2015 IEEE Symposium on Security and Privacy

Towards Making Systems Forget with Machine Unlearning

Yinzhi Cao and Junfeng Yang
Columbia University
{yzcao, junfeng}@cs.columbia.edu

***Abstract*—Today's systems produce a rapidly exploding amount of data, and the data further derives more data, forming a complex data propagation network that we call the data's *lineage*. There are many reasons that users want systems to forget certain data including its lineage. From a privacy perspective, users who become concerned with new privacy risks of a system often want the system to forget their data and lineage. From a security perspective, if an attacker pollutes an anomaly detector by injecting manually crafted data into the training data set, the detector must forget the injected data to regain security. From a usability perspective, a user can remove noise and incorrect entries so that a recommendation engine gives useful recommendations. Therefore, we envision forgetting systems, capable of forgetting certain data and their lineages, completely and quickly.**

This paper focuses on making learning systems forget, the process of which we call *machine unlearning*, or simply *unlearning*. We present a general, efficient unlearning approach by transforming learning algorithms used by a system into a summation form. To forget a training data sample, our approach simply updates a small number of summations – asymptotically faster than retraining from scratch. Our approach is general, because the summation form is from the statistical query learning in which many machine learning algorithms can be implemented. Our approach also applies to all stages of machine learning, including feature selection and modeling. Our evaluation, on four diverse learning systems and real-world workloads, shows that our approach is general, effective, fast, and easy to use.

I. Introduction

A. The Need for Systems to Forget

Today's systems produce a rapidly exploding amount of data, ranging from personal photos and office documents to logs of user clicks on a website or mobile device [15]. From this data, the systems perform a myriad of computations to derive even more data. For instance, backup systems copy data from one place (e.g., a mobile device) to another (e.g., the cloud). Photo storage systems re-encode a photo into different formats and sizes [23, 53]. Analytics systems aggregate raw data such as click logs into insightful statistics. Machine learning systems extract models and properties (e.g., the similarities of movies) from training data (e.g., historical movie ratings) using advanced algorithms. This derived data can recursively derive more data, such as a recommendation system predicting a user's rating of a movie based on movie similarities. In short, a piece of raw data in today's systems often goes through a series of computations, "creeping" into many places and appearing in many forms. The data, computations, and derived data together form a complex data propagation network that we call the data's *lineage*.

For a variety of reasons, users want a system to forget certain sensitive data and its complete lineage. Consider privacy first. After Facebook changed its privacy policy, many users deleted their accounts and the associated data [69]. The iCloud photo hacking incident [8] led to online articles teaching users how to completely delete iOS photos including the backups [79]. New privacy research revealed that machine learning models for personalized warfarin dosing leak patients' genetic markers [43], and a small set of statistics on genetics and diseases suffices to identify individuals [78]. Users unhappy with these newfound risks naturally want their data and its influence on the models and statistics to be completely forgotten. System operators or service providers have strong incentives to honor users' requests to forget data, both to keep users happy and to comply with the law [72]. For instance, Google had removed 171,183 links [50] by October 2014 under the "right to be forgotten" ruling of the highest court in the European Union.

Security is another reason that users want data to be forgotten. Consider anomaly detection systems. The security of these systems hinges on the model of normal behaviors extracted from the training data. By polluting[1] the training data, attackers pollute the model, thus compromising security. For instance, Perdisci et al. [56] show that PolyGraph [55], a worm detection engine, fails to generate useful worm signatures if the training data is injected with well-crafted fake network flows. Once the polluted data is identified, the system must completely forget the data and its lineage to regain security.

Usability is a third reason. Consider the recommendation or prediction system Google Now [7]. It infers a user's preferences from her search history, browsing history, and other analytics. It then pushes recommendations, such as news about a show, to the user. Noise or incorrect entries in analytics can seriously degrade the quality of the recommendation. One of our lab members experienced this problem first-hand. He loaned his laptop to a friend who searched for a TV show ("Jeopardy!") on Google [1]. He then kept getting news about this show on his phone, even after he deleted the search record from his search history.

We believe that systems must be designed under the core principle of completely and quickly forgetting sensitive data and its lineage for restoring privacy, security, and usability. Such *forgetting systems* must carefully track data lineage even across statistical processing or machine learning, and make this lineage visible to users. They let users specify

[1] In this paper, we use the term *pollute* [56] instead of *poison* [47, 77].

DOI 10.1109/SP.2015.35

the data to forget with different levels of granularity. For instance, a privacy-conscious user who accidentally searches for a sensitive keyword without concealing her identity can request that the search engine forget that particular search record. These systems then remove the data and revert its effects so that all future operations run as if the data had never existed. They collaborate to forget data if the lineage spans across system boundaries (e.g., in the context of web mashup services). This collaborative forgetting potentially scales to the entire Web. Users trust forgetting systems to comply with requests to forget, because the aforementioned service providers have strong incentives to comply, but other trust models are also possible. The usefulness of forgetting systems can be evaluated with two metrics: how completely they can forget data (*completeness*) and how quickly they can do so (*timeliness*). The higher these metrics, the better the systems at restoring privacy, security, and usability.

We foresee easy adoption of forgetting systems because they benefit both users and service providers. With the flexibility to request that systems forget data, users have more control over their data, so they are more willing to share data with the systems. More data also benefit the service providers, because they have more profit opportunities services and fewer legal risks. In addition, we envision forgetting systems playing a crucial role in emerging data markets [3, 40, 61] where users trade data for money, services, or other data because the mechanism of forgetting enables a user to cleanly cancel a data transaction or rent out the use rights of her data without giving up the ownership.

Forgetting systems are complementary to much existing work [55, 75, 80]. Systems such as Google Search [6] can forget a user's raw data upon request, but they ignore the lineage. Secure deletion [32, 60, 70] prevents deleted data from being recovered from the storage media, but it largely ignores the lineage, too. Information flow control [41, 67] can be leveraged by forgetting systems to track data lineage. However, it typically tracks only direct data duplication, not statistical processing or machine learning, to avoid taint explosion. Differential privacy [75, 80] preserves the privacy of each individual item in a data set *equally and invariably* by restricting accesses only to the whole data set's statistics fuzzed with noise. This restriction is at odds with today's systems such as Facebook and Google Search which, authorized by billions of users, routinely access personal data for accurate results. Unsurprisingly, it is impossible to strike a balance between utility and privacy in state-of-the-art implementations [43]. In contrast, forgetting systems aim to *restore privacy on select data*. Although private data may still propagate, the lineage of this data within the forgetting systems is carefully tracked and removed completely and in a timely manner upon request. In addition, this fine-grained data removal caters to an individual user's privacy consciousness and the data item's sensitivity. Forgetting systems conform to the trust and usage models of today's systems, representing a more practical privacy vs utility tradeoff. Researchers also proposed mechanisms to make systems more robust against training data pollution [27, 55].

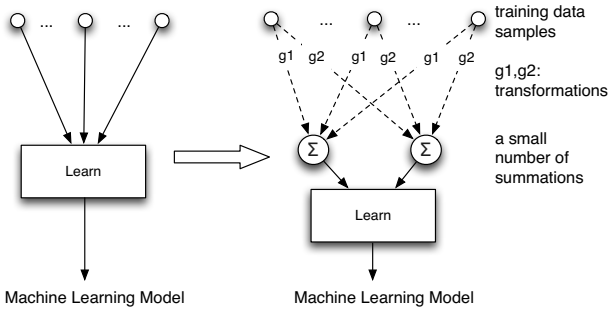

Fig. 1: Unlearning idea. Instead of making a model directly depend on each training data sample (left), we convert the learning algorithm into a summation form (right). Specifically, each summation is the sum of transformed data samples, where the transformation functions g_i are efficiently computable. There are only a small number of summations, and the learning algorithm depends only on summations. To forget a data sample, we simply update the summations and then compute the updated model. This approach is asymptotically much faster than retraining from scratch.

However, despite these mechanisms (and the others discussed so far such as differential privacy), users may still request systems to forget data due to, for example, policy changes and new attacks against the mechanisms [43, 56]. These requests can be served only by forgetting systems.

B. Machine Unlearning

While there are numerous challenges in making systems forget, this paper focuses on one of the most difficult challenges: making machine learning systems forget. These systems extract features and models from training data to answer questions about new data. They are widely used in many areas of science [25, 35, 37, 46, 55, 63–65]. To forget a piece of training data completely, these systems need to revert the effects of the data on the extracted features and models. We call this process *machine unlearning*, or *unlearning* for short.

A naïve approach to unlearning is to retrain the features and models from scratch after removing the data to forget. However, when the set of training data is large, this approach is quite slow, increasing the timing window during which the system is vulnerable. For instance, with a real-world data set from Huawei (see §VII), it takes Zozzle [35], a JavaScript malware detector, over a day to retrain and forget a polluted sample.

We present a general approach to efficient unlearning, without retraining from scratch, for a variety of machine learning algorithms widely used in real-world systems. To prepare for unlearning, we transform learning algorithms in a system to a form consisting of a small number of summations [33]. Each summation is the sum of some efficiently computable transformation of the training data samples. The learning algorithms depend only on the summations, not individual data. These summations are saved together with the trained model. (The rest of the system may still ask for individual data and there is no injected noise as there is in differential privacy.) Then, in the unlearning process, we subtract the data to forget

from each summation, and then update the model. As Figure 1 illustrates, forgetting a data item now requires recomputing only a small number of terms, asymptotically faster than retraining from scratch by a factor equal to the size of the training data set. For the aforementioned Zozzle example, our unlearning approach takes only less than a second compared to a day for retraining. It is general because the summation form is from statistical query (SQ) learning [48]. Many machine learning algorithms, such as naïve Bayes classifiers, support vector machines, and k-means clustering, can be implemented as SQ learning. Our approach also applies to all stages of machine learning, including feature selection and modeling.

We evaluated our unlearning approach on four diverse learning systems including (1) LensKit [39], an open-source recommendation system used by several websites for conference [5], movie [14], and book [4] recommendations; (2) an independent re-implementation of Zozzle, the aforementioned closed-source JavaScript malware detector whose algorithm was adopted by Microsoft Bing [42]; (3) an open-source online social network (OSN) spam filter [46]; and (4) PJScan, an open-source PDF malware detector [51]. We also used real-world workloads such as more than 100K JavaScript malware samples from Huawei. Our evaluation shows:

- All four systems are prone to attacks targeting learning. For LensKit, we reproduced an existing privacy attack [29]. For each of the other three systems, because there is no known attack, we created a new, practical data pollution attack to decrease the detection effectiveness. One particular attack requires careful injection of multiple features in the training data set to mislead feature selection and model training (see §VII).
- Our unlearning approach applies to all learning algorithms in LensKit, Zozzle, and PJScan. In particular, enabled by our approach, we created the first efficient unlearning algorithm for normalized cosine similarity [37, 63] commonly used by recommendation systems (e.g., LensKit) and for one-class support vector machine (SVM) [71] commonly used by classification/anomaly detection systems (e.g., PJScan uses it to learn a model of malicious PDFs). We show analytically that, for all these algorithms, our approach is both complete (completely removing a data sample's lineage) and timely (asymptotically much faster than retraining). For the OSN spam filter, we leveraged existing techniques for unlearning.
- Using real-world data, we show empirically that unlearning prevents the attacks and the speedup over retraining is often huge, matching our analytical results.
- Our approach is easy to use. It is straightforward to modify the systems to support unlearning. For each system, we modified from 20 – 300 lines of code, less than 1% of the system.

C. Contributions and Paper Organization

This paper makes four main contributions:

- The concept of forgetting systems that restore privacy, security, and usability by forgetting data lineage completely and quickly;
- A general unlearning approach that converts learning algorithms into a summation form for efficiently forgetting data lineage;
- An evaluation of our approach on real-world systems/algorithms demonstrating that it is practical, complete, fast, and easy to use; and
- The practical data pollution attacks we created against real-world systems/algorithms.

