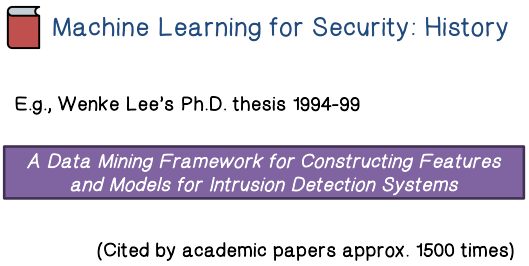
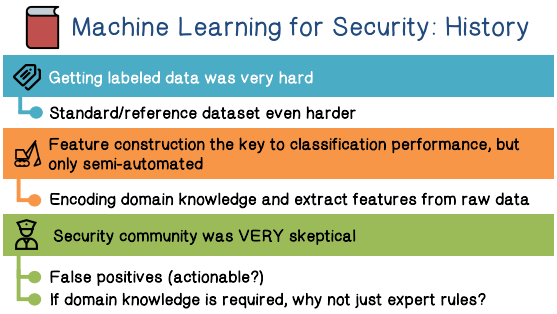
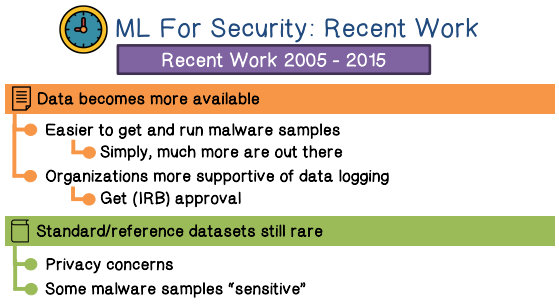
With machine learning now being applied to many critical applications, including security analytics, we can expect that attackers will try to compromise the process of machine learning. In fact, attackers have begun to do just that. In this lesson, we discuss attacks on machine learning including data poisoning, where the attacker can cause machine-learning to produce the role model and model evasion, where the attacker can learn about the decision boundary of a model and then bypass it.



You might have thought that machine learning for security is a recent phenomenon, but it has a long history, at least 20 years. For example, I did my PhD thesis in this topic from 1994-1999.

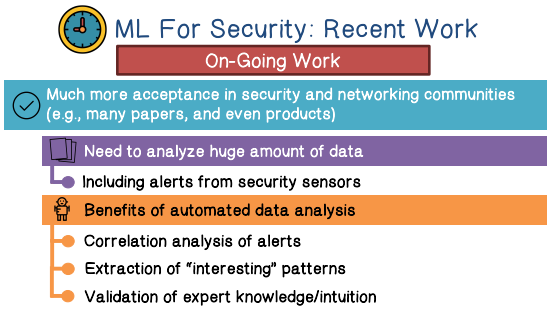


In the early days, getting labeled training data was very hard, and that is because our understanding of attacks was limited and there were very few attack examples. Since researchers were still trying to learn how to apply machine learning for security, creating a standard or reference dataset so that every researcher could use was even harder. As we have discussed, the key to have good performance in machine learning for security is to have a very good set of features. However, feature construction in the early days was mostly manual. The feature construction process typically involves encoding domain knowledge and extract the features from raw data. For example, the way that we process tcpdump data involves using our domain knowledge to decide what features are useful for detecting intuitions. The security experts were typically very skeptical about the approach of applied machine learning for security, and there were several reasons. One was the perceived high rate of false positives from these models. We know that if a model produces too many false positives, then this model is not useful. Another reason was that if we had to use domain knowledge to extract features, then the experts can ask, why don't we just hand code expert rules?

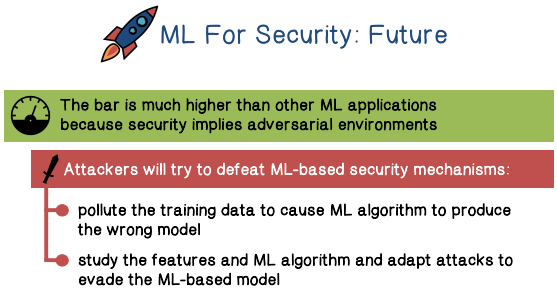


Things have changed quite a bit over the last ten years. First of all, there is a lot more data now available for research. For example, now, it is very easy to obtain malware samples and run these samples to obtain data. There is a lot of data sharing going on. When it comes down to normal training data, organizations now are more supportive of data logging. For example, many

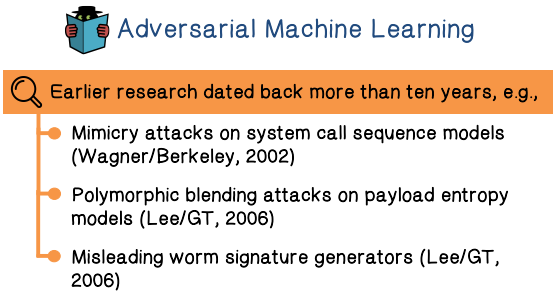
organizations now allow network traffic to be recorded. Of course, if you work for a research institution, before you start logging data, you need to get approval from the Institutional Review Board. However, it is still hard to find standard or reference datasets, that is because it is hard to share normal training data because of privacy concerns and some malware samples are considered sensitive, that is, they are not to be shared.



We have seen many papers and even products in applying ML for security. Basically, everyone now realizes the need to analyze a huge amount of data. That is, manual process will not work, and therefore, everyone realizes that we have to apply data analysis in particular machine learning to process such a huge amount of data. Interestingly, the security sensors themselves also produce a huge amount of data, for example, the alerts. Therefore, the huge number of alerts also need to be processed. Typically, machine learning algorithm can be used to analyze these alerts. There are many benefits of automatic data analysis. For example, we can perform a link or correlation analysis on alerts. We can also extract interesting or previously unknown patterns from data. For example, we may find that suddenly activities tend to appear together. We can also validate if our so-called expert knowledge or intuition is correct. For example, if our intuition is that it is extremely rare for machine to send out email traffic at 3:00 AM, we can actually validate this in data.

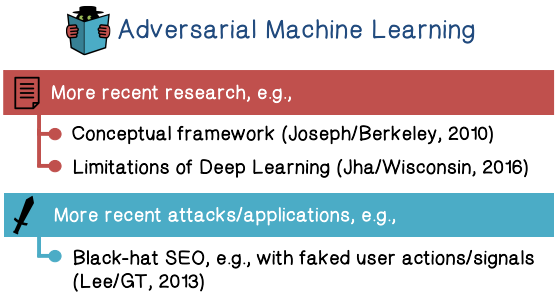


Machine learning is now being applied to many different applications. For security, the bar is much higher because machine learning has to work in not only the normal cases, but also in cases where there are attackers around. We know from history that, attackers have tried to evade or defeat firewalls, IDS and malware analysis systems. So, we know that attackers will try to defeat machine learning-based security mechanisms. For example, attackers can pollute the training data so that the machine learning process will produce the wrong model. Attackers can also study a machine learning model and adapt their attacks to evade the model.

