Paragraph: Thwarting Signature Learning by Training Maliciously

James Newsome[1], Brad Karp[2], and Dawn Song[1]

[1] Carnegie Mellon University
[2] University College London

Abstract. Defending a server against Internet worms and defending a user's email inbox against spam bear certain similarities. In both cases, a stream of *samples* arrives, and a *classifier* must automatically determine whether each sample falls into a malicious *target* class (*e.g.,* worm network traffic, or spam email). A *learner* typically generates a classifier automatically by analyzing two labeled training pools: one of innocuous samples, and one of samples that fall in the malicious target class.
Learning techniques have previously found success in settings where the content of the labeled samples used in training is either random, or even constructed by a helpful teacher, who aims to speed learning of an accurate classifier. In the case of learning classifiers for worms and spam, however, an *adversary* controls the content of the labeled samples to a great extent. In this paper, we describe practical attacks against learning, in which an adversary constructs labeled samples that, when used to train a learner, prevent or severely delay generation of an accurate classifier. We show that even a *delusive* adversary, whose samples are all correctly labeled, can obstruct learning. We simulate and implement highly effective instances of these attacks against the Polygraph [15] automatic polymorphic worm signature generation algorithms.

Key words: automatic signature generation, machine learning, worm, spam

1 Introduction

In a number of security applications, a *learner* analyzes a pool of samples that fall in some malicious *target* class and a pool of innocuous samples, and must produce a *classifier* that can efficiently and accurately determine whether subsequent samples belong to the target class. High-profile applications of this type include automatic generation of worm signatures, and automatic generation of junk email (spam) classifiers.

Prior to the deployment of such a system, samples in the target class are likely to include a number of distinguishing features that the learner can find, and that the classifier can use to successfully filter target-class samples from a stream of mixed target-class and innocuous samples. Before the wide deployment of automatic spam classification, spam emails often contained straightforward sales pitches. Likewise, as no automatic worm signature generation system has yet been widely deployed, all instances of a particular worm's infection attempts contain nearly an identical payload. The first generation of automatic signature generation systems was highly successful against these *non-adaptive* adversaries.

Once such a system is widely deployed, however, an incentive exists for *elusive* adversaries to evade the generated classifiers. We observe this phenomenon today because of the wide-spread deployment of spam classifiers. Senders of spam employ a variety of techniques to make a spam email look more like a legitimate email, in an attempt to evade the spam classifier [6]. Similarly, while worm signature generation systems are not yet widely deployed, it is widely believed that once they are, worm authors will use well known *polymorphism* techniques to minimize the similarity between infection payloads, and thus evade filtering by worm signatures.

In the case of worm signature generation we have a significant advantage: a worm infection attempt *must* contain specific exploit content to cause the vulnerable software to begin

executing the code contained in the payload. Further, the *vulnerability,* not the worm's author, determines this specific exploit content. Newsome *et al.* [15] showed that, for many vulnerabilities, messages that exploit a particular vulnerability must contain some set of *invariant* byte strings, and that it is possible to generate an accurate and efficient signature based on this set of byte strings, even if the rest of the worm's payload is *maximally varying*—that is, contains no persistent patterns.

Unfortunately, such an elusive adversary is not the worst case. In this work, we emphasize that these applications attempt to learn a classifier from samples that are *provided by a malicious adversary*. Most learning techniques used in these applications do not target this problem setting. In particular, most machine learning algorithms are designed and evaluated for cases where training data is provided by an indifferent entity (*e.g.,* nature), or even by a helpful teacher. However, in the applications under discussion, training data is provided by a *malicious* teacher.

Perdisci *et al.* [18] demonstrate that it is not sufficient for the learner to tolerate *random* noise (mislabeled training samples) in the training data. In particular, Perdisci *et al.* describe noise-injection attacks on the Polygraph suite of automatic worm signature generation algorithms [15], through which an attacker can prevent these algorithms from generating an accurate classifier. These attacks work by causing the Polygraph learner to use specially crafted non-worm samples as target-class-labeled (worm-labeled) training data. This type of attack is of concern when the initial classifier that identifies target-class samples for use in training is prone to false positives. Such an attack can be avoided by using a *sound* initial classifier to ensure that non-target-class samples cannot be mislabeled into the target-class training data. In the case of automatic generation of worm signatures, host monitoring techniques such as dynamic taint analysis [3, 4, 16, 23] can prevent such mislabeling, as they reliably detect whether the sample actually results in software being exploited.

In this work, we show that there is an even more severe consequence to training on data provided by a malicious teacher. We show that a *delusive*[1] adversary can manipulate the training data to prevent a learner from generating an accurate classifier, *even if the training data is correctly labeled*. As a concrete demonstration of this problem, we analyze several such attacks that are highly effective against the Polygraph automatic worm signature generation algorithms. We also illustrate the generality of this problem by describing how these same attacks can be used against the Hamsa [9] polymorphic worm signature generation system, and against Bayesian spam classifiers.

Our contributions are as follows:

- We define the classifier generation problem as a learning problem in an adversarial environment.
- We describe attacks on learning classifier generators that involve careful placement of features in the target-class training data, the innocuous training data, or both, all toward forcing the generation of a classifier that will exhibit many false positives and/or false negatives.
- We analyze and simulate these attacks to demonstrate their efficacy in the polymorphic worm signature generation context. We also implement them, to demonstrate their practicality.

We conclude that the problem of a delusive adversary must be taken into account in the design of classifier generation systems to be used in adversarial settings. Possible solutions include designing learning algorithms that are robust to maliciously generated training data, training using malicious data samples *not* generated by a malicious source, and performing deeper analysis of the malicious training data to determine the semantic significance of the features being included in a classifier, rather than treating samples as opaque "bags of bits."

We proceed in the remainder of this paper as follows. In Section 2, we define the classifier generation problem in detail. We next describe attacks against learning classifier generators in

[1] delusive: Having the attribute of deluding, . . . , tending to delude, deceptive [17].

Sections 3 and 4. We discuss implications of these attacks, both for worm signature generation and for spam filtering, in Section 5. After reviewing related work in Section 6, we conclude in Section 7.

2 Problem Definition: Adversarial Learning

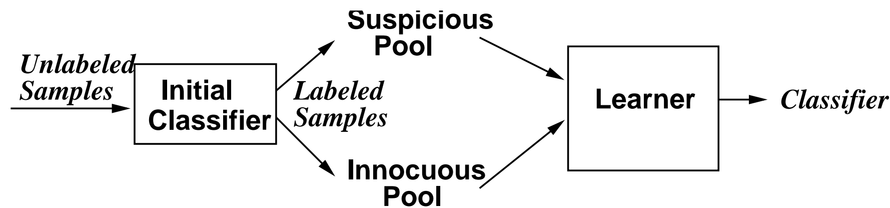

Fig. 1. Schematic of a learner, which uses innocuous and suspicious training pools to generate an accurate classifier.

We now elaborate on the learning model mentioned in the previous section, as followed by Polygraph for worm signature generation, and by Bayesian systems for spam filter generation, with the aim of illuminating strategies an adversary may adopt in an attempt to cause learning to fail. We begin by describing the learning model, and examining the criteria that must be met for learning to succeed. We then consider the assumptions the learning model makes, and why they may not always hold in practice. Finally, we describe general strategies for *forcing* the assumptions the model makes to be violated.

2.1 Learning Model

Identifying worms or spam so that they may be filtered is at its heart a classification problem: we seek a classifier that, given a sample, will label that sample as being of the *target class* (*e.g.,* a worm infection attempt, or a spam email) or as innocuous. One may derive a classifier automatically by *learning* one. Overall, learning involves initially labeling some set of samples to train a *learner*, which, based on their content, generates a classifier. This process is depicted in schematic form in Figure 1.

The raw input to a learning system consists of *unlabeled samples.* In the case of worm signature generation, these are individual network flow payloads observed at a network monitoring point; in the case of Bayesian spam filter generation, they are individual email messages arriving in a user's inbox. Note that an adversary may influence the content of these unlabeled samples to a varying extent; we return to this point later in this section.

