LEMNA: Explaining Deep Learning based Security Applications

Wenbo Guo[1,2], Dongliang Mu[5,1], Jun Xu[4,1], Purui Su[6], Gang Wang[3], Xinyu Xing[1,2]
[1]The Pennsylvania State University, [2]JD Security Research Center, [3]Virginia Tech,
[4]Stevens Institute of Technolog, [5]Nanjing University, [6]Chinese Academy of Sciences
{wzg13,dzm77,xxing}@ist.psu.edu,jxu69@stevens.edu,purui@iscas.ac.cn,gangwang@vt.edu

ABSTRACT

While deep learning has shown a great potential in various domains, the lack of transparency has limited its application in security or safety-critical areas. Existing research has attempted to develop *explanation techniques* to provide interpretable explanations for each classification decision. Unfortunately, current methods are optimized for non-security tasks (*e.g.*, image analysis). Their key assumptions are often violated in security applications, leading to a poor explanation fidelity.

In this paper, we propose LEMNA, a high-fidelity explanation method dedicated for security applications. Given an input data sample, LEMNA generates a small set of interpretable features to explain how the input sample is classified. The core idea is to approximate a local area of the complex deep learning decision boundary using a simple interpretable model. The local interpretable model is specially designed to (1) handle feature dependency to better work with security applications (*e.g.*, binary code analysis); and (2) handle nonlinear local boundaries to boost explanation fidelity. We evaluate our system using two popular deep learning applications in security (a malware classifier, and a function start detector for binary reverse-engineering). Extensive evaluations show that LEMNA's explanation has a much higher fidelity level compared to existing methods. In addition, we demonstrate practical use cases of LEMNA to help machine learning developers to validate model behavior, troubleshoot classification errors, and automatically patch the errors of the target models.

CCS CONCEPTS

• **Security and privacy → Software reverse engineering**;

KEYWORDS

Explainable AI, Binary Analysis, Deep Recurrent Neural Networks

ACM Reference Format: Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, Xinyu Xing. 2018. LEMNA: Explaining Deep Learning based Security Applications. In CCS '18: 2018 ACM SIGSAC Conference on Computer & Communications Security, Oct. 15–19, 2018, Toronto, ON, Canada. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3243734.3243792

1 INTRODUCTION

In recent years, Deep Neural Networks have shown a great potential to build security applications. So far, researchers have successfully applied deep neural networks to train classifiers for malware classification [2, 16, 21, 48, 68], binary reverse-engineering [15, 52, 71] and network intrusion detection [24, 62], which all achieved an exceptionally high accuracy.

While intrigued by the high-accuracy, security practitioners are concerned about *the lack of transparency* of the deep learning models and thus hesitated to widely adopt deep learning classifiers in security and safety-critical areas. More specifically, deep neural networks could easily contain hundreds of thousands or even millions of neurons. This network, once trained with massive datasets, can provide a high classification accuracy. However, the high complexity of the network also leads to a low "interpretability" of the model. It is very difficult to understand how deep neural networks make certain decisions. The lack of transparency creates key barriers to establishing trusts to the model or effectively troubleshooting classification errors.

To improve the transparency of deep neural networks, researchers start to work on *explanation methods* to interpret the classification results. Most existing works focus on non-security applications such as image analysis or natural language processing (NLP). Figure 1a shows an example. Given an input image, the explanation method explains the classification result by pinpointing the most impactful features to the final decision. Common approaches involve running forward propagation [17, 19, 32, 76] or backward propagation [3, 50, 53] in the network to infer important features. More advanced methods [34, 45] produce explanations under a "blackbox" setting where no knowledge of classifier details is available. The basic idea is to approximate the *local* decision boundary using a linear model to infer the important features.

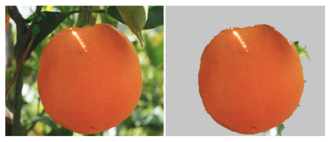
Unfortunately, existing explanation methods are not directly applicable to security applications. First, most existing methods are designed for image analysis, which prefers using Convolutional Neural Networks (CNN). However, CNN model is not very popular in security domains. Security applications such as binary reverse-engineering and malware analysis either have a high-level feature dependency (*e.g*, binary code sequences), or require high scalability. As a result, Recurrent Neural Networks (RNN) or Multilayer Perceptron Model (MLP) are more widely used [15, 21, 52, 68]. So far, there is no explanation method working well on RNN. Second, existing methods still suffer from a low explanation fidelity, as validated by our experiments in §5. This might be acceptable for image analysis, but can cause serious troubles in security applications. For example, in Figure 1a, the highlighted pixels are not entirely accurate (in particular at the edge areas) but are sufficient to provide an intuitive understanding. However, for security applications such as binary analysis, incorrectly highlighting one byte of code may lead to serious misunderstandings or interpretation errors.

Our Designs. In this paper, we seek to develop a novel, high-fidelity explanation method dedicated for *security applications*. Our method works under a black-box setting and introduces specialized designs to address the above challenges. Given an input data instance $\mathbf{x}$ and a classifier such as an RNN, our method aims to identify a small set of features that have key contributions to the classification of $\mathbf{x}$. This is done by generating a local approximation of the target classifier's decision boundary near $\mathbf{x}$. To significantly improve the fidelity of the approximation, our method no longer assumes the local detection boundary is linear, nor does it assume the features are independent. These are two key assumptions made by existing models [34, 45] which are often violated in security applications, causing a poor explanation fidelity. Instead, we introduce a new approach to approximate the non-linear local boundaries based on a *mixture regression model* [27] enhanced by *fused lasso* [64].

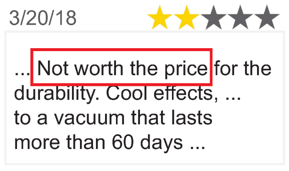
Our design is based on two key insights. First, a mixture regression model, in theory, can approximate both linear and non-linear decision boundaries given enough data [35]. This gives us the flexibility to optimize the local approximation for a non-linear boundary and avoid big fitting errors. Second, "fused lasso" is a penalty term commonly used for capturing feature dependency. By adding fused lasso to the learning process, the mixture regression model can take features as a group and thus capture the dependency between adjacent features. In this way, our method produces high-fidelity explanation results by simultaneously preserving the local non-linearity and feature dependency of the deep learning model. For convenience, we refer to our method as "Local Explanation Method using Nonlinear Approximation" or LEMNA.

Evaluations. To demonstrate the effectiveness of our explanation model, we apply LEMNA to two promising security applications: classifying PDF malware [55], and detecting the function start to reverse-engineer binary code [52]. The classifiers are trained on 10,000 PDF files and 2,200 binaries respectively, and both achieve an accuracy of 98.6% or higher. We apply LEMNA to explain their classification results and develop a series of *fidelity metrics* to assess the correctness of the explanations. The fidelity metrics are computed either by directly comparing the approximated detection boundary with the real one, or running end-to-end feature tests. The results show that LEMNA significantly outperforms existing methods across all different classifiers and application settings.

Going beyond the effectiveness assessment, we demonstrate how security analysts and machine learning developers can benefit from the explanation results. First, we show that LEMNA could help to establish trusts by explaining how classifiers make the correct decisions. In particular, for both binary and malware analyses, we demonstrate the classifiers have successfully learned a number of well-known heuristics and "golden rules" in the respective domain. Second, we illustrate that LEMNA could extract "new knowledge" from classifiers. These new heuristics are difficult to be manually summarized in a direct way, but make intuitive sense to domain experts once they are extracted by LEMNA. Finally, with LEMNA's capability, an analyst could explain why the classifiers produce errors. This allows the analyst to automatically generate targeted patches



(a) Image classification.



3/20/18 ★★★★★

... Not worth the price for the durability. Cool effects, ... to a vacuum that lasts more than 60 days ...

(b) Sentiment analysis.

Figure 1: Examples of machine learning explanation: (a) the image is classified as an "orange" due to the highlighted pixels; (b) The sentence is classified as "negative sentiment" due to the highlighted keywords.

by augmenting training samples for each of the explainable errors, and improve the classifier performance via targeted re-training.

Contributions. Our paper makes three key contributions.

- We design and develop LEMNA, a specialized explanation method for deep learning based security applications. Using a mixture regression model enhanced by fused lasso, LEMNA generates high-fidelity explanation results for a range of deep learning models including RNN.
- We evaluate LEMNA using two popular security applications, including PDF malware classification and function start detection in binary reverse-engineering. We propose a series of "fidelity" metrics to quantify the accuracy of the explanation results. Our experiments show that LEMNA outperforms existing explanation methods by a significant margin.
- We demonstrate the practical applications of the explanation method. For both binary analysis and malware detection, LEMNA sheds lights on why the classifier makes correct and incorrect decisions. We present a simple method to automatically convert the insights into actionable steps to patch the targeted errors of the classifiers.

To the best our knowledge, this is the first explanation system specially customized for security applications and RNN. Our work is only the initial step towards improving the model transparency for more effective testing and debugging of deep learning models. By making the decision-making process interpretable, our efforts can make a positive contribution to building reliable deep learning systems for critical applications.

2 EXPLAINABLE MACHINE LEARNING

In this section, we start with introducing the background of explainable machine learning, and then discuss existing explanation techniques. Following that, in Section §3, we introduce key security applications using deep learning models and discuss why existing explanation techniques are not applicable to security applications.

2.1 Problem Definition

Explainable machine learning seeks to provide interpretable explanations for the classification results. More specifically, given an input instance $\mathbf{x}$ and a classifier C, the classifier will assign a label y for $\mathbf{x}$ during the testing time. Explanation techniques then aim to illustrate why instance $\mathbf{x}$ is classified as y. This often involves identifying a set of important features that make key contributions to the classification process (or result). If the selected features are interpretable to human analysts, then these features can offer an "explanation". Figure 1 shows examples for image classification and sentiment analysis. The classifier decision can be explained by selected features (*e.g.*, highlighted pixels and keywords).

In this paper, we focus on the *deep neural networks* to develop explanation methods for security applications. Up to the present, most existing explanation methods are designed for image analysis or NLP. We categorize them into "whitebox" and "blackbox" methods and describe how they work.

2.2 Whitebox Explanation Methods

Most existing explanation techniques work under the *whitebox* setting where the model architecture, parameters, and training data are known. These techniques are also referred as *Deep Explanation Methods* and mainly designed for CNN. They leverage two major strategies to infer feature importance: (1) forward propagation based input or structure occlusion; and (2) gradient-based backpropagation. We discuss those techniques in the following.

Forward Propagation based Methods. Given an input sample, the key idea is to *perturb* the input (or hidden network layers) and observe the corresponding changes. The intuition behind is that perturbing important features is more likely to cause major changes to the network structure and the classification output. Existing methods either nullify a subset of features or removing intermediate parts of the network [17, 32, 74, 76]. A recent work [19] extends this idea to detecting adversarial examples (*i.e.*, malicious inputs aiming to cause classification errors).

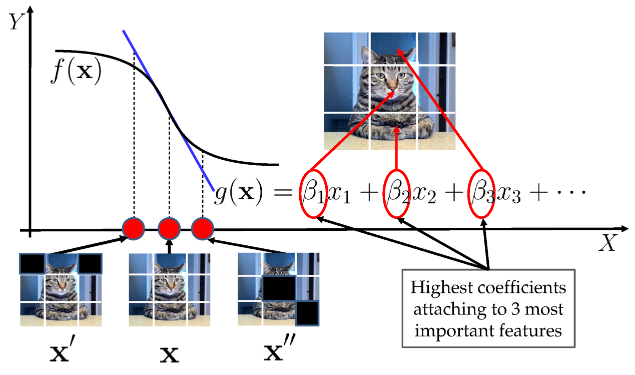
Backward Propagation based Methods. Back-propagation based methods leverage the gradients of the deep neural network to infer feature importance. The gradients can be the partial derivatives of classifier output with respect to the input or hidden layers. By propagating the output back to the input, these methods directly calculate the weights of input features. For image classifiers, the basic method is to compute a feature "saliency map" using the gradients of output with respect to the input pixels in images [54, 57] or video frames [18]. Later works improve this idea by applying saliency map layer by layer [3] or mapping groups of pixels [50].

