GraphAudit: Privacy Auditing for Massive Graph Mining

Adwait Nadkarni
NC State University
anadkarni@ncsu.edu

Anmol Sheth
Technicolor
anmol.sheth@technicolor.com

Udi Weinsberg
Technicolor
udi.weinsberg@technicolor.com

Nina Taft
Technicolor
nina.taft@technicolor.com

William Enck
NC State University
enck@cs.ncsu.edu

ABSTRACT

Data privacy audits that ensure policy compliant usage of personal data are increasingly enforced (internally or externally) on service providers that amass and process user data. Existing approaches to privacy auditing fall short of addressing the challenges introduced by modern personalized services that analyze user data using large scale machine learning and data mining (MLDM) algorithms, and provide users custom privacy controls. In this paper, we present GraphAudit, an auditing framework for large-scale graph mining platforms that can check the compliance of a wide range of expressive privacy policies. GraphAudit achieves this by reconstructing the runtime context of data use by MLDM algorithms by logging data accesses and tracing data flows. We implement GraphAudit over GraphLab, a popular distributed graph mining framework and evaluate its performance using commonly used MLDM algorithms and real worlds graph datasets. Our evaluation shows that GraphAudit performance overheads are moderate and exhibits scaling properties similar to GraphLab. Moreover, the overheads can be reduced by amortizing the I/O overhead of logging across distributed machines in the cluster. This results in overheads relative to GraphLab that are as low as 5.76x, with execution times that can automate privacy auditing compared to the otherwise manual and error prone approaches currently used.

1. INTRODUCTION

Online services commonly personalize the user experience by aggregating and analyzing user's personal data and preferences. Users must consent to the use and disclosure of their personal data by accepting the service provider's privacy policy. These policies typically describe what data is collected, how it is used, and with whom the data is shared.

Violations of end-user privacy policies by service providers are growing in frequency. Many of these violations are internal to a service provider. A recent study [20] shows that unintentional or intentional internal disclosure of user data accounted for approximately half of the privacy data violations publicly reported in the US from 2005 to 2011. For example, one of the complaints filed by the US Ferderal Trade Commission (FTC) against Facebook [5], listed instances where user data that was supposed to be shared with "Friends Only" was accessible to third-party Facebook applications and advertisers. Penalities for such violation include not only financial penalties, but also mandated periodic *privacy audits* to ensure that service provider maintains policy compliant use of end-user personal data. Despite this mandate for auditing, the FTC has few suggestions as to how to go about auditing the data use practices of these service providers.

Privacy audits today [11, 14, 26, 40] are largely based on guidelines from government bodies or industry consortia [4,6,12] that are typically carried out using a checklist of questions related to the following steps: i) make an inventory of all data collected; ii) explain the purpose of such data collection; and iii) state data management practices such as how the data is stored, who has access, and how and when it is deleted, etc. While some of these questions can be automated, the checklist based approach audits a service providers data use practices at a "high-level" without reasoning about the implementation details of the private data use. Alternately, manual approaches based on interviews, code reviews, and log inspection can potentially audit the service provider's practices in detail. However, these approaches cannot scale to large distributed software systems, are inherently error prone and cannot analyze the runtime context of private data use. Consequently, the view today is that there is a vacuum in automated auditing of privacy policy compliance that extend beyond rudimentary checklists [9].

In recent years, auditing "big data" personalized services (e.g., social networks and recommenders) is becoming an increasingly difficult task for a couple of reasons. First, service providers commonly use sophisticated Machine Learning and Data Mining (MLDM) frameworks (e.g., GraphLab [31] and Apache Mahout [2]) to analyze user data. Such frameworks typically load the *complete* data once and then execute different MLDM algorithms on partial views of the loaded data. Consequently, auditing the access and flow of user data through these frameworks requires runtime monitoring and logging of data use. Second, modern web services increasingly provide user-specific, fine-grained privacy controls, enabling users to limit the usage and disclosure of their personal data, e.g., *name* is public, but *age* should only be visible to friends. Such user-specific privacy policies substantially increases audit complexity.

In this paper, we propose GraphAudit as a novel approach for auditing service providers using MLDM frameworks for data analysis. GraphAudit combines lightweight data flow tracking and data access logging to enable automated proofs of policy compliance. Specifically, GraphAudit builds upon the common vertex-centric model for parallel graph processing, wherein algorithms iteratively execute on each vertex of the data graph. GraphAudit logs access to data attributes made by the vertex program, automatically derives a taint label from data attributes (e.g. *rating, movieid*), and propagates the labels throughout the computation. Logs generated by GraphAudit are post-processed and represented as a data flow graph upon which policy compliance checks are performed using efficient graph queries.

Related to GraphAudit is a recent work by Sen *et al.* [39], sharing a similar vision of automating the currently manual auditing process. The authors propose a system and policy language that automates privacy policy compliance auditing, focusing on the Bing search engine. However, unlike GraphAudit, their work only audits

global policies that apply to *all* users using the service. More importantly, policies are audited based on *static* data analysis, namely analyzing database queries, source code, and metdata associated with user roles and data files. Such static analysis techniques cannot track data use in MLDM algorithms that have complex data flows with iterative processing of the data, a shortcoming that is even identified by the authors.

We address several key challenges in the design and implementation of GraphAudit, which form our main contributions:

Fine-grained logging. A key design objective of GraphAudit is to log accesses to *individual* data fields analyzed by the MLDM algorithms to support fine-grained, user-specific privacy settings. To address this, GraphAudit intercepts and logs all data accesses along with their runtime context performed by the MLDM algorithm.

Tracking sensitive data. MLDM algorithms access and combine multiple data attributes during computation. For example, a classifier may select *age*, *gender*, and *location* attributes to develop a model, but that combination of attributes may violate the privacy policy [41]. GraphAudit provides a label propagation mechanism that automatically extracts labels from attribute names and tracks the label set associated with derived values.

No changes to MLDM algorithm logic. GraphAudit does not require the MLDM expert to modify the core logic of the vertex program. The only change required is to call two designated methods when reading and writing data, a change that can be mostly automated. Our experience with modifying existing MLDM algorithms shows that these changes are easy and quick to implement and require the developer to change a small fraction (3-9%) of the original C++ code.

Supporting different deployment scenarios. The overheads introduced by the runtime tracking and logging of data use should not restrict the deployment scenarios of GraphAudit. Our evaluation of GraphAudit's computational and I/O overheads using large real-world datasets shows that the overheads are moderate (as low as 5.7×), and can be further reduced with parallelization. This enables GraphAudit to either be deployed online for continuous monitoring of policy compliance for services that do not require interactive responses (e.g., computing recommendations), or be deployed on demand as requested by the auditor for services that require interactive latencies (e.g., social graph search).

Auditing diverse privacy policies. GraphAudit supports the auditing of a wide range of privacy controls and policies provided by existing online services like Facebook and Google. These policies range from limiting access to private data (e.g., user's location must not be used) to controling the flow of private data (e.g., share only with friends). Private data access policies can be directly audited from GraphAudit logs while monitoring the extent of data disclosure is audited by generating data flow graphs from GraphAudit logs and querying the resultant graph. Generally, these tasks are practically impossible on modern massive datasets using current manual auditing techniques.

The remainder of this paper proceeds as follows. Section 2 motivates GraphAudit and provides background necessary for understanding its design. In Section 3 we describe the GraphAudit design and Section 4 details our C++ implementation of GraphAudit on GraphLab. Section 5 details the performance evaluation of GraphAudit using several MLDM algorithms. Section 6 describes a used case showing how GraphAudit logs are used to check policy compliance. We discuss practical deployment considerations in Section 7 and related work in Section 8. We conclude the paper in Section 9.

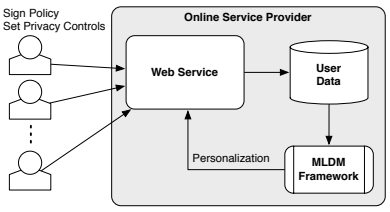


Figure 1: Overview of an online personalized service.

