Don't Paint It Black: White-Box Explanations for Deep Learning in Computer Security

Alexander Warnecke, Daniel Arp, Christian Wressnegger, Konrad Rieck

Institute of System Security
TU Braunschweig

ABSTRACT

Deep learning is increasingly used as a basic building block of security systems. Unfortunately, deep neural networks are hard to interpret, and their decision process is opaque to the practitioner. Recent work has started to address this problem by considering *black-box* explanations for deep learning in computer security (CCS'18). The underlying explanation methods, however, ignore the structure of neural networks and thus omit crucial information for analyzing the decision process. In this paper, we investigate *white-box* explanations and systematically compare them with current black-box approaches. In an extensive evaluation with learning-based systems for malware detection and vulnerability discovery, we demonstrate that white-box explanations are more *concise*, *sparse*, *complete* and *efficient* than black-box approaches. As a consequence, we generally recommend the use of white-box explanations if access to the employed neural network is available, which usually is the case for stand-alone systems for malware detection, binary analysis, and vulnerability discovery.

1 INTRODUCTION

Over the last years, deep learning has been increasingly recognized as an effective tool for computer security. Different types of neural networks have been integrated into security systems, for example, for malware detection [19, 22, 32], binary analysis [10, 39, 47], and vulnerability discovery [29]. Deep learning, however, suffers from a severe drawback: Neural networks are hard to interpret and their decisions are opaque to practitioners. Even simple tasks, such as determining which features of an input contribute to a prediction, are challenging to solve on neural networks. Consequently, the fascination of deep learning has started to mix with skepticism in the security community, as opaque systems are tedious to analyze and difficult to protect from attacks [7, 34].

As a remedy, Guo et al. [20] have recently started to explore techniques for explaining the decisions of deep learning in security. The proposed method, LEMNA, can determine the relevance of features contributing to a prediction by approximating the decision function of a neural network. The method achieves promising results in different security applications, yet the approach is designed as a *black-box method* that ignores the structure of the neural networks and thereby omits vital information for generating explanations. In practice, this is an unnecessary limitation. In all cases where predictions and explanations are computed from within the same system, access to the neural network is naturally available. This, for example, is the case in all stand-alone systems for malware detection and vulnerability discovery.

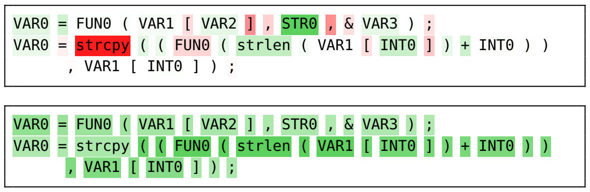
In this paper, we investigate *white-box explanations* for deep learning that analyze the structure of neural networks for explaining predictions and have not been studied in the context of security so far. To compare the different explanation concepts, we implement three state-of-art white-box methods [5, 41, 44] and compare them to three black-box approaches [20, 30, 35] in different security applications. In particular, we build on recent research on deep learning in security and explain the predictions of two systems for Android malware detection [19, 32], one system for detecting malicious PDFs [46], and one system for discovering security vulnerabilities [29].

For our analysis, we introduce performance measures that enable quantatively assessing and comparing explanations. We find that white-box explanations significantly outperform black-box approaches under these measures and thereby provide the following advantages in practice:

- *Conciseness:* The explanations generated by the white-box methods are more concise than the black-box explanations. That is, the identified features have 30 % more impact to the predictions on average.
- *Sparsity:* The white-box explanations are more sparse than those from black-box approaches, providing few important features for the interpretation of a decision. On average, white-box explanations are 19 % sparser than black-box ones.
- *Completeness:* The white-box methods are applicable to all decisions of the four considered security systems. The black-box methods suffer from restrictions and non-determinism, and can not provide meaningful results in 29 % of the cases.
- *Efficiency:* The white-box methods are significantly faster than black-box approaches and yield a performance increases of at least one order of magnitude in each of the considered security applications.