Backward propagation based methods face the challenge of "zero gradient". Inside a neural network, the activation functions often have saturated parts, and the corresponding gradients will become zero. Zero gradients make it difficult (if not impossible) for the "saliency map" to back-track the important features. Recent works [53, 59] attempted to address this problem through approximation. However, this sacrifices the fidelity of the explanation [34].

2.3 Blackbox Explanation Methods

Blackbox explanation methods require no knowledge about the classifier internals such as network architecture and parameters. Instead, they treat the classifier as a "blackbox" and analyze it by sending inputs and observing the outputs (*i.e., Model Induction Methods*).

The most representative system in this category is LIME [45]. Given an input $\mathbf{x}$ (*e.g.*, an image), LIME systematically perturbs $\mathbf{x}$ to obtain a set of artificial images from the nearby areas of $\mathbf{x}$ in the feature space (see $\mathbf{x}'$ and $\mathbf{x}''$ in Figure 2). Then, LIME feeds the artificial images to the target classifier $f(\mathbf{x})$ to obtain labels, and uses the labeled data to fit a linear regression model $g(\mathbf{x})$. This $g(\mathbf{x})$ aims to approximate the small part of $f(\mathbf{x})$ near the input image in the feature space. LIME assumes that the *local area of the classification boundary near the input instance is linear*, and thus it is reasonable to use a linear regression model to locally represent the classification decision made by $f(\mathbf{x})$. Linear regression is self-explanatory, and thus LIME can pinpoint important features based on the regression coefficients. A recent work SHAP [34] tries to extend LIME by adding weights to the artificially generated data samples. Other works propose to use other linear models (*e.g.*, decision tree [6] and decision set [31]) to incrementally approximate the target detection boundaries.



Y
$f(\mathbf{x})$
$g(\mathbf{x}) = \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \cdots$
X
$\mathbf{x}'$
$\mathbf{x}$
$\mathbf{x}''$
Highest coefficients attaching to 3 most important features

Figure 2: Illustrating how a Blackbox Explanation Method works. The key idea is to use a local linear model (g, the blue straight line) to approximate the detection boundary f near the input instance x. Then the linear model can help to select the key contributing features to classifying x.

As a side note, we want to clarify that machine learning explanation is *completely different* from feature selection methods such as Principal Component Analysis (PCA) [26], Sparse Coding [39] or Chi-square Statistics [49]. Explanation methods aim to identify the key features of *a specific input instance* $\mathbf{x}$ to specifically explain how an instance $\mathbf{x}$ is classified. On the other hand, feature selection methods such as PCA are typically applied before training on the whole training data to reduce the feature dimension (to speed up the training or reduce overfitting), which cannot explain how a specific classification decision is made.

3 EXPLAINING SECURITY APPLICATIONS

While deep learning has shown a great potential to build security applications, the corresponding explanation methods are largely falling behind. As a result, the lack of transparency reduces the trust. First, security practitioners may not trust the deep learning model if they don't understand how critical decisions are made. Second, if security practitioners cannot troubleshoot classification errors (*e.g.*, errors introduced by biased training data), the concern is that these errors may be amplified later in practice. In the following, we introduce two key security applications where deep learning has recently achieved success. Then we discuss why existing explanation methods are not applicable to the security applications.

Explanation Method	Support RNN/MLP	Local Non-linear	Support Blackbox	Representative Works
Whitebox method (forward)	◐	○	◐	`Occlusion` [17, 32, 74, 76], `AI`^{2} [19],
Whitebox method (backword)	◐	○	○	`Saliency Map` [3, 54, 57], `Grad-carm` [50], `DeepLIFT` [53]
Blackbox method	◐	○	●	`LIME` [45], `SHAP` [34], `Interpretable Decision Set` [31]
Our method `LEMNA`	●	●	●	`LEMNA`

Table 1: Design space of explainable machine learning for security applications (●=true; ○=false; ◐=partially true).

3.1 Deep Learning in Security Applications

In this paper, we focus on two important classes of security applications: *binary reverse engineering* and *malware classification*.

Binary Reverse-Engineering. The applications of deep learning in binary analysis include identifying function boundaries [52], pinpointing the function type signatures [15] and tracking down similar binary code [71]. More specifically, using a bi-directional RNN, Shin *et al.* improve the function boundary identification and achieve a nearly perfect performance [52]. Chua *et al.* also use RNN to accurately track down the arguments and types of functions in binaries [15]. More recently, Xu *et al.* employ an MLP to encode a control flow graph to pinpoint vulnerable code fragments [71].

Malware Classification. Existing works mainly use MLP models for large-scale malware classifications. For example, researchers have trained MLP to detect malware at the binary code level [48] and classify Android malware [2, 21]. More recently, Wang *et al.* [68] propose an adversarial resistant neural network for detecting malware based on audit logs [7].

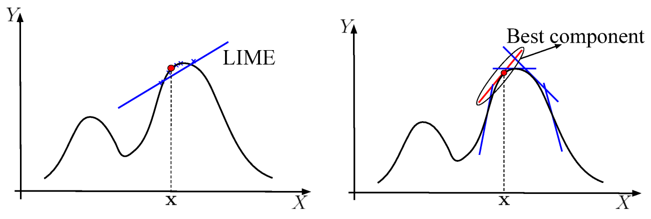
A key observation is that RNN and MLP are more widely adopted by these security applications compared to CNN. The reason is that RNN is designed to handle *sequential data*, which performs exceptionally well in processing the long sequences of binary code. Particularly, `Bi-directional RNN` can capture the bi-directional dependencies in the input sequences between each hex [52]. For malware classification, MLP is widely used for its high *efficiency*. On the other hand, CNN performs well on images since it can take advantage of the grouping effect of features on the 2D images [30]. These security applications do not have such "matrix-like" data structures to benefit from using CNN.

3.2 Why Not Existing Explanation Methods

There are key challenges to directly apply existing explanation methods to the security applications. In Table 1, we summarize the desired properties, and why existing methods fail to deliver them.

Supporting RNN and MLP. There is a clear mismatch between the model choices of the above security applications and existing explanation methods. Most existing explanation methods are designed for CNN to work with image classifiers. However, as mentioned in §3.1, security applications of our interests primarily adopt RNN or MLP. Due to model mismatches, existing explanation methods are not quite applicable. For example, the back-propagation methods including "saliency map" [3, 18, 54, 57] and activation difference propagation [53] require special operations on the convolutional layers and pooling layers of CNN, which do not exist in RNN or MLP [1].

[1][15] presents some case studies using saliency map to explain RNN, but is forced to ignore the feature dependency of RNN, leading to a low explanation fidelity.



(a) Linear regression model. **(b) Mixture regression model.**

Figure 3: Approximating a locally non-linear decision boundary. The linear regression model (a) can easily make mistakes; Our mixture regression model (b) achieves a more accurate approximation.

Blackbox methods such as LIME do not support RNN well either (validated by our experiments later). Methods like LIME assume features are *independent*, but this assumption is violated by RNN which explicitly models the dependencies of sequential data.

Supporting Locally Non-linear Decision Boundary. Most existing methods (*e.g.,* LIME) assume the local linearity of the decision boundary. However, when the local decision boundary is non-linear, which is true for most complex networks, those explanation methods would produce serious errors. Figure 3a shows an example where the decision boundary around **x** is highly non-linear. In other words, the linear part is heavily restricted to a very small region. The typical sampling methods can easily hit the artificial data points beyond the linear region, making it difficult for a linear model to approximate the decision boundary near **x**. Later in our experiments (§ 5), we confirm that a simple linear approximation will significantly degrade the explanation fidelity.

Supporting Blackbox Setting. Although both whitebox and blackbox methods have their application scenarios, blackbox methods are still more desirable for security applications. Noticeably, it is not uncommon for people to use pre-trained models (*e.g.,* "`Bi-directional RNN`" [52], "`prefix tree`" in Dyninst [5]) where the detailed network architecture, parameters or training data are not all available. Even though a few forward propagation methods can be forced to work under a blackbox setting (by giving up the observations of intermediate layers), it would inevitably lead to performance degradation.

Summary. In this paper, we aim to bridge the gaps by developing dedicated explanation methods for security applications. Our method aims to work under a blackbox setting and efficiently support popular deep learning models such as RNN, MLP, and CNN. More importantly, the method need to achieve a much higher explanation fidelity to support security applications.

4 OUR EXPLANATION METHOD

To achieve the above goals, we design and develop LEMNA. At the high-level, we treat a target deep learning classifier as a blackbox and derive explanation through model approximation. In order to provide a high fidelity explanation, LEMNA needs to take a very different design path from existing methods. First, we introduce *fused lasso* [64] to handle the feature dependency problems that are often encountered in security applications and RNN (*e.g.*, time series analysis, binary code sequence analysis). Then, we integrate fused lasso into a *mixture regression model* [28] to approximate locally non-linear decision boundaries to support complex security applications. In the following, we first discuss the insights behind the design choices of using fused lasso and mixture regression model. Then, we describe the technical details to integrate them into a single model to handle feature dependencies and locally nonlinearity at the same time. Finally, we introduce additional steps to utilize LEMNA to derive high-fidelity explanations.

4.1 Insights behind Our Designs

Fused Lasso. Fused lasso is a penalty term commonly used for capturing feature dependencies, and is useful to handle the dependent features in deep learning models such as RNN. At the high-level, "fused lasso" forces LEMNA to group relevant/adjacent features together to generate meaningful explanations. Below, we introduce the technical details of this intuition.

To learn a model from a set of data samples, a machine learning algorithm needs to minimize a loss function $L(f(\mathbf{x}), y)$ that defines the dissimilarity between the true label and the predicted label by the model. For example, to learn a linear regression model $f(\mathbf{x}) = \boldsymbol{\beta}\mathbf{x} + \epsilon$ from a data set with N samples, a learning algorithm needs to minimize the following equation with respect to the parameter $\boldsymbol{\beta}$ using Maximum Likelihood Estimation (MLE) [38].

$$L(f(\mathbf{x}), y) = \sum_{i=1}^{N} \|\boldsymbol{\beta}\mathbf{x}_i - y_i\| \,. \tag{1}$$

Here, $\mathbf{x}_i$ is a training sample, represented by an M-dimensionality feature vector $(x_1, x_2, \cdots, x_M)^T$. The label of $\mathbf{x}_i$ is denoted as y_i. The vector $\boldsymbol{\beta} = (\beta_1, \beta_2, \cdots \beta_M)$ contains the coefficients of the linear model. $\|\cdot\|$ is the $L2$-norm measuring the dissimilarity between the model prediction and the true label.

Fused lasso is a penalty term that can be introduced into any loss functions used by a learning algorithm. Take linear regression for example. Fused lasso manifests as a constraint imposed upon coefficients, *i.e.,*

$$\begin{aligned} L(f(\mathbf{x}), y) &= \sum_{i=1}^{N} \|\boldsymbol{\beta}\mathbf{x}_i - y_i\| \,, \\ \text{subject to} &\sum_{j=2}^{M} \|\beta_j - \beta_{j-1}\| \leq S \,. \end{aligned} \tag{2}$$

Fused lasso restricts the dissimilarity of coefficients assigned to adjacent features within a small threshold S (*i.e.,* a hyper-parameter) when a learning algorithm minimizes the loss function. As a result, the penalty term forces a learning algorithm to assign equal weights to the adjacent features. Intuitively, this can be interpreted as forcing a learning algorithm to take features as groups and then learn a target model based on feature groups.