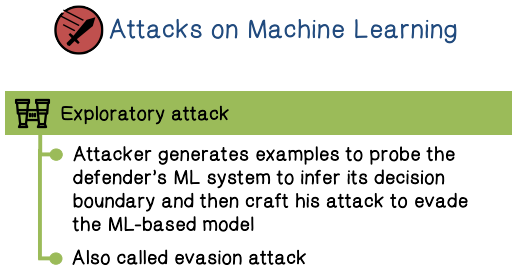


Adversarial machine learning refers to machine learning in a setting where there are attackers around. Early research in this field dates back to early 2000s. For example, researchers showed that by inserting normal system calls into attack sequences, the attacks can evade the anomaly detection model. Similarly, by including normal contents in the attack payload, an attack can evade a network anomaly detection model. The

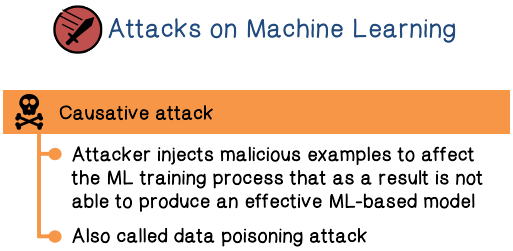
researchers also demonstrated that by inserting incorrect network data, attackers can force the machine learning process to produce incorrect worm signatures.



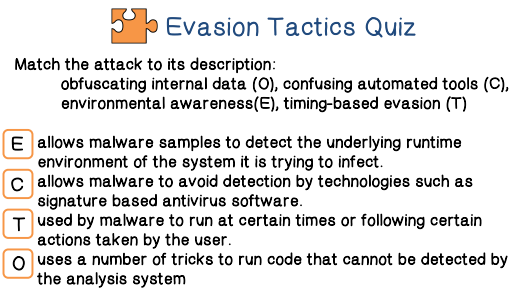
More recent research has been focusing on the conceptual framework of adversarial machine learning, and the limitation of the emerging machine learning paradigms such as deep learning. There are also demonstrations of attacks on new machine learning applications such as web services. For example, attackers can inject fake user actions and signals to influence such results.



Conceptually, there are two types of attacks on machine learning. The first type is the exploratory attack. Here, an attacker uses examples to find out the decision boundary of a machine learning model, and then crafts his attack to evade this model, this is also called evasion attack.



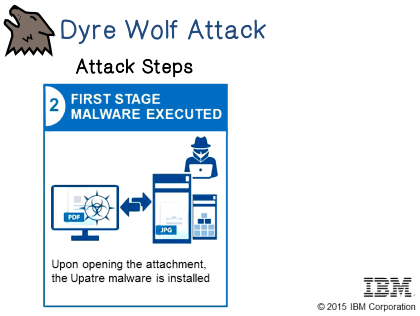
The second type is the causative attack. Here, the attacker injects bad or malicious examples into the training data, so that the wrong model is produced. This is also called the data poisoning attack.



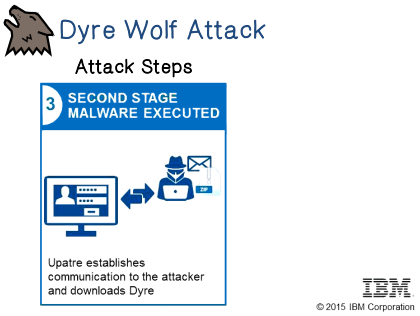
Let us do a quiz on evasion tactics. Match the attack to its description. We use O to represent obfuscating internal data, C to represent confusing automatic tools, and E to represent environmental awareness, and T to represent timing-based evasion.

The first description allows malware samples to detect the underlying runtime environment of the system it is trying to infect. This is E. Environmental awareness. The second description allows malware to avoid detection by technologies such as signature-based antivirus software. This is C. Confusing automated tools. The third description used by malware to run at certain times or following certain actions taken by the user. This is T. Timing-based evasion. The fourth description uses a number of tricks to run code that cannot be detected by the analysis system. This is obfuscating internal data.

Here is an example of real-life evasion attack. The first step of this attack is spear phishing. That means an employee receives an email with the malware.

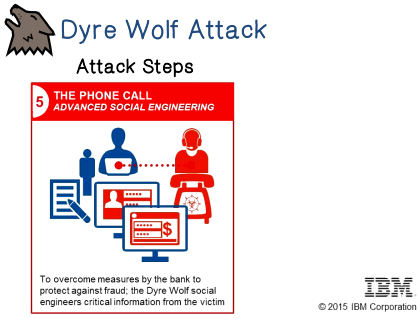


And the second step when the employee opens the attachment of the email, the malware is installed.

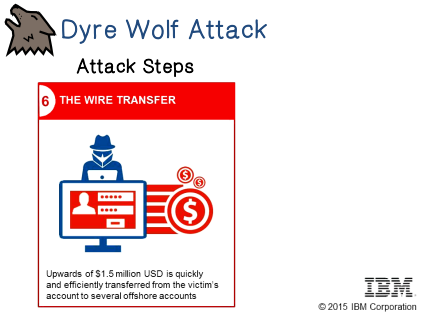


Next, the malware executes and connects to the attack server to download a new malware program.

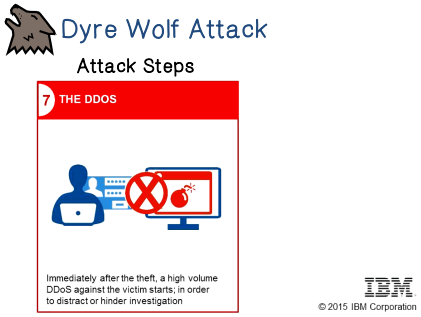
When the victim logs to a bank, the malware running on his machine will change the response from the bank so that the victim is checked to call an illegitimate number.



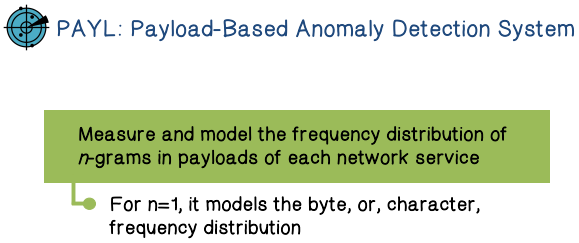
When a victim calls this attacker's number, the attacker can then use social engineering techniques to extract critical personal information from the victim.



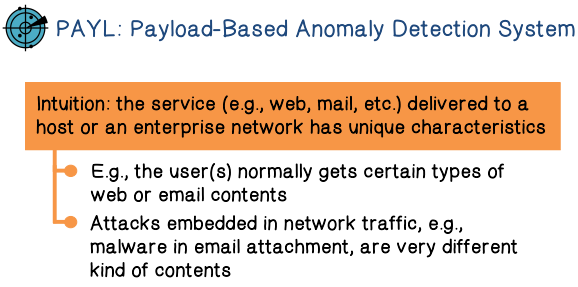
The victim's personal information can then be used by the attacker to bypass the fraud detection techniques employed by the bank to convince the bank to transfer money from the user's account to the attacker's account.



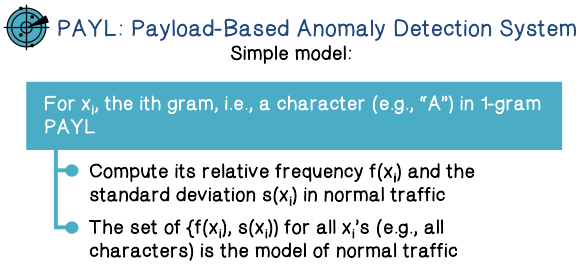
After the money transfer, the attacker launched a noisy Denial-of-Service attack to distract or hinder investigation.



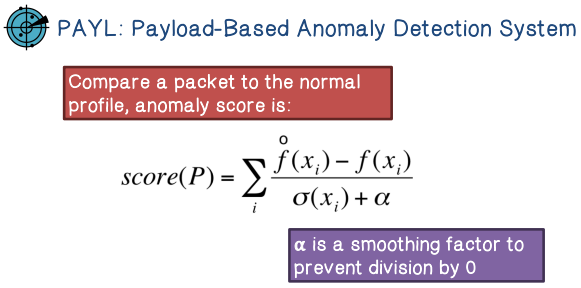
Now, let us discuss evasion attacks in more details. In particular, is it possible or how easy is it? We use a simple anomaly detection system as an example. This system is called PAYL, or, Payload-Based Anomaly Detection System. It is a very simple system. Essentially for each network service such as HTTP, it would measure and model the frequency distribution of the n-grams in traffic payloads. If n equals to one, that means it is measuring the byte frequency distribution.