In addition to this quantitative analysis, we also qualitatively examine the generated explanations. As an example of this investigation, Figure 1 shows a black-box and a white-box explanation for the deep learning system VulDeePecker [29]. While the white-box approach provides a fine-grained and nuanced representation of the relevant features, the black-box method generates an unsharp explanation that is difficult to interpret and not sufficient to understand how the neural network arrives at a decision.

As the final part of our analysis, we investigate the validity of the decisions of the security systems with the help of the generated explanations. This analysis demonstrates the utility of explainable learning, yet it unveils a considerable number of artifacts encoded in the neural networks. Note, as an example, that both approaches in Figure 1 highlight punctuation characters, such as a semicolon



```
VAR0 = FUN0 ( VAR1 [ VAR2 ] , STR0 , & VAR3 ) ;
VAR0 = strcpy ( ( FUN0 ( strlen ( VAR1 [ INT0 ] ) + INT0 ) )
       , VAR1 [ INT0 ] ) ;
```

```
VAR0 = FUN0 ( VAR1 [ VAR2 ] , STR0 , & VAR3 ) ;
VAR0 = strcpy ( ( FUN0 ( strlen ( VAR1 [ INT0 ] ) + INT0 ) )
       , VAR1 [ INT0 ] ) ;
```

Figure 1: Explanations for a sample from the VulDeePecker dataset using both white-box (top) and black-box (bottom) methods.

or brackets after function calls, that are irrelevant for identifying security vulnerabilities. In each of the four deep learning systems, we uncover similar features that are unrelated to security but strongly contribute to the predictions. As a consequence, we argue that methods for explanations need to become an integral part of learning-based security systems—first, for understanding the decisions generated by neural networks and, second, for identifying and eliminating artifacts in the learning process.

The rest of this paper is organized as follows: We briefly review the technical background of explainable learning in Section 2 and analyze state-of-the-art explanation methods suitable for security in Section 3. The considered learning-based security systems are presented in Section 4, before we discuss their quantitative and qualitative evaluation in Section 5 and Section 6, respectively. Section 7 concludes the paper.

2 EXPLAINABLE DEEP LEARNING

Neural networks have been used in artificial intelligence and machine learning for over 50 years. Concepts for explaining their decisions, however, have just recently started to be explored with the advance and remarkable success of deep learning in several application domains, such as image recognition [27] and machine translation [45]. In the following, we provide a brief overview of this work and consider two key aspects that are crucial for interpreting the decision of neural networks: the *network architecture* and the *explanation strategy*.

2.1 Network Architectures

Depending on the learning task at hand, different architectures can be used for constructing a neural network, ranging from general-purpose networks to highly specific architectures for image and speech recognition. In the area of security, three of these architectures are prevalent: *multilayer perceptrons*, *convolutional neural networks*, and *recurrent neural networks*. We discuss the design and applications of these architectures in the following (see Figure 2). For a more detailed description of network architectures, we refer the reader to the book by Rojas [36] and the introduction to deep learning by Goodfellow et al. [18].

Multilayer Perceptrons (MLPs). Multilayer perceptrons, also referred to as *feedforward networks*, are a classic and general-purpose network architecture [37]. The network is composed of multiple fully connected layers of neurons, where the first and last layer correspond to the input and output of the network, respectively. MLPs have been successfully applied to a variety of security problems, such as intrusion and malware detection [19, 22]. While MLP architectures are not necessarily complex, explaining the contribution of individual features is still difficult, as several neurons impact the decision when passing through the layers.

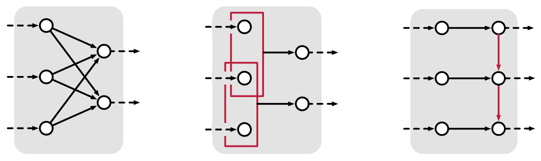
Convolutional Neural Networks (CNNs). These neural networks share a similar architecture with MLPs, yet they differ in the concept of *convolution* and *pooling* [28]. The neurons in convolutional layers receive input only from a local neighborhood of the previous layer. These neighborhoods overlap and create receptive fields that provide a powerful primitive for identifying spatial structure in images and data. CNNs have thus been used for detecting Android malware directly from raw Dalvik bytecode [32]. Due to the convolution and pooling layers, however, it is hard to explain the decisions of a CNN, as its output needs to be "unfolded" for understanding the impact of different features.