The unlabeled samples are first labeled by an initial classifier. Samples labeled as being in the target class are placed in the *suspicious pool*. Samples labeled as *not* being in the target class are placed in the *innocuous pool*. It may seem circular to begin the process of deriving a classifier with a classifier already in hand. It is not. The classifier used to perform the initial labeling of samples typically has some combination of properties that makes it unattractive for general use, such as great computational cost or inaccuracy. We consider this classifier used for the initial labeling of samples below.

Once these samples have been labeled, the learner analyzes the *features* found in the samples in each pool, and produces a classifier. Machine learning allows a very broad definition of what may constitute a feature. In this work we focus on the case where each feature is the presence or absence of a *token*, or contiguous byte string, though our results are generalizable to other types of features.

Feedback Note that throughout this paper, we optimistically assume that the system uses an intelligent feedback loop. For example, if the system collects 10 target-class samples,

generates a classifier, and later collects 10 new target-class samples, it generates an updated classifier using all 20 samples in its suspicious pool, rather than generating a new classifier using only the latest 10. How to achieve this property is application-specific, and outside the scope of this work. This property is crucially important, as otherwise the attacker can prevent the learner from *ever* converging to a correct classifier.

2.2 Successful Learning

To understand how an adversary might thwart learning, we must first understand what constitutes successful learning. Using labeled pools of samples, the learner seeks to generate a classifier that meets several important criteria. First, the classifier should be computationally efficient; it should be able to label samples at their full arrival rate (in the case of worm filtering, at a high link speed). The classifier should also exhibit no false negatives; it should correctly classify all target-class samples as such. It should also exhibit very few or no false positives; it should not classify non-target-class samples as being in the target class.

The learner must be able to generate an accurate classifier using a reasonably small number of labeled target-class samples. An adversary can severely undermine the usefulness of the system by increasing the number of labeled target-class samples necessary to generate an accurate classifier. This is especially true in the case of automatic worm signature generation, where a worm infects ever-more vulnerable hosts while training data is being collected.

2.3 Limitations of Initial Classifier

Let us now return to the initial classifier used to label samples, and the properties that make it inappropriate for general use (and thus motivate the automated derivation of a superior classifier through learning). First, the initial classifier may be too expensive to use on all samples. For example, systems like TaintCheck [16] and the execution monitoring phase of Vigilante [3] identify flows that cause exploits very accurately, but slow execution of a server significantly. In the case of spam, it is most often a user who initially labels inbound emails as spam or non-spam. Clearly, the user is an “expensive” classifier. In both these application domains, the aim is to use the expensive classifier sparingly to train a learner to generate a far less expensive classifier.

In addition, the classifier used to label samples initially is often error-prone; it may suffer from false positives and/or false negatives. For example, classifying all samples that originate from a host whose behavior fits some coarse heuristic (*e.g.,* originating more than a threshold number of connections per unit time) risks flagging innocuous samples as suspicious. A coarse heuristic that errs frequently in the opposite direction (*e.g.,* classifying as suspicious only those samples from source addresses previously seen to port scan) risks flagging suspicious samples as innocuous (*e.g.,* a hit-list worm does not port scan, but is still in the target class).

2.4 Assumptions and Practice

Given that the initial classifier is error-prone, consider the content of the two labeled pools it produces. Ideally, the innocuous pool contains legitimate traffic that exactly reflects the distribution of current traffic. In reality, though, it may not. First, because the classifier used in initial labeling of samples is imperfect, the innocuous pool might well include target-class traffic not properly recognized by that classifier. Moreover, the innocuous pool may contain traffic that is not target-class traffic, but not part of the representative innocuous traffic mix; an adversary may send non-target-class traffic to cause this sort of mislabeling. Finally, the innocuous pool may not reflect *current* traffic; it may be sufficiently old that it does not contain content common in current traffic.

The suspicious pool is essentially a mirror image of the innocuous pool. Ideally, it contains only samples of the target class. But as before, the flawed classifier may misclassify

innocuous traffic as suspicious, resulting in innocuous traffic in the suspicious pool. Additionally, an adversary may choose to send non-target-class traffic in such a way as to cause that traffic (which is innocuous in content) to be classified as suspicious.

Formal proofs of desirable properties of machine learning algorithms (*e.g.,* fast convergence to an accurate classifier with few labeled samples) tend to assume that the features present in samples are determined randomly, or in some applications, that a *helpful teacher* designs the samples' content with the aim of speeding learning. We note that using learning to generate classifiers for worms constitutes learning with a *malicious teacher;* that is, the adversary is free to attempt to construct target-class samples with the aim of thwarting learning, and to attempt to force the mislabelings described above to occur.

2.5 Attack Taxonomy

There are a number of adversarial models to consider. In particular, there are three potential adversary capabilities that we are interested in:

- **Target feature manipulation.** The adversary has some power to manipulate the features in the target-class samples. Some features are *necessary* for the target-class samples to accomplish their purpose (*e.g.*, successfully hijack program execution in a worm sample, or entice the reader to visit a web-site in a spam email). There are a variety of techniques to minimize or obfuscate these necessary features, such as worm polymorphism. A less-studied technique that we investigate is the inclusion of additional, *spurious*, features in the target-class samples, whose sole purpose is to mislead the learner.
- **Suspicious pool poisoning.** The adversary may attempt to fool the initial classifier, such that non-target-class samples are put into the suspicious pool. These samples may be specially constructed to mislead the learner.
- **Innocuous pool poisoning.** The adversary may attempt to place samples into the innocuous pool. These could be target-class samples, or non-target-class samples that nonetheless mislead the learner.

We propose two types of attack that the adversary can perform using one or more of the above techniques:

- **Red herring attacks.** The adversary incorporates spurious features into the target-class samples to cause the learner to generate a classifier that depends on those spurious features instead of or in addition to the necessary target-class features. The adversary can evade the resulting classifier by not including the spurious features in subsequently generated target-class samples.
- **Inseparability attacks.** The adversary incorporates features found in the innocuous pool into the target-class samples in such a way as to make it impossible for the learner to generate a classifier that incurs both few false positives and few false negatives.

In this work we demonstrate highly effective attacks of both types that assume only a *delusive* adversary—one who provides the learner with correctly labeled training data, but who manipulates the features in the target-class samples to mislead the learner. We further demonstrate how an adversary with the ability to poison the suspicious pool, the innocuous pool, or both, can more easily perform inseparability attacks.

Having sketched these strategies broadly, we now turn to describing the attacks based on them in detail.

3 Attacks on Conjunction Learners

One way of generating a classifier is to identify a set of features that appears in every sample of the target class. The classifier then classifies a sample as positive if and only if it contains every such feature.

We construct two types of red herring attacks against learners of this type. We use the Polygraph conjunction learner as a concrete example for analysis [15]. In the Polygraph conjunction learner, the signature is the set of features that occur in every sample in the malicious training pool.[2] In Section 5 we discuss the effectiveness of these attacks against Hamsa [9], a recently proposed Polygraph-like system. We show that the attacks described here are highly effective, even under the optimistic assumption that the malicious training pool contains only target-class samples.

In Section 3.3, we show that even in a highly optimistic scenario, a polymorphic worm that Polygraph could stop after only .5% of vulnerable hosts are infected can use these attacks to improve its infection ratio to 33% of vulnerable hosts.

3.1 Attack I: Randomized Red Herring Attack

Attack Description The learner's goal is to generate a signature consisting only of features found in every target-class sample. In the *Randomized Red Herring* attack, the attacker includes unnecessary, or *spurious*, features in some target-class samples, with the goal of tricking the learner into using those features in its signature. As a result, target-class samples that do *not* include the set of spurious features that are in the signature are able to evade the signature.

The attacker first chooses a set of α spurious features. The attacker constructs the target-class samples such that each one contains a particular spurious feature with probability p. As a result, the target-class samples in the learner's malicious pool will all have some subset of the α spurious features in common, and those spurious features will appear in the signature. The signature will then have false negatives, because many target-class samples will not have *all* of those features.

