MUTT: A Watchdog for OSN Applications

Amre Shakimov and Landon P. Cox
Duke University

Abstract

Services such as Facebook and Twitter host and disseminate data on behalf of billions of users. Because these services often manage personal data, they allow users to specify access policies controlling how their data is shared with others within the service. However, services also act as programming platforms, exporting users' data to third-party applications via remote APIs. Nearly all of these third-party applications execute on server infrastructure that is not controlled by the service. As a result, a service has no way to guarantee that data shared with a third-party application will be managed according to users' policies. Delegation protocols such as OAuth allow a user and service to confer or deny an application's right to access a data item, but once the item has been released there is no oversight of what the application does with it. In this paper, we present the design and implementation of a Multi-User Taint Tracker (MUTT), which ensures that third-party applications adhere to access policies defined by service users. We motivate MUTT's design by analyzing 170 Facebook apps and several services' Terms of Service, and demonstrate the feasibility of our design through experiments with a prototype implementation.

1 Introduction

Online social networks (OSNs) such as Facebook and Twitter archive and disseminate messages and media for billions of users. These large-scale, multi-user services also augment their core functionality through third-party extensions called *applications* (i.e., "apps"). Apps typically consist of server-side code that reads and updates users' state through calls to a remote API. Apps are popular because they simplify identity management for users and allow third-party developers to plug-in new features to an existing service.

An application framework would ideally provide these benefits without creating new risks for users' private data, but today's frameworks offer no way to monitor what an app does with the data it accesses. This lack of oversight has led to several privacy violations, each of which falls into one of two categories: (1) unauthorized *external sharing* with entities such as third-party advertising networks [6, 10, 11], and (2) unauthorized *internal sharing* with other users of the service [21, 24].

To prevent unauthorized external sharing, xBook [29] proposed refactoring apps into encapsulated components and restricting each component's communication to a set of whitelisted administrative domains specified by the developer. However, controlling information flows at the granularity of an administrative domain is too coarse to prevent unauthorized internal sharing. For example, the "Top Friends" app exposed the personal information of Facebook users who installed the app to anyone else who installed the app [24].

Unauthorized internal sharing is a result of apps' using permissions obtained from many users to circumvent a service's primary access-control mechanisms. Services provide users with settings for controlling how their data may be shared with others, what data an app may access, and which users may view messages posted by an app on their behalf. However, these measures cannot prevent an app from forwarding data from a source it is authorized to read to a sink it is authorized to write.

This paper presents *MUTT*, a set of extensions for a "Platform as a Service" (PaaS) that tracks and controls the flow of users' information through hosted apps. MUTT integrates dynamic information-flow tracking (i.e., taint tracking) into a PaaS' runtime environment and storage layer to monitor and control how apps handle user data. The key challenges for MUTT are (1) labeling an app's data, processes, and communication channels, and (2) enforcing access-control policies expressed in terms of these labels without introducing onerous new programming and administrative burdens for developers, services, and users.

TRIOS'13, November 03 2013, Farmington, PA, USA

http://dx.doi.org/10.1145/2524211.2524215

MUTT addresses the first challenge through a generic labeling scheme that allows service-specific drivers to interpose on and interpret communication between a hosted app and an administrative domain. To understand which policies to enforce, we manually reviewed 170 Facebook apps, classified their behavior, and recorded any potential for unauthorized sharing. We also studied the disclosure guidelines for major services such as Facebook, Google, and Twitter. These studies revealed that (1) of the apps we reviewed, only dating apps require information flows to unauthorized users as part of their core functionality, and (2) to ensure that apps comply with service disclosure guidelines, MUTT must actively monitor the freshness of any access control lists associated with cached data.

To summarize, this paper makes the following contributions:

- It presents the design of MUTT, a set of PaaS extensions that prevents unauthorized internal sharing by third-party apps for multi-user services.
- It characterizes the inter-user data sharing of 170 popular Facebook apps.
- It characterizes the Terms of Service for Facebook, Flickr, Google, and Twitter and analyzes how these policies can be violated by third-party apps.
- It presents an evaluation of our MUTT prototype and shows that its performance is reasonable.

The rest of this paper is organized as follows: Section 2 provides background information on the technologies underlying MUTT, Section 3 describes MUTT's trust and threat model, Section 4 presents the results of our application study, Section 5 presents MUTT's design, Section 6 describes our prototype implementation, Section 7 presents our evaluation results, Section 8 describes related work, and Section 9 provides our conclusions.

2 Background

In this section we provide background information on technologies referenced throughout the rest of the paper.

2.1 OSN apps

Many online social network (OSN) services allow third-party developers to build service extensions that utilize a user's data, such as her social graph and personal information. These *applications*, or *apps*, add new functionality to existing services or use OSN data to enrich their own functionality.

We are primarily interested in apps whose functionality depends on a user's social graph. For example, many OSN games are popular because players' friends also play, and the OSN provides an easy way for game developers to manage player identities and communication. Other apps use friends' pictures, videos and status updates to create collages, ask questions, or collect opinions.

Since Facebook is the world's most popular OSN and has the most popular application framework, we will describe its framework in greater detail. Frameworks for other services (e.g., Twitter) are similar. According to Facebook's official statistics, users install about 20 million applications each day from developers from 190 countries [7]. The Facebook Platform provides developers with a rich API for accessing users' data. All third-party apps not developed by Facebook run outside of Facebook's datacenters, whether inside their own [9] or on public cloud infrastructure [1].

To access a user's data, each app must obtain the user's consent during installation. These requests are authorized by a service using the OAuth protocol described in Section 2.2. Apps can also request access to a user's data in "offline mode." This allows an app to retrieve data even when the user is not actively interacting with the app.

2.2 OAuth

OAuth is an open standard for federated authorization and delegation used by nearly all modern web services. OAuth allows a user to give a third party (e.g., an app) permission to access data hosted by a service without disclosing credentials such as a username and password to the third-party. OAuth allows a service to create a capability called an OAuth *token* for an app that can be presented along with requests to access a user's data. Typically, a token contains a list of resources that can be accessed, an expiration date, and other relevant information.