Recurrent Neural Networks (RNNs). Recurrent networks, such as LSTM and GRU networks [9, 21], are characterized by a recurrent structure, that is, some neurons are connected in a loop. This structure enables storing and processing information across predictions and enables RNNs to operate on sequences of data [14]. As a result, RNNs have been successfully applied in security tasks involving sequential data, such as the recognition of functions in native program code [10, 39] or the discovery of vulnerabilities in software [29]. Interpreting the prediction of an RNN is even more difficult, as the relevance of an input depends on the network architecture and the sequence of previous inputs.

2.2 Explanation Strategies

The success of deep learning in several application domains has initiated the development of strategies for better understanding and interpreting their decision process. As part of this research effort, several methods have been proposed that aim at explaining individual predictions of neural networks [e.g., 5, 20, 35, 44, 48]. We focus on this form of explainable learning that can be formally defined as follows:

Explainable learning. Given a neural network N, an input vector $x = (x_1, \ldots, x_d)$ and a prediction $f_N(x) = y$, one aims at finding an explanation *why* the label y has been selected by the network. This explanation is typically represented as a vector $r = (r_1, \ldots, r_d)$ that describes the *relevance* or *importance* of the different dimensions of x for the prediction. The computed relevance values can be overlayed with the input and thus enable highlighting relevant features, such as the tokens in the code snippet shown in Figure 1.



(a) MLP layer (b) CNN layer (c) RNN layer

Figure 2: Overview of common network architectures in security applications: Multilayer perceptrons (MLP), convolutional neural networks (CNN), and recurrent neural networks (RNN).

While there exist several different techniques for determining the relevance vector for a prediction, most methods can be categorized into two classes—*black-box* and *white-box*—depending on the underlying explanation strategy.

Black-box Explanations. These methods operate under a black-box setting that assumes no knowledge about the neural network and its parameters. Black-box methods are an effective tool if no access to the neural network is available, for example, when a learning service is audited remotely. Technically, black-box methods rest on an approximation of the function f_N which enables them to estimate how the dimensions of x contribute to a prediction. Although black-box methods are a promising approach for explaining deep learning systems, they can be impaired by the black-box setting and omit valuable information provided through the network architecture and parameters.

White-box Explanations. White-box approaches operate under the assumption that all parameters of a neural network are known and can be used for determining an explanation. As a result, these methods do not rely on approximations but can directly compute explanations for the function f_N on the structure of the network. In practice, predictions and explanations are often computed from within the same system, such that the neural network is readily available for generating explanations. This is usually the case for stand-alone systems for malware detection, binary analysis, and vulnerability discovery. However, several white-box methods are designed for specific network layouts and not applicable to all considered architectures [e.g., 41, 43, 48].
The authors of LEMNA argue that black-box approaches are a preferable alternative to white-box methods in the domain of computer security. In this paper, we challenge this argumentation and demonstrate that several state-of-the-art methods for white-box explanations can be applied in security applications and outperform black-box approaches.

2.3 Relation to Other Concepts

Explainable learning shares similarities with adversarial learning and feature selection. Although methods for generating adversarial examples or selecting features often rest on similar techniques, they pursue different goals and hence cannot be directly applied for explaining the decisions of neural networks. To clarify these difference, we briefly review the objectives of both concepts.