Security applications, such as time series analysis and code sequence analysis, often need to explicitly model the feature dependency of sequential data using RNN. The resulting classifier makes a classification decision based on the co-occurrence of features. If we use a standard linear regression model (*e.g.*, LIME) to derive an explanation, we cannot approximate a local decision boundary correctly. This is because a linear regression model cannot capture feature dependency and treat them independently.

By introducing fused lasso in the process of approximating local decision boundary, we expect the resulting linear model to have the following form:

$$f(\mathbf{x}) = \beta_1 x_1 + \beta_2(x_2 + x_3) + \beta_3(x_4 + x_5) + \cdots + \beta_k x_M \,, \tag{3}$$

where features are grouped together and thus important features are likely to be selected as a group or multiple groups. Explicitly modeling this process in LEMNA helps to derive a more accurate explanation, particularly for the decision made by an RNN. We further explain this idea using an example of sentiment analysis in Figure 1b. With the help of fused lasso, a regression model would collectively consider adjacent features (*e.g.*, words next to each other in a sentence). When deriving the explanations, our model does not simply yield a single word "*not*"[2], but can accurately capture the phrase "*not worth the price*" as the explanation for the sentiment analysis result.

Mixture Regression Model. A mixture regression model allows us to approximate locally nonlinear decision boundaries more accurately. As shown in Figure 3b, a mixture regression model is a combination of multiple linear regression models, which makes it more expressive to perform the approximation:

$$y = \sum_{k=1}^{K} \pi_k(\boldsymbol{\beta}_k \mathbf{x} + \epsilon_k) \,, \tag{4}$$

where K is a hyper-parameter indicating the total number of linear components combined in the mixture model; π_k indicates the weight assigned to that corresponding component.

Given sufficient data samples, whether the classifier has a linear or non-linear decision boundary, the mixture regression model can nearly perfectly approximate the decision boundary (using a finite set of linear models) [35]. As such, in the context of deep learning explanation, the mixture regression model can help avoid the aforementioned non-linearity issues and derive more accurate explanations.

To illustrate this idea, we use the example in Figure 3. As shown in Figure 3a, a standard linear approximation cannot guarantee the data sampled around the input $\mathbf{x}$ still remain in the locally linear region. This can easily lead to imprecise approximation and low-fidelity explanations. Our method in Figure 3b approximates the local decision boundary with a polygon boundary, in which each blue line represents an independent linear regression model. The best linear model for producing the explanation should be the red line passing through the data point $\mathbf{x}$. In this way, the approximation process can yield an optimal linear regression model for pinpointing important features as the explanation.

[2]In sentiment analysis, "not" does not always carry negative sentiment, *e.g.*, "not bad".

4.2 Model Development

Next, we convert these design insights into a functional explanation system. We introduce the technical steps to integrate fused lasso in the learning process of a mixture regression model so that we can handle feature dependency and decision boundary non-linearity at the same time. Technically speaking, we need to derive a mixture regression model by minimizing the following equation

$$\begin{aligned} L(f(\mathbf{x}), y) &= \sum_{i=1}^{N} \|f(\mathbf{x}_i) - y_i\|\,, \\ \text{subject to} &\sum_{j=2}^{M} \|\beta_{kj} - \beta_{k(j-1)}\| \leq S\,,\ k = 1, \ldots, K\,. \end{aligned} \tag{5}$$

where $f(\cdot)$ represents the mixture regression model shown in Equation (4), and β_{kj} indicates the parameter in the k^{th} linear regression model tied to its j^{th} feature.

Different from a standard linear regression, our optimization objective is intractable and we cannot simply utilize MLE to perform minimization. To effectively estimate parameters for the mixture regression model, we utilize an alternative approach.

First, we represent the mixture regression model in the form of probability distributions

$$y_i \sim \sum_{k=1}^{K} \pi_k \mathcal{N}(\boldsymbol{\beta}_k \mathbf{x}_i, \sigma_k^2)\,. \tag{6}$$

Then, we treat $\pi_{1:K}$, $\boldsymbol{\beta}_{1:K}$ and $\sigma^2_{1:K}$ as parameters[3]. By taking a guess at these parameters, we initialize their values and thus perform parameter estimation by using Expectation Maximization (EM) [37], an algorithm which estimates parameters by repeatedly performing two steps – E-Step and M-Step. In the following, we briefly describe how this EM algorithm is used in our problem. More details can be found in Appendix-A.

In the Equation (6), y_i follows a distribution which combines K Gaussian distributions, and each of these distributions has the mean $\boldsymbol{\beta}_k \mathbf{x}_i$ and the variance σ_k^2. In the E-Step, we assign each of the data samples to one of the Gaussian distributions by following the standard procedure applied in learning an ordinary mixture regression model. Based on the data samples assigned in the previous E-Step, we then re-compute the parameters $\pi_{1:K}$, $\boldsymbol{\beta}_{1:K}$ and $\sigma^2_{1:K}$. For the parameters $\pi_{1:K}$ and $\sigma^2_{1:K}$, the re-computation still follows the standard procedure used by ordinary mixture model learning. But, for each parameter in $\boldsymbol{\beta}_{1:K}$, re-computation follows a customized procedure. That is to compute $\boldsymbol{\beta}_k$ by minimizing the following equation with respect to $\boldsymbol{\beta}_k$

$$\begin{aligned} L(x, y) &= \sum_{i=1}^{N_k} \|\boldsymbol{\beta}_k \mathbf{x}_i - y_i\|\,, \\ \text{subject to} &\sum_{j=2}^{M} \|\beta_{kj} - \beta_{k(j-1)}\| \leq S\,, \end{aligned} \tag{7}$$

where N_k refers to the number of samples assigned to the k^{th} component. Here, the reason behind this re-computation customization is that fused lasso has to be imposed to parameters $\boldsymbol{\beta}_{1:K}$ in order to grant a mixture regression model the ability to handle feature dependency. As we can observe, the equation above shares the same form with that shown in Equation (2). Therefore, we can minimize the equation through MLE and thus compute the values for parameters $\boldsymbol{\beta}_{1:K}$.

Following the standard procedure of EM algorithm, we repeatedly perform the E-step and M-Step. Until stability is reached (*i.e.*, the Gaussian distributions do not vary much from the E-step to the M-step), we output the mixture regression model. Note that we convert $\sigma^2_{1:K}$ into the model parameter $\epsilon_{1:K}$ by following the standard approach applied in ordinary mixture model learning.

[3] $\pi_{1:K}$ indicates parameters $\pi_1, \cdots, \pi_K$. $\boldsymbol{\beta}_{1:K}$ represents parameters $\boldsymbol{\beta}_1, \cdots, \boldsymbol{\beta}_K$. $\sigma^2_{1:K}$ are the parameters $\sigma_1^2, \cdots, \sigma_1^K$, each of which describes the variance of the normal distribution that ϵ_k follows, *i.e.*, $\epsilon_k \sim \mathcal{N}(0, \sigma_k^2)$.

4.3 Applying the Model for Explanation

With the enhanced mixture regression model, we now discuss how to derive high-fidelity explanations for deep learning classifiers.

Approximating Local Decision Boundary. Given an input instance $\mathbf{x}$, the key to generate the explanation is to approximate the local decision boundary of the target classifier. The end product is an "interpretable" linear model that allows us to select a small set of top features as the explanation. To do so, we first synthesize a set of data samples locally (around $\mathbf{x}$) following the approach described in [45]. The idea is to randomly nullify a subset of features of $\mathbf{x}$.

Using the corpus of synthesized data samples, we then approximate the local decision boundary. There are two possible schemes: one is to train a single mixture regression model to perform multi-class classification; the other scheme is to train multiple mixture regression models, each of which performs binary classification. For efficiency considerations, we choose the second scheme and put more rigorous analysis to the Appendix-B.

Deriving Explanations. Given the input data instance $\mathbf{x}$, and its classification result y, we now can generate explanations as a small set of important features to $\mathbf{x}$'s classification. More specifically, we obtain a mixture regression model enhanced by fused lasso. From this mixture model, we then identify the linear component that has the best approximation of the local decision boundary. The weights (or coefficients) in the linear model can be used to rank features. A small set of top features is selected as the explanation result.

Note that LEMNA is designed to simultaneously handle non-linearity and feature dependency, but this does not mean that LEMNA cannot work on deep learning models using relatively independent features (*e.g.*, MLP or CNN). In fact, the design of LEMNA provides the flexibility to adjust the explanation method according to the target deep learning model. For example, by increasing the hyper-parameter S (which is a threshold for fused lasso), we can relax the constraint imposed upon parameter $\boldsymbol{\beta}_{1:K}$ and allow LEMNA to better handle less dependent features. In Section §5, we demonstrate the level of generalizability by applying LEMNA to security applications built on both RNN and MLP.

5 EVALUATION

In this section, we evaluate the effectiveness of our explanation method on two security applications: malware classification and binary reverse engineering. This current section focuses evaluating

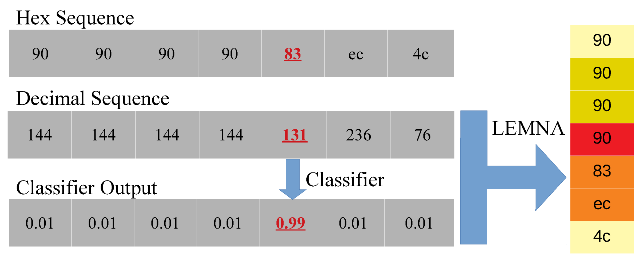


Figure 4: Applying LEMNA **to explain binary function start.** 83 **is the real function start, and** 0.99 **is the output probability of the RNN classifier. By sending the tuple** (hex-sequence, 83) **to** LEMNA**, our system explains the classification decision by color-coding the most important hex. Feature importance decreases from red to yellow.**

on the *accuracy* of the explanation through a series of *fidelity* metrics. In the next section (§6), we will present practical use cases of LEMNA to understand classifier behavior, troubleshoot classification errors, and patch the errors of the classifiers.

5.1 Experimental Setup

We apply LEMNA to two security applications: detecting the "function start" for reverse-engineering binary code using RNN, and classifying PDF malware based on MLP. Below, we introduce details about the two security applications, the implementation of LEMNA, and the comparison baselines.

Binary Reverse-Engineering. Binary code reverse-engineering, which transfers binary code to assembly code, is a crucial step in (1) examining and detecting malware [51], (2) hardening the security of software [75], and (3) generating security patches [56]. For years, binary analysis is primarily done manually by experienced security analysts. Recently, researchers show that well-trained RNN can help handle critical reverse-engineering steps such as *detecting the function start* [52], which can significantly save human efforts. Considering the importance of detecting function start (*i.e.*, all binary code reverse-engineering requires knowing the function start), we choose this application to test LEMNA.

We follow [52] to build a RNN based on a widely used dataset that contains 2200 binaries [5]. We compile these binaries under x86 architecture and gcc compiler with four different optimization levels O0, O1, O2, and O3 respectively. This produces 4 training datasets, one for each optimization level. Like [52], we use the bi-directional RNN and train 4 different classifiers.

Each binary in the dataset is presented as a sequence of hex code. As shown in Figure 4, we first transfer the hex code to their decimal values, and treat each element in the sequence as a feature. For training, each element in the sequence has a label of either "a function start" or "not a function start". As shown in Figure 4, suppose the original binary code is "90 90 90 83 ec 4c" and the function start is at "83", then the label vector is (0, 0, 0, 0, 1, 0, 0). We follow [52] to truncate very long binary sequences and set the maximum length to 200. Then we feed the sequences into the RNN. We used Keras [14] to train the model, with Theano [63] as a backend. We split the dataset randomly using 70% of the samples for training, and the rest 30% for testing.