The procedure for obtaining an OAuth token is as follows. When an app initially requests access to a resource hosted by a service, it redirects the user interacting with the app to a service web server, passing along a list of data types it would like to access (e.g., photos, messages, or videos). The user directly authenticates herself to the service and approves the requested permissions. Then the service redirects the client back to the app, while including a secret *string* (i.e., an authorization code) that the app can use to retrieve its token directly from the service.

After presenting the string to the service and receiving a token, the app can store the token on its own servers and use it to generate a unique *user ID* to be stored on

the user's client (e.g., in a web browser cache). Upon receiving a client request, an app uses the user ID included in the cookie to retrieve the corresponding token from its local storage. The token can then be used by the app to make API calls to the OSN for accessing a user's data.

2.3 Taint tracking

Dynamic information-flow analysis (i.e., taint tracking) is a well-known technique for monitoring runtime data dependencies [18, 25, 26, 30]. Imagine an app that fetches a user's pictures, processes them, and returns the result (e.g., a collage) to the user's client. Taint tracking can be used to ensure that the app does not disclose the user's pictures and collage to anyone except the requester.

A platform's runtime environment performs taint tracking by assigning a *label* to each *storage object* capable of holding secret information. A label contains one or more *tags*, each denoting a source of secret data. *Operations* change objects' state by transferring information from a set of source objects to a set of destination objects. Trackers *propagate* tags by intercepting operations that could transfer secret information and updating objects' labels accordingly. Finally, the runtime checks whether a tainted object may be copied outside of the app according to *taint export policies*.

2.4 Platform as a Service (PaaS)

Platform as a Service (PaaS) is a cloud-computing model that provides a high-level programming environment for hosted apps. Incoming client requests are dispatched by a PaaS app *server* to an appropriate app *instance*. An app instance is a process that executes the application logic and handles incoming requests. PaaS environments often automatically scale the number of instances for an app (e.g., create more of fewer instances) depending on its resource requirements. The most popular PaaS is the Google App Engine (GAE). For our MUTT prototype, we modified AppScale, which is an open source implementation of GAE [15].

3 Trust and threat model

As with other taint-tracking systems [16, 18, 23] MUTT's trust model is based on the language-level protections provided by the runtime in which third-party apps execute. Apps must be written in a high-level language such as Python or Java, which allows the runtime to interpose on every program-variable access. As a result, apps cannot directly modify the taint tags that indicate what sensitive data a variable's value depends on.

MUTT's confidentiality guarantees depend on (1) eliminating unauthorized information flows, (2) utilizing knowledge of existing service APIs to correctly tag data before it can be accessed by app, and (3) correctly propagating taint tags throughout the PaaS language runtime and built-in libraries. The MUTT runtime also mediates apps' access to the network and persistent storage; it uses variables' taint tags to prevent unauthorized network writes and ensures that persistent storage properly reflects the sensitivity of the data it holds. MUTT provides no guarantees once sensitive information exits the PaaS (e.g., to a user's browser or to a service-controlled server).

We designed MUTT as an extension to AppScale [15] to maximize compatibility with existing apps, and MUTT inherits AppScale's trusted computing base. If the language runtime, AppScale's application server, or a standard library becomes compromised, an attacker could directly obtain sensitive data.

Implicit flows, in which a program transfers information to program state via its control flow, is a long-standing challenge for all taint-tracking systems [17]. Ignoring these flows creates opportunities for attackers, but existing techniques for propagating labels as a result of timing events or changes to a program's control flow frequently lead to crippling false positives. To help mitigate this problem, MUTT supports a special flag that prevents a tainted object from being used to evaluate a conditional statement. Disallowing tainted conditional branching would prevent implicit flows but could break some applications. We discuss this tradeoff further in Section 5.3.

Due to these limitations, MUTT is primarily meant to prevent overzealous developers from oversharing sensitive data or handling it inappropriately (e.g., forgetting to delete data after the user uninstalls the app). MUTT is not designed to prevent attacks by a determined, malicious adversary.

Users can verify that an app instance is sandboxed by a MUTT-enabled PaaS in a number of ways. For example, an infrastructure provider could generate a chain of signed attestations rooted in trusted hardware such as a Trusted Platform Module (TPM) [27]. Under this approach, a user or service must verify that all hardware and measured software in the chain is trustworthy. Alternatively, users and services could trust a handful of organizations such as a university or Google "by fiat" to faithfully execute third-party apps in a MUTT-enabled PaaS. This approach would require users and services to verify the identify of infrastructure providers through a protocol such as SSL or TLS.

Ideally, services such as Facebook, Google, and Twitter would take a leading role in providing users with information about the trustworthiness of third-party apps'

runtime environments. For example, a service could ease users' burden by including meta-data endorsing an app's hosting platform during the OAuth authorization process.

4 Application and framework survey

MUTT would ideally support all legitimate apps and user-privacy policies without requiring changes to existing app code and OSN APIs. To better understand the feasibility of this goal, we performed a survey of existing apps and OSN app frameworks. Our survey was intended to characterize what OSN apps do with users' data and how services expect apps to handle users' data.

4.1 Apps

To understand third-party apps we examined 200 popular Facebook apps listed in an open app directory [3]. Although our study was limited to Facebook apps, we believe that these are representative of apps written for other OSNs. The Facebook Platform is the most popular application framework and offers the most flexible and complete API for developers. To characterize Facebook apps, we chose a broad sample based both on popularity (i.e., *daily active users (DAU)* and *monthly active users (MAU)*) and functional diversity.

We excluded three kinds of apps from our study. First, we did not consider apps developed by Facebook (e.g., FBML and Discussions) since they remain under the service's control. Second, we did not study Facebook apps that do not run in the cloud such as mobile apps. Finally, we did not study apps that only connect Facebook with another established service like Bing, Windows Live Messenger, Yahoo!, or YouTube. Our primary reasons for excluding these apps are that these apps' functionality does not require access to a user's OSN data, and that they run on closed infrastructure rather than on an open PaaS environment.

