DELF: Safeguarding deletion correctness in Online Social Networks

Katriel Cohn-Gordon, Facebook; Georgios Damaskinos, Facebook, EPFL; Divino Neto, Joshi Cordova, Benoît Reitz, Benjamin Strahs, and Daniel Obenshain, Facebook; Paul Pearce, Facebook, Georgia Tech; Ioannis Papagiannis, Facebook

https://www.usenix.org/conference/usenixsecurity20/presentation/cohn-gordon

This paper is included in the Proceedings of the 29th USENIX Security Symposium.
August 12-14, 2020
978-1-939133-17-5

Open access to the Proceedings of the 29th USENIX Security Symposium is sponsored by USENIX.
**DELF: Safeguarding deletion correctness in Online Social Networks**

Katriel Cohn-Gordon  
*Facebook*

Georgios Damaskinos  
*Facebook, EPFL*

Divino Neto  
*Facebook*

Joshi Cordova  
*Facebook*

Benoît Reitz  
*Facebook*

Benjamin Strahs  
*Facebook*

Daniel Obenshain  
*Facebook*

Paul Pearce  
*Facebook, Georgia Tech*

Ioannis Papagiannis  
*Facebook*

---

**Abstract**

Deletion is a core facet of Online Social Networks (OSNs). For users, deletion is a tool to remove what they have shared and control their data. For OSNs, robust deletion is both an obligation to their users and a risk when developer mistakes inevitably occur. While developers are effective at identifying high-level deletion requirements in products (e.g., users should be able to delete posted photos), they are less effective at mapping high-level requirements into concrete operations (e.g., deleting all relevant items in data stores). Without framework support, developer mistakes lead to violations of users’ privacy, such as retaining data that should be deleted, deleting the wrong data, and exploitable vulnerabilities.

We propose DELF, a deletion framework for modern OSNs. In DELF, developers specify deletion annotations on data type definitions, which the framework maps into asynchronous, reliable and temporarily reversible operations on backing data stores. DELF validates annotations both statically and dynamically, proactively flagging errors and suggesting fixes.

We deployed DELF in three distinct OSNs, showing the feasibility of our approach. DELF detected, surfaced, and helped developers correct thousands of omissions and dozens of mistakes, while also enabling timely recovery in tens of incidents where user data was inadvertently deleted.

---

**1 Introduction**

The ability to delete data is a core privacy expectation for users entrusting Online Social Networks (OSNs) with their personal information [1, 2]. Users make use of deletion to retract posts shared with their friends, to avoid future conflicts, to forget past experiences, to remove content they shared by accident, to comply with policies of their organization, and to address perceived privacy or security concerns regarding providers they do not trust [3–6]. Deletion empowers users to safeguard their privacy in a straightforward way. It is increasingly enshrined worldwide as a user privacy right [7, 8].

Providing robust deletion infrastructure is important for service providers. On one hand, bugs in deletion undermine the integrity of the service when the wrong data is deleted; and may also manifest as exploitable vulnerabilities that allow users to delete arbitrary data. On the other hand, failing to delete user data undermines user trust and can trigger substantial regulatory fines [7, 8]. Both types of issues are common, affect numerous services, and are reported widely [9–16].

OSNs are particularly challenging applications in regards to identifying what to delete and when. In traditional communication services, such as email or messaging, data ownership is clear and limited to the items in a user’s inbox. Instead, the data model of OSNs is much more complex and changes frequently to support novel features. Billions of users interact on shared containers (e.g., profiles, groups, events, live videos, marketplace items, and stories) where they perform many different actions (e.g., comment, share, retweet, react, link, paint over, buy, watch, and upvote). Deleting a shared container (e.g., a group) should delete all of its subcontainers (e.g., all posts and photos in the group), and recursively delete all actions for each leaf container (e.g., all reactions, shares and upvotes of each photo) independently of who created the data. However, deleting a subcontainer (e.g., a retweet) should not delete the original container (e.g., the original tweet), in particular if this would allow malicious users to delete content they do not control. Identifying what to delete and when is a challenge [3, 17, 18], and is cited as an important reason not to trust that services delete data correctly [19, 20].

Our insight is that both developer input and control in the deletion process can and should be minimized. Rather than expecting developers to delete data correctly on their own, the process should be facilitated by a framework which executes all deletions in an application. Having centralized control of all deletions enables us to provide at the framework level three important features to safeguard correctness: (a) we can enforce that all developers specify how user data should be deleted before any data is collected, (b) we can validate developer specifications to surface mistakes early, and (c) when undetected mistakes inevitably occur we can recover any inadvertently deleted data with minimal engineering effort.

We demonstrate these ideas in DELF. DELF is a deletion...
framework that exposes a simple declarative API to specify deletion behavior, supporting OSNs’ complex data models and abstracting away heterogeneous storage infrastructure. It validates deletion specifications via static and dynamic analysis, helps developers correct mistakes, and executes all deletions to completion, despite any transient infrastructure errors. To our knowledge, DELF is the first framework that helps developers delete data correctly at scale.

We deploy DELF at FACEBOOK—a large OSN service provider—and explore its effectiveness. Via a case study of developer actions, we measure that even when forced to specify how data should be deleted during product development—a scenario that occurs about a hundred times every day at this service provider—developer precision is limited to 97.6%.

DELF detects the majority of the resulting mistakes with high confidence. DELF independently validates developer data models for deletion correctness and when it detects mismatches it raises these to developers for consideration. In our deployment we observe that due to DELF static validation developers change how 62.2% of the object types they create are deleted, while dynamic validation of edge types achieves 95.0% precision at 74.8% recall, i.e., DELF discovers how three quarters of all edge types should be deleted, and is correct 95% of the time, showing that it can independently pinpoint most developer mistakes when annotating edge types. In practice, DELF surfaced thousands of historical omissions and dozens of mistakes which developers corrected. When undetected mistakes lead to inadvertent data deletion DELF enables recovery with significantly less engineering effort.

The main contributions of this paper are:

1. We perform a case study of developer actions at FACEBOOK, quantifying the rate of mistakes developers introduce when asked to specify how data should be deleted in OSNs spanning tens of thousands of data types and hundreds of millions of lines of code.

2. We design DELF, an application-agnostic and robust framework for controlling deletion with restoration capabilities. We show how DELF simplifies and unifies the deletion process on top of distinct data store types, including a relational database, a graph database, a key-value store, and a blob store.

3. We demonstrate how DELF detects and helps developers correct common types of mistakes.

4. We deploy DELF at FACEBOOK. DELF detected thousands of omissions and dozens of mistakes which would have otherwise undermined deletion correctness resulting in privacy violations or vulnerabilities.

The rest of this paper is organized as follows. §2 introduces common types of data stores used to persist user data and lays out the deletion policy of one popular OSN service provider. §3 establishes our baseline in regards to observed frequency of developer mistakes in a large-scale codebase demanding complicated data models. §4 introduces the high level technical design of DELF and §5 discusses topics pertaining to its implementation. §6 assesses the effectiveness of our system in production. We close with a discussion of related work in §7, areas for future work in §8, our conclusions in §9, and we acknowledge contributions to our work in §10.

2 Background

Modern large-scale OSNs are supported by a variety of scalable persistent data stores [21–27]. Data stores expose different data models to optimize for the workloads required by the specific applications they target. It is common for a modern OSN to leverage multiple data stores simultaneously. For example, photos and videos may be persisted in a blob store, while social interactions, such as like or follow, may be persisted in a graph database.

We refer to application-level delete operations as subgraph deletions or just deletions. This is in contrast to row-level or object-level delete operations permitted by most data store APIs; we refer to those as point deletes.

2.1 Data Models

Scaling relational databases to handle large numbers of read and write is non-trivial [24, 28]. Product workloads in modern OSNs are read-heavy [21], their scale requires sharding user data across thousands of servers, and even data a single user creates is sharded across multiple servers to facilitate reads. For example, all comments on a post are typically stored on the same shard as the post for faster loading. In a sharded deployment multiple database servers follow the same database schema but each server stores only a subset of rows from each logical table [21, 29].