Application	Binary Function Start				PDF Malware
	O0	O1	O2	O3	
Precision	99.99%	99.65%	98.57%	99.53%	99.12%
Recall	99.97%	99.49%	98.81%	99.06%	98.13%
Accuracy	99.99%	99.99%	99.99%	99.99%	98.64%

Table 2: Classification accuracy of the trained classifiers.

As shown in Table 2, the detection accuracy is extremely high, with a 98.57% or higher precision and recall for all cases. The results are comparable to those reported in [52]. The hyper-parameters of the RNNs can be found in the Appendix-C.

PDF Malware Classifier. We follow [21, 48] to construct a MLP-based malware classifier based on a widely used dataset (4999 malicious PDF files and 5000 benign files) [55]. We follow [55, 58] to extract 135 features for each file. The features were manually crafted by researchers based on the meta-data and the structure of the PDF, such as *number of object markers* and *number of javascript markers*. The full feature list can be found in the Mimicus [1]. We follow the standard method to transform the feature values into a binary representation [41] (*i.e.*, nonzero feature values are converted to 1), which helps avoid certain high-value features skewing the training process. Like before, we randomly select 70% of the datasets (malware and benign 1:1) as the training data, and use the remaining 30% as the testing data. As shown in Table 2, our precision and recall are both above 98.13%, which are similar to [55].

LEMNA **Implementation.** We treat the above RNN and MLP as the target classifiers to run LEMNA. Given an input instance, LEMNA approximates the target classifier and explain the classification result. "Explanations" are presented as the most important features for the given input. For the malware classifier, LEMNA outputs a small set of top features that explains why a file is (not) a malware. For the "function start" detector, an example is shown in Figure 4. Given an input hex sequence and the detected function start (*i.e.*, "83"), LEMNA marks out a small set of hex code in the sequence that has the biggest contribution. Here, "83" is the function start, and LEMNA points out that the hex code "90" before the function start is the most important reason of the detection.

LEMNA has 3 hyper-parameters that are configurable. First, to approximate the local decision boundary, we set to craft N data samples for the model fitting (see §4). The second and third parameters are the number of mixture components K, and the threshold of the fused lasso S. For binary function start detection, we set parameters as: N=500, K=6, S=1e − 4. For malware classification, we set parameters as: N=500, K=6, S=1e4. Note that the parameter S is set very differently because malware analysis features are relatively independent, while the binary analysis features have a high dependency level. We fix these parameters to run most of our experiments. Later, we have a dedicated section to perform sensitivity tests on the parameter settings (which shows LEMNA is not sensitive to these hyper-parameters).

LEMNA**'s Computational Costs.** The computational costs of LEMNA are relatively low. For both security applications, the time to generate the explanation for a given instance is about 10 seconds. This computation task further benefits from parallelization. For example, using a server with Intel Xeon CPU E5-2630, one Nvidia Tesla K40c

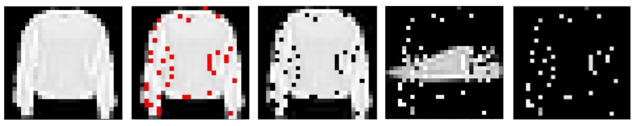


Figure 5: We use an image classifier as an toy example to explain the fidelity test. Figure 5a is the original input image ("sweater"). Figure 5b is the explanation produced by LEMNA where important features (pixels) are highlighted in red. Figure 5c–5e are three testing instances we generated to test the fidelity of the explanation.

GPU and 256G RAM, it takes about 2.5 hours to explain all 25, 040 binary testing sequences for O0 with 30 threads.

Comparison Baselines. We use two baselines for comparison. First, we use the state-of-the-art blackbox method LIME [45] as our comparison baseline. LIME [45] has been used to explain image classifiers and NLP applications. Its performance on security applications and RNN is not yet clear[4]. For a fair comparison, we also configure LIME with N=500 which is the number of artificial samples used to fit the linear regression model. Second, we use a *random* feature selection method as the baseline. Given an input, the *Random* method selects features randomly as the explanation for the classification result.

5.2 Fidelity Evaluation

To validate the correctness (fidelity) of the explanation, we conduct a two-stage experiment. In the first stage, we directly examine the accuracy of our local approximation with respect to the original decision boundary. This is likely to give an initial estimation of the explanation accuracy. In the second stage, we perform end-to-end evaluation on the explanation fidelity. We design *three fidelity tests* to show whether the selected features are indeed the main contributors to the classification results.

Evaluation 1: Local Approximation Accuracy. This metric is directly computed by comparing the approximated decision boundary and the original one. We measure Root Mean Square Error (RMSE): $RMSE = \sqrt{\frac{\sum_{i=1}^{n}(p_i - \hat{p}_i)}{n}}$, where p_i represents a single prediction obtained from a target deep learning classifier, $\hat{p}_i$ denotes the approximated prediction obtained from the explanation method, and n is the total number of testing data samples. More specifically, we start from a given classifier and a set of testing data samples. For each testing data sample $\mathbf{x}_i$, we first obtain a prediction probability p_i using the classifier. Then for $\mathbf{x}_i$, we follow Equation (6) to generate a regression model, which can produce an *estimated* prediction probability $\hat{p}_i$. After running these steps for all n testing samples, we obtain a prediction vector $\mathbf{P} = (p_1, p_2, ..., p_n)$ and the corresponding approximation vector $\hat{\mathbf{P}} = (\hat{p}_1, \hat{p}_2, ..., \hat{p}_n)$. Finally, we computer RMSE based on the two vectors. A lower RMSE means the approximated decision boundary ($\hat{\mathbf{P}}$) is closer to the true boundary ($\mathbf{P}$), indicating a higher fidelity of explanation.

Evaluation 2: End-to-end Fidelity Tests. To validate the correctness of the selected features, we design *three* end-to-end fidelity tests. To help readers to understand the testing process, we use "image classifier" as a toy example[5]. The procedure works in the same way for other classifiers. As shown in Figure 5, the image classifier is trained to classify "shoe" from "sweater". Figure 5a is the input image ($\mathbf{x}$) with the label as "sweater". In Figure 5b, the explanation method explains the reasons for the classification by highlighting important pixels (features) in red. We denote the selected features as $\mathbf{F_x}$. To test the fidelity of the explanation, we have three intuitions:

- If features $\mathbf{F_x}$ are accurately selected, then removing $\mathbf{F_x}$ from the input $\mathbf{x}$ will lead to classifying this image to a different label, *i.e.*, "shoe" (Figure 5c).
- If features $\mathbf{F_x}$ are accurately selected, then adding the feature values of $\mathbf{F_x}$ to an image of "shoe" is likely to lead to a misclassification, *i.e.*, classifying it as a "sweater" (Figure 5d).
- If features $\mathbf{F_x}$ are accurately selected, we can craft a synthetic images that only contains the features in $\mathbf{F_x}$, and this synthetic image is likely to be classified as "sweater" (Figure 5e).

Using these intuitions, we construct 3 different fidelity tests to validate the selected features. More formally, given an input instance $\mathbf{x}$ and its classification label y, LEMNA identifies a small set of important features ($\mathbf{F_x}$) as the "explanation". We then follow the steps below to generate 3 testing samples $\mathbf{t(x)}_1$, $\mathbf{t(x)}_2$ and $\mathbf{t(x)}_3$ for feature validation:

- **Feature Deduction Test:** we construct a sample $\mathbf{t(x)}_1$ by nullifying the selected features $\mathbf{F_x}$ from the instance $\mathbf{x}$.
- **Feature Augmentation Test:** we first select one random instance r from the opposite class (*i.e.*, as long as r's label is not y). Then we construct $\mathbf{t(x)}_2$ by replacing the feature values of the instance r with those of $\mathbf{F_x}$.
- **Synthetic Test:** we construct $\mathbf{t(x)}_3$ as a synthetic instance. We preserve the feature values of the selected features $\mathbf{F_x}$ while randomly assigning values for the remaining features.

The key variable in this experiment is the number of important features selected as the "explanation" (*i.e.*, $|\mathbf{F_x}|$). Intuitively, a larger $|\mathbf{F_x}|$ may yield a better explanation fidelity, but hurts the interpretability of results. We want to keep $|\mathbf{F_x}|$ small so that human analysts are able to comprehend.

For each classifier, we run the fidelity tests on the *testing* dataset (30% of the whole data). Given an instance $\mathbf{x}$ in the testing dataset, we generate 3 samples, one for each fidelity test. We feed the 3 samples into the classifier, and examine the *positive classification rate* (PCR). PCR measures the ratio of the samples still classified as $\mathbf{x}$'s original label. Note that "positive" here does not mean "malware" or "function start". It simply means the new sample is still classified as the $\mathbf{x}$'s original label. If the feature selection is accurate, we expect the *feature deduction* samples return a low PCR, the *feature augmentation* samples return a high PCR, and the *synthetic testing* samples return a high PCR.

5.3 Experimental Results

Our experiments show that LEMNA outperforms LIME and the random baseline by a significant margin across *all* fidelity metrics.

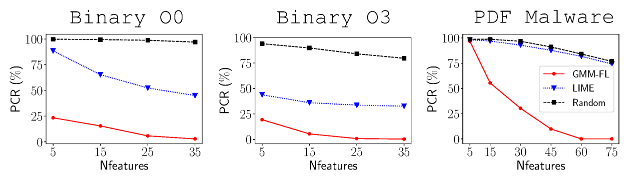
Local Approximation Accuracy. As shown in Table 3, LEMNA has a RMSE *an order of magnitude* smaller than that of LIME. This

[4]We have tested SHAP [34], which is an extension of LIME. We find that SHAP is very slow and its performance is worse than LIME for our applications.

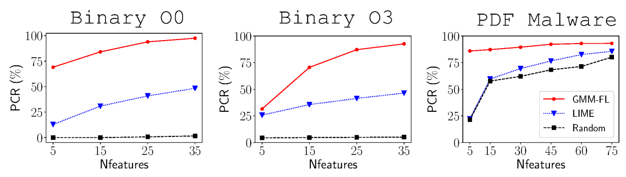
[5]The image is selected from the Fashion-mnist dataset [69].

Method	Binary Function Start				PDF malware
	00	01	02	03	
LIME	0.1784	0.1532	0.1527	0.1750	0.1178
LEMNA	0.0102	0.0196	0.0113	0.0110	0.0264

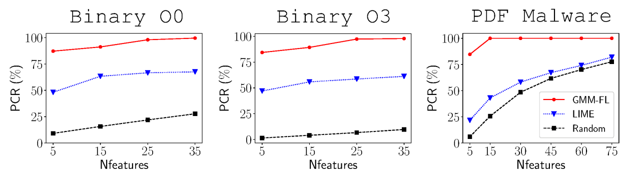
Table 3: The Root Mean Square Error (RMSE) of local approximation. LEMNA is more accurate than LIME.



(a) Feature Deduction test. A lower PCR reflects a higher explanation fidelity.



(b) Feature Augmentation test. A higher PCR reflects a higher explanation fidelity.



(c) Synthetic test. A higher PCR reflects a higher explanation fidelity.

Figure 6: Fidelity test results. y-axis denotes the positive classification rate PCR and y-axis denote the number of selected features NFeature by the explanation method. Due to the space limit, the results of Binary-O1 and O2 are shown in Appendix-D.

observation holds for both the malware classifier and the function start detection. The best performing result of LIME has a RMSE of 0.1532, which is still almost 10 times higher than the worse performing result of LEMNA (0.0196). This result confirms that our mixture regression model is able to build a much more accurate approximation than a simple linear model. Note that this metric is not applicable to the random baseline since the random baseline does not construct a decision boundary.