Analysis We first find how selection of α and p affect the expected false negative rate.

Theorem 1 *The expected false negative rate $F[s]$ for a signature generated from s target-class samples, where each target-class sample has probability p of including each of α spurious features, is $F[s] = 1 - p^{\alpha p^s}$.*

Derivation The expected number of spurious features that will be included in a signature after collecting s samples is $\sigma = \alpha p^s$. The chance of all σ of those spurious features being present in any given target-class samples is p^σ. Hence, the expected false negative rate of the signature is $y = 1 - p^\sigma$, which we rewrite as $y = 1 - p^{\alpha p^s}$.

The attacker has two parameters to choose: the number of spurious features α, and the probability of a spurious feature occurring in a target-class sample p. The attacker will use as high an α as is practical, often limited only by the number of additional bytes that the attacker is willing to append.

The ideal value of p is not clear by inspection. A higher p results in more spurious features incorporated into the signature, but it also means that the spurious features that do get included in the classifier are more likely to occur in other target-class samples. We find the best value of p by finding the roots of the derivative: $\frac{dy}{dp} = -\alpha p^{\alpha p^s + s - 1}(s\ln(p) + 1)$. There are two roots. $p = 0$ minimizes false negatives (it is equivalent to not performing the attack at all), and $p = e^{-\frac{1}{s}}$ maximizes false negatives.

Theorem 2 *The value of p that maximizes the false negative rate in the Randomized Red Herring attack is: $p = e^{-\frac{1}{s}}$.*

[2] In Section 5, we show that the hierarchical clustering algorithm used by Polygraph to tolerate noise does not protect against these attacks.

The p that generates the highest false negative rate depends on the number of target-class samples seen by the learner, s. Hence, the optimal value of p depends on the exact goals of the attacker. For a worm author, one way to choose a value of p would be to set a goal for the number of machines to compromise before there is an effective classifier, and calculate the number of positive samples that the learner is likely to have gathered by that time, based on the propagation model of his worm and the deployment of the learner, and then set p to a value that ensures there are still a large number of false negatives at that time.

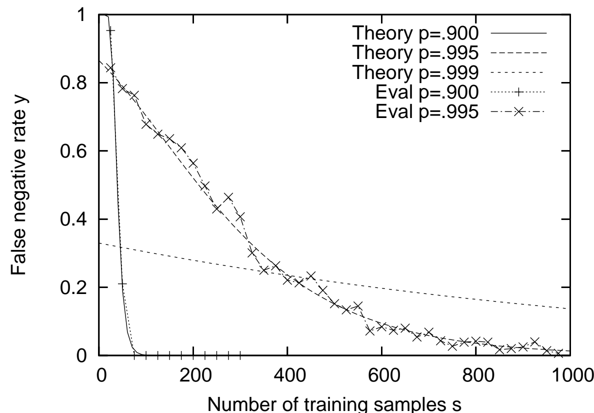

Fig. 2. Randomized Red Herring attack. $\alpha = 400$

We implemented a version of the Randomized Red Herring attack based on this model. We took a real buffer-overflow exploit against the ATPhttpd web server [19], filled the attack-code with random bytes to simulate polymorphic encryption and obfuscation, and replaced the 800 bytes of padding with 400 unique two-byte spurious features. Specifically, we set each two-byte token to the binary representation of its offset with probability p, and to a random value with probability $1 - p$. Note that the number of spurious features used here is *conservative*. In this attack, the 800 padding bytes were already used, because they were necessary to overflow the buffer. The attacker could easily include *more* bytes to use as spurious features. For example, he could include additional HTTP headers for the sole purpose of filling them with spurious features.

Figure 2 shows the predicted and actual false negative rates as the number of training samples increases, for several values of p. We used values that maximized the false negative rate when $s = 10$ ($p = .900$), when $s = 200$ ($p = .995$), and when $s = 500$ ($p = .999$). For each data point, we generate s worm samples, and use the Polygraph conjunction learner to generate a classifier. We then generate another 1000 worm samples to measure the false negative rate. There are two things to see in this graph. First, our experimental results confirm our probability calculations. Second, the attack is quite devastating. Low values of p result in very high initial false negatives, while high values of p prevent a low-false-negative signature from being generated until many worm samples have been collected.

3.2 Attack II: Dropped Red Herring Attack

Attack description In the Dropped Red Herring attack, the attacker again chooses a set of α spurious features. Initially, he includes all α features in every target-class sample. As a result, the target-class samples in the learner's malicious training pool will all have all α spurious features, and all α spurious features will be included in the signature.

Once the signature is in place, all the attacker needs to do to evade the signature is to stop including *one* of the spurious features in subsequent target-class samples. The signature

will have a 100% false negative rate until the learner sees a target-class sample missing the spurious feature, and deploys an updated signature that no longer requires that feature to be present. At that point, the attacker stops including another spurious feature. The cycle continues until the attacker has stopped including all of the spurious features.

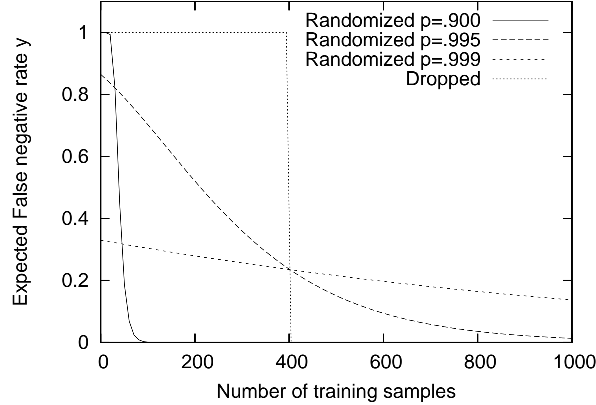

Fig. 3. Dropped Red Herring compared to Randomized Red Herring, $\alpha = 400$

Attack analysis For sake of comparison to the Randomized Red Herring attack, assume that the attacker stops including a single spurious feature the instant that an updated signature is deployed. Also assume that the learner deploys a new signature each time it collects a new worm sample, since each successive sample will have one fewer spurious feature than the last. In that case, the classifier will have 100% false negatives until α positive samples have been collected.

Theorem 3 *The false negative rate $F[s]$ for the signature generated after s target-class samples have been collected is*

$$F[s] = \begin{cases} 100\% & if s < \alpha \\ 0\% & if s \geq \alpha \end{cases}$$

With these assumptions, the Dropped Red Herring attack is compared to the Randomized Red Herring attack in Figure 3. When the attack is executed in this way, and there are a moderate number of spurious features, the attack can be quite devastating. The generated signatures are useless until all α features have been eliminated from the signature.

While the Dropped Red Herring attack is far more effective than the Randomized Red Herring attack (until the learner has dropped all α spurious features from the signature), the Randomized Red Herring attack has one important advantage: it is simpler to implement. The Dropped Red Herring attack must interact with the signature learning system, in that it must discover when a signature that matches the current target-class samples has been published, so that it can drop another feature, and remain unfiltered. There is no such requirement of the Randomized Red Herring attack. This is not to say that the Dropped Red Herring attack is impractical; the attacker has significant room for error. While dropping a feature prematurely will ‘waste’ a spurious feature, there is little or no penalty for dropping a feature some time after an updated signature has been deployed.

3.3 Attack Effectiveness

We show that even with an optimistic model of a distributed signature generation system, and a pessimistic model of a worm, employing the Randomized Red Herring or Dropped Red

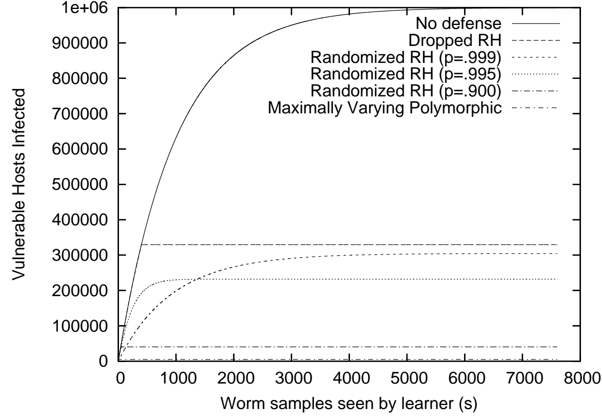

Fig. 4. Worm propagation. L=1000, V=1000000, $\alpha = 400$

Herring attack delays the learner enough to infect a large fraction of vulnerable hosts before an accurate signature can be generated.