After filtering out these apps, we were left with 170. Most could be placed into one of three categories: *gaming*, *dating*, and *personal* apps. Table 1 shows a representative sample, with five apps per category. The table is sorted by DAU and contains additional information about each app, such as their popularity and information about the developers. As the table shows, the most popular app categories are gaming and dating. Personal apps offer a wide range of functionality, but, in general, create aggregate views of a user's social graph, photos, or friends. We could not identify a dominant app in this category, but, in aggregate, personal apps are quite popular.

Data-sharing analysis

We first sought to characterize two aspects of OSN app behavior:

- Do apps require access to public data such as name, user ID, profile picture, or gender, or do the apps require access to private data such as pictures, location, birthday, or religious/political views?
- Is it necessary for apps to share data with other app users? Are those users inside or outside of a user's social graph?

Some apps request access to more data than is strictly necessary for their stated functionality. However, Facebook actively tries to prevent such behavior [20], and in our study we assumed that all app requests are legitimate. It is also important to note that we did not have access to any app source code. Our conclusions are based only on observations of apps' behavior, an analysis of the Facebook Platform API, and experience building simple apps.

Table 2 summarizes the result of our analysis. The table shows that apps in all categories require access to a user's public data (e.g., name, user ID, profile picture, gender, hometown, networks, and friend list). Dating and personal apps also require access to some of a user's private data (e.g., pictures, bio, posts, or favorite places).

Furthermore, each app category has different information-flow requirements. Gaming apps often propagate updates about in-game events to a user's friends, but do not propagate users' public or private OSN data. On the other hand, though personal apps often access a user's private data, they do not propagate this information directly through the social graph. Rather, the output of a personal app is often shared indirectly and under the control of the user. For example, the FriendMatrix app generates a collage using pictures of a user's friends and stores the result in a photo album. In this case the owner controls access to the collage by adjusting the privacy settings for the album containing the picture. Perhaps most interestingly, in order to match users, dating apps must directly share a user's data with people outside of her social graph.

Social graph usage

While studying apps and the Platform API we found that a user's social graph is treated differently across categories. For example, personal apps rarely cache a user's social graph since these apps are often stateless. Instead, they gather information at the time they run to generate a result. When run again a personal app will regenerate its result by retrieving the latest data from the social graph.

Name	Category	DAU	MAU	Developer	Location
FarmVille	Game	10M	30M	Zynga	USA
Texas HoldEm Poker	Game	7M	35M	Zynga	USA
Daily Horoscope	Personal	6M	14M	Soft Reklam	Turkey
Badoo	Dating/Matching	2M	35M	Badoo	Cyprus
Pet Society	Game	1.5M	9M	Electronic Arts	USA
Mafia Wars	Game	1.4M	7.6M	Zynga	USA
Phrases	Personal	1.3M	16M	Takeoff Monkey	N/A
Zoosk	Dating/Matching	1.3M	22.8M	Zoosk	USA
Are You Interested?	Dating/Matching	0.8M	12M	SNAP Interactive	USA
Monster Galaxy	Game	0.6M	17M	Gaia Online	USA
Cupid	Dating/Matching	0.2M	8M	Cupid	USA
Friend.ly	Dating/Matching	0.2M	7M	Friend.ly	USA
FriendMatrix	Personal	0.2M	4M	Friend Matrix Studios	N/A
My Top Friends	Personal	5K	140K	MiniMax	China
FriendCollage	Personal	N/A	303K	N/A	N/A
Museum of Me	Personal	N/A	200K	Intel	USA

Table 1: Facebook applications

Category	Required access to		Required data sharing	
	Public data	Private data	within social graph	outside of social graph
Games	✓		✓	
Dating/Matching	✓	✓		✓
Personal	✓	✓		

Table 2: Categories of Facebook applications

On the other hand, many apps in our study maintain their own social graphs as a part of their functionality. In FarmVille [8] friends can be virtual neighbors, and send requests and gifts to each other, including leaving notifications on friends' profiles. Badoo [2] also maintains its own social graph using the initial friends list. Users of this app can answer trivial questions about their friends to receive bonus points and post these answers on their friends' profile pages. Moreover, users can rate their friends' pictures as a part of the app's functionality. We found that in both cases, even if user A "unfriends" user B, user B can still post notifications on user A's profile page through the app. User A can only block the notifications by blocking the app entirely.

This varying behavior raises the question of how we should treat third-party apps. If an app functions as a service extension then it must keep an up-to-date view of the user's data, particularly her social graph. Otherwise, we could treat each third-party app as a separate OSN, and allow it to maintain its own social graph. However, we believe that this approach would be counter-intuitive for a user since it requires managing multiple social graphs within a single framework.

App permissions

In September of 2011, Facebook introduced a beta version of an Open Graph – its Platform API [4]. The most relevant change was that app permissions were split into two categories [5]: *required* and *extended*. Required permissions include basic information about a user and her friends. Extended permissions allow access to a user's news feed, location check-ins, mailbox, and other data. To install an app, a user has to give the app access to the items from the required permissions set. However, each extended permission is optional. In theory, an app should be able to work without extended permissions. While this division of requirements provides more fine-grained privacy control, it does not solve the problem of inappropriate internal sharing.

4.2 Service policies

We also wanted to understand how services expect third-party apps to handle users' data. We reviewed the Terms of Services (TOS) for Facebook, Flickr, Google, and Twitter and identified three common requirements for third-party apps.

First, service providers require that any data fetched by a third-party app be shared only with "authorized users." For example, if a picture on Flickr has a tag "public" it can be shared with everyone. In contrast, if the owner of the picture specified its access control list (ACL), the app must respect this restriction. As another example, Facebook states that no data (even if aggregated, anonymous, or derivative) can be shared with third-parties, even with a user's consent.

Second, services have special requirements for caching user data. Platforms require that cached data be updated within a "reasonable" amount of time. In addition, Facebook requires apps to delete all of a user's information if the user make such a request or if Facebook disables the app.