Fidelity Tests. Figure 6a shows the results from *feature deduction test*. Recall feature deduction test is to remove important features from the input instances. A lower PCR indicates that selected features are more important to the classification decision. By only nullifying the top 5 features produced by LEMNA, the function start detector drops the PCR to 25% or lower. Considering the extremely high accuracy of the classifier (99.5%+, see Table 2), this drastic decrease of PCR indicates the small set of features are highly important to the classification. Note that the feature nullification is consider minor since the top 5 features only count of 2.5% of the

(N, K, S)	RMSE	Deduc. test	Augme. test	Synthet. test
(500, 6, 1e-4)	0.0102	5.79%	93.94%	98.04%
(300, 6, 1e-4)	0.0118	5.94%	94.32%	98.18%
(500, 4, 1e-4)	0.0105	5.80%	93.71%	97.89%
(500, 6, 1e-3)	0.0114	5.83%	93.21%	97.73%

Table 4: Hyper-parameters sensitivity testing results.

200 total features in the input sequence. If we nullify the top 35 features, the PCR is dropped to almost 0.

Figure 6b shows the results of the *feature augmentation test*. Recall that feature augmentation is to add the selected features of input **x** to an instance of the opposite class, expecting the classifier to produce a label of **x**. A higher PCR indicates the selected features are more important to **x**. The results are relatively consistent with the previous test: (1) adding a small number of top features can flip the label of the instance in the opposite class; (2) our method outperforms both baselines by a big margin. Noticeably, for the PDF malware classifier, by replacing the top 5 features, 75% of the testing cases flip their labels.

Figure 6c shows a similar trend for the *synthetic test*. Using our selected features from a given **x**, the synthetic instances are more likely to be labeled as **x**'s label. Using only 5 top features, the synthetic instances have a 85%–90% of the chance to take **x**'s label, indicating that the core patterns have been successfully captured.

Across all three tests, our LEMNA outperforms LIME and the random baseline by a big margin. Interestingly, for the malware classifier, LIME performs as poor as random feature selection. This is because the feature vectors are sparse, which hurts the "smoothness" of the decision boundary. LIME has a hard time to accurately approximate the non-smooth boundary, which again validates our design intuition. Our system is more suitable for security applications, considering that security applications require a much higher explanation precision compared to image analysis tasks.

Sensitivity of Hyper-parameters. Finally, we test how our results would change if the parameters are set differently. We tested a large number of parameter configurations, and find that our conclusions remain consistent. Due to the space limit, we summarize key results in Table 4. The three hyper-parameters are the "number of crafted data samples" for model fitting (N), the "total number of mixture components" (K), and the "threshold for fused lasso" (S). Table 4 presents the results of the binary function start detector on the 00 dataset. We show 4 groups of configurations where we change one parameter at a time. For the fidelity tests, we fix the number of selected features as 25 to calculate the PCR. The results confirm that changing the hyper-parameters do not significantly influence the performance of LEMNA.

6 APPLICATIONS OF ML EXPLANATION

So far, we have validated the fidelity of the explanation results. In this section, we present practical applications of LEMNA. We use case studies to show how the explanation results can help security analysts to 1) establish trusts to the trained classifiers, 2) troubleshoot classification errors, 3) and systematically *patch* the targeted errors. In the following, we primarily focus on the binary reverse-engineering application since this application domain of deep learning is relatively new and not well-understood. We have

Cases	ID	Opt.-level	F. Start	Explanation	Assembling code
C.W.H.	1	O0	*55*	5b 5d c3 55 89 e5	`pop ebx; pop ebp; ret;` **push ebp**; `mov ebp, esp`
	2	O1	*53*	5b 90 c3 53 83 ec 18	`pop ebx; nop; ret;` **push ebx;** `sub esp,0x18`
	3	O2	*89*	8d b4 26 00 00 00 00 89 c1 8b 40 0c	`lea esi, [esi+eiz*1+0];` **mov ecx, eax**
	4	O3	*56*	90 90 90 90 56 53	`nop; nop; nop; nop;` **push esi**; `push ebx`
D.N.K.	5	O0	*31*	e9 00 f9 ff ff 31 ed 5e	`jmp 0xfffff900;` **xor ebp, ebp**; `pop esi`
	6	O1	*b8*	90 90 90 b8 e7 20 19 08 2d e4 20 19 08	`nop; nop; nop;`**mov eax, 0x81920e7**;`sub eax, 0x81920e4`
	7	O2	*83*	83 c4 1c c3 83 ec 1c	`add esp, 0x1c; ret;` **sub esp, 0x1c**
	8	O3	*8b*	90 90 90 90 8b 44 24 04	`nop; nop; nop; nop;` **mov eax, DWORD PTR [esp+0x4]**
	9	O3	*55*	8d bc 27 00 00 00 00 55 57 56	`lea edi, [edi+eiz+0x0];` **push ebp**; `push edi; push esi`
R.F.N.	10	O0	*31**	e9 50 fd ff ff 31 ed 5e	`jmp 0xfffffd50;` **xor ebp, ebp**; `pop esi`
	11	O2	*89**	e9 85 fe ff ff 90 89 c2 31 c0	`jmp 0xfffffe8a; nop;` **mov edx, eax**; `xor eax, eax`
	12	O3	*a1**	8d b4 26 00 00 00 00 a1 d0 14 20 08	`lea esi, [esi+eiz*1+0];` **mov eax, ds:0x82014d0**
R.F.P.	13	O1	*83*	0f b6 c0 c3 83 ec 1c	`movzx eax,al; ret;` **sub esp, 0x1c**
	14	O2	*b8*	8d 74 26 00 b8 01 00 00 00	`lea esi, [esi+eiz*1+0x0];` **mov eax, 0x1**
	15	O3	*83*	8d 74 26 00 83 ec 1c c7 04	`lea esi, [esi+eiz*1+0x0];` **sub esp, 0x1c**

Table 5: Case study for the binary analysis (15 cases). Our explanation method ranks features and marks the most important features as red, followed by orange, gold, yellow. We also translate the hex code to assembling code for the ease of understanding. Note that the `F. start` refers to the function start detected by the deep learning classifier. The function start is also marked by a black square in the hex sequence. *For false negatives under R.F.N., we present the *real* function start that the classifier failed to detect, and explain why the function start is missed.

performed the same analysis for the PDF malware classifier, and the results are in Appendix-E.

6.1 Understanding Classifier Behavior

The primary application of our explanation method is to assess the reliability of the classifiers and help to establish the "trust". We argue that classifier reliability and trusts do not necessarily come from a high classification accuracy on the training data. Often cases, the training data is not complete enough to capture all the possible variances. Instead, trusts are more likely to be established by *understanding the model behavior*. In this section, we examine two key directions to understand how classifier makes decisions: (1) capturing and validating "golden rules" and well-established heuristics; and (2) discovering new knowledge.

Capturing Well-known Heuristics (C.W.H.). A reliable classifier should *at least* capture the well-known heuristics in the respective application domain. For example, in the area of binary reverse-engineering, security practitioners have accumulated a set of useful heuristics to identify the function start, some of which are even treated as "golden rules". Certain "golden rules" are derived from the specifications of the Application Binary Interface (ABI) standards [22]. For example, the ABI requires a function to store the old frame pointer (ebp) at the start if this function maintains a new frame pointer. This leads to the most commonly seen prologue `[push ebp; mov ebp, esp]`. Another set of well-established rules come from mainstream compilers. For example, GNU GCC often inserts nop instructions before a function start, which aligns the function for architectural optimization [43].

By analyzing the explanation results, we observed strong evidence that deep learning classifiers have successfully captured well-known heuristics. In Table 5, we show 4 most representative cases, one for each classifier (or optimization level). In Case-1, the classifier correctly detected the function start at "55". Then our LEMNA shows why 55 is marked as the function start by highlighting the importance of features (*i.e.*, the hex code nearby). The result matches the well-known golden rule, namely `[push ebp; mov ebp,esp]`. This suggests the classifiers are making decisions in a reasonable way. Similarly, Case-2 captures the function start "53" right after a "c3". This corresponds to a popular heuristic introduced by compilers as compilers often make a function exit in the end through a "`ret`" instruction (particularly at the O0 and O1 level).

In Case-4, "83" is the function start and LEMNA highlighted the "90" in red. This indicates that the classifier follows the " nop right before a function start" rule, which is caused by compilers padding "nop"s prior to aligned functions. Similarly, in Case-3, LEMNA highlighted padding instruction `[lea esi,[esi+eiz*1+0]]`, which is another pattern introduced by compilers. Overall, LEMNA shows that well-known heuristics are successfully captured by the classifiers.

During our analysis, we observe that well-known heuristics are widely applicable at the lower optimization levels (O0, O1), but do not cover as many binaries at the higher levels (O2, O3). For example, 95% of the functions at O0-level start with `[55 89 E5]`, matching the heuristics of Case-1. 74% of the O1-optimized functions have `ret` as the ending instruction (Case-2). On the contrary, only 30% of the binary functions at the O2 or O3 level match the well-known heuristics, *e.g.*, padding instructions at the function end ("`[90 90 90 90]`", "`[8d b4 26 00 00 00 00]`". This makes intuitive sense because the higher-level optimization would significantly *diversify* the code structure, making golden rules less effective.

Discovering New Knowledge (D.N.K.). In addition to matching well-known heuristics, we also examine if the classifiers have picked up new heuristics beyond existing knowledge. For security applications, we argue that the new heuristics need to be interpretable by domain experts. In the domain of binary analysis, many potentially useful heuristics are *specific to individual functions*, and it is hard to summarize all of them manually. For example, the utility functions inserted by the linker often have unique beginning code segments and those segments rarely appear elsewhere (*e.g.,* the `_start` function always start with `[xor ebp, ebp; pop esi]`). Manually organizing such rules are not practical. However, these rules, once derived by LEMNA, would make intuitive sense to domain experts.

As shown in Table 5, we analyze the explanation results and find that classifiers indeed learned new knowledge. We select five representative cases (ID 5–9). Case-5 shows that "`31`" is detected as the function start because of the subsequent `[ed 5e]`. "`[31 ed 5e]`" corresponds to the start of utility function `_start` (namely `[xor ebp, ebp; pop esi]`). This illustrates that our explanation method can help summarize unique prologues pertaining to special functions. Note that the function start "`31`" itself is not necessarily an important indicator. In fact, "`31`" represents an opcode (`xor`) that often appears in the middle of the functions. It is "`[ed 5e]`" that leads to the correct detection.

Case-6 illustrates another interesting pattern where "`2b`" is the most important feature to detect the function start at "`b8`". "`2b`" resides in instruction following the pattern `[mov eax, CONS1; sub eax, CONS2]` where `CONS1` and `CONS2` are constant values and `CONS1 - CONS2 = 0 or 3`. This pattern appears only in the prologues of "`register_tm_clones`" and "`deregister_tm_clones`", which are utility functions for transactional memory. Again this is a function-specific pattern to detect function start.

Case-7, Case-8 and Case-9 all have some types of "preparations" at the function start. In Case-7, "`[83, ec]`" is marked as the most important feature, which corresponds to the instruction `[sub esp, 0x1c]`. Instructions of this form are frequently used at function start to prepare the stack frame. For Case-8, `[mov eax, DWORD PTR [esp+0x4]]` is marked as the most indicative feature. This instruction is usually inserted to fetch the first argument of a function. Note that "`04`" has the red color, which is because "`04`" is used as the offset for `[esp+0x4]` to fetch the argument of the function. If this offset is of a different value, this instruction would not necessarily be an indicator of the function start. For Case-9, it starts with preserving the registers that are later modified (`[push ebp; push edi; push esi]`). Preservation of those registers, which is required by the calling convention (a common ABI standard), also frequently appears at the function start.

