update	5.63s	5.24s	0.082s
Record Update	0.82ms	0.49ms	0.43ms
Update 1 story	0.22ms	0.01ms	0.01ms
Update 1 user	<0.01ms	<0.01ms	0.01ms
Update other row	<0.001ms	<0.001ms	<0.001ms

Figure 8-9: Applying the various updates adds greater overheads per story than per user or per other table row, as more updates apply to the stories table. URL normalization is the most expensive update because initialization of a URL object takes 0.2ms.

Global Update Performance. First, as a baseline, this section looks at the cost of performing global updates, which applications already incur even without Edna (Table 8-9, line 1).

The global URL normalization update takes 5.63s, the longest of all updates. This matches the optimized global update execution time of 6s in the deployed Lobsters application, which normalizes the URLs of all stories in one batch.[1] URL normalization modifies all 120k stories.

Adding the `show_email` attribute to the `users` table takes 82ms, as it performs only one query to change the database schema.

Finally, creating and populating the `story_texts` table and removing a column from the `stories` table takes 5.24s. This modifies all 120k stories, and changes the database schema by adding tables and removing columns.

Record Update. Invoking Edna's hook to insert a new update specification in Edna's replay log takes on average 0.58ms (Table 8-9, line 2), a small amount compared to the time to perform the update/migration. This includes the time to persist the update (one database insert query) and add it to Edna's in-memory update replay log.

Reveal with Updates. Next, we evaluate the latency of reveal operations in Lobsters when updates corresponding to the implemented three global updates are applied after reveal. Edna applies update specifications to data to reveal in batch, but the cost of applying updates to revealed data should still be proportional to the amount of data to reveal.

First, we measure the cost to apply the updates to diff record rows. Updating `stories` diff records is the most costly, as both URL normalization and the `story_texts` updates apply: this takes 0.2–0.3ms per story (Table 8-9, line 3). Diff records for modified or decorrelated stories have both placeholder and original rows. Edna applies updates to both placeholder and original rows, and thus updates take 0.5–0.6ms per `stories` modify or decorrelate diff record. Initializing a URL normalizer object during the update takes a one-time cost of 0.2ms. Because Edna applies updates to batches of rows of the same table (§6.4), Edna amortizes the cost of URL normalizer initialization by creating a URL normalizer object only once per reveal (when

[1]The Lobsters developers first implemented an unoptimized version of URL normalization that normalizes stories one-by-one—this implementation took 1789s.

Edna reveals all stories).

Applying all three updates to `users` diff records incurs < 0.1ms per user: the `show_email` update appends a single column to user rows (Table 8-9, line 4).

Although the three implemented updates do not affect other row types, the implementation of Lobsters' updates does iterate through all rows to check whether to update them. Each row iteration takes $\approx 1\mu s$ (Table 8-9, line 5). At scale (e.g., for a user with thousands of comments), this cost can add up to a couple milliseconds over the entire reveal.

Finally, we measure the cost to restore updated rows to the database compared to the cost to restore rows without updates.[2] The cost to restore rows for all tables other than `stories` remains the same with or without updates, since the queries to restore the rows are the same. However, restoring a story now requires additional queries to perform consistency checks for and insert `story_texts` table rows, which add $\approx$0.6ms per story.

8.7 Summary

This evaluation used Edna to add seven disguising transformations to three web applications. The benchmarks show that the effort required was reasonable, that Edna's disguising and revealing operations are fast enough to be practical, and that they impose little to moderate overheads on normal application operation depending on the amount of data being disguised or revealed.

[2]Although restoring updated rows is more expensive than restoring the original rows, the application's global updates would have incurred the cost of updating these rows had the rows not been disguised at the time. Thus, Edna moves the cost of updating these rows to reveal time, rather than global updates execution time (although the cost may be higher because the update is not batched over all table rows).

Chapter 9

Conclusion

This thesis envisions a world in which web services routinely protect, store, and reveal disguised user data with user permission. To move towards this goal, this thesis presents Edna, a system that gives users flexible privacy control over their data via data disguising. With Edna, developers can provide data disguising and revealing transformations. This thesis illustrates how these transformations help users protect inactive accounts, selectively dissociate personal data from public profiles, and remove a web service's access to their data without permanently losing their accounts; and demonstrates that Edna can support disguised data in real-world applications with reasonable performance.

However, Edna is just the beginning. To truly achieve a world in which web applications meet the desired flexible privacy standards, several challenges remain unsolved, which we describe here as future opportunities for research.

9.1 Deploying Disguised Data

Supporting disguised data in web applications that have millions or more users, operate with low cost margins, and/or distribute their compute and storage across multiple regions requires addressing several issues.