Third, apps are required to keep items' ACLs fresh. For example, when an app fetches a user's picture it might have ACL a. However, at some point the user might change the picture's ACL to a more restrictive ACL a', such that $a' \subsetneq a$. If the app continued to use a it would violate Facebook's TOS. As a result, third-party apps are obligated to abide by a service's caching policies, and must refresh each data item's ACL within a reasonable amount of time.

We summarize OSNs' Terms of Service as follows:

Defitnition 1 *Terms of service (TOS) policies:*

#1: No unauthorized disclosure of data.

#2: Reasonable caching policies.

#3: Fresh cached and remote ACLs.

4.3 Discussion

MUTT's goals are (1) to ensure that apps abide by a service's TOS, and (2) to support the kind of functionality observed in our app study.

First, consider the TOS requirement that data be shared only with the authorized users. We examined the APIs of Google+, Twitter, Facebook and popular Russian OSN Vkontakte [12] to understand whether existing APIs can be used to enforce users' access control policies. For Google+, Twitter, and Vkontakte, each (1) assigns a unique global ID to every user-supplied data item, such as a wall post or personal note, and (2) allows apps to retrieve the ACL of a data item through its ID. As a result, for these services, MUTT can use existing APIs to prevent apps from violating users' access control policies.

Unfortunately, Facebook's API does not give full access to many data items' ACLs. For example, imagine a user giving an app permission to access her pictures. If the ACL for an album includes a subset of her friends, then MUTT must ensure that the app does not share pictures from the album with friends that are missing from its ACL. However, the Facebook Platform API does not give access to "complex" ACLs such as the album's. A complex ACLs is often created manually, as opposed to standard ACLs such as *"friends"*, *"friends of friends"*,

or a user's networks. The reason for this limitation of the Facebook API is likely that the content of a complex ACL may be privacy-sensitive itself, but without a full enumeration MUTT cannot ensure that apps will abide by Facebook's TOS.

Second, depending on the app, the definition of "authorized disclosure" may be ambiguous. For example, consider any dating or matching app. A dating app typically has access to a user's personal data such as her name, location, and pictures. Using this data, the app tries to match her with other users outside of her social graph. As a result, the user's information may flow anywhere within the entire social network, including malicious, data-gathering profiles. An attacker could gather information it is unauthorized to access by forging profile features (e.g., gender, age, location and picture) that might ensure a high rate of matching. For such apps, it is not clear how can to interpret "authorized disclosure", even assuming a non-malicious app.

Due to these issues, MUTT cannot simultaneously support existing apps and ensure adherence to Facebook's TOS. As a result, rather than enforce TOS policy #1 from Section 4.2, MUTT targets a more explicit requirement: *"sensitive data can only be shared only within the owner's social graph."* Note that this requirement prevents MUTT from hosting dating apps since these apps share sensitive data with users outside of an owner's social graph.

In addition, *TOS policies #2 and #3*, which require cached data to be reasonably consistent with the service's copy, are hopelessly vague. Ensuring strongest consistency using Facebook's API would degrade app performance, because it would require an expensive API call for nearly every operation. Thus, MUTT manages the consistency of cached data via a tunable timeout parameter. This parameter can be set by either the platform or service and can shared with users and services.

Below we present an updated set of policies that reflect these choices:

Defitnition 2 *MUTT policies:*

#1: Restrict data flows to a user's social graph.

#2: Timeout-based caching of user data.

#3: Timeout-based caching of data ACLs.

5 MUTT design

Before presenting MUTT's design, we first present a coarse-grained *Strawman PaaS* and use its shortcomings to motivate our use of fine-grained taint-tracking.

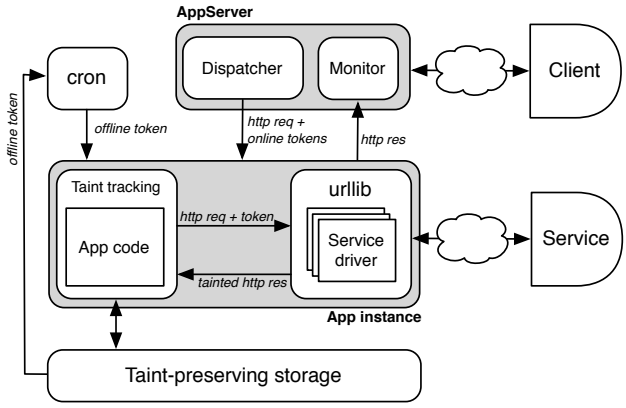

Figure 1: Architecture overview

An app running on a PaaS must obtain an *OAuth token* from the OSN using the protocol described in Section 2.2 before it can access a user's OSN data. After retrieving a token, the app stores it locally and encodes the token as a *user ID* within a cookie. The user's client stores this cookie for use in subsequent requests. An app can use the user ID included in a client request to retrieve the corresponding token. The token can then be used by the app to access a user's OSN data.

The Strawman can interpose on incoming client requests and interactions with a local database. When handling an incoming request, the Strawman assigns the user ID embedded in a request to the app instance that handles it, just as UNIX and other operating systems assign UIDs to processes forked on behalf of a particular user.

The Strawman then annotates any data added to the app's database with the user ID of the instance performing the insertion and enforces user-ID-based access control as other instances access the database. This approach is analogous to the way that operating systems regulate access to a file system, although it is stricter since the Strawman ensures that a data item can be accessed by *only* processes with a matching user ID. To further ensure that the instance handling a user's request will not access another user's tokens or data, the Strawman can disallow IPC between instances.

Though a good start, the PaaS described above is impractical. First, we must prevent an app from leaking sensitive data to other servers. For example, an app instance could use *urllib* to send data to arbitrary external sources via HTTP. On the other hand, an app may need to retrieve non-OSN data to complete its request. Thus, the PaaS must distinguish communication with outside servers that could leak user data from communication that is safe.

