Efficient Repair of Polluted Machine Learning Systems via Causal Unlearning

Yinzhi Cao
Lehigh University, Bethlehem PA
yinzhi.cao@lehigh.edu

Alexander Fangxiao Yu
Columbia University, New York, NY
afy2103@columbia.edu

Andrew Aday
Columbia University, New York, NY
aza2112@columbia.edu

Eric Stahl
Lehigh University, Bethlehem PA
ems316@lehigh.edu

Jon Merwine
Lehigh University, Bethlehem PA
jmm214@lehigh.edu

Junfeng Yang
Columbia University, New York, NY
junfeng@cs.columbia.edu

ABSTRACT

Machine learning systems, though being successful in many real-world applications, are known to remain prone to errors and attacks. A major attack, called data pollution, injects maliciously crafted training data samples into the training set, causing the system to learn an incorrect model and subsequently misclassify testing samples. A natural solution to a data pollution attack is to remove the polluted data from the training set and relearn a clean model. Unfortunately, the training set of a real-world machine learning system can contain millions of samples; it is thus hopeless for an administrator to manually inspect all of them to weed out the polluted ones.

This paper presents an approach called *causal unlearning* and a corresponding system called KARMA to efficiently repair a polluted learning system. KARMA dramatically reduces the manual effort of administrators by automatically detecting the set of polluted training data samples with high precision and recall. Evaluation on three learning systems show that KARMA greatly reduces manual effort for repair, and has high precision and recall.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; • **Security and privacy** → *Systems security*;

KEYWORDS

Causality; Data Pollution Attacks; Machine Unlearning

ACM Reference format:
Yinzhi Cao, Alexander Fangxiao Yu, Andrew Aday, Eric Stahl, Jon Merwine, and Junfeng Yang. 2018. Efficient Repair of Polluted Machine Learning Systems via Causal Unlearning. In *Proceedings of 2018 ACM Asia Conference on Computer and Communications Security, Incheon, Republic of Korea, June 4–8, 2018 (ASIA CCS '18),* 13 pages.
https://doi.org/10.1145/3196494.3196517

ASIA CCS '18, June 4–8, 2018, Incheon, Republic of Korea

ACM ISBN 978-1-4503-5576-6/18/06…$15.00
https://doi.org/10.1145/3196494.3196517

1 INTRODUCTION

Machine learning systems play an increasingly important role in today's world, from recommending products, contents, and friends to self-driving cars. However, they remain vulnerable to a variety of attacks, and mechanisms to defend against or cope with the attacks remain understudied.

A major attack, called *data pollution* [35], injects maliciously crafted training data samples into the training set, causing the system to learn an incorrect model and subsequently misclassify testing samples. The most recent real-world example is Microsoft's AI powered chatbot Tay. Tay learned racism because some Twitter users interacted with Tay using offensive, racist words, and these words were included in Tay's training set [36]. In another proof-of-concept example, Wang et al. [45] show that injected false samples in the training set can mislead the machine learning classifier detecting malicious crowdsourcing workers. Similarly, Perdisci et al. [35] show that well-crafted fake network flows as part of the training data can significantly influence the worm signatures generated by PolyGraph [34], a worm detection engine.

A natural solution to a data pollution attack is to remove the polluted data from the training set and relearn a clean model. Unfortunately, the training set of a real-world machine learning system can contain millions of samples; it is thus hopeless for an administrator to manually inspect all of them to weed out the polluted ones. This overwhelming amount of manual cleaning required is perhaps why Microsoft brought Tay offline for repair but has yet to bring it back online.

This paper presents KARMA,[1] the first system designed for efficient repair of a polluted machine learning system. It dramatically reduces the manual effort administrators need to do by automatically detecting the set of polluted training data samples with high precision and recall. Key in KARMA is an idea we call *casual unlearning*. Specifically, to launch a data pollution attack, an attacker inevitably leaves a trace—a causality chain that goes from the polluted training samples, to a polluted learning model, to misclassified testing samples. Leveraging this causality trace, KARMA searches through different subsets of training samples and returns the subset that causes the most misclassifications as the set of polluted training samples. KARMA then determines how many misclassified samples are caused by a subset of training samples by removing the subset from the training set, computing a new model, and checking whether the new model correctly classifies the previously misclassified samples.

[1] Karma, originated from Hinduism and Buddhism, means destiny or fate.

KARMA thus reduces the manual effort required down to two parts. First, it assumes that some users report misclassified testing samples (e.g., as in the Microsoft Tay example or spam detection) and, for added security, it relies on administrators to verify the user reports. KARMA does not require all misclassified samples to be collected upfront before it repairs a system; instead, our evaluation shows that it can incrementally clean a system as users gradually report misclassifications. Second, it relies on administrators to inspect the set of polluted samples it returns. KARMA determines the set of polluted samples leveraging causality of misclassifications, not contents of the samples. Therefore, it may have false positives, such as flagging (unpolluted) outliers in the training set. However, we view it an advantage to use KARMA to detect outliers from the training set. It may also have false negatives, such as missing polluted training samples. However, if the remaining polluted samples do not cause user-noticeable misclassifications, their harm may be little.

To ease discussion, we term the set of user-reported misclassified test samples the *oracle set*. Administrators can augment this oracle set with correctly classified test samples for better results. We assume that all samples in the oracle set are assigned their correct classifications. They may come from aforementioned administrator verification, or automated approaches such as malware detection via sophisticated dynamic analysis. In either case, we can afford to verify the oracle set but not the entire training set because the oracle set is often orders of magnitude smaller than the training set.

Although the causal unlearning idea is intuitive, KARMA faces two challenges. First, the search space for causality in the training set is very large, but at the same time KARMA needs to inspect the entire space to avoid evasion. Second, a large training set can also make it costly to compute a new model after removing a subset. To speed up the search for causality, KARMA adopts a heuristics that balances search coverage and speed based on that similar causes will lead to similar effects with a high probability. Specifically, if two training samples, serving as causes in KARMA, are very similar and share the same label, their influences on the learning system are also similar, i.e, there is a higher chance that they are both polluted or unpolluted. To speed up model computation, KARMA leverages *machine unlearning* [14], but also works with incremental or decremental machine learning [15, 20, 37, 42, 43].

We evaluated KARMA on three systems covering two popular learning algorithms (Bayes and SVM) and two application domains (spam and malware detection). Our results show:

- KARMA reduces manual efforts. Specifically, in an attack scenario from Nelson et al. [33], i.e., 1% of samples are polluted, KARMA reduces the manual effort from the entire training set to 3% of the training set, i.e., 2% as an oracle set and 1% as the identified polluted samples, an over 30× reduction.
- KARMA is robust to a variety of attacks with different parameters. Specifically, KARMA repairs learning models affected by a wide variety of 95 data pollution attacks ranging from mislabelling to injection attacks, with different tactics such as targeted and blind, and having different pollution rates from 0.5% to 30%.
- KARMA is accurate. Specifically, KARMA identifies 99.2% polluted samples in median with the minimum as 98.0% and the maximum 99.97%.
- KARMA is effective. Specifically, KARMA restores the accuracy of polluted learning models against a third dataset to the vanilla one within 1% differences.

This paper makes three main contributions. At a conceptual level, causal unlearning is the first approach to efficient repair of learning systems, and may inspire many possible systems toward this direction. At a system level, we have built KARMA, a causal unlearning system that uses several mechanisms to efficiently determine the set of polluted data samples with high precision and recall. KARMA is open source and available at the following repository (https://github.com/CausalUnlearning/KARMA). At an evaluation level, we show that our approach works with real-world machine learning systems and greatly reduces manual effort required to repair a polluted system.

Our work is only the first step toward practical repair of learning systems; more challenges lie ahead. How can we perform causal unlearning on other machine learning algorithms and systems? How can we repair a system that experienced other types of attacks targeting machine learning? While removing training samples is one way to repair or improve a learning system, adding samples is another which KARMA does not support. We hope other researchers will join us in addressing these challenges.

2 THREAT MODEL

The threat model of KARMA assumes one learning system and three parties—i.e., an administrator of the system, users of the system, and an attacker. The administrator is absolutely trustworthy, being responsible for training and maintaining the learning system; the attacker is malicious and tries to subvert the system by polluting the training dataset; most users are trustworthy, but some of them may have malicious intent. Note that we do not restrict the capability of the attacker, i.e., theoretically the attacker can pollute arbitrary number of training data. In practice, as long as the pollution is effective, the attacker also wants to minimize the number of polluted training data and reduce her chance of being caught. Depending on how the administrator collects the training dataset, we list two attack scenarios where an attacker can pollute training set.