While prior work proposed incremental machine learning for several specific learning algorithms [31, 62, 73], the key difference in our work is that we propose a general efficient unlearning approach applicable to any algorithm that can be converted to the summation form, including some that currently have no incremental versions, such as normalized cosine similarity and one-class SVM. In addition, our unlearning approach handles all stages of learning, including feature selection and modeling. We also demonstrated our approach on real systems.

Our unlearning approach is inspired by prior work on speeding up machine learning algorithms with MapReduce [33]. We believe we are the first to establish the connection between unlearning and the summation form. In addition, we are the first to convert non-standard real-world learning algorithms such as normalized cosine similarity to the summation form. The conversion is complex and challenging (see §VI). In contrast, the prior work converts nine standard machine learning algorithms using only simple transformations.

The rest of the paper is organized as follows. In §II, we present some background on machine learning systems and the extended motivation of unlearning. In §III, we present the goals and work flow of unlearning. In §IV, we present the core approach of unlearning, i.e., transforming a system into the summation form, and its formal backbone. In §V, we overview our evaluation methodology and summarize results. In §VI–§IX, we report detailed case studies on four real-world learning systems. In §X and §XI, we discuss some issues in unlearning and related work, and in §XII, we conclude.

II. Background and Adversarial Model

This section presents some background on machine learning (§II-A) and the extended motivation of unlearning (§II-B).

A. Machine Learning Background

Figure 2 shows that a general machine learning system with three processing stages.

- Feature selection. During this stage, the system selects, from all features of the training data, a set of features most crucial for classifying data. The selected feature set is typically small to make later stages more accurate and efficient. Feature selection can be (1) manual where system builders carefully craft the feature set or (2) automatic where the system runs some learning algorithms

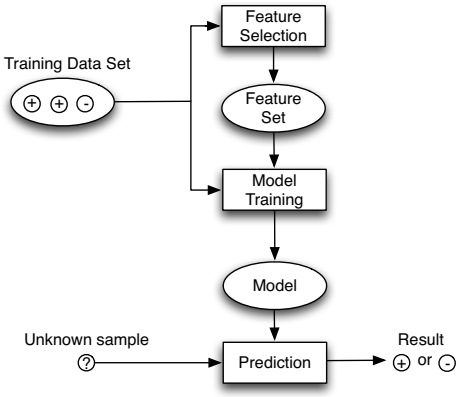

Fig. 2: A General Machine Learning System. Given a set of training data including both malicious (+) and benign (−) samples, the system first selects a set of features most crucial for classifying data. It then uses the training data to construct a model. To process an unknown sample, the system examines the features in the sample and uses the model to predict the sample as malicious or benign. The lineage of the training data thus flows to the feature set, the model, and the prediction results. An attacker can feed different samples to the model and observe the results to steal private information from every step along the lineage, including the training data set (*system inference* attack). She can pollute the training data and subsequently every step along the lineage to alter prediction results (*training data pollution* attack).

such as clustering and chi-squared test to compute how crucial the features are and select the most crucial ones.

- Model training. The system extracts the values of the selected features from each training data sample into a feature vector. It feeds the feature vectors and the malicious or benign labels of all training data samples into some machine learning algorithm to construct a succinct model.
- Prediction. When the system receives an unknown data sample, it extracts the sample's feature vector and uses the model to predict whether the sample is malicious or benign.

Note that a learning system may or may not contain all three stages, work with labeled training data, or classify data as malicious or benign. We present the system in Figure 2 because it matches many machine learning systems for security purposes such as Zozzle. Without loss of generality, we refer to this system as an example in the later sections of the paper.

B. Adversarial Model

To further motivate the need for unlearning, we describe several practical attacks in the literature that target learning systems. They either violate privacy by inferring private information in the trained models (§II-B1), or reduce security by polluting the prediction (detection) results of anomaly detection systems (§II-B2).

1) System Inference Attacks: The training data sets, such as movie ratings, online purchase histories, and browsing histories, often contain private data. As shown in Figure 2, the private data lineage flows through the machine learning algorithms into the feature set, the model, and the prediction results. By exploiting this lineage, an attacker gains an opportunity to infer private data by feeding samples into the system and observing the prediction results. Such an attack is called a *system inference* attack [29].[2]

Consider a recommendation system that uses *item-item collaborative filtering* which learns item-item similarities from users' purchase histories and recommends to a user the items most similar to the ones she previously purchased. Calandrino et al. [29] show that once an attacker learns (1) the item-item similarities, (2) the list of recommended items for a user before she purchased an item, and (3) the list after, the attacker can accurately infer what the user purchased by essentially inverting the computation done by the recommendation algorithm. For example, on LibraryThing [12], a book cataloging service and recommendation engine, this attack successfully inferred six book purchases per user with 90% accuracy for over one million users!

Similarly, consider a personalized warfarin dosing system that guides medical treatments based on a patient's genotype and background. Fredrikson et al. [43] show that with the model and some demographic information about a patient, an attacker can infer the genetic markers of the patient with accuracy as high as 75%.

2) Training Data Pollution Attacks: Another way to exploit the lineage in Figure 2 is using training data pollution attacks. An attacker injects carefully polluted data samples into a learning system, misleading the algorithms to compute an incorrect feature set and model. Subsequently, when processing unknown samples, the system may flag a big number of benign samples as malicious and generate too many false positives, or it may flag a big number of malicious as benign so the true malicious samples evade detection.

Unlike system inference in which an attacker exploits an easy-to-access public interface of a learning system, data pollution requires an attacker to tackle two relatively difficult issues. First, the attacker must trick the learning system into including the polluted samples in the training data set. There are a number of reported ways to do so [54, 56, 77]. For instance, she may sign up as a crowdsourcing worker and intentionally mislabel benign emails as spams [77]. She may also attack the honeypots or other baiting traps intended for collecting malicious samples, such as sending polluted emails to a spamtrap [17], or compromising a machine in a honeynet and sending packets with polluted protocol header fields [56].

Second, the attacker must carefully pollute enough data to mislead the machine learning algorithms. In the crowdsourcing case, she, the administrator of the crowdsourcing sites, directly pollutes the labels of some training data [77]. 3% mislabeled training data turned out to be enough to significantly decrease detection efficacy. In the honeypot cases [17, 56], the attacker cannot change the labels of the polluted data samples because the honeypot automatically labels them as malicious. However,

[2]In this paper, we use *system inference* instead of *model inversion* [43].

she controls what features appear in the samples, so she can inject benign features into these samples, misleading the system into relying on these features for detecting malicious samples. For instance, Nelson et al. injected words that also occur in benign emails into the emails sent to a spamtrap, causing a spam detector to classify 60% of the benign emails as spam. Perdisci et al. injected many packets with the same randomly generated strings into a honeynet, so that true malicious packets without these strings evade detection.

III. Overview

This section presents the goals (§III-A) and work flow (§III-B) of machine learning.

A. Unlearning Goals

Recall that forgetting systems have two goals: (1) completeness, or how completely they can forget data; and (2) timeliness, or how quickly they can forget. We discuss what these goals mean in the context of unlearning.

1) Completeness: Intuitively, completeness requires that once a data sample is removed, all its effects on the feature set and the model are also cleanly reversed. It essentially captures how consistent an unlearned system is with the system that has been retrained from scratch. If, for every possible sample, the unlearned system gives the same prediction result as the retrained system, then an attacker, operator, or user has no way of discovering that the unlearned data and its lineage existed in the system by feeding input samples to the unlearned system or even observing its features, model, and training data. Such unlearning is complete. To empirically measure completeness, we quantify the percentage of input samples that receive the same prediction results from both the unlearned and the retrained system using a representative test data set. The higher the percentage, the more complete the unlearning. Note that completeness does not depend on the correctness of prediction results: an incorrect but consistent prediction by both systems does not decrease completeness.

Our notion of completeness is subject to such factors as how representative the test data set is and whether the learning algorithm is randomized. In particular, given the same training data set, the same randomized learning algorithm may compute different models which subsequently predict differently. Thus, we consider unlearning complete as long as the unlearned system is consistent with one of the retrained systems.

2) Timeliness: Timeliness in unlearning captures how much faster unlearning is than retraining at updating the features and the model in the system. The more timely the unlearning, the faster the system is at restoring privacy, security, and usability. Analytically, unlearning updates only a small number of summations and then runs a learning algorithm on these summations, whereas retraining runs the learning algorithm on the entire training data set, so unlearning is asymptotically faster by a factor of the training data size. To empirically measure timeliness, we quantify the speedup of unlearning over retraining. Unlearning does not replace retraining. Unlearning works better when the data to forget is small compared to the training set. This case is quite common. For instance, a single user's private data is typically small compared to the whole training data of all users. Similarly, an attacker needs only a small amount of data to pollute a learning system (e.g., 1.75% in the OSN spam filter [46] as shown in §VIII). When the data to forget becomes large, retraining may work better.

B. Unlearning Work Flow

Given a training data sample to forget, unlearning updates the system in two steps, following the learning process shown in Figure 2. First, it updates the set of selected features. The inputs at this step are the sample to forget, the old feature set, and the summations previously computed for deriving the old feature set. The outputs are the updated feature set and summations. For example, Zozzle selects features using the chi-squared test, which scores a feature based on four counts (the simplest form of summations): how many malicious or benign samples contain or do not contain this feature. To support unlearning, we augmented Zozzle to store the score and these counts for each feature. To unlearn a sample, we update these counts to exclude this sample, re-score the features, and select the top scored features as the updated feature set. This process does not depend on the training data set, and is much faster than retraining which has to inspect each sample for each feature. The updated feature set in our experiments is very similar to the old one with a couple of features removed and added.

Second, unlearning updates the model. The inputs at this step are the sample to forget, the old feature set, the updated feature set, the old model, and the summations previously computed for deriving the old model. The outputs are the updated model and summations. If a feature is removed from the feature set, we simply splice out the feature's data from the model. If a feature is added, we compute its data in the model. In addition, we update summations that depend on the sample to forget, and update the model accordingly. For Zozzle which classifies data as malicious or benign using naïve Bayes, the summations are probabilities (e.g., the probability that a training data sample is malicious given that it contains a certain feature) computed using the counts recorded in the first step. Updating the probabilities and the model is thus straightforward, and much faster than retraining.

IV. Unlearning Approach

As previously depicted in Figure 1, our unlearning approach introduces a layer of a small number of summations between the learning algorithm and the training data to break down the dependencies. Now, the learning algorithm depends only on the summations, each of which is the sum of some efficiently computable transformations of the training data samples. Chu et al. [33] show that many popular machine learning algorithms, such as naïve Bayes, can be represented in this form. To remove a data sample, we simply remove the transformations of this data sample from the summations that depend on this sample, which has $O(1)$ complexity, and

compute the updated model. This approach is asymptotically faster than retraining from scratch.

More formally, the summation form follows statistical query (SQ) learning [48]. SQ learning forbids a learning algorithm from querying individual training data samples. Instead, it permits the algorithm to query only statistics about the training data through an oracle. Specifically, the algorithm sends a function $g(x, l_x)$ to the oracle where x is a training data sample, l_x the corresponding label, and g an efficiently computable function. Then, the oracle answers an estimated expectation of $g(x, l_x)$ over all training data. The algorithm repeatedly queries the oracle, potentially with different g-functions, until it terminates.

Depending on whether all SQs that an algorithm issues are determined upfront, SQ learning can be non-adaptive (all SQs are determined upfront before the algorithm starts) or adaptive (later SQs may depend on earlier SQ results). These two different types of SQ learning require different ways to unlearn, described in the following two subsections.