Second, some "mashup" apps may want to process data from multiple OSNs, but this is difficult when user IDs are bound to a single OAuth token (and, hence, a single OSN). Finally, the Strawman PaaS only allows apps to export data to their owner, which is incompatible with many of the third-party apps discussed earlier.

5.1 Architecture overview

To address the shortcomings of the coarse-grained Strawman, we designed MUTT, a Multi-User Taint Tracker for third-party OSN apps. We extended the Strawman by adding dynamic taint analysis [25, 26, 30] to the language runtime of the PaaS. For a PaaS, taint sources are the OSNs from which an app fetches a user's data. Thus, the runtime must interpose on and interpret any OSN API calls made by an app so that the OSN-of-origin and type information can be appropriately encoded in any retrieved object's label before the object can be handled by app code.

Figure 1 shows a high-level overview of MUTT's design. App code runs as an isolated *app instance* in a taint-tracking runtime. A *client* interacts with the app instance via the *app server*. The app server dispatches requests to the appropriate app instance. It also gives the instance any "online" OAuth tokens needed to handle the request. The server takes care to appropriately label these tokens before forwarding them to the instance.

MUTT also supports offline requests such as *cron* jobs using "offline" OAuth tokens. Similar to online requests, offline requests are dispatched to an app instance, which can export processed data directly to a *client* or to an OSN via the appropriate *service driver*. For example, an offline app instance may want to modify a user's profile, upload a picture, or send a notification.

Using the tokens given to it by the app server, an app instance can also fetch a user's OSN data through a service driver. The driver is responsible for labeling any retrieved data items. In addition, the app can make arbitrary HTTP requests through *urllib* as long as the outgoing requests do not contain tainted objects. Data objects can be saved to *taint-preserving storage* that regulates access to data and serializes/deserializes taint labels.

Finally, a *monitor* within the app server regulates communication between an app instance and client. The monitor checks the taint labels of the outgoing objects and ensures that the client is authorized to access these objects. In the remainder of the section we will discuss MUTT's most important components in greater detail.

5.2 Service drivers

MUTT ensures that data is properly labeled by introducing a suite of trusted service drivers. Each driver is a small module that proxies incoming and outgoing data between an OSN and app instance. All data retrieved from the OSN on behalf of the app is labeled before being handed to the app. When the app instance needs to update remote OSN data, it must do so through the driver. In these cases the driver verifies that the user receiving the outgoing object is within the social graph of the object owner (or all owners in case of an aggregated data item).

Implementing separate drivers for a handful of popular OSNs is reasonable given the dominance of a few services. Drivers are also relatively easy to implement since OSNs expose well-defined, RESTful APIs. Note that a service driver needs to handle the initial token negotiation as well as subsequent API calls. Drivers label any tokens retrieved from an OSN, and an app is responsible for passing the appropriate token back to the driver when it tries to interact with the OSN.

5.3 Language runtime

In this subsection we discuss MUTT's taint labels and how they are propagated by the Python runtime.

Taint structure

Performance is a critical concern for dynamic taint analysis because the runtime must interpose on all operations in order to update objects' labels. The key challenge for a PaaS taint tracker is to provide an efficient way to represent and update labels that is also flexible enough to support the policies and functionality needed by users and apps.

```
typedef struct {                    1
    char* token_set;                2
    unsigned long flags;            3
} _taint_struct;                    4
```

Listing 1: Taint structure

MUTT uses simple taint labels to capture information about an object's sources. Listing 1 shows the structure of a taint label. The taint label contains a *token-set*, which stores any of the OAuth tokens used to retrieve or construct the tainted object. It also has a *flags* field that carries additional information about a data object. Currently, we use this field to indicate type information such as OAuth tokens or passwords. This can be used to prevent an app from using such an object to evaluate a conditional branch. It can also be used to encode information about how data should be handled, including retention or inter-user sharing policies.

Taint labeling and propagation

Any object can be tagged by an app through the extended class interface. In MUTT, all objects retrieved through

a service driver get an appropriate taint value with their token-sets containing the OAuth token used for retrieval. After creation, a taint structure cannot be changed by Python-level code. Attempts to create an object from an already-tainted object causes an exception.

MUTT propagates taint structures in two cases: (1) during object assignment, and (2) when two or more objects are used to produce another one. The former requires copying the taint structure, whereas for the latter requires merging labels. To merge labels, MUTT sets the destination operands' labels to the union of the source-operand token-sets and flags. Computing the union of two or more token-sets can be slow when sets are large. However, we have found that these slow operations usually occur during offline jobs when good performance is less critical.

Taint sinks

Tainted objects can be exported outside of the system through two interfaces: it can be sent to a user's client as a response or used to update OSN data via a service driver. The reference monitor ensures that all exported data can only be accessed by an authorized user, i.e., the data owner or her friends. More formally, object O with token-set containing tokens $t_1, t_2, ..., t_n$ can leave the system through a sink authorized by token t_k only if $\forall t_i, i = 1, 2, .., n$: an owner of t_k is a friend of the owner of t_i.

Implicit flows

One use of our labels' flag field is to identify *unbranchable* objects when evaluating conditional branches. For some sensitive data such as OAuth tokens this approach would reduce the potential for leaks through covert channels. Thus, if an unbranchable object is used in a conditional, the language runtime throws an exception error (Listing 2, lines 1-4). However, it is possible to clear this flag for certain aggregate functions, such as *strlen* or *count* (Listing 2, lines 6-9). We leave this decision for the specific implementations since for some data, even aggregated information may be sensitive.

```
# token_obj is tainted as non-conditional                    1
if token_obj is guess_value:                                 2
    # raises an exception                                    3
    return guess_value                                       4
                                                             5
""" however, this is fine if function len clears             6
 the non-conditional flag off """                            7
if len(token_obj) == 0:                                      8
    print "wrong token size"                                 9
```

Listing 2: Implicit flow example

5.4 Platform

Persistent storage

Besides language support, the app should be able to store a tainted object and retrieve it later without losing its taint value. AppScale uses the Google Datastore API as a database-agnostic interface to store objects persistently. We modified this interface to support serialization and deserialization of taint labels whenever tainted objects pass to or from storage. These modifications were performed in the Google Datastore's code which allowed us to support all databases shipped with the AppScale.