We assume that the learner is monitoring L addresses. Each time the worm scans one of these addresses, the learner correctly identifies it as a worm, and instantaneously updates and distributes the signature. At that point, any scan of any vulnerable host has probability $F[s]$ of succeeding (the false negative rate of the current signature). There are several optimistic assumptions for the learner here, most notably that updated signatures are distributed *instantaneously*. In reality, distributing even a single signature to all hosts in less than the time it takes to infect all vulnerable hosts is a challenge [22].[3]

We assume that the worm scans addresses uniformly at random. In reality, there are several potential strategies a worm author might use to minimize the number of samples seen by the learner. An ideally coordinated worm may scan every address exactly once, thus minimizing the number of samples sent to any one of the learner's addresses, and eliminating 'wasted' scans to already-infected hosts. The worm could further improve this approach by attempting to order the addresses by their likelihood of being monitored by the learner, scanning the least likely first.

We model the worm by estimating the number of additional vulnerable hosts infected in-between the learner receiving new worm samples. Note that because we assume signature updates are instantaneous, the scan rate of the worm is irrelevant. Intuitively, both the rate of infection and the rate of the learner receiving new samples are proportional to the scan rate, thus canceling each other out.

Theorem 4 *For a worm scanning uniformly at random, where there are V vulnerable hosts, L addresses monitored by the learner, and N total hosts, the expected number of infected hosts I after s worm samples have been seen by the learner is:*

$$I[s] = I[s-1] + (V - I[s-1])\left(1 - \left(1 - \frac{F[s-1]}{N}\right)^{(N/L)}\right)$$

Derivation The expected number of worm scans in-between the learner receiving a new worm sample is $\frac{1}{P(\text{scan is seen by learner})} = \frac{N}{L}$.

[3] The Dropped Red Herring attack in particular is much more devastating when taking the signature generation and distribution time into account, since the next spurious feature is not revealed before an updated signature is distributed. Hence, a worm using α spurious features is allowed to propagate freely for at least α times the time needed to generate and distribute a signature.

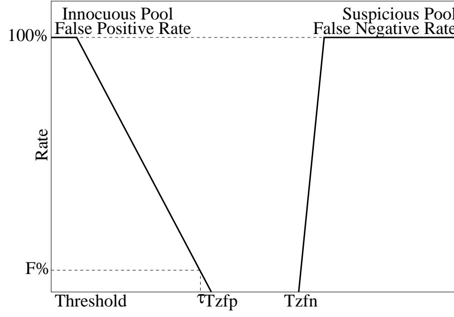

Fig. 5. Training data distribution graph, used to set τ. τ *could* be set to perfectly classify training data.

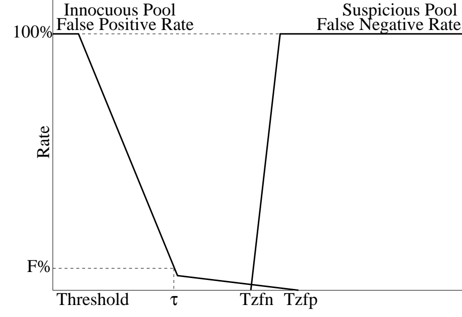

Fig. 6. Overlapping training data distribution graph. No value of τ perfectly classifies training data.

$$I[s] = I[s-1] + (\text{\# vulnerable uninfected hosts})P(\text{host becomes infected})$$
$$I[s] = I[s-1] + (V - I[s-1])(1 - P(\text{host does not become infected}))$$
$$I[s] = I[s-1] + (V - I[s-1])(1 - P(\text{scan does not infect host})^{(\text{\# scans})})$$
$$I[s] = I[s-1] + (V - I[s-1])(1 - (1 - P(\text{scan infects host}))^{(\text{\# scans})})$$
$$I[s] = I[s-1] + (V - I[s-1])(1 - (1 - P(\text{scan contacts host})P(\text{scan not blocked}))^{(\text{\# scans})})$$
$$I[s] = I[s-1] + (V - I[s-1])(1 - (1 - \frac{1}{N}F[s-1])^{(N/L)})$$

In Figure 4, we model the case of $V =$ one million vulnerable hosts, $L =$ one thousand learner-monitored addresses, and $N = 2^{32}$ total addresses. In the case where the worm is maximally-varying polymorphic, we assume that the learner needs five samples to generate a correct signature. In that case, only 4,990 (.0499%) vulnerable hosts are infected before the correct signature is generated, stopping further spread of the worm. By employing the Dropped Red Herring attack, the worm author increases this to 330,000 (33.0%). The Randomized Red Herring attack is only slightly less effective, allowing the worm to infect 305,000 (30.5%) vulnerable hosts using $p = .999$.

Given that the Dropped Red Herring and Randomized Red Herring attacks allow a worm to infect a large fraction of vulnerable hosts even under this optimistic model, it appears that the Conjunction Learner is not a suitable signature generation algorithm for a distributed worm signature generation system.

4 Attacks on Bayes Learners

Bayes learners are another type of learner used in several adversarial learning applications, including worm signature generation and spam filtering. We present several practical attacks against Bayes learners, which can prevent the learner from *ever* generating an accurate signature, regardless of how many target-class samples it collects. As a concrete example, we use Polygraph's implementation of a *Naive* Bayes learner. That is, a Bayes learner that assumes independence between features. Non-Naive Bayes learners are not as commonly used, due partly to the much larger amount of training data that they require. We believe that the attacks described here can also be applied to other Bayes learners, possibly even non-naive ones that do not assume independence between features.

4.1 Background on Bayes Learners

In the following discussion, we use the notation $P(x|+)$ to mean the probability that the feature or set of features x occurs in malicious samples, and $P(x|-)$ to denote the probability that it occurs in innocuous samples. This learner can be summarized as follows:

- The learner identifies a set of tokens, σ, to use as features. σ is the set of tokens that occur more frequently in the suspicious pool than in the innocuous pool. That is, $\forall \sigma_i \in \sigma, P(\sigma_i|+) > P(\sigma_i|-)$. This means that the presence of some σ_i in a sample to be classified can never *lower* the calculated probability that it is a worm.
- Classifies a sample as positive (*i.e.*, in the target class) whenever $\frac{P(\gamma|+)}{P(\gamma|-)} > \tau$ where τ is a threshold set by the learner, and γ is the subset of σ that occurs in the particular sample. We refer to $\frac{P(\gamma|+)}{P(\gamma|-)}$ as the *Bayes score*, denoted score(γ)
- We assume conditional independence between features. Hence, $\frac{P(\gamma|+)}{P(\gamma|-)} = \prod \frac{P(\gamma_i|+)}{P(\gamma_i|-)}$
- $P(\sigma_i|-)$ is estimated as the fraction of samples in the innocuous pool containing σ_i.
- $P(\sigma_i|+)$ is estimated as the fraction of samples in the suspicious pool containing σ_i.
- τ is chosen as the value that achieves a false positive rate of no more than F% in the innocuous pool.

Setting the τ Threshold The attacks we describe in this section all involve making it difficult or impossible to choose a good matching threshold, τ. For clarity, we describe the method for choosing τ in more detail.

After the learner has chosen the feature set σ and calculated $\frac{P(\sigma_i|+)}{P(\sigma_i|-)}$ for each feature, it calculates the Bayes score $\frac{P(\sigma|+)}{P(\sigma|-)}$ for each sample in the innocuous pool and suspicious pool, allowing it to create the *training data distribution graph* in Figure 5. The training data distribution graph shows, for every possible threshold, what the corresponding false positive and false negative rates would be in the innocuous and suspicious training pools. Naturally, as the threshold increases, the false positive rate monotonically decreases, and the false negative rate monotonically increases. Note that Figure 5 and other training data distribution graphs shown here are drawn for illustrative purposes, and do not represent actual data.