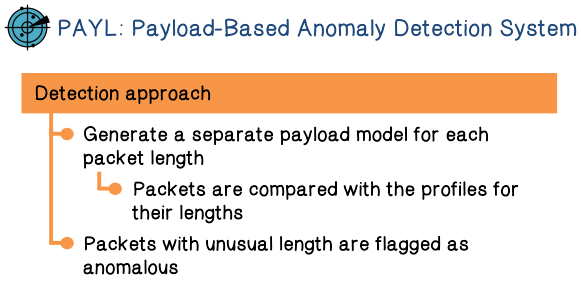
The intuition is that each service, such as web or email that is delivered to a host on an enterprise network has its own unique characteristics. For example, a particular user normally gets certain types of web or email contents. On the other hand, attacks embedded in network traffic, such as malware in an email attachment, are very different kind of contents.



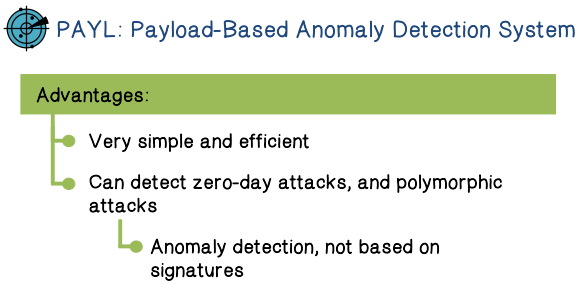
We use n equal to one or one gram, as example to explain PAYL. A one gram is essentially a character, therefore, PAYL computes for each character its relative frequency and a standard deviation. Then a set of frequencies and standard deviations for all the characters in a normal traffic data is then the normal traffic profile.



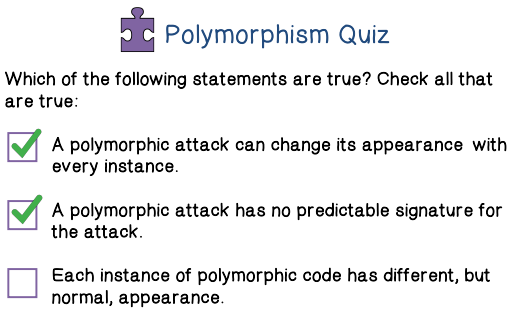
Once we have the model or the profile of normal traffic, which is a set of byte frequencies and standard deviations, we can then compute the anomaly score of a packet. It is essentially the sum of fraction of deviation across all bytes in the packet. Alpha here is a smoothing factor to prevent division by zero.



PAYL is an anomaly detection system that uses a normal payload model for each packet length. A packet is identified as anomalous if its payload deviates significantly from the normal profile associated with the packet length or it has unusual length.



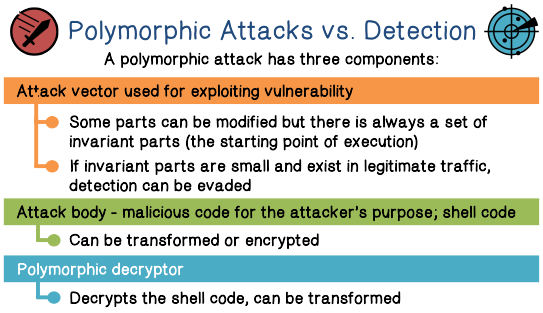
As you can see, PAYL is really simple and therefore it is also very efficient in run-time. Since it is an anomaly detection system, it can detect zero-day attacks, and polymorphic attacks, as long as these attacks look different from the normal traffic.



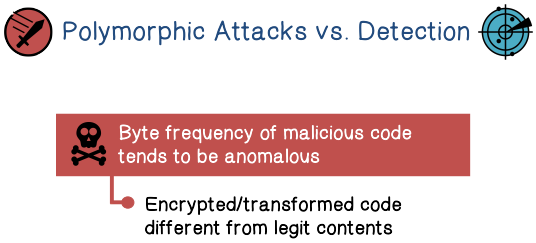
Let us do a quiz on polymorphism. Which of the following statements are true?

First, a polymorphic attack can change its appearance with every instance. This is true. Second, a polymorphic attack has no predictable signature for the attack. This is also true. The third statement, each instance of polymorphic code has

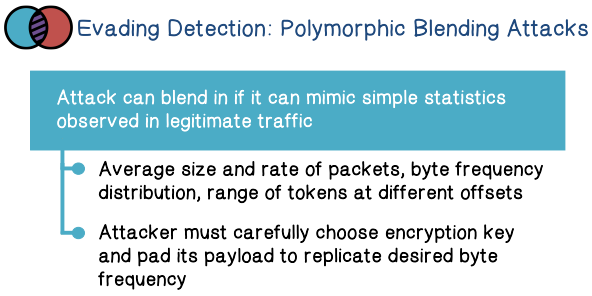
different by normal appearance. This is false, because polymorphic code does not resemble normal code.



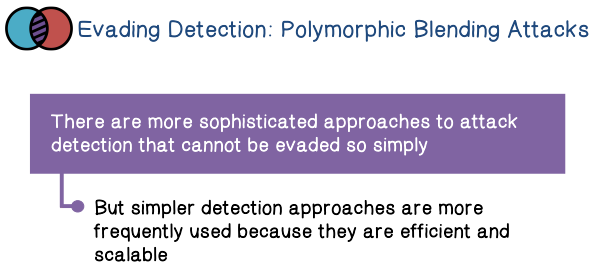
Polymorphic attack has three main components. The first is an attack vector used for exploiting vulnerability. There is always a small invariant part of the attack vector, typically, the starting point of execution. But, if this part is small enough and exist in normal traffic, then this part can evade detection. The second component is the attack body. This contains the attacker's shell code. This part is typically transformed or encrypted. Since the attack body is encrypted, it needs to be decrypted in order to execute. Therefore, the third component of a polymorphic attack is the polymorphic decryptor, and it can also be transformed.



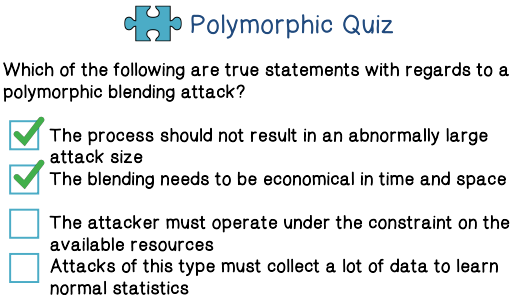
Since in a polymorphic attack, the transformed attack body and decryptor tend to have different byte frequencies than a normal traffic, the polymorphic attack can in theory be detected by anomaly detection system such as PAYL.



In order to defeat anomaly detection system, an attacker can use polymorphic blending attack instead of the simple polymorphic attack. In a polymorphic blending attack, each polymorphic instance will try to blend in with normal traffic by matching the normal profile of the legitimate traffic. For example, each polymorphic instance can have its byte frequency distribution matching the normal profile.

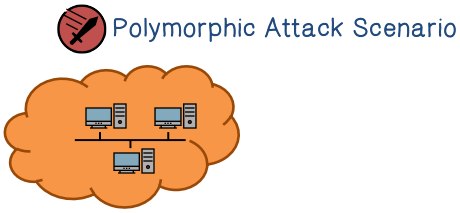


Now, of course, if an IDS uses a very comprehensive set of features and more sophisticated modeling algorithms, it would be harder for attacker to evade detection. That is, evasion attacks are more feasible against a simpler IDS, such as PAYL.