Scenario One—Mislabelling Attack: In this scenario, an administrator of a learning system adopts crowdsourcing, such as asking Amazon Mechanical Turks, to label training samples. Some of the crowdsourcing workers have malicious intents, i.e., they will mislabel[2] samples provided by the administrator, to pollute the learning model. In this scenario, the capability of attackers is limited in polluting the labels but not contents of training samples, because all the samples are provided by the administrator.

Scenario Two—Injection Attack: In this scenario, an administrator of a learning system tries to collect malicious samples, such as spam and malware, through a honeypot-based technique. An attacker figures out the purpose of the honeypot and then intentionally sends crafted, polluted samples to the honeypot so that such samples will be include in the training set. Note that attackers, different from the first scenario, are able to craft and inject contents. However, the

[2] In this paper, misclassified samples (emails) refer to these that are incorrectly classified by the learning model; mislabeled samples (emails) refer to these in the training set that are incorrectly labeled by the attacker.

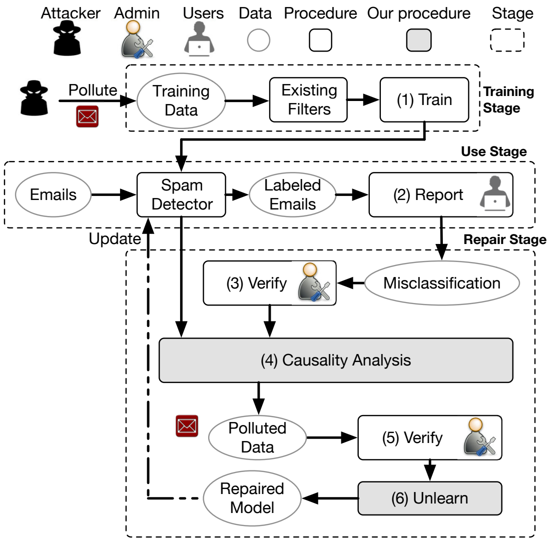

Figure 1: Deployment Model (spam detectors as an example).

attacker can only control one class of samples, i.e., malicious ones, because a honeypot usually collects just malicious samples.

3 DEPLOYMENT MODEL

When we deploy KARMA with a learning system, there are tree stages in the lifecycle of deployment: training, use and repair. In the training stage, the administrator will train a learning model based on potentially polluted training data. Then, in the use stage, the administrator will obtain feedbacks of the learning system from third parties, such as users of the model and other independent testing parties including VirusTotal for malware detection. After that, based on the feedbacks especially misclassification reports, in the repair stage, the administrator will repair the model with the help of an oracle, such as a human performing code reviews and a dynamic analyzer exploring and examining program behaviors.

An Example Deployment with Spam Detectors. Figure 1 shows the deployment model of KARMA by using an example of spam detectors where the oracle is a trusted human. Say, a spam detector is trained by an administrator with a potential polluted training set (step one) and deployed together with an email client. When training the system, the administrator might have already deployed existing approaches, which are orthogonal to KARMA, to filter potential polluted emails [17, 34, 39] and make the model robust. However, some polluted emails may have bypassed the filter and still make the learning model misclassify samples as evident by existing attacks [35, 45].

Then, the users of this email client complain about misclassifications and report misclassified emails to the administrator (step two). The administrator or other trusted person, i.e., an oracle, verifies these reported misclassifications, and uses them as an input dataset for KARMA called the oracle set (step three). To improve accuracy, the oracle set can include a small number of correctly classified emails as well.

Next, the administrator deploys KARMA to find the cause of misclassifications in the oracle set (step four). The cause, a subset of the training set, will be verified by the administrator to confirm which samples are polluted and whether the misclassification is caused by data pollution (step five). After that, the administrator can ask our system to repair the learning model by removing verified polluted samples (step six).

Note that KARMA greatly relieves the burden of the oracle. Without KARMA, an administrator needs to first verify misclassifications reported by users and confirms that the model misbehaves. Then, the oracle needs to confirm misclassifications, go over all training samples and find pollutions. Now, with KARMA, the oracle still verifies misclassifications reported by users, but then only needs to verify the dataset reported by the users and misclassification cause, both smaller than the entire training dataset. As shown in the evaluation (Section 6), the size of oracle set is less than 2% of training data. The size of misclassification cause highly depends on the attacker's strategies, which varies from 0.5% to 30% of training data in our experiment. According to Nelson et al. [33], only 1% of samples are needed to subvert an email filter. It is our future work to further decrease the size of samples to be inspected by the administrator.

4 DESIGN

We present the design of KARMA in this section.

4.1 Overview

Let us first discuss the inputs and outputs of KARMA. Specifically, KARMA takes three inputs: one machine learning model (M), usually a replicate of the deployed learning model for analysis purpose, and two datasets. The first set $S_{training}$—the one used to generate M—is large and potentially polluted by an attacker; the second set S_{oracle} is a small dataset mostly coming from misclassification reported by users of M (Step 2 of the deployment model in Figure 1). S_{oracle} is verified by an oracle, such as the administrator of M. The output of KARMA is another dataset S_{cause} that leads to the misclassifications of S_{oracle} when classified by M. In KARMA, the degree of misclassifications can be represented as the detection accuracy of M against S_{oracle} defined in Equation 1.

$$Accuracy_{S_{oracle}} = 1 - \frac{|\{x \ is \ misclassified \ by \ M \ | x \in S_{oracle}\}|}{|S_{oracle}|} \quad (1)$$

Now let us discuss how the administrator validates S_{oracle}, which comes from what the users report. Specifically, the administrator's job can be summarized as follows. Note that the amount work for the administrator is minimized because everything is performed on a small number of S_{oracle} not the entire $S_{training}$.

- Adding user-reported misclassified samples to S_{oracle} iteratively. Once the administrator collects some misclassified samples as S_{oracle}, she can run KARMA using to partially repair M by finding the misclassification cause and removing a subset of polluted training samples. Because M is still polluted and produces incorrect results, i.e., misclassifying samples, users of M will report further misclassifications to the administrator. Then, the administrator can construct a new S_{oracle}, and ask KARMA to further repair M. We have a detailed evaluation about this scenario in Section 6.5.
- Removing falsely reported samples from users with malicious intent. Once the administrator finds that some reported samples are correctly classified, she can remove such samples as shown

in Step 2 of the deployment model in Figure 1. None of the false report will be fed into and thus influence KARMA. Further, the administrator may even block such users from reporting more samples.

- Understanding the output of KARMA and adding more samples to the training set, if necessary, to improve M. KARMA will find the cause of misclassification, which could be some correctly labeled samples failing to represent the misclassified samples. That is, the training data may be insufficient, e.g., lacking a specific category of samples, so that M misclassifies the entire category. In this case, the administrator can rely on the cause found by KARMA to introduce new samples that can differentiate the cause and the misclassified samples. This is considered beyond the scope of the paper though, because no attackers or data pollutions exist.

4.2 Causality Analysis

We present how causality analysis works in this section. From a high level, what KARMA does is to try removing different samples from the training set and observe whether samples are still misclassified. That is, based on the effect (i.e., misclassified samples), KARMA tries to inspect the cause (e.g., polluted samples) by searching through the cause space (i.e., training set). If the effect (misclassification) is mitigated when removing a subset of training data, we can consider this subset as the cause, which is the polluted samples when the model is polluted.

In the rest of the subsection, we first present the causality analysis from three perspectives: causality search, causality growth and causality determination.

4.2.1 Causality Search. The first step of causality analysis is to search for the potential causality that leads to the misclassification of a learning system against S_{oracle}. Although ultimately KARMA needs to search every training sample and try different combinations, in order to speed up the search, KARMA will conduct a guided process that prioritize exploration of training samples that have higher probability of being polluted. Note that it is the job of causality determination not the search stage to determine whether samples are polluted.

Here is how the search with a two-phase procedure works. In the first phase, KARMA clusters the misclassified data into different parts, and the centers of the clusters are extracted. Then, in the second phase, KARMA prioritizes the search of similar data samples in the training set based on the extracted centers of misclassified data clusters. These training samples can be used for the causality growth stage of KARMA as seeds.