There are several potential methods for choosing a threshold τ based on the training data distribution graph. The method described in Polygraph [15] is to choose the value that achieves no more than F% false positives in the innocuous pool. One alternative considered was to set τ to T_{zfp}, the lowest value that achieves zero false positives in the innocuous pool. However, in the examples studied, a few outliers in the innocuous pool made it impossible to have zero false positives without misclassifying all of the actual worm samples, as in Figure 6. Of course, a highly false-positive-averse user could set F to 0, and accept the risk of false negatives.

Another tempting method for choosing τ is to set it to T_{zfn}, the highest value that achieves zero false negatives in the suspicious pool. However, we show in Section 4.2 that this would make the Bayes learner vulnerable to a red herring attack.

4.2 Dropped Red Herring and Randomized Red Herring Attacks are Ineffective

Dropped Red Herring Attack The method just described for choosing τ may seem unintuitive at first. However, it was carefully designed to prevent Dropped Red Herring attacks, as illustrated in Figure 7. Suppose that τ was set to T_{zfn}, the threshold just low enough to achieve zero false negatives in the training data. This may seem more intuitive, since it reduces the risk of false positives as much as possible while still detecting all positive samples in the malicious pool.

Now suppose that the attacker performs the Dropped Red Herring attack. Since the spurious features occur in 100% of the target-class samples, they will be used in the feature set σ. Since each target-class sample in the malicious pool now has more incriminating features, the Bayes score of every target-class sample in the suspicious pool increases, causing the false negative curve to be artificially shifted to the right.[4]

[4] The false positive curve may also shift towards the right. We address this in Section 4.3.

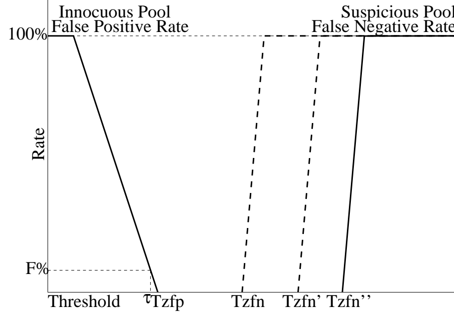

Fig. 7. Dropped Red Herring Attack. Spurious tokens artificially shift false negative curve to the right. It shifts back to the left when worm samples without the spurious tokens are added to the suspicious pool.

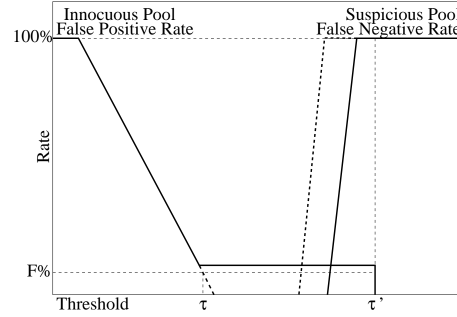

Fig. 8. Correlated Outlier attack

If the learner were to set τ to T''_{zfn} (see Figure 7), then the attacker could successfully perform the Dropped Red Herring attack. When a target-class sample includes one less spurious feature, its Bayes score becomes less than T'_{zfn}, where $T'_{zfn} < T''_{zfn}$. Hence it would be classified as negative. Eventually the learner would get target-class samples without that spurious feature in its malicious pool, causing the false negative curve to shift to the left, and the learner could update the classifier with a threshold of T'_{zfn}. At that point the attacker could stop including another feature.

However, setting τ to the value that achieves no more than F% false positives is robust to the Dropped Red Herring attack. Assuming that the spurious features do not appear in the innocuous pool, the false positive curve of the training data distribution graph is unaffected, and hence the threshold τ is unaffected.

Randomized Red Herring Attack The Randomized Red Herring attack has little effect on the Bayes learner. The Bayes score for any given target-class sample will be *higher* due to the inclusion of the spurious features. The increase will vary from sample to sample, depending on which spurious features that sample includes. However, again assuming that the spurious features do not appear in the innocuous pool, this has no effect on τ. Hence, the only potential effect of this attack is to *decrease* false negatives.

4.3 Attack I: Correlated Outlier Attack

Unfortunately, we have found an attack that *does* work against the Bayes learner. The attacker's goal in this attack is to increase the Bayes scores of samples in the innocuous pool, so as to cause significant overlap between the training data false positive and false negative curves. In doing so, the attacker forces the learner to choose between significant false positives, or 100% false negatives, independently of the exact method chosen for setting the threshold τ.

Attack Description The attacker can increase the Bayes score of innocuous samples by using spurious features in the target-class samples, which also appear in some innocuous samples. By including only a fraction β of the α spurious features, S, in any one target-class sample, innocuous samples that have all α spurious features can be made to have a higher Bayes score than the target-class samples.

The result of the attack is illustrated in Figure 8. The spurious features in the target-class samples cause the false negative curve to shift to the right. The innocuous samples that contain the spurious features result in a tail on the false positive curve. The tail's height corresponds to the fraction of samples in the innocuous pool that have the spurious tokens. As the figure shows, regardless of how τ is chosen, the learner is forced either to classify

innocuous samples containing the spurious features as false positives, or to have 100% false negatives.

The challenge for the attacker is to choose spurious features that occur in the innocuous training pool (which the attacker cannot see) in the correct proportion for the attack to work. The attacker needs to choose spurious features that occur *infrequently* enough in the innocuous pool that the corresponding Bayes score $\frac{P(S|+)}{P(S|-)}$ is large, but *frequently* enough that a significant fraction of the samples in the innocuous pool contain all of the spurious features; *i.e.* so that the forced false positive rate is significant.

Attack Analysis We show that the attack works for a significant range of parameters. The attacker's *a priori* knowledge of the particular network protocol is likely to allow him to choose appropriate spurious features. A simple strategy is to identify a type of request in the protocol that occurs in a small but significant fraction of requests (*e.g.* 5%), and that contains a few features that are not commonly found in other requests. These features are then used as the spurious features.

We first determine what parameters will give the innocuous samples containing the spurious features a higher Bayes score than the target-class samples. For simplicity, we assume that $P(s_i|-)$ is the same for each spurious feature s_i.

Theorem 5 *Given that each target-class sample contains the feature set W and $\beta\alpha$ spurious features s_i chosen uniformly at random from the set of α spurious features S, samples containing all α spurious features in S have a higher Bayes score than the target-class samples when:*

$$P(s_i|-) < \beta \text{ and } \left(\frac{\beta}{P(s_i|-)}\right)^{\beta\alpha-\alpha} \leq P(W|-)$$

The condition $P(s_i|-) < \beta$ is necessary to ensure that $P(s_i|-) < P(s_i|+)$. Otherwise, the learner will not use the spurious features in the Bayes classifier.

The second condition is derived as follows:

$$\frac{P(S|+)}{P(S|-)} \geq \frac{P(\beta S,W|+)}{P(\beta S,W|-)} \quad \text{Innocuous samples have a higher Bayes score}$$

$$\frac{P(s_i|+)^{\alpha}}{P(s_i|-)^{\alpha}} \geq \frac{P(s_i|+)^{\beta\alpha}P(W|+)}{P(s_i|-)^{\beta\alpha}P(W|-)} \quad \text{Independence assumption}$$

$$\frac{\beta^{\alpha}}{P(s_i|-)^{\alpha}} \geq \frac{\beta^{\beta\alpha}(1)}{P(s_i|-)^{\beta\alpha}P(W|-)} \quad \text{Substitution}$$

$$\left(\frac{\beta}{P(s_i|-)}\right)^{\beta\alpha-\alpha} \leq P(W|-) \quad \text{Rearrangement}$$

Note that while we have assumed independence between features here, the attack could still apply to non-Naive Bayes learners, provided that $\frac{P(S|+)}{P(S|-)} \geq \frac{P(\beta S,W|+)}{P(\beta S,W|-)}$ is satisfied. Whether and how it can be satisfied will depend on the specific implementation of the learner.

