An Enforcement Model for Preventing Inference Attacks in Social Computing Platforms

Seyed Hossein Ahmadinejad
Nulli
shan@nulli.com

Philip W. L. Fong
University of Calgary
pwlfong@ucalgary.ca

ABSTRACT

Social Network Systems (SNSs) allow third-party extensions to access user profiles by providing an Application Programming Interface (API). It has been demonstrated in the literature that this API can be exploited by malicious extensions to infer users' sensitive information from the information that is accessible through the API. To prevent this type of privacy violation, we propose a view-based protection model, in which a sanitizing transformation, called a view, is applied to the user profile when it is queried by a third-party extension. We demonstrate empirically that such a protection mechanism effectively reduces the statistical correlation between sensitive and accessible information. We also propose an optimization in which the materialization of views is performed lazily: rather than sanitizing the entire profile during a query, only the parts of the profile that are visible to the query are transformed. We demonstrate empirically that this optimization offers visible performance advantage, and propose a programming language-independent, probabilistic automata model for encoding such transformations.

CCS CONCEPTS

• **Security and privacy → Social network security and privacy**;

KEYWORDS

Social network systems, Facebook applications, privacy, sanitizing transformations, view-based protection, statistical correlation, optimization, tree transducers

ACM Reference format:
Seyed Hossein Ahmadinejad and Philip W. L. Fong. 2017. An Enforcement Model for Preventing Inference Attacks in Social Computing Platforms. In *Proceedings of SACMAT'17, Indianapolis, IN, USA, June 21-23, 2017,* 12 pages.
https://doi.org/http://dx.doi.org/10.1145/3078861.3078867

1 INTRODUCTION

Privacy is a serious concern in information systems that store sensitive, personal information. This concern has elevated in recent years by the pervasive adoption of SNSs. Users willingly and often indiscriminately upload personal information to their social profiles. It has been shown that the information in social profiles can be exploited for malicious purposes [2]. For example, consider the commonplace mechanism of asking security questions during an authentication session. "Who is your youngest sibling?" "Who is your favorite author?" When the primary authentication mechanism fails to verify an identity claim, these security questions become a fallback, and access will be granted to an account (e.g., bank, email, etc) if the security questions are correctly answered. The answers to security questions can often be gathered from a user's social profile. This means the social profile of a user contains information that, if properly harvested, helps a malicious party to launch an identity theft attack [3, 4].

There are at least two approaches by which the aforementioned identity theft attack can be launched. The first is by crawling the publicly accessible region of a social network, and collecting publicly available information of user profiles. Intelligent data mining techniques can be deployed to the resulting dataset for harvesting answers to typical security questions [6, 7, 12–15, 19, 20]. A second approach is to hide information harvesting logic in third-party extensions to SNS platforms. The present work targets this latter approach.

In 2006, Facebook extended its business model to provide further functionalities for its users in the form of applications, games, etc. Facebook recognized that its core business is constructing and maintaining the social graph of users rather than developing an endless array of social applications. The company therefore turned Facebook into an extensible platform. A platform API was released, allowing third-party developers to augment the feature set of Facebook through the development of software extensions called Facebook applications. This API allows Facebook applications to access user information in profiles. Intended for protecting user privacy, a permission-based access control model is imposed. A Facebook application can only access a profile attribute if the user grants the corresponding permission to the application.

This protection model, however, is ineffective in protecting user privacy against malicious applications. Suppose a user considers her birthday to be sensitive, and denies an untrusted application of the permission to access this attribute. The user also recognizes that the application needs access to its timeline, so that permission is granted. Now the application can use the platform API to scan through the timeline of the user, identifying a day when birthday greetings ("Happy birthday!") are sent to the user, and inferring that that day is likely the birthday of the user. The privacy preference of the user is therefore violated. Previous work has termed this the ***SNS API Inference Attack***, and has demonstrated that such attacks can be launched through the Facebook API with alarming accuracy [2–4]. As some of the Facebook applications have millions of active users, even a modest success rate would translate to a large

SACMAT'17, June 21-23, 2017, Indianapolis, IN, USA

ACM ISBN 978-1-4503-4702-0/17/06…$15.00
https://doi.org/http://dx.doi.org/10.1145/3078861.3078867

number of victims. Besides, Facebook applications may be granted permissions to access profile attributes that are not even visible through a public crawling of the social network, thus there are more channels for information leakage. Lastly, Facebook applications run on third-party servers, meaning that Facebook does not have the means to verify if the third-party applications are benign.

1.1 View-based Protection

The discussion above shows that controlling access is insufficient for preventing SNS API inference attacks. The reason is that there exists statistical correlation between sensitive information (which the user attempts to hide) and accessible information (which the user allows access). A malicious third-party application can exploit this correlation to infer sensitive information from the information that is legitimately accessible under the access control model. The key to protection is thus the breaking of correlation rather than simply denying access to sensitive information.

We advocate in this work a ***view-based protection model***. Under this model, when a third-party application A queries the profile P of user u through the API, the query Q is not evaluated against P itself. Instead, P first undergoes a sanitizing transformation T, before Q is evaluated against the sanitized profile $T(P)$. The transformation T is called a ***view***, which is specified by the user and/or the platform. T is thus an enforcement-layer privacy policy. A view may eliminate certain attributes (access control), or probabilistically transform the profile with the aim of perturbing the statistical correlation between sensitive and accessible information. In other words, view-based protection subsumes access control.

One of the contributions of this paper is to report first empirical evidence that view-based protection, implemented via query-time sanitizing transformations, can effectively reduce the success rate of SNS API inference attacks.

The mathematical formulation of privacy and utility goals, the proof method for establishing that a given view satisfies the two goals, as well as the mathematical trade-off between privacy and utility, are the topics of a related work [5], and are outside the scope of this paper.

1.2 The Challenge of View Materialization

How shall one implement view-based protection in an efficient manner? There are at least two possible approaches:

1. A naive approach is to compute $T(P)$ every time P is queried. The problem is that P can be large (imagine everything in one's timeline, photo albums, etc.), thereby causing even the most innocent query Q to be penalized in performance.
2. Another approach is to have the SNS store both P and $T(P)$. The problem is that $T(P)$ will have to be recomputed every time P is updated (which happens frequently). Not only that, T is specific to the user u and the application A, meaning that the SNS needs to store a $T(P)$ for every application A that user u subscribes to — a space inefficient option.

We propose a middle way in this work. The computation of $T(P)$ is called the ***materialization*** of view T. We argue that materialization should be performed in a lazy manner, at the time of query. To see this, the query Q may not access all components of profile P. Instead of eagerly applying T to the entire profile P, we apply T to the parts of P that are visible to query Q. A simple query that involves only a small fragment of profile P will therefore incur only a meager amount of materialization, thereby preventing the performance penalty of Approach 1 above. As T is computed at the time of query, there is no need to maintain multiple materialized views, thereby preventing the view maintenance problem of Approach 2.

1.3 Contributions

This paper has three contributions:

1. **Empirical evidence for effectiveness of view-based protection.** We offer first evidence that view-based protection can effectively protect the privacy of sensitive information in user profiles (§3). Specifically, we implemented view-based protection mechanisms, based on handcrafted sanitizing transformations, against the 8 sample inference algorithms of Ahmadinejad *et al.* [3, 4], and demonstrated that the implemented views effectively reduce the correlation between accessible and sensitive information, thereby significantly lowering the accuracy of the inference algorithms.
2. **Empirical evidence for the advantage of eschewing view materialization.** We empirically demonstrate the performance advantage for avoiding the eager materialization of views (§4). Specifically, we implemented the aforementioned view-based protection mechanisms in Haskell, a programming language equipped with lazy evaluation. The latter feature was employed to simulate the effect of lazy materialization of views. Empirical data suggests that a view-based protection mechanism shall materialize views lazily.
3. **Language-independent enforcement mechanism with lazy materialization of views.** We propose a programming language-independent enforcement mechanism that materializes a view in a lazy manner (§6). Specifically, we encode a sanitizing transformation (aka a view) using a Probabilistic Multi Bottom-Up Tree Transducer (PMBUTT). As we model a user profile as a tree (e.g., an XML document), a tree transducer is an automata-based model for specifying a sanitization transformation of user profiles. The particular model that we propose, PMBUTT, is an extension of tree transducer models in existing literature. While supporting probabilistic sanitization (important for implementing, say, noise addition), PMBUTT has the theoretical property of closure under composition: there exists an algorithm for composing two sanitizing transformations to obtain a transformation that is also a PMBUTT. Composeability has two practical applications. First, it allows a policy engineer to build a complex view out of simpler views. Secondly, and more importantly, composeability is the key to lazy materialization of views. It turns out that an access query against a view can also be encoded as a PMBUTT. Suppose the query is specified as a PMBUTT Q, and the view is specified as a PMBUTT T. Then evaluating the query Q against a profile P protected by view T is equivalent to computing $Q \circ T(P)$. The composition PMBUTT $Q \circ T$ does not apply the sanitization of T to the entire tree P, but only the parts of P that query Q examines. This is precisely the idea of lazy materialization of views.