2. MOTIVATION AND BACKGROUND

Before discussing GraphAudit, we begin by describing the types of online services it is designed to audit and motivate the need to automate the policy compliance audits for such services. Finally, we provide a short background on graph parallel abstractions, which is an increasingly popular and scalable approach to processing data aggregated by these online services.

2.1 Personalized Services

Figure 1 overviews the main entities involved in a personalized service. Before using the service, users sign and consent to a privacy policy provided by the service provider. Optionally, users provide fine-grained privacy controls to limit the use and disclosure of their data. User data aggregated by the service provider is made available to internal employees (or authorized external partners) to analyze user behavior and improve the personalized service. This aggregated data is often large and consists of a number of different attributes about the user. Analytics over this data uses large scale distributed Machine Learning and Data Mining (MLDM) frameworks. GraphAudit focuses on enabling policy compliance auditing for the data analytics performed on the user data.

To better understand the entities, we provide the following two representative personalized services.

Movie Recommendation Service. Consider an online movie streaming service that provides personalized movie recommendations for users to watch. Users provide their preferences by rating movies. The service uses collaborative filtering algorithms to mine user preferences and provide personalized recommendations. As privacy controls, the service lets users (a) completely opt-out of using ratings for recommendations, (b) opt-out of using specific watched or rated movies (e.g., don't use any viewed x-rated movies), or (c) opt-out of using specific user profile information for recommendations (e.g., age, gender or location).

Social Networking Service. Consider an online social networking service that allows users to share updates (e.g., posts, pictures, video) by setting explicit *friendship* relationships. The service mines data about users and their social network to recommend new friends, deliver targeted advertising, or let users search interesting facts about their friends or the social network in general. As privacy controls, the service lets users tag specific attributes of their profile or uploaded content as (a) private, (b) shared only with friends, (c) shared with friends of friends, or (d) shared with everyone.

These privacy policies are either agreed to by users when they sign (or accept) the legal privacy policy of the service, or set by the users via fine-grained privacy controls.

By studying the privacy controls and polcies provided by existing popular personalized services, we group them into one of the following two broad types.

P1 *Prevent use of specific private information.* In the movie recommendation service, users may opt-out completely. In the social networking service, users may choose to keep pri-

vate specific attributes of their profile (e.g., age, income) or friendship relationships. These private attributes and all derived information should not be used or accessed by MLDM algorithms for personalization.

P2 *Limit the extent of disclosure of private information.* In the social networking service, users may explicitly limit the disclosure of private information (e.g., sexual orientation) to specific friends or friend-of-friends. MLDM algorithms should limit the flow of this information or any information derived from it to the intended scope defined by the user.

2.2 Policy Compliance Audits

Privacy audits are increasingly used to prevent privacy policy violations. Some audits are judicially mandated. These audits investigate the reported misuse of user data and are commonly performed periodically to ensure service providers maintain compliance. For example, the FTC mandated audits by independent external parties for Facebook [5] and Google [7] for violating privacy policies. Additionally, audits are sometimes voluntarily performed internally to ensure policy compliance. Such internal audits can minimize or potentially even prevent the damages caused by non-compliant practices.

In this paper, we seek to develop an auditing mechanism for existing graph-based MLDM frameworks that can automatically monitor and log the runtime context of private data use by MLDM algorithms. Capturing such context enables the programmatic reconstruction of the sequence of data accesses made by the algorithms, as well as the resultant flow of private information. Privacy policy compliance can be then checked for by analyzing how private data was accessed and its propagation in the graph. We choose a dynamic analysis approach, because violations are driven by the user-data. Static analysis considers all possible cases, which may not map to reality. Furthermore, dynamic analysis can more easily react to privacy controls specified in user-data.

Threat Model. GraphAudit is designed to automate the MLDM portion of a privacy audit. Recall that audits may be voluntary or judicially mandated. In both cases, we assume that the service provider does not attempt to maliciously circumvent auditing. For example, a service provider might find that complying to privacy policies reduces the value of MLDM inference and therefore decides to knowingly violate the judicial mandate. We do not consider such cases. Rather, GraphAudit allows the service provider to demonstrate *due diligence* in complying to end-user privacy policies. GraphAudit's automated MLDM privacy auditing can be combined with manual audits to ensure GraphAudit is properly used.

2.3 Graph Parallel Abstractions

Many personalization algorithms focus on modeling the explicit dependencies that exist in the data. For example, a movie recommendation system explicitly models users that rate movies similarly, and a social network analyzes the structure of the social graph to recommend new friends. The limitations of data parallel abstractions, like MapReduce [23], to scale such dependent computation to large datasets, has recently led to new *graph parallel abstractions* [2, 31, 32] that maintain the explicit data dependencies. The dependencies in the data are represented as a graph and computation is structured as programs that run on each vertex of the graph. For example, user data from movie recommendation service is structured as a bipartite graph between users and movies with the edges representing the rating given by a user to a specific movie. Algorithms represented as vertex programs compute on the data associated with the vertex or edge and read or write to data on adjacent vertices and edges. Such graph parallel abstractions enable MLDM algorithms to scale to massive datasets by executing vertex programs in parallel while providing consistency guarantees.

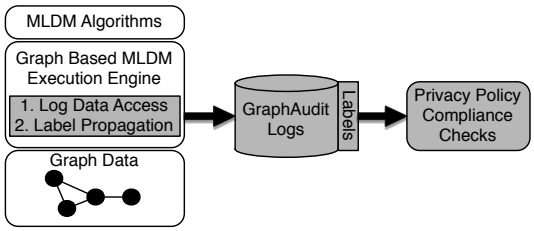


Figure 2: GraphAudit Design Overview

Vertex programs are composed of three primary phases – *Gather*, *Apply*, and *Scatter*. In the *gather* phase, the vertex program gathers data from adjacent vertices and edges and data is aggregated by an algorithm specific sum operation. The result of this summation is passed to the *apply* phase in which a vertex updates its own state. Finally, *scatter* phase uses the new data to update the data on adjacent edges and signal adjacent vertices to re-execute. The execution engine manages the scheduling of vertex programs to support a particular consistency model such that vertex programs operate on consistent data in spite of parallel execution.

3. GRAPHAUDIT DESIGN

In this section we describe the design of GraphAudit, a policy compliance auditing framework that extends existing graph-based MLDM frameworks. GraphAudit's extensions enable the reconstruction of the *runtime context* of data accesses made by the MLDM algorithms operating on graph data. This is achieved by interposing and logging data accesses made by MLDM algorithms and tracing the flow of private data through the graph. The programmatic analysis of these data flows enables automated auditing of expressive privacy policies.

Figure 2 provides an overview of an MLDM framework extended by GraphAudit to support auditing based policy compliance. The shaded components represent the GraphAudit specific components. The design of each GraphAudit component addresses several challenges and requirements that are specific to MLDM algorithms operating on large graph data. However, the GraphAudit design is general and applies to many practical MLDM frameworks that support the vertex oriented graph-parallel computation abstraction [31, 32]. In the rest of this section we describe these components.

3.1 Logging Data Accesses and Label Propagation

GraphAudit extends existing MLDM frameworks by introducing two functionalities: 1) a fine-grained logging mechanism to log access to the structural graph elements as well as the data itself, and 2) a coarse grained label propagation mechanism to track data flows across the execution context of the vertex program.

Access to graph elements. A common use case of MLDM frameworks is to have a data processing pipeline that starts with reading a complete (large) graph into the MLDM framework. Different MLDM algorithms may create partial views of the graph, e.g., to support user's privacy controls that restrict the use of specific data, and execute on this partial view of the graph.

To support policy compliant usage of structural graph elements (i.e., vertices and edges), GraphAudit provides two precondition

```
v_score = vertex.get_data("score")
v_gender = vertex.get_data("gender")
if (vertex.get_data("age") < 20) {
  v_score += 1
  vertex.set_data("score", v_score)
} else {
  v_score +=  2
  vertex.set_data("score", v_score)
}
```

Figure 3: Example code showing the GraphAudit data access implementation for GraphLab using `get_data` and `set_data` wrapper functions.

abstract functions that enable the developer to add logic for excluding access to a vertex or an edge based on a policy. By logging the graph elements that were not excluded, GraphAudit can determine whether a policy violation occurred, regardless of whether data accesses were performed.