Many scalable data stores trade off advanced querying capabilities, support for transactions, or consistency of the full relational data model in favor of throughput, availability, and latency improvements possible with more constrained data models [23, 24, 26, 27]. Under a key-value model, data is indexed by arbitrary strings [22, 29]. Keys may be generated automatically [22] or chosen by the application [22, 29]. Values may be structured [27, 30] or unstructured [22, 25]. Under a graph model, data forms a graph [21, 31] whose main entities are objects and associations.

There are domain-specific data stores that empower specialized functionality within OSNs. Sets approximated by Bloom filters [32] or stream processing systems leveraging HyperLogLog [33, 34] store aggregate hashes of input and have applications in security, abuse prevention, analytics, and performance optimization [35, 36]. Data warehouse systems [37–39] store large amounts of logs and shard based on time to facilitate daily batch processing for analytics and machine learning. In such domain-specific data stores, indexes
to enable point queries may be prohibitively demanding—
fre quently they are not available at all. Suggested techniques
to address deletion when point deletes are not feasible are
storing all data with short retention, anonymization, and en-
cryption at write time with a key that can be deleted sepa-
rately [40,41]. The rest of this paper focuses on deletion from
relational, key-value, and graph data stores where indexes to
perform point deletes are available.

2.2 Dangling Data

We describe a reference to a deleted object and the correspon-
ding object storing such a reference as dangling. Dangling data
conveys information about deleted objects, e.g., a key-value
entry linking a phone number to a deleted account may retain
how to contact the account and a graph association from an
account to a deleted video may retain who watched the video.
For correct deletion no dangling data should remain.

Relational databases rely on integrity constraints [42,43]
to achieve referential integrity and identify what should be
deleted once a row is deleted. With foreign key declara-
tions and appropriate indexes in place, a relational database
propagates point deletes for rows on the parent table to cas-
cade and delete dangling rows in child tables. Developers
control this process via referential actions on foreign key
declarations, such as ON DELETE CASCADE and ON DELETE
SET NULL. There is no guarantee that developers define ei-
ther foreign keys or referential actions correctly. There is no
mechanism to detect omissions. Modern popular shared data
stores such as MongoDB [44], Dynamo [29], and Redis [30]
offload enforcing referential integrity to applications [45].

2.3 Recovery via Backups

Data store backups enable service providers to recover from
hardware failures, system crashes, and application-level bugs.
In a typical configuration a full database snapshot is sched-
uled periodically [46]. The data store is separately configured
to log incremental mutations [47]. To recover the data store
to any point in time a full snapshot is restored and any sub-
sequent incremental mutations are replayed. Reverting only
specific deletions is not practical without additional informa-
tion, since incremental mutations do not store metadata about
application-level actions [48]. We illustrate these challenges
in the context of a data loss incident in our case study (§3).

2.4 FACEBOOK

FACEBOOK is a service provider in the space of social net-
working. Its products collectively have approximately 3 bil-
lion monthly active users [49]. FACEBOOK products in-
clude multiple distinct consumer OSNs, such as Facebook
(the OSN), Instagram, and Dating, with a variety of features

---

Figure 1: A Photo object type definition for storing photo
metadata in TAO (line 4) with an edge type to the photo blob
object in Everstore (line 13). DELF object type (line 5) and
dge type (lines 14, 17, 20) annotations specify how data
should be deleted when the data type is defined.

spanning—amongst others—private and public media sharing,
messaging, groups, video streaming, and a marketplace.

Infrastructure. In the backend FACEBOOK products define
tens of thousands of distinct data types to empower external-
facing features across existing products, new products un-
dergoing testing, and internal tools. Major data stores are
TAO [21], Everstore [22], MySQL [50] and ZippyDB [26]; a
graph, blob, relational, and key-value data store, respectively.
None of these data stores enforces referential integrity for
references across shards and across data stores. Objects of
several popular data types, such as photos, videos, and group
posts, may get deleted as a result of dozens of actions.

FACEBOOK infrastructure requires developers across most
products to define their data types before they are used in a
structured format, at minimum exposing object types con-
nected via edge types. Figure 1 presents an example in pseu-
docode of such a definition. The implementation depends
on the backing data store. For example, MySQL maps an
object type to a table and an edge type to a different table
with columns to store the primary keys of the referenced ta-
bles; TAO maps objects and edges to objects and associations
directly. A subsequent code generation step creates imple-
mentation classes with strongly-typed read, write and delete
methods for common languages used to develop applications.
This intermediate abstraction layer for defining and manipulat-
ing data types is similar to object-relational mapping [51]. It
facilitates access control [52] and improves performance [26].

Deletion Policy. FACEBOOK’s deletion policy prescribes that
users can exercise control over content they provide by delet-
ing it. Users may explicitly delete individual pieces of content
or their account. Data types such as ephemeral or draft posts
are automatically deleted after a fixed time period.

Deletions should be fully effected within 90 days, after
which the data can no longer be used in products or services
in the ordinary course of business. Most deletions involve
a relatively small amount of data and should finish quickly,
i.e., within one day. 90 days provides sufficient time to delete
an account despite transient errors. Deleted data may subse-
---
quently persist in backups for up to 90 days for recovery from inadvertent deletions or other infrastructure failures.

A few deletions may take more than 90 days to complete. Typical reasons entail deleting an unusually high number of pieces of content, e.g., an account which has been creating content consistently over many years, and persistent infrastructure failures. In such cases deletions may run for more than 90 days but are required to make continuous progress towards completion. Any failures must be resolved.

2.5 Threat Model

DELF safeguards the deletion process against developer mistakes. In our threat model developers are employed by their organization and aim to uphold its deletion policy, but may erroneously—not maliciously—fail to do so in practice. We consider the following types of mistakes.

Developers may altogether omit to specify what to delete when a user triggers a deletion. For example, developers may add a new edge type from the photo data type to a user data type and store which users are tagged in the photo, but omit to implement deletion of the edge when the photo is deleted.

Developers may specify to retain data that should be deleted according to the deletion policy. For example, developers may opt to retain who voted in a poll after a voter deletes their account to ensure that poll results cannot change retroactively, overlooking that users should be able to delete any data they provide including how they voted in polls.

Developers may inadvertently delete the wrong data. For example, developers may specify that deleting a comment should entail deleting all its attachments, such as photos. Yet they may overlook that users can attach third-party photos to their comments and these will get deleted too. Mistakes can introduce security vulnerabilities. We consider users to be potentially malicious. For example, an adversarial user may try to delete arbitrary photos by attaching a photo to a comment they create and then deleting the comment.

Developers may fail to execute the specification they have provided. For example, developers may attempt to delete all comments when a post is deleted, but they may not anticipate that the list can include millions of items, that the process may take days, and that any data store may temporarily or permanently fail to delete individual comments.

Malicious developers are outside the scope of this work. In our experience the vast majority of developers faithfully try to implement their employer’s policy and are subject to disciplinary action if they systematically fail to do so. As we demonstrate next, without ongoing detection benign developer mistakes account for frequent bugs in deletion.

3 Case Study: Unassisted Deletion

We motivate the need for DELF by conducting a case study within FACEBOOK. We measure (a) how likely developers are to remember their obligation to delete data (§3.1), (b), whether they do so correctly (§3.2), and (c), the operational overhead of recovering from inadvertent deletions without framework support (§3.3). To our knowledge, this is the first study to measure such developer actions.

3.1 Developer omissions

We measure developer proactiveness specifying how collected data should be deleted, i.e., without enforcement from a framework. We look at the deployment phase of DELF in Instagram starting on April 2019. To facilitate backwards compatibility, developers were able, for a limited period of time, to define a certain category of new TAO edge types without specifying how any collected data should be deleted in advance (§5.4). When developers omitted to provide deletion specifications they were reminded to do so retroactively (§4.3).

Between April and July 2019 40 distinct Instagram developers introduced 70 new edge types without enforcement from DELF. We inspect each of them, finding that 32 distinct developers created 56 new edge types and did not remember to update the corresponding code to handle their deletion when either of the objects they reference is deleted. In effect without any enforcement developers handled deletion proactively for only 20% of the edge types they created.