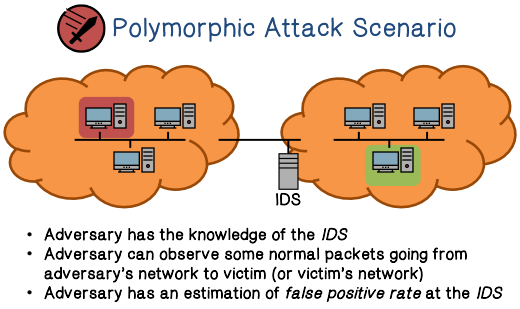


Now, let us do a quiz. Which of the following statements are true with regards to polymorphic blending attack?

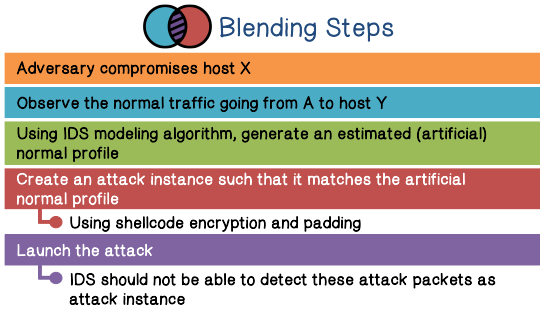
First, the process should not result in an abnormally large attack size. This is true because otherwise this by itself can raise an alert. Second, the blending needs to be economical in time and space. This is true because otherwise, it will not only slow down the attack, but also increase the chance of detection by the IDS. Third, attacks of this type must collect a lot of data to learn normal statistics. As we will show with PAYL, this is not true.



Now, let us discuss polymorphic blending attacks against PAYL. And here is their attack scenario.

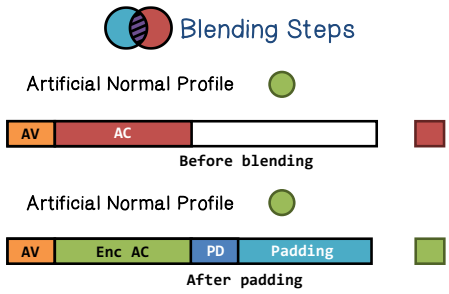


Suppose, the attack is already in a network that has a relationship with a target network, that is, the attacker can choose a machine in network A and this network B is the target. B receives traffic from network A, and it has an IDS. The attacker has knowledge of the IDS. For example, the attacker knows that this IDS is based on PAYL. This is a reasonable assumption because, for example, because he can read purchase and sales news and read the documents on PAYL. The attacker can then send traffic from network A to network B, that is, the attacker can observe which traffic is accepted or rejected by the IDS and can estimate the accuracy of the IDS.

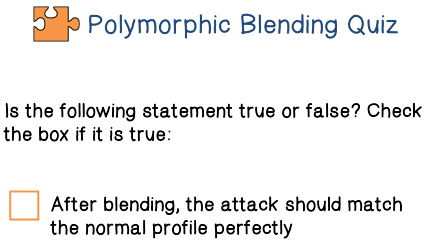


Here are the steps of a polymorphic blending attack. The attacker first compromises a machine. He can then observe normal traffic going from network A to another host in network B. The attacker already knows the algorithm behind IDS at B. For example, he knows that PAYL is used. Therefore, using normal traffic data from network A to network B, the attacker can use PAYL to produce an estimated or artificial normal profile.

Then, the attacker can craft his attack, so that it matches the artificial normal profile. As we will demonstrate, this involves shellcode encryption and padding. He can send this attack traffic, which should be able to evade the detection of the IDS. The reason is that this attack already matches the artificial profile, which should be very close to real profile used in network B.

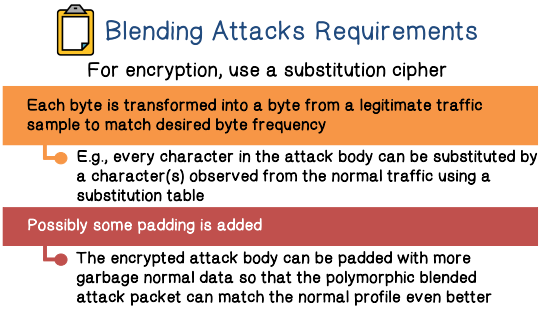


Here is the original attack packet. They include the attack vector, and the attack body, or, the attack code. Of course, this attack traffic looks very different from the normal traffic. Therefore, it can be detected by an IDS such as PAYL. In blending, the attacker can encode or encrypt the attack code using only the normal characters. Of course, he now needs to include the decryptor. He can transform the attack vector and the decryptor to use as many normal characters as possible. On the other hand, the packet so far may not match the normal frequency distribution, but the attacker can add more padding of normal characters, so that the byte frequency distribution of the whole packet will now match the artificial normal profile. Therefore, the attacker can now evade the anomaly detection system at the target network.



Now, let us do a quiz on polymorphic blending attack. Is the following statement true or false? Check the box if it is true.

After blending, the attack should match the normal profile perfectly. This is false. Because, first of all, attackers only blend the profile that approximates the real profile. Second, as long as the packet matches the profile, within variance, it is accepted, and therefore, the blended attack does not have to match perfectly with the actual normal profile. It just needs to be close enough.



To recap, in a polymorphic blending attack, the most important step is to transform the attack code into normal data, that is, each byte is transformed to a byte from legitimate traffic, so that the desired byte frequency is matched. We can use a substitution table for this purpose since the attack vector and decryptor cannot be randomly transformed. We may need to add more paddings of normal characters, so that the entire packet will match the desired byte frequency.

Encrypting Attack Contents

Greedy Method

Map the most frequent byte value in attack traffic to the most frequent byte value in normal traffic

Map the second most frequent byte in attack traffic to the second most frequent by value in normal traffic

- and so on until all byte values in attack traffic have been mapped

Finding an optimum substitution table that only requires minimum padding is a very hard problem. We can instead use a greedy method, that is, we can map the most frequent byte value in attack traffic to the most frequent byte value in normal traffic. We then map the second most frequent byte value in attack traffic to the second most frequent byte value in normal traffic, and so on and so forth, until all byte values in attack traffic have been mapped.

Decryptor

Decryption removes the padding and reverses the substitution steps

This code cannot be blended but can be transformed into equivalent instructions that contain (mostly) normal byte values

The substitution table used for decryption can be stored in an array where the i^{th} entry is the byte value in normal traffic that was used to substitute byte value *i* in attack traffic

- This table contains only normal byte values

The job of the decryptor is to remove the padding and reverse the substitution step. The decryptor code can be transformed into equivalent instructions that contain mostly normal byte values. The substitution table itself can also be represented in such a way that it contains only the normal byte values. For example, the i^{th} entry is the byte value in normal traffic that was used to substitute the byte value i in attack traffic.

Evaluation

Create polymorphic blending attacks to evade PAYL

First create polymorphic attacks using CLET and verify that they are all detected by PAYL

Next create polymorphic *blending* attacks and demonstrate that they can evade detection

- Evaluate easiness of attack construction and cost

Now, let us discuss some evaluation results. First, we verify that PAYL can detect polymorphic attacks without blending. Next, we create polymorphic blending attacks and demonstrate that they can evade detection by PAYL. We also evaluate how easy it is to construct such blending attacks.