Adversarial Learning. Given a neural network N, an input vector $x = (x_1, \ldots, x_d)$ and a prediction $f_N(x) = y$, adversarial learning is concerned with the task of finding a minimal perturbation δ such that $f_N(x + \delta) \neq f_n(x)$ [7, 34]. The perturbation δ includes the information which features need to be *modified* in order to change the result of the classification which is not directly explaining *why* x was given the label y by N. However, the Gradient explanation method, as described in Section 3, is closely related to this approach as the gradient $\partial f_N / \partial x_i$ quantifies the difference of f_N when changing a feature x_i slightly.

Feature Selection. Feature selection aims at reducting the dimensionality of a learning problem by selecting a subset of discriminative features [13]. While the selected features can be investigated and often capture characteristics of the underlaying data, they are determined independent from a particular learning model. As a result, feature selection methods cannot be direclty applied for explaining the decision of a neural network.

3 EXPLANATION METHODS

Over the last years, several methods have been proposed for explaining the decision of neural networks. Table 1 provides an overview of recent explanation methods. While all address the same objective, they differ in technical properties that are relevant in security applications. For example, some methods do not provide unique solutions when computing an explanation, whereas others cannot be applied to all possible inputs. In the following, we briefly introduce each method and discuss these properties. An in-depth analysis of the properties is carried out in Section 5.

Note that although white-box methods dominate Table 1, to the best of our knowledge, none of these methods have been applied and evaluated in the context of security so far.

Gradients and Integrated Gradients. One of the first white-box methods to compute explanations for neural networks has been introduced by Simonyan et al. [41] and is based on simple gradients. The output of the method is given by $r_i = \partial y / \partial x_i$ which the authors call a *saliency map*. Here r_i measures how much y changes with respect to x_i. Sundararajan et al. [44] extend this approach and propose Integrated Gradients (IG), that use a baseline x', for instance a vector of zeros, and calculate the shortest path from x' to x, given by $x - x'$. To compute the relevance of x_i, the gradients wrt. x_i are cumulated along this path yielding

$$r_i = (x_i - x_i') \int_0^1 \frac{\partial f_N(x' + \alpha(x - x'))}{\partial x_i} \mathrm{d}\alpha.$$

Both gradient-based methods can be applied to all relevant network architectures and thus are considered in our comparative evaluation of explanation methods.

Layer-wise Relevance Propagation and DeepLift. These two methods determine the relevance of a prediction by performing a *backward pass* through the neural network, starting at the output layer and performing calculations until the input layer is reached. The central idea of layer-wise relevance propagation (LRP) is the use of a *conservation property*, that needs to hold true during the backward pass. If r_i^l is the relevance of unit i in layer l then

$$\sum_i r_i^1 = \sum_i r_i^2 = \cdots = \sum_i r_i^L$$

needs to be true for all L layers in the network. Bach et al. [5] further refine this property with the specific ϵ-rule and z-rule. DeepLift also performs a backward pass but takes a reference activation $y' = f_N(x')$ of a reference input x' into account. The method forces the conservation law,

$$\sum_i r_i = y - y' = \Delta y\,,$$

that is, the relevance assigned to the features must sum up to the *difference between* the outcome of x and x'. Both approaches support explaining the decisions of feed-forward, convolutional and recurrent neural networks. Moreover, Ancona et al. [2] show that IG, DeepLift and LRP's z-rule are closely related. We thus use the

Table 1: Overview of white-box (□) and black-box (■) explanation methods. The columns remaining list properties relevant for security applications. Methods considered in our evaluation are marked in bold face.

Explanation Method	Strategy	Unique solution	Support for RNNs	Support for batches	Complete
Gradient [41], **Integrated Gradients [44]**	□	✓	✓	✓	✓
LRP [5], DeepLift [40]	□	✓	✓	✓	✓
PatternNet/ PatternAttribution [23]	□	✓	–	✓	✓
DeConvNet [41, 48], GuidedBackprop [43]	□	✓	–	✓	✓
CAM [49], GradCAM [38], GradCAM++ [8]	□	✓	–	✓	✓
RTIS [11], MASK [16]	□	✓	–	–	✓
LIME [35], **KernelSHAP [30]**	■	–	✓	–	–
LEMNA [20]	■	–	✓	–	–

ϵ-rule of LRP for our experiments in order to evaluate a variety of different approaches.