We attribute limited developer proactiveness to the lack of feedback triggered by development tools while prototyping new features. The situation has parallels with common types of security concerns; in particular managing memory without help from a framework [53,54]. Developers can store data persistently (resp. allocate memory) and—assuming sufficient storage capacity (resp. memory)—they observe no failures if they forgo to specify how their application should behave when deletes occur (resp. when in-memory objects are no longer referenced). For memory management, common techniques forgo developer education and automate the process fully [55], or expect developers to specify application behavior ahead of time before memory is allocated [56]. For data deletion, no similar tools exist (§2.2). Another contributing factor is deletion seldom being a driving requirement while prototyping new features. It is common for deletion to only be introduced as a requirement retroactively and only after core pieces of functionality have already been implemented.

3.2 Developer mistakes

To prevent dangling data when a new edge type between a source and a target object type is introduced, developers need to specify what should happen if the source object is deleted. Developers may opt to delete or retain the target object and their choice is subject to peer review. In the next section we discuss in detail how developers achieve this via DELF edge type annotations (§4). Here we measure the precision developers achieve in the task when unassisted.
We highlight the operational overhead and risk introduced by
when Facebook developers discovered a bug causing inadver-
tent deletion of hundreds of millions of videos and performed
restorations from backups without framework support. For each annotation the expert considered incorrect we surfaced the issue with the original changeset authors or peer reviewers and established ground truth.

Table 1 summarizes our results. Developers misclassified edge types demonstrating an overall precision of 97.6%, with mistakes leading to inadvertent retention and mistakes leading to inadvertent deletion occurring at similar rates. Reasons for mistakes included (a) a developer confusing the direction of deletion for a pair of edge types, (b) two developers copying annotations without confirming correctness, (c) a developer prototyping a new feature who intended to revisit annotations at a later stage, and (d), a developer who had not thought through all scenarios that should trigger deletion. While we were not able to construct exploits for the 5 edges incorrectly annotated deep in our sample (Table 1, shallow false positives), we anticipate that a proportion of such mistakes will be exploitable externally, i.e., they can be exploited to delete data without validating necessary permissions.

Using the rate at which edge types are being introduced in FACEBOOK infrastructure at the time of our case study, we interpolate that developers incorrectly annotate approximately 2 edge types every day. In absolute numbers, mistakes that result in inadvertent deletion are approximately twice as common than those that result in inadvertent data retention.

3.3 Recovery

We collect all changesets introducing at least one new edge type annotation submitted between June 24 and June 27 2019, totaling 327 changesets created by 129 developers, and for each changeset we request retroactive expert review. The expert is a tenured privacy engineer with extensive experience in deletion, FACEBOOK’s deletion policy, products and infrastructure. For each annotation the expert considered incorrect we surfaced the issue with the original changeset authors or peer reviewers and established ground truth.

Table 1 summarizes our results. Developers misclassified edge types demonstrating an overall precision of 97.6%, with mistakes leading to inadvertent retention and mistakes leading to inadvertent deletion occurring at similar rates. Reasons for mistakes included (a) a developer confusing the direction of deletion for a pair of edge types, (b) two developers copying annotations without confirming correctness, (c) a developer prototyping a new feature who intended to revisit annotations at a later stage, and (d), a developer who had not thought through all scenarios that should trigger deletion. While we were not able to construct exploits for the 5 edges incorrectly annotated deep in our sample (Table 1, shallow false positives), we anticipate that a proportion of such mistakes will be exploitable externally, i.e., they can be exploited to delete data without validating necessary permissions.

Using the rate at which edge types are being introduced in FACEBOOK infrastructure at the time of our case study, we interpolate that developers incorrectly annotate approximately 2 edge types every day. In absolute numbers, mistakes that result in inadvertent deletion are approximately twice as common than those that result in inadvertent data retention.

## 4 Design

**DELF** forces developers to specify how data is deleted when data types are defined. It achieves this by introducing annotations related to deletion to a domain-specific language used to define data types. **DELF** then intercepts application-level deletions and transparently executes them to completion independently of the underlying data store. **DELF** offers two safety mechanisms: the ability to verify correctness of developer annotations and undo deletions for a short time period.

### 4.1 Deletion Specification

**DELF** forces developers to specify deletion annotations for all new object and edge types they create. The data type definition step is instrumental to **DELF**’s design. The edge type definitions in particular provide a statically-known list of all potential references between objects. When an object is deleted, **DELF** enumerates all potential data stores where dangling data may reside based on edge types and deletes it all according to developer annotations. Edge types enable **DELF** to perform subgraph deletions and delete dangling data.

Table 2 summarizes **DELF** annotations, categorized based on their applicability and purpose. *Edge* annotations apply on edge types while *object* annotations apply on object types. The goal of an annotation is to increase deletion coverage, i.e.,
not leave dangling data, or to prevent inadvertent deletions, i.e., to preclude deleting the wrong data. Figure 1 provides example annotations for the Photo object type from §2.4.

**Edge type annotations.** These specify what happens to referenced objects upon deletion. Each edge is unidirectional, pointing from a source object to a target object. An edge type annotation prescribes the expected deletion behavior once the source object gets deleted. Developers choose between deleting only the source object (shallow), cascading and deleting the associated object by following the edge (deep) and cascading only when the last edge to the target object is deleted (refcount). All edge type annotations result in the deletion of the edges themselves. Edge annotations improve deletion coverage because they force developers to declare how referenced data should be deleted.

**Object type annotations.** These specify how objects of a particular type should be deleted. By default DELF assumes that all object types contain data users create therefore objects of every type should be deletable in some form. There are three main object annotations. The default annotation is by_any. It requires that each object type declares at least one deep-annotated inbound edge type from another object type; thus individual objects of this object type are deleted via traversal of that edge type. Developers may pick instead short_ttl or directly. The former ensures that all objects of this type get deleted by virtue of limited retention—the precise maximum retention allowed should be consistent with the service provider’s deletion policy. The latter is appropriate for object types that users can delete via direct action in product, e.g., object types designating user accounts.

DELF exposes three object annotations that help protect objects against inadvertent deletions trigged by deleting objects of other types. The by_x_only annotation is a restricted form of by_any. It is parameterized by a whitelist of edge types that may trigger deletions of objects of this type. DELF prevents developers from accidentally declaring deep edges of any other type not found in the whitelist. The directly_only annotation is a more restrictive form of directly; DELF prevents any inbound edge types to be marked deep. The not_deleted annotation prevents all deletions altogether by both rejecting all inbound deep edge types and by not generating code to perform object deletes. To prevent overuse of not_deleted, DELF requires developers to reference a documented privacy or legal decision which mandates retaining the data, e.g., a task in the service provider’s tracking system.

The custom annotation allows developers to provide arbitrary procedural code to execute when objects of a given type are deleted. Unlike declarative annotations, code in a custom section can express complicated deletion logic dependent on arbitrary state. For example, code in a custom section may inspect the object being deleted and delete one of its edges with either shallow or deep semantics based on the value of a particular object field. Executing procedural code at deletion time enables additional expressiveness which may be necessary for certain data types yet its use is heavily discouraged. Procedural code in custom sections precludes correctness validation (§4.3), is hard for developers to keep up to date (§6.1), and bugs may result in stuck deletions that do not make progress (§6.4). DELF supports writing procedural code in custom sections for backwards compatibility with legacy deletion logic and data models while applications migrate to object and edge type annotations (§5.4).