Evaluation Results

The attacker needs to learn/estimate the normal profile used by PAYL at the target network

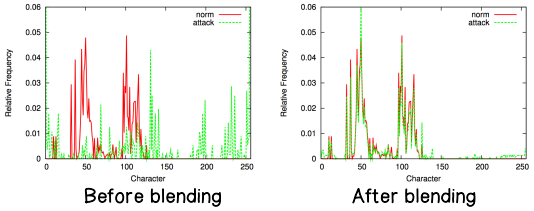
- Training of the artificial profile is stopped when there is no significant improvement over existing profile (measured using Manhattan distance) within two packets
- Number of packets required for convergence

Packet Length	1-gram	2-gram
418	8	20
730	8	18
1460	14	40

One measure on how easy it is to construct such blending attack is the number of packets required to train an artificial normal profile. For two-gram PAYL, we need more, but still no more than 100 packets. This result shows that it is quite easy to learn the artificial normal profile.

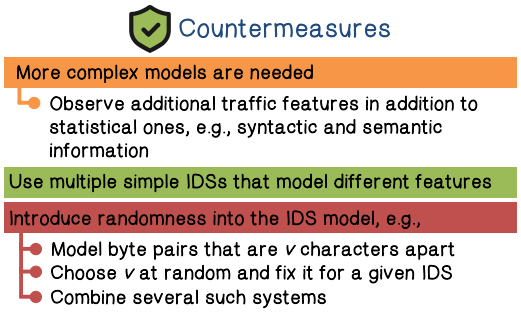
Evaluation Results

Frequency Distribution

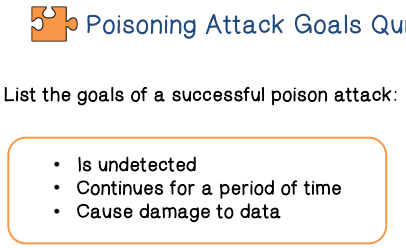


Before blending After blending

Here we show that, before blending, the byte frequency distribution of the data packet looks very different from the byte frequency distribution of the normal traffic. We show that after blending the byte frequency distribution of the attack data packet now matches the normal profile. That is, with blending, the attack traffic can evade detection by PAYL.

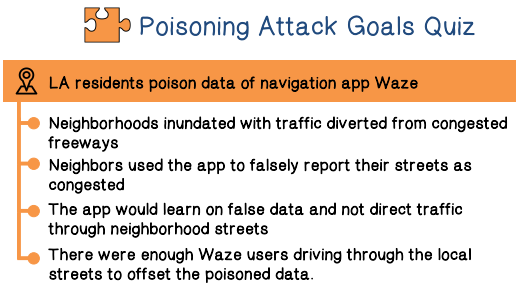


So, what are the possible countermeasures to polymorphic planning attacks? Obviously, we should use more complex models. PAYL is a very simple model that uses only the statistical features. A more sophisticated model can use syntactic and semantic information, such as the syntax and meanings of email or web contents. Even with simple features, we can combine multiple models, or we can introduce randomness into IDS model. For example, we can use 2-gram PAYL, but instead of two consecutive bytes, we can model byte pairs that are v characters apart. We can choose v at random and fix it for given IDS model. Then we can combine multiple such systems.

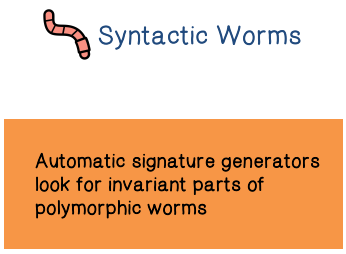


Now, let's do a quiz on the goals of a successful poisoning attack.

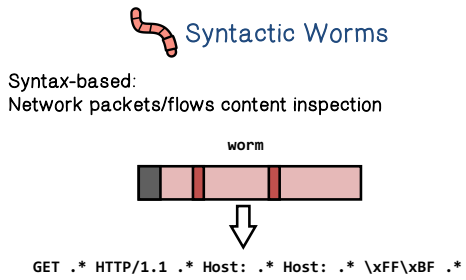
These goals include degree of role model so that the attack is not detected. Second, the attack can be subtle so that no one knows that the data has been poisoned for quite a while. The attack can be permanent so that it is not possible to repair the damage training data.



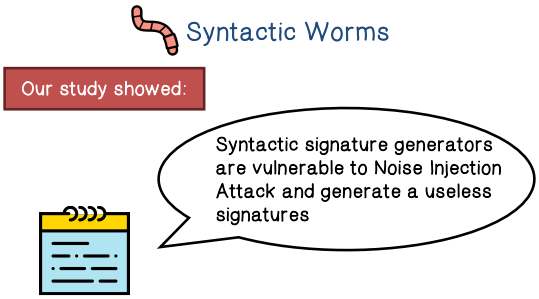
Here is a real-life example of poisoning attack. Residence of a neighborhood in LA attempted to poison data used by a navigation app. They were tired of the app diverting drivers from the congested 405 freeway through their neighborhood. So, they used the app to falsely report congestion on the neighborhood streets. Their hope was that the app will learn on false data and not divert traffic through the neighborhood streets. Unfortunately, this did not work. The reason was that, far greater number of drivers using the app reported that the neighborhood streets were not congested. The moral of the story is that, in this case, the signal far outweighs the noise. Therefore, the noise could not affect the outcome. As we shall see, in other cases, the attackers can control the noise level to affect the learning outcome.



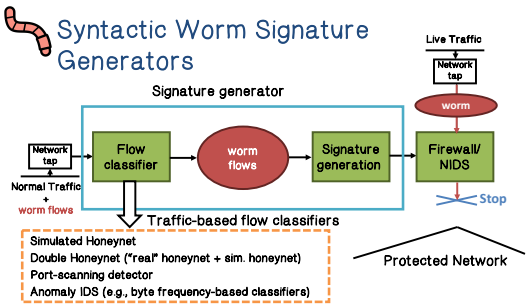
Now, let us discuss the case of data poisoning attack. In this attack, the attacker injects noise in the training data, so that a machine learning algorithm cannot learn the proper worm signatures. First, some background. Researchers have proposed to use machine learning algorithms to produce signatures for worms including polymorphic worms.



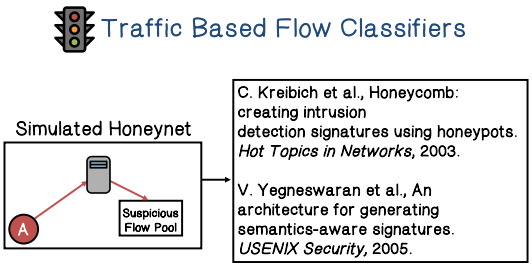
In particular, by examining the network packets related to a worm, we can extract a regular expression that describes all variants of the worm, and the worm signatures are based on this regular expression, and such regular expressions form the basis of worm signatures.



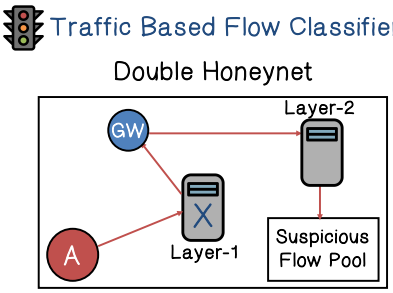
Our studies show that if the attacker can inject noise in the training data, then the machine learning algorithm cannot generate effective worm signatures.



Let us discuss the workflow of worm signature generation. We first capture normal traffic and worm traffic. Then the traffic can be clustered and classified. Then a set of worm flows can be used to generate signatures. The signatures can be installed at a firewall or NIDS. These signatures can then be used to stop traffic of worm. The first question is how we cluster or classify flows into normal versus worms. There are a number of techniques here.