For example, a social network can recommend a new friend to a user based on their shared friends. If some of their shared friends choose not to disclose these friendship links, an algorithm that simply uses these edges without accessing any associated data already violates a policy. The programmer can use GraphAudit's precondition functions to exclude these private friendship links *before* the friendship recommendation algorithm runs and thereby ensure that the excluded links are indeed not used. If the programmer decides to not filter a vertex or an edge, GraphAudit logs its use, irrespective of whether a data access was performed.

Access to graph data. Vertex programs compute on *data* that broadly refers to either input data (e.g., a user profile in a social network) or algorithm data (e.g., rank variable for Pagerank, latent factors for collaborative filtering, etc.). These data attributes are commonly stored in a programmer-defined structure, for example a C-language `struct` in GraphLab. A key challenge for GraphAudit is to log accesses to individual data attributes contained in the programmer-defined structures while allowing the MLDM algorithms the flexibility to define their own data representations. Moreover, GraphAudit should not require *any* modifications to the core logic of the MLDM algorithm implementation.

To address this, GraphAudit provides the MLDM developer with wrapper functions to read and write individual data fields associated with the vertex and edge. These wrapper functions receive the data field name as a parameter, and store the values in a generic key-value map. In addition, they log accesses to the data fields along with the id of the vertex or edge of the graph associated with the data. As a concrete example consider the code fragment in Figure 3, which uses the actual syntax of our C++ implementation of GraphAudit. In this code, the algorithm developer simply replaces direct accesses to the `struct` fields with GraphAudit's `get_data()` and `set_data()` wrapper functions.

Label Propagation. An important aspect of auditing data use by MLDM algorithms is to monitor how data attributes are combined with each other, which could potentially result in a non-compliant use of data. Consider the following example policy – an MLDM algorithm can use a user's age, gender or location fields individually but cannot disclose any data derived by *combining* them, as doing so may make users personally identifiable [41]. Auditing such policies requires tagging data fields with labels and propagating labels to track derived data.

GraphAudit performs coarse grain label propagation within the data access wrapper functions and propagates labels throughout the execution context of the vertex program. Each data field is associated with a *data field label set* which is initialized to the data field name (e.g., "age"). Additionally, GraphAudit maintains a *context label set* that tracks all labels associated with the execution context. Within the execution of a gather, apply or scatter, every read access of a data field accumulates its label set into the context label set (essentially holding the labels of all fields that were read in the context). When writing a data field, the context label set is added to the data field label set (distinct union of the two sets).

The set of labels associated with a data field are stored alongside the data field value, and are thus persistent across the entire execution of the algorithm. To support explicit data flows across execution contexts, GraphAudit also propagates the context label set between the gather and apply phases.

For the code snippet in Figure 3, each call to `get_data()` results in the accumulation of the corresponding label in the context label set. In `set_data("score")`, the accumulated context labels $\{age, gender\}$ propagate to the `score` field. This example illustrates a caveat of coarse grained label propagation where every data attribute read by the vertex program influences subsequent writes – `score` is tainted with the label `gender` even though gender was not used explicitly. This conservative approach to label propagation is suitable for the purposes of privacy auditing, where careless access of data (even if not used eventually) should be avoided. Another advantage that comes with this approach is the ability to account for implicit flows. Furthermore, the alternative approach of performing fine-grained runtime data flow analysis can prohibitively increase the runtime overheads of the vertex program, especially in massive datasets.

3.2 GraphAudit Log Structure

GraphAudit logs are designed to provide the following two properties. First, the logs must be *compact*; MLDM algorithms running on massive graphs may generate a large number of data accesses and the resultant logs should reduce storage. Second, each data access log must be *complete* and contain the necessary runtime information about the vertex program. To address this, GraphAudit stores the following two types of logs:

Graph Elements Access Logs. GraphAudit logs the set of vertices and edges that were used by the MLDM algorithm. This is achieved by logging the graph structure at the end of the MLDM algorithm using the mechanism described in Section 3.1.

Data Access Logs. GraphAudit logs every data access occurring in each phase of the vertex program using the following log format:

```
<source_type>|<access_type>|<id>|<field>|<label>
```

Each log entry contains a `source_type` which indicates whether a vertex or an edge is associated with the data access; `access_type` indicates whether a read or a write operation was performed; `id` is the identity of the vertex (*vid*) or edge ($< source\_vid, destination\_vid >$) associated with the data access; `field` is the data field being accessed, which is encoded in a single byte to reduce memory and disk storage; `label` is the data field label associated with the field at the time of access which is encoded as a bit array. Additionally, before the execution of a gather/ apply or scatter, GraphAudit logs information about the current execution context, i.e., the vertex program's identity, the phase of execution (gather, apply or scatter), and the iteration number. This information is essential for correctly analyzing the logs in order to enable a correct understanding of the runtime data flow, as we will further describe in Section 3.3.2.

3.3 Log Analysis for Policy Compliance

In this section we describe the log analysis methods used by GraphAudit to audit policy compliance for the two main policy categories, i.e., policies that 1) limit the use of private data, and 2) limit the extent of data disclosure. For each category we provide specific examples motivated by the social network and movie recommendation web services described in Section 2.1. Note that the categories are not limited to the two services but rather encompass a wide range of privacy controls and policies that we surveyed across many services.

3.3.1 Policies that limit use of private data

Since data is explicitly structured as a graph, monitoring the use of private data can be directly audited from the GraphAudit logs as follows:

Vertex Use. Users may explicitly want to opt-out of any personalization service or opt-out certain entities (e.g., opt-out all x-rated movies for recommendations). Auditing such a policy is done by directly checking if the vertex corresponding to a opted-out user or entity is present in the GraphAudit graph access logs.

Edge Use. Users may not want to have associations with specific users or entities be used by the web service. Auditing such policies is done by checking if any edges associated with the user or entity are present in the GraphAudit graph access logs.

Data field use. Users may mark specific data fields as private, e.g., a user's age in her social network profile. Such policies are audited by directly searching the data access logs for the specific edge or vertex `id` and `field` entries that match the user and the field name.

Derived Data. Users may preclude certain data fields from being combined, e.g., to prevent learning algorithms from inferring undisclosed user characteristics. Auditing such policies requires checking the field labels associated with the access logs and verifying that none of the field labels have all the bit positions set that correspond to the fields that should not be combined.

3.3.2 Policies that limit extent of data disclosure

Auditing policies that limit the extent of data disclosure requires re-constructing the flow of data from the GraphAudit data access logs. The primary challenges in re-constructing the data flows from logs are 1) to account for the explicit ordering of read and write log events to prevent any false flows, and 2) ensuring that the sequence of logged events follow the consistency model adopted by the execution engine. We discuss this latter challenge in Section 4 and focus on preventing false flows.

GraphAudit constructs a data flow graph (hereby called a DFG) that encodes the explicit dependencies between read and write accesses made to data fields by the vertex program. A vertex in the DFG corresponds to a data field associated with a vertex or edge that was accessed. For example, access (read or write) to field $f1$ associated with vertex A and field $f2$ associated with edge $A \rightarrow B$ are represented as DFG vertices $A : f1$ and $(A,B) : f2$, respectively. Directed edges in the DFG denote a read followed by a write. For example, $A : f1 \rightarrow (A,B) : f2$ indicates that $f1$ of A was read followed by a write to $f2$ of the edge $A \rightarrow B$.