Memcache is a distributed in-memory key-value data store useful for caching. We also patched memcache's implementation of its binary protocol to enable label serialization and deserialization when storing and retrieving tainted objects.

App server

AppScale's built-in app server is responsible for dispatching incoming requests and sending back response objects to clients. Under MUTT, the app server also serves the role of reference monitor, checking if a response can leave the system, given the object's token-set and sink's credentials.

5.5 Policy enforcement

Having described MUTT's design we now explain how it enforces the three policies outlined in Definition 2. MUTT enforces *policy #1* by ensuring that all OSN data is properly tagged and that the propagation logic is correctly implemented. MUTT enforces *policy #2* by including information about when an object was fetched in its taint label. An offline job periodically checks the database to see if an object has expired and deletes it if needed. Our current retention time is set to one week.

Finally, to enforce *policy #3*, MUTT's reference monitor verifies that a data owner (or owners in the case of a compound object) and object recipient are OSN friends before exporting a tainted object. The monitor verifies friend relationships synchronously through the appropriate service drivers.

5.6 Discussion and limitations

In the remainder of this section we discuss some of the challenges we encountered into while designing (and implementing) our prototype. We found that overcoming many of these issues would require changes to existing platform APIs. However, since we wanted to make our initial MUTT design backwards compatible *we did*

not implement any features that required OSN cooperation. In particular, we believe that future platform APIs should adopt the following features:

- Retrieval of complex object ACLs.
- Frameworks for reasoning about the trustworthiness of a PaaS hosting a third-party app.
- Support for advanced caching and data retention policies.

We argue that the ability to input custom app-specific security and privacy policies by users and service providers is one of the most important features to add to MUTT. We do not see any fundamental problems with extending MUTT to support custom information-flow or data-caching policies. In fact, this feature mostly requires a number of changes on the service provider's side.

First, reviewing app-specific privacy policies can be performed when a user reviews an app's data access requirements during installation. Creating user-friendly interfaces for inputting and managing custom privacy policies is difficult, but perhaps users could manage these policies through interfaces similar to those already supported by OSNs. Moreover, to support more fine-grained information-flow policies all service providers should expose data objects' ACLs. Currently, Facebook's Platform does not provide access to custom ACLs.

Another important feature is a framework for reasoning about an app's hosting environment. Users currently have no way to verify if a given cloud service properly handles their data. It is in service providers' best interests to provide such information to its users. There are a number of bases for trust that users might be interested in: app developers' reputation, an endorsement from a trustworthy authority, and *active attestations* [27]. We believe, that service providers should supply this metadata to users so that they can make informed decisions when installing an app.

Finally, as mentioned earlier, service providers should help MUTT choose good caching policies by either explicitly setting objects' expiration times or by actively invalidating cached objects.

6 Implementation

We have built a MUTT prototype based on the design described in Section 5. We use the AppScale [15] open-source implementation of Google AppEngine as a foundation for our system. In this section we discuss our modifications to the Python runtime and its libraries to support taint tracking and the changes we made to AppScale.

We have modified fewer than 2,000 lines of C-code (including new modules and policies) in the Python source, and about 5,000 lines of Python-code in AppScale.

6.1 Python

To implement taint tracking, we modified AppScale's Python runtime and support library.

Python runtime

To make our prototype's code more useful and extensible, we modified the existing Python runtime so that most changes were at the highest abstraction level. However, since Python is written in C, we added the taint structure to PyObject_HEAD – a preamble of every object in Python. This allowed us to implement generic modules for tainting and merging Python objects. Moreover, this helped minimize changes to the Python code. Instead of making changes to every primitive type, we could place most of the taint propagation and merging logic in the *abstract* and *object* class implementations. However, the downside of this approach is that even non-taintable objects such as containers or function objects contain a taint structure. We discuss the overhead of this design choice in Section 7.1.

Adding taint support to Python's primitive types allowed us to automatically support any complex objects such as classes or containers with tainted members. For example, an *array* or *dict* may include tainted and untainted items, and a class object could have tainted fields. However, when such a complex object is serialized (e.g., in order to be exported), the resulting taint label will be constructed from their taint labels.

An additional challenge arose from the fact that all objects in the original Python implementation are immutable, which allowed its developers to optimize the creation of primitive types. For example, all small integers are constructed beforehand, stored in a table, and can be shared among different objects. When an application wants to create a new object with a small *int* value or a short *string*, instead of construction of a brand new object, the runtime just returns a reference to a previously created object and increments its *refcount* by one (for automatic garbage collection). However, this behavior is not desirable for tainted objects, since otherwise we must support objects with the same value and different taint labels. We modified this mechanism to prevent sharing of objects when they are tainted, discarding the original optimization for tainted objects.

Python libraries

As discussed in Section 5.1, for our privacy guarantees to hold MUTT must (1) disallow modifying the taint tags, and (2) control the sinks and sources of sensitive data. It is easy to achieve the former by disallowing dynamically loaded C-modules, since only C-level code can modify objects' taint structure. As a consequence, we had to statically link all common dynamic C-modules.

To control taint sinks, we modified the Python socketmodule to provide a more narrow interface and disallow arbitrary network connections. Besides AppScale's built-in application server and Django, the only uncontrolled source of data in the AppScale is *urllib* – a Python library for working with HTTP. We rewrote this library in C so that only untainted data could leave the system through urllib's interface.

Since all data flowing from OSNs should be tainted according to the specified policies, each OSN source needs to have a trusted module that can retrieve data from the service on behalf of the user and taint it before handling this data to the application. We implemented two simple modules to handle Facebook and Twitter data.

6.2 AppScale

We used AppScale as a basis for our prototype. AppScale can be deployed on Amazon EC2 as well as on clusters of local machines. To support taint-tracking and security policies we changed the application server to handle tokens and modified the database drivers so that tainted objects could be saved and retrieved from persistent storage without compromising data security.