Overall, LEMNA validates that the classifiers' decision-making has largely followed explainable logics, which helps to establish the trust to these classifiers.

6.2 Troubleshooting Classification Errors

The deep neural networks, although highly accurate, still have errors. These errors should not be simply ignored since they often indicate insufficient training, which may be amplified in practice (due to the biased training). Our explanation method seeks to provide insights into "what caused the error" for a given misclassification. By inspecting the reason of errors, we seek to provide actionable guidelines for targeted error correction.

Reasons for False Negatives (R.F.N.). For the binary analysis application, the classifiers would occasionally miss the real function start. As shown in Table 5 (under "R.F.N."), given a false negative, we explain "*why the real function start is not classified as a function start*". Specifically, we feed the tuple `(Code-sequence, Real-function-start)` into LEMNA, and the red-colored features are the reasons for not recognizing the function start. For example, in Case-10, "`[50 fd]`" is marked as the main reason, which correspond to "`[jmp 0xfffffd50]`". This instruction almost always appears in the middle of routines or functions, which misleads the classifier to think the substantial `31` is not a function start. This is an outlier case because this "`[50 fd]`" happens to be the last instruction of a special region `.plt`, which is followed by the `_start` function. Case-11 and Case-12 are mis-classified due to instructions "`[mov edx,eax]`" and "`[mov eax,ds:0x82014d0]`", which often appear in the middle of functions.

Reasons for False Positives (R.F.P.). Table 5 also show examples where the classifier picked the wrong function start. Here, we feed the tuple `(Code-Sequence, Wrong-function-start)` into LEMNA to explain why the wrong function start is picked. For example, Case-13 highlighted "`c3`" in red which represents the "`ret`" instruction. Typically, "`ret`" is located at the end of a function to make the exit, which makes the next byte "`83`" a strong candidate for the function start. However, Case-13 is special because "`ret`" is actually placed in the middle of a function for optimization purposes. Case-14 and Case-15 are both misled by the padding instruction `[lea esi,[esi+eiz*1+0x0]]` which is often used to align functions. However, in both cases, this padding instruction is actually used to align the basic blocks inside of the function.

Overall, LEMNA shows that the errors are largely caused by the fact that the misleading patterns are dominating over the real indicators. To mitigate such errors, we need to pinpoint the corresponding areas in the feature space and suppress the misleading patterns.

6.3 Targeted Patching of ML Classifiers

Based on the above results, we now develop *automatic* procedures to convert the"insights" into actions to patch the classifiers.

Patching Method. To patch a specific classification error, our idea is to identify the corresponding parts of the classifier that are under-trained. Then we craft targeted training samples to augment the original training data. Specifically, given a misclassified instance, we apply LEMNA to pinpoint the small set of features ($\boldsymbol{F_x}$) that cause the errors. Often cases, such instances are outliers in the training data, and do not have enough "counter examples". To this end, our strategy is to augment the training data by adding related "counter examples", by replacing the feature values of $\boldsymbol{F_x}$ with *random values.*

We use an example (Case-10 in Table 5) to describe the patching procedure. The classifier missed the function start due to "`[50 fd]`", a hex pattern that often exists in the middle of a function. Ideally, the classifier should have picked up the other pattern "`[31 ed 5e]`" to locate the function start. Unfortunately, the impact of the wrong pattern is too dominating. To this end, we can add new samples to reduce the impact of the misleading features ("`[50 fd]`") and

Application	Num. of Samples	k_n	k_p	Before		After	
				FN	FP	FN	FP
Binary O0	4,891,200	5	5	3	1	0	0
Binary O1	4,001,820	3	4	48	33	23	29
Binary O2	4,174,000	4	5	107	129	59	62
Binary O3	5,007,800	2	5	83	41	15	39
PDF Malware	3,000	6	15	28	13	10	5

Table 6: Classification result before and after patching. k_n (k_p) referes to the number of augmented samples generated for each false negative (false positive). Note that for function start detection, the number of samples refers to the number of total hex code in the testing set.

promote the right indicator ("[31 ed 5e]"). The new samples are generated by replacing the hex value of "[50 fd]" with random hex values. By adding the new samples to the training data, we seek to reduce the errors in the retrained classifier.

Evaluation Results. To demonstrate the effectiveness of patching, we perform the above procedure on all 5 classifiers. For each false positive and false negative, we generate k_p and k_n new samples respectively. Note that k_p and k_n are not necessarily the same, but they both need to be *small*. After all, we want to patch the targeted errors without hurting the already high accuracy of the classifiers. Consistently for all the classifiers, we replace the top 5 misleading features and retrain the models with 40 epochs.

Table 6 shows the classifier performance before and after the patching. We have tested the sensitivity of the parameters and find the results remain relatively consistent as long as we set k_p and k_n between 2 to 10 (Appendix-F). Due to the space limit, Table 6 only presents one set of the results for each classifier. Our experiment shows that both false positives and false negatives can be reduced after retraining for all five classifiers. These results demonstrate that by understanding the model behavior, we can identify the weaknesses of the model and enhance the model accordingly.

7 DISCUSSION

Benefits v.s. Risks. LEMNA is designed to assist security analysts to understand, scrutinize and even patch a deep learning based security system. While designed from the defense perspective, it might be used by an attacker to seek the weakness of a deep learning classifier. However, we argue that this should not dilute the value of LEMNA, and should not be a reason for not developing explanation tools. The analogy is the software fuzzing techniques [13, 73]: while fuzzing tools can be used by hackers to seek vulnerabilities to exploit, the fuzzing techniques have significantly benefited the software industry by facilitating software testing to find and fix vulnerabilities before the software release.

Guidelines for Analyzing LEMNA's Outputs. LEMNA outputs an "explanation" to each testing case. To thoroughly examine a classifier, developers might need to run a large number of testing cases through LEMNA. Manually reading each case's explanation is time-consuming, and thus we suggest a more efficient method, which is to *group* similar explanations first. In §6, we grouped explanations that are exactly the same before picking the "most representative" cases. In practice, developers can use any other clustering techniques to group explanations as needed.

Broader Security Applications. LEMNA is evaluated using two popular security applications. There are many other security applications such as detecting the "function end" for binary code, pinpointing the function types and detecting vulnerable code [15, 24, 47, 52, 66]. They can also potentially benefit from LEMNA, given that their deep learning architectures are RNN or MLP. Note that models like CNN share some similarities with MLP, and thus LEMNA can potentially help with related applications (*e.g.*, image analysis). Future work will explore the applicability of LEMNA in broader application domains.

Other Deep Learning Architectures. In addition to MLP and RNN, there are other deep learning architectures such as sequence-to-sequence networks [4, 60], and hybrid networks [25, 36, 71]. Although, these architectures primarily find success in fields such as machine translation [4] and image captioning [25], initial evidence shows that they have the potential to play a bigger role in security [36, 71]. Once concrete security applications are built in the future, we plan to test LEMNA on these new architectures.

Feature Obfuscation. LEMNA is useful when features are interpretable, but this may not be true for all applications. In particular, researchers recently proposed various methods [8, 67, 70] to obfuscate input features to increase the difficulty of running adversarial attacks. Possibly because feature obfuscation often degrades classifier accuracy, these techniques haven't received a wide usage yet. LEMNA is not directly applicable to classifiers trained on obfuscated features. However, if the model developer has a mapping between the raw and obfuscated features, the developer can still translate LEMNA's output to the interpretable features.

8 OTHER RELATED WORK

Since most related works have been discussed in §2 and §3, we briefly discuss other related works here.

Improving Machine Learning Robustness. A deep learning model can be deceived by an adversarial sample (*i.e.,* a malicious input crafted to cause misclassification) [61]. To improve the model resistance, researchers have proposed various defense methods [9, 20, 36, 40, 67]. The most relevant work is adversarial training [20]. Adversarial training seeks to add adversarial examples to the training dataset to retrain a more robust model. Various techniques are available to craft adversarial examples for adversarial training [11, 33, 42, 72]. A key difference between our patching method and the standard adversarial training is that our patching is based on the understanding of the errors. We try to avoid blindly retraining the model which may introduce new vulnerabilities.

Mitigating the Influence of Contaminated Data. Recent research has explored ways to mitigate misclassifications introduced by contaminated training data [10, 12, 46, 65]. A representative method is "machine unlearning" [10], which is to remove the influence of certain training data by transforming the standard training algorithms into a summation form. A more recent work [29] proposes to utilize an influence function to identify data points that contribute to misclassification. Our approach is complementary to existing works: we propose to augment training data to fix under-trained components (instead of removing bad training data). More importantly, LEMNA helps the human analysts to understand these errors before patching them.

9 CONCLUSION

This paper introduces LEMNA, a new method to derive high-fidelity explanations for individual classification results for security applications. LEMNA treats a target deep learning model as a blackbox and approximates its decision boundary through a mixture regression model enhanced by fused lasso. By evaluating it on two popular deep learning based security applications, we show that the proposed method produces highly accurate explanations. In addition, we demonstrate how machine learning developers and security analysts can benefit from LEMNA to better understand classifier behavior, troubleshoot misclassification errors, and even perform automated patches to enhance the original deep learning model.

10 ACKNOWLEDGMENTS

We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Tesla K40 GPU used for this research. We also would like to thank the anonymous reviewers and Jia Yan for their helpful feedback. This project was supported in part by NSF grants CNS-1718459, CNS-1750101 and CNS-1717028. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of any funding agencies.

REFERENCES

[1] 2014. Mimcus. https://github.com/srndic/mimicus. (2014).

[2] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. 2014. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *Proceedings of the 20th Network and Distributed System Security Symposium (NDSS).*

[3] Sebastian Bach, Alexander Binder, Grégoire Montavon, Frederick Klauschen, Klaus-Robert Müller, and Wojciech Samek. 2015. On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. *PloS one* (2015).

[4] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[5] Tiffany Bao, Johnathon Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. Byteweight: Learning to recognize functions in binary code. In *Proceedings of the 23rd USENIX Security Symposium (USENIX Security).*

[6] Osbert Bastani, Carolyn Kim, and Hamsa Bastani. 2017. Interpreting blackbox models via model extraction. *arXiv preprint arXiv:1705.08504* (2017).

[7] Konstantin Berlin, David Slater, and Joshua Saxe. 2015. Malicious behavior detection using windows audit logs. In *Proceedings of the 8th Workshop on Artificial Intelligence and Security (AISec).*

[8] Arjun Nitin Bhagoji, Daniel Cullina, and Prateek Mittal. 2017. Dimensionality reduction as a defense against evasion attacks on machine learning classifiers. *arXiv preprint arXiv:1704.02654* (2017).

[9] Xiaoyu Cao and Neil Zhenqiang Gong. 2017. Mitigating evasion attacks to deep neural networks via region-based classification. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC).*

[10] Yinzhi Cao and Junfeng Yang. 2015. Towards making systems forget with machine unlearning. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P).*

[11] Nicholas Carlini and David Wagner. 2017. Towards evaluating the robustness of neural networks. In *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P).*

[12] Gert Cauwenberghs and Tomaso Poggio. 2000. Incremental and decremental support vector machine learning. In *Proceedings of the 13th Conference on Neural Information Processing Systems (NIPS).*

[13] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P).*

[14] François Chollet et al. 2017. Keras. (2017).