### Table 2: DELF annotations allow developers to control deletion. They apply on either object or edge type definitions and can be validated via different methods to prevent dangling data and inadvertent deletions (§4.3)

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Applies on</th>
<th>Goal</th>
<th>Validation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>shallow</td>
<td>Edge Types</td>
<td>Coverage</td>
<td>Dynamic</td>
<td>When deleting the source object delete only the edge.</td>
</tr>
<tr>
<td>deep</td>
<td>Edge Types</td>
<td>Coverage</td>
<td>Dynamic</td>
<td>When deleting the source object cascade and delete the target.</td>
</tr>
<tr>
<td>refcount</td>
<td>Edge Types</td>
<td>Coverage</td>
<td>Dynamic</td>
<td>Cascade only when the last source object gets deleted.</td>
</tr>
<tr>
<td>by_any</td>
<td>Object Types</td>
<td>Coverage</td>
<td>Static</td>
<td>One or more inbound edge types should be deep.</td>
</tr>
<tr>
<td>short_ttl</td>
<td>Object Types</td>
<td>Coverage</td>
<td>Static</td>
<td>The object type should specify limited retention.</td>
</tr>
<tr>
<td>directly</td>
<td>Object Types</td>
<td>Coverage</td>
<td>Static &amp; Dynamic</td>
<td>Objects are deleted via direct action in product.</td>
</tr>
<tr>
<td>directly_only</td>
<td>Object Types</td>
<td>Prevention</td>
<td>Static</td>
<td>Deletions may be registered to start at an arbitrary point of time in the future. Developers can achieve this at object creation time by virtue of creating an object of a type under TTL.</td>
</tr>
<tr>
<td>not_deleted</td>
<td>Object Types</td>
<td>Prevention</td>
<td>Static</td>
<td>Prevents objects of this type from being deleted.</td>
</tr>
<tr>
<td>custom</td>
<td>Object Types</td>
<td>Coverage</td>
<td>N/A</td>
<td>Developers specify arbitrary procedural deletion logic in code.</td>
</tr>
</tbody>
</table>

#### 4.2 Deletion Execution

Figure 2 presents the timeline of a deletion in DELF. Deletions go through the stages of registration ($t_1$–$t_3$), initiation ($t_3$–$t_4$) and asynchronous execution ($t_4$–$t_5$). Data retained in restoration logs and backups is deleted once a deletion finishes after a fixed interval ($t_5$–$t_6$).

- $t_1$ marks object creation. Deletions may be registered to start at an arbitrary point of time in the future. Developers can achieve this at object creation time by virtue of creating an object of a type under TTL.
- $t_2$ marks explicit actions to schedule objects for deletion in the future by using a DELF-provided method, e.g., when a
4.3 Deletion Validation

Dangling data and inadvertent deletions may occur for a variety of reasons including missing edge or object type annotations, mistakes in annotations, insufficient authorization checks, and developers storing references to other objects without declaring edge types. DELF introduces validation methods and drives mitigation for each of these types of mistakes. Every DELF object and edge type annotation is validated with at least one method, as designated in Table 2.

**Static validation.** DELF confirms that there is at least one possible path to delete data of every defined object and edge type. This is checked statically: (a) DELF rejects any data types found to lack annotations, and (b), DELF performs a reachability analysis starting from every object type annotated with `directly`, `directly_only`, `short_ttl`, and `not_deleted` visiting all their edge types annotated `deep`. The analysis must reach all defined object types in the system. Any object types not reached are part of a cycle without at least one declared entry point to delete it. Unreachable object types are rejected. By including `not_deleted` types as starting points the reachability analysis transitively treats any referenced object types as valid exceptions from deletion. Unreachable data types annotated with `custom` are similarly ignored and their correctness is only verified with peer review.

**Dynamic validation.** DELF introduces three dynamic validation methods. The first method confirms that objects of types annotated with `directly` and `directly_only` are observed to be deleted at runtime. The process inspects logs of all deletions executed in DELF per object type and confirms that developers follow up and expose accessible entry points in product to trigger deletions. Runtime validation of these two annotations guarantees that all declared paths to delete data types are triggered by users in production.

The second dynamic validation method is a set of heuristics to retroactively annotate edge types and detect edge types misclassified by developers. Deep heuristics suggest that
a particular edge type should be annotated as deep—they surface dangling data occurring due to edge types misconfigured as shallow or refcount. In contrast to static validation which ensures there is at least one path to delete objects of every type, deep heuristics attempt to discover all paths. Shallow heuristics instead suggest that an edge type should be annotated as shallow—they provide a proactive detection method for edges misconfigured as deep or refcount. DELF surfaces edge types it detects to be misclassified, notifies developers, and recommends changes.

Edge type annotation heuristics leverage features collected at runtime after data has been collected. In particular, the heuristics to_new_object and to_old_object inspect the edge and target object creation timestamps. If all edges of a particular type are found to consistently be created at the same time as the target object, this is an indication that the edge and target are created together and hence should be deleted together—DELF suggests deep. Similarly, if all edges of a particular type are found to consistently be written at a later time compared to the target object, this is an indication that the target object predates the edge and hence should persist after the edge gets deleted—DELF suggests shallow.

Tables 3 and 4 list all of the edge type classification heuristics used by DELF. Figure 3 illustrates an example where to_new_object is applicable—DELF predicts deep. Figure 4 illustrates an example where many_to_one is applicable—DELF predicts shallow.

The third dynamic validation method is a check for privilege escalation before writing edges of all types annotated with deep. A typical exploit of deep edge types involves two steps: (a) writing an edge from an object under attacker control to a target object in the system, and (b), deleting the object under attacker control hence—as a side effect of deep—deleting the target object. The attack leverages application endpoints where application authorization checks for writ-

---

1DELF does not currently offer heuristics to suggest refcount edge types since these are substantially less common than deep and shallow (§3.2).

Figure 3: Applying to_new_object on the created_photo edge type (Figure 1). If the creation timestamps (ts) of all edges match the creation timestamp of their target object the edge type should likely be annotated deep.

Figure 4: Applying many_to_one on a likes edge type. If multiple different objects all have edges to the same target object, the edge type should likely be annotated shallow.
Any failing restorations, e.g., if certain objects cannot be restored, are surfaced to an engineer. Some manual effort is justified since restorations are used only for disaster recovery. The deletion and restoration traversals collectively ensure that outbound edges are consistent between the time an object is restored and the time the object is restored, i.e., any outbound edges can be fetched and referenced objects are restored before source objects. Restorations fail early if that is impossible, e.g., a user subsequently claims the phone number the deleted account used to log in. Restorations should also not surface partially restored data to users.

Delif makes restorations safer via a staged restoration process. Delif traverses the serialized restoration log in reverse creation order. Figure 5 illustrates both the deletion and the restoration graph traversals. The deletion traversal is depth-first with deletion of objects pre-order and outbound edges post-order. Restorations traverse log entries in reverse. The restoration traversal ensures outbound edges and target objects are restored before source objects. Restorations fail early if that is impossible, e.g., a user object will only be restored if the restoration of the phone number succeeds.

The deletion and restoration traversals collectively ensure that outbound edges are consistent between the time an object is deleted and the time the object is restored, i.e., any outbound edges can be fetched and referenced objects are available. Consequently, the same read-time checks used to achieve quick hiding during deletion execution (§4.2) ensure that partially restored subgraphs are not visible to users.

Delif retries restorations indefinitely until they complete. Any failing restorations, e.g., if certain objects cannot be restored, are surfaced to an engineer. Some manual effort is justified since restorations are used only for disaster recovery.

4.5 Discussion
Delif requires developers to annotate every object and edge type they create; an additional step during product development which can be perceived as superfluous or error-prone. However, assuming correct deletion is a core product requirement, Delif offers a robust implementation approach.

Delif highlights deletion as a core requirement to developers early while developing new product features. Static validation in particular surfaces omissions and mistakes within regular development tools. Developers undergo ongoing education by virtue of understanding and resolving surfaced errors. No separate education process is necessary.

Developers are only expected to provide annotations when data types change. These events are typically much less frequent than subsequent changes in product behavior. At Facebook, for example, we observe that changesets altering data types (§3.2) are an order of magnitude less frequent than changesets altering product behavior [58].

Delif annotations simplify deletion correctness validation for both human developers at code review time and—as we demonstrated in §4.3—for automated methods. To validate correctness developers and automated methods can inspect Delif annotations only, avoiding the laborious, error-prone alternative of having to infer deletion semantics by inspecting the product implementation directly. Delif simplifies peer review and complements it with automated validation.