PatternNet and PatternAttribution. These white-box methods are inspired by the explanation of linear models. While PatternNet determines gradients and replaces neural network weights by so-called *informative directions*, PatternAttribution builds on the LRP framework and computes explanations relative to so-called root points whose output are 0. Both approaches are restricted to feed-forward and convolutional networks. Recurrent neural networks, in turn, are not supported. Consequently, we do not consider these explanation methods in our evaluation.

DeConvNet and GuidedBackProp. These methods aim at reconstructing an input x given output y, that is, mapping y back to the input space. To this end, the authors present an approach to revert the computations of a convolutional layer followed by a rectified linear unit (ReLu) and max-pooling, which is the essential sequence of layers in neural networks for image classification. Similar to LRP and DeepLift, both methods perform a backwards pass through the network. The major drawback of these methods is the restriction to convolutional neural networks. Both methods are thus not considered in our evaluation.

CAM, GradCAM, and GradCAM++. These three white-box methods compute relevance scores by accessing the output of the last convolutional layer in a CNN and performing global average pooling. Given the activations a_{ki} of the k-th channel at unit i, GradCam learn weights w_k such that

$$y \approx \sum_i \sum_k w_k a_{ki}.$$

That is, the classification is modeled as a linear combination of the activations of the last layer of all channels and finally $r_i = \sum_k w_k a_{ki}$. GradCam and GradCam++ extend this approach by including specific gradients in this calculation. All three methods are only applicable if the neural network uses a convolutional layer as the final layer. While this setting is common in image recognition, it is rarely used in security applications and thus we do not study these methods in our evaluation. Moreover, these methods do not handle recurrent neural networks.

RTIS and MASK. These methods compute relevance scores by solving an optimization problem for a *mask m*. A mask m is applied to x as $m \circ x$ in order to affect x, for example by setting features to zero. To this end, Fong and Vedaldi [16] propose the optimization problem

$$m^* = \arg\min_{m \in [0,1]^d} \lambda \|\mathbf{1} - m\|_1 + f_N(m \circ x),$$

which determines a sparse mask, that identifies relevant features of x. This can be solved using gradient descent, which thus makes these white-box approaches. However, solving the equation above often leads to noisy results which is why RTIS and MASK add additional terms to achieve smooth solutions using regularization and blurring. These concepts, however, are only applicable for images and cannot be transferred to other types of features. As a consequence, we do not consider these white-box methods in the evaluation.

LIME and KernelSHAP. Ribeiro et al. [35] has proposed one of the first black-box methods for explaining neural networks and Lundberg and Lee [30] further extend this approach. Both methods aim at approximating the decision function f_N by creating a series of l perturbations of x, denoted as $\tilde{x}_1, \ldots, \tilde{x}_l$ by setting entries in the vector x to 0 randomly. The methods then proceed by predicting a label $f_N(\tilde{x}_i) = \tilde{y}_i$ for each $\tilde{x}_i$ of the l perturbations. This sampling strategy enables the methods to approximate the local neighborhood of f_N at the point $f_N(x)$. LIME approximates the decision boundary by a weighted linear regression model,

$$\arg\min_{g \in \mathcal{G}} \sum_{i=1}^{l} \pi_x(\tilde{x}_i)\big(f_N(\tilde{x}_i) - g(\tilde{x}_i)\big)^2,$$

where $\mathcal{G}$ is the set of all linear functions and π_x is a function indicating the difference between the input x and a perturbation $\tilde{x}$. KernelSHAP follows the same approach but additionally applies concepts of game theory for the selection of relevant features. As both approaches can be applied to all considered network architectures, we study them in our empirical evaluation.