A. Non-adaptive SQ Learning

A non-adaptive SQ learning algorithm must determine all SQs upfront. It follows that the number of these SQs is constant, denoted m, and the transformation g-functions are fixed, denoted g_1, g_2, ..., g_m. We represent the algorithm in the following form:

$$Learn(\sum_{x_i \in X} g_1(x_i, l_{x_i}), \sum_{x_i \in X} g_2(x_i, l_{x_i}), ..., \sum_{x_i \in X} g_m(x_i, l_{x_i}))$$

where x_i is a training data sample and l_{x_i} its label. This form encompasses many popular machine learning algorithms, including linear regression, chi-squared test, and naïve Bayes.

With this form, unlearning is as follows. Let G_k be $\sum g_k(x_i, l_{x_i})$. All G_ks are saved together with the learned model. To unlearn a data sample x_p, we compute G'_k as $G_k - g_k(x_p, l_{x_p})$. The updated model is thus

$$Learn(G_1 - g_1(x_p, l_{x_p}), G_2 - g_2(x_p, l_{x_p}), ..., G_m - g_m(x_p, l_{x_p}))$$

Unlearning on a non-adaptive SQ learning algorithm is complete because this updated model is identical to

$$Learn(\sum_{i \neq p} g_1(x_i, l_{x_i}), \sum_{i \neq p} g_2(x_i, l_{x_i}), ..., \sum_{i \neq p} g_m(x_i, l_{x_i}))$$

the model computed by retraining on the training data excluding x_p. For timeliness, it is also much faster than retraining because (1) computing G'_k is easy: simply subtract $g_k(x_p, l_{x_p})$ from G_k and (2) there are only a constant number of summations G_k.

We now illustrate how to convert a non-adaptive SQ learning algorithm into this summation form using naïve Bayes as an example. Given a sample with features F_1, F_2, ..., and F_k, naïve Bayes classifies the sample with label L if $P(L|F_1, ..., F_k)$, the conditional probability of observing label L on a training data sample with all these features, is bigger than this conditional probability for any other label. This conditional probability is computed using Equation 1.

$$P(L|F_1, ..., F_k) = \frac{P(L)\prod_{i=0}^{k} P(F_i|L)}{\prod_{i=0}^{k} P(F_i)} \quad (1)$$

We now convert each probability term P in this equation into summations. Consider $P(F_i|L)$ as an example. It is computed by taking (1) the number of training data samples with feature F_i and label L, denoted $N_{F_i,L}$, and dividing by (2) the number of training data samples with label L, denoted N_L. Each counter is essentially a very simple summation of a function that returns 1 when a sample should be counted and 0 otherwise. For instance, N_L is the sum of an indicator function $g_L(x, l_x)$ that returns 1 when l_x is L and 0 otherwise. Similarly, all other probability terms are computed by dividing the corresponding two counters. $P(L)$ is the division of N_L over the total number of samples, denoted N. $P(F_i)$ is the division of the number of training data samples with the feature F_i, denoted N_{F_i}, over N.

To unlearn a sample, we simply update these counters and recompute the probabilities. For instance, suppose the training sample to unlearn has label L and one feature F_j. After unlearning, $P(F_j|L)$ becomes $\frac{N_{F_jL}-1}{N_L-1}$, and all other $P(F_i|L)$s become $\frac{N_{F_iL}}{N_L-1}$. $P(L)$ becomes $\frac{N_L-1}{N}$. $P(F_j)$ becomes $\frac{N_{F_j}-1}{N-1}$, and all other $P(F_i)$s become $\frac{N_{F_i}}{N-1}$.

B. Adaptive SQ Learning

An adaptive SQ learning algorithm issues its SQs iteratively on the fly, and later SQs may depend on results of earlier ones. (Non-adaptive SQ learning is a special form of adaptive SQ learning.) Operationally, adaptive SQ learning starts by selecting an initial state s_0, randomly or heuristically. At state s_j, it determines the transformation functions in the SQs based on the current state, sends the SQs to the oracle, receives the results, and learns the next state s_{j+1}. It then repeats until the algorithm converges. During each iteration, the current state suffices for determining the transformation functions because it can capture the entire history starting from s_0. We represent these functions in each state s_j as $g_{s_j,1}$, $g_{s_j,2}$, ..., $g_{s_j,m}$. Now, the algorithm is in the following form:

(1) s_0: initial state;

(2) $s_{j+1} = Learn(\sum_{x_i \in X} g_{s_j,1}(x_i, l_{x_i}), \sum_{x_i \in X} g_{s_j,2}(x_i, l_{x_i}), ..., \sum_{x_i \in X} g_{s_j,m}(x_i, l_{x_i}))$;

(3) Repeat (2) until the algorithm converges.

The number of iterations required for the algorithm to converge depends on the algorithm, the initial state selected, and the training data. Typically the algorithm is designed to robustly converge under many scenarios. This adaptive form of SQ learning encompasses many popular machine learning algorithms, including gradient descent, SVM, and k-means.

Unlearning this adaptive form is more changing than non-adaptive because, even if we restart from the same initial state,

if the training data sample to forget changes one iteration, all subsequent iterations may deviate and require computing from scratch. Fortunately, our insight is that, after removing a sample, the previously converged state often becomes only slightly out of convergence. Thus, unlearning can simply "resume" the iterative learning algorithm from this state on the updated training data set, and it should take much fewer iterations to converge than restarting from the original or a newly generated initial state.

Operationally, our adaptive unlearning approach works as follows. Given the converged state S computed on the original training data set, it removes the contributions of the sample to forget from the summations that $Learn$ uses to compute S, similar to unlearning the non-adaptive form. Let the resultant state be S'. Then, it checks whether S' meets the algorithm's convergence condition. If not, it sets S' as the initial state and runs the iterative learning algorithm until it converges.

We now discuss the completeness of our adaptive unlearning approach in three scenarios. First, for algorithms such as SVM that converge at only one state, our approach is complete because the converged state computed from unlearning is the same as that retrained from scratch. Second, for algorithms such as k-means that converge at multiple possible states, our approach is complete if the state S' is a possible initial state selected by the algorithm (e.g., the algorithm selects initial state randomly). A proof sketch is as follows. Since S' is a possible initial state, there is one possible retraining process that starts from S' and reaches a new converged state. At every iteration of this retraining process, the new state computed by $Learn$ is identical to the state computed in the corresponding iteration in unlearning. Thus, they must compute the same exact converged state, satisfying the completeness goal (§III-A1). Third, our approach may be incomplete if (a) S' cannot be a possible initial state (e.g., the algorithm selects initial state using a heuristic that happens to rule out S') or (b) the algorithm does not converge or converges at a state different than all possible states retraining converges to. We expect these scenarios to be rare because adaptive algorithms need to be robust anyway for convergence during normal operations.

Our adaptive unlearning approach is also timely. The speedup over retraining is twofold. First, unlearning is faster at computing the summations if there are old results of the summations to use. For example, it updates the state S by removing contributions of the removed sample. Second, unlearning starts from an almost-converged state, so it needs fewer iterations to converge than retraining. In practice, we expect that the majority of the speedup comes from the reduced number of iterations. For instance, our evaluation shows that, on average, PJScan retraining needs 42 iterations while unlearning only 2.4, a speedup of over 17x (§IX). The implication is that, in principle, our adaptive unlearning approach should speed up any robust iterative machine learning algorithm, even if the algorithm does not follow SQ learning. In practice, however, very few practical learning algorithms cannot be converted to the adaptive SQ learning form. Specifically, many machine learning problems can be cast as optimization problems, potentially solvable using gradient descent, an adaptive SQ learning algorithm. Thus, we used adaptive SQ learning to represent the more general class of iterative algorithms in our discussion.

Now, we illustrate how to convert an adaptive SQ learning algorithm into the summation form using k-means clustering as an example. K-means clustering starts from an initial set of randomly selected cluster centers, $c_1, ..., c_k$, assigns each data point to a cluster whose center has the shortest Euclidean distance to the point, and then updates each c_i based on the mean value of all the data points in its cluster. It repeats this assignment until the centers no longer change.

To support unlearning, we convert the calculation of each c_i into summations. Because k-means clustering is unsupervised, labels are not involved in the following discussion. Let us define $g_{c_i,j}(x)$ as a function that outputs x when the distance between x and c_i is minimum, and otherwise 0; and define $g'_j(x)$ as a function that outputs 1 when the distance between x and c_i is minimum, and otherwise 0. Now, the new c_i in the $j+1$ iteration equals $\frac{\sum_{x \in X} g_{c_i,j}(x)}{\sum_{x \in X} g'_{c_i,j}(x)}$.

To unlearn a sample x_p, we update $\sum_{x \in X} g_{c_i,j}(x)$ and $\sum_{x \in X} g'_{c_i,j}(x)$ by subtracting $g_{c_i,j}(x_p)$ and $g'_{c_i,j}(x_p)$ from the summations. Then, we continue the iteration process until the algorithm converges.

V. Evaluation Methodology and Result Summary

To evaluate our unlearning approach, we chose four real-world systems whose purposes range from recommendation to malicious PDF detection. They include both open-source and closed-source ones, covering six different learning algorithms. We focused on real-world systems because they do not use just the standard learning algorithms studied by Chu et al. [33] or have just one learning stage [73]. We believe this diverse set of programs serves as a representative benchmark suite for our approach. We briefly describe each evaluated system below.

- LensKit [39] is an open-source recommendation system used by Confer (a conference recommendation website), MovieLens (a movie recommendation website), and BookLens (a book recommendation website). LensKit's default recommendation algorithm is a flexible, fast item-item collaborative filtering algorithm from Sarwar et al. [63] and Deshpande et al [37]. This algorithm first infers similarities of every two items based their user ratings. Then, if a user likes another item, it recommends the most similar items to the user.
- Zozzle [35] is a closed-source JavaScript malware detector. Its learning has two stages: (1) a chi-squared test to select features and (2) a naïve Bayes classifier to classify JavaScript programs into malicious or benign. Its algorithms have been adopted by Microsoft Bing [42]. Since Zozzle is closed-source, we obtained an independent re-implementation [21]. This re-implementation was also used in our prior work [30].
- The system described in Gao et al. [46] is an open-source OSN spam filter. It uses manually selected features and a

TABLE I: Summary of Unlearning Results. Note that m is the number of items, n is the number of users, q is the number of features, N is the size of training data set, p and l are numbers between 2 and 3, and k is a number between 0 and 1. See the next four sections for more details on these symbols.

System	Attack	Summation?	Analytical Results				Empirical Results				
			Complete-ness	Unlearn Speed	Retrain Speed	Speedup	Complete-ness	Unlearn Speed	Retrain Speed	Speedup	Modification LoC (%)
LensKit	Old	✓	100%	$O(m^2)$	$O(nm^2)$	$O(n)$	100%	45s	4min56s	6.57	302 (0.3%)
Zozzle	New	✓	100%	$O(q)$	$O(Nq)$	$O(N)$	100%	987ms	1day2hours	9.5×10^4	21 (0.4%)
OSNSF	New	✗	<100%	$O(logN)$	$O(NlogN)$	$O(N)$	99.4%	21.5ms	30mins	8.4×10^4	33 (0.8%)
PJScan	New	✓	100%	$O(N^p)$	$O(N^l)$	$O(N^k)$	100%	156ms	157ms	1	30 (0.07%)

distance-based algorithm to cluster OSN wall posts into a moderate number of clusters, and then builds a C4.5 decision tree [58] for classifying posts into "ham" or spam. The clustering helps to make the classifying step real time. For brevity, we refer to this system as OSNSF in the remainder of this paper.

- PJScan [51] is an open-source PDF malware detector tool. It uses one-class SVM which takes only malicious samples as the training data to classify PDFs.

For each system, we focused our evaluation on the following four research questions.