Similarity in KARMA is measured by a definition called divergence score in Equation 2, i.e., one divided by the number of common features between two samples. That is, if two samples share many common features, they are close to each other and their divergence score is low.

$$d(x, y) = \frac{1}{|\{F|F\ is\ x's\ feature\} \cap \{F|F\ is\ y's\ feature\}|} \tag{2}$$

Now, let us introduce these two phases in details. In the first phase, misclassified data are first divided into groups based on their labels. In the most common case where only two labels are available, such as malicious and benign, all the misclassified benign samples are in group one, and all the misclassified malicious ones group two. In the rest of the subsection, for convenience, misclassified data is referred to only one group of misclassified data. After grouping, we start clustering in each group individually, and the clustering algorithm is very similar to k-means but using our divergence scores.

Here is how the first phase, clustering misclassified data, works. KARMA randomly selects k samples from misclassified data, c_1, c_2, ..., c_k, which are the centers of the initial clusters. Then, KARMA iterates through all other samples in the misclassified data, and calculates the divergence scores between all other samples and each center. A sample will be included into the cluster where the center has the smallest divergence score with the sample. Next, the center of each cluster, c_1, c_2, ..., c_k, will be updated based on the common feature list of that cluster. The entire process is then repeated using the updated centers until convergence. The final c_1, c_2, ..., c_k are used in the second phase.

In the second phase, for each c_i, KARMA iterates through the training set, and finds the s_i that has the smallest divergence score with c_i. All s_1, s_2, ..., s_k will be used as the seeds for the peak finder and the unlearning module. That is, these samples have a higher probability to be polluted than others in the training set. This phase of finding all the seeds can be further divided into two sub-phases: pre-computing and searching.

- Pre-computing Sub-phase. The pre-computing sub-phase constructs a so-called judging tree that can be used in and expedites the searching. The judging tree has one root node with all the features used in the training set. The root node has k children (e.g., k equals the square root of the size of the training set), where each child represents a subset of the training set and the value of each child is the union of all the features used in the subset. Each child also has l descendants (e.g., l equals the square root of the size of the subset), and the descendant will also have children. The structure is repeated and extended until the leaf node only has one sample.

  The construction of a judging tree is as follows. KARMA adopts a similar clustering mechanism, i.e., a variation of k-means, used in the first phase to compute one level of the judging tree. The difference is that instead of using the common feature list as the center, the construction of a judging tree adopts the union of all the features in the cluster as the center. KARMA still computes the divergence scores between each sample and the centers, and then dispatches the sample to the closest cluster. The overall process is repeated until convergence, and the k clusters are served as the k nodes of the judging tree. Then, the next levels are computed using similar algorithm until the construction reaches the leaf node.

- Searching Sub-phase. With the judging tree, in the searching sub-phase, KARMA performs a depth-first, priority searching algorithm, and maintains a minimum divergence score d_{min} between c_i and the searched samples so far. In the beginning, d_{min} equals infinity. On each level, KARMA ranks the search priority based on the divergence scores between the nodes and c_i: Nodes with smaller divergence scores are searched first, and nodes with divergence scores larger than d_{min} are skipped directly. d_{min} is only updated in the leaf node level, i.e., if the divergence score between c_i and a sample is smaller than d_{min}, d_{min} is updated to that score. The sample corresponding to the final d_{min} is selected as s_i.

4.2.2 Causality Growth. The second step of KARMA is to grow the causality found in the first stage by finding more training samples and forming a cluster. Similar to causality search, causality growth does not determine whether samples are polluted either. Our key observation for causality growth is that the common feature list, i.e., a list of features whose values are identical in two samples as defined in the divergence score (Equation 2), only shrinks as the causality cluster grows. Correspondingly, the divergence score between the causality cluster and each sample in the target dataset increases when the size of the cluster increases.

Thus, because the divergence score increases as the causality cluster grows, instead of calculating the divergence score in each round, we can maintain a lower bound of the divergence score and only update the score if necessary. The detailed steps are as follows and shown in Algorithm 1. We maintain a sorted list (one can also use a priority queue) of the lower bound of the divergence scores between the cluster and each sample in the target dataset.

The initial list is consisted of all the divergence scores between the cluster and samples in the target dataset and correctly ordered. As shown in Line 1–4 of Algorithm 1, we fetch each sample from the target dataset ($TSet$), calculate the divergence score between the cluster ($CSet$) and each sample, and then put the result to the list ($DivergenceScoreList$). After iterating through all the samples in $CSet$, the initial list is generated.

In each round, we fetch the element on the top of the list, i.e., the one with the smallest value, and update the value to the latest divergence score between the cluster and sample. Particularly, we pop out the element from the top of $DivergenceScoreList$ (Line 8 and 9), update the value (Line 10), and then put it back to $DivergenceScoreList$ (Line 11). If the updated value still stays on top of the list, we will include the sample corresponding to the value into the cluster; otherwise, we will insert the value back to the list in its correct position to maintain the order, fetch the new element on the top of the list, and repeat the process until we have an updated value that stays on the top. That is, we test whether the updated top of $DivergenceScoreList$ is calculated using the current cluster ($CSet$) in Line 7. If not, Line 8–11 will be repeated; if yes, we pop out the top of $DivergenceScoreList$ (Line 13), and add the sample to $CSet$ (Line 14–15).

The rationale behind the algorithm is as follows. In each round, we want to select the one with the lowest divergence score. Because each value in $DivergenceScoreList$ is a lower bound, the real divergence score will be higher than the value in the list. If we can select a real divergence score that is smaller than all the lower bound values, the selected score will be automatically smaller than all other real divergence scores. Such lazy updates will help us save time in the calculation.

Algorithm 1 The Algorithm of Growing Causality Clusters.

```
Input:
    Target Dataset: $TSet$
    Target Causality Cluster: $CSet$ (Initialized with a seed)
    Causality Cluster Size: $size$ (Intended cluster size)
Process:
 1: Create an empty, automatically-sorted $DivergenceScoreList$.
 2: for $i$ in $TSet$ do
 3:     $d = calculateDivergenceScore(CSet, i)$;
 4:     $DivergenceScoreList.put(d)$;
 5: end for
 6: for $iter$ in $[0 : size - 1]$ do
 7:     while $DivergenceScoreList.get().getCSet()! = CSet)$ do
 8:         $d = DivergenceScoreList.pop()$;
 9:         $i = d.getSample()$;
10:         $new\_d = calculateDivergenceScore(CSet, i)$;
11:         $DivergenceScoreList.put(new\_d)$;
12:     end while
13:     $d = DivergenceScoreList.pop()$;
14:     $i = d.getSample()$;
15:     $CSet.add(i)$;
16: end for
```

4.2.3 Causality Determination. In the third step, KARMA determines whether a causality cluster is polluted as well as the size of the cluster. Now, let us introduce the details about how the determination works. The detection accuracy of a machine learning model can be defined as a function: $accuracy = f(M, S_{oracle})$, which takes M the machine learning model and S_{oracle} the oracle set as inputs. After unlearning a causality cluster of a certain size, the detection accuracy can be represented as $accuracy = f(M', S_{oracle})$ where M' is the new machine learning model and equals $unlearn(M, seed, size)$. Further, if we substitute M' in the $accuracy$ equation, we obtain the following: $accuracy = f(unlearn(M, seed, size), S_{oracle})$. M is generated from $S_{training}$, another constant. Therefore, we can simplify the f to a single variable function $accuracy = g(size)$.

Then, let us discuss the single variable function $accuracy = g(size)$. If starting from $size$ as one, the detection accuracy increases in the first place, KARMA will unlearn more polluted samples similar to the first seed. As the cluster grows, the increasing speed of the accuracy decreases until the accuracy reaches a peak. The reason is that the polluted samples close to the cluster will become fewer, when including more polluted samples into the cluster. Then, the detection accuracy starts to decrease, because KARMA will include unpolluted samples into the cluster. At contrast, if starting from $size$ as one, the detection accuracy decreases in the first place, KARMA is likely to encounter unpolluted, normal samples that should remain in the training set. To sum up, our goal is to first identify whether the cluster contains polluted data by observing the detection accuracy changes, and then find the corresponding size to the peak of the detection accuracy. Therefore, the task boils down to find the peak value (the first local maximum close to the zero point) of a single, discrete-value variable function ($accuracy = g(size)$) and its corresponding variable ($size$) value.

Note that the first local maximum is sufficient, because KARMA will sift through every training sample that is not included in a cluster. That is, all the training samples will be inspected by KARMA. At contrast, if KARMA tries the second or the global maximum, many unpolluted, normal samples may be mistakenly considered as polluted, thus influencing the overall accuracy.