The above described approach of generating the flow graph does not explicitly account for the *order* of data accesses and can potentially cause false data flows. Consider a graph with three vertices, A, B, C, and an ordered sequence of data accesses $B : f1 \rightarrow C : f1$ followed by $A : f1 \rightarrow B : f1$. Simply creating these two edges in the DFG would result in an incorrect flow of $f1$ from vertex A to C. To address this, before adding an edge to the flow graph, GraphAudit checks if the destination vertex is contained in the flow graph and it already has an out edge. This implies that data from this vertex already propagated to other vertices, thus new writes to the data should not impact previous reads. Therefore, if an out edge exists, GraphAudit creates a duplicate of the destination vertex, creates an edge from the original to the duplicate destination vertex, and then creates the new edge between the source and the duplicated destination vertex. In our example, this results in a new vertex B^* in the DFG and the following three edges: $B : f1 \rightarrow C : f1$, $B : f1 \rightarrow B^* : f1$, and $A : f1 \rightarrow B^* : f1$.

GraphAudit processes the log events along with the context information of the vertex program to generate a data flow graph. Within the execution context of a gather, apply or scatter, GraphAudit creates a vertex for every read access log and adds these vertices to a *read set*. One encountering a write access log, GraphAudit creates an edge from every vertex in the read set to the vertex on which data is being written. To account for explicit data flows that occur from the gather to the apply phase within a given iteration of the vertex program, GraphAudit propagates the complete read set from the corresponding gather phase and continues processing the logs as described above. The read set is otherwise local to a vertex program and is cleared after every iteration of the vertex program.

The resultant data flow graph enables the auditing of a wide range of data flow related policies. A policy where a user in a social network chooses to limit disclosure of a private data field (e.g., age) to friends-of-friends is audited by computing all paths from the vertex associated with the data field and checking that all destination vertices are within two hops in the original graph.

4. GRAPHAUDIT IMPLEMENTATION

In this section we describe the implementation of the GraphAudit design. We chose to implement GraphAudit over GraphLab version 2.1 [25], which is implemented in C++. GraphAudit extends two primary components of GraphLab – the *Distributed Graph API* and the *Execution Engine*. For evaluation, we also modified three MLDM algorithms, namely Pagerank[1], Connected Components, and Alternative Least Squares (ALS). For each component, we first describe its function within GraphLab, and then the modifications required to incorporate the GraphAudit design. It is important to note that the GraphAudit design makes no assumptions of the specific implementation of the graph parallel abstractions. Alternately, GraphAudit can be implemented for other graph parallel abstractions like Apache Giraph [1].

4.1 Distributed Graph API

GraphLab provides a unified API to interface with the directed graph data structure that is distributed across multiple machines in the cluster. The API enables the construction of the graph, storing and retrieving data associated with the graph, and performing basic operations on the graph. The API enables the MLDM algorithm developer to define custom edge and vertex data structures, commonly in the form of a C style `struct`. These user-defined data structures are used to initialize two GraphLab templates, `vertex_data_type` and `edge_data_type`, that are used for storing the user-defined data attributes on vertices and edges, respectively. GraphAudit makes the following modifications to the distributed graph API:

Explicit data access. GraphAudit enforces all data accesses to be logged by precluding direct access to user-defined data. GraphAudit provides the developer with a `DataMap` class that permits data

[1]The appendix provides C++ code snippets for Pagerank implementation in GraphLab and GraphAudit.

accesses only through two functions – `get_data(key)` and `set_data(key, value)` where the `key` is the name of the data field that needs to be accessed. The `DataMap` object stores the data in a designated associative map array, that uses the field name as a key. In addition to mediating access to the data fields, the functions also logs every access to the data field using the log format described in Section 3.2. In order to maintain type-safety when accessing data attributes, we used a `double` type for all attributes, which is commonly used data type for MLDM algorithms. GraphAudit signals the execution engine to start only if both the `vertex_data_type` and `edge_data_type` are not initialized using the `DataMap` class.

Data field labels. In addition to the data field storage, `DataMap` stores the label sets associated with each data field (data field label set). The labels are stored in a compact dynamically-sized bit array initialized to 32 bits (we use the Boost C++ library [3]). Each data field name is assigned a unique index in the label bit array upon the first time the `set_data()` function is invoked. This assignment is done by maintaining a globally synchronized map of field names and bit position. To reduce the synchronization overhead, threads cache the assignment of field names once they are initialized.

Label propagation within an execution context. In GraphLab, each execution context is represented by a Posix thread (pthread) that runs a vertex program function (gather, apply or scatter). GraphAudit performs label propatation within an execution context requires the propagation of labels between the *context label set* and the *data field label set*. GraphAudit uses thread specific key-value storage (with the standard pthread functions `pthread_getspecific()` and `pthread_setspecific()`) for accessing the context labels. For every read access of a data field (`get_data()`), GraphAudit propagates the labels associated with the data field to the context label, and for every write to a data field (`set_data()`), GraphAudit propagates the context label set to the data field label set. Label propagation is performed using binary OR operations on the label bit arrays.

4.2 Execution Engine

GraphLab's execution engine performs two key tasks. First, it implements a scheduler that schedules the execution of vertex program for vertices that are signaled to execute in the current iteration of the MLDM algorithm. The order of execution (including the gather, apply and scatter phases) determines the consistency properties of the data accesses made by the vertex program. Second, the execution engine manages synchronization of data across all the replicated vertices in the cluster[2]. Each replicated vertex (or mirror) is assigned a master vertex running on one of the machines in the cluster. At the end of the gather phase, all the mirrors of a vertex message the accumulated data to the master. The master alone executes the apply phase and messages all the mirrors the updated data after the apply phase terminates.

GraphAudit makes the following changes to the synchronous execution engine of GraphLab:

Logging graph use. GraphAudit execution scheduler calls the `vertex_verify()` and `edge_verify()` precondition filter functions that are implemented in the vertex program. The `vertex_verify()` function is called before the scheduler invokes the vertex program on a vertex and the `edge_verify()` function is called before invoking the gather on edges in the graph. If the verify functions returns `True`, the vertex/edge access continues and is marked as *used*. To limit overhead, GraphAudit assumes the first response as final and logs the set of vertices and edges used in the computation at the end of the execution of the MLDM algorithm.

Label propagation across execution contexts. GraphAudit monitors information flows that occur across execution contexts between the gather and apply phase. GraphAudit propagates the context labels of the gather phase by extending the default GraphLab `gather_data` type to `labeled_gather_data` type that contains the context label set of the gather phase. The master node receiving the `labeled_gather_data` objects first applies the labels to it's context label set before executing the $+=$ operation on the gathered data. Finally, when the engine schedules the apply phase, GraphAudit propagates the accumulated labels to the context label set associated with the apply phase. This, GraphAudit ensures that labels propagate along with the data even when data flows occur in a distributed setting.

Consistency and ordering of logs. The consistency and ordering properties of GraphAudit logs are based on the scheduler and consistency mechanisms provided by the execution engine. Since the GraphAudit implementation does not impact these properties of the execution engine, the resultant logs maintain the same properties. For example, the synchronous engine ensures a deterministic execution even in a distributed setting. Additionally, the engine implements a vertex consistency model by locking vertices in the execution context of current vertex program. Consequently, GraphAudit logs maintain the same vertex consistency property and log the same runtime ordering of accesses as made by the MLDM algorithm.

Flushing logs to disk. GraphAudit creates thread-specific output streams for logs. GraphAudit writes the logs to the streams, but does not explicitly flush logs to disk, relying instead on the underlying OS for that purpose. We evaluate the performance overheads with writing logs in Section 4.

4.3 MLDM Algorithms

The GraphAudit implementation requires the following two straightforward modifications to the MLDM vertex programs that does not require any changes to the core logic of the MLDM algorithm. First, GraphAudit requires that the vertex and edge data be initialized using the DataMap and all accesses to the data fields use the `get_data(key)` and `set_data(key, value)` functions. Making these changes for the three algorithms requires modifying 71 lines of code (9.69%) for ALS, 9 lines of code (3.38%) for Pagerank, and 11 lines of code (4.44%) for Connected Components. Second, GraphAudit optionally requires the MLDM programmer to implement the two precondition filter functions[3].