1) Is the system vulnerable to any attack exploiting machine learning? We answered this question by either reproducing an existing attack if there is one or constructing a new attack otherwise. To ensure that the attack is practical, we used real-world workloads and typically injected different portions of polluted data to observe how effective the attack became.
2) Does our unlearning approach apply to the different machine learning algorithms and stages of the system? We answered this question by understanding the algorithms in the system and revising them based on our approach. Analytically, we also computed the unlearning completeness and timeliness (i.e., asymptotic speedup over retraining).
3) Empirically, how does our unlearning approach perform? These empirical results are crucial to understand that the actual performance of our approach matches the analytical performance. We answered this question by running the system on real-world data such as a Facebook data set with over one million wall posts. We measured completeness by comparing prediction results on test data sets before and after unlearning; we additionally studied whether unlearning prevents the attack against the system. We measured timeliness by quantifying the actual speedup of unlearning of retraining.
4) Is it easy to modify the system to support unlearning? We answered this question by implementing our approach in the system and counting the lines of code that we modified.

Table I shows the summary of our results. All four evaluated systems turned out to be vulnerable. For LensKit, we reproduced an existing privacy attack [29]. For each of the other three systems, we created a new practical data pollution attack. Our unlearning approach applies to all learning algorithms in three systems, LensKit, Zozzle, and PJScan, with 100% completeness. For OSNSF, we leveraged existing techniques for unlearning and got less than 100% completeness. For all systems, the speedup of unlearning over retraining is often asymptotically as large as the size of the training data set. Empirically, using the largest real-world data sets we obtained, we show that unlearning was 100% complete for except that it was 99.4% for the OSNSF. It obtained up to $10^4\times$ speedup except for PJScan because its largest data set has only 65 PDFs, so the execution time was dominated by program start and shutdown not learning. These empirical results match the analytical ones. Lastly, modification to support unlearning for each system ranges from 20 – 300 lines of code (LoC), less than 1% of total LoC of the system. The next four sections describe these results in more detail for each system.

VI. Unlearning in LensKit

We start by describing LensKit's recommendation algorithm. Recall that it by default recommends items to users using item-item collaborative filtering [37, 63] that computes the similarity of every two items based on user ratings of the items because, intuitively, similar items should receive similar ratings from the same user. Operationally, LensKit starts by constructing a user-item matrix based on historical user ratings of items, where row i stores all ratings given by user i, and column j all ratings received by item j. Then, LensKit normalizes all ratings in the matrix to reduce biases across users and items. For instance, one user's average rating may be higher than another user's, but both should contribute equally to the final item-item similarity. Equation 2 shows the normalized rating a_{ij} for r_{ij}, user i's original rating of item j, where μ_i is the average of all ratings given by user i, η_j the average of all ratings received by item j, and g is the global average rating.

$$a_{ij} = \begin{cases} r_{ij}-\mu_i-\eta_j+g & when \quad rij \neq null \\ 0 & when \quad rij = null \end{cases} \tag{2}$$

Based on the normalized user-item rating matrix, LensKit computes an item-item similarity matrix within which the cell at row k and column l represents the similarity between items k and l. Specifically, as shown in Equation 3, it computes the cosine similarity between columns k and l in the user-item rating matrix, where $||\vec{x}||_2$ represents the Euclidean norm of $\vec{x}$, and $\vec{a}_{*,k}$ is a vector, $(a_{1k}, a_{1k}, ..., a_{nk})$, representing all the ratings received by item k.

$$sim(k,l) = \frac{\vec{a}_{*,k} \cdot \vec{a}_{*,l}}{||\vec{a}_{*,k}||_2 ||\vec{a}_{*,l}||_2} \quad (3)$$

Now, to recommend items to a user, LensKit computes the most similar items to the items previously rated by the user.

The workload that we use is a public, real-world data set from movie recommendation website MovieLens [14]. It has three subsets: (1) 100,000 ratings from 1,000 users on 1,700 movies, (2) 1 million ratings from 6,000 users on 4,000 movies, and (3) 10 million ratings from 72,000 users on 10,000 movies. We used LensKit's default settings in all our experiments.

A. The Attack – System Inference

Since there exists a prior system inference attack against recommendation systems [29], we reproduced this attack against LensKit and verified the effectiveness of the attack. As described by Calandrino et al. [29], the attacker knows the item-item similarity matrix and some items that the user bought from the past. To infer a newly bought item of the user, the attacker computes a delta matrix between the current item-item similarity matrix and the one without the item. Then, based on the delta matrix, the attacker could infer an initial list of items that might lead to the delta matrix, i.e., potential items that the user might have newly bought. Comparing the list of inferred items and the user's purchasing history, the attacker could infer the newly bought item. Following the attack steps, we first record the item-item similarity matrix of LensKit and one user's rating history. Then, we add one item to the user's rating history, compute the delta matrix and then successfully infer the added rating from the delta matrix and the user's rating history.

B. Analytical Results

To support unlearning in LensKit, we converted its recommendation algorithm into the summation form. Equation 4 shows this process. We start by substituting $\vec{a}_{*,k}$ and $\vec{a}_{*,l}$ in Equation 3 with their corresponding values in Equation 2 where n is the number of users and m the number of items, and expanding the multiplications. We then simplify the equation by substituting some terms using the five summations listed in Equation 5. The result shows that our summation approach applies to item-item recommendation using cosine similarity.

$$S_{kl} = \sum_{i=1}^{n} (r_{ik} - \mu_i)(r_{il} - \mu_i)$$
$$S_k = \sum_{i=1}^{n} (r_{ik} - \mu_i) \quad S_l = \sum_{i=1}^{n} (r_{il} - \mu_i) \quad (5)$$
$$S_{kk} = \sum_{i=1}^{n} (r_{ik} - \mu_i)^2 \quad S_{ll} = \sum_{i=1}^{n} (r_{il} - \mu_i)^2$$

We now discuss analytically the completeness and timeliness of unlearning in LensKit. To forget a rating in LensKit, we must update its item-item similarity matrix. To update the similarity between items k and l, we simply update all the summations in Equation 4, and then recompute $sim(k,l)$ using the summations. This unlearning process is 100% complete because it computes the same value of $sim(k,l)$ as recomputing from scratch following Equation 3. The asymptotic time to unlearn $sim(k,l)$ is only $O(1)$ because there is only a constant number of summations, each of which can be updated in constant time. Considering all m^2 pairs of items, the time complexity of unlearning is $O(m^2)$. In contrast, the time complexity of retraining from scratch is $O(nm^2)$ because recomputing $sim(k,l)$ following Equation 3 requires the dot-product of two vectors of size n. Thus, unlearning has a speedup factor of $O(n)$ over retraining. This speedup is quite huge because a recommendation system typically has much more users than items (e.g., Netflix's users vs movies).

Algorithm 1 Learning Stage Preparation in LensKit

Input:
All the users: 1 to n
All the items: 1 to m
Process:
1: Initializing all the variables to zero
2: **for** $i = 1$ to n **do**
3: **for** $j = 1$ to m **do**
4: **if** $r_{ij} \neq null$ **then**
5: $Sum_{\mu_i} \leftarrow Sum_{\mu_i} + r_{ij}$
6: $Count_{\mu_i}$++
7: $Sum_{\eta_j} \leftarrow Sum_{\eta_j} + r_{ij}$
8: $Count_{\eta_j}$++
9: $Sum_g \leftarrow Sum_g + r_{ij}$
10: $Count_g$++
11: **end if**
12: **end for**
13: $\mu_i \leftarrow Sum_{\mu_i}/Count_{\mu_i}$
14: **end for**
15: $g \leftarrow Sum_g/Count_g$
16: **for** $k = 1$ to m **do**
17: $\eta_k \leftarrow Sum_{\eta_k}/Count_{\eta_k}$
18: **for** $i = 1$ to n **do**
19: $S_k \leftarrow S_k + (r_{ik} - \mu_i)$
20: **end for**
21: **for** $l = 1$ to m **do**
22: **for** $i = 1$ to n **do**
23: $S_{kl} \leftarrow S_{kl} + (r_{ik} - \mu_i) * (r_{il} - \mu_i)$
24: **end for**
25: Calculate $sim(k,l)$
26: **end for**
27: **end for**

Now that we have shown mathematically how to convert LensKit's item-item similarity equation into the summation form and its analytical completeness and timeliness, we proceed to show algorithmically how to modify LensKit to support unlearning. While doing so is not difficult once Equation 4 is given, we report the algorithms we added to provide a complete picture of how to support unlearning in LensKit.

We added two algorithms to LensKit. Algorithm 1 runs during the learning stage of LensKit, which occurs when the system bootstraps or when the system operator decides to retrain from scratch. This algorithm computes the necessary summations for later unlearning. To compute the average rating of each user i (μ_i, line 13), it tracks the number of ratings given by the user ($Count_{\mu_i}$, line 6) and the sum of these ratings (Sum_{μ_i}, line 5). It similarly computes the average rating of each item k (η_k, line 17) by tracking the number of all ratings received by item k ($Count_{\eta_k}$, line 8) and the sum of these ratings (Sum_{η_k}, line 7). It computes the

$$
\begin{aligned}
sim(k,l) &= \frac{\sum_{i=1}^{n} a_{ik}a_{il}}{\sqrt{\sum_{i=1}^{n} a_{ik}^2 \sum_{i=1}^{n} a_{il}^2}} = \frac{\sum_{i=1}^{n} (r_{ik}-\mu_i-\eta_k+g)(r_{il}-\mu_i-\eta_l+g)}{\sqrt{\sum_{i=1}^{n} (r_{ik}-\mu_i-\eta_k+g)^2 \sum_{i=1}^{n} (r_{il}-\mu_i-\eta_l+g)^2}} \\
&= \frac{\sum_{i=1}^{n} (r_{ik}-\mu_i)(r_{il}-\mu_i) - \eta_k \sum_{i=1}^{n} (r_{il}-\mu_i) - \eta_l \sum_{i=1}^{n} (r_{ik}-\mu_i) - g(\eta_k+\eta_l)N + \eta_k\eta_l N + g^2 N + g \sum_{i=1}^{n} (r_{ik}+r_{il}-2\mu_i)}{\sqrt{\sum_{i=1}^{n} (r_{ik}-\mu_i)^2 - 2(\eta_k-g)\sum_{i=1}^{n} (r_{ik}-\mu_i) + (\eta_k-g)^2 N}\sqrt{\sum_{i=1}^{n} (r_{il}-\mu_i)^2 - 2(\eta_l-g)\sum_{i=1}^{n} (r_{il}-\mu_i) + (\eta_l-g)^2 N}} \\
&= \frac{S_{kl} - \eta_k S_l - \eta_l S_k + g(S_k+S_l) - g(\eta_k+\eta_l)N + \eta_k\eta_l N + g^2 N}{\sqrt{S_{kk} - 2(\eta_k-g)S_k + (\eta_k-g)^2 N}\sqrt{S_{ll} - 2(\eta_l-g)S_l + (\eta_l-g)^2 N}} \\
&= Learn(S_{kl}, S_k, S_l, S_{kk}, S_{ll}, g, \eta_k, \eta_l)
\end{aligned}
\tag{4}
$$

Algorithm 2 Unlearning Stage in LensKit

Input:
- u : the user who wants to delete a rating for an item ▷ *User u*
- t : the item, of which the user wants to delete the rating ▷ *Item t*
- r_{ut} : the original rating that the user gives ▷ *Rating r_{ut}*

Process:

1: $old\mu_u \leftarrow \mu_u$
2: $\mu_u \leftarrow (\mu_u * Count_{\mu_u} - r_{ut})/(Count_{\mu_u} - 1)$
3: $\eta_t \leftarrow (\eta_t * Count_{\eta_t} - r_{ut})/(Count_{\eta_t} - 1)$
4: $g \leftarrow (g * Count_g - r_{ut})/(Count_g - 1)$
5: $S_t \leftarrow S_t - (r_{ut} - old\mu_u)$
6: $S_t \leftarrow S_{tt} - (r_{ut} - old\mu_u) * (r_{ut} - old\mu_u)$
7: **for** $j = 1$ to m **do**
8: **if** $r_{uj} \neq null$ && $j \neq t$ **then**
9: $S_j \leftarrow S_j + old\mu_u - \mu_u$
10: $S_{jj} \leftarrow S_{jj} - (r_{uj} - old\mu_u) * (r_{uj} - old\mu_u) + (r_{uj} - \mu_u) * (r_{uj} - \mu_u)$
11: **end if**
12: **end for**
13: **for** $k = 1$ to m **do**
14: **for** $l = 1$ to m **do**
15: **if** $r_{uk} \neq null$ && $r_{ul} \neq null$ && $k \neq l$ **then**
16: **if** $j = t$ || $l = t$ **then**
17: $S_{kl} \leftarrow S_{kl} - (r_{uk} - old\mu_u) * (r_{ul} - old\mu_u)$
18: **else**
19: $S_{kl} \leftarrow S_{kl} - (r_{uk} - old\mu_u) * (r_{ul} - old\mu_u) + (r_{uk} - \mu_u) * (r_{ul} - \mu_u)$
20: **end if**
21: **end if**
22: Update $sim(k,l)$
23: **end for**
24: **end for**

average of all ratings (g, line 15) by tracking the total number of ratings ($Count_g$, line 10) and the sum of them (Sum_g, line 9). In addition, it computes additional summations S_k and S_{kl} (line 19 and 23) required by Equation 4. Once all summations are ready, it computes the similarity of each pair of items following Equation 4. It then stores the summations μ_i and $count_{\mu_i}$ for each user, η_j and $count_{\eta_j}$ for each item, g and $count_g$, S_k for each item, and S_{kl} for each pair of items for later unlearning.