Next, let us solve the problem of finding the peak of the single-variable function. In particular, KARMA needs to find the first local maxima starting from a cluster with the size as zero. The problem is not as straightforward as finding a local peak of a normal single-variable function, because the computation of one value in our function is very expensive. That is, KARMA needs to feed all the samples of the oracle set into the machine learning engine, obtain the results, and then calculate the accuracy. The detection of one sample with a machine learning engine is already expensive, and our computation time needs to multiply the size of oracle set, making the

Algorithm 2 The Algorithm of Causality Determination.

```
Input:
    Initial Step Value: $M$
    The Set Size: $Size$
    The Initial Seed: $seed$
    The Detection Accuracy Function after Unlearning a Cluster of Size $x$ given a $seed$: $g(x) = f(unlearn(M, seed, size), S_{oracle})$
Process:
 1: $start = 0$
 2: $end = Size$
 3: $i = M$
 4: if $g(i) < g(0)$ then
 5:     Exit {Note: the cluster is considered unpolluted in this iteration.}
 6: end if
 7: while $start < end$ & $i < end$ do
 8:     if $g(i) < g(i - M)$ then
 9:         $start = max(i - 2M, start)$
10:         $end = i$
11:         $M = round(sqrt(M))$
12:         $i = start$
13:     end if
14:     $i = i + M$
15: end while
16: return a cluster with size $i$.
```

computation even more expensive. Therefore, we cannot afford calculating the detection accuracy for each point of the single-variable function.

Faced with this, we adopt two techniques to make the search faster. First, Karma starts from a coarse-grained search, narrows down the interval with the peak, and then performs a more fine-grained search within the interval. The target interval becomes smaller each time, and Karma does not need to waste our time upon these intervals that do not contain the peak.

The detailed process is below. and shown in Algorithm 2. At Line 1–3, all the variables are initialized. At Line 4–6, Karma first judges whether the causality cluster is polluted by observing the detection accuracy changes: If the accuracy increases, the cluster is polluted and Karma continues; otherwise, Karma stops here. Then, Karma At Line 7–15, Karma starts from a cluster with zero size, and each time the size of the cluster is incremented by M. If the accuracy starts to decrease (Line 8), Karma will combine the current interval and the one before together, and start a new search with a smaller incremental step as $\sqrt{M}$ within the new interval. The search will stop once the target interval shrinks to only one point. Because $g(Size) = 0 < g(0)$, Karma could expect that the accuracy has to decrease in between 0 and $Size$. However, to make the algorithm deterministically stop, we also put another condition to stop the algorithm once there is no target interval found. Note that the g function in Algorithm 2 remembers the value inquired before, and does not probe the learning model for accuracy with cached values.

Second, Karma can reuse the detection results of the oracle set in the computed $g(size)$ function to make the calculation of other $g(x)$ (where x is an unknown size) faster. For example, if Karma has calculated $g(x - 1)$, now Karma needs to calculate $g(x)$, and the difference between a cluster of size x and $x - 1$ is a sample with features $F1$ and $F2$. Therefore, the detection results of samples in the oracle set without $F1$ and $F2$ does not change between $g(x)$ and $g(x - 1)$. That is, Karma can reuse the detection results of these samples produced for $g(x - 1)$ in the calculation of accuracy for $g(x)$.

In practice, Karma may not find a $g(x - 1)$ that is so close to $g(x)$. Karma needs to find another $g(y)$ to minimize $|y - x|$, and then obtain the features in all the different samples between $g(x)$ and $g(y)$. Then, the samples in S_{oracle} with these features will be tested against M, and Karma will calculate the accuracy based on part of the detection results in $g(y)$ and the rest in $g(x)$.

Algorithm 3 The Algorithm of the Unlearning Module.

```
Input:
    Training Set: $S_{training}$
    Oracle Set: $S_{oracle}$
    Learning Model: $M$
Process:
 1: $S_{misclassified} = findMisclassified(M, S_{oracle})$
 2: $S_{seeds} = causalitySearch(S_{misclassified}, S_{training})$
 3: $S_{tmp} = S_{training}$
 4: $S_{clusters} = NULL$
 5: while $S_{tmp}! = NULL$ do
 6:     $seed = S_{seeds}.pop()$
 7:     if $seed = NULL$ then
 8:         $seed = S_{tmp}.get()$
 9:     end if
10:     $cluster = causalityDetermination(seed, S_{training}, M)$
11:     if cluster is polluted then
12:         $S_{clusters}.push(cluster)$
13:     end if
14:     $S_{tmp}.remove(cluster)$
15: end while
16: while $i$ in $S_{clusters}$ do
17:     $accuracyDelta = accuracy(unlearn(M, i), S_{oracle}) - originalAccuracy$
18:     if $accuracyDelta > 0$ then
19:         $M = unlearn(M, i)$ {Note: $i$ is selected to remove.}
20:     end if
21: end while
```

4.3 Causality Removal (Unlearning)

In this part of the section, we present the unlearning module in Algorithm 3. First, a list of seeds is generated by the causality search module based on the learning model, misclassified data, and training data (Line 1–2). Next, the algorithm sets the initial value of the working set S_{tmp} to be $S_{training}$ (Line 3). Starting from each seed (Line 6), the causality determination module decides the cluster size and forms a cluster (Line 10). The cluster, if polluted, is pushed to a set $S_{clusters}$ (Line 11–13) and removed from our working dataset S_{tmp} (Line 14). Then, the algorithm will repeat forming clusters until the working set is empty (Line 5). During the repetition, if the algorithm runs out of seeds, it randomly select a sample from the the working set as the seed (Line 7–9).

Then, the algorithm goes through the generated list of clusters of the training set (Line 16–20). Specifically, the algorithm calculates the current accuracy delta (Line 17), ensures that each cluster does cause the accuracy to increase (Line 18), and then unlearns the cluster from M (Line 19). Next, the entire algorithm (Line 1–21) is repeated based on a new M with polluted clusters unlearned and a new $S_{training}$ with polluted clusters removed. That is, the seed finder will find a new list of seeds fed into the peak finder, and then the algorithm will have a new list of clusters, and find the cluster to unlearn.

The iteration of Algorithm 3 is stopped based on two possible conditions. First, if the detection accuracy meets the expected value of the administrator of the learning model, the algorithm will stop. That is, the produced new learning model will be enough for use in the view of the administrator. Second, if the algorithm cannot find any new clusters to unlearn (or precisely no clusters that cause the detection accuracy to increase), the algorithm will also stop. That is, in this case, ideally, all the polluted data samples are removed from the training set, and unlearned from the learning model. In practice, as shown in the evaluation, we may still have a small number of samples, such as less than one or two percent of polluted data, scattered in the training set, however such polluted samples will have little impacts on the learning model as evident by the fact that the detection accuracy goes back to the vanilla value. The reason is that learning model itself is somehow robust to a small number of erroneous data, especially when they are not in clusters.

5 IMPLEMENTATION

Our prototype implementation of KARMA framework contains 3,009 lines of Python code. In particular, the implementation of divergence score contains 330 lines of code, the cluster forming 324 lines of code, the core of KARMA 1,968 lines of code, and other parts (such as the interface with learning model) 717 lines of code. The prototype implementation uses stand-alone files that provide APIs to be interacted with other learning systems. The main interaction (unless otherwise specified in Section 6.8 where we use an SVM-based spam filter and another Bayes-based JavaScript malware detector) is with SpamBayes [6], a naïve Bayes classifier capable of identifying spam and ham (non-spam) emails. We choose SpamBayes, because there is a published paper [33] documenting how to pollute SpamBayes, and as shown in Section 6 we will follow the paper to launch dictionary attacks. We believe that using the same learning system for data pollution documented in the literature will greatly reduce any potential bias.

6 EVALUATION

In the section of evaluation, we are going to evaluate our prototype implementation of KARMA framework.

6.1 Attacks and Datasets

In this subsection, following the two attack scenarios described in the threat model of Section 2, we present how we evaluate KARMA against these two attacks. Our vanilla dataset is a publicly available spam dataset called Enron-Spam [2, 41] consisting of 517,401 emails. Nine tenths of the vanilla dataset, i.e., 465,661 emails, are used for the vanilla training set, and the rest will be used for the oracle and testing dataset.