LEMNA. This black-box method, proposed by Guo et al. [20], has been specifically designed for security applications. It uses a *mixture regression model* for approximation, that is, a weighted sum of linear models used for perturbations that takes the form

$$f(x) = \sum_{j=1}^{K} \pi_j(\beta_j \cdot x + \epsilon_j).$$

The parameter K specifies the number of models, the random variables $\epsilon = (\epsilon_1, \ldots, \epsilon_K)$ originate from a normal distribution $\epsilon_i \sim N(0, \sigma)$ and $\pi = (\pi_1, \ldots, \pi_K)$ holds the weights for each model. Variables $\beta_1, \ldots, \beta_K$ are the regression coefficients and can be interpreted as k linear approximations of the decision boundary near $f_N(x)$.

A Toy Example. As an illustrative example for the differences between black-box and white-box methods, Figure 3 shows explanations for a simple feed-forward network trained on the well-known MNIST dataset. We compare explanations of LEMNA with LRP where we use $K = 1$ and $S = 10^4$ for LEMNA and the ϵ-rule with $\epsilon = 10^{-3}$ for LRP. Moreover, we randomly flip blocks of size 2×2 to create perturbations for LEMNA.

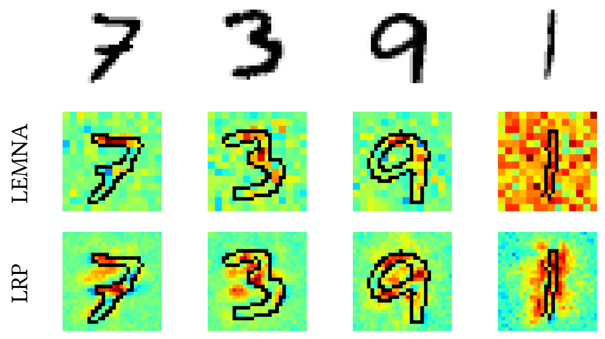


Figure 3: Illustration of explanations for LEMNA (second row) and LRP (third row). Red indicates relevances towards the decision of the neural network, blue indicates relevances against the decision, and green indicates no relevance.

Even in this toy example the explanations show notable differences. The leftmost column is classified as 7 and LEMNA explains this decision with the top bar of the seven. This makes sense, as it is likely that perturbations of the picture which do not have this horizontal bar are classified as 1. LRP, however, also puts relevance on the free space underneath the top bar and thereby indicates that this free space is important so that the image is not classified as the digit 9. By design, LEMNA is unable to mark this area, as its perturbations are generated by setting features from 1 to 0, which is insufficient to transform a 7 to a 9. Similar observations hold true for the second and third columns.

The rightmost picture depicts a limitation of black-box methods. The image is classified as 1, but when perturbing the pixels by setting them to 0, the label does not change. As a consequence, LEMNA cannot find a correlation between 1 and other digits and produces a random explanation. LRP, in this case, highlights the free areas left and right to the digit 1 as having the greatest impact.

4 SYSTEMS UNDER TEST

For our empirical evaluation, we consider four security systems that have been recently proposed and employ deep learning techniques. The systems cover the three major architectures introduced in Section 2.1 and comprise between 4 to 6 layers of different types. An overview of the systems is provided in Table 2.

Table 2: Overview of the consideredd security systems.

System	NN Implementation	Arch.	# Layers
Drebin+	ESORICS'17 [19]	MLP	4
Mimicus+	CCS'18 [20]	MLP	4
DAMD	CODASPY'17 [32]	CNN	6
VulDeePecker	NDSS'18 [29]	RNN	5

System 1 — Drebin+. The first system that we call Drebin+ uses a multilayer perceptron for identifying Android malware. The system has been proposed by Grosse et al. [19] and builds on features originally developed by Arp et al. [3]. The network consists of two hidden layers, each comprising 200 neurons. The input features are statically extracted from Android applications and cover data from the application's manifest, such as hardware details, requested permissions, and filtered intents, as well as information based on the application's code, such as the used permissions, suspicious API calls, and network addresses.

To verify the correctness of our implementation, we train the system on the original Drebin dataset [3], where we use 75 % of the 129,013 Android application for training and 25 % for testing. Table 3 shows the results of this experiment which match the performance numbers published by Grosse et al. [19].