Algorithm 2 is the core algorithm for unlearning in LensKit. To forget a rating, it updates all relevant summations and relevant cells in the item-item similarity matrix. Suppose that user u asks the system to forget a rating she gave about item t. Algorithm 2 first updates user u's average rating μ_i, item t's average rating η_j, and the global average rating g by multiplying the previous value of the average rating with the corresponding total number, subtracting the rating to forget r_{ut}, and dividing by the new total number (lines 1–3). It then updates item t's summations S_t and S_{tt} by subtracting the value contributed by r_{ut} which simplifies to the assignments shown on lines 4–5. It then updates S_j and S_{jj} (lines 6–11) for each of the other items j that received a rating from user m. Because the ratings given by the other users and their averages do not change, Algorithm 2 subtracts the old value contributed by user u and adds the updated value. Algorithm 2 updates S_{jk} similarly (lines 12–20). Finally, it recomputes $sim(j,k)$ based on updated summations following Equation 4 (line 21).

Note that while these algorithms require additional $n + m^2 + 2m$ space to store the summations, the original item-item recommendation algorithm already uses $O(nm)$ space for the user-item rating matrix and $O(m^2)$ space for the item-item similarity matrix. Thus, the asymptotic space complexity remains unchanged.

C. Empirical Results

To modify LensKit to support unlearning, we have inserted 302 lines of code into nine files spanning over three LensKit packages: lenskit-core, lenskit-knn, and lenskit-data-structures.

Empirically, we evaluated completeness using two sets of experiments. First, for each data subset, we randomly chose a rating to forget, ran both unlearning and retraining, and compared the recommendation results for each user and the item-item similarity matrices computed. We repeated this experiment ten times and verified that in all experiments, the recommendation results were always identical. In addition, the maximum differences between the corresponding similarities were less than 1.0×10^{-6}. These tiny differences were caused by imprecision in floating point arithmetic.

Second, we verified that unlearning successfully prevented the aforementioned system inference attack [29] from gaining any information about the forgotten rating. After unlearning, LensKit gave exactly the same recommendations as if the forgotten rating had never existed in the system. When we launched the attack, the delta matrices (§IV in [29]) used in the attack contained all zeros, so the attacker cannot infer anything from these matrices.

We evaluated timeliness by measuring the time it took to unlearn or retrain. We used all three data subsets and repeated each experiment three times. Table II shows the results. The first row shows the time of retraining, and the

TABLE II: Speedup of unlearning over retraining for LensKit. The time of retraining increases by the factor of the number of total ratings, and the overhead of unlearning increases by the factor of the number of total users.

	100K Ratings from 1,000 users & 1,700 items	1M Ratings from 6,000 users & 4,000 items	10M Ratings from 72,000 users & 10,000 items
Retraining	4.2s	30s	4min56s
Unlearning	931ms	6.1s	45s
Speedup	4.51	4.91	6.57

second row the time of unlearning, and the last row the speedup of unlearning over retraining. Unlearning consistently outperforms retraining. The speedup factor is less than the $O(n)$ analytical results because there are many empty ratings in the data set, i.e., a user does not give ratings for every movie. Therefore, the retraining speed is closer to $O(Nm)$, and the speedup factor is closer to $O(N/m)$, where N is the number of ratings and m is the number of users. For a larger data set, the speedup may be even larger. For example, IMDb contains 2,950,317 titles (including TV shows, movies, etc.) and 54 million registered users [9, 19], which may produce billions or even trillions of ratings. In that case, the unlearning may take several hours to complete, while the retraining may take several days.

VII. Unlearning in Zozzle

We start by describing Zozzle's JavaScript malware detection algorithm. Zozzle first extracts the structure of the JavaScript abstract syntax tree (AST) nodes and uses chi-squared test to select representative features. The chi-squared test is shown in Equation 6, where $N_{+,F}$ means the number of malicious samples with feature F, $N_{-,F}$ benign samples with F, $N_{+,\hat{F}}$ malicious samples without F, and $N_{-,\hat{F}}$ benign samples without F.

$$\chi^2 = \frac{(N_{+,F}N_{-,\hat{F}} - N_{+,\hat{F}}N_{-,F})^2}{(N_{+,F}+N_{+,\hat{F}})(N_{-,F}+N_{-,\hat{F}})(N_{+,F}+N_{-,F})(N_{+,\hat{F}}+N_{-,\hat{F}})} \quad (6)$$

Then, Zozzle trains a naïve Bayes classifier using the selected AST features, and then classifies an incoming JavaScript sample as malicious or benign. (The details of a naïve Bayes classifier have been shown before in Equation 1.)

Because Zozzle is closed-source, to avoid any bias, we ask Xiang Pan [21] to follow the original system [35] and re-implement Zozzle. The re-implemented Zozzle uses the Eclipse JavaScript development tools [11] to generate ASTs of JavaScript and extract the corresponding features into a MySQL database. Then, it performs chi-squared test with the same threshold as in the original system (10.83, which "corresponds with a 99.9% confidence that the values of feature presence and script classification are not independent" [35]) to select features for the feature set. It also implements its own naïve Bayes classifier with the same training steps.

To test Zozzle, we used the following workload. We crawled the top 10,000 Alexa web sites [2] to serve as the benign data set. In addition, from Huawei, we obtained 142,350 JavaScript malware samples, collected at their gateways or reported by their customers; all malware samples were manually verified by Huawei and cross-tested by multiple anti-virus software systems. We divided the whole data set equally into ten parts, nine of which are for training and the remaining one for testing. After training, Zozzle selected 2,398 features in total, out of which 1,196 are malicious features and 1,202 are benign features. The detection rate is shown in the first column of Table III. The re-implemented Zozzle achieved a 93.1% true positive rate and a 0.5% false positive rate, comparable to the original Zozzle system.

A. The Attack – Training Data Pollution

To perform data pollution and influence the detection of Zozzle, an attacker might craft malicious samples by injecting features that do not appear in any benign samples. In such case, those crafted malicious samples could be captured by Zozzle's ground-truth detector (such as Nozzle [59]), and included in the training data set. The injected features (such as an *if* statement with an unusual condition) in the crafted malicious samples can influence both the feature selection and the sample detection stage of Zozzle. In the feature selection stage, the injected features are likely to be selected, and influence existing malicious features so that they are less likely to be picked. In the sample detection stage, the injected features influence the decision for a true malicious sample.

First, because the injected features do not appear in benign samples but only malicious samples, in the chi-squared test (Equation 6), $N_{+,F}$ and $N_{-,\hat{F}}$ are large, and $N_{-,F}$ and $N_{+,\hat{F}}$ are small. Therefore, the feature selection process is likely to pick the injected features.

In addition, the injected features can make a real malicious feature that would have been selected – F_{real} – less likely to be selected. Because the attacker does not change any benign training sample or remove the F_{real} in existing malicious samples, but only add new samples, in the chi-squared test, $N_{+,F_{real}}$, $N_{-,F_{real}}$ and $N_{-,\hat{F}_{real}}$ remain the same, and $N_{+,\hat{F}_{real}}$ increases. Therefore, the feature selection process is less likely to pick up F_{real} as a malicious feature.

Second, the presence of an injected feature, F_{inject}, lowers the accuracy of the naïve Bayes classifier. Let us consider how the detection of a sample with one malicious feature, F_{mal}, is influenced in the presence of F_{inject} – i.e., how the value of $P(+|F_{mal})$ is lowered. Intuitively, since both F_{mal} and F_{inject} appear to be good indicators that a sample is malicious, Zozzle splits the weight it were to place on F_{mal} alone onto both F_{mal} and F_{inject}. Therefore, the weight on F_{mal} is lowered.

In addition, if a single F_{inject} cannot lower the accuracy of the naïve Bayes classifier significantly, the attacker could inject a large number of F_{inject} to further lower the accuracy. Because of the independence assumption in the naïve Bayes

classifier, the larger the number of F_{inject}, the smaller the weight of F_{mal}.

Now, we formally show how the accuracy of the naïve Bayes classifier is lowered. Before computing $P(+|F_{mal})$ – the probability of malice of samples with F_{mal}, let us take a first look at the computation of $P(+|F_{mal}, F_{inject})$, and then we deduce $P(+|F_{mal})$ based on $P(+|F_{mal}, F_{inject})$. Suppose, the attacker can inject samples with F_{mal} and another crafted feature, F_{inject}. As discussed, in the feature selection stage, F_{inject} is selected based on the chi-squared test. Now, we have Equation 7 for samples with F_{mal} and F_{inject}.

$$\begin{aligned} P(+|F_{mal}, F_{inject}) &= \frac{P(F_{mal}, F_{inject}|+)P(+)}{P(F_{mal}, F_{inject})} \\ &= \frac{P(F_{inject}|+)P(F_{mal}|+)P(+)}{P(F_{inject})P(F_{mal})} \\ &= \frac{P(F_{inject}|+)}{P(F_{inject})} P(+|F_{mal}) \end{aligned} \tag{7}$$

Through transformation (note that in Equation 7, naïve Bayes assumes that F_{mal} and F_{inject} are independent – i.e., derived from the independence assumption), we have Equation 8 for $P(+|F_{mal})$, the probability of malice for samples with just F_{mal}.

$$P(+|F_{mal}) = P(+|F_{mal}, F_{inject}) \frac{P(F_{inject})}{P(F_{inject}|+)} \tag{8}$$

In Equation 8, if $P(+|F_{mal})$ is below 0.5 and thus less than $P(-|F_{mal})$, Zozzle classifies samples with F_{mal} as benign $(-)$. The purpose of pollution is to make this happen. Because $\frac{P(F_{inject})}{P(F_{inject}|+)}$ is a value corresponding to the percentage of malicious samples in the training set and thus less than 1, $P(+|F_{mal})$ is lowered due to the existence of F_{inject}.

As discussed in last few paragraphs, the injection of one F_{inject} lowers $P(+|F_{mal})$. Now, we show that more injected features lower $P(+|F_{mal})$ even further. Suppose, the injection of one feature cannot decrease $P(+|F_{mal})$ in Equation 8 to a value below 0.5. An adversary can introduce enough number of F_{inject}, and as a generalization of Equation 8, we have Equation 9.

$$\begin{aligned} P(+|F_{mal}) = P(+|F_{mal}, F_{1,inject}, ..., F_{k,inject}) \\ \times \frac{P(F_{1,inject})}{P(F_{1,inject}|+)} \cdots \frac{P(F_{k,inject})}{P(F_{k,inject}|+)} \end{aligned} \tag{9}$$

Because each $\frac{P(F_{i,inject})}{P(F_{i,inject}|+)}$ is less than 1, if k is large enough, the attacker can eventually decrease $P(+|F_{mal})$ to a value below 0.5, and hence Zozzle classifies the sample with F_{mal} as benign. In summary, by injecting enough features, an adversary could subvert the detection decision of a malicious sample in Zozzle.