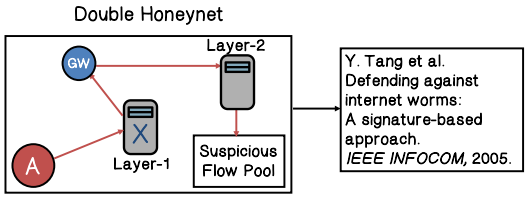


We can use Simulated Honeynet. A Simulated Honeynet simulates host with individual networking personalities. They intercept traffic sent to nonexistent hosts and use a similar system to respond to this traffic. Any flow that is sent to this Honeynet is inserted into the suspicious flow pool. You can read these papers about Honeynets.



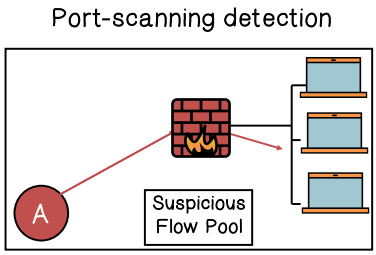
We can also use the so-called Double Honeynet. In this case, the first layer honeynet is made of real hosts. Whenever a first layer honeypot is infected by a worm, its outgoing traffic is redirected to a second layer simulated honeynet and the flow is inserted into the suspicious flow pool.

Traffic Based Flow Classifiers



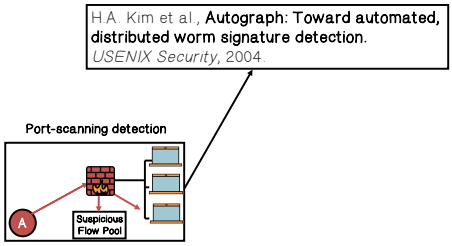
And you can read this paper about Double Honeynet.

Traffic Based Flow Classifiers



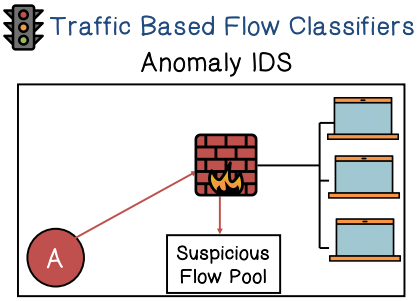
Since many worms perform port scanning, we can use port scanning detection to capture worm flows. That is, traffic flows that perform post scanning are collected into the suspicious flow pool.

Traffic Based Flow Classifiers



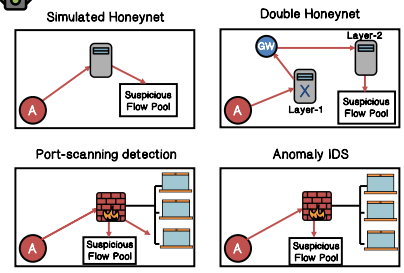
And you can read this paper about this approach.

Traffic Based Flow Classifiers

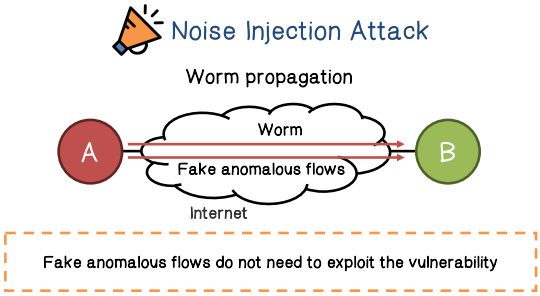


We can also use an Anomaly detection system to capture suspicious worm flows and any anomalous flow can be automatically put into the suspicious flow pool. For example, PAYL uses the byte frequency distribution of the normal packets to detect anomalies and any anonymous flow can be put into the suspicious flow pool.

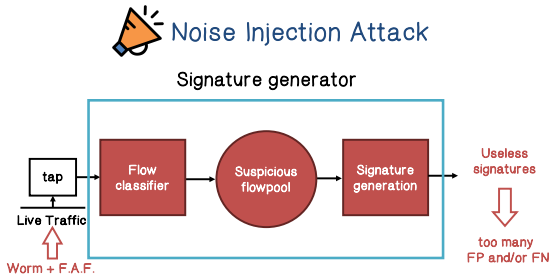
Traffic Based Flow Classifiers



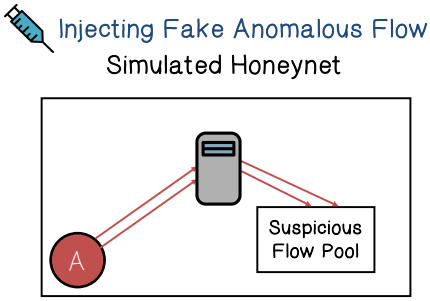
So, these are the techniques that we can use to collect suspicious worm flows.



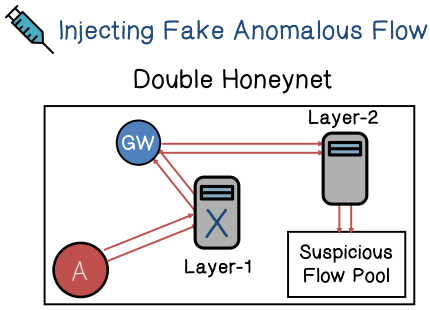
What will happen if an attacker on purpose inject noise in the training data? But how can the attacker do that? We know that worm propagates by sending out worm traffic. What if the worm also sends out fake anomalous flows? These fake anomalous flows do not need to exploit any vulnerability. They just need to look anomalous or look like worms.



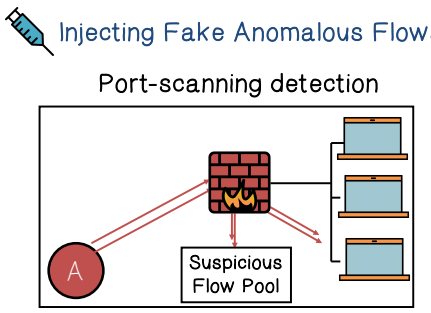
With fake anonymous flows injected, the flow classifier cannot detect or identify these fake anonymous flows, and therefore, these fake anonymous flows will be included into the suspicious flow pool. As a result, the machine-learning algorithm cannot generate useful worm signatures because these signatures will have too many false positives and false negatives.



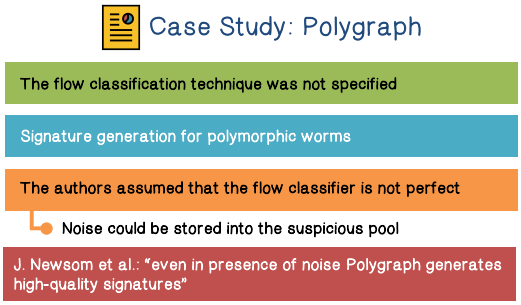
Let us take a look at why the flow classifiers cannot identify the fake anomalous flows. In a case of simulated honeynet, since both the real worm and the fake anonymous flow are sent to the same destination at the same time, they will be both consider suspicious, and therefore, they will be both included into the suspicious flow pool.



In the case of double honeynet, the real worm will infect a first layer honeypot, whereas the fake anomalous flows will not, and will be disregarded. However, the real worm infection will again send a real worm flow and a fake anomalous flow. Therefore, both the real worm flow and a fake anomalous flow will be included in the suspicious flow pool.