When these conditions are satisfied, the classifier must either classify innocuous samples containing the spurious features S as positive, or suffer 100% false negatives. Either way can be considered a 'win' for the attacker. Few sites will be willing to tolerate a significant false positive rate, and hence will choose 100% false negatives. If sites *are* willing to tolerate the false positive rate, then the attacker has succeeded in performing a denial-of-service attack. Interestingly, the attacker could choose his spurious tokens in such a way as to perform a very targeted denial-of-service attack, causing innocuous samples of a particular type to be filtered by the classifier.

For the threshold-choosing algorithm used by Polygraph, τ will be set to achieve 100% false negatives if $\frac{P(S|-)}{P(S|+)} \geq F$. Otherwise it will be set to falsely classify the samples containing the spurious features S as positive.

Evaluation The Correlated Outlier is practical for an adversary to implement, even though he must make an educated guess to choose the set of spurious features that occur with a suitable frequency in the innocuous pool.

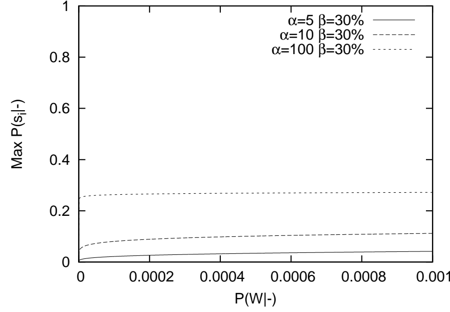

Fig. 9. Correlated Outlier attack evaluation

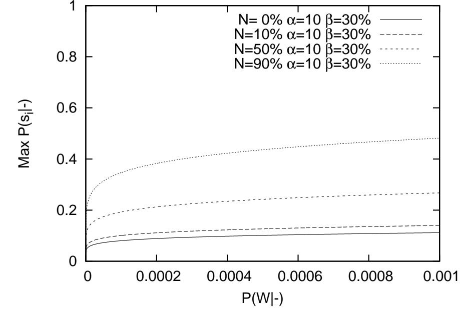

Fig. 10. Correlated Outlier attack evaluation, with chaff

There are four parameters in Theorem 5 that determine whether the attack is successful. The attacker chooses α, how many spurious features to use, and β, the fraction of those spurious features to include in each target-class sample. The likelihood of success increases with greater α. However, since he must find α spurious features that are highly correlated in the innocuous pool, relatively low values are the most practical.

The third parameter, the frequency of the target-class features in the innocuous pool $P(W|-)$ is out of the attacker's hands. High values of $P(W|-)$ make the attack easiest. Indeed, if $P(W|-)$ is high, the learner is already forced to choose between false negatives, and significant false positives. We show the attack is still practical for low values of $P(W|-)$.

The fourth parameter, the frequency of the spurious features in the innocuous pool $P(s_i|-)$, is not directly controlled by the attacker. The attacker's challenge is to choose the spurious features such that $P(s_i|-)$ is low enough that the attacker succeeds in getting the innocuous features with all α of the spurious features S to have a higher Bayes score than the target-class samples.

Figure 9 shows that the attack can succeed for a wide range of realistic parameters. Each curve in the graph represents a different attacker choice of α. As $P(W|-)$ increases, the maximum value of $P(s_i|-)$ also increases. Even for very low values of $P(W|-)$ and α, the attacker has a great deal of room for error in his estimation of $P(s_i|-)$.

Again, any value that satisfies these constraints will force the the learner to choose between false negatives and false positives, and the classifier will not improve as more target-class samples are obtained. If the learner uses the Polygraph threshold-setting algorithm, then τ will be set to achieve 100% false negatives if $\frac{P(S|-)}{P(S|+)} \geq F$. Otherwise it will be set to have low false negatives, but will classify the samples containing the spurious features S as positive. The signature will not improve, and as long as it is in use, legitimate samples containing those samples will be false positives, causing a targeted denial of service.

4.4 Attack II: Suspicious Pool Poisoning

Up to this point we have assumed that the suspicious and innocuous pools are noise-free. That is, the suspicious pool contains only target-class samples, and the innocuous pool contains only innocuous samples. In some cases, however, the attacker may be able to inject constructed samples into the suspicious pool, the innocuous pool, or both, as described in Section 2. We first consider the case where the attacker is able to inject *chaff*, specially constructed samples, into the suspicious pool.

Attack Description The chaff can simultaneously have two effects. First, by not including the actual target-class features W, the classifier will calculate a lower $P(W|+)$. The actual target-class samples in the suspicious pool will have lower Bayes scores as a result, stretching the false negative curve of the training data distribution graph to the left.

Second, the classifier will calculate a higher $P(s_i|+)$ for any spurious feature s_i included in the chaff. This will cause innocuous samples containing those features to have a higher Bayes score, stretching the false positive curve of the training data distribution graph to the right, in the same manner as in the Correlated Outlier attack (Figure 8). *Unlike* the target-class samples, each chaff sample can contain *all* of the spurious features, since it makes no difference to the attacker whether the chaff samples are classified as positive by the resulting Bayes classifier.

Attack Analysis The attacker's goal is again to force the learner to choose between false positives and false negatives, by ensuring that the score of a sample containing all α of the spurious features S has a higher Bayes score than a sample containing the true target-class features W, and a fraction β of the spurious features. Assuming that the chaff in the suspicious pool contains all α of the spurious features, the attacker can include fewer spurious features in the actual target-class samples, or even none at all.

Theorem 6 *Suppose that the fraction N of samples in the suspicious pool is chaff containing the spurious features S. Samples containing all α spurious features have a higher Bayes score than samples containing the actual target-class features W and the fraction β of the α spurious features when:*

$$P(s_i|-) < N + (1-N)\beta \text{ and } (1-N)\left(\frac{N+\beta(1-N)}{P(s_i|-)}\right)^{\beta\alpha-\alpha} \leq P(W|-)$$

When these conditions are satisfied, this attack becomes equivalent to the Correlated Outlier attack. Notice that when there is no chaff ($N = 0$) these conditions simplify to the conditions presented in Section 4.3.

Evaluation We perform a similar evaluation as in Section 4.3. In this case, the attacker uses a relatively low number of spurious features ($\alpha = 10$), and each curve of the graph represents different ratios of chaff in the suspicious pool. Figure 10 shows that the addition of chaff to the suspicious pool greatly improves the practicality of the attack. The resulting classifier will again either have 100% false negatives, or cause legitimate samples with the spurious features to be blocked.

4.5 Attack III: Innocuous Pool Poisoning

We next consider the case where the attacker is able to poison the *innocuous* training pool. The most obvious attack is to attempt to get samples with the target-class features W into the innocuous pool. If the target-class samples include only the features W (no spurious features), then it would be impossible to generate a classifier that classified the target-class samples as positive without also classifying the samples that the attacker injected into the innocuous pool as positive. Hence, the learner could be fooled into believing that a low-false-positive classifier cannot be generated.

The solution to this problem proposed by Polygraph [15] for automatic worm signature generation is to use a network trace taken some time t ago, such that t is greater than the expected time in-between the attacker discovering the vulnerability (and hence discovering what the worm features W will be), and the vulnerability being patched on most vulnerable machines. The time period t is somewhat predictable assuming that the attacker does not discover the vulnerability before the makers of the vulnerable software do. Conversely, t could be an arbitrary time period for a "zero-day" exploit. However, we show that a patient attacker can poison the innocuous pool in a useful way *before* he knows what the worm features W are.

Attack Description The attacker can aid the Correlated Outlier attack by injecting *spurious* tokens into the innocuous pool. In this case, using an old trace for the innocuous pool does not help at all, since the attacker does not need to know W at the time of poisoning the innocuous pool. That is, an attacker who does not yet have a vulnerability to exploit can choose a set of spurious features S, and preemptively attempt to get samples containing S into the learner's

innocuous pool, thus increasing $P(S|-)$. The attacker can then use these spurious features to perform the Correlated Outlier attack, optionally poisoning the suspicious pool as well as described in Section 4.4.