In practice, to pollute the training data, we inject five features that do not exist in any existing benign samples into 10% of the training set. We retrained the system with the same training set as well as the newly added samples with injected features, and the testing samples are still the same as those before pollution. The detection results are shown in the second column of Table III. Because of the injection of features, the true positive rate drops significantly from 93.1% to 37.8%. The false positive rate also drops a little bit from 0.5% to 0.3%, because more benign features are selected from the feature sets.

TABLE III: Zozzle's Detection Rate. The first column and the third column are exactly the same, illustrating that our unlearning is complete.

	Original	**Polluted**	**Unlearned**
True Positive	93.1%	37.8%	93.1%
False Positive	0.5%	0.3%	0.5%

B. Analytical and Empirical Results

Unlearning in Zozzle works as follows. First, the unlearning process groups all the data to forget together and extract corresponding features from the data to forget. Then, the chi values of all the features are updated. Since $N_{+,F}$, $N_{-,F}$, $N_{+,\hat{F}}$, and $N_{-,\hat{F}}$ in the chi value calculation (Equation 6) are counts of samples, or a summation of outputs of indicator functions, one can easily update the chi value of features. If the chi value of one feature cannot meet the threshold, the feature is removed. We reuse the feature selection process implemented by Zozzle to extract features from the data to forget, and then update the chi values stored in the database. Then, a new list of features is generated based on the new chi values. Second, the unlearning process updates all the conditional probability values related to updated features found in the first step. The detailed process has already been described in §IV. Note that because none of the aforementioned updates involve the size of training data set, the time complexity is $O(q)$, where q is the number of features.

Empirically, we added only 21 lines of code to support unlearning in Zozzle, i.e., updated all the chi values and conditional probability. Then, we evaluated the completeness of unlearning by removing all the crafted samples from Zozzle. The results show that the feature sets and conditional probabilities after pollution and unlearning are the same as those generated by unpolluted data, as if the whole pollution process does not exist. Further, the true positive and false positive after pollution and unlearning shown in the third column of Table III are the same as those without data pollution as expected.

Next, we evaluate the timeliness of unlearning. Because the size of training samples is huge, the overall learning process takes one day and two hours. In contrast, unlearning one training sample takes less than one second on average, and the speedup is 9.5×10^4. The overhead of unlearning is a linear function to the number of the samples to forget. As mentioned in §III-A2, when the number of samples to forget increases, the overhead of unlearning increases, but the overhead of retraining decreases. When the number of samples to forget exceeds 63% of the total training samples, the retraining process is faster than the unlearning technique

used in this section. In such an unlikely case, one may use retraining to achieve the unlearning task.

VIII. Unlearning in An OSN Spam Filter

We start by introducing how OSNSF [46] uses machine learning to filter OSN spams. In the training stage, OSNSF first clusters OSN messages based on the text shingling distance [26] and the URL elements of the messages, and then extracts the features of each cluster, such as cluster size, average social degree, and average time interval of messages. Next, the features of clusters are used to train a C4.5 decision tree[3] [58] for spam filtering. In the detection/filtering stage, OSNSF incrementally clusters incoming messages and classifies spam based on the features of the cluster that the incoming messages belong to.

We obtained the original implementation [45] of OSNSF. During our evaluation, we adopted all the default values for all the parameters within their released source code. That is, the window size is 100,000, aging factor is 0.8, and the shingling size is 10. The only exception is that the number of threads is undefined and only affecting the training speed. Thus, we arbitrarily chose to execute the system in two-thread mode.

In their paper, they evaluated their system upon both Twitter and Facebook, however we only obtained the original Facebook data set from the authors. We believe that the pollution and unlearning process should be similar for the Facebook and Twitter data sets, which only differ in the data format but use the same core algorithm. The Facebook data set contains 217,802 spam wall posts and 995,630 legitimate wall posts. We partition the data set into ten parts, from which nine parts are used for training and one part is used for testing. Next, we introduce how we pollute the training data to affect the detection rate of the spam filtering system.

A. The Attack – Training Data Pollution

OSNSF is more robust to training data pollution than a traditional machine learning-based detection system. The reasons are twofold: manually picked features and pre-clustering before machine learning. While manually picked features introduce human efforts and cannot evolve over time, they prevent an attacker from polluting the feature selection stage (in which she can craft data to inject fake features). Pre-clustering of incoming samples can reduce the noises (polluted samples) introduced by attacks. For example, in the experiment, if we directly input random mislabeled training data, the filtering rate is affected to a very small degree (approximately 1% difference).

However, during pre-clustering stage, if the injected samples are crafted so that they can form a cluster, they significantly influence the filtering result. We utilize this fact to successfully manipulate OSNSF by polluting its training data. In particular, we craft training data based on two features in OSNSF: average message interval and social degree, because we find that the two features are more effective than other features such as cluster size. In the evaluation, we entirely removed the cluster size parameter from OSNSF and found that the filtering rate only drops a little. In comparison, both average message interval and social degree have large impacts on OSNSF, and further an attacker can manipulate the two parameters. For example, an attacker can send more or less messages to affect the average message interval, and friend or de-friend spam accounts that she owns to change the social degree.

By manipulating the two parameters in OSNSF, we successfully pollute the training data and lower the true positive rate as shown in Figure 3. The x-axis is the rate of polluted samples in all the training data and the y-axis is the true positive rate of the system, i.e., the number of true spams divided by the number of detected spams. As mentioned, because of the existence of clustering, when the pollution rate is lower than 1.7%, the effect of pollution on true positive rate is small. However, when the polluted samples successfully form into a new cluster, the system utilizes the feature values represented by the new cluster containing the polluted samples. Therefore, the true positive rate decreases significantly. This is also reflected in the generated decision tree, containing a branch of the injected feature values.

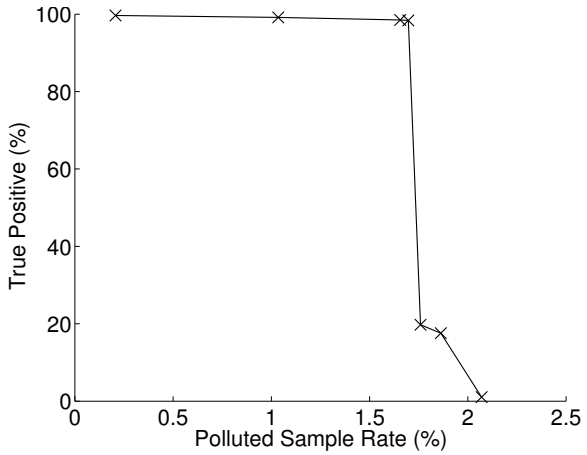

Fig. 3: True Positive Rate of OSNSF vs Percentage of Polluted Training Data. After only 1.75% of the training data is polluted, OSNSF's true positive rate drops sharply because the polluted training data starts to form a cluster.

B. Analytical and Empirical Results

To enable unlearning for OSNSF, there are two steps: decremental clustering and decremental decision tree. The first step – decremental clustering – can be implemented by feeding polluted samples with the opposite labels into the incremental clustering interface of the current system, which either merges new samples into the existing cluster or creates a new cluster. Therefore, we focus on integrating a decremental decision tree into the system.

The C4.5 decision tree incorporated by OSNSF does not support any incremental or decremental algorithm for incom-

[3]In their paper, they have tested both decision tree and support vector machine (SVM). However, they report that "decision tree yields better accuracy, i.e., higher true positive rate and lower false positive rate", and consequently they pick decision tree as the classifying module.

TABLE IV: OSNSF's Detection Rate.

	Original	Polluted	Unlearned
True Positive	99.1%	17.6%	98.5%
False Positive	0.4%	0.0%	0.4%

ing data. Therefore, to support unlearning, we adopted another incremental decision tree – VFDT [38], the implementation of which is available in the VFML (Very Fast Machine Learning) toolkit, and the analytical overhead of which is $O(logN)$ for learning one sample as shown in the VFDT paper. To incorporate VFDT into OSNSF, we modify VFDT to read the decision tree generated by C4.5 and learn polluted samples with the opposite labels. Since VFDT supports the C4.5 data format, the intermediate output of OSNSF (i.e., the extracted feature values for each cluster) can be easily fed into VFDT.

Now we show the empirical results. We first feed polluted samples into OSNSF, and OSNSF outputs the features of newly created clusters. Next, our modified version of VFDT reads the polluted C4.5 decision tree and starts incremental building upon the polluted decision tree instead of a single leaf in the original implementation. After that, a new decision tree is outputted in the C4.5 format and used as the decision tree of OSNSF. The modified version only introduces 33 lines of code into $main.cpp$ of the implementation of OSNSF to support the aforementioned process.

We then show the completeness results in Table IV. The first column shows the original true positive and false positive rate. After polluting 1.75% of the training data, both true and false positive rates decrease. Then, after unlearning the polluted samples, both the true positive rate and false positive rate increase. Note that the unlearning process does not restore the true positive rate to its pre-pollution value . This is because the two decision trees are generated by C4.5 and VFDT – different algorithms. VFDT will not generate the same decision tree as the C4.5 algorithm, even if the inputs as well as the order of the inputs are the same. However, we believe that the filtering rate (true positive) after unlearning is enough for a spam filtering system. Further, for 99.4% of testing samples, the original system and the unlearned system generate the same result, achieving 99.4% for the completeness of unlearning, a fairly high number. We also evaluate the timeliness of unlearning. The retraining takes 30mins, and the unlearning of one sample takes only 21.5ms, leading to a speedup factor of 8.4×10^4.

IX. Unlearning in PJScan

We start by describing the learning technique used in PJScan [51]. PJScan first extracts all the JavaScript using Poppler, analyzes it by Mozilla's SpiderMonkey, and then tokenizes it. After that, PJScan learns and classifies malicious JavaScript using one-class SVM, i.e., PJScan uses only malicious but not benign JavaScript from PDFs to train an SVM engine. To implement one-class SVM, PJScan first uses an existing SVM classifier called libSVM [13] to generate the α values [20] of all support vectors, and then calculates the center and the radius of an n-sphere in a space with $n+1$ dimensions. Then, if an incoming data sample falls within the n-sphere, the data sample is classified as malicious; otherwise, the data sample is classified as benign. We are aware of other machine learning-based PDF detection engines [66, 68] after PJScan, however we choose PJScan because it is open-source and unique in adopting one-class machine learning.

TABLE V: PJScan's Detection Rate. After unlearning, the detection rate of PJScan is the same as it was before pollution, illustrating that our unlearning is complete.

	Pollution Rate			Unlearned
	0%	21.8%	28.2%	
Detection Rate	81.5%	69.3%	46.2%	81.5%

Because the source code of PJScan [16] dates back to 2011, PJScan does not support some of the recent up-to-date libraries. In particular, we installed an old Poppler with version 0.18.2, because some of APIs used in PJScan are not supported by the latest version of Poppler. Meanwhile, the most recent version of Boost library has some conflicts with PJScan, and we made several modifications to PJScan so that those two are compatible with each other. In particular, we needed to modify $boost :: filesystem :: path.file\_string()$ to $boost :: filesystem :: path.string()$, change the definition of $BOOST\_FILESYSTEM\_VERSION$ from 2 to 3, and delete an obsolete catch block dealing with a file reading exception. Then, we simply executed the "install.sh" provided by PJScan for installation. For the PJScan experiment, we also use the data set from Huawei, which contains 65 malicious PDF samples with corresponding JavaScript. Because the size of the data set is small, half of the PDFs are used for training, and the other half are used for testing.