Disguised Data Storage. If millions of users or more disguise their data over time, disguised data may become too much for applications to store. Even though storage is historically cheap, and applications today willingly store and retain user data nearly indefinitely, eventually disguised data (particularly since it cannot be sold or processed) may burden the application. Edna currently retains disguised data in its disguised table until a user reveals their disguised data, which could be forever if users choose to never reveal data. Instead, Edna could allow applications to put reasonable, coarse-grained time limits (e.g., 10 years) on disguised data to eventually clean it up, without leaking fine-grained information about which data was disguised at the same time.

Deriving Value from Disguised Data. Edna enables an application to keep as much data as possible if users wish to protect their privacy via disguising, and to allow users to return to the application. For many applications, this provides a better solution than handling user data deletion requests via permanent user deletion, or decentralized approaches that move all user data out of the application.

However, with Edna, applications store disguised data, but cannot derive value from the now-confidential data. Cryptographic approaches such as additive homomorphic secret sharing as done in Zeph [**zeph**], may potentially allow Edna to support limited computation such as aggregations over disguised data from which an application can derive value. However, such cryptographic approaches require some initial setup to determine whose data can be aggregated with whose (in order to split the secrets appropriately) and who should be able to reveal the result. Furthermore, defining groups of users whose data can be combined requires more thought into what these groups may reveal about their constituents.

Edna in Large-Scale Applications. Many web services, particularly those at scale, run on machines in multiple regions and datacenters, and replicate and shard their database. While distributed databases used by these applications (e.g., Spanner [**spanner**]) already implement parallel, distributed, and large-scale transactions, Edna's disguising and revealing transactions may potentially impose high burdens on distributed databases due to the size and frequency of these transactions. For example, a disguise could touch all database tables spread across multiple datacenters.

Today, many applications will already break down large transactions into smaller pieces (at the cost of transactional consistency) or run large transactions in batches overnight to reduce their impact on database performance. Similarly, we can imagine Edna breaking down large disguise and reveal transactions into multiple, per-table transactions, so as to not lock all tables at once; or delaying their execution until night and notifying the user once their disguise or reveal has completed.

9.2 Disguising Beyond Edna: Flexible Access Control

Disguising and revealing in Edna protects disguised data from any external party other than the user themselves. However, disguising and revealing can also apply in more nuanced settings, such as disguising data in different ways to protect from different parties. In these settings, disguising and revealing can offer potential opportunities for more flexible access control to control user data exposure to web application insiders.

As part of a collaboration with Hannah Gross [**funhouse**], we investigated using disguising abstractions to allow the application to control user data exposure to insiders. Such an approach could help organizations whose employees need to query data, but may not need to see all of it. Our work draws on the key observation that company employees do not *always* need all of the

user information that they can access. Just as Edna allows users to expose their data only while using the application, perhaps the application can use disguising and revealing to only expose user data to employees when they actually need and use it.

To achieve this goal, we envision a world where every employee operates atop a personalized disguised database that contains the minimum amount of data they need continuously. For example, a hotel customer service representative might only need to be able to look at room occupancy and see free rooms when not actively helping a customer. However, disguised databases would present only the minimal amount of data the employee needs; if, for example, a particular customer needs assistance, the employee currently helping the customer should be able to temporarily reveal parts of their disguised database to dynamically expose that user's data. Different employees simultaneously helping different customers can reveal different disguises to get a partially disguised view of the database individualized for that employee. As soon as the employee no longer needs the user's data, the employee's access returns back to its original, maximally disguised state.

To achieve this flexible access control, we need a database system that applies disguises and dynamically reveals them at a per-employee granularity. A strawman solution might attempt to do so with personalized database views (similar to the proposal in Multiverse DB [**multiverse**]), but this would require an enormous number of views (potentially one for each employee, and for each reveal the employee requests). Furthermore, just as with Edna, disguises cannot break the database: they would need to handle referential integrity, work even in the presence of indexes and database views, and work with existing query optimizations without overly affecting performance. Employees must also have the ability to flexibly perform fine-grained reveals of their disguised views without compromising security. While Edna's disguising and revealing abstractions can help address some of these challenges (e.g., handling referential integrity with decorrelation), achieving a database system for flexible access control requires solving many new challenges.

9.3 Final Thoughts

Today, user data increasingly lands in the hands of web services, which reduce or remove users' ability to control the privacy of their data as they wish. While Edna represents just one step forward in improving user data privacy on the web, this thesis and Edna thus demonstrate that with disguised data, applications can support flexible privacy features beyond what they offer users today.

Edna provides proof to both web services and regulators that flexible privacy features can be supported with reasonable effort, and thus can and should be widely implemented and enforced.