[15] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. 2017. Neural Nets Can Learn Function Type Signatures From Binaries. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security).*

[16] George E Dahl, Jack W Stokes, Li Deng, and Dong Yu. 2013. Large-scale malware classification using random projections and neural networks. In *Proceedings of the 38th International Conference on Acoustics, Speech and Signal Processing (ICASSP).*

[17] R.C. Fong and A. Vedaldi. 2017. Interpretable Explanations of Black Boxes by Meaningful Perturbation. In *Proceedings of the 16th International Conference on Computer Vision (ICCV).*

[18] Chuang Gan, Naiyan Wang, Yi Yang, Dit-Yan Yeung, and Alex G Hauptmann. 2015. Devnet: A deep event network for multimedia event detection and evidence recounting. In *Proceedings of the 28th Conference on Computer Vision and Pattern Recognition. (CVPR).*

[19] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. 2018. AI^2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P).*

[20] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and harnessing adversarial examples. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR).*

[21] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. 2016. Adversarial perturbations against deep neural networks for malware classification. *arXiv preprint arXiv:1606.04435* (2016).

[22] The Santa Cruz Operation Inc. 1997. System V application binary interface. (1997).

[23] Anil K. Jain and B. Chandrasekaran. 1982. Dimensionality and Sample Size Considerations in Pattern Recognition Practice. *Handbook of Statistics* (1982).

[24] Ahmad Javaid, Quamar Niyaz, Weiqing Sun, and Mansoor Alam. 2016. A deep learning approach for network intrusion detection system. In *Proceedings of the 9th International Conference on Bio-inspired Information and Communications Technologies (BIONETICS).*

[25] Justin Johnson, Andrej Karpathy, and Li Fei-Fei. 2016. Densecap: Fully convolutional localization networks for dense captioning. In *Proceedings of the 29th Conference on Computer Vision and Pattern Recognition (CVPR).*

[26] Ian T Jolliffe. 1986. Principal component analysis and factor analysis. In *Principal component analysis.*

[27] Michael I Jordan and Robert A Jacobs. 1994. Hierarchical mixtures of experts and the EM algorithm. *Neural computation* (1994).

[28] Abbas Khalili and Jiahua Chen. 2007. Variable selection in finite mixture of regression models. *Journal of the american Statistical association* (2007).

[29] Pang Wei Koh and Percy Liang. 2017. Understanding Black-box Predictions via Influence Functions. In *Proceedings of the 34th International Conference on Machine Learning (ICML).*

[30] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th Conference on Neural Information Processing Systems (NIPS).*

[31] Himabindu Lakkaraju, Stephen H Bach, and Jure Leskovec. 2016. Interpretable decision sets: A joint framework for description and prediction. In *Proceedings of the 22nd International Conference on Knowledge Discovery and Data Mining (KDD).*

[32] Jiwei Li, Will Monroe, and Dan Jurafsky. 2016. Understanding Neural Networks through Representation Erasure. *arXiv preprint arXiv:1612.08220* (2016).

[33] Yanpei Liu, Xinyun Chen, Chang Liu, and Dawn Song. 2017. Delving into transferable adversarial examples and black-box attacks. In *Proceedings of the 5th International Conference on Learning Representations (ICLR).*

[34] Scott M Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. In *Proceedings of the 30th Conference on Neural Information Processing Systems (NIPS).*

[35] Mengersen K. Marin, J.M. and C.P. Robert. 2005. Bayesian modelling and inference on mixtures of distributions. *Handbook of statistics* (2005).

[36] Dongyu Meng and Hao Chen. 2017. Magnet: a two-pronged defense against adversarial examples. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS).*

[37] Bengt Muthén and Kerby Shedden. 1999. Finite mixture modeling with mixture outcomes using the EM algorithm. *Biometrics* (1999).

[38] In Jae Myung. 2003. Tutorial on maximum likelihood estimation. *Journal of mathematical Psychology* (2003).

[39] Bruno A Olshausen and David J Field. 1996. Emergence of simple-cell receptive field properties by learning a sparse code for natural images. *Nature* (1996).

[40] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. 2016. Distillation as a defense to adversarial perturbations against deep neural networks. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P).*

[41] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *Journal of machine learning research* (2011).

[42] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP)*.

[43] Paradyn Project. 2016. Dyninst: An application program interface (api) for runtime code generation. *Online, http://www.dyninst.org* (2016).

[44] Sarunas J. Raudys and Anil K. Jain. 1991. Small Sample Size Effects in Statistical Pattern Recognition: Recommendations for Practitioners. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (1991).

[45] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. Why should i trust you?: Explaining the predictions of any classifier. In *Proceedings of the 22nd International Conference on Knowledge Discovery and Data Mining (KDD)*.

[46] Enrique Romero, Ignacio Barrio, and Lluís Belanche. 2007. Incremental and decremental learning for linear support vector machines. In *Proceedings of the 17th International Conference on Artificial Neural Networks (ICANN)*.

[47] Sherif Saad, Issa Traore, Ali Ghorbani, Bassam Sayed, David Zhao, Wei Lu, John Felix, and Payman Hakimian. 2011. Detecting P2P botnets through network behavior analysis and machine learning. In *Proceedings of the 9th International Conference on Privacy, Security and Trust (PST)*.

[48] Joshua Saxe and Konstantin Berlin. 2015. Deep neural network based malware detection using two dimensional binary program features. In *Proceedings of the 10th International Conference on Malicious and Unwanted Software (MALWARE)*.

[49] Henry Scheffe. 1947. The relation of control charts to analysis of variance and chi-square tests. *J. Amer. Statist. Assoc.* (1947).

[50] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. 2016. Grad-cam: Visual explanations from deep networks via gradient-based localization. *arxiv. org/abs/1610.02391 v3* (2016).

[51] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. 2009. Automatic reverse engineering of malware emulators. In *Proceedings of the 30th IEEE Symposium on Security and Privacy (S&P)*.

[52] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing Functions in Binaries with Neural Networks. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security)*.

[53] Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. 2017. Learning Important Features Through Propagating Activation Differences. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*.

[54] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. 2013. Deep inside convolutional networks: Visualising image classification models and saliency maps. *arXiv preprint arXiv:1312.6034* (2013).

[55] Charles Smutz and Angelos Stavrou. 2012. Malicious PDF detection using metadata and structural features. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*.

[56] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. 2008. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of 4th International Conference on Information Systems Security (ICISS)*.

[57] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller. 2014. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806* (2014).

[58] Nedim Srndic and Pavel Laskov. 2014. Practical evasion of a learning-based classifier: A case study. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*.

[59] Mukund Sundararajan, Ankur Taly, and Qiqi Yan. 2016. Gradients of counterfactuals. *arXiv preprint arXiv:1611.02639* (2016).

[60] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Proceedings of the 27th Conference on Neural Information Processing Systems (NIPS)*.

[61] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2013. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199* (2013).

[62] Tuan A Tang, Lotfi Mhamdi, Des McLernon, Syed Ali Raza Zaidi, and Mounir Ghogho. 2016. Deep learning approach for network intrusion detection in software defined networking. In *Proceedings of the 12th International Conference on Wireless Networks and Mobile Communications (WINCOM)*.

[63] Theano Development Team. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arxiv.org/abs/1605.02688* (2016).

[64] Robert Tibshirani, Michael Saunders, Saharon Rosset, Ji Zhu, and Keith Knight. 2005. Sparsity and smoothness via the fused lasso. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* (2005).

[65] Cheng-Hao Tsai, Chieh-Yen Lin, and Chih-Jen Lin. 2014. Incremental and decremental training for linear classification. In *Proceedings of the 20th International Conference on Knowledge Discovery and Data Mining (KDD)*.

[66] Grigorios Tzortzis and Aristidis Likas. 2007. Deep belief networks for spam filtering. In *Proceedings of the 19th International Conference on Tools with Artificial Intelligence (ICTAI)*.

[67] Qinglong Wang, Wenbo Guo, Kaixuan Zhang, II Ororbia, G Alexander, Xinyu Xing, Xue Liu, and C Lee Giles. 2016. Learning adversary-resistant deep neural networks. *arXiv preprint arXiv:1612.01401* (2016).

[68] Qinglong Wang, Wenbo Guo, Kaixuan Zhang, Alexander G Ororbia II, Xinyu Xing, Xue Liu, and C Lee Giles. 2017. Adversary resistant deep neural networks with an application to malware detection. In *Proceedings of the 23rd International Conference on Knowledge Discovery and Data Mining (KDD)*.

[69] Han Xiao, Kashif Rasul, and Roland Vollgraf. 2017. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms. *arXiv preprint arXiv:1708.07747* (2017).

[70] Cihang Xie, Jianyu Wang, Zhishuai Zhang, Zhou Ren, and Alan Yuille. 2017. Mitigating adversarial effects through randomization. In *Proceedings of the 6th International Conference on Learning Representations (ICLR)*.

[71] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proceedings of the 24th Conference on Computer and Communications Security (CCS)*.

[72] Zhaogui Xu, Shiqing Ma, Xiangyu Zhang, Shuofei Zhu, and Baowen Xu. 2018. Trojanning Attack on Neural Networks. In *Proceedings of the 25th Network and Distributed System Security Symposium (NDSS)*.

[73] Michal Zalewski. 2007. American fuzzy lop. (2007).

[74] Matthew D Zeiler and Rob Fergus. 2014. Visualizing and understanding convolutional networks. In *Proceedings of the 13th European Conference on Computer Vision (ECCV)*.

[75] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22nd USENIX Conference on Security (USENIX Security)*.

[76] Luisa M Zintgraf, Taco S Cohen, Tameem Adel, and Max Welling. 2017. Visualizing deep neural network decisions: Prediction difference analysis. In *Proceedings of the 5th International Conference on Learning Representations (ICLR)*.

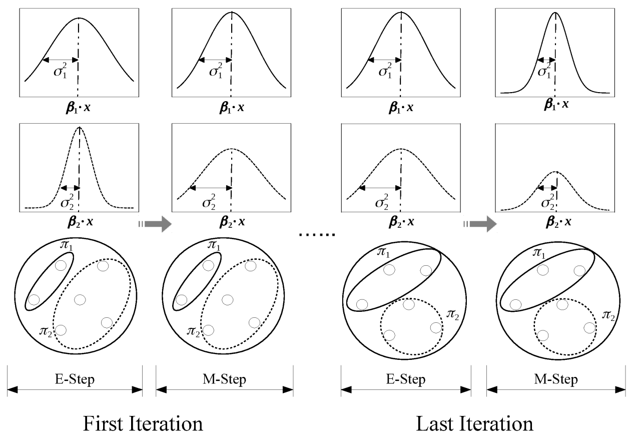


Figure 7: The illustration of an EM algorithm. In each iteration, the algorithm first assigns each data sample to a corresponding Gaussian distribution obtained from the previous iteration (E-Step). Then, it re-computes the Gaussian distributions based on the assignment of the data samples (M-Step). The algorithm repeatedly perform E-Step and M-Step until there is no change to the Gaussian distributions or the assignment of the data samples.

APPENDIX - A. DETAIL OF EM ALGORITHM

As is specified in Section §4, we utilize EM algorithm to estimate parameters while learning a mixture regression model enhanced by fused lasso. Here, we provide more detail about this process.

Recall that a mixture regression model contains K components, each of which indicates an individual linear regression model. In the E-Step, we assign each data sample $\mathbf{x}_i$ to a Gaussian distribution corresponding to one of the components. To achieve this, we introduce a set of latent variables $\{z_{i1}, z_{i2}, ..., z_{iK}\}$, and use it to indicate to which distribution a data sample is assigned. Note that we use $z_{ik} = 1$ to represent that the data sample $\mathbf{x}_i$ is assigned to the k^{th} distribution.

Application	Model Structure	Activation	Optimizer	Learning Rate	Dropout Rate	Batch Size	Epoch
Binary Func. Start	255-8-2	relu	adam	0.001	0.5	100	100
PDFmalware	135-100-50-10-2	sigmoid	adam	0.001	0.2	100	30

Table 7: The hyper-parameters of corresponding deep learning models. Here "model structure" depicts the number of layers in the model as well as the number of units in each layer. Note that for the four model in the function start identification application (*i.e.,* `O0-O3`), we use the same set of hyper-parameters.