System 2 — Mimicus+ The second system that we denote as Mimicus+ also uses a multilayer perceptron and is capable of detecting malicious PDF documents. The systems is re-implemented based on the work of Guo et al. [20] and builds on features originally introduced by Smutz and Stavrou [42]. Our implementation uses two hidden layers with 200 nodes each and is trained with 135 features extracted from PDF documents. These features cover properties about the document structure, such as the number of sections and fonts in the document, and are mapped to binary values as described by Guo et al. [20]. For a full list of features we refer the reader to the implementation by Šrndić and Laskov [46].

For verifying our implementation, we make use of the original dataset that contains 5,000 benign and 5,000 malicious PDF files and again split the dataset into 75 % for training and 25 % for testing. Our results are shown in Table 3 and come close to a perfect detection.

System 3 — DAMD The third security system studied in our evaluation is called DAMD and uses a CNN for identifying malicious Android applications [32]. The system processes the raw Dalvik bytecode of Android applications and its network is comprised of six layers for embedding, convolution and max-pooling of the extracted instructions. As the system processes entire applications, the number of features depends on the size and structure of the applications. For a detailed description of this process, we refer the reader to the original publication by McLaughlin et al. [32].

As a reference, we apply the system to data from the Malware Genome Project [50]. This dataset consists of 2,123 applications in total, with 863 benign and 1,260 malicious samples. Strictly speaking, this is a subset of the Drebin dataset [3]. However, in order to replicate the results of McLaughlin et al. [32], we choose to use the same data. We again split the dataset into 75 % of training and 25 % of testing data and obtain results similar to those presented in the original publication, as shown in Table 3.

System 4 — VulDeePecker The fourth system is denoted as VulDeePecker and uses a recurrent neural network (RNN) for discovering vulnerabilities in source code [29]. The network comprises five layers, uses 300 LSTM cells [21], and applies a word2vec embedding [33] with 200 dimensions for analyzing C/C++ code. As a preprocessing step, the source code is sliced into code gadgets that comprise short snippets of lexical tokens. The gadgets are truncated or padded to a length of 50 tokens. To avoid overfitting, identifiers and symbols are substituted with generic placeholders.

For verifying the correctness of our implementation, we use the CWE-119 dataset, which consists of 39,757 code gadgets, with 10,444 gadgets corresponding to vulnerabilities. In line with the original study, we split the dataset into 80 % training and 20 % testing data. In our experiments, we yield an accuracy of 0.908 similar to the original publication.

Table 3: Performance of our re-implementations of the security systems on the original datasets.

System	Accuracy	Precision	Recall	F1-Score
Drebin+	0.980	0.926	0.924	0.925
Mimicus+	0.994	0.991	0.998	0.994
DAMD	0.949	0.967	0.924	0.953
VulDeePecker	0.908	0.837	0.802	0.819

The four selected security systems provide a broad view on the current use of deep learning in security. Drebin+ and Mimicus+ are examples of systems that make use of MLPs for detecting malware. They differ, however, in the dimensionality of the input, where Mimicus+ works on a small set of engineered features and Drebin+ analyzes inputs with thousands of dimensions. DAMD is an example of a system using a CNN in security and capable of learning from inputs of variable length, whereas VulDeePecker makes use of an RNN, similar to other learning-based approaches analyzing program code [e.g., 10, 39, 47].

5 QUANTITATIVE EVALUATION

To carve out the differences of black-box and white-box methods, we start with a quantitative evaluation of the six methods. As the practical efficacy of an explanation depends on different factors, we introduce the following four performance measures for our evaluation:

(1) *Conciseness.* An explanation is concise if the top-ranked features play a crucial role in the prediction of the neural network. We thus measure conciseness by successively removing relevant features from a sample and monitoring the impact on the prediction (Section 5.1).
(2) *Sparsity.* An explanation needs to be sparse for a human to be understandable. Neural networks can operate on vectors with thousands of dimensions and thus we investigate how well an explanation method can condense its result without loosing effectivity (Section 5.2).
(3) *Completeness.* An explanation method must be able to create results for all data that is potentially interesting to a user. We measure this property by examining cases in which the methods may not generate results and compute a ratio of incomplete explanations (Section 5.3).
(4) *Efficiency.* Finally, an explanation needs to be computed in a reasonable amount of time. While low run-time is not a strict requirement in practice, time differences between minutes and milliseconds are still important and thus considered in our evaluation (Section 5.4).