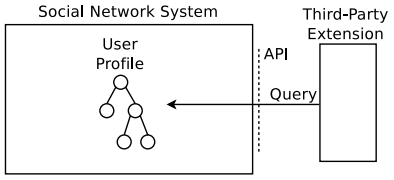

Figure 1: Original Design of Facebook Applications

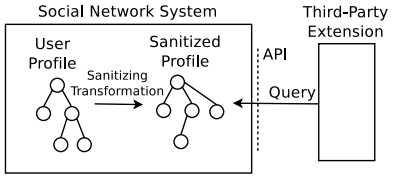

Figure 2: View-based Protection

2 SYSTEM OVERVIEW

2.1 Facebook Applications

An SNS such as Facebook maintains a social profile for each user (Fig. 1). A user may subscribe to third-party extensions (aka Facebook applications). These third-party extensions deliver their functionalities by querying personal information of the user through the SNS platform API. The queries are evaluated against the profile of the user.

A permission-based access control model determines if query evaluation is allowed or not, based on what permissions the owner of the profile has granted to the application. In other words, users declare their privacy preferences through the granting of permissions. As we have explained, access control is insufficient for preventing malicious third-party extensions from inferring sensitive information, simply because it fails to break the statistical correlation between sensitive and accessible information.

2.2 Query-time Profile Sanitization

In view-based protection, queries issued by a third-party application is evaluated against a sanitized version of the user profile (Fig. 2). The sanitized profile is constructed by a probabilistic transformation that is called a ***view***. A view may (a) remove certain sensitive information, or (b) probabilistically transform the profile in such a way that any correlation between sensitive and accessible information is reduced or eliminated.

Since user profiles are semi-structured data, we model a user profile as a tree-shaped data structure (e.g., XML document). A view is therefore a probabilistic tree transformer.

The exact formulation of the view depends on the privacy preference of the user (what to hide) as well as the utility need of the application (what to release). These topics are discussed in details in a related work [5].

The view is materialized (i.e., applied to the user profile) when a query is issued by the application. To reduce computation overhead, view materialization is performed in a lazy manner: only the parts of the profile relevant to the query is transformed. See §4 and §5 respectively for a prototype and an automata model (PMBUTT) for lazy materialization. With lazy materialization, the SNS does not need to maintain a sanitized version of the profile for each application that the user subscribes to. Only a PMBUTT is stored for each user-application pair. *The size of the PMBUTT remains constant even as the user profile grows in size.* This is a major benefit of our approach.

Algorithm	Description
`author`	Infer the user's favorite author to be someone who authored the majority of books in the user's list of favorite books.
`genre`	Infer the user's favorite movie genre to be a genre that accounts for the majority of the movies in the user's list of favorite movies.
`birthday`	Infer the user's birthday to be a day when he/she received a considerable number of birthday wishes on his/her wall.
`sibling`	Infer the user's youngest sibling to be the youngest friend who listed the user in his/her profile as a sibling.
`partner`	Infer the user's partner to be a friend who is tagged in the the majority of the user's photos.
`hometown`	Infer the user's hometown to be a town where the user's primary school is located.
`oldestF`	Infer the user's oldest friend to be a friend who went to the same primary school as the user did.

Table 1: Sample inference algorithms

3 EFFECTIVENESS OF VIEW-BASED PROTECTION

We began our investigation by an experiment conducted to demonstrate the effectiveness of view-based protection in reducing the success rate of SNS API inference algorithms.

3.1 Inference Algorithms and Dataset

In the study reported in [4], 8 sample inference algorithms are executed on the profiles of 424 users. In our experiment, we evaluate the success rate of these same inference algorithms against *profiles that are sanitized by probabilistic tree transformations*, and compare the result with what was reported in [4]. The intention is to demonstrate the effectiveness of query-time sanitization against inference algorithms. We, however, excluded one of those 8 algorithms because it had a very low success rate. Table 1 briefly summarizes the 7 sample inference algorithms that we have adopted from [4].

Since we did not have access to the user profiles collected in [4],[1] we generated a synthetic set of profiles. We generated the same number of profiles, which was 424, as [4] reported. The synthetic profiles are generated in such a way that preserve the success rates of the 7 sample inference algorithms. The following example illustrate how this is done for the inference algorithm `author`.

[1]To protect the privacy of the participants, the experiments of [4] were set up in such a way that the profile data is not stored.

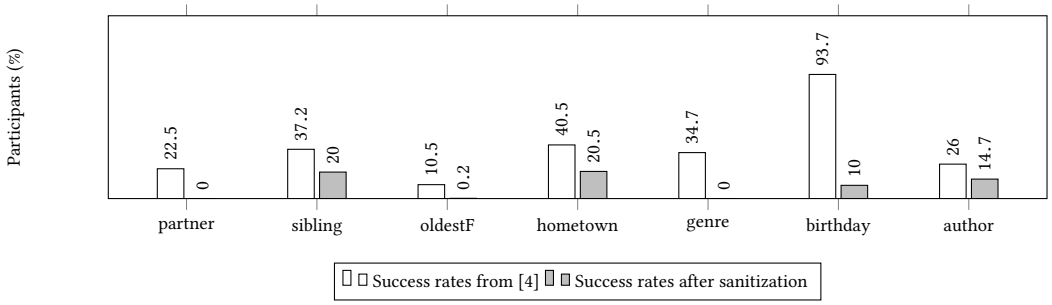

Figure 3: Comparing the success rate of inference algorithms before and after sanitizing profiles

Example 3.1 (Generating Profile Data for `author`*).* One of the inference algorithms of [4] is `author` (Table 1), which infers the user's favorite author to be the peron who has authored the majority of the books on the user's list of favorite books.

To generate a profile against which `author` will succeed in inferring the user's favorite author to be *Mark Twain*, all it takes is to add enough number of books written by *Mark Twain* into the list of favorite books in the user profile. On the contrary, to generate a profile against which `author` will fail, we simply create an empty list of favorite books.

The success rate of `author` was evaluated to 28.3% when it was allowed to make only one guess [4]. Using the method described in the previous paragraph, we thus generated 120 (= 0.283×424) profiles such that `author` would succeed, and 304 (= $424 - 120$) profiles such that `author` would fail.

A similar approach is adopted for replicating the profile distribution for the other inference algorithms.

Once the synthetic dataset of profiles was generated, we implemented the inference algorithms described in [4] and ran them against the profiles. The success rates of algorithms were almost the same as the ones reported in [4], which shows that our simulation process was accurate enough.

3.2 Sanitizing Transformations

To demonstrate the effects of query-time sanitization, we design sanitizing transformations (aka views) that could eliminate the statistical correlation exploited by the inference algorithms reported in [4]. The user profiles are represented as trees, and the sanitizing transformations are programmed as probabilistic tree transformations. Here is an example.

Example 3.2. To counter the inference of the `author` algorithm, we transform user profiles by taking the following steps:

1. Suppose author a is the author who wrote the highest number of books in the user's list of favorite books. Let m be the number of favorite books written by author a.
2. With equal probabilities (0.5 and 0.5), randomly set n to be either $m + 1$ or $m - 1$,
3. Randomly select an author b who does not have any book in the favorite book list.
4. Insert n different book titles written by author b into the list of favorite books.

The effect is that, with probability 0.5, a will cease to be the dominating author in the favorite book list. Note that the transformation is probabilistic. The probabilistic behaviour is important. is important. Even when an attacker who knows how the sanitizing transformation work, it would still be confused because there is an equal chance that the dominating author in the favorite book list may or may not be the favorite author. Had we simply set $n = m + 1$, an attacker who knows how the sanitizing transformation works will conclude that the second dominating author in the favorite book list is the favorite author.

A sanitizing transformation algorithm was designed for each of the 7 sample inference algorithms. Due to space limitation, we refer the reader to §6.4 of [1] for details of the other 6 probabilistic tree transformations.

3.3 Results

Fig. 3 compares the success rate of the sample inference algorithms before and after transforming user profiles. The numbers in Fig. 3 show that sanitizing user profiles by probabilistic tree transformations can significantly reduce the success rate of the inference algorithms.