A. The Attack – Training Data Pollution

To pollute PJScan, we need to move the center of the new, polluted n-sphere far away from the original center. In practice, we repeat the same alert functions ($alert(1);$) and inject such a fixed pattern into PDFs to achieve it.

We show the pollution results in Table V. The first column of Table V shows that without pollution PJScan classifies 81.5% of the malicious PDFs as malicious. When we pollute 21.8% of the training samples, the detection rate decreases to 69.3%. Then, when we increase the percentage of polluted training samples to 28.2%, the detection rate finally drops to 46.2%. The result indicates that one-class machine learning is harder to pollute than two-class machine learning. Specifically, two-class SVM classifier needs to draw a line between the sphere labeled as benign and the sphere labeled as malicious, and therefore the radius of one sphere is constrained by the other. In contrast, one-class SVM classifier can always increase the radius of the sphere and keep the percentage of included data samples in the sphere as a constant. This also aligns with the fact that one-class machine learning is robust to mimic attacks [57].

B. Analytical and Empirical Results

The unlearning process can be divided into two stages due to inherent properties of one-class SVM in PJScan. The first stage of unlearning is to re-calculate the new α value [20] of each support vector calculated by the solver in libSVM, and the second stage is to recalculate the center and the radius of the sphere. In the first stage, the calculation of α values is an iterative process of adaptive SQ learning, which starts from an initial assignment of α values, and in the end reaches the optimal point. Since the changes of α during unlearning is relatively small, we just need to feed the old values of α into the iteration process and then let the iteration process output the new α.

The second stage of unlearning recalculates the value of the center and the radius. The calculation of the center is in the form of $\alpha_1 x_1 + \alpha_2 x_2 + ...$ – a summation. If the value of α changes in the first stage, we need to multiply the delta of α by the support vector and add it to the summation. If a new support vector emerges or an old one disappears, we also need to add or subtract corresponding values from the summation. Meanwhile, the calculation of the radius is based on the distance between the support vector with the smallest α and the center. If that support vector changes, the unlearning needs to recalculate the value of radius from scratch. Otherwise, as in the calculation of the center, the unlearning can utilize some of the partial calculation results from the past.

Empirically, in total, we added 30 lines of code into the libsvm_oc module of PJScan to update the α value, the center and the radius. Next, we evaluate the completeness by showing the unlearning result in the last column of Table V, the same as the original detection rate. In addition, we also evaluate the timeliness of unlearning. To calculate the α value, retraining takes 42 iterations, and unlearning one data sample takes 2.4 iterations on average, significantly smaller than retraining.

X. Discussions

Unlearning, or forgetting systems in general, aim to restore privacy, security, and usability. They give users and service providers the flexibility to control *when* to forget *which* data. They do not aim to protect the data that remains in the system. Private data in the system may still be leaked, polluted training data in the system may still mislead predication, and incorrect analytics in the system may still result in bogus recommendations.

Before unlearning a data sample, we must identify this sample. This identification problem is orthogonal to what we have solved in this paper. Sometimes, the problem is straightforward to solve. For instance, a user knows precisely which of her data items is sensitive. Other times, this problem may be solved using manual analysis. For instance, a vigilant operator notices an unusual drop in spam detection rate, investigates the incident, and finds out that some training data is polluted. This problem may also be solved using automated approaches. For instance, a dynamic program analysis tool analyzes each malware training data sample thoroughly and confirms that the sample is indeed malicious. Unlearning is complimentary to these manual or automatic solutions.

We believe our unlearning approach is general and widely applicable. However, as shown in our evaluation (§VIII), not every machine learning algorithm can be converted to the summation form. Fortunately, these cases are rare, and custom unlearning approaches exist for them.

XI. Related Work

In §I, we briefly discussed related work. In this section, we discuss related work in detail. We start with some attacks targeting machine learning (§XI-A), then the defenses (§XI-B), and finally incremental machine learning (§XI-C).

A. Adversarial Machine Learning

Broadly speaking, adversarial machine learning [22, 47] studies the behavior of machine learning in adversarial environments. Based on a prior taxonomy [47], the attacks targeting machine learning are classified into two major categories: (1) *causative attacks* in which an attacker has "write" access to the learning system – she pollutes the training data and subsequently influences the trained models and prediction results; and (2) *exploratory attacks* in which an attacker has "read-only" access – she sends data samples to the learning system hoping to steal private data inside the system or evade detection. In the rest of this subsection, we discuss these two categories of attacks in greater detail.

1) Causative Attacks: These attacks are the same as data pollution attacks (§II-B2). Perdisci et al. [56] developed an attack against PolyGraph [55], an automatic worm signature generator that classifies network flows as either benign or malicious using a naïve Bayes classifier. In this setup, the attacker compromises a machine in a honeynet and sends packets with polluted protocol header fields. These injected packets make PolyGraph fail to generate useful worm signatures. Nelson et al. [54] developed an attack against a commercial spam filter SpamBayes [18] which also uses naïve Bayes. They showed that, by polluting only 1% of the training data with well-crafted emails, an attacker successfully causes SpamBayes to flag a benign email as spam 90% of the time. While these two attacks target Bayesian classifiers, other classifiers can also be attacked in the same manner, as illustrated by Biggio et al.'s attack on SVM [24]. Instead of focusing on individual classifiers, Fumera et al. [44] proposed a framework for evaluating classifier resilience against causative attacks at the design stage. They applied their framework on several real-world applications and showed that the classifiers in these applications are all vulnerable.

Our practical pollution attacks targeting Zozzle [35], OSNSF [46], and PJScan [51] fall into this causative attack category. All such attacks, including prior ones and ours, serve as a good motivation for unlearning.

2) Exploratory Attacks: There are two sub-categories of exploratory attacks. The first sub-category of exploratory attacks is system inference or model inversion attacks, as discussed in §II-B1. Calandrino et al. [29] showed that, given

some auxiliary information of a particular user, an attacker can infer the transaction history of the user. Fredrikson et al. [43] showed that an attacker can infer the genetic markers of a patient given her demographic information. These attacks serve as another motivation for unlearning.

In the second sub-category, an attacker camouflages malicious samples as benign samples, and influences the prediction results of a learning system. In particular, for those systems that detect samples with malicious intentions, an attacker usually crafts malicious samples to mimic benign samples as much as possible, e.g., by injecting benign features into malicious samples [30, 49, 76, 77]. As suggested by Srndic et al. [76], in order to make learning systems robust to those attacks, one needs to use features inherent to the malicious samples. These attacks are out of scope of our paper because they do not pollute training data nor leak private information of the training data.

B. Defense of Data Pollution and Privacy Leakage

In this subsection, we discuss current defense mechanisms for data pollution and privacy leakage. Although claimed to be robust, many of these defenses are subsequently defeated by new attacks [43, 56]. Therefore, unlearning serves as an excellent complimentary method for these defenses.

1) Defense of Data Pollution: Many defenses of data pollution attacks apply filtering on the training data to remove polluted samples. Brodley et al. [27] filtered mislabeled training data by requiring absolute or majority consensus among the techniques used for labeling data. Cretu et al. [34] introduced a sanitization phase in the machine learning process to filter polluted data. Newsome et al. [55] clustered the training data set to help filter possible polluted samples. However, they are defeated by new attacks [56]. None of these techniques can guarantee that all polluted data is filtered. Another line of defense is to increase the resilience of the algorithms. Dekel et al. [36] developed two techniques to make learning algorithms resilient against attacks. One technique formulates the problem of resilient learning as a linear program, and the other uses the Perceptron algorithm with an online-to-batch conversion technique. Both techniques try to minimize the damage that an attacker could cause, but the attacker may still influence the prediction results of the learning system. Lastly, Bruckner et al. [28] model the learner and the data-pollution attacker as a game and prove that the game has a unique Nash equilibrium.

2) Defense of Privacy Leaks: In general, differential privacy [75, 80] preserves the privacy of each individual item in a data set *equally and invariably*. McSherry et al. [52] built a differentially private recommendation system and showed that in the Netflix Prize data set the system can preserve privacy without significantly degrading the system's accuracy. Recently, Zhang et al. [80] proposed a mechanism to produce private linear regression models, and Vinterbo [75] proposed privacy-preserving projected histograms to produce differentially-private synthetic data sets. However, differential privacy requires that accesses to data fit a shrinking privacy budget, and are only to the fuzzed statistics of the data set. These restrictions make it extremely challenging to build usable systems [43]. In addition, in today's systems, each user's privacy consciousness and each data item's sensitivity varies wildly. In contrast, forgetting systems aim to *restore privacy on select data*, representing a more practical privacy vs utility tradeoff.

C. Incremental Machine Learning

Incremental machine learning studies how to adjust the trained model incrementally to add new training data or remove obsolete data, so it is closely related to our work. Romero et al. [62] found the exact maximal margin hyperplane for linear SVMs so that a new component can be easily added or removed from the inner product. Cauwenberghs et al. [31] proposed using adiabatic increments to update a SVM from l training samples to $l + 1$. Utgoff et al. [74] proposed an incremental algorithm to induce decision trees equivalent to the trees formed by Quinlan's ID3 algorithm. Domingos et al. [38] proposed a high performance construction algorithm of decision trees to deal with high-speed data streams. Recently, Tsai et al. [73] proposed using warm starts to practically build incremental SVMs with linear kernels.

Compared to prior incremental machine learning work, our unlearning approach differs fundamentally because we propose a general efficient unlearning approach applicable to any algorithm that can be converted to the summation form, including some that currently have no incremental versions. For instance, we successfully applied unlearning to normalized cosine similarity which recommendation systems commonly use to compute item-item similarity. This algorithm had no incremental versions prior to our work. In addition, we applied our learning approach to real-world systems, and demonstrated that it is important that unlearning handles all stages of learning, including feature selection and modeling.

Chu et al. [33] used the summation form to speed up machine learning algorithms with map-reduce. Their summation form is based on SQ learning, and provided inspiration for our work. We believe we are the first to establish the connection between unlearning and the summation form. Furthermore, we demonstrated how to convert non-standard machine learning algorithms, e.g., the normalized cosine similarity algorithm, to the summation form. In contrast, prior work converted nine standard machine learning algorithms using only simple transformations.

XII. Conclusion and Future Work

We have presented our vision of *forgetting systems* that completely and quickly forget data and its lineage to restore privacy, security, and usability. They provide numerous benefits. For privacy, they enable a concerned user to remove her sensitive data, ensuring that no residual linage of the data lurks around in the system. For security, they enable service providers to remove polluted training data including its effect from anomaly detectors, ensuring that the detectors operate correctly. For usability, they enable a user to remove

noise and incorrect entries in analytics data, ensuring that a recommendation engine gives useful recommendations.

We have also presented *machine unlearning*, the first step towards making system forget. Our unlearning approach targets machine learning systems, an important and widely used class of systems. Unlearning transforms some or all learning algorithms in a system into a summation form. To forget a training data sample, unlearning updates a small number of summations, and is asymptotically faster than retraining from scratch. Our unlearning approach is general and applies to many machine learning algorithms. Our evaluation on real-world systems and workloads has shown that our approach is general, effective, fast, and easy to use.

Unlearning is only the first (albeit important) step; the bulk of work still lies ahead. We plan to build full-fledged forgetting systems that carefully track data lineage at many levels of granularity, across all operations, and at potentially the Web scale. We invite other researchers to join us in exploring the exciting direction opened up by forgetting systems.

XIII. Acknowledgement

We thank the anonymous reviewers for their constructive feedback; Daniel Hsu for insightful discussions and background on statistical query learning; David Williams-King for careful proofreading and editing; Xin Lu for early contributions to the LensKit experiment; and Yang Tang, Gang Hu, Suzanna Schmeelk, and Marios Pomonis for their many valuable comments. This work was supported in part by AFRL FA8650-11-C-7190 and FA8750-10-2-0253; ONR N00014-12-1-0166; NSF CCF-1162021, CNS-1054906; an NSF CAREER award; an AFOSR YIP award; and a Sloan Research Fellowship.