For our evaluation, we implement LEMNA in accordance to Guo et al. [20] and use the Python package `cvxpy` [12] to solve the linear regression problem with fused lasso restriction. We set the number of mixture models to $K = 3$ and the number of perturbations to $l = 500$. The parameter S is set to 10^4 for Drebin+ and Mimicus+, as the underlying features are not sequential, and to 10^{-3} for the sequences of DAMD and VulDeePecker, according to Guo et al. [20]. We implement LIME with $l = 500$ perturbations, cosine similarity for proximity measure and use the regression solver from the `scipy` package with L^1 regularization. For KernelSHAP we make use of the open source implementation from Lundberg and Lee [30][1].

To generate the explanations for LRP, Gradient and IG we make use of the `iNNvestigate` toolbox by Alber et al. [1], except for the VulDeePecker model for which we apply the RNN implementation from Arras et al. [4] for LRP. For all experiments we set $\epsilon = 10^{-3}$ and use $N = 64$ steps for IG, except for VulDeePecker where we use $N = 256$.

5.1 Conciseness of Explanations

We start by investigating the conciseness of the explanations generated for the four considered security systems. To this end, we stick to the approach by Guo et al. [20] for measuring the impact of removing features on the classification result. In particular, we compute the *average remaining accuracy* (ARA). For some $k = 1, \ldots, F$ the ARA is calculated by removing the k most relevant features from a sample and running it through the neural network again. This procedure is performed on all samples and the average softmax probability of the original class of the samples is reported.

How is the ARA expected to behave? If we successively remove relevant features, the ARA will decrease, as the neural network has less information for making a correct prediction. The better the explanation, the quicker the ARA will drop as more relevant information has been removed. In the long term, ARA converges to the probability score of the sample with no information, i.e. containing only zeros.

Technically, we implement ARA by removing the specific features of the four systems under test as follows. For Drebin+ and Mimicus+, features are removed by setting the features to 0. For DAMD we replace the top instructions with the no-op byte code and for VulDeePecker we simply replace the top lexical tokens from the code with an embedding of only zeros. Moreover, we introduce a Brute-Force method as a baseline for this experiment. This method calculates the relevance r_i by setting x_i to zero and measuring the difference in the softmax probability, i.e. $r_i = f_N(x) - f_N(x|x_i = 0)$. We call this method Brute-Force because a sample with d features has to be classified d times again, which can be time consuming for data with lots of features.

Evaluation Results: ARA Figure 4 (top row) presents the results of our evaluation for the four considered security systems. For

[1] https://github.com/slundberg/shap

the ARA curves shown in the upper row, we observe that white-box methods (green) consistently perform better than black-box methods (red) on all systems. VulDeePecker is the only dataset where LIME can obtain results similar to IG and LRP. Notably, for the DAMD dataset, IG and LRP are the only methods to generate real impact on the average accuracy. For Mimicus+, white-box methods achieve a perfect ARA of 0 after only 25 features.

In addition to Figure 4, Table 4a shows the *area under curve* (AUC) for the ARA curves. A lower AUC indicates better explanations and we observe that white-box methods are 30 % better than black-box methods on average over all systems and methods. Also, white-box methods always achieve results close to or even better than the Brute-Force approach.

5.2 Sparsity of Explanations

We continue to compare the two explanation methods and investigate the sparsity of the generated explanations. To this end, we normalize the explanations of all methods by dividing the relevance vectors by the maximum of their