To compute values for latent variables, we define

$$p(z_{ik} = 1) = \pi_k \,, \tag{8}$$

and thus have the following

$$p(y_i|\mathbf{x}_i, z_{i1:K}) = \prod_{k=1}^{K} [\mathcal{N}(y_i|\boldsymbol{\beta}_k \mathbf{x}_i, \sigma_k^2)]^{z_{ik}} \,, \tag{9}$$

where $\mathcal{N}(y_i|\boldsymbol{\beta}_k \mathbf{x}_i, \sigma_k^2)$ indicates the k^{th} Gaussian distribution with the mean and variance equal to $\boldsymbol{\beta}_k \mathbf{x}_i$ and σ_k^2 respectively.

From the Equation (9), we can derive a likelihood function below

$$\begin{aligned} p(y, z|\mathbf{x}, \Theta) &= \prod_{i=1}^{N} p(y_i, z_{i1}, \dots, z_{in}|\boldsymbol{\beta}\mathbf{x}_i, \sigma^2) \\ &= \prod_{k=1}^{K} \prod_{i=1}^{N} [\pi_k \mathcal{N}(x_i|\boldsymbol{\beta}_k \mathbf{x}_i, \sigma_k^2)]^{z_{ik}} \\ &= \prod_{k=1}^{K} \pi_k^{n_k} \prod_{i=1}^{N} [\mathcal{N}(x_i|\boldsymbol{\beta}_k \mathbf{x}_i, \sigma_k^2)]^{z_{ik}} \end{aligned} \tag{10}$$

from which we can further compute the expectation of this log-likelihood function (*i.e., Q* function) as follow:

$$\begin{aligned} Q(\Theta, \Theta^{(t)}) =& E[\log p(y, z|\mathbf{x}, \Theta)|y, \mathbf{x}, \Theta^{(t)}] \\ =& \sum_{k=1}^{K} \{n_k \log \pi_k + \sum_{i=1}^{N} \hat{z}_{ik} \cdot \\ & [\log(\frac{1}{\sqrt{2\pi}}) - \log \pi_k - \frac{1}{\sigma_k^2}(y_i - \boldsymbol{\beta}_k \mathbf{x}_i)^2]\} \,. \end{aligned} \tag{11}$$

Here, $n_k = \sum_{k=1}^{N} Ez_{ik}$. Θ indicates all of the parameters. $\hat{z}_{ik} = Ez_{ik}$ which can be further represented as

$$\hat{z_{ik}} = \frac{\pi_k \mathcal{N}(y_i|\boldsymbol{\beta}_k \mathbf{x}_i, \sigma_k^2)}{\sum_{k=1}^{K} \pi_k \mathcal{N}(y_i|\boldsymbol{\beta}_k \mathbf{x}_i, \sigma_k^2)} \,, i = 1, \dots, N \,, k = 1, \dots, K \,, \tag{12}$$

With the latent variables computed through the Equation (12), we can assign each data sample to a corresponding Gaussian distribution. Then, in the M-step, we re-compute the parameters by maximizing the aforementioned Q function with respect to each parameter. More specifically, we can compute parameter σ_k^2 and π_k by using the following equations

$$\begin{aligned} \sigma_k^2 &= \frac{\sum_{i=1}^{N} \hat{z}_{ik}(y_i - \boldsymbol{\beta}_k \mathbf{x}_i)^2}{n_k} \,, k = 1, 2, \dots, K \,, \\ \pi_k &= \frac{n_k}{N} \,, k = 1, 2, \dots, K \,. \end{aligned} \tag{13}$$

Recall that we re-compute parameter $\boldsymbol{\beta}_{1:K}$ by minimizing the Equation (7) shown in Section §4. While it can be resolved by using `MLE`, in order to improve the efficiency of resolving this equation, we can also an alternative algorithm introduced in [64]. As is depicted in Figure 7, we can repeatedly perform E-step and then M-step until the parameters converge, and thus output the mixture regression model enhanced by fused lasso.

APPENDIX - B. MULTI-CLASS VS MULTIPLE SINGLE-CLASS APPROXIMATION

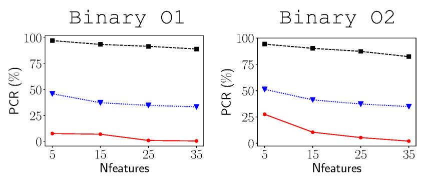
As is mentioned in Section 4.3, we choose to perform model approximation with multiple single-class approximation rather than a single muti-class approximation. Here, we discuss the rationale behind our choice.

As is stated in Section 4.1, the Equation (4) represents a practice that estimates parameters for a binary classifier, in which there are $K \times (2 + M)$ parameters involved in the process of model learning. For a single mixture regression model that classifies a data sample $\mathbf{x}_i$ into one of L categories ($L > 2$), the parameter $\boldsymbol{\beta}_k$ and $\boldsymbol{\sigma}_k^2$ no longer represent a vector and a singular value. Rather, they denote matrices with the dimensionality of $L \times M$ and $L \times L$ respectively. In the process of learning a mixture regression model, this means that, in addition to $\pi_{1:K}$ which still represents K parameters, the learning algorithm needs to estimate $\boldsymbol{\beta}_{1:K}$ and $\boldsymbol{\sigma}_{1:K}^2$, which denote $L \times K \times M$ and $L^2 \times K$ parameters respectively.

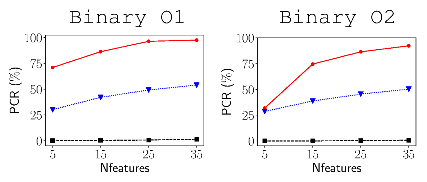
According to learning heuristics [23, 44], the more parameters a learning algorithm needs to estimate, the more data samples it would typically need. Technically speaking, following the data point sampling approach commonly used by other model induction explanation techniques, we have no difficulty in synthesizing sufficient data samples to perform model learning (*i.e.,* parameter estimation) reasonably well. However, the practice shows that learning a model with a large amount of data samples typically requires substantial amount of computation resources. Recall that for each data sample we have to train an individual mixture regression model in order to derive an explanation. Therefore, we select the single-class approximation scheme that can yield an explanation in a more efficient fashion, even though both of the approximation schemes could yield model(s) representing the equally good approximation for the corresponding local decision boundary.

APPENDIX - C. HYPER-PARAMETERS OF TARGET DEEP LEARNING MODEL

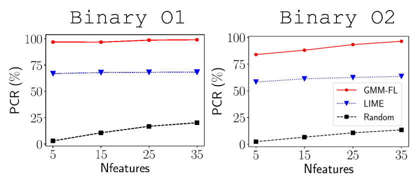
In Table 7, we show the hyper-parameters used for training corresponding deep learning models. Regarding function start detector, we utilized a recurrent neural network in which its first, second and output layers are an embedding layer with 256 units, a bi-directional RNN with 8 hidden units and a softmax classifier respectively. With respect to the application of PDF malware classification, we used a standard `MLP` which contains one input layer, three hidden layers and one output layer. The number of hidden units tied to each layer is presented in Table 7.



(a) `Feature Deduction` **testing.**



(b) `Feature Augmentation` **testing.**

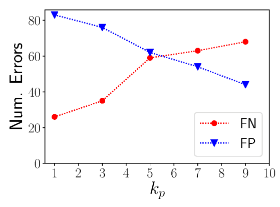


(c) `Synthetic` **testing.**

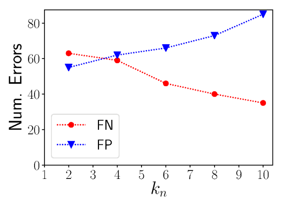
Figure 8: Fidelity validation results of `O1` **and** `O2`**. y-axis denotes the positive classification rate** `PCR` **and y-axis denote the number of selected features** `NFeature` **by the explanation method.**

Cases	ID	Classifier’s Label	Explanation (Important features)			
C.W.H.	16	Malware	F_{31}[JavaScript]=1	F_{114}[prod. oth]=0 F_{33}[JS Obfu.]=1	F_{56}[crea. uc]=0	F_{112}[producer mis.]=0
	17	Benign	F_{114}[prod. oth]=1	F_{112}[producer mis.]=1 F_{31}[JavaScript]=0	F_{33}[JS Obfu.]=0,	F_{56}[crea. uc]=1
R.F.N	18	Benign	F_{114}[prod. oth]=1 F_{33}[JS Obfu.]=0	F_{112}[producer mis.]=1	F_{31}[JavaScript]=0	F_{56}[crea. uc]=1
R.F.P	19	Malware	F_{31}[JavaScript]=1	F_{33}[JS Obfu.]=1 F_{114}[prod. oth]=0	F_{56}[crea. uc]=0	F_{112}[producer mis.]=0

Table 8: Case study for PDF malware classification (4 cases). The feature 31 and 33 are related to “*JavaScript Object Markers*” and “*Obfuscated JavaScript Object Markers*” which are indicators of “malware” files; Feature 56, 112 and 114 refer to “*Creator: Upper Case Characters*”, “*Differences in Producer Values*”, and “*Producer: Other Characters*” which are indicators of “benign” files. The feature values have been normalized to 0 or 1. We mark the most important features as red , followed by orange , gold , yellow .



(a) Fixing $k_n = 4$.



(b) Fixing $k_p = 5$.

Figure 9: Sensitivity tests on k_n and k_p.

APPENDIX - D. FIDELITY TEST FOR O1 AND O2 COMPILATION OPTIONS

Figure 8 shows the results of fidelity tests for O1 and O2 datasets. The results are consistent with those of other classifiers.

APPENDIX - E. MALWARE CLASSIFIER CASES

Table 8 shows 4 cases studies on the PDF Malware classifier’s decisions, which correspond to true positives, true negatives, false positives and fale negatives respectively. We also present the labels assigned by the classifier.

Catching Well-known Heuristics (C.W.H.). Case-16 classified as a malware primarily because feature F_{31} and F_{33} are set to non-zero values. As is shown in Table 8, these features are related to javascript objects, which match well-known heuristics and indicators of malicious PDF files. In the contrary, Case-17 has a benign file and features related to javascripts have zero values (*e.g.*, no javascript code in the file).

Reasons for False Positives/Negative (R.F.P., R.F.N). Case-18 and Case-19 represents false positives and negatives. Our explanation results show that the two instances are mis-classified because they violated the well-known heuristics learned by the classifier. For example, Case-18 is a malware that contains “ng” injected in the javascript. As a result, the Features F_{31} and F_{33} both have a zero value, and the classifier cannot detect this type of malware. On the contrary, if the benign file somehow contains some javascript code (*e.g.*, Case-19), the classifier will incorrectly label them as malware.

APPENDIX - F. SENSITIVITY OF k_n AND k_p

In section §6.3, the patching method has two hyper-parameters k_n and k_p. Here, we show the results of the sensitivity tests on these two these parameters. We select the classifier trained for binary function start detection using the O2 dataset. Our experiment mythology is to fix one parameter and swap the other one. Then we observe the changes of the re-trained classifier’s false positives and false negatives. In Figure 9a, we fix $k_n = 4$ and then set $K_p = 1, 3, 5, 7, 9$. In Figure 9b, we fix $k_p = 5$ and set $k_n = 2, 4, 6, 8, 10$. The results show that increasing k_p will reduce false positives but may increase false negatives. On the contrary, increasing k_n will reduce false negatives but may increase false positives. The results confirm our statements in §6.3. Targeted patching should limit to using small k_p and k_n to patch the target errors while avoiding introducing new errors. By adjusting k_p and k_n, security analysts can reduce false positives and false negatives at the same time. In §6.3 we present the selected results where the false positives and the false negatives are relatively balanced ($k_n = 4$ and $k_p = 5$ for this classifier).