Mislabelling Attacks. In this attack scenario, as detailed in Section 2, the administrator relies on crowdsourcing workers to label training dataset, and the attacker's capability is limited to altering the labels. We assume that the administrator randomly assign all the training emails to 1,000 crowdsourcing workers, and each worker will label about 466 emails. Among the workers, some have malicious intents to mislabel emails. In the experiment, we assume that the ratio of malicious workers ranges from 5% to 30%.We categorize malicious workers below:

- *Blind Mislabelling.* A blind mislabelling worker will mislabel all the emails assigned to her. We collect 6 polluted datasets with different malicious worker ratios.
- *Targeted Mislabelling.* There are two subtypes of targeted mislabelling workers. First, the worker only intentionally mislabels one class of emails, either spam as ham or ham as spam. We collect 24 polluted datasets with different malicious worker ratios. Second, the worker only mislabels emails with certain features (words). In the experiment, we choose several popular words, such as "laptop", "schedule" and "magazine", and collect 12 polluted datasets.

Additionally, we also collect 3 datasets mixing blind and targeted mislabelling: one blind and spam as ham, one blind and ham as spam, another blind and mislabelling with certain features. In sum, 45 polluted datasets are collected from mislabelling attacks. The pollution technique is effective as demonstrated in the final accuracy of learning model ranging from 15.2% to 51%.

Injection (Dictionary) Attacks. In this attack scenario, as detailed in Section 2, the attacker has more control over the spam emails but no control over ham emails, because a honeypot collects spams from spammers. Particularly, we launch a special injection attack, i.e., the dictionary attack proposed by Nelson et al. [33] to generate spams with many injected dictionary words, i.e., spams with crafted features. The purposes of a dictionary attack are twofold. First, if legitimate dictionary words are considered as spam features, many ham emails with such words will be misclassified. Second, if non-spam words are considered as spam features, the real spam features might be buried and not selected by the learning model.

Based on the aforementioned dictionary attacks, we collect 50 polluted datasets. The polluted emails are about 0.5% to 4% of the entire training set, and the final detection accuracy after pollution ranges from 59.3% to 81.4%. Specifically, in each attack, an attacker can craft spams based on four parameters: number of clusters (*NC*), cluster size (*CS*) that is the number of emails in a cluster, number of words (*NW*) that is the length of each email, and maximum deviation of word set between two emails in one cluster (*D*). In our experiment, *NC* is selected randomly between 1 and 26 for each dataset, *CS* randomly between 1 and 1000 for each cluster, *NW* randomly between 1000 and 2000 for each cluster, and *D*% randomly between 10% and 50% for each dataset.

Oracle and Testing Datasets. The samples used for oracle and testing datasets are further divided into two parts. The first part with 25,870 emails is used as an independent, third dataset (S_{test}) for validation purpose such as measuring the accuracy of the learning model. All the accuracies reported in this section are using this dataset.

The second part is used to generate S_{oracle}, the oracle set. We feed emails in this part (about 1/20 of the entire dataset) into each polluted learning model, and include the misclassified emails and one tenth of the correctly classified emails into S_{oracle}. Based on different attacks, S_{oracle} varies, but roughly contains 5K–15K emails.

6.2 Comparing with Learning-based Approaches

In this part of the section, we evaluate two naïve learning-based approaches and show that they cannot repair a polluted learning model in our experiment. Specifically, the first approach is to construct a learning model with only S_{oracle}, and the second is to improve the already polluted model trained from a polluted $S_{training}$ by learning S_{oracle} incrementally.

First, let us see whether S_{oracle} can be used to train a learning model with good accuracy. Because S_{oracle} varies for different attacks, we randomly select five S_{oracle}, train a learning model, and test the learning model against S_{test}. The accuracies are all below 60%. The reason is that S_{oracle} is a good representation of misclassified emails, but not correctly classified emails. In sum, the take-away for this experiment is that although S_{oracle} can be used to find the polluted data and fix the learning model, S_{oracle} cannot be used as a stand-alone dataset for training purposes.

Second, we incrementally learn S_{oracle} based on the polluted learning model, i.e., the new training set of the model will be the polluted dataset plus S_{oracle}. We then evaluate whether the polluted learning model can be repaired. We call this naïve approach corrective learning in this paper. Similar to the previous experiment, we choose five random S_{oracle} for the experiment. The results show that the accuracies are all below 70%.

The reason is as follows. Due to the small size of S_{oracle}, the correctly labelled samples in S_{oracle} cannot negate the pollution effects. Because there is no prior work on this corrective learning approach and we do not have enough number of additional correctly labeled samples, we are not sure how to determine the number of correctly labelled samples to learn. One thing worth noting is that the number is very challenging to determine: Fewer samples do not negate the pollution effects, but more will cause overfitting. Further investigation is out of scope of the paper.

Now from a high level, let us compare corrective learning and KARMA. We believe that in this specific problem, KARMA is superior to corrective learning due to the following two reasons. (i) KARMA not only repairs the polluted model, but also helps to block future pollution attempts, because the administrator can find the pollution source based on the misclassification cause. As a comparison, corrective learning only repairs the model, but not blocks future attempts. (ii) When the size of S_{oracle} increases, KARMA only becomes more effective in repairing the polluted model, but corrective learning may correct the polluted learning model to the other extreme, i.e., causing overfitting.

6.3 True Positives and Negatives

In this part of the section, we evaluate the true positives and negatives of KARMA using all the attack datasets mentioned in Section 6.1. True positives are defined as the percentage of polluted identified, and true negative the percentage of unpolluted remained.

The true positives of KARMA against all the datasets are higher than 98.0% for the lowest, 99.2% for the median, and reaching 99.97% for the highest. The very high true positives prove the effectiveness of KARMA in identifying polluted samples altered or injected by adversaries. The rest of less than 1–2% polluted data is scattered in the training data, and has little impacts on the learning model as evident by our final detection accuracy, which is very close to the vanilla (±0.9%) after removing the polluted shown in Section 6.4.

The true negatives are higher than 85.5% for the lowest, 90.8% for the median, and reaching 94.3% for the highest. Such true negatives are partially due to the fact that the original dataset contain some noisy data. Such noisy data can be classified into two categories. First, if one unlearns some unpolluted, noisy data directly from the training set, the detection accuracy also increases. Second, some noisy data might not have impacts on the detection accuracy of the learning model due to its small size. However, when these noisy data is combined with a large, polluted data cluster, the combination may have impacts of the detection accuracy. As evident by our final detection accuracy, to add or remove such noisy data has very little impacts on the learning model. In the future, as discussed in Section 7, such noisy data, just like active learning, may guide the learning algorithm to query the administrator or other oracles to label new similar data samples to further improve the model.

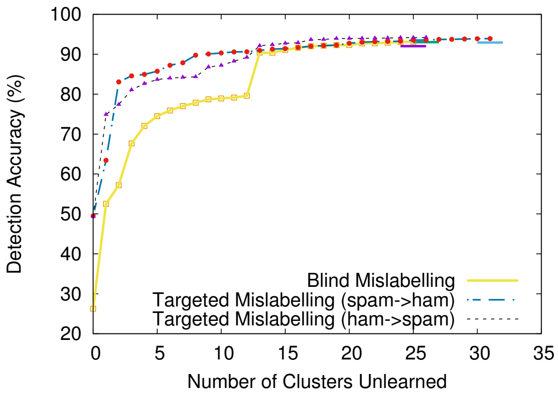

Figure 2: The Detection Accuracy of Learning Models Polluted by Mislabelling Attacks when Applying KARMA (The horizontal line at the end of each curve indicates the vanilla accuracy for that dataset).

6.4 Effectiveness in Accuracy Repair

In this section, we show the effectiveness of KARMA in the metrics of detection accuracy.

Mislabelling Attacks. We first evaluate KARMA against mislabelling attacks. Figure 2 shows the detection accuracy when clusters are unlearned from learning models polluted by three mislabelling attacks. The x-axis is the number of clusters unlearned, and the y-axis is the detection accuracy. Due to the space limit in the figure, we select three representative attacks: one with blind mislabelling attacker, and two with targeted mislabelling attacker (ham as spam and spam as ham respectively). The ratio of malicious workers is 15% for all three attacks. Note that the results for the rest mislabelling attacks are similar to the ones shown in the figure: The number of clusters unlearned may vary a little, but the trend and final accuracy differences are very close.

In Figure 2, we use a horizontal line at the end of each curve to indicate the vanilla accuracy for that dataset. These vanilla accuracy values are 92.02% (blind mislabelling), 93.03% (ham as spam), and 92.93% (spam as ham) from the left to the right for the horizontal lines in the figure. The final accuracies after KARMA are very close to the vanilla ones, within ±0.9%. Note that some of the final accuracies are higher than the vanilla, because the vanilla dataset without pollution also contains some errors. If we perform KARMA upon the vanilla dataset, the accuracy can also increase by 0.1%–0.2%.