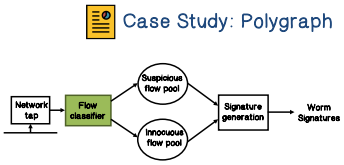


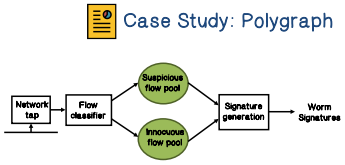
In the case of port-scanning detection, since the real worm flow and a fake anomalous flow both behave like scanning traffic, they will be both included in a suspicious flow pool. Similarly, in the case of anomaly detection, a fake anomalous flow will be detected as an anomaly, and therefore, it will also be included to the suspicious flow pool.

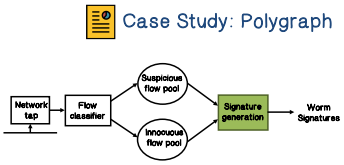


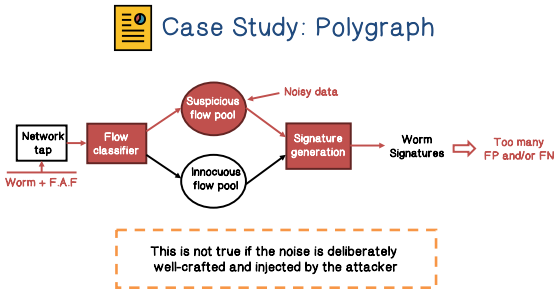
Now let us take a look at polygraph. Polygraph generates warm signatures. Polygraph can use any of the flow classification techniques that we have discussed. It can generate signatures for polymorphic worms. The system was designed to handle noise in the training data because no flow classifier is perfect. The system was believed to be able to generate high-quality signatures even in the presence of noise.

Below is the workflow of Polygraph. First, flow classifier is applied to network flows. The flows are classified into suspicious flows or innocuous flows. Machine-learning algorithms are then applied to this training data to learn the warm signatures.

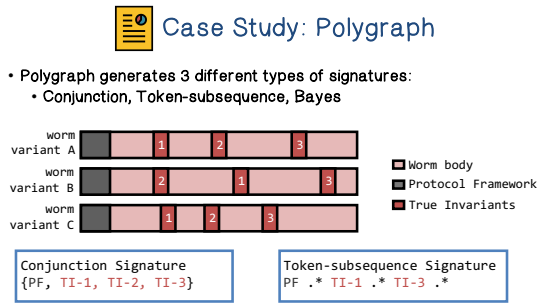




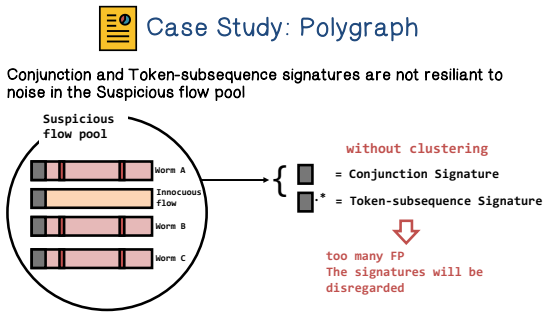




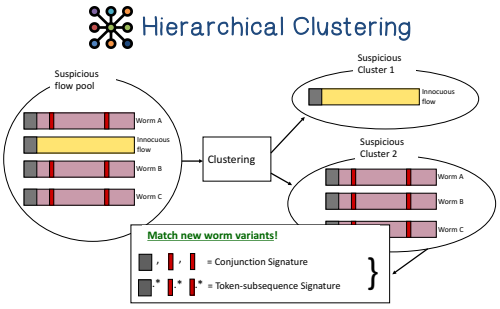
We are going to show that this is not true when noise is deliberately injected by an attacker. That is, if the attacker crafts and injects fake anonymous flows, then these flows will be included into the suspicious flow pool. As a result, the machine-learning algorithms which generally signatures that have too many false positives and false negatives.



Polygraph generates three types of signatures. These are conjunction, token-subsequence, and Bayes. Suppose we have several polymorphic variants of the worm. Each workflow, we include protocol framework, warm body, and true invariants. First, the strings that are common to all suspicious flows are extracted. These strings are called tokens. A conjunction signature is an ordered set of tokens. On the other hand, token-subsequent signature is a sequence or order set of tokens.



We are going to show that conjunction and token-subsequent signatures are not resilient to noise injected by attackers.



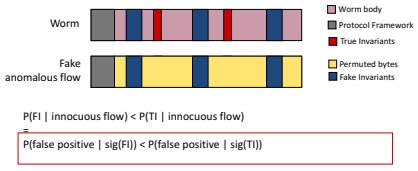
First, let us discuss the clustering step in Polygraph that is designed to filter out noise. As shown in the example, if the suspicious flow pool happens to include this innocuous flow which is a noise, the signature will only include the protocol framework which will result in many false positives. Such signatures are useless. Therefore, before generating signatures, Polygraph first performs clustering so that the worm flows will be clustered together, and the innocuous flow will be in its own cluster. Therefore, the proper worm signatures can be generated.

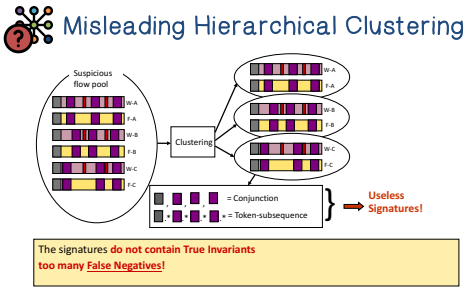
As we have just discussed, clustering is used by Polygraph to filter the noise in training data. Therefore, the fake anomalous flows inserted by the attacker need to first defeat clustering so that only useless signatures will be produced.

Objective: Caft the fake anomalous flows so that the Hierarchical Clustering cannot filter the noise

- the extracted signatures will produce False Negatives

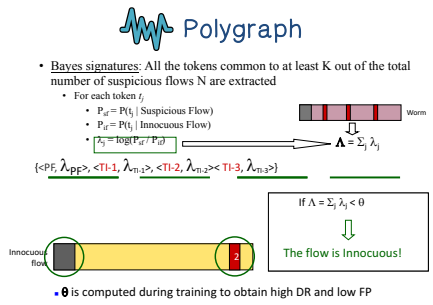
Let us illustrate how an attacker can craft fake anomalous flows. Here is the original worm flow. The attacker can permute the worm body. He then creates fake invariants. These fake invariants are set in such a way that they are even less likely to appear in innocuous flows than true invariants. Which means, if signatures are based on these fake invariants, they are going to produce less false positives. This means that, signatures that are based on fake invariants will be selected over signatures based on true invariants. In other words, the incorrect signatures are now produced.



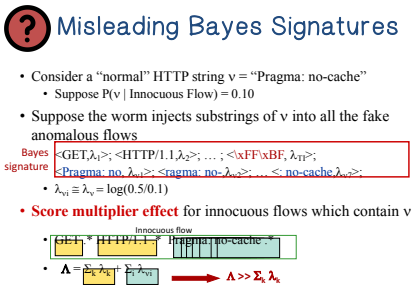


Keeping in mind the in the clustering step, we can make sure that after clustering, each cluster has a real worm flow and a corresponding crafted fake anomalous flow. This guarantees that all the signatures are based on the fake invariants. Again,

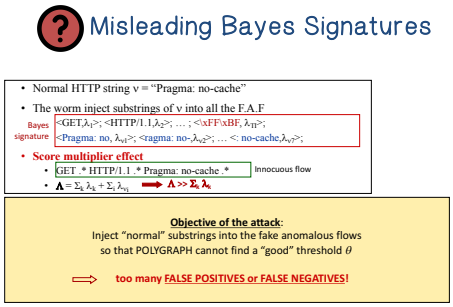
these are useless signatures because they do not detect the real worms.