Delif overall reduces product complexity and speeds up product development by eliminating the need to write an maintain procedural, custom deletion code. In §6.1 we show that in a scenario where developers have the ability to bypass Delif and implement deletion as they wish, most do not, suggesting that Delif is the preferred, straightforward choice.

5 Implementation
Delif’s deployment at Facebook supports user-facing deletion functionality in Facebook, Instagram, and Dating, including account deletion and the deletion of individual items such as posts. In this section we provide implementation information, pertaining to how the system achieves reliable execution of all deletions, maximizes throughput, and limits retention of restoration logs. We cover last the development and deployment sequencing of Delif at Facebook.

5.1 Redundant Deletion Tracking
All deletions should complete despite intermittent failures in underlying infrastructure. There are three important reliability concerns: (a) all deletions start on time, (b) no deletion remains idle, and (c) deletions make progress when they run. Typical issues include service outages, transient overload of data stores or the asynchronous execution tier, bugs in custom sections, and corrupt data. Failures should be noticed
even if they affect a small number of deletions and are within expected failure rates of the underlying systems.

DEL F performs redundant tracking of all deletions using the analytics infrastructure at FACEBOOK. The tracking is orthogonal to the state kept by the underlying batch processing system and data stores which are responsible for scheduling and executing deletion jobs. DEL F logs all events relevant to deletion lifecycle including the scheduled start time, initiation, subsequent reruns, all exceptions, and eventual completion alongside timestamps. Event logging occurs via Scribe [59], and a Hive [39] pipeline inspects all events logged per deletion to identify anomalous deletions in regards to timely initiation, idleness, continuous progress, and completion. Any deletions found not to make progress are reported for engineers to investigate and resume automatically once fixes are deployed.

5.2 Throughput

DEL F aims to minimize the end-to-end execution time for each deletion, which is the main system performance consideration (§2.4). Consequently, DEL F executes deletions for different top-level objects in parallel and batches point deletes within each deletion. DEL F maximizes throughput, i.e., the rate of point deletes against data store APIs.

The upper bound for aggregate throughput is imposed by shard utilization. The same shards DEL F deletes data from serve production traffic and their performance should not degrade due to asynchronous deletion execution. DEL F monitors replication lag and CPU utilization to detect highly utilized shards and applies exponential back off on spikes. Another limiting factor can be the number of available machines to execute deletion jobs; DEL F shifts deletion execution to run off-peak when necessary.

Deletions triggered by users are executed immediately and in parallel with other existing deletions in the system. The average deletion at FACEBOOK involves few objects, e.g., deleting a rejected friend request. DEL F favors such deletions because they are executed without any coordination with existing deletions beyond an initial check to confirm no two deletions operate on the same top-level object during the same time. The resulting point deletes are spread across shards.

DEL F batches writes within each deletion, i.e., point deletes and writes to restoration logs. Batching amortizes write latency and increases throughput. Writes entail cross-regional latency due to either a roundtrip to the master region or to achieve consistency across replicas. To amortize this, each deletion reads items to delete from local replicas, collects those in memory, and once the batch reaches a pre-configured size all deletes are flushed concurrently. Each batch of point deletes entails a single write for a batch of restoration log entries. DEL F also batches reads to increase throughput further.

5.3 Restoration Logs Retention

Long-running deletions which remain in asynchronous execution for more than 90 days are required to make continuous progress (§2.4). To satisfy this requirement DEL F’s deployment at FACEBOOK does not apply a single retention period for the entire restoration log of each deletion, e.g., 90 days from the last log entry. Instead each log entry is retained for 90 days after its creation. Deletions running for more than 90 days may therefore not get restored fully since log entries persisted more than 90 days in the past will have been deleted. Data store backup retention matches restoration log entry retention with each snapshot being retained for 90 days. The setup guarantees that data stored in restoration logs and backups is deleted 90 days after each point delete.

Restoration logs should not be retained beyond 90 days. Persisting log entries in a data store which itself maintains backups must be avoided to not extend retention. DEL F uses Everstore and ZippyDB to handle the desired throughput. Yet both data stores mandate backups for all use cases to safeguard against bugs in the data store itself. DEL F, instead, relies on encryption to enforce precisely 90 days of log entry retention. Restoration log entries are encrypted using AES-256-CBC with HMAC-SHA-256 for authentication. The encryption key is stored in memory for 90 days, protected from inadvertent logging, and rotated daily.

5.4 Deployment sequencing

DEL F was iteratively developed at FACEBOOK over several years; progressively gaining its key design properties and coverage across data stores. We discuss major phases in its deployment alongside improvements delivered in each phase.

In Phase 1, DEL F replaced product code performing deletes to data store APIs directly, mandating the use of a DEL F-provided procedural API which performed the same deletes while transparently maintaining restoration logs. This phase mitigated developer mistakes leading to inadvertent deletion.

In Phase 2, DEL F introduced dynamic validation techniques (§4.3). This phase enabled detection of developer omissions and mistakes leading to inadvertent data retention. DEL F heuristics enabled remediation when detecting omissions by pinpointing mishandled edge types to developers.

In Phase 3, DEL F introduced its declarative API based on object and edge annotations (§4.1). Applications hosted in FACEBOOK infrastructure rely on two distinct proprietary data definition languages to create data types across TAO, Everstore, MySQL, and ZippyDB in line with DEL F’s requirements. We extended both to support DEL F annotations. Developers were able to use—optionally—the declarative API rather than the procedural API introduced in Phase 1. This phase helped speed up the development of new products by eliminating the need for writing procedural deletion code.

In Phase 4, DEL F introduced static and data type validation
techniques (§4.3); while the use of the declarative API became mandatory. This phase helped developers catch mistakes early in the product development process when data models are defined and improved correctness validation capabilities. It also reduced the operational overhead of DELF by making stuck deletions which do not make progress less likely (§6.4).

6 Evaluation

Our goal in this section is to quantify DELF’s ability to mitigate the privacy and security concerns raised in our threat model. For each concern we discuss identified issues flagged by DELF during its deployment and we then quantify the system’s ongoing prevention capabilities. All identified issues have been fixed and any retained data deleted.

Experiments in this section involve instrumenting the deletion process at FACEBOOK to assess the effectiveness of DELF under real system operation. We design all our experiments to avoid incurring any adverse effect in FACEBOOK’s ability to enforce its deletion policy across its products.

6.1 Developer Omissions

We start by assessing DELF’s impact in helping developers remember their obligation to delete user data. This assessment draws upon data collected between May 2018 and April 2019. Identified issues. During the course of our assessment DELF via data type validation (§4.3) detected 3266 instances where developers omitted handling deletion of an edge type. All identified cases result in inadvertent data retention independently of the eventual edge type annotation—a retained edge itself stores data about deleted objects. We routed all identified omissions to developers for retroactive annotation.

DELF identified one broad category of developer omissions responsible for the majority of issues in our results. The prevailing scenario involves deletions being driven via by procedural code in custom sections which developers failed to keep up to date when the applications or data models change; resulting in dangling data. At the time of this assessment the transition to DELF was ongoing (§5.4) and procedural code in custom sections handling deletion was common. To better understand identified omissions we discuss two examples.

In June 2018 DELF flagged that an edge type indicating the existence of a mailbox is being left dangling when a Dating user is deleted. The edge type was created in April 2015 and was initially only used for Facebook users. Yet in November 2017 developers introduced a new user object type to represent users of the upcoming Dating product and reused the same mailbox edge type to implement its messaging functionality. DELF detected the edge type reuse and highlighted the missing edge type annotation for the new user object type. The resulting investigation uncovered that: (a) the process of mailbox deletion relied on custom procedural deletion code invoked when Facebook user object types are deleted, and (b), developers omitted to update this logic to handle cases where a Dating user is deleted. The bug was identified during the internal beta testing period for Dating prior to launch. If it remained undetected it would have resulted in retaining all private messages Dating users exchanged post account deletion for people who delete their account. An important followup was the removal of the procedural deletion code controlling the invocation of mailbox deletion on account deletion and replacing it with a deep-annotated edge type between any user type and its mailbox. In subsequent months DELF seamlessly handled mailbox deletion for an additional 4 new user account types introduced in FACEBOOK.