Dictionary Attacks. In this subsection, we evaluate KARMA against the dictionary attacks. Figure 3 shows the detection accuracy over the iteration when clusters are unlearned. The x-axis is the number of clusters unlearned, and the y-axis is the detection accuracy. The vanilla detection accuracy is 94.825% in our experiment. Similar to mislabelling attacks, the final accuracies after KARMA are close to the vanilla one, within ±0.3%. We only include the results of five datasets out of fifty because of the space limit of the graph. The parameters for the five dictionary attacks are as follows: *NC* is

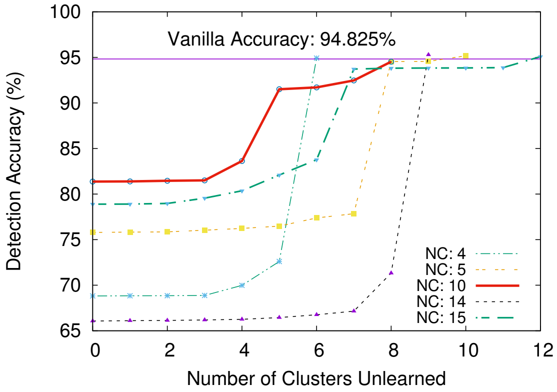

Figure 3: The Detection Accuracy of Learning Models Polluted by Dictionary Attacks when Applying Karma (*NC* means the number of clusters injected in dictionary attacks, maximum deviation *D*% equals 30% for all five datasets, and both cluster size (*CS*) and number of words (*NW*) vary for different clusters in each dataset).

marked in the figure, *D*% is 30% for all five, and both *CS* and *NW* are selected randomly for each cluster in the datasets. The results for the rest attacks are very similar to the ones depicted in Figure 3, and our observation below also applies to the rest.

First, the number of clusters (NC) in each dictionary attack roughly matches with but does not equal to the number of clusters unlearned. In some cases when two separate injected clusters are close enough, Karma may consider them as one; at contrast, due to the existence of parameter *D*, one cluster may be considered as two separate clusters in Karma.

Second, one major difference of Karma between mislabelling and dictionary attacks is the number of cluster unlearned (the x-axis in Figure 2 and 3). The number of clusters unlearned for mislabelling attacks is larger than the one for dictionary on average. The reason is that dictionary attacks are launched with clusters (and some variations within each cluster) as opposed to mislabelling attacks, which flip over the labels of emails randomly assigned by an administrator. That is, the Karma algorithm needs to find and form clusters when finding polluted samples for mislabelling attacks.

6.5 Effects of Oracle Sets

In this section, we are going to evaluate how oracle sets affect Karma. We evaluate Karma using two types of oracle sets: one with less misclassified emails and one with only misclassified emails.

First, we randomly pick a mislabelling attack, and then reduce the number of misclassified emails in the oracle set to 1/2 ($S_{oracle,1/2}$), 1/3 ($S_{oracle,1/3}$) and 1/10 ($S_{oracle,1/10}$). Both true positive and negative are measured to evaluate the effect of these oracle sets. The original true positive is 98.2%, and true negative 90%. The true positive drops to 98% for $S_{oracle,1/2}$, 96.9% for $S_{oracle,1/3}$, and then 83.2% for $S_{oracle,1/10}$. The true negative stays the same when the number of misclassified reduces.

This experiment tells us that Karma requires that the oracle set contains some amount of misclassified samples to make Karma effective. Now let us answer how to determine the size of S_{oracle}. The administrator can first use a small number of misclassifications as the oracle set and repair the learning model. If users still report misclassifications for the repaired learning model and the misclassifications are verified, the administrator can form a new oracle set and further repair the model. The feedback loop can be repeated for many times.

To prove this, we first apply Karma with $S_{oracle,1/10}$, and repair the learning model. Then, we obtain another $S_{oracle,1/10}$ based on the current misclassifications, and repair the learning model again. When we repeat the process for three times, the true positive arises to 98.2%, the same as the original with S_{oracle}.

Second, we adopt a new oracle set that only contains all the misclassified emails but no correctly classified emails. All the dictionary and mislabeled attacks are used to test the effectiveness of Karma under this circumstance. The true positives are between 97.4% and 99.2%, and the true negatives are between 79.3% and 85.7%. It is worth noting that when we only include misclassified emails, the true positives stay the same, but the true negatives drop a little. The reason is as follows: Karma does not know anything about correctly classified emails, and therefore may make some mistakes. Therefore, in order to maintain both true positive and true negative, we recommend that the administrator forms S_{oracle} using both misclassified and some correctly classified emails.

6.6 Performance Overhead

In this part of the section, we evaluate the performance of Karma. Let us first take a look at the theoretical value. In each iteration, the time spent on the seed finder is a linear function with regards to the size of misclassified data. The time spent on the peak finder – which may be invoked many times – is with regards to the size of training data, because each cluster, once decided, is deducted from the working set. The time spent on the unlearning module is with regards to the number of clusters. The number of clusters is relatively small compared to the training data size, and the size of misclassified data is smaller than the size of oracle set, and thus much smaller than the size of training data. Therefore, the time complexity of Karma in each iteration is $O(N)$, where N is the size of training dataset, and the overall time complexity is $O(kN)$, where k is the number of iterations.

Our empirical evaluation show that the median overhead of performance is 6.21 times of the training time with the maximum as 34.1 and the minimum as 3.81. It is worth noting that Karma can be incrementally deployed, i.e., any partial outputs can be used by the administrator to repair the polluted learning model. For example, if the administrator sets a satisfactory detection accuracy as 90%, the maximum performance overhead is only 15.1 times of the training time.

Note that although the performance of Karma, an offline analysis tool, is not ideal, Karma is faster than the most naïve method that searches the training set by brute force. The performance overhead is exponential, because the method needs to include all the possible subsets of the training set. As a comparison, our Karma brings down the performance overhead from exponential to linear in the number of training points.

6.7 Effects of Divergence Score Definition

In this part of the section, we study the effect of the divergence score definition upon KARMA. As mentioned in Section 4.2.1, any non-negative and symmetric definition of divergence score – which represents similar contributions of samples made to a learning model – can be used in KARMA. Thus, we compare three definitions: the one defined in Section 4.2.1 ($D1$), Euclidean distance, and another definition ($D2$) taking account into the feature frequency defined in Appendix A.

Now let take a look at how $D1$, $D2$, and Euclidean distance affect KARMA. We first use Euclidean distance, which performs the worst among the three. In many cases, KARMA with Euclidean distance does not converge, i.e., the final detection accuracy cannot reach the vanilla level. The reason is that when computing the Euclidean distance between two samples, we not only consider the common features between these two, but also the different features. For example, if two samples share many common features, but also have plenty of different features, the Euclidean distance between these two samples is large, contradicting with the fact that these samples make similar contributions to learning model in their common features.

Next, we use $D2$ in KARMA. Unlike Euclidean distance, KARMA with $D2$ converges, and the final detection accuracy is within ±1% of the vanilla accuracy. The major difference between KARMA with $D1$ and $D2$ lies in the true positives and negatives. Evaluated against the dictionary and mislabelling attacks, KARMA with $D2$ has a median true negative as 92.3% ranging from 90.2% to 96.5%, and a median true positive also as 92.3% ranging from 88.0% to 95.2%. That is, the true negative of $D2$ is higher than the one of $D1$, but the true positive is lower. Like all other researches, there is a trade-off between the true positives and negatives in KARMA: When we try to boost one, the other is lowered correspondingly.

To sum up, for all three definitions, $D1$ performs the best and Euclidean distance performs the worst. This order aligns with how these definitions deal with common and different features among samples, especially different ones. $D1$ only considers common features but ignore different ones all the time. $D2$ only considers common features in the calculation of divergence scores, but when the cluster grows, different features are included in the feature list. Euclidean distance considers both common and different features among samples in the calculation and the cluster growing stage.

6.8 Integrating Other Learning Systems

In this section, we evaluate two other learning systems to show the generality of KARMA. The first system is a SVM-based spam detector, showing that KARMA works with other learning algorithms, and the second is a Bayes-based JavaScript malware detector (in Appendix B), showing that KARMA works with other application scenarios.