4.4 Audit Log Analysis

Implementing the log analysis for policies that limit the extent of disclosure requires processing the data flow graph in a way that is specific to the policy. Consider the policy where a user wants to limit the disclosure of location data to friends-of-friends in the social graph. Simply computing path lengths originating from vertices that contain the location data field is not sufficient – the path length computation must discount the redundant vertices in the flow graph, edges in original graph that are represented as vertices in the flow graph, and vertices in the flow graph corresponding to other data fields associated with the same user.

To address this, we make the following subtle change to the data flow graph by introducing edge weights as follows. An edge weight

[2]GraphLab distributes the edges across machines (e.g., random assignment) in the cluster and replicates all the vertices.

[3]The preconditions default to `True`, i.e., the graph element is used.

of 0 is added to every out edge from a redundant vertex, vertex that corresponds to an edge in the original graph, and from a vertex containing a data field that is not being audited. Every other edge has a weight of 1. Path lengths are computed by a modified Dijkstras algorithm that traverses a path but counts lengths by adding the edge weights. This resultant path length directly maps to the path lengths in the original graph.

Implementing the log analysis for policies that limit the use of data is straightforward as it is performed by running regular expressions matches over the logs.

5. PERFORMANCE EVALUATION

In this section we empirically evaluate the performance of GraphAudit and characterize the performance overheads associated with it. We study the scalability of GraphAudit using three different MLDM algorithms, executing on varying graph sizes, across two deployments – a multi-core server and Amazon EC2 cluster instance.

Our performance evaluation of GraphAudit seeks to address the following three main questions:

- What attributes of the vertex program impact the runtime execution time of GraphAudit? (Section 5.1)
- How does the performance of GraphAudit scale relative to GraphLab for increasing data graph size? (Section 5.3)
- How does the GraphAudit overhead scale relative to GraphLab with increasing machines in the cluster? (Section 5.4)

5.1 MLDM Algorithms

In order to understand the attributes of MLDM algorithms that GraphAudit's performance, we consider the following three MLDM algorithms:

Pagerank is a widely used link analysis algorithm that measures the relative importance of a vertex in a graph. In a social network, Pagerank can be used for identifying influential people for applications such as targeted advertising and friend recommendations. The algorithm iteratively updates the rank of each vertex and converges when the updates made to the rank estimate are below a given threshold.

Connected Components is an algorithm used to identify connected sub-graphs, e.g., to identify a connected group of friends that share a specific attribute (e.g., they all like Sports). The algorithm converges when all connected components have been identified.

Alternating Least Squares (ALS) is a matrix factorization method used by recommender systems to provide personalized recommendations to users. Matrix factorization decomposes a sparse user-item rating matrix into dense user and item profiles with small dimensions, which were shown to be sufficient to accurately predict the ratings a user will give to an item [30]. ALS is a matrix factorization algorithm that iteratively solves linear least squares equations for the user and item profiles, and converges when the error in the estimated ratings is below a threshold (or stops when the maximum number of iterations was reached).

Table 1 summarizes the three key properties of MLDM algorithms that impact the performance overheads of GraphAudit.

Data Type. Unlike GraphLab where data fields are accessed by reference, the vertex program in GraphAudit accesses data fields by value, which means that data needs to be copied. This is required to support GraphAudit's logging and label propagation mechanisms. Consequently, the data types used by the MLDM algorithm have a direct impact on the memory consumed by the GraphAudit.

Table 1: Data and accesses information for Pagerank (PR), Connected Components (CC) and Alternating Least Squqares (ALS)

	Data Type		Edges Used		#Accesses/Iteration		
	Vertex	Edge	Gather	Scatter	Gather	Apply	Scatter
PR	double	-	In	Out	1	2	0
CC	double	-	-	All	0	2	3
ALS	int, vector, double	double int	All	All	3	6	5

Number of Data Accesses. Each access to an edge or vertex data field requires GraphAudit to log the access and perform label propagation. Thus, the performance of GraphAudit explicitly depends on the number of accesses made in each phase of the vertex program.

Edges Used. Gather and scatter phases of the vertex program access data stored on the edge or the adjacent vertices of the edge. Thus, the number of edges that are used by the MLDM algorithm directly impacts the GraphAudit overhead.

Finally, most MLDM algorithms iteratively update their state until they converge to slow low error or reach a predefined number of iterations. This convergence is tightly related with the structure of the input graph, and although GraphAudit does not modify the termination criteria of the algorithm, each additional iteration includes the above described overheads.

Table 1 shows that the three algorithms store different data field types, provide different edge usage patterns for the gather and scatter functions, and make varying number of data accesses in each phase of the vertex program. This enables us to characterize the performance of GraphAudit for MLDM algorithms with varying properties.

5.2 Experimental Setup

Dataset for graph analytics. We evaluate Pagerank and Connected Componenets using large social network graphs. Using the Boost Graph Library [10], we generate 10 synthetic small world graphs that have the characteristics of a social network graph structure, with number of users (vertices) ranging from 1 to 10 million in steps of 1 million. Based on the small-world model, each graph contains approximately 10 times more edges than vertices.

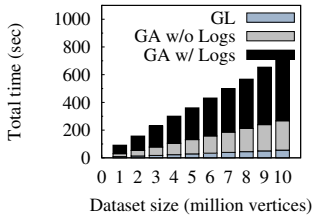
Dataset for collaborative filtering. We use the MovieLens dataset [8], which contains 10M ratings given by 71k users to a set of 10k movies. This dataset is represented as a bi-partite graph, with vertices representing users and movies, and connecting edges that represent the ratings that users assign to movies. We sub-sampled this graph and created 10 graphs, ranging from 1M to 10M ratings in steps of 1 million.

We deploy GraphAudit and GraphLab in two different settings:

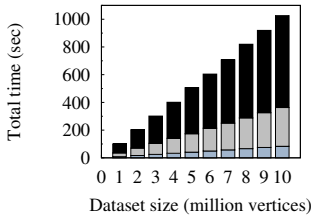
Multi-core server. We use a 32-core 2.6GHz AMD Opteron server with 256GB of RAM, and physically attached 1.7TB disk.

Amazon EC2 Cluster. We configured an 8-node Amazon EC2 cluster. Each node is a quad core 2.6GHz Intel Xeon E5-2670 machine with 15GB RAM and network attached storage (Amazon EBS).

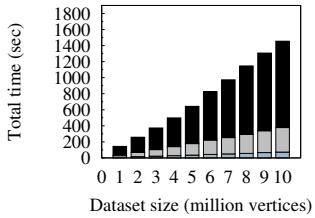
Metrics. For each experiment, we measure the execution time of the algorithm for the following three settings: (1) GraphLab version 2.1 with no modifications (*GraphLab*), (2) GraphAudit with only data access and label propagation with logging disabled (*GraphAudit without logging*), which represents the processing overhead of GraphAudit, and (3) GraphAudit with data access, label propa-



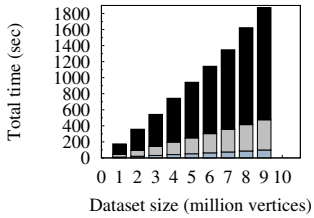
(a) Pagerank EC2



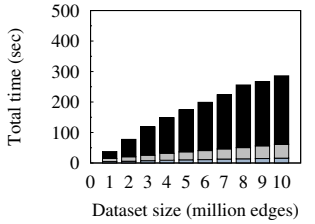
(b) Pagerank Server



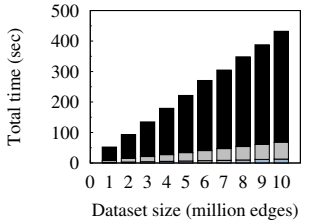
(c) Connected Components EC2



(d) Connected Components Server

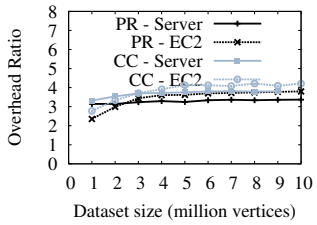


(e) ALS EC2

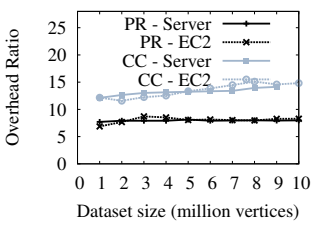


(f) ALS Server

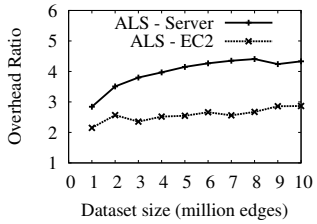
Figure 4: Performance for Pagerank, Connected Component and ALS, on an EC2 cluster and a Server, with various dataset sizes.