In May 2018 DELF flagged an edge type storing the most recent pages a user views is being left dangling when some Facebook users are deleted. The edge type was created in November 2013 and data was used ever since to generate recommendations for accounts to follow. Developers initially ensured that edges of this type are deleted when a Facebook user deletes their account via updating the custom procedural deletion logic used at the time. Yet in May 2018 DELF detected that the same edge type was subsequently reused to log page views for a different type of user accounts in Facebook, i.e., page admins. The subsequent investigation confirmed that every time a page admin deleted their account the list of their most recent viewed pages persisted and page admin deletion—which relied on procedural code in a custom section—did not delete these edges. DELF detected the edge type reuse and highlighted the missing edge type annotation.

Prevention. DELF enforces that all new data types are created alongside deletion annotations. In doing so it eliminates developer omissions as a correctness concern. The protection DELF entails, however, is only effective assuming available annotations can express sufficiently-complicated deletion logic. Developers would otherwise bypass DELF and continue to rely on custom procedural code to perform deletions. DELF permits this via the custom object type annotation (§4.1). To better assess the system’s ability to prevent omissions, we study how developers bypass DELF by using the custom object type annotation in new applications.

We retroactively inspect 408 changsets introduced in FACEBOOK infrastructure throughout October 2019 by 279 distinct developers. Each changset in our sample creates or modifies at least one object type annotation. Only 7 changsets designate the custom annotation. We observe no new legitimate instances where DELF annotations are lacking expressiveness. 6 changsets use custom to maintain backwards compatibility with legacy procedural deletion logic introduced before DELF was available, i.e., in one instance data to be deleted was stored in a TAO edge using a legacy serialization format and required special handling. We also notice one changset misusing custom to approximate reference counting, i.e., developers were oblivious to native support of refcount. We conclude that DELF annotations can express deletion logic necessary in modern OSNs and the system is
6.2 Inadvertent data retention

We continue by assessing how effective DELF is in identifying instances of dangling data engineers actively misclassified.

**Identified issues.** We start with examples where DELF pinpointed developer mistakes which would otherwise resulted in dangling data. We inspect 91 reports generated by DELF static heuristics during January 2020, of cases where developers annotated edges as shallow or refcount while DELF suggests deep. We submitted these reports to FACEBOOK’s privacy team for expert review to establish ground truth. The assessment established that developers incorrectly annotated 66 of these edges as shallow. Most of the remaining edges were ambiguous; we discuss those later.

We look closely at one representative example of inadvertent data retention in these reports which DELF identified and then developers remediated successfully. DELF surfaced that 23 distinct edge types used to represent different types of major life events for Facebook users, such as weddings, house moves, and changes to their citizenship, were mislabeled. The 23 edge types associated the user account object with objects of a separate type storing detailed information about the life event, e.g., the date the user got married. All were annotated shallow rather than deep, indicating erroneously that life event data should be retained post account deletion.

The report investigation confirmed the developer mistakes. The affected edge types were introduced at different times dating back to the introduction of the product feature in 2011. Legacy procedural deletion logic historically ensured correct deletion of associated life event data. Yet as part of DELF’s deployment two different developers—unaware of the historical deletion logic—annotated the edges in 2017 and 2018 as shallow instead. The DELF report highlighted the mistake prior to disabling the legacy procedural deletion logic, and hence no inadvertent data retention of life event data occurred.

**Prevention.** DELF helps developers annotate edges as deep via static and dynamic validation. We measure the impact of static validation in the the developer workflow, and we then assess how comprehensive deep heuristics are as a safety net.

1) Static validation. We conduct an experiment to measure how often static validation leads to developers changing their annotations during product development. DELF enforces statically that all object types must define at least one deep inbound edge type by virtue of treating by_any as the default annotation (§4.1). We inspect (a) a sample of changsets successfully creating 151 new object types in production during January 2020, and (b), logs of DELF static validation failures triggered during development starting from December 2019. We find that 62.2% of the new object types introduced failed static validation at some point during their development, e.g., developers did not define deep-annotated edge types to delete data stored therein. Developers subsequently corrected these mistakes and all 94 new object types were subsequently created while satisfying the chosen object type annotation.

2) Dynamic validation. We report on the precision and recall achieved by DELF heuristics on a sample of edge types already annotated by developers, treating developer annotations as ground truth\(^2\). We sample approximately 2.4 trillion individual edges deleted in production in January 2020. Of these, we pick at random 2000 shallow and 2000 deep edge types to ensure equal representation in our assessment. We ignore edge types with fewer than 20 samples since some heuristics require at least 20 items to classify an edge type.

We run DELF heuristics on all edges in our sample. For each heuristic, we count a true positive when the heuristic type matches the edge annotation and the heuristic triggers, a false positive when the heuristic type does not match the edge annotation but the heuristic triggers, a true negative when the heuristic type does not match and it does not trigger, and a false negative when the heuristic type matches but it does not trigger. We define the aggregate deep (resp. shallow) heuristic to trigger if any deep (resp. shallow) heuristic triggers, and the overall heuristic to trigger if exactly one of the deep or shallow aggregate heuristics trigger. In particular, if both deep and shallow heuristics trigger on an edge type, we consider the overall heuristic not to trigger.

Table 5 summarizes our results. DELF deep heuristics demonstrate precision of 89.7% at 60.7% recall, and DELF accurately discovers the majority of deep types in our sample.

Delf heuristics prioritize precision. In our experience

<table>
<thead>
<tr>
<th>Heuristic</th>
<th>Predicts</th>
<th>Prec. (%)</th>
<th>Recall (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>to_new_object</td>
<td>deep</td>
<td>86.9</td>
<td>4.7</td>
</tr>
<tr>
<td>to_leaf_object</td>
<td>deep</td>
<td>86.3</td>
<td>12.6</td>
</tr>
<tr>
<td>to_owned_object</td>
<td>deep</td>
<td>91.9</td>
<td>19.3</td>
</tr>
<tr>
<td>id1_ref_in_id2</td>
<td>deep</td>
<td>88.9</td>
<td>29.7</td>
</tr>
<tr>
<td>to_old_object</td>
<td>shallow</td>
<td>94.5</td>
<td>80.2</td>
</tr>
<tr>
<td>self_reference</td>
<td>shallow</td>
<td>100.0</td>
<td>12.0</td>
</tr>
<tr>
<td>many_to_one</td>
<td>shallow</td>
<td>96.4</td>
<td>54.5</td>
</tr>
<tr>
<td>same_obj_type</td>
<td>shallow</td>
<td>90.8</td>
<td>13.9</td>
</tr>
<tr>
<td>to_deleted</td>
<td>shallow</td>
<td>91.7</td>
<td>0.6</td>
</tr>
</tbody>
</table>

**Overall**

<table>
<thead>
<tr>
<th></th>
<th>deep</th>
<th>89.7</th>
<th>60.7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>shallow</td>
<td>93.0</td>
<td>89.5</td>
</tr>
<tr>
<td></td>
<td>either</td>
<td>95.0</td>
<td>74.8</td>
</tr>
</tbody>
</table>

Table 5: Precision and recall achieved by DELF heuristics on our sample of 4000 edge types. DELF discovers the correct annotation for the majority of edge types in our sample, which provides for an important discovery mechanism of developer mistakes. The overall precision is higher than both deep and shallow individually because we discard conflicting predictions; deep or shallow false positives with conflicting predictions are not considered valid predictions.

\(^2\)This assumption conservatively penalizes DELF heuristics when mismatches occur. Obtaining ground truth data at this scale is impractical.
developers are likely to ignore all predictions altogether when precision drops. One obstacle to further increasing deep heuristics recall without sacrificing precision are ambiguous edge types. Consider the photo object type example from Figure 1. The created_photo edge type pinpoints all photos a user creates and is annotated deep. Assume this photo object type is extended with an additional, optional edge type from a user to a photo to mark the user’s current profile photo. Such an edge type should be annotated deep; yet a shallow annotation does not result in inadvertent data retention. The original deep edge type triggers the deletion of all photos—including the current profile photo—when a user object is deleted. We observe that developers prefer to annotate ambiguous edge types as shallow to avoid inadvertent data deletion.