4 ADVANTAGE OF ESCHEWING VIEW MATERIALIZATION

While view-based protection is shown above to be an effective mitigation mechanism against SNS API inference attacks, a social networking platform will have to deal with the very practical issue of deciding when to materialize a view. We have argued in §1.2 that both the naive approach (sanitizing the entire profile when it is queried) and the stored-view approach (storing the sanitized profile) are inadequate. We claimed in §1.2 that performing lazy materialization at query time offers performance advantage against the naive approach, and prevents the frequent updates and storage overhead of the stored-view approach. In this section, we report an experiment conducted to demonstrate the performance advantage of lazy materialization of views.

4.1 Design

As an exploratory experiment, we are not tied to any particular programming language. We therefore take advantage of the ***lazy evaluation*** feature in the Haskell programming language, which is a purely functional language with mature implementations and

extensive libraries, including those for tree/XML transformation. With lazy evaluation in Haskell, an expression is only evaluated if its value is absolutely needed for subsequent computation. Recall that, lazy materialization of views entails the application of a sanitizing transformation only to the parts of the tree that are relevant to the answering of a query. In this experiment, we model both the view $T(P)$ and the query $Q(P)$ as Haskel functions that take a tree P as argument. The querying of the sanitized tree is therefore the functional composition $Q(T(P))$. Haskell lazy evaluation will attempt to defer the application of T to the branches of P until Q actually asks for them. This is a convenient implementation of lazy view materialization. (In §6, we will examine a language-independent implementation of lazy view materialization via probabilistic tree transducers.)

Suppose *trans* is a sanitizing transformation. For now, assume *trans* involves noise addition designed for countering the `author` inference algorithm. Suppose further that *none* is a trivial query that returns a constant. That is, *none* is a transformation that suppresses the entire input profile.

To assess the effect of lazy view materialization, we evaluate $none(trans(t))$ for every profile t in the synthetic dataset of the previous section. Specifically, we do so in two different experimental configurations. In Configuration 1, we evaluate the expression in the regular Haskell runtime environment, allowing the full effect of lazy evaluation to be felt. Since the transformation *none* does not actually consume the return value of $trans(t)$, it is expected that lazy evaluation will optimize away the overhead of computing $trans(t)$. In Configuration 2, the Haskell runtime environment is forced to strictly evaluate $trans(t)$ first, which results in t', and then evaluate $none(t')$. Note that in Configuration 2, lazy evaluation is not completely turned off. Rather, eager evaluation is only enforced for the specific computation of $none(trans(t))$.

Haskell profiling tools are employed to compare the performance of evaluating the expression in the two configurations. The following metrics are used to compare the performance of view materialization in the two configurations:

1. allocated bytes in the heap,
2. copied bytes during garbage collection,
3. elapsed time to complete the evaluation, and
4. time spent for garbage collection.

4.2 Results

Table 2 compares the two configurations. Lazy evaluation (Configuration 1), which avoids materializing $trans(t)$, produced a significant improvement in performance. In Configuration 2, 82.5% of the total time of the computation were spent on garbage collection compared to only 0.4% in the first configuration. In Configuration 2, the Haskell runtime environment actually evaluated $trans(t)$, and that is why it needed to perform a lot of garbage collection afterwards. The above experiment was repeated by changing the transformation performed by *trans* to other sanitizing transformations that were designed for the remaining 7 sample inference algorithms in the previous section. We observed that the value of the performance metrics remain the same in Configuration 1, whereas they change in the second configuration. In other words, in Configuration 1, it does not matter for the Haskell runtime what transformation is used in *trans* because that transformation does not need to be evaluated at all.

	Configurations	
Metrics	**Config. 1**	**Config. 2**
Bytes allocated in the heap	219 KB	95572 KB
Bytes copied during GC	3 KB	15645 KB
Elapsed time	0.07 S	1.01 S
Time for GC	0.4%	82.6%

Table 2: Performance metrics for the Haskell compiler

4.3 Discussion

The above experiment shows that avoiding view materialization as much as possible can make view-based protection much more efficient than otherwise. In the above experiment, we used Haskell as our programming language simply because of the convenience of built-in lazy evaluation. This may not always be acceptable in an industrial software development project. For instance, Haskell may not be the developer's preferred programming language. In addition, the behaviour of lazy evaluation may not appear predictable for programmers who do not possess deep knowledge of the Haskell programming model. What is needed, therefore, is a programming language-independent enforcement model that supports lazy materialization of views. Such an enforcement model is a main contribution of this work, a topic to which we now turn.

5 TREE TRANSDUCERS

Since we model user profiles as trees, the computation model that is to be proposed must be capable of transforming trees. In the literature, tree transducers have been proposed for exactly this purpose. We therefore adopt the same notion in order to propose the required computation model. This section is both an introduction to tree transducers as well as a requirement analysis for the kind of tree transducers needed by lazy view materialization.

5.1 Representation of Trees

User profiles are represented as trees, which are in turn represented by terms formed by function symbols.

Definition 5.1 (Ranked alphabet). A ranked alphabet is a pair (Σ, rk) where Σ is a finite set of function symbols, and $rk : \Sigma \to \mathbb{N}$ maps a function symbol to its arity.

We write Σ^n for the set of function symbols in Σ with rank n. We also write σ^k to highlight that $\sigma \in \Sigma^k$.

Example 5.2. One of the 8 sample inference algorithms in [4] aims at inferring a user's birthday by (a) scanning the user's wall in his/her profile, and (b) identifying the day in which a large number of birthday greetings ("Happy birthday") were posted.

The user's wall can be modeled using the ranked alphabet $\Sigma = \{en^3, bg^0, nbg^0, ts_1^0, ts_2^0, nil^0\}$. The function symbol en denotes an entry on the wall. It has three input arguments: (i) the timestamp of the entry, corresponding to constant symbols ts_1 or ts_2, (ii) the message of the entry which could be either a birthday greeting

(denoted by constant symbol *bg*) or a message that does not correspond to a birthday greeting (denoted by constant symbol *nbg*), and (iii) the next entry on the wall, which could be either the constant *nil* (terminating the list of entries), or another *en* function.

For instance, $en(ts_1, bg, en(ts_2, nbg, nil))$ represents a wall with two entries. The first entry is a birthday greeting posted at time ts_1. The second entry, posted at time ts_2, is not a birthday greeting (*nbg*).

A tree transducer transforms a tree formed by a ranked alphabet to a sanitized tree of the same alphabet.

5.2 Requirement Analysis

Different types of tree transducers have been proposed in the literature. To identify which kind of tree transducers fits the requirements of view-based protection model, we need to first determine the properties we seek in a tree transducer based on the needs of the view-based protection model. The two main features we need are (a) closure under composition, and (b) high expressive power.

The first property, closure under composition, is required due to the fact that it is usually infeasible to create a single transducer that can completely handle a transformation task. A transformation is typically formed by composing two or more primitive transformations. Consequently, if we model two transformations as tree transducers of a particular class, we want their composition to belong to the same class of tree transducers. More importantly, we will model a profile query also as a tree transformation (i.e., transforming a profile tree to a query answer). By composing a sanitizing transformation T with a query Q, the composed transformation $Q \circ T$ will materialize only the part of T that is relevant to answering Q. Closure under composition is therefore the main vehicle by which lazy materialization is achieved.

The second property, high expressive power, requires the tree transducer to be expressive enough so that it can express commonly used sanitizing transformations, such as suppression, generalization, noise addition, and sampling. Most of the existing classes of tree transducers can perform simple types of transformations (e.g., replacing a subtree with another subtree). However, there are three specific properties that are rarely found simultaneously in existing tree transducer models, namely, look-ahead, differential copying and probabilistic behaviour. We informally define them in the following.

Look-ahead. A tree transducer with the look-ahead feature can look at deeper (one or more levels lower) nodes in a tree while parsing the nodes at upper parts of the tree. Consider the task of protecting the user's birth date in Example 5.2. To protect the user's birth date, we want to replace the timestamp of all the birthday greeting entries with ϵ (a constant symbol representing no information). For such a transformation, we need the look-ahead property. In other words, when the state machine arrives at a state where input is a wall entry, e.g., $en(ts_1, bg, en(ts_2, nbg, nil))$, it needs to be able to look at the function symbol in the second input argument, and if that function symbol is *bg*, then the machine replaces the first argument (the timestamp ts_1) with ϵ. Without look-ahead, the state machine only sees the *en* function symbol, and not its children nodes, thereby failing to respond accordingly.

Differential Copying. Random sampling is a classical sanitizing transformation. Rather than publishing the entire data set, a sanitized sample of it is published. In that way, statistics are preserved without disclosing the entire data set. In the birth date example in Example 5.2, one can sanitize the wall by first removing the timestamps, and then releasing a sample subset of the wall entries.