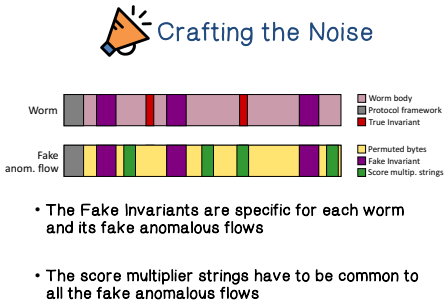
Now let us discuss the third type of signatures produced by Polygraph. These are the Bayes signatures. First, all the tokens common to at least K out of N flows in a suspicious flow pool are extracted, and then for each token, we compute the probability that it appears in a suspicious flow and the probability that in appears in an innocuous flow. Then we compute the log ratio of these two probabilities. We can call this the score of the token, then a Bayes signature is a set of tokens and the tokens scores. When we examine a flow, we sum up the token scores according to the signature. For example, suppose we have a workflow, we identify the tokens and sum up their scores. If the total score is above a threshold, then we say this is a worm flow. Suppose we have an innocuous flow, again, we identify the tokens and sum up their scores. But a total score is going to be below the threshold and therefore we say this flow is innocuous. The detection threshold is computed during training in order to have high true positive rate and low false positive rate.



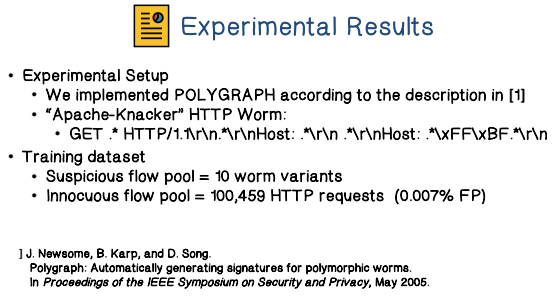
Now let us discuss how an attacker can inject fake anonymous flows to defeat Bayes signatures. Let us use HTTP traffic as an example. Consider a string in normal traffic, such as Pragma no-cache. Suppose this string shows up 10 percent of the innocuous flows. Suppose the attacker injects substrings of v into all fake anonymous flows, that is, roughly half of the suspicious flows will have these substring tokens. When we inject these substring tokens into a flow, recall that the Bayes signatures will then include these substring tokens and their scores. Now, an innocuous flow that happens to contain this string v is going to match all these substring tokens, and therefore, each signature score looks like this. Again, all the substring tokens will match, and the scores will be added together. That is, now, its signature score is much greater than before the substring tokens are used.



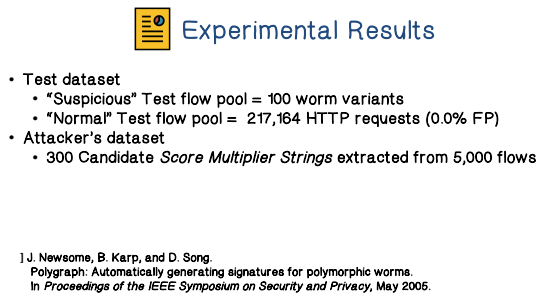
Since now the innocuous flows have very high signature scores, they will likely be classified as worm flows. That is, by injecting the normal substrings into the fake anonymous flows, the attacker can force the machine learning algorithm to produce Bayes signatures that have too many false positives or false negatives.



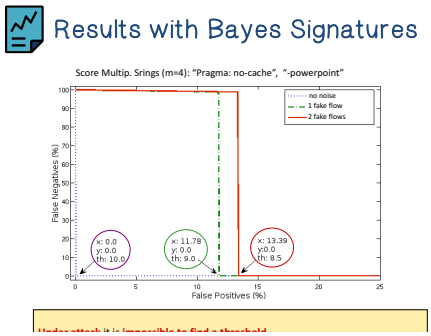
Here is how an attacker can craft a fake anomalous flow. The attacker can start with a real worm flow. Then, he can permute the bytes of the worm body, and then inject fake invariants. Again, the fake invariants are used to defeat conjunction and tokens subsequence signatures. Then, the attacker can select some normal strings in normal traffic samples. Then, inject substrings of this normal strings into the fake anomalous flows. These normal substrings are used to defeat Bayes signatures. The fake invariants are specific to each worm flow and fake anomalous flows. The fake invariants are used to defeat conjunction and token subsequence signatures. The normal substrings or the score multiplier strings are common to all fake anomalous flows. They are used to defeat Bayes signatures.



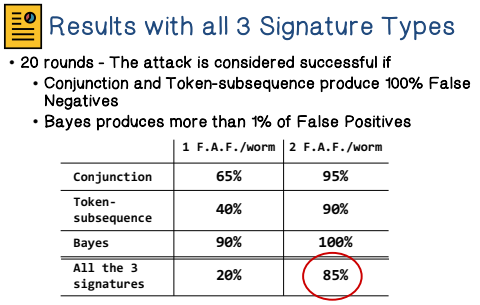
Let us briefly discuss our experiments. We implemented the Polygraph system according to this paper, and we used HTTP worm as an example. All worm and variants match this regular expression. The training dataset includes 10 worm variants and a 100,000 innocuous HTTP flows. There are a few flows that match the worm signature. These are false positives.



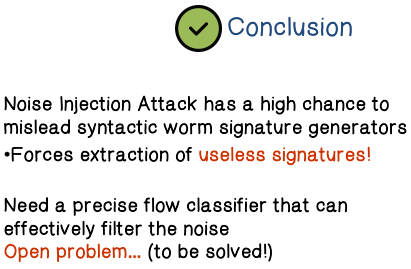
The test dataset includes 100 worm variants, and more than 20,000 normal HTTP requests. The attacker's data includes 300 score multiplayer strings extracted from normal traffic.



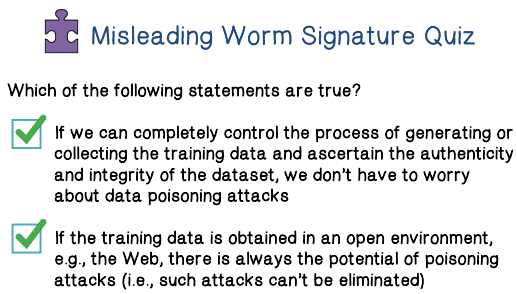
Here is an example of our experimental results. The y-axis here is false negative and the x-axis is false positive. We can see that, without noise injected by attacker, the original Polygraph performs very well, particularly, zero percent false negative and zero percent false positive. However, with noise injected by the attacker, Polygraph was not able to produce signatures that have low false positive and false negative at the same time.



We also evaluated how real worm flows can bypass these signatures. We send each real worm flow 20 times. We compared two scenarios, in the first, one fake anonymous flow is created per one real worm flow. In a second case, two fake anonymous flows are created per one real worm flow. As we can see, if we use more fake anonymous flows, then the attack success rate is higher. We also see that, if all three signatures are used to detect worms, the detection rate is higher. But, even if Polygraph uses all three types of signatures, the attacker can simply use more fake anonymous flows, so that the attack still has a high chance of succeeding.



To conclude, our case studies show that, attackers can inject noise to mislead worm signature generators. We have shown that useless worm signatures were produced when the attacker injected noise into the training data. To mitigate the noise injection attack, we need a precise flow classifier, and this remains an open problem.



Now, let us do a quiz. Which of the following statements are true?

The first statement, if we can completely control the process of generating or collecting the training data and ascertain the authenticity and integrity of the data set, we do not have to worry about data poisoning attacks, this is of course true. The second statement, if the training data is obtained in an open environment, such as the Web, there is always the potential of poisoning attacks, that is, such attacks cannot be eliminated, this is also true because in an open environment, the attacker can inject that data into the environment.