We also modified the memcache library that ships with AppScale. Memcache is a distributed in-memory key-value data store useful for caching. We patched memcache's implementation of its binary protocol to enable serialization and deserialization of tainted objects.

7 Evaluation

For MUTT to be practical, it must have acceptable performance. To understand the sources of overhead in our prototype, we wanted answers to the following questions:

- What is the storage overhead of adding a taint structure to every object in the Python runtime?
- How does taint tracking affect the performance of basic operations in Python?

To evaluate our prototype we used a AppScale image with the modified Python runtime on a *large instance* (64-bit architecture, 7.5 GB memory, 2 virtual cores) on Amazon's EC2 cloud computing infrastructure.

7.1 Storage overhead

Listing 1 shows the actual taint structure used in our prototype. The size of this structure is 12 bytes on a 64-bit machine. As described in Section 6.1, each object in our Python runtime stores a pointer to a taint structure. Thus, the size of PyObject_HEAD increases from the original 24 bytes to 32 since the size of the pointer is 8 bytes. In addition, each object in Python has a pointer to a PyTypeObject that serves as a reference to the common methods and properties shared among all objects of the particular type. We added a flag to PyTypeObject indicating whether this data type is taint-aware for fast checks. The unmodified size of each PyTypeObject is 384 bytes, so after adding the flag of type *int* (along with additional 8 bytes from PyObject_HEAD) it increased to 396 bytes.

7.2 Runtime overhead

We first wanted to measure the overhead of taint tracking on basic primitive operations. Table 3 shows the results of these microbenchmarks. We recorded the latencies of three basic operations in (a) the unmodified Python, as well as the modified runtime with (b) non-tainted, and (c) tainted objects.

To evaluate the performance of tainted objects we varied the number of tokens in the arguments' token-sets from 1 to 50. For one token we tainted only one argument with exactly one token. For 10, 20 and 50 tokens we used objects with each token-set containing 5, 10 and 25 tokens accordingly. The size of the tokens we chose for the evaluation was 20 characters (Facebook's fbuid is 21 and Google's 15).

The latency of assigning a tainted integer is 0.22 μs and is comparable to assigning a non-tainted object (0.19μs). Since our taint-merging policies are identical for all binary operations we do not report a breakdown of our results for all of them, and show only the latencies for string concatenation and integer addition. Merging a non-tainted object and an object with one token is about two times slower than the same operation in an unmodified runtime.

We believe that the case with one token is most common. On the other hand, the latencies of merging objects with multiple tokens grows linearly with the number of tokens in their token-sets. However, we envision that tasks that manipulate data objects containing multiple tokens (e.g., complex objects) can be performed offline

(e.g., as cron jobs) and are not as time critical as online requests.

Finally, we measured the overhead of our implicit flow prevention mechanism. The additional check adds about 0.01 μs to the original comparison.

We also evaluated our prototype on *pybench* – a standard test suite shipped with the Python source code. In contrast to our own microbenchmarks where we measure the latencies of isolated operations, pybench is a more comprehensive set of tests that performs a combination of operations over specific types of data.

The latencies of running tests from pybench using both untainted and tainted objects are shown in Table 4. For these tests we used tainted objects with one token per object. Similar to our previous results, the latency of the tainted objects is about 2.5 times higher than the original. We believe that these numbers are reasonable, given that high network latencies dominate app performance.

8 Related work

xBook [29] is a framework for building privacy-preserving third-party applications for OSNs. Like MUTT its goal is to prevent malicious or buggy applications running outside of service-controlled datacenters from leaking users' data. Unlike MUTT, xBook's protects a client's web browser. xBook isolates DOM components with different privacy expectations by arbitrating all information flows between them. We did not address client-side threats in our work and instead focused on server-side attacks and violations. In addition, xBook provides protections at the granularity of a domain (e.g., Facebook), rather than a user. This prevents external sharing of users' information, but cannot prevent unwanted internal sharing. Finally, xBook requires applications to be split into functional components written in ADsafe – a safe subset of Javascript, whereas MUTT is integrated with a PaaS environment below a standard Django interface.

A number of systems have used coarse-grained information-flow control to prevent untrusted code from leaking sensitive information [33, 34]. Under such tracking, a trusted monitor maintains a set of labels indicating what kind of sensitive information coarse-grained objects, such as address spaces and files, contain. The primary difference between these systems and taint-tracking is the granularity at which labels are maintained. We used taint tracking in MUTT because it can be implemented below existing programming interfaces, whereas coarser-grained labels require decomposing even a simple OSN application into multiple functional components, e.g., a mashup getting data from multiple service providers.

Hails [19] and W5 [22] exemplify this coarse-grained approach to securing web apps. Like MUTT, these systems execute third-party apps within a trusted information-flow tracking execution environment. Both Hails and W5 separate apps into different classes of software components; one class can directly access sensitive data, while the other class can interact with the external clients. Communication between component classes is regulated according to access policies expressed using a simple logic in terms of integrity and sensitivity labels. The primary drawback of both systems is that they require all existing apps to be refactored. MUTT avoids this by implementing taint-tracking behind AppScale's existing runtime interface.

Like MUTT, Fabric [23] also uses language-level information-flow tracking to ensure consistent object dissemination and remote procedure calls. Fabric can also track sensitive information across distributed components and provide strong guarantees through static type checking. However, as with coarse-grained tracking, Fabric requires existing applications to be rewritten.

Finally, there is a large body of work [13, 16, 32, 28, 31, 14] using taint-tracking to defend against application bugs or system misconfigurations. The closest to our system in terms of implementation and design is Resin [32].

Resin is a language runtime that evaluates data-flow assertions when tainted data is about to leave the system. Resin assumes that programmers manually annotate objects containing sensitive data, and supply merging and export policies for such objects. Though Resin is the closest to MUTT's design and implementation, duplicating MUTT's protections on top of Resin would not be straightforward. First, one would have to develop default policies without breaking existing third-party applications. Second, one must isolate the PaaS components and language runtime. Third, one must provide a way to handle sensitive data from multiple service providers and multiple users to support mashups. MUTT addresses each of these issues.