Our experience with encoding random sampling with tree transducers allows us to discover that we require a tree transducer model that supports a feature that we call *differential copying*.

Example 5.3 (Differential copying). Suppose there is an input tree $\sigma(t)$ where $\sigma \in \Sigma^1$ and $t \in \mathcal{T}_\Sigma$ is a branch of the input tree. We want to transform $\sigma(t)$ to $\sigma'(t, t')$ where t' is a transformation of t. This requires the tree transducer to copy t from the input tree to both children of σ', but the two children are treated differently: the one on the left (t) is preserved as is, but the one on the right is further transformed (t').

In short, the feature that we desire in a tree transducer is the ability to make copies of an input subtree, and to continue transformation of these copies in different states.

Probabilistic behaviour. To break statistical correlations, transformations may need to sanitize profiles in a probabilistically manner. To see this, recall the random mutation of the favorite book list in Example 3.2. In general, sanitizing strategies such as noise addition and random sampling require probabilistic transformations.

5.3 Top-down & Bottom-up Tree Transducers

Top-down and bottom-up tree transducers are the two main categories of tree transducers. Intuitively, a top-down tree transducer is like a Deterministic Finite Automaton (DFA), with the difference that the input is a tree. At the beginning, there is one read-head associated with the root of the input tree. Once the root is parsed, a different read-head will be associated to every child of the root node. In other words, the transducer spawns concurrent threads of execution to scan children of the root node. This process continues from the root to the bottom of the tree. Note that a DFA always has one read-head (one thread of execution) whereas a top-down tree transducer could have multiple read-heads concurrently.

Bottom-up tree transducers are different from top-down tree transducers in the sense that they parse the input tree from the bottom (leaves) to the top (root).

To facilitate comparison with our proposed transducer model in §6, we list below the definition of top-down and bottom-up tree transducers:

Definition 5.4 (Terms and variables). Let Σ be a ranked alphabet and $X = \{x_1, \ldots, x_n\}$ be a set of constant symbols (aka variables). The set $\mathcal{T}_\Sigma(X)$ of terms formed by function symbols in Σ and variables in X is defined as follows:

a) $\Sigma^0 \cup X \subseteq \mathcal{T}_\Sigma(X)$.
b) If $t_0, \ldots, t_{n-1} \in \mathcal{T}_\Sigma(X)$, and $\sigma \in \Sigma^n$, then $\sigma(t_0, \ldots, t_{n-1}) \in \mathcal{T}_\Sigma(X)$.

When $X = \varnothing$, we abbreviate $\mathcal{T}_\Sigma(X)$ to $\mathcal{T}_\Sigma$. Given $t \in \mathcal{T}_\Sigma(X)$, we write $var(t)$ to denote the set of variables (i.e., members of X) appearing in t.

Definition 5.5 (Terms with roots taken from a given alphabet). Suppose Σ_1 and Σ_2 are two ranked alphabets. For a set $\Gamma \subseteq \Sigma_1$, and

$H \subseteq \mathcal{T}_{\Sigma_2}(X)$, $\Gamma(H)$ is defined to be $\{\gamma(t_1, \ldots, t_k) \mid \exists k \geq 0,\ \gamma \in \Gamma^k,\ t_1, \ldots, t_k \in H\}$. That is, $\Gamma(H)$ consists of terms formed by applying a function symbol from Γ to terms from H.

Definition 5.6 (Tree transducer). Let $M = (Q, \Sigma, q_0, R)$ be a tuple where

a) Q, the set of states, is a finite, ranked alphabet containing only unary function symbols (i.e., arity is 1),
b) Σ is a ranked alphabet of input and output symbols,
c) $F \subseteq Q$ is the set of designated states, and
d) R is a finite set of transition rules.

Then M is:

- a top-down tree transducer if transition rules in R are of the form $q(\sigma(x_1, \ldots, x_k)) \rightarrow r$ for some $q \in Q$, $\sigma \in \Sigma^k$, and $r \in \mathcal{T}_\Sigma(Q(X))$ such that $var(r) \subseteq \{x_1, \ldots, x_k\}$, or
- a bottom-up tree transducer if the transition rules in R are of either the form $\sigma(q_1(x_1), \ldots, q_k(x_k)) \rightarrow q(t)$ with $k \geq 1$, $\sigma \in \Sigma^k$, $q_1, \ldots q_k \in Q$, $t \in \mathcal{T}_\Sigma(X)$, and $var(t) \subseteq \{x_1, \ldots, x_k\}$, or the form $\sigma \rightarrow q(t)$ where $q \in Q$, $\sigma \in \Sigma^0$, and $t \in \mathcal{T}_\Sigma$.

Despite the rich expressive power of the top-down and bottom-up tree transducers, they do not satisfy the requirements of view-based protection. Top-down tree transducers suffer from the lack of look-ahead. Bottom-up tree transducers has the look-ahead feature, because they transform the input tree from the bottom to the top. Moreover, total deterministic bottom-up tree transducers are closed under composition, and every deterministic bottom-up tree transducer can be converted to an equivalent total deterministic bottom-up tree transducer. Deterministic transition in such a tree transducer model also eases the extension of the model to a probabilistic one. This stands in stark contrast to non-deterministic tree transducers.

The bottom-up tree transducer, however, has its own weaknesses. More specifically, while top-down tree transducers support differential copying, bottom-up tree transducers do not. In short, these two classes of tree transducers are incomparable in expressiveness [10], and none of the them support both look-ahead and differential copying.

5.4 Multi Bottom-up Tree Transducers

The weakness of bottom-up tree transducers is addressed in the ***Multi Bottom-up Tree Transducer (MBUTT)***, which is a generalization of bottom-up tree transducer proposed by Maletti [11, 17]. More specifically, an MBUTT allows outputting a sequence of trees in each transition rule. This extension is signified by the word "multi". According to Definition 5.6, the right-hand side of every transition rule in a bottom-up tree transducer belongs to $Q(\mathcal{T}_\Sigma(X))$, while all states in Q are unary. As a result, only one tree from $\mathcal{T}_\Sigma(X)$ can be output in a transition rule. On the contrary, in the MBUTT, a sequence of trees might be output in a transition rule. This has been made feasible by allowing states to be of ranks larger than one. This means, in Example 5.3, an MBUTT can be designed so that when it parses t, it keeps one copy of t and one copy of its transformed form t'. When the MBUTT reaches the root σ, it has already transformed t to t' while it has preserved the original copy of t.

Below is the formal definition of MBUTTs.

Definition 5.7 (Linear and normalized terms). A term t is ***linear*** in $V \subseteq X$ if every variable $x \in V$ appears at most once in t. A term t is ***normalized*** if (a) $var(t) = \{x_1, \ldots, x_m\}$ for some $m \in \mathbb{N}$, (b) each variable appears exactly once, and (c) the variables appear in exactly the order $x_1, \ldots, x_m$.

Definition 5.8 (Multi bottom-up tree transducer [17]). A MBUTT is a tuple (Q, Σ, F, R) where

a) Q, the set of states, is a uniquely-ranked alphabet, disjoint with $\Sigma \cup X$,
b) Σ is a ranked alphabet of input and output symbols, disjoint with X,
c) $F \subseteq Q^1$ is the set of final states, and
d) R is a finite set of transition rules of the form $l \rightarrow r$ where $l \in \Sigma(Q(X))$ is normalized and linear in X, $r \in Q(\mathcal{T}_\Sigma(X))$, and $var(r) \subseteq var(l)$.

An MBUTT is ***deterministic*** (respectively, ***total deterministic***) if for every $l \in \Sigma(Q(X))$, there exists at most one (respectively, exactly one) r such that $l \rightarrow r \in R$.

The semantics of MBUTT, specified by way of term rewriting, is detailed in [17]. It has been proved in the same work that every deterministic MBUTT can be converted to an equivalent total deterministic MBUTT. A composition construction has been given to show how two MBUTTs can be composed to create another MBUTT. It is proved in [17] that given two total deterministic MBUTTs, their composition is also a total deterministic MBUTT.

As explained in the previous section, the MBUTT has almost all the requirements we need including closure under composition (if it is total deterministic), look-ahead and differential copying. The only required feature that is missing is probabilistic behaviour. Therefore, the original definition of the MBUTT needs to be extended so that it allows the transducer to behave probabilistically.

6 PMBUTT

In this section we propose a new type of tree transducers, called the ***Probabilistic Multi Bottom-up Tree Transducer (PMBUTT)***, which is obtained by extending MBUTT to incorporate probabilistic behaviour. We advocate the use of PMBUTT as a programming language-independent implementation of view materialization.