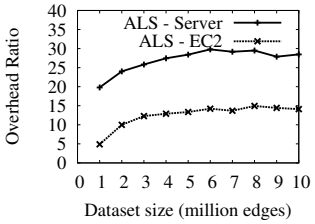
(a) Overhead ratio without logging



(b) Overhead ratio with logging



(c) Overhead ratio without logging



(d) Overhead ratio with logs

Figure 5: Overhead ratio between GraphLab and GraphAudit with an without logging for Pagerank, Connected Componenet and ALS.

gation and logging enabled (*GraphAudit with logging*), which accounts for the processing as well as I/O overhead of GraphAudit. We define the *overhead ratio* as the ratio between the algorithm execution time with GraphAudit and GraphLab without GraphAudit.

We set the following parameters: Pagerank – convergence tolerance is 0.01; ALS – user and item profiles dimension (i.e., the number of latent factors) is 10, the maximal number of updates (signals) per vertex is 5, convergence tolerance is 0.001.

We run the three algorithms using the corresponding datasets on the EC2 cluster and the server. We report the average execution time over 5 runs to account for any variability. We verified that the variance of all runs was less than 5% of the average.

5.3 Varying Graph Size

We characterize the performance of GraphAudit for increasing graph data sizes for the EC2 cluster, using all eight machines, and the multi-core server.

Figure 4 shows the breakdown of the execution time taken by GraphLab, GraphAudit without logs, and GraphAudit with logs. Note that GraphAudit requires the logs for policy compliance checks; we study the execution without logs to better understand the source of GraphAudit's performance overhead. We observe that across all algorithms, the I/O overhead associated with writing logs to disk dominates the execution time. The I/O overhead is higher for the server because writes are bottlenecked by a single disk physically attached to the server. However, we observe that the processing overheads of GraphAudit (GraphAudit without logs) range between 3-4.5x across all algorithms for both deployments. In the following section we show how we can significantly reduce the I/O overhead by a modest increase in the number of machines in the cluster.

Pagerank and Connected Components. Figures 5(a) and 5(b) show the overhead ratios for Pagerank and Connected Components with and without logs, respectively[4]. The overhead ratios do not vary with the input graph size. This shows that both the processing and I/O components of GraphAudit provide scaling properties similar to GraphLab.

The processing overhead of GraphAudit is comparable for the two algorithms (∼4x for 10M vertices) which is approximately 50% of the overhead for Pagerank with logs (8x for 10M vertices) and 30% of the overhead for Connected Component with logs (15x for 10M vertices). The higher overhead of Connected Component is because the scatter function signals *all* edges of a vertex whereas Pagerank signals only the out edges. This causes the Connected Component vertex program to execute over significantly more vertices compared to Pagerank causing additional processing and I/O overheads.

ALS. Figures 5(c) and 5(d) show the scaling properties for ALS as the number of edges are increased from 1 to 10 million. Beyond 5 million edges, ALS exhibits the same scaling behavior as Pagerank and Connected Components, i.e., the overhead ratio does not change with increasing number of edges. However, from 1 to 5 million edges, the overhead associated the GraphAudit with logs increases significantly (4-12x on EC2 and 20-27x on the server)[5]. This increase is explained by the following two properties. First, the apply phase of ALS makes the highest number of data accesses

[4]Connected Components for the 10M input graph size on our server did not complete due to a memory limit, hence not shown in the figure.

[5]The increase in the overhead ratio for GraphAudit without logging is not as steep as GraphAudit with logging.

Table 2: The reduction in processing and logging overhead achieved by increasing cluster size.

	Processing overhead		Logging Overhead	
Cluster	1	8	1	8
ALS	5.79	2.36	30.21	3.41
CC	7.61	3.37	26.77	9.13
PR	7.03	2.6	10.41	4.9

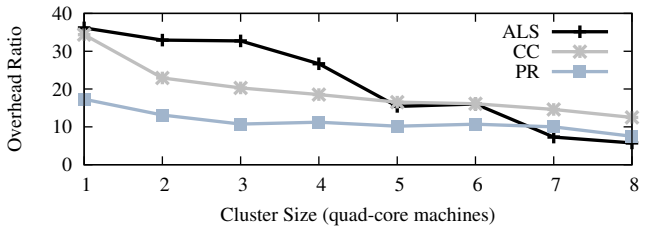


Figure 6: GraphAudit Overhead for Pagerank, Connected Component and ALS, with a constant dataset size but varying Cluster sizes.

due to the dimensions of the latent factors (see Table 1). This makes the apply phase the dominant source of the processing and I/O overheads of GraphAudit. However, beyond the first 5 million ratings, only 1.47% new vertices (movies) are added to the graph, causing the overheads associated with the apply phase to remain roughly the same. Second, for graphs smaller than 5 million edges (ratings) the algorithm converges before the maximum number of iterations. Beyond 5 million ratings, the denser graphs cause the algorithm to make the same number of iterations, stopping only when reaching the maximum allowed iterations.

5.4 Varying Cluster Size

In this section we evaluate the extent to which the I/O overhead of GraphAudit can be reduced by amortizing the writing of logs across multiple machines. To evaluate this overhead, we maintain fixed graph size and increase the number of machines in the Amazon EC2 cluster from 1 to 8. For Pagerank and Connected Components, we use a graph with 1M vertices and 10M edges, and for ALS we use 1M edges and 16K vertices.

Figure 6 shows the reduction in the overhead ratio between GraphAudit with logs and GraphLab for increasing number of machines in the EC2 cluster across the three MLDM algorithms. Table 2 shows the breakdown of the processing overhead (*GraphAudit without logs*) and logging overhead (*GraphAudit with logs - GraphAudit without logs*) for cluster size of 1 and 8 machines. We observe that the relative improvement in performance across the three algorithms is ordered by the extent of I/O processing involved in each algorithm. The reduction of I/O overhead is the greatest for ALS (30.21x to 3.41x with a speedup of 8.8x), followed by Connected Components (speedup pf 2.9x) and Pagerank (speedup of 2.1x). The improvement in processing overheads are modest and range between 2.2-2.7x for the three algorithms.

5.5 Performance Evaluation Summary

The above described experiments highlight several properties of the performance of GraphAudit. First, we observe the factors that impact impact the performance of GraphAudit are: 1) the data types used by the MLDM algorithm, 2) the scope of edges used, and 3) the number of data accesses performed by the algorithm. MLDM algorithm developers should account for these factors when building new algorithms for GraphAudit. Second, both the processing and I/O component of GraphAudit incur a constant overhead over GraphLab, scaling similarly to GraphLab for increasing graph size. The I/O overhead associated with writing logs is the dominant source of overhead, whereas the processing overhead associated with runtime data tracking is relatively low (3-4.5x). Finally, the I/O overhead can be significantly reduced by amortizing the log I/O across a modest size compute cluster. This makes the overall overhead of GraphAudit manageable and feasible for deployment in practice.

6. LOG ANALYSIS CASE STUDY

We present a case study of using GraphAudit to audit policy compliance for a real social networking service, and evaluate the ability to detect policy violations and characterize their extent. Our analysis shows that GraphAudit's log analysis scales to log sizes generated from real world social network graphs. Moreover, GraphAudit's ability to map data flows enables the auditor to go beyond detection of violation, and perform a detailed characterization of the *extent and impact* of the policy violation.

Dataset. We use anonymized data from a real-world social networking service, Pokec [42], which is a popular social networking service in Slovakia. The dataset contains 1.6 million users (vertices) and 30 million directed friendship relationships (directed edges). The diameter (longest shortest path) is 11, which means that when data associated with the vertex propagates 11 hops or more, it essentially can reach all vertices in the graph.