We conclude that DEFL static and dynamic verification methods, when used as a safety net to validate developer annotations, provide an important privacy protection against mistakes leading to inadvertent data retention. While DEFL cannot detect all instances of inadvertent retention, it detects most. Hence it makes mistakes significantly less common.

6.3 Inadvertent data deletion

We cover next DEFL’s impact avoiding data loss in situations where mistakes leading to inadvertent deletions occur.

**Identified issues.** We start with inadvertent deletion vulnerabilities where DEFL altogether avoided exploitation. We sample all reports generated during one week of November 2019 by DEFL privilege escalation checks while blocking suspicious writes of edge types annotated with deep (§4.3). Our sample contains 38 distinct edge types, which we forwarded to FACEBOOK’s security team for inspection. The team considered the 38 edge types in the list to be potentially exploitable, modulo the DEFL privilege escalation checks and the existence of public API methods to perform writes. To the best of our knowledge inadvertent deletion never occurred despite the underlying insufficient authorization checks.

We look next at incidents where inadvertent deletions occurred, detection required separate logging or user reports, and DEFL restoration logs were used for recovery. We inspect all 21 such incidents between February and December 2019. For effective mitigation inadvertent deletions must be detected before restoration logs expire and the restoration process must be operationally simple.

A notable incident of an exploited deletion vulnerability involved deletion of popular photos in Instagram. In October 2019 developers changed how photos were handled. The incident involved an edge type initially used to associate a discussion thread with the photo object posted therein. The edge type annotation was deep—deleting the thread necessitated deleting the associated photo. Developers later reused the same edge type when implementing photo sharing; an edge of the same type now associated a new share discussion thread with the original photo object. In doing so users who shared a photo in a new share thread obtained the ability to delete it by virtue of deleting the new share thread they created.

Instagram users triggered the vulnerability—knowingly or not—to delete approximately 17,000 photos, including multiple popular public photos with tens of millions of interactions such as likes and comments. Exploitation was possible because DEFL privilege escalation checks were not enforced in Instagram when the bug occurred. The issue was surfaced by user reports within 10 days. The recovery process involved one product engineer and the DEFL oncall; the former provided the list of objects to restore and the latter monitored progress. Restorations ran for approximately 10 days.

Many incidents in our sample did not require exploitation by a third party. Inadvertent deletions were triggered by internal maintenance processes or as a result of user action and affected only the user who performed the action.

A representative example occurred in April 2019. An Facebook developer triggered a cleanup data migration to delete objects representing invalid user devices, i.e., objects created erroneously. The developer ran a database scan over all existing device objects and scheduled deletions via DEFL. Yet a bug in the object selection logic of the scan triggered the deletion of a batch of devices every time one object in the batch was deemed invalid. The process inadvertently deleted approximately 100 million devices and adversely affected the ability of users to login as well as service integrity protections. Product-specific alerts surfaced the mistake to the team on the same day. The recovery process spanned 12 hours and involved 2 engineers. One provided a list of deleted objects for DEFL to restore; the other monitored the process.

**Prevention.** Assuming timely detection restoration logs reduce the issue of data loss to temporary data unavailability. To quantify DEFL’s ability to detect data loss independently, i.e., without any user reports or application-specific logging, we measure the effectiveness of shallow edge type annotation heuristics. Table 5 summarizes our results in the scenario from §6.2. DEFL shallow heuristics demonstrate precision of 93.0% at 89.5% recall. DEFL shallow heuristics independently pinpoint the majority of mistakes leading to inadvertent deletion when annotating edge types.

Data loss remains possible. Most notably, shallow heuristics cannot flag cases where application logic requests the deletion of the wrong object. During our investigation period significant data loss occurred in a single incident. The bug involved application logic requesting the deletion of the wrong video objects, was surfaced by user reports, and remained undetected for 2 years, i.e., significantly longer than the deletion policy allowed DEFL restoration logs to persist.

We conclude that DEFL restoration logs offer practical data loss prevention capabilities for most scenarios where inadvertent deletions occur. While some data loss risk remains, usable restoration logs combined with a sufficiently-long backup retention period provide a practical protection mechanism even when automated detection mechanisms fail.
Figure 6: Cumulative distribution of completion time.

Figure 7: Deletion size with respect to completion time.

6.4 Execution

We continue with an assessment of the system’s impact executing all deletions to completion. Our analysis is based on observed deletion end-to-end wall time in a production workload. Our sample includes approximately 12 billion deletions that finished execution at FACEBOOK on July 31, 2019, illustrated in Figure 6. Deletions execute in a shared pool of servers in FACEBOOK’s multi-tenant execution tier.

Identified issues. We observe transient and persistent errors delaying the execution of deletions in our sample. DELF drives deletions to completion despite such errors by retrying deletions persistently and surfacing detected issues for engineers to fix (§5.1). We discuss in detail one representative deletion facing transient and one facing persistent errors.

The longest-running deletion in our sample involved deleting a photo and performed 30,134 restoration log writes. The deletion suffered from at least three distinct types of transient infrastructure failures. The first type involves inadvertent drops of jobs from FACEBOOK’s asynchronous execution tier. DELF detected and rescheduled the dropped job in a number of occasions after a timeout. The second type involves exponential backoff and rate limits DELF enforces to avoid overloading underlying data stores (§5.2). The shards involved in this deletion were frequently under heavy load and DELF postponed the deletion multiple times to prevent further issues. The third type involves transient write errors frequent when operating on overloaded shards; those occurred at times despite rate limiting. Overall, the deletion ran for more than 90 days while making consistent progress.

A deletion affected by persistent errors involved deleting a user account and performed 1,770 restoration log writes. The deletion was stuck for 45 days due to two distinct issues, both requiring engineering intervention. The first issue involved procedural code in a custom section which contained a data serialization bug. The second issue was triggered by changes in the semantics of the point-delete operation in an underlying data store. DELF flagged both issues for engineers to fix and the deletion completed within 52 days.

Prevention. To quantify how many deletions benefit from DELF we look at the distribution of end-to-end wall time of all deletions in our sample. We observe three important points in Figure 6: P1, P2 and P3, respectively 31 seconds (35th percentile), 45 seconds (86th percentile) and 90 days (99.99999th percentile). P1 captures deletions of a single object. DELF executes those within the triggering web request without using the asynchronous execution tier. Shortly after 30 seconds, i.e., a configuration parameter of DELF’s deployment at FACEBOOK, the first run in the asynchronous tier starts. P2 indicates that a single run within the asynchronous execution tier is enough to complete the majority of deletions, i.e., most deletions involve few objects and complete without issues. P3 illustrates that 99.99999% of deletions complete within 90 days since they started.

In absence of infrastructure reliability and capacity issues, deletions would execute to completion without monitoring from DELF, and completion time would demonstrate a strong positive correlation with deletion size. To validate their prevalence we look into the long tail of deletions running for more than one day. Figure 7 plots end-to-end wall time required to complete deletions with respect to the number of restoration log writes each deletion performed. The number of writes to restoration logs approximates the size of each deletion.

We observe the correlation between wall time and deletion size exists yet it is weak for the tail of long-running deletions. Some deletions consistently leverage additional wall time to delete more data. In our sample a large deletion running for 30 days performed around 4.8 million restoration log writes while the largest deletion running for one day was limited to 0.5 million restoration log writes. However, the majority of deletions running for more than one day are moderately-sized.