We show how to integrate KARMA with another support vector machine (SVM) based spam filter. Because we cannot find an open-source filter written in Python, we implement a version of a spam filter with approximately 3000 lines of Python code by following what has described in an online machine learning course [1, 4] taught in Stanford University. Here are some details of the spam filter that we implemented. The SVM library that we adopt is LIBLINEAR [3, 21], a popular open source tool that supports linear support vector machines. The training phase of our spam filter can be divided as three steps: preprocessing, feature extraction, and training. In the preprocessing, we normalize each email by nine tactics, such as removing non-words and replacing $ with 'dollar' (Details can be found in the online course). Then, we extract features based on 1,900 spam words found commonly in SpamAssassin dataset [5], and train the LIBLINEAR classifier. Our spam filter is consisted of ˜3,000 lines of Python code, and achieves 95.88% vanilla detection accuracy.

Next, we integrate the KARMA with this SVM-based spam filter (LIBLINEAR supports unlearning, called decremental learning), and evaluate KARMA with polluted datasets. The results are very similar to the one with SpamBayes and show that KARMA can successfully restore the detection accuracy to the vanilla value with less than 1% difference.

7 DISCUSSION

We are now discussing several important problems of KARMA.

Robustness of KARMA to Attacks. We are discussing the robustness of KARMA to attacks in adversarial environment. There are two cases to discuss: (i) attacks during the repairing process, and (ii) attacks after the repairing process.

First, KARMA is robust to attacks during the repairing, because there exists an oracle-in-the-loop feedback that can enhance the security and mitigate potential attacks. Then, KARMA performs a complete though prioritized search over the training set, i.e., KARMA goes over every training sample in the search.

Second, KARMA is robust to attacks after the repairing, because the administrator will block the data pollution attacks from the same source. For example, the administrator can block the turk who intentionally mislabels data in the training set and only adopt results from trusted turks. We understand that attackers may compromise new turks, but this at least reduces the attack surface.

False Positives and Negatives. In addition to the false positives and negatives of KARMA, we believe that the oracle-in-the-loop feedback can also help to improve false positives and negatives. Specifically, the administrator will mitigate false positives by manually inspecting the samples to unlearn and the users will report false negatives if the model is not fully repaired.

Effects of KARMA on Unpolluted Model. If the learning model is unpolluted, KARMA might still help to improve the model. Say, for example, if the misclassification is caused by other reasons, such as lack of data, the cause found by KARMA may help the administrator or the developer to introduce new samples that can clearly distinguish the misclassified and its cause. This, however, is considered as beyond scope of the paper, and one may refer to the literature [25, 32, 38] on introducing new samples.

Overfitting and Underfitting. The general problems of overfitting and underfitting are orthogonal to the paper, and one may refer to the literature [16, 40] for the problem. To the best of our knowledge, KARMA does not cause additional overfitting or underfitting issues, which we will discuss from two aspects: explanation and empirical evaluation.

First, as shown in our deployment model, before an administrator repairs the learning model with the misclassification cause, the administrator will verify that samples found by KARMA are indeed polluted. That is, the administrator will preserve correctly-labeled samples to avoid overfitting and underfitting.

Second, our restored detection accuracy—without any verification from the administrator—are very high against an independent dataset showing no underfitting or overfitting. If there is either underfitting or overfitting, the detection accuracy against the independent dataset will be much lower. That is, in empirical evaluation, we do not observe any overfitting or underfitting.

8 RELATED WORK

In this section, we will discuss related work. We start from a general problem, adversarial machine learning, in Section 8.1, and then discuss existing defenses to data pollution attacks in Section 8.2. After that, we discuss other similar techniques in Section 8.3.

8.1 Adversarial Machine Learning

The problem that KARMA solves, i.e., the data pollution, belongs to a broad research topic, called adversarial machine learning [8, 26]. Adversarial machine learning defines the behavior of machine learning models under the existence of adversaries. In particular, prior works classify such attacks into two major categories: causative attacks where an attacker has access to the training set, and exploratory attacks where an attacker can only craft samples to probe or explore the learning model.

8.1.1 Causative Attacks. There are many causative attacks, or called data pollution defined in the literature. Perdisci et al. [35] attacks PolyGraph [34] by injecting well-crafted invariants and misleading the signature generation. Particularly, the attacker sends the crafted traffic to a honeypot collecting worm traffic and such traffic will be picked up by automatic worm signature generation tools, such as PolyGraph. Similarly, Nelson et al. [33] pollute SpamBayes by injecting emails with dictionary words. According to them, only 1% such emails in the training set can cause SpamBayes to misclassify an email with 90% probability. Other than Bayes classifier, Biggio et al. [9] target support vector machine (SVM) and study the corresponding pollution tactics. Fumera et al. [24] evaluate pattern classification systems in general, and conclude that all of them are vulnerable to data pollution attacks. Wang et al. [45] show that crafted training samples can mislead the machine learning classifier detecting malicious crowdsourcing workers.

All these causative attacks, or called data pollutions, serve as a good motivation for the KARMA. For example, the pollution technique proposed by Nelson et al. [33] has been used in our evaluation.

8.1.2 Exploratory Attacks. Exploratory attacks are out of scope of KARMA. For completeness, we still briefly talk about such attacks in adversarial machine learning. Exploratory attacks can be further classified as two sub-categories: model inversion [12, 23] where an attacker infers training data samples based on the learning model, and data evasion [7, 13, 29, 44, 45] where an attacker crafts samples to evade the learning model. Model inversion is within the scope of the machine unlearning proposed by Cao et al. However, because the samples to unlearn are known in such scenario, which is the private data identified by the user, it is not necessary to apply KARMA. Data evasion is also beyond the scope of the machine unlearning paper.

8.2 Defense of Data Pollution

In this part of the section, we introduce prior works that defend data pollution. Such works can be divided into two categories: filtering polluted samples before training, and training a robust learning model. First, both Brodley et al. [10] and Cretu et al. [17] introduce an additional filtering layer to get rid of polluted samples. Brodley et al. use majority consensus among different techniques, and Cretu et al. adopt sanitization with micro-models in a voting scheme. Similarly, Newsome et al. [34] cluster samples beforehand so that outliers, such as polluted samples, can be filtered. Second, Dekel et al. [19] minimize the damage that an attacker could make by formulating the learning as a linear program and using an online-to-batch conversion. Bruckner et al. [11] model the learner and the attacker as a game with Nash equilibrium.

Techniques that filter polluted samples before training or make learning model robust are orthogonal to and can be combined with KARMA. If polluted samples bypass these approaches as evident by new pollution attacks [19, 35], KARMA serves as a remedy approach that repairs polluted learning models and brings the model to healthy states.

8.3 Other Similar Techniques

Machine unlearning is a technique proposed by Cao et al. [14] that makes learning systems forget what they have learned before. Cao et al. convert a learning algorithm to a special form in statistical query learning [28], which consists of a small number of summations. The learning algorithm only depends on these summations, which are the sum of some efficiently computable transformation of the training data samples. Therefore, to unlearn a training sample becomes easy: One just needs to subtract the transformations of that sample from all the summations, and then update the learning model.

Machine unlearning only removes samples from a learning model when the samples are specified. At contrast, KARMA tries to find what data to remove from a learning model. As discussed in Section 4.1, KARMA does utilize machine unlearning technique by Cao et al., but is compatible with other incremental or decremental machine learning [15, 20, 22, 37, 42, 43]. The reason we use machine unlearning is that machine unlearning is general, which makes KARMA general as well.

BoostClean [31] detects and repairs domain value violations, i.e., an attribute value is outside of its value domain, using statistical boosting. As a comparison, their repairing is to correct the prediction results of a machine learning model, but KARMA is to correct the machine learning model itself.

Koh et al. [30] propose using influence functions to estimate the influence of training samples upon the prediction results. Therefore, their approach can be used to prioritize the administrator's efforts in inspecting the training set. The advantage of their approach is that they do not need to remove any training samples and observe causality. As a comparison, KARMA is more accurate and further reduces the administrator's efforts because KARMA directly observes the effects by unlearning samples. For example, according to the evaluation of Koh et al., the administrator needs to inspect 30% of training data if 10% is polluted. At the same time, KARMA also improves the efficiency from retraining models from scratch.

9 CONCLUSIONS

In this paper, we present a new technique, called causal unlearning, which actively searches the training set for the misclassification cause in an iterative manner and then removes the cause to repair a polluted machine learning system.