Policy violation setup. We setup the following scenario to emulate a policy violation. The social network service provider uses the social graph for targeted advertising. This is achieved by running Pagerank to identify influential users in the directed graph. Users are provided with privacy controls enabling them to explicitly opt-out of targeted advertising. We consider two users - the most connected user and a user with the median degree. The service provider violates the policy by running Pagerank that uses the complete social graph while ignoring those users' privacy settings.

GraphAudit Log analysis for policy compliance. Executing Pagerank on the dataset results in a total of 800 million log lines. For the above scenario, policy compliance can be audited directly from logs by simply checking if the vertices associated with those users that have opted-out exist in the graph access logs or in the data access logs. Since each log line is self contained, a single appearance of a non-compliant vertex already indicates a policy violation.

Data flow graph. We generate the data flow graph from the data access logs. Since no data field is accessed by the Pagerank algorithm, the data flow graph maps the flow of a vertex's rank. The resultant data flow graph has 18 million nodes and 436 million edges. Of the 18 million vertices, 16 million are replicas generated to prevent false flows, resulting in an average replication rate of 10.22 replicas per node.

Extent of violation. Beyond simply checking for the existence of a violation, GraphAudit's data flow graph can provide a detailed characterization of the *extent* of the violation by empirically measuring the number of users impacted by the policy violation. For the above scenario, the auditor can measure the number of users (vertices in the graph) whose rank computation was influenced by the 2 users that opted-out and whose policy was violated.

In the Pokec data, the most connected user had an out-degree of 8763, and the user with the median degree had an out-degree of 11. Using the generated data flow graph we found that 88% of the vertices (users) in the graph had their rank computation impacted by the rank of the highest-degree vertex. We also observed that the median-degree user influenced 85% of all other vertices in the

graph. The similarity in the extent of spread is a result of the small-world structure of the Pokec social network graph: low-degree vertices are mostly connected to high-degree vertices, which causes information to quickly disseminate in the graph. This analysis implies that for this social network, a non-compliant algorithm can cause data that a user deems as private to potentially reach almost all other users in the network, even for someone with few friends.

7. GRAPHAUDIT DEPLOYMENT

GraphAudit can be deployed under three different scenarios that we outline below. The first two deployment scenarios enable continuous policy compliance checks while the last scenario is based on compliance checks done at the time of audit.

Live deployment with non-interactive responses. Oftentimes, MLDM algorithms operating on massive datasets are run on a periodic basis. For example, a movie recommendation service may not require updating its learned collaborative filtering model on every rating submitted by the users, but instead recompute it once a week, when sufficiently new ratings were added. In this scenario, GraphAudit can be deployed on the existing compute resources of the service provider, while the modest overhead it introduces does not alter the way the service provider runs its algorithms.

Live deployment with interactive responses. For scenarios that require interactive latencies, e.g., executing Pagerank on the social network graph in response to a user query, GraphAudit can be deployed by increasing the number of machines in the cluster to mask the compute and I/O overheads associated with it.

On demand deployment to facilitate an audit. Finally, GraphAudit can also be deployed on-demand during an audit. In this case, existing MLDM algorithms are modified and executed separately on GraphAudit. The resultant logs are then provided to the auditor for policy compliance checks.

8. RELATED WORK

Auditing. Audits are routinely conducted (automated and manual) to test compliance of an organization's *security* policies. However, auditing the compliance of privacy policies today relies predominantly on manual labor, code inspections and reviews of data storage and usage. There has been little previous work done on automating this process [11, 14, 26, 40], and as far as we know, GraphAudit is the first work that addresses auditing MLDM algorithms in a modern, large-scale graph-mining framework.

Database Provenance and Access Control. The most similar works to our paper study data provenance in the context of databases [13, 15, 16, 21, 22, 27, 43, 44] and meta-data in storage systems [33]. In these works, data stored in the database is associated with privacy policies stating who can access it, for what purpose (e.g., which queries can be run), and when it expires. Although these works can support auditing the data acquisition and distribution performed by the service provider, they operate at the database transaction level, making them prohibitively slow for auditing the runtime context of complicated MLDM algorithms operative on massive data. Furthermore, using the data flow graph, GraphAudit enables far more expressive policies than queries.

Privacy preserving data mining. A different line of work aim at reducing the probability of private data violations, through input or output distortion, such as differential privacy techniques [24, 28, 29], cryptographic methods that enable computation on encrypted data [35, 37], and end-user tools that improve transparency of data usage [17, 19]. Some of these techniques, such as Airavat [38], were applied to a large-scale data-mining platform. These approaches are orthogonal to GraphAudit, and even if these are made practical and deployed, auditing will still be required to ensure that service provider's data use practices are policy compliant.

Information flow analysis. Runtime taint tracking is a key component in GraphAudit, enabling the analysis of data accesses and propagation during the runtime of the MLDM algorithm. Related efforts like [45, 46] seek to protect against malicious data accesses and [18, 34, 36] seek to protect the systems integrity. While these systems provide runtime enforcement, GraphAudit only provides detailed logs for post-analysis. Hence, GraphAudit's logging does not impact the runtime behavior of the MLDM framework. Moreover, GraphAudit logs all implicit flows for the purpose of auditing and does not require precise taint propagation that can significanlty increase the runtime overheads.

9. CONCLUSION

In this paper we present GraphAudit, a policy compliance auditing framework for large-scale graph-based machine learning platforms. GraphAudit mitigates the need for the traditional manual process of auditing machine learning algorithms that process private user data. It does so by tracking and logging the data use and propagation of potentially private data at runtime. These logs can then be analyzed to detect privacy violations, and furthermore, build a complete data flow graph that can assist an auditor in characterizing the extent of the policy violation and its potential harm. We build GraphAudit on top of GraphLab, and show that it scales well with the size of the input graph. We also demonstrate how GraphAudit's overhead can be significantly reduced into a practical deployment by using a modest cluster.

We believe that GraphAudit can fundamentally transform how privacy auditing is perceived and performed by companies – from being a slow, expensive and tedious manual process into an efficient and scalable process, that is an integral part of the deployment of machine-learning algorithms. As future work, we plan to implement GraphAudit on other popular machine-learning platforms and integrate its logging mechanism with existing large-scale logging infrastructures.

10. REFERENCES

[1] Apache Giraph. `https://giraph.apache.org/`.
[2] Apache mahout. `http://mahout.apache.org`.
[3] Boost C++ Libraries. `http://www.boost.org/doc/libs/`.
[4] European interactive digital advertising alliance. `http://www.edaa.eu`.
[5] FTC Facebook Complaint. `http://www.ftc.gov/sites/default/files/documents/cases/2011/11/111129facebookcmpt.pdf`.
[6] Generally accepted privacy pinciples. `http://www.aicpa.org`.
[7] Google Facebook Complaint. `http://www.ftc.gov/sites/default/files/documents/cases/2011/10/111024googlebuzzcmpt.pdf`.
[8] MovieLens Dataset. `http://grouplens.org/datasets/movielens/`.
[9] Privacy audits. `http://peterfleischer.blogspot.com/2010/03/privacy-audits.html`, year = 2010.
[10] The Boost Graph Library. `http://www.boost.org/doc/libs/1_55_0/libs/graph/doc/index.html`.
[11] Truste data privacy management platform. `http://www.truste.com/about-TRUSTe/`.
[12] Personal information protection act. `http://servicealberta.ca/pipa`, 2004.

[13] AGARWAL, R., KEIRNAN, J., SRIKANT, R., AND XU, Y. Hippocratic databases. In *Proc. of VLDB* (2002).

[14] BAKIS, B., AND MCEWEN, J. How to conduct a privacy audit. `http://http://www.mitre.org/sites/default/files/pdfHowToConductPrivacyAudit.pdf`, 2007.

[15] BENJELLOUN, O., SARMA, A. D., HALEVY, A., AND WIDOM, J. Uldbs: Databases with uncertainty and lineage. In *Proceedings of the 32Nd International Conference on Very Large Data Bases* (2006), VLDB '06, VLDB Endowment, pp. 953–964.