We conclude that in the long tail reliability and capacity limitations are the root cause for long-running deletions. DELF therefore contributes to completing a significant proportion of all deletions. Any deletions that require at least two runs in the asynchronous execution tier—approximately 14% of all deletions—benefit. This includes deletions that require additional execution time because they entail deleting a lot of data and deletions that run into capacity and reliability issues. If developers were left to implement deletion unassisted up to 14% of all deletions triggered would potentially not complete.
We close with an assessment of system overhead. We profile writing to restoration logs since only one roundtrip is required per batch compared to a roundtrip per point delete. Instead, writes. Batching writes with a batch size of 100 (write-intensive ones, i.e., point deletes and restoration log writes) within each deletion: reads, point deletes, stack management, which requires 10 factor of 5 batching. Read batching reduces time spent on reads by a factor of approximately 5. The speedup highlights that write batching directly controls the frequency of writing to restoration logs since only one roundtrip is required per batch compared to a roundtrip per point delete. Instead, point deletes entail latency that is not amortized linearly while batching. Read batching reduces time spent on reads by a factor of 5× and has limited impact on the rest (C3 over C4).

We conclude that DELF introduces limited overhead during deletion, in line with systems offering similar guarantees [48].

Figure 8: Time spent in different operations during deletion.

6.5 Overhead

We close with an assessment of system overhead. We profile deletions and break down how time is spent in different operations within each deletion. We measure throughput while deleting a tree-structured graph with unit height stored in TAO which requires 10⁴ point deletes. The tree contains 100 edges from types annotated deep with the remaining types annotated shallow. We execute each deletion 10 times on distinct machines in FACEBOOK’s asynchronous execution tier. We measure throughput under 4 distinct batching configurations, varying the size of the read and write batching windows. We report the 10th, 50th, and 90th percentiles we observe.

Figure 8 shows our results. We notice four major operations within each deletion: reads, point deletes, stack management, and restoration log writes. The latter two entail synchronous writes for each batch of point deletes: in ZippyDB where DELF maintains a stack to implement depth-first graph traversal, and (b), in LogDevice [60] where DELF persists restoration logs temporarily, respectively. The remaining wall time, i.e., processing, involves periods of CPU-intensive operations, such as data serialization. We consider any wall time spent on operations beyond reads and point deletes as DELF overhead.

We observe that batching reads and writes reduces overall system overhead from 336% down to 29% (C1 over C4). Noteworthy, the most time consuming operations are the write-intensive ones, i.e., point deletes and restoration log writes. Batching writes with a batch size of 100 (C2 over C4) has substantial impact on both, reducing time spent in restoration log writes by a factor of approximately 100× and in point deletes by a factor of approximately 5×. The speedup highlights that write batching directly controls the frequency of writing to restoration logs since only one roundtrip is required per batch compared to a roundtrip per point delete. Instead, point deletes entail latency that is not amortized linearly while batching. Read batching reduces time spent on reads by a factor of 5× and has limited impact on the rest (C3 over C4).

We conclude that DELF introduces limited overhead during deletion, in line with systems offering similar guarantees [48].

7 Related Work

There is little prior work on the problem of deletion correctness. Garg et al. [61] formalize deletion to mandate deletion of dangling data yet their work does not suggest technical solutions developers may leverage to achieve the goal. A presentation from Doshi and Shah outlines Uber’s deletion service [62] focusing on reliability of user account deletions specifically. The system shares design traits with DELF yet does not offer any capabilities to safeguard correctness [63]. Ritzdorf et al. [64] study deletion correctness motivated by the problem of helping users delete related data from their local file system. They leverage data loss prevention techniques [65] to detect files storing similar content. Similar to DELF the authors suggest heuristics to identify what to delete, e.g., files accessed together or found to contain duplicate information should be deleted together. To the best of our knowledge, DELF is the first system to apply such techniques within complex web applications built on top of distributed data stores. Our work quantifies their effectiveness.

DELF restoration logs are an example of checkpointing, a technique for recovering from exploited security vulnerabilities that lead to unauthorized mutation of application state [48,66–68]. WARP [48]—similar to DELF—targets web applications, uses a browser extension for intercepting user actions, and is assessed in a single-node deployment. DELF demonstrates the applicability of checkpointing in modern, large-scale, distributed OSNs as a safety net for preventing inadvertent data deletion while attempting to delete user data.

Recent user research on deletion explores how users of modern web applications perceive the deletion process and highlight a pervasive lack of understanding. Murtillo et al. [69] interview users of Gmail and Facebook, report widespread misconceptions and mistrust, and suggest greater transparency in products. Ramokapaneet al. [4] document the coping strategies users employ when they cannot figure out how to delete data in web applications. Another line of user research studies the different motives people have to delete data [2,5].

A well-studied privacy concern regarding deletion is the effectiveness of individual point deletes [3,70,71]. Prior work explores the ability to delete data from physical media in a way that renders the data irrecoverable; suggesting special file systems [3] and scrubbing tools [72]. Sarkar et al. [73] introduce techniques to improve the ability of modern data stores to propagate point delete operations to physical media within a bounded time frame. The underlying assumption in this line of work is that users or developers know what to delete and when in their applications. We demonstrate that this assumption is not valid in complex web applications and we suggest techniques to safeguard deletion correctness. Minai et al. [74] highlight a conceptually-similar problem of ineffective deletion for public content in OSNs introduced by adversarial data mirroring services and suggest mitigations.
8 Future Work

Exploring the applicability of DELF outside OSNs requires further research. We anticipate deletion frameworks based on declarative annotations similar to DELF to be widely applicable across application domains and data stores. Some DELF validation techniques can be adapted to discover mistakes in existing applications without necessitating changes, facilitating correctness studies. We expect that in any domain with complex applications handling user data, deletion correctness validation will surface mistakes on an ongoing basis.

DELF’s ability to validate data type annotations can be extended further. A straightforward approach involves replacing edge type classification heuristics with machine-learned models trained on prior developer annotations. We expect such approaches to significantly improve the precision and recall of our current system, perhaps even surpassing developers.

Developers may create objects that do not get deleted even when all type-level annotations are correct. One way involves creating individual objects and omitting writing the corresponding edges necessary for deletion, e.g., creating a photo without a deep-annotated edge from its creator. DELF can enforce annotations at the data item level in addition to the data type level to preclude the creation of undeletable objects.

DELF annotations can be extended to be tied to tooling used for Privacy Impact Assessments [75]. When a particular deletion product behavior is mandated by an assessment, one could tie that decision to product implementation via DELF.

We expect further improvements in deletion transparency, accountability, and external correctness verification. Systems such as DELF can expose a transparency interface to indicate what data items get deleted from data stores and when; security researchers could use such interfaces to construct reproducible scenarios where dangling data remains; and bug bounty programs could reward their discovery.

9 Conclusion

We presented DELF, a system to safeguard deletion correctness in large-scale OSNs in presence of developer mistakes and complex data models. DELF’s main novelty lies in forcing developers to annotate all their data types for deletion before they are used and then detecting mistakes resulting into inadvertent retention or inadvertent deletion. DELF entails overhead during deletion yet the system enables developers to delete data safely despite mistakes which invariably occur. We showed how DELF prevented or minimized disruption at FACEBOOK due to multiple bugs in deletion.

10 Acknowledgements

Many engineers contributed to DELF during its development. We would like to acknowledge Ben Mathews and Scott Renfro for bootstrapping DELF; and Adarsh Koyya, Akin Il erle, Amimits Chandele, Andrei Bajenov, Anurag Sharma, Boris Grubic, Gerard Goossen, Cristina Grigoruta, Gustavo Pacianotto Gouveia, Gustavo Pereira De Castro, Huseyni Ol gac, Jordan Webster, Mahdy Nasr, Maria Mateescu, Masha Kereb, Merna Rezk, Nikita Efanson, Ohad Almagor, Oleksandr Manzyuk, Prakash Verma, Shradha Budhiraja, Shubhanshu Agrawal, Sneha Padgalwar, Tudor Tiplea, and Vasil Vasilev for contributing significant components. Our paper builds upon the work of FACEBOOK developers who annotated their data models and investigated discrepancies DELF reported.

We would like to thank our shepherd, Sarah Meiklejohn, the anonymous reviewers, and members of FACEBOOK’s legal team including Bathilde Waquet, Sumit Shah and Scott Mellon, for their invaluable feedback on prior paper drafts.

References