Attack Analysis If the attacker is able to inject samples containing S into the innocuous pool, $P(S|-)$ will be increased. The attacker's best strategy may be to use spurious features that do not occur at all in normal traffic. This would allow him to more accurately estimate the learner's $P(S|-)$ when designing the worm.

Aside from this additional knowledge, the attack proceeds exactly as in Section 4.4.

Evaluation The success of the attack is determined by the same model as in Theorem 6. The addition of the injected spurious features helps make the attack more practical by allowing him to more accurately predict a set of spurious features that occur together in a small fraction of the innocuous training pool. Success in the attack will again either result in the classifier having 100% false negatives, or result in innocuous samples containing the spurious features to be blocked.

5 Discussion

5.1 Hierarchical Clustering

Polygraph [15] implements a hierarchical clustering algorithm to enable its conjunction and subsequence learners to work in the presence of non-worm samples in the suspicious training pool. Each sample starts as its own cluster, and clusters are greedily merged together. Each cluster has a signature associated with it that is the intersection of the features present in the samples in that cluster. The greedy merging process favors clusters that produce low-false-positive signatures; *i.e.*, those that have the most distinguishing set of features in common. When no more merges can be performed without the resulting cluster having a high-false-positive signature, the algorithm terminates and outputs a signature for each sufficiently large cluster. Ideally, samples of unrelated worms are each in their own cluster, and non-worm samples are not clustered.

One might wonder whether the hierarchical clustering algorithm helps to alleviate the Randomized Red Herring or Dropped Red Herring attacks. It does not.

First consider the Randomized Red Herring attack. Each worm sample has the set of features that must be present, W, and some subset of a set of spurious features, S. Keep in mind that the attacker's goal is for the resulting signature to be too *specific*. If the hierarchical clustering algorithm puts all the worm samples into one cluster, which is likely, the resulting signature will be exactly the same as if no clustering were used. If it does not, the resulting signature can only be *more* specific, which further increases false negatives.

For example, suppose one cluster contains spurious features s_1, s_2, and s_3, and another cluster contains spurious features s_2, s_3, and s_4. Both clusters contain the necessary worm features W. If these clusters are merged together, the resulting signature is the conjunction
$(W \wedge s_2 \wedge s_3)$
If the clusters are not merged, then the learner will publish two signatures. Assuming both signatures are used, this is equivalent to the single signature
$(W \wedge s_1 \wedge s_2 \wedge s_3) \vee (W \wedge s_2 \wedge s_3 \wedge s_4)$
This can be rewritten as:
$(W \wedge s_2 \wedge s_3) \wedge (s_1 \vee s_4)$
Obviously, this is more specific than the signature that would have resulted if the two clusters were merged, and hence will have strictly more false negatives.

The same is true for the Dropped Red Herring attack, by similar reasoning. Again, if all samples of the worm are merged into one cluster, the result is equivalent to if no clustering were used. *Not* merging the samples into a single cluster can only make the signature more *specific*, which further increases false negatives.

5.2 Attack Application to Other Polymorphic Worm Signature Generation Systems

At this time, the only automatic polymorphic worm signature generation systems that are based on learning are Polygraph [15] and Hamsa [9]. Throughout this paper, we have used Polygraph's algorithms as concrete examples. Hamsa generates conjunction signatures, with improved performance and noise-tolerance over Polygraph. To generate a conjunction signature, Hamsa iteratively adds features found in the suspicious pool, preferring features that occur in the *most* samples in the suspicious pool and result in sufficiently low false positives in the innocuous pool.

We begin with two observations. First, the adversary can cause Hamsa to use spurious features in its signature, as long as those features occur sufficiently infrequently in the innocuous pool, and occur at least as often in the suspicious pool as the true target-class features. Second, the false-negative bounds proven in the Hamsa paper only apply to the target-class samples actually found in the suspicious pool, and not necessarily to subsequently generated samples.

Unlike Polygraph, Hamsa stops adding features to the signature once the signature causes fewer false positives in the innocuous pool than some predetermined threshold. As a result, Hamsa is relatively resilient to the Randomized Red Herring attack. For example, using $\alpha = 400$, $p = .995$, Hamsa exhibits only 5% false negatives after collecting 100 target-class samples. While this incidence is still non-trivial, it is an improvement over Polygraph's corresponding 70% false negatives with these parameters.

Hamsa is also less vulnerable to the Dropped Red Herring attack, but unfortunately not completely invulnerable. First, let us assume that Hamsa's method of breaking ties when selecting features is not predictable by the adversary (the method does not appear to be defined in [9]). In this case, the simplest form of the attack will not succeed, as the adversary cannot predict which spurious features are actually used, and hence which to drop to avoid the generated classifier. However, suppose that the attacker is able to inject noise into the suspicious pool, and the spurious features follow some ordering of probabilities with which they appear in a particular noise sample. This ordering then specifies the (probable) preferential use of each spurious feature in the generated signature. That is, the most probable spurious feature will be chosen first by Hamsa, since it will have the highest coverage in the suspicious pool, and so on. In that case, an adversary who can inject n noise samples into the suspicious pool can force up to n iterations of the learning process.

5.3 Attack Application to Spam

The correlated outlier attack described in Section 4.3 is also applicable to Bayesian spam filters, though the specific analysis is dependent on the exact implementation. There is already an attack seen in the wild where a spam email includes a collection of semi-random words or phrases to deflate the calculated probability that the email is spam [6].[5] To perform the correlated outlier attack on a spam filter, the adversary would use as spurious features words that tend to occur together in a fraction of non-spam emails. If a classifier is trained to recognize such an email as spam, it may suffer false positives when legitimate email containing those words is received. Conversely, if a classifier's threshold is biased toward not marking those legitimate mails as spam, it may suffer from false negatives when receiving spam with the chosen features.

As in the worm case, it may be possible for a spam author to guess what words occur in the correct frequency in the innocuous training data. It seems likely that such an attack could succeed were it tailored to an individual user, though it would not be a financial win for the spam author. However, the spam author might be able to tailor the spurious features to a broader audience, for example by selecting jargon words that are likely to occur together in the legitimate mail of a particular profession. Another tactic would be to use words that

[5] Note that the Polygraph implementation of a Bayes classifier is not vulnerable to this attack, because it discards features that have a higher probability of occurring in negative samples than positive samples.

occur in a certain kind of email that occurs at the needed low-but-significant frequency. For example, adding words or phrases in spam emails that one would expect to see in a job offer letter could result in very high-cost false positives, or in the savvy user being hesitant to mark such messages as spam for that very reason.

5.4 Recommendation for Automatic Worm Signature Generation

Current pattern extraction insufficient Most currently proposed systems for automatically generating worm signatures work by examining multiple samples of a worm and extracting the common byte patterns. This is an attractive approach because monitoring points can be deployed with relative ease at network gateways and other aggregation points.

Unfortunately, most previous approaches [7, 8, 21, 24] do not handle the case where the worm varies its payload by encrypting its code and using a small, randomly obfuscated decryption routine. In this paper, we have shown that the only proposed systems that handle this case of polymorphism [9, 15] can be defeated by a worm that simply includes spurious features in its infection attempts.

We believe that if there is to be any hope of generating signatures automatically by only examining the byte sequences in infection attempt payloads, a more formal approach will be needed. Interestingly, while there has been some research in the area of spam email classification in the scenario where an adversary *reacts* to the current classifier in order to evade it [6, 13], there has been little research in the machine learning scenario where an adversary constructs positive samples in such a way as to prevent an accurate classifier from being generated in the first place. One approach that bears further investigation is Winnow [11, 12], a machine learning algorithm with proven bounds on the number of mistakes made before generating an accurate classifier.

Automatic Semantic Analysis Recent research proposes automated *semantic* analysis of collected worm samples, by monitoring the execution of a vulnerable server as it becomes compromised [2, 3, 5]. These approaches can identify which features of the worm request *caused* it to exploit the monitored software, and are hence likely to be invariant, and useful in a signature. This approach is also less susceptible to being fooled by the worm into using spurious features in a signature, since it will ignore features that have no effect on whether the vulnerable software actually gets exploited. The features so identified can also be more expressive than the simple presence or absence of tokens; *e.g.,* they may specify the minimum length of a protocol field necessary to trigger a buffer overflow.