9 Conclusion

In this paper we have presented the design and implementation of MUTT. MUTT is designed to prevent unauthorized internal sharing by OSN apps. It achieves this by integrating taint tracking into the runtime of a PaaS infrastructure and using a trusted driver architecture to proxy all interactions between an app and an OSN service. Our evaluation results show that taint tracking induces only 2.5 times overhead.

Operation	Unmodified (μs)	Untainted (μs)	Tainted (μs)			
			1 token	10 tokens	20 tokens	50 tokens
Assign (int)	0.19	0.19	0.22	0.22	0.22	0.22
Int addition	0.10	0.11	0.21	1.62	2.88	6.26
String concat	0.13	0.14	0.26	1.66	2.82	6.23
Compare (int)	0.07	0.07	0.08	0.08	0.08	0.08

Table 3: Microbenchmarks results

Test	Untainted (ms)	Tainted (ms)
CreateStringsWithConcat	115	250
StringSlicing	93	175
CreateUnicodeWithConcat	104	246
SimpleIntegerArithmetic	123	278
SimpleFloatArithmetic	120	266
SimpleLongArithmetic	104	272
SimpleIntFloatArithmetic	105	269

Table 4: Latencies of selected tests from *pybench* suite

Acknowledgements

We would like to thank our shepherd, Eric Eide, and the anonymous reviewers for their helpful comments. Our work was supported by the National Science Foundation under NSF awards CNS-0747283 and CNS-0916649.

References

[1] Amazon EC2 Facebook Application Hosting. http://aws.amazon.com/facebook-application-hosting/.

[2] Badoo. www.badoo.com.

[3] Facebook application statistics. http://statistics.allfacebook.com/applications.

[4] Facebook Developers Blog: Open Graph Beta. https://developers.facebook.com/docs/beta.

[5] Facebook Developers Blog: Permissions. https://developers.facebook.com/docs/reference/api/permissions.

[6] Facebook in Privacy Breach: Top-Ranked Applications Transmit Personal IDs, a Journal Investigation Finds. Wall Street Journal, October 18, 2010.

[7] Facebook statistics. https://www.facebook.com/press/info.php?statistics.

[8] Farmville. www.farmville.com.

[9] Lessons from FarmVille: How Zynga uses the Cloud. http://www.informationweek.com/news/global-cio/interviews/229402805.

[10] The Facebook Blog: debunking rumors about advertising and photos. https://www.facebook.com/blog.php?post=110636457130.

[11] Twitter Suspends UberMedia Clients For Privacy And Monetization Violations, Trademark Infringement. http://techcrunch.com/2011/02/18/twitter-suspends-ubermedia-clients-ubertwitter-and-twidroyd-for-violating-policies/.

[12] VKontakte. http://vk.com.

[13] W. Chang, B. Streiff, and C. Lin. Efficient and extensible security enforcement using dynamic data flow analysis. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 39–50, New York, NY, USA, 2008. ACM.

[14] E. Chin and D. Wagner. Efficient character-level taint tracking for java. In *Proceedings of the 2009 ACM workshop on Secure web services*, SWS '09, pages 3–12, New York, NY, USA, 2009. ACM.

[15] N. Chohan, C. Bunch, S. Pang, C. Krintz, N. Mostafa, S. Soman, and R. Wolski. Appscale: Scalable and open appengine application development and deployment. In *International Conference on Cloud Computing*, October 2009.

[16] S. Chong, K. Vikram, and A. C. Myers. Sif: Enforcing confidentiality and integrity in web applications. In *In Proc. 16th USENIX Security*, 2007.

[17] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976.

[18] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, 2010.

[19] D. B. Giffin, A. Levy, D. Stefan, D. Terei, D. Mazières, J. C. Mitchell, and A. Russo. Hails: protecting data privacy in untrusted web applications. In *OSDI '12*, October 2012.

[20] P.-E. Gobry. Huge facebook app loses 75% uniques after facebook threatens it, April 2011. http://www.businessinsider.com/badoo-facebook-2011-4.

[21] P. Goss. Facebook shuts down application over privacy, July 2008. Tech Radar.

[22] M. Krohn, A. Yip, M. Brodsky, R. Morris, and M. Walfish. A World Wide Web Without Walls. In *HotNets '07*, November 2007.

[23] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: a platform for secure distributed computation and storage. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 321–334, New York, NY, USA, 2009. ACM.

[24] E. Mills. Facebook suspends app that permitted peephole, June 2008. CNET News.

[25] A. C. Myers and B. Liskov. A decentralized model for information flow control. *SIGOPS Oper. Syst. Rev.*, 31:129–142, October 1997.

[26] A. Sabelfeld and A. Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal*, January 2003.

[27] F. B. Schneider, K. Walsh, and E. G. Sirer. Nexus authorization logic (nal): Design rationale and applications, September 2009. Cornell Computing and Information Science Technical Report.

[28] J. Seo and M. S. Lam. Invisitype: Object-oriented security policies. In *Network and Distributed System Security Symposium*, 2010.

[29] K. Singh, S. Bhola, and W. Lee. xBook: redesigning privacy control in social networking platforms. In *Usenix Security Symposium*, 2009.

[30] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. *SIGARCH Comput. Archit. News*, 32:85–96, October 2004.

[31] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. Taj: effective taint analysis of web applications. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 87–97, New York, NY, USA, 2009. ACM.

[32] A. Yip, X. Wang, N. Zeldovich, and M. F. Kaashoek. Improving application security with data flow assertions. In *Proceedings of the 22th ACM Symposium on Operating Systems Principles (SOSP '09)*, Big Sky, Montana, October 2009.

[33] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazires. Making Information Flow Explicit in HiStar. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI)*, November 2006.

[34] N. Zeldovich, S. Boyd-wickizer, and D. Mazieres. Securing distributed systems with information flow control. In *In Proc. of the 5th NSDI*, pages 293–308, 2008.