[16] BUNEMAN, P., CHAPMAN, A., AND CHENEY, J. Provenance management in curated databases. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2006), SIGMOD '06, ACM, pp. 539–550.

[17] CHAKRABORTY, S., SHEN, C., RAGHAVAN, K., MILLAR, M., AND SRIVASTAVA, M. ipShield: A Framework for Enforcing Context-Aware Privacy. In *Usenix NSDI* (2014).

[18] CLAUSE, J., LI, W., AND ORSO, A. Dytan: A Generic Dynamic Taint Analysis Framework. In *Proceedings of the 2007 international symposium on Software testing and analysis* (2007), ACM New York, NY, USA, pp. 196–206.

[19] CONSOLVO, S., JUNG, J., MAGANIS, G., POWLEDGE, P., GREENSTEIN, B., AND AVRAHAMI, D. The Wi-Fi Privacy Ticker: Improving Awareness & Control of Personal Information Exposure on Wi-Fi.

[20] CORTEZ, P., AND HAY, D. Privacy disclosure and auditing: An exploratory study.

[21] CUI, Y., AND WIDOM, J. Lineage tracing for general data warehouse transformations. *The VLDB JournalâĂŤThe International Journal on Very Large Data Bases 12*, 1 (2003), 41–58.

[22] CUI, Y., WIDOM, J., AND WIENER, J. L. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems (TODS) 25*, 2 (2000), 179–227.

[23] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM 51*, 1 (2008), 107–113.

[24] DINI, G., AND PERAZZO, P. Uniform obfuscation for location privacy. In *in ACM DBSec* (2012).

[25] GONZALEZ, J. E., LOW, Y., GU, H., BICKSON, D., AND GUESTRIN, C. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2012), pp. 17–30.

[26] GOVERNMENT OF ALBERTA, CANADA. Conducting a privacy audit (based on pipa). `http://servicealberta.ca/pipa/documents/PrivacyAudit.pdf`, May 2010.

[27] GREEN, T. J., KARVOUNARAKIS, G., AND TANNEN, V. Provenance semirings. In *Proceedings of the Twenty-sixth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (New York, NY, USA, 2007), PODS '07, ACM, pp. 31–40.

[28] HE, X., ZHANG, X., AND KUO, C. A distortion-based approach to privacy-preserving metering in smart grids. *IEEE Access Journal* (2013).

[29] JIANG, T., WANG, H., AND HU, Y. Preserving location privacy in wireless lans. In *Proc. of Mobisys* (2007).

[30] KOREN, Y., BELL, R., AND VOLINSKY, C. Matrix factorization techniques for recommender systems. *Computer* (2009).

[31] LOW, Y., GONZALEZ, J., KYROLA, A., BICKSON, D., GUESTRIN, C., AND HELLERSTEIN, J. M. GraphLab: A New Parallel Framework for Machine Learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)* (2010).

[32] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2010), SIGMOD '10, ACM, pp. 135–146.

[33] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. I. Provenance-aware storage systems. In *USENIX Annual Technical Conference, General Track* (2006), pp. 43–56.

[34] NEWSOME, J., AND SONG, D. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the 12th Network and Distributed System Security Symposium (NDSS)* (2005).

[35] NIKOLAENKO, V., IOANNIDIS, S., WEINSBERG, U., JOYE, M., TAFT, N., AND BONEH, D. Privacy preserving matrix factorization. In *ACM Conference on Computer and Communications Security (CCS)* (2013).

[36] QIN, F., WANG, C., LI, Z., SEOP KIM, H., ZHOU, Y., AND WU, Y. LIFT: A Low-Overhead Practical Information Flow Tracking System for Detecting Security Attacks. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (2006), IEEE Computer Society Washington, DC, USA, pp. 135–148.

[37] RAKESH, A., AND KIERNAN, J. Order preserving encryption for numerica data. In *ACM SIGMOD* (June 2004).

[38] ROY, I., SETTY, S., KILZER, A., SHMATIKOV, V., AND WITCHEL, E. Airavat: Security and privacy for mapreduce. In *Usenix NSDI* (2010).

[39] SEN, S., GUHA, S., DATTA, A., RAJAMANI, S. K., TSAI, J., AND WING, J. M. Bootstrapping privacy compliance in big data systems. In *Proceedings of the 35th IEEE Symposium on Security & Privacy (Oakland)* (2014).

[40] SINGLETON, T. IT and Privacy Audits. *ISACA Journal* (2009).

[41] SWEENEY, L. Computational disclosure control: A primer on data privacy protection. Massachusetts Institute of Technology.

[42] TAKAC, L., AND ZABOVSKY, M. Data analysis in public social networks. In *IntâĂŹl. Scientific Conf. & IntâĂŹl Workshop Present Day Trends of Innovations* (2012).

[43] WANG, Y. R., MADNICK, S. E., ET AL. A polygen model for heterogeneous database systems: The source tagging perspective. In *VLDB* (1990), vol. 90, pp. 519–538.

[44] WIDOM, J. Trio: A system for integrated management of data, accuracy, and lineage. *Technical Report* (2004).

[45] YIN, H., SONG, D., EGELE, M., KRUEGEL, C., AND KIRDA, E. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *Proceedings of the 14th ACM conference on Computer and Communications Security* (2007), pp. 116–127.

[46] ZHU, D., JUNG, J., SUNG, D., KOHNO, T., AND WETHERALL, D. Privacy Scope: A Precise Information Flow Tracking System For Finding Application Leaks. Tech. Rep. EECS-2009-145, Department of Computer Science, UC Berkeley, 2009.

APPENDIX

Appendix - Pagerank Code With and Without GraphAudit

Listing 1 and 2 shows simplified C++ code snippets with and without GraphAudit modifications, respectively. The data associated with each vertex is an `int`. In the original GraphLab code, this data is returned by reference with a call to `vertex.data()`, thus is used both for reading and for writing the updated value. In Listing 2, GraphAudit replaces this with explicit calls to `get_data()` and `set_data()` to a field we call "`rank`". In addition, we added an edge precondition function, i.e., `gather_verify()`, that returns `true` only if edge is not "confidential" and can be used.

```
  /* Over all IN_EDGES, Gather the weighted rank of
  the adjacent vertex  */
  double gather(icontext_type& context, const
  vertex_type& vertex, edge_type& edge) const {
    return (edge.source().data() / edge.source().
    num_out_edges());
  }
  /* Use the total rank of adjacent pages to update
  this vertex */
  void apply(icontext_type& context, vertex_type&
  vertex, const gather_type& total) {
    const double newval = compute_rank(total);
    last_change = (newval - vertex.data());
    vertex.data() = newval;
  }
  /* The scatter function just signals adjacent
  vertices */
  void scatter(icontext_type& context, const
  vertex_type& vertex, edge_type& edge) const {
      if(last_change > TOLERANCE)
                context.signal(edge.target());
  }
```

Listing 1: Simplified Code snippet for Pagerank - GraphLab

```
  /* Verify Edge */
  bool gather_verify(edge_type& edge) const {
        return !confidential(edge);
  }
  /* Over all IN_EDGES, Gather the weighted rank of
  the adjacent vertex  */
  double gather(icontext_type& context, const
  vertex_type& vertex, edge_type& edge) const {
    return (edge.source().get_data("rank") / edge.
    source().num_out_edges());
  }
  /* Use the total rank of adjacent pages to update
  this vertex */
  void apply(icontext_type& context, vertex_type&
  vertex, const gather_type& total) {
    const double newval = compute_rank(total);
    last_change = (newval - vertex.get_data("rank"))
    ;
    vertex.set_data("rank",newval);
  }
  /* The scatter function just signals adjacent
  vertices */
  void scatter(icontext_type& context, const
  vertex_type& vertex, edge_type& edge) const {
      if(last_change > TOLERANCE)
                context.signal(edge.target());
  }
}
```

Listing 2: Simplified Code snippet for Pagerank - GraphAudit