While monitoring points employing semantic analysis are not as easily deployed as those that do not, since they must run the vulnerable software, they are more likely to produce signatures with low false positives and false negatives than those produced by pattern extraction alone.

Given the state of current research, we believe that future research on automatic worm signature generation should focus on provable mistake bounds for pattern-extraction-based learners and on further analysis of and improvements to automated semantic analysis techniques.

6 Related Work

Attacking learning algorithms Barreno *et al.* independently and concurrently investigate the challenge of using machine learning algorithms in adversarial environments [1]. The authors present a high-level framework for categorizing attacks against machine learning algorithms and potential defense strategies, and analyze the properties of a hypothetical outlier detection algorithm. Our work is more concrete in that it specifically addresses the challenge of machine learning for automatic signature generation, and provides in-depth analysis of several practical attacks.

Perdisci *et al.* independently and concurrently propose attacks [18] against the learning algorithms presented in Polygraph [15]. Their work shows how an attacker able to systematically inject noise in the suspicious pool can prevent a correct classifier from being generated, for both conjunction and Bayes learners. Their attack against the Polygraph Bayes signature generation algorithm is similar to our correlated outlier attack, though we further generalize the attack to show both how it can be performed even without suspicious pool poisoning, and how it can be strengthened with innocuous pool poisoning.

Pattern-extraction signature generation Several systems have been proposed to automatically generate worm signatures from a few collected worm samples. Most of these systems, such as Honeycomb [8], EarlyBird [21], and Autograph [7], have been shown not to be able to handle polymorphic worms [15]. While PADS [24] has been shown to be robust to obfuscation of the worm code, it is unclear whether it would work against encrypted code combined with only a small obfuscated decryption routine.

Polygraph [15] demonstrates that it is possible to generate accurate signatures for polymorphic worms, because there are some features that must be present in worm infection attempts to successfully exploit the target machine. Polygraph also demonstrates automatic signature-generation techniques that are successful against maximally-varying polymorphic worms.

Hamsa [9] is a recently proposed automatic signature generation system, with improvements in performance and noise-tolerance over Polygraph. As we discuss in Section 5, it is more resilient than Polygraph to the attacks presented here, but not entirely resilient.

Semantic analysis Recent research proposes performing automated *semantic* analysis of collected worm samples, by monitoring the execution of a vulnerable server as it gets compromised [2,3,5,10,25]. These approaches can identify what features of the worm request *caused* it to exploit the monitored software, and are hence likely to be invariant, and useful in a signature. This approach is also less susceptible to be fooled by the worm into using spurious features in the signature, since it will ignore features that have no effect on whether the vulnerable software actually gets exploited. The features identified can also be more expressive than the simple presence or absence of tokens, specifying such things as the minimum length of a protocol field necessary to trigger a buffer overflow.

Execution filtering In this paper we seek to address the problem of automatically generating worm signatures. Other recent research proposes using semantic analysis to generate *execution filters*, which specify the location of a vulnerability, and how to detect when it is exploited by automatically emulating [20] or rewriting [14] that part of the program.

7 Conclusion

Learning an accurate classifier from data largely controlled by an adversary is a difficult task. In this work, we have shown that even a *delusive* adversary, who provides correctly labeled but misleading training data, can prevent or severely delay the generation of an accurate classifier. We have concretely demonstrated this concept with highly effective attacks against recently proposed automatic worm signature generation algorithms.

When designing a system to learn in such an adversarial environment, one must take into account that the adversary will provide the *worst possible* training data, in the *worst possible* order. Few machine learning algorithms provide useful guarantees when used in such a scenario.

The problem of a delusive adversary must be taken into account in the design of malicious classifier generation systems. Promising approaches include designing learning algorithms that are robust to maliciously generated training data, training using malicious data samples *not* generated by a malicious source, and performing deeper analysis of the malicious training data to determine the semantic significance of features before including them in a classifier.

References

1. Marco Barreno, Blaine Nelson, Russell Sears, Anthony D. Joseph, and J. D. Tygar. Can machine learning be secure? In *ASIA CCS*, March 2006.
2. David Brumley, James Newsome, Dawn Song, Hao Wang, and Somesh Jha. Towards automatic generation of vulnerability-based signatures. In *IEEE Symposium on Security and Privacy*, 2006.
3. Manuel Costa, Jon Crowcroft, Miguel Castro, and Antony Rowstron. Vigilante: End-to-end containment of internet worms. In *SOSP*, 2005.
4. Jedidiah R. Crandall and Fred Chong. Minos: Architectural support for software security through control data integrity. In *International Symposium on Microarchitecture*, December 2004.
5. Jedidiah R. Crandall, Zhendong Su, S. Felix Wu, and Frederic T. Chong. On deriving unknown vulnerabilities from zero-day polymorphic and metamorphic worm exploits. In *12th ACM Conference on Computer and Communications Security (CCS)*, 2005.
6. Nilesh Dalvi, Pedro Domingos, Mausam, Sumit Sanghai, and Deepak Verma. Adversarial classification. In *Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2004.
7. Hyang-Ah Kim and Brad Karp. Autograph: toward automated, distributed worm signature detection. In *13th USENIX Security Symposium*, August 2004.
8. Christian Kreibich and Jon Crowcroft. Honeycomb - creating intrusion detection signatures using honeypots. In *HotNets*, November 2003.
9. Zhichun Li, Manan Sanghi, Yan Chen, Ming-Yang Kao, and Brian Chavez. Hamsa: fast signature generation for zero-day polymorphic worms with provable attack resilience. In *IEEE Symposium on Security and Privacy*, May 2006.
10. Zhenkai Liang and R. Sekar. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *12th ACM Conference on Computer and Communications Security (CCS)*, 2005.
11. N. Littlestone. Learning quickly when irrelevant attributes abound: A new linear threshold algorithm. *Machine Learning*, 2(285-318), 1988.
12. N. Littlestone. Redundant noisy attributes, attribute errors, and linear-threshold learning using winnow. In *Fourth Annual Workshop on Computational Learning Theory*, pages 147–156, 1991.
13. Daniel Lowd and Christopher Meek. Adversarial learning. In *Eleventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2005.
14. James Newsome, David Brumley, and Dawn Song. Vulnerability-specific execution filtering for exploit prevention on commodity software. In *13th Symposium on Network and Distributed System Security (NDSS'06)*, 2006.
15. James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. In *IEEE Symposium on Security and Privacy*, May 2005.
16. James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *12th Annual Network and Distributed System Security Symposium (NDSS)*, February 2005.
17. delusive (definition). In *Oxford English Dictionary*, Oxford University Press, 2006.
18. Roberto Perdisci, David Dagon, Wenke Lee, Prahlad Fogla, and Monirul Sharif. Misleading worm signature generators using deliberate noise injection. In *IEEE Symposium on Security and Privacy*, May 2006.
19. Yann Ramin. ATPhttpd. http://www.redshift.com/∼yramin/atp/atphttpd/.
20. Stelios Sidiroglou, Michael E. Locasto, Stephen W. Boyd, and Angelos D. Keromytis. Building a reactive immune system for software services. In *USENIX Annual Technical Conference*, 2005.
21. Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *6th ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*, December 2004.
22. S. Staniford, D. Moore, V. Paxson, and N. Weaver. The top speed of flash worms. In *ACM CCS WORM*, 2004.
23. G. Edward Suh, Jaewook Lee, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS*, 2004.
24. Yong Tang and Shigang Chen. Defending against internet worms: A signature-based approach. In *IEEE INFOCOM*, March 2005.
25. Jun Xu, Peng Ning, Chongkyung Kil, Yan Zhai, and Chris Bookholt. Automatic diagnosis and response to memory corruption vulnerabilities. In *12th Annual ACM Conference on Computer and Communication Security (CCS)*, 2005.