References

[1] Private Communication with Yang Tang in Columbia University.
[2] Alexa Top Websites. http://www.alexa.com/topsites.
[3] BlueKai — Big Data for Marketing — Oracle Marketing Cloud. http://www.bluekai.com/.
[4] Booklens. https://booklens.umn.edu/about.
[5] Confer. http://confer.csail.mit.edu/.
[6] Delete search history. https://support.google.com/websearch/answer/465?source=gsearch.
[7] Google now. http://www.google.com/landing/now/.
[8] iCloud security questioned over celebrity photo leak 2014: Apple officially launches result of investigation over hacking. http://www.franchiseherald.com/articles/6466/20140909/celebrity-photo-leak-2014.htm.
[9] IMDb database status. http://www.imdb.com/stats.
[10] iText - programmable pdf software. http://itextpdf.com/.
[11] Javascript development tools (JSDT). http://www.eclipse.org/webtools/jsdt/.
[12] LibraryThing. https://www.librarything.com/.
[13] LibSVM – a library for support vector machines. http://www.csie.ntu.edu.tw/~cjlin/libsvm/.
[14] Movielens. http://movielens.org/login.
[15] New IDC worldwide big data technology and services forecast shows market expected to grow to $32.4 billion in 2017. http://www.idc.com/getdoc.jsp?containerId=prUS24542113.
[16] PJScan source code. http://sourceforge.net/projects/pjscan/.
[17] Project honey pot. https://www.projecthoneypot.org/.
[18] SpamBayes. http://spambayes.sourceforge.net/.
[19] Wikipedia: Internet Movie Database. http://en.wikipedia.org/wiki/Internet_Movie_Database.
[20] Wikipedia: Support Vector Machine. http://en.wikipedia.org/wiki/Support_vector_machine.
[21] Xiang Pan's LinkedIn home page. https://www.linkedin.com/pub/xiang-pan/38/454/258.
[22] M. Barreno, B. Nelson, A. D. Joseph, and J. D. Tygar. The security of machine learning. *Mach. Learn.*, 81(2):121–148, Nov. 2010.
[23] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2010.
[24] B. Biggio, B. Nelson, and P. Laskov. Poisoning attacks against support vector machines. In *Proceedings of International Conference on Machine Learning*, ICML, 2012.
[25] M. Brennan, S. Afroz, and R. Greenstadt. Adversarial stylometry: Circumventing authorship recognition to preserve privacy and anonymity. *ACM Trans. Inf. Syst. Secur.*, 15(3):12:1–12:22, Nov. 2012.
[26] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. *Comput. Netw. ISDN Syst.*, 29(8-13):1157–1166, Sept. 1997.
[27] C. E. Brodley and M. A. Friedl. Identifying mislabeled training data. *Journal of Artificial Intelligence Research*, 11:131–167, 1999.
[28] M. Brückner, C. Kanzow, and T. Scheffer. Static prediction games for adversarial learning problems. *J. Mach. Learn. Res.*, 13(1):2617–2654, Sept. 2012.
[29] J. A. Calandrino, A. Kilzer, A. Narayanan, E. W. Felten, and V. Shmatikov. You might also like: Privacy risks of collaborative filtering. In *Proceedings of 20th IEEE Symposium on Security and Privacy*, May 2011.
[30] Y. Cao, X. Pan, Y. Chen, and J. Zhuge. JShield: Towards real-time and vulnerability-based detection of polluted drive-by download attacks. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC, 2014.
[31] G. Cauwenberghs and T. Poggio. Incremental and decremental support vector machine learning. In *Advances in Neural Information Processing Systems (NIPS*2000)*, volume 13, 2001.
[32] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum. Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proceedings of the 14th Conference on USENIX Security Symposium*, 2005.
[33] C. T. Chu, S. K. Kim, Y. A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. In B. Schölkopf, J. C. Platt, and T. Hoffman, editors, *NIPS*, pages 281–288. MIT Press, 2006.
[34] G. F. Cretu, A. Stavrou, M. E. Locasto, S. J. Stolfo, and A. D. Keromytis. Casting out Demons: Sanitizing Training Data for Anomaly Sensors. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP, 2008.
[35] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. Zozzle: Fast and precise in-browser javascript malware detection. In *Proceedings of the 20th USENIX Conference on Security*, 2011.
[36] O. Dekel, O. Shamir, and L. Xiao. Learning to classify with missing and corrupted features. *Mach. Learn.*, 81(2):149–178, Nov. 2010.
[37] M. Deshpande and G. Karypis. Item-based top-n recommendation algorithms. *ACM Trans. Inf. Syst.*, 22(1):143–177, Jan. 2004.
[38] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD, 2000.
[39] M. D. Ekstrand, M. Ludwig, J. A. Konstan, and J. Riedl. Rethinking the recommender research ecosystem: reproducibility, openness, and lenskit. In *RecSys*, pages 133–140. ACM, 2011.
[40] G. Elbaz. Data markets: The emerging data economy. http://techcrunch.com/2012/09/30/data-markets-the-emerging-data-economy/.
[41] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2010.
[42] M. J. Foley. How microsoft's bing-related research is funneling back into products. http://www.zdnet.com/how-microsofts-bing-related-research-is-funneling-back-into-products-7000013001/.
[43] M. Fredrikson, E. Lantz, S. Jha, S. Lin, D. Page, and T. Ristenpart. Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing. In *Proceedings of USENIX Security*, August 2014.
[44] G. Fumera and B. Biggio. Security evaluation of pattern classifiers under attack. *IEEE Transactions on Knowledge and Data Engineering*, 99(1), 2013.

[45] H. Gao. A syntactic-based spam detection tool. http://list.cs.northwestern.edu/osnsecurity/syntactic_files/download.php.

[46] H. Gao, Y. Chen, K. Lee, D. Palsetia, and A. N. Choudhary. Towards online spam filtering in social networks. In *Proceedings of Network and Distributed Systems Security Symposium*, NDSS, 2012.

[47] L. Huang, A. D. Joseph, B. Nelson, B. I. Rubinstein, and J. D. Tygar. Adversarial machine learning. In *Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence*, AISec, 2011.

[48] M. Kearns. Efficient noise-tolerant learning from statistical queries. *J. ACM*, 45(6):983–1006, Nov. 1998.

[49] M. Kearns and M. Li. Learning in the presence of malicious errors. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC, 1988.

[50] A. Kharpal. Google axes 170,000 'right to be forgotten' links. http://www.cnbc.com/id/102082044.

[51] P. Laskov and N. Šrndić. Static detection of malicious javascript-bearing pdf documents. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC, 2011.

[52] F. McSherry and I. Mironov. Differentially private recommender systems: Building privacy into the netflix prize contenders. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD, 2009.

[53] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar. F4: Facebook's warm blob storage system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2014.

[54] B. Nelson, M. Barreno, F. J. Chi, A. D. Joseph, B. I. P. Rubinstein, U. Saini, C. Sutton, J. D. Tygar, and K. Xia. Exploiting machine learning to subvert your spam filter. In *Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats*, LEET, 2008.

[55] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, 2005.

[56] R. Perdisci, D. Dagon, W. Lee, P. Fogla, and M. I. Sharif. Misleading worm signature generators using deliberate noise injection. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, 2006.

[57] R. Perdisci, G. Gu, and W. Lee. Using an ensemble of one-class svm classifiers to harden payload-based anomaly detection systems. In *Proceedings of the Sixth International Conference on Data Mining*, ICDM, 2006.

[58] J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, Mar. 1986.

[59] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *Proceedings of 18th USENIX Security Symposium*, 2009.

[60] J. Reardon, S. Capkun, and D. Basin. Data node encrypted file system: Efficient secure deletion for flash memory. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security, 2012.

[61] C. Riederer, V. Erramilli, A. Chaintreau, B. Krishnamurthy, and P. Rodriguez. For sale : Your data: By : You. In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, HotNets-X, 2011.

[62] E. Romero, I. Barrio, and L. Belanche. Incremental and decremental learning for linear support vector machines. In *Proceedings of the 17th International Conference on Artificial Neural Networks*, ICANN, 2007.

[63] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th International Conference on World Wide Web*, WWW, 2001.

[64] D. Sculley, M. E. Otey, M. Pohl, B. Spitznagel, J. Hainsworth, and Y. Zhou. Detecting adversarial advertisements in the wild. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD, 2011.

[65] M. A. Shipp, K. N. Ross, P. Tamayo, A. P. Weng, J. L. Kutok, R. C. T. Aguiar, M. Gaasenbeek, M. Angelo, M. Reich, G. S. Pinkus, T. S. Ray, M. A. Koval, K. W. Last, A. Norton, T. A. Lister, J. Mesirov, D. S. Neuberg, E. S. Lander, J. C. Aster, and T. R. Golub. Diffuse large B-cell lymphoma outcome prediction by gene-expression profiling and supervised machine learning. *Nature Medicine*, 8(1):68–74, Jan. 2002.

[66] C. Smutz and A. Stavrou. Malicious pdf detection using metadata and structural features. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC, 2012.

[67] R. Spahn, J. Bell, M. Z. Lee, S. Bhamidipati, R. Geambasu, and G. Kaiser. Pebbles: Fine-grained data management abstractions for modern operating systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI, 2014.

[68] N. Srndic and P. Laskov. Detection of malicious PDF files based on hierarchical document structure. In *20th Annual Network and Distributed System Security Symposium*, NDSS, 2013.

[69] J. Sutter. Some quitting facebook as privacy concerns escalate. http://www.cnn.com/2010/TECH/05/13/facebook.delete.privacy/.

[70] Y. Tang, P. P. C. Lee, J. C. S. Lui, and R. Perlman. Secure overlay cloud storage with access control and assured deletion. *IEEE Trans. Dependable Secur. Comput.*, 9(6):903–916, Nov. 2012.

[71] D. Tax and R. Duin. Support vector data description. *Machine Learning*, 54(1).

[72] The Editorial Board of the New York Times. Ordering google to forget. http://www.nytimes.com/2014/05/14/opinion/ordering-google-to-forget.html?_r=0.

[73] C.-H. Tsai, C.-Y. Lin, and C.-J. Lin. Incremental and decremental training for linear classification. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD, 2014.

[74] P. E. Utgoff. Incremental induction of decision trees. *Mach. Learn.*, 4(2):161–186, Nov. 1989.

[75] S. A. Vinterbo. Differentially private projected histograms: Construction and use for prediction. In P. A. Flach, T. D. Bie, and N. Cristianini, editors, *ECML/PKDD (2)*, volume 7524 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2012.

[76] N. Šrndic and P. Laskov. Practical evasion of a learning-based classifier: A case study. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, 2014.

[77] G. Wang, T. Wang, H. Zheng, and B. Y. Zhao. Man vs. machine: Practical adversarial detection of malicious crowdsourcing workers. In *Proceedings of USENIX Security*, August 2014.

[78] R. Wang, Y. F. Li, X. Wang, H. Tang, and X. Zhou. Learning your identity and disease from research papers: Information leaks in genome wide association study. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS, pages 534–544, New York, NY, USA, 2009. ACM.

[79] V. Woollaston. How to delete your photos from iCloud: Simple step by step guide to stop your images getting into the wrong hands. http://www.dailymail.co.uk/sciencetech/article-2740607/How-delete-YOUR-photos-iCloud-stop-getting-wrong-hands.html.

[80] J. Zhang, Z. Zhang, X. Xiao, Y. Yang, and M. Winslett. Functional mechanism: Regression analysis under differential privacy. *Proceedings of VLDB Endow.*, 5(11):1364–1375, July 2012.