6.1 Syntax

The PMBUTT is defined by adding probabilistic behaviour to the MBUTT (Def. 5.8). In other words, a left-hand side may probabilistically transitions to multiple right-hand sides. The transition probabilities must add up to 1 for the same left-hand side. Deterministic behaviour can be simulated by assigning the maximum transition probability, which is 1, to a transition.

Definition 6.1 (Probabilistic multi bottom-up tree transducers). A PMBUTT is a tuple (Q, Σ, F, R) where

a) Q, the set of states, is a uniquely-ranked alphabet, disjoint with $\Sigma \cup X$.
b) Σ is a ranked alphabet of input and output symbols, disjoint with X.
c) $F \subseteq Q^1$ is a set of final states.

d) R is a finite set of probabilistic transition rules of the form $l \xrightarrow{p} r$ where $l \in \Sigma(Q(X))$ is normalized and linear in X, $r \in Q(\mathcal{T}_\Sigma(X))$, $var(r) \subseteq var(l)$, and $p \in [0,1]$.

e) For every rule $l \xrightarrow{p} r \in R$, $peer(l \xrightarrow{p} r)$ is defined to be $\{l \xrightarrow{p'} r' \mid l \xrightarrow{p'} r' \in R\}$. Then for every rule $u \in R$, the following must hold

$$\sum_{l \xrightarrow{p} r \in peer(u)} p = 1$$

In the above definition, $peer(u)$ (for some $u \in R$) contains all transition rules in R that share the same left-hand side with u. Therefore, item (e) enforces that transitions from the same left-hand side sum up to 1. Note that all transition rules in the PMBUTT consumes exactly one input symbol.

A PMBUTT M is total if for every $l \in \Sigma(Q(X))$, there exists at least one r such that $l \xrightarrow{p} r \in R$.

6.2 Semantics

The rewrite semantics for the PMBUTT $M = (Q, \Sigma, F, R)$ is defined by Definition 6.2.

Definition 6.2. Let $t, t' \in \mathcal{T}_\Sigma(Q(\mathcal{T}_\Sigma))$, and $l \xrightarrow{p} r \in R$. We write $t \Rightarrow_{M,p}^{l \xrightarrow{p} r} t'$ if $p > 0$ and there exists $w \in \mathcal{P}os(t)$ and $\theta : X \to \mathcal{T}_\Sigma$ such that $t|_w = l\theta$ and $t' = t[r\theta]_w$, and we write $t \Rightarrow_{M,p} t'$ if there exists $l \xrightarrow{p} r \in R$ such that $t \Rightarrow_{M,p}^{l \xrightarrow{p} r} t'$. Moreover, we write $t \rightsquigarrow_{M,p}^{s} t'$ if there exists a finite sequence of transition rules $s : l_1 \xrightarrow{p_1} r_1, l_2 \xrightarrow{p_2} r_2, \ldots, l_n \xrightarrow{p_n} r_n$ in R such that $t \Rightarrow_{M,p_1}^{l_1 \xrightarrow{p_1} r_1} t_1 \Rightarrow_{M,p_2}^{l_2 \xrightarrow{p_2} r_2} t_2 \ldots \Rightarrow_{M,p_n}^{l_n \xrightarrow{p_n} r_n} t'$, and $p = p_1 \times p_2 \times \ldots \times p_n$. If there are exactly $k \geq 1$ distinct sequences $s_1, \ldots, s_k$ of transition rules such that $t \rightsquigarrow_{M,p_i}^{s_i} t'$ for two fixed t and t', then we write $t \Rightarrow_{M,p}^{*} t'$ where $p = \sum_{i=1}^{k} p_i$. If all such sequences $s_1, \ldots, s_k$ are of length n, we write $t \Rightarrow_{M,p}^{n} t'$. The tree transformation computed by M is $\tau_M = \{(t, t'|_1, p) \in \mathcal{T}_\Sigma \times \mathcal{T}_\Sigma \times [0,1] \mid t \in \mathcal{T}_\Sigma,\ t' \in F(\mathcal{T}_\Sigma), t \Rightarrow_{M,p}^{*} t'\}$.

Note that given a tree $t \in \mathcal{T}_\Sigma$, the first rules that we have to apply are the ones that do not have any state in their left-hand side. Moreover, it is possible that a term t be transformed to t' by taking two different sequences of probabilistic transition rules. That is why that associated probability to $t \Rightarrow_{M,p}^{*} t'$ is the summation of all probabilities p_i for which there is a distinct sequence of transitions s_i such that $t \rightsquigarrow_{M,p_i}^{s_i} t'$.

Computation of M for a given input tree is always terminating because all transition rules are input consuming. Therefore, after finite number of transitions, there will be a point that either all input symbols are consumed or there is no more applicable transition rules. Note that τ_M is always a finite set for the same reason.

Example 6.3. Consider a user profile with a list of favorite books. Each book has a title and an author. For the sake of simplicity, we consider only the author of each book. Suppose also that there are only two names that could appear as the author of a book, a and b. Such a profile could be modelled using an alphabet $\Sigma = \{profile^1, book^2, a^0, b^0, nil^0\}$ where *profile* is a function symbol that appears only at the root of the profile. Fig. 4 shows a sample user profile created from Σ. Given such a profile, one can infer the user's favorite author by counting the number of times every author appeared in the user's list of favorite books. To counter that, we want to randomly inject books into that list. The PMBUTT $M = (\{q_0, q_1, q_2, q_f\}, \Sigma, \{q_f\}, R)$ can produce such a transformation with the following rules in R:

Figure 4: A sample user profile

1) $a \xrightarrow{1} q_1(a)$ 2) $b \xrightarrow{1} q_1(b)$

3) $nil \xrightarrow{1} q_0(nil)$

4) $book(q_1(x_1), q_0(x_2)) \xrightarrow{0.5} q_0(book(x_1, x_2))$

5) $book(q_1(x_1), q_0(x_2)) \xrightarrow{0.125} q_0(book(x_1, book(a, x_2)))$

6) $book(q_1(x_1), q_0(x_2)) \xrightarrow{0.125} q_0(book(x_1, book(b, x_2)))$

7) $book(q_1(x_1), q_0(x_2)) \xrightarrow{0.125} q_0(book(a, book(x_1, x_2)))$

8) $book(q_1(x_1), q_0(x_2)) \xrightarrow{0.125} q_0(book(b, book(x_1, x_2)))$

9) $profile(q_0(x_1)) \xrightarrow{1} q_f(profile(x_1))$

When the PMBUTT parses a book in the list of books, with probability 0.5 it outputs the book as it is. Otherwise, it injects noise: with equal probabilities it randomly injects a book written by a or b before or after the current item in the list of books. The size of the list is at most doubled.

Example 6.4. The computation of the PMBUTT M, described in Example 6.3, is demonstrated on the input tree $profile(book(a, book(b, nil)))$. Note that r_i denotes rule i from Example 6.3.

$$\begin{aligned}
& profile(book(a, book(b, nil))) \\
\Rightarrow_{M,1}^{r_3} \quad & profile(book(a, book(b, q_0(nil)))) \\
\Rightarrow_{M,1}^{r_2} \quad & profile(book(a, book(q_1(b), q_0(nil)))) \\
\Rightarrow_{M,1}^{r_1} \quad & profile(book(q_1(a), book(q_1(b), q_0(nil)))) \\
\Rightarrow_{M,0.125}^{r_8} \quad & profile(book(q_1(a), q_0(book(b, book(b, nil))))) \\
\Rightarrow_{M,0.125}^{r_5} \quad & profile(q_0(book(a, book(a, book(b, book(b, nil)))))) \\
\Rightarrow_{M,1}^{r_9} \quad & q_f(profile(book(a, book(a, book(b, book(b, nil))))))
\end{aligned}$$

As a result, the following holds:

$$profile(book(a, book(b, nil))) \Rightarrow_{M,0.015625}^{*} q_f(profile(book(a, book(a, book(b, book(b, nil)))))))$$

6.3 Composition construction

The composition of two PMBUTTs M and N are denoted by $M; N$, in which the input tree is first processed by M and then the result will go through N. In this work, the composition construction proposed

by Maletti in [17] is extended to compose two PMBUTTss. The main idea is similar to other composition constructions proposed for combining state machines, e.g., combining two DFAs to compute another DFA that accepts only sequences that are accepted by both of the automata. In these constructions, the set of states for the composed machine is the cross-product of the state sets of the two input machines. Moreover, this approach simulates the second state machine on the right-hand side of the transition rules of the first machine. We follow the same approach for creating the set of states in $M;N$. However, the difference is that a state from M may output more than one tree. That is why a state in a PMBUTT may have an arity larger than one. As a result, when we take the cross-product of the state sets from M and N to create the set of states in $M;N$, we should take this into consideration: i.e., output trees of a state in M arrives at multiple states in N. As a result, the states in $M;N$ will be of the form $q\langle h_1,\dots,h_k\rangle$ where q is a state of rank k in M and $h_1,\dots,h_k$ are states in N. The arity of a state $q\langle h_1,\dots,h_k\rangle$ in $M;N$ will be the summation of the arities of h_1 to h_k.

Let $M=(Q,\Sigma,F_M,R_M)$ and $N=(H,\Sigma,F_N,R_N)$ where Q, H, and Σ are disjoint. A uniquely-ranked alphabet is defined as follows:

$$Q\langle H\rangle=\{q\langle h_1,\dots,h_n\rangle \mid q^n\in Q, h_1,\dots,h_n\in H\}$$

where $rk(q\langle h_1,\dots,h_n\rangle)=\sum_{i=1}^{n} rk(h_i)$ for every $q^n\in Q$ and $h_1,\dots,h_n\in H$.

Now that we know how to create the set of states in $M;N$ from the set of states in M and N, we move on to show how the transition rules are created for the composed machine. Let $q\langle h_1,\dots,h_n\rangle$ be a state in $M;N$ of rank k. This means we expect the term $q\langle h_1,\dots,h_n\rangle(u_1,\dots,u_k)$ to appear in the transition rules of $M;N$ where $u_i\in\mathcal{T}_\Sigma(X)$ for $1\le i\le k$. Intuitively, this term is actually computed by $q(h_1(u_1,\dots,u_{rk(h_1)}),\dots,h_n(u_{k-rk(h_n)+1},\dots,u_k))$. As a result, a mapping needs to be defined to describe the equality between these two terms. Let $U=\mathcal{T}_\Sigma(X)$.

The mapping $\varphi:\mathcal{T}_\Sigma(Q\langle H\rangle(U))\to\mathcal{T}_\Sigma(Q(H(U)))$ is defined such that for every $q\langle h_1,\dots,h_n\rangle\in Q\langle H\rangle^k$, $u_1,\dots,u_k\in U$, $\sigma\in\Sigma^k$, and $t_1,\dots,t_k\in\mathcal{T}_\Sigma(Q\langle H\rangle(U))$:

$$\varphi(q\langle h_1,\dots,h_n\rangle(u_1,\dots,u_k))=q(h_1(u_1,\dots,u_{rk(h_1)}),\dots,h_n(u_{k-rk(h_n)+1},\dots,u_k))$$
$$\varphi(\sigma(t_1,\dots,t_k))=\sigma(\varphi(t_1),\dots,\varphi(t_k))$$

So far, it has been shown that given $M=(Q,\Sigma,F_M,R_M)$ and $N=(H,\Sigma,F_N,R_N)$, $Q\langle H\rangle$ will be the set of states in $M;N$. This implies that transitions in $M;N$ will be of the form $l\xrightarrow{p} r$ where $l\in\Sigma(Q\langle H\rangle(X))$, $r\in Q\langle H\rangle(\mathcal{T}_\Sigma(X))$, and $p\in[0,1]$. $l\xrightarrow{p} r$ appears in the transition rules of $M;N$, if there exist a u such that l is transformed to u by M, and then u is transformed to r by N. In other words, $l\Rightarrow_{M,p_1} u$ and $u\Rightarrow_{N,p_2} r$ where $p=p_1.p_2$. To make the syntax right, and simplify this requirement, we have to use the mapping φ and require $\varphi(l)(\Rightarrow_{M,p_1};\Rightarrow_{N,p_2})\varphi(r)$ to hold in order for $l\xrightarrow{p_1.p_2} r$ to appear in the transition rules in $M;N$. However, this requirement is not completely correct. The reason is that u may contain multiple symbols from Σ that must be all consumed by N. This is clarified via an example. Let $a,b,c\in\Sigma^1$, $q\in Q^1$, and $h\in H^1$. Moreover, assume:

1) $a(q(x_1))\xrightarrow{1} q(b(c(x_1)))\in R_M$

2) $c(h(x_1))\xrightarrow{1} h(c(x_1))\in R_N$

3) $b(h(x_1))\xrightarrow{1} h(b(x_1))\in R_N$

Now, given a left-hand side $l=a(q\langle h\rangle(x_1))$, we want to find a right-hand side r and a probability p such that $l\xrightarrow{p} r$ appears in $R_{M;N}$. Remember the requirement was that $\varphi(l)(\Rightarrow_{M,p_1};\Rightarrow_{N,p_2})\varphi(r)$ holds. We follow the below steps to find r:

$$\varphi(a(q\langle h\rangle(x_1)))=a(q(h(x_1)))$$
$$a(q(h(x_1)))\Rightarrow_{M,1} q(b(c(h(x_1))))\text{ via rule 1}$$
$$q(b(c(h(x_1))))\Rightarrow_{N,1} q(b(h(c(x_1))))\text{ via rule 2}$$

$\varphi(r)$ equals $q(b(h(c(x_1))))$, which means $r=\varphi^{-1}(q(b(h(c(x_1)))))$. However, r cannot be computed because $q(b(h(c(x_1)))))$ does not belong to the co-domain of the mapping φ which is $\mathcal{T}_\Sigma(Q(H(\mathcal{T}_\Sigma(X))))$. The reason is that in the right-hand side of the rule 1, there are more than one output symbol that must be consumed by N. Therefore, we need to continue applying transition rules from the second machine, N, to consume the other symbol too. This means the below step must be further followed:

$$q(b(h(c(x_1))))\Rightarrow_{N,1} q(h(b(c(x_1))))\text{ via rule 3}$$

Now r is computed by $\varphi^{-1}(q(h(b(c(x_1)))))$ which equals $q\langle h\rangle(b(c(x_1)))$. As a result, $a(q\langle h\rangle(x_1))\xrightarrow{p} q\langle h\rangle(b(c(x_1)))$ is inserted in $R_{M;N}$ where $p=1\times1\times1=1$ because the associated probabilities to the three rules involved are all 1.

It is concluded that the requirement must be revised to $\varphi(l)(\Rightarrow_{M,p_1};\Rightarrow^*_{N,p_2})\varphi(r)$ in order for $l\xrightarrow{p_1.p_2} r$ to appear in the transition rules of $M;N$. Last but not least, since there might be different paths from $\varphi(l)$ to $\varphi(r)$ for two fixed l and r, we have to take all those paths into consideration and add up their probabilities.

Now, the composition algorithm is defined as follows:

Theorem 6.5. *Let $M=(Q,\Sigma,F_M,R_M)$ and $N=(H,\Sigma,F_N,R_N)$ be two PMBUTTs s.t. Q, H and Σ are mutually disjoint. The composition of M and N is the PMBUTT*

$$M;N=(Q\langle H\rangle,\Sigma,F_M\langle F_N\rangle,R)$$

s.t. for every $l\in\Sigma(Q\langle H\rangle(X))$ and $r\in Q\langle H\rangle(\mathcal{T}_\Sigma(X))$, if there are exactly $k\ge1$ rules $u_1,\dots,u_k$ in R_M such that $\varphi(l)(\Rightarrow^{u_i}_{M,p_i^M};\Rightarrow^*_{N,p_i^N})\varphi(r)$, then $l\xrightarrow{p} r\in R$ where

$$p=\sum_{i=1}^{k} p_i^M\times p_i^N.$$

Therefore, any two PMBUTTss can be composed. A proof of this theorem is given in §5.4.4 (Theorem 5.4.6) of [1].

Example 6.6. This example goes through the details of composing two transformations M and N where the input profile is similar to the profile used in Example 6.3. M, which is a simpler version of the PMBUTT introduced in Example 6.3, randomly (with probability 0.5) injects a book authored by author a before every book in the list of favorite books in the user profile. Then, N transforms the

output of M by generalizing the author of every book to the value ab which shows the author is either a or b. Note that the goal is only to exemplify how the composition algorithm works and actual effectiveness of the transformations performed by M and N is not the concern of this example.

The set of transition rules for $M = (Q_M, \Sigma, \{q_f\}, R_M)$ with $Q_M = \{q_0, q_1, q_f\}$ is defined as follows:

$$a \xrightarrow{1} q_1(a) \qquad b \xrightarrow{1} q_1(b) \qquad nil \xrightarrow{1} q_0(nil)$$
$$book(q_1(x_1), q_0(x_2)) \xrightarrow{0.5} q_0(book(x_1, x_2))$$
$$book(q_1(x_1), q_0(x_2)) \xrightarrow{0.5} q_0(book(a, book(x_1, x_2)))$$
$$profile(q_0(x_1)) \xrightarrow{1} q_f(profile(x_1))$$

The set of transition rules for $N = (Q_N, \Sigma, \{h_f\}, R_N)$ with $Q_N = \{h_0, h_1, h_f\}$ is defined as follows:

$$a \xrightarrow{1} h_1(ab) \qquad b \xrightarrow{1} h_1(ab) \qquad nil \xrightarrow{1} h_0(nil)$$
$$book(h_1(x_1), h_0(x_2)) \xrightarrow{1} h_0(book(x_1, x_2))$$
$$profile(h_0(x_1)) \xrightarrow{1} h_f(profile(x_1))$$

The first step to compose M and N is to convert them to total PMBUTTs using the technique describe in the proof of Lemma 5.4.9 in [1]. We skip this step as it is uninspiring. Now following the algorithm of Theorem 6.5 to obtain $M;N$ results in a PMBUTT with the following states:

$$\{\, q_0\langle h_0\rangle^1, q_0\langle h_1\rangle^1, q_0\langle h_f\rangle^1, q_1\langle h_0\rangle^1, q_1\langle h_1\rangle^1, q_1\langle h_f\rangle^1, q_f\langle h_0\rangle^1, q_f\langle h_1\rangle^1, q_f\langle h_f\rangle^1 \,\}$$

Some of the relevant transition rules of $M;N$ are as follows:

$$a \xrightarrow{1} q_1\langle h_1\rangle(ab) \qquad b \xrightarrow{1} q_1\langle h_1\rangle(ab) \qquad nil \xrightarrow{1} q_0\langle h_0\rangle(nil)$$
$$book(q_1\langle h_1\rangle(x_1), q_0\langle h_0\rangle(x_2)) \xrightarrow{0.5} q_0\langle h_0\rangle(book(x_1, x_2))$$
$$book(q_1\langle h_1\rangle(x_1), q_0\langle h_0\rangle(x_2)) \xrightarrow{0.5} q_0\langle h_0\rangle(book(ab, book(x_1, x2)))$$
$$profile(q_0\langle h_0\rangle(x_1)) \xrightarrow{1} q_f\langle h_f\rangle profile(x_1)$$

As can be seen in the above transition rules, $M;N$ transforms a's and b's to ab and it may inject new book items with ab as the value of the author.

6.4 Evaluation

In the following, we assess the proposed model of view materialization. More specifically, the expressiveness of PMBUTT is evaluated in §6.4.1. Then the complexity of the composition construction is studied in §6.4.2.

6.4.1 Expressiveness. To assess the expressive power of PMBUTT, we demonstrate that it can capture classical sanitizing transformations from the literature, namely, suppression, generalization, noise addition, and random sampling.

Example 6.7 (Suppression). Suppression can easily be described by a PMBUTT. All it takes is to replace a branch with the constant symbol ϵ. For instance, consider the PMBUTT N from Example 6.6. We can simply suppress the author of every book with the following rule:

$$book(h_1(x_1), h_0(x_2)) \xrightarrow{1} h_0(book(\epsilon, x_2))$$

Example 6.8 (Generalization). The PMBUTT N from Example 6.6 performs generalization.

Example 6.9 (Sampling). Sampling requires probabilistic behaviour. Suppose we want to devise a transformation for the type of profiles used in Example 6.6, to sample from the list of books in the user profile. The following set of transitions can achieve this effect:

$$a \xrightarrow{1} h_1(a) \qquad b \xrightarrow{1} h_1(b) \qquad nil \xrightarrow{1} h_0(nil)$$
$$book(h_1(x_1), h_0(x_2)) \xrightarrow{0.5} h_0(book(x_1, x_2))$$
$$book(h_1(x_1), h_0(x_2)) \xrightarrow{0.5} h_0(x_2)$$
$$profile(h_0(x_1)) \xrightarrow{1} h_f(profile(x_1))$$

Example 6.10 (Noise addition). The PMBUTT M in Example 6.3 implements noise addition.

Note that the PMBUTT cannot model noise addition when the number of injected branches is selected randomly. The reason is that for such a transformation, we need to have a cycle with at least one transition that does not consume any input. However, the definition of PMBUTT does not allow epsilon rules: i.e., rules that do not consume any input. A PMBUTT can be defined to randomly inject up to m noisy branches, though the machine may get very large. However, if m is not pre-defined, then an arbitrary number is randomly selected at runtime, and such a transformation cannot be expressed in PMBUTT. In general, any transformation that requires epsilon rules is not expressible in PMBUTT.

The reader is referred to [1, §5.4.5.2] for examples of how PMBUTT can be used for specifying two other classical sanitizing transformations: noise multiplication and permutation.

6.4.2 Complexity of the Composition Construction. This section analyzes the complexity of the composition construction described in §6.3.

Given $t \in \mathcal{T}_\Sigma(X)$ and $V \subseteq \Sigma \cup X$, we write $|t|_V$ to denote the number of time symbols from V appear in term t. For example $|t|_{\Sigma \cup X}$ is the size of t.

We now attend to the complexity of the composition construction as described in Theorem 6.5. Suppose PMBUTTs $M = (Q_M, \Sigma, F_M, R_M)$ and $N = (Q_N, \Sigma, F_N, R_N)$ are composed to $M;N$. Three parameters are defined as follows:

$$c_1 = \max_{l \xrightarrow{p} r \in R_M} |l|_X \qquad c_2 = \max_{l \xrightarrow{p} r \in R_M} |r|_\Sigma$$
$$c_3 = \max_{u \in R_N} |peer(u)|$$

Parameter c_1 is the maximum number of variables that appear in the left-hand sides of the transition rules in R_M. Assume there is a transition rule in R_M as follows:

$$f(q_1(x_1, x_2), q_2(x_3)) \xrightarrow{p} q_1(g(m(x_1), x_2), m(x_3)) \qquad (1)$$

where $f^2, m^1, g^2 \in \Sigma$ and $q_1^2, q_2^1 \in Q_N$. Since there are three variables in the left-hand side of the above rule, when we compose M and N,

three states from Q_N are needed to combine the above rule with rules from Q_N. For instance, if $Q_N = \{h_1^1, h_2^2\}$, then corresponding to the above rule, the following terms will appear in the left-hand side of the rules in $M; N$:

$$\begin{aligned} &1) \quad f(q_1\langle h_1, h_1\rangle(x_1, x_2), q_2\langle h_1\rangle(x_3)) \\ &2) \quad f(q_1\langle h_1, h_1\rangle(x_1, x_2), q_2\langle h_2\rangle(x_3, x_4)) \\ &3) \quad f(q_1\langle h_1, h_2\rangle(x_1, x_2, x_3), q_2\langle h_1\rangle(x_4)) \\ &\vdots \\ &8) \quad f(q_1\langle h_2, h_2\rangle(x_1, x_2, x_3, x_4), q_2\langle h_2\rangle(x_5, x_6)) \end{aligned} \tag{2}$$

In the above example, since Q_N contains two states, and there are three variables in the left-hand side of the rule denoted by (1), there will be 8 ($= 2^3$) corresponding left-hand sides in the transition rules of $M; N$. We can generalize this figure to $|R_M|.|Q_N|^{c_1}$. Once we applied a transition rule from M on one of such left-hand sides, the result term could go through different probabilistic execution paths in N. For example, assume we apply the rule (1) from R_M on the $\varphi(l)$ where l is a candidate left-hand side denoted by (2). This means:

$$\begin{aligned} &\varphi(f(q_1\langle h_1, h_1\rangle(x_1, x_2), q_2\langle h_1\rangle(x_3))) \\ &\quad = f(q_1(h_1(x_1), h_1(x_2)), q_2(h_1(x_3))) \\ &\quad \Rightarrow_{M,p} q_1(g(m(h_1(x_1)), h_1(x_2), m(h_1(x_3)))) \end{aligned}$$

Now we keep applying rules from R_N on the resulted right-hand side to consume all the symbols. In the above example, there are three symbols that must be consumed. c_2 is the upper bound on the number of symbols in the resulted right-hand-sides. If N is probabilistic, every one of those symbols may be consumed through applying a different transition. c_3 is the upper bound on the number of transitions in R_N that share the same left-hand side. As a result, $c_2^{c_3}$ possible right-hand sides might be resulted for every left-hand side. In total, there will be $\mathcal{O}(|R_M|.|Q_N|^{c_1}.c_2^{c_3})$ rules in $R_{M;N}$. Overall, the complexity of composing two PMBUTTs is $\mathcal{O}(|R_M|.|Q_N|^{c_1}.c_2^{c_3})$.

Although the complexity of the composition construction is exponential. Composition is performed only when a user subscribes to an application, and not during query time. Consequently, the system only needs to perform this costly computation once, and then store a constant number of PMBUTTs for each user-application pair

7 RELATED WORK

Inference Attacks against Social Network Datasets. Inference attacks were studied in the context of statistical databases [18]. To prevent inference attacks in statistical databases, a typical approach is to disallow some access queries. For instance, queries that retrieve too small or too large number of records will not be allowed.

In the context of social computing platforms, the dominant focus has been on inference attacks against social network datasets, i.e., a dataset of user profiles, no matter how the profiles have been collected. Zheleva and Getoor [20] discuss how friendship links and group affiliations can result in information disclosure. Two sources of information are used to infer the value of a sensitive attribute in a private user profile: (a) values of the same attribute public user profiles that participate in a friendship relation with the private profile, and (b) values of the same attribute in public user profiles that belong to the same social groups as the private profile does. They conducted an experiment to leverage these two sources of information for making inferences. A similar approach is adopted by Becker and Chen in [6]. Labitzke *et al.* [14] demonstrate empirically that some attributes such as location and age are strongly correlated among friends. By contrast, there are some attributes that are rarely made available to public and correlate little. Note that their experiment uses a dataset of profiles with publicly available attributes. Similar results were reported in [7].

There are also some works that use supervised machine learning methods for launching inference attacks against user profiles. [12, 13, 15, 19] employ Naive Bayesian Network classifiers to infer users' private information.

Extending the work of Ahmadinejad *et al.*, this work is unique in that the attacker does not have access to a full social network dataset, but instead performs inference attacks covertly via the extension API of an SNS [2–5]. A mathematical framework for specifying and establishing the privacy and utility goals of view-based protection is formulated by Ahmadinejad *et al.* [5]. They also formally articulate the trade-off between privacy and utility when a view is imposed on user profiles.

Protection Mechanisms for Social Network Systems. Previously proposed protection mechanisms for enhancing the privacy of user profiles have focused on either encrypting the content of user profiles, or providing a finer-grained authorization mechanism. These techniques do not prevent SNS API inference attacks in particular. For instance, Lucas *et al.* [16] proposed an architecture for social networks where the SNS has access to only encrypted user information. A Javascript client-side Facebook application, called *FlyByNight* plays the role of an information broker that encrypts every information before being uploaded to the SNS.

Egele *et al.* [8, 9] embed a fine-grained access control system, PoX (proxy on the client side), in a client-side Facebook application (i.e., no change to Facebook). Every request to access a user profile is sent to PoX where users can impose fine-grained access control on their data. If access is granted, the request is forwarded to Facebook.

As we argued in this paper, access control cannot break the statistical correlation between sensitive and accessible data, and inference attacks cannot be eliminated by authorization mechanisms only. Our proposed view-based protection model sanitizes the user profile using a probabilistic tree transformation, before the sanitized profile is queried by potentially malicious SNS extensions.

8 CONCLUSION

In this work, we first demonstrated that carefully sanitizing user profiles can significantly reduce the chance of malicious third-party extensions inferring users' private information. More specifically, we evaluated the reduction in success rate of the Ahmadinejad *et al.*'s inference algorithms [4] when view-based sanitization is performed. The benefit of probabilistic profile transformation is highly pronounced in our results. The performance of lazy view materialization is then evaluated. We made use of Haskell's lazy evaluation to demonstrate that when view materialization is performed in a lazy manner, significant performance gain can be observed. Lastly, we proposed a programming language-independent, probabilistic

tree transducer model, PMBUTT, for materializing views. An important feature of the model is that it is closed under composition. We also showed via examples that the proposed tree transducer model is expressive enough for capturing classical sanitizing transformations such as suppression, generalization, noise addition and random sampling.

ACKNOWLEDGMENTS

This work is supported in part by a Discovery Grant from the Natural Sciences and Engineering Research Council of Canada (RGPIN-2014-06611) and a Canada Research Chair (950-229712).

REFERENCES

[1] Seyed Hossein Ahmadinejad. 2016. *A View-Based Protection Model to Prevent Inference Attacks by Third-Party Extensions to Social Computing Platforms.* Ph.D. Dissertation. University of Calgary.

[2] Seyed Hossein Ahmadinejad, Mohd Anwar, and Philip W. L. Fong. 2011. Inference attacks by third-party extensions to social network systems. In *Proceedings of 2011 IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops'2011).* Seattle, USA, 282–287.

[3] Seyed Hossein Ahmadinejad and Philip W. L. Fong. 2013. On the feasibility of inference attacks by third-party extensions to social network systems. In *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security (ASIACCS'13).* Hangzhou, China, 161–166.

[4] Seyed Hossein Ahmadinejad and Philip W. L. Fong. 2014. Unintended disclosure of information: Inference attacks by third-party extensions to Social Network Systems. *Computers and Security* 44 (2014), 75–91.

[5] Seyed Hossein Ahmadinejad, Philip W. L. Fong, and Rei Safavi-Naini. 2016. Privacy and Utility of Inference Control Mechanisms for Social Computing Applications. In *Proceedings of the 11th ACM Asia Conference on Computer and Communication Security (ASIACCS'2016).* Xi'an, China, 829–840.

[6] Justin Becker and Hao Chen. 2009. Measuring privacy risk in online social networks. In *Proceedings of the Web 2.0 Security and Privacy Workshop (W2SP'09).* Oakland, CA, USA, 8.

[7] Ratan Dey, Cong Tang, Keith Ross, and Nitesh Saxena. 2012. Estimating age privacy leakage in online social networks. In *Proceedings of the 2012 IEEE INFOCOM.* Orlando, Florida, USA, 2836–2840.

[8] Manuel Egele, Andreas Moser, Christopher Kruegel, and Engin Kirda. 2011. PoX: Protecting users from malicious Facebook applications. In *Proceedings of the IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops'2011).* Seattle, WA, USA, 288–294.

[9] Manuel Egele, Andreas Moser, Christopher Kruegel, and Engin Kirda. 2012. PoX: Protecting Users from Malicious Facebook Applications. *Computer Communications* 35, 12 (July 2012), 1507–1515.

[10] Joost Engelfriet. 1975. Bottom-up and Top-down Tree Transformations – A Comparison. *Mathematical Systems Theory* 9, 2 (June 1975), 198–231.

[11] Joost Engelfriet, Eric Lilin, and Andreas Maletti. 2009. Extended multi bottom-up tree transducers: Composition and decomposition. *Acta Informatica* 46, 8 (Dec. 2009), 561–590.

[12] Jianming He, Wesley W. Chu, and Zhenyu (Victor) Liu. 2006. Inferring privacy information from social networks. In *Proceedings of the 4th IEEE International Conference on Intelligence and Security Informatics (ISI'06).* San Diego, CA, USA, 154–165.

[13] R. Heatherly, M. Kantarcioglu, and B. Thuraisingham. 2009. *Preventing private information inference attacks on social networks.* Technical Report. Computer Science Department, University of Texas at Dallas.

[14] Sebastian Labitzke, Florian Werling, Jens Mittag, and Hannes Hartenstein. 2013. Do Online Social Network Friends Still Threaten My Privacy?. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy (CODASPY'13).* San Antonio, Texas, USA, 13–24.

[15] Jack Lindamood, Raymond Heatherly, Murat Kantarcioglu, and Bhavani Thuraisingham. 2009. Inferring private information using social network data. In *Proceedings of the 18th International Conference on World Wide Web (WWW'09).* Madrid, Spain, 1145–1146.

[16] Matthew M Lucas and Nikita Borisov. 2008. FlyByNight: Mitigating the privacy risks of social networking. In *Proceedings of the 7th ACM Workshop on Privacy in the Electronic Society (WPES'08).* Alexandria, VA, USA, 1–8.

[17] Andreas Maletti. 2008. Compositions of extended top-down tree transducers. *Information and Computation* 206, 9–10 (2008), 1187–1196.

[18] Dorothy Elizabeth Robling Denning. 1982. Inference Controls. In *Cryptography and Data Security.* Addison-Wesley, 331–390.

[19] Wanhong Xu, Xi Zhou, and Lei Li. 2008. Inferring privacy information via social relations. In *Proceedings of the 24th IEEE International Conference on Data Engineering Workshop (ICDEW'08).* Cancun, Mexico, 525–530.

[20] Elena Zheleva and Lise Getoor. 2009. To join or not to join: the illusion of privacy in social networks with mixed public and private user profiles. In *Proceedings of the 18th International Conference on World Wide Web (WWW'09).* Madrid, Spain, 531–540.