We implemented a prototype of KARMA for causal unlearning, and evaluated it using SpamBayes, another SVM-based spam filter and a JavaScript malware detection engine. Our evaluation results show that KARMA can successfully identify the misclassification cause, i.e., polluted samples, with true positive ranging between 98.0% and 99.97% and true negative ranging between 85.5% and 94.3%. Further, KARMA can repair polluted learning model and restore the learning model's accuracy to the vanilla value with less than 1% differences.

10 ACKNOWLEDGEMENT

We would like to thank Nicolas Papernot (our shepherd), Alex Yang and anonymous reviewers for their helpful comments and feedback. This work was supported in part by National Science Foundation (NSF) grants CNS-15-63843, and CNS-15-64055. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF.

REFERENCES

[1] CS229 machine learning (stanford university). http://cs229.stanford.edu/materials/ML-advice.pdf.

[2] Enron-spam dataset. http://www.aueb.gr/users/ion/data/enron-spam/index.html.

[3] Liblinear. https://www.csie.ntu.edu.tw/~cjlin/liblinear/.

[4] Machine learning (stanford university). https://www.coursera.org/learn/machine-learning.

[5] Spamassassin dataset. https://spamassassin.apache.org/publiccorpus/.

[6] SpamBayes. http://spambayes.sourceforge.net/.

[7] M. Q. Ali, A. B. Ashfaq, E. Al-Shaer, and Q. Duan, "Towards a science of anomaly detection system evasion," in *IEEE Conference on Communications and Network Security (CNS)*, September 2015.

[8] M. Barreno, B. Nelson, A. D. Joseph, and J. D. Tygar, "The security of machine learning," *Mach. Learn.*, vol. 81, no. 2, pp. 121–148, Nov. 2010. [Online]. Available: http://dx.doi.org/10.1007/s10994-010-5188-5

[9] B. Biggio, B. Nelson, and P. Laskov, "Poisoning attacks against support vector machines," in *Proceedings of International Conference on Machine Learning*, ser. ICML, 2012.

[10] C. E. Brodley and M. A. Friedl, "Identifying mislabeled training data," *Journal of Artificial Intelligence Research*, vol. 11, pp. 131–167, 1999.

[11] M. Brückner, C. Kanzow, and T. Scheffer, "Static prediction games for adversarial learning problems," *J. Mach. Learn. Res.*, vol. 13, no. 1, pp. 2617–2654, Sep. 2012.

[12] 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.

[13] 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*, ser. ACSAC, 2014.

[14] Y. Cao and J. Yang, "Towards making systems forget with machine unlearning," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, 2015.

[15] G. Cauwenberghs and T. Poggio, "Incremental and decremental support vector machine learning," in *Advances in Neural Information Processing Systems (NIPS*2000)*, vol. 13, 2001.

[16] J. Cheng and R. Greiner, "Learning bayesian belief network classifiers: Algorithms and system," in *Proceedings of the 14th Biennial conference*, 2001, pp. 141–151.

[17] 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*, ser. SP, 2008.

[18] 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.

[19] O. Dekel, O. Shamir, and L. Xiao, "Learning to classify with missing and corrupted features," *Mach. Learn.*, vol. 81, no. 2, pp. 149–178, Nov. 2010.

[20] 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*, ser. KDD, 2000.

[21] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, "Liblinear: A library for large linear classification," *The Journal of Machine Learning Research*, vol. 9, pp. 1871–1874, 2008.

[22] Ó. Fontenla-Romero, B. Guijarro-Berdiñas, D. Martinez-Rego, B. Pérez-Sánchez, and D. Peteiro-Barral, "Online machine learning," *Efficiency and Scalability Methods for Computational Intellect*, pp. 27–54, 2013.

[23] 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.

[24] G. Fumera and B. Biggio, "Security evaluation of pattern classifiers under attack," *IEEE Transactions on Knowledge and Data Engineering*, vol. 99, no. 1, 2013.

[25] D. J. Hsu, "Algorithms for active learning," Ph.D. dissertation, University of California, San Diego, 2010.

[26] 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*, ser. AISec, 2011.

[27] J. Juneau, J. Baker, F. Wierzbicki, L. Soto, and V. Ng, *The Definitive Guide to Jython: Python for the Java Platform*, 1st ed. Berkely, CA, USA: Apress, 2010.

[28] M. Kearns, "Efficient noise-tolerant learning from statistical queries," *J. ACM*, vol. 45, no. 6, pp. 983–1006, Nov. 1998.

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

[30] P. W. Koh and P. Liang, "Understanding black-box predictions via influence functions," *ICML*, 2017.

[31] S. Krishnan, M. J. Franklin, K. Goldberg, and E. Wu, "Boostclean: Automated error detection and repair for machine learning," *CoRR*, vol. abs/1711.01299, 2017. [Online]. Available: http://arxiv.org/abs/1711.01299

[32] C. Monteleoni, "Learning with online constraints: Shifting concepts and active learning," Ph.D. dissertation, Massachusetts Institute of Technology, 2006.

[33] 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*, ser. LEET, 2008.

[34] 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.

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

[36] S. Perez. Microsoft silences its new a.i. bot tay, after twitter users teach it racism. http://techcrunch.com/2016/03/24/microsoft-silences-its-new-a-i-bot-tay-after-twitter-users-teach-it-racism/.

[37] 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*, ser. ICANN, 2007.

[38] B. Settles, "Active learning literature survey," University of Wisconsin–Madison, Computer Sciences Technical Report 1648, 2009.

[39] S. Shen, S. Tople, and P. Saxena, "Auror: defending against poisoning attacks in collaborative deep learning systems," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 2016, pp. 508–519.

[40] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, Jan. 2014.

[41] V. M. Telecommunications and V. Metsis, "Spam filtering with naive bayes – which naive bayes?" in *Third Conference on Email and Anti-Spam (CEAS)*, 2006.

[42] 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*, ser. KDD, 2014.

[43] P. E. Utgoff, "Incremental induction of decision trees," *Mach. Learn.*, vol. 4, no. 2, pp. 161–186, Nov. 1989. [Online]. Available: http://dx.doi.org/10.1023/A:1022699900025

[44] 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.

[45] 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.

Appendices

A ANOTHER DIVERGENCE SCORE

In the appendix, we define another divergence score between a cluster and a sample called $D2$. To calculate $D2$, we need to maintain two lists: (1) all the features in the cluster, $< F1, F2, F3, \ldots >$, and (2) the number of occurrence of each features in the cluster, $< N1, N2, N3, \ldots >$. Then, we still obtain the common feature list between the cluster and the sample, $< Fk_1, Fk_2, \ldots Fk_j >$. This divergence score between the cluster and the sample is defined in Equation 3.

$$\frac{1}{j^2}\left(\frac{1}{Nk_1+1}+\frac{1}{Nk_2+1}+\ldots+\frac{1}{Nk_j+1}\right) \tag{3}$$

If the cluster only contains one sample, i.e., we want to compute the divergence score between two samples, Equation 3 boils down to $\frac{1}{2j}$, half of the divergence score defined in $D1$. Because divergence score is a relative value, the definitions in $D1$ and $D2$ of the score between two samples are consistent. Note that what the new divergence score definition introduces is the concept of frequency. When a feature occurs more in the cluster, the contribution to the divergence score between the cluster and a sample with the feature is smaller, as one divided the frequency of the feature plus one is smaller. At contrast, when a feature occurs less, the contribution to the score is larger. So, during clustering, the active unlearning algorithm tends to include samples with more high frequency features and less low frequency ones in the current cluster.

B EVALUATION ON A BAYES-BASED JAVASCRIPT MALWARE DETECTOR

In this section, we integrate KARMA with Zozzle [18], a JavaScript malware detection engine using Naïve Bayes. The purpose of the experiment is to show that KARMA works with not only spam detectors but also malware detectors. Because Zozzle is closed-source, we reimplement a Java version by following their paper, and obtain an implementation from Cao et al. [14] where they implement machine unlearning and evaluate the effectiveness. Their Zozzle implementation is based on Java, and we use Jython [27] to integrate our Python implementation of KARMA with their Zozzle. Together with their source code, we also obtain their The dataset that we use contain 142,350 real-world JavaScript malware samples from Huawei, JavaScript from top 10,000 Alexa web sites, and 15,520 polluted JavaScript. All other setups are similar to the setup of our previous experiment, we divide unpolluted samples into 10 equal parts: nine parts plus the polluted samples for training, and the rest equally divided for the oracle and the testing dataset.

The result shows that KARMA can successfully identify 98.9% polluted JavaScripts and restore the detection accuracy against the testing dataset to the vanilla value with less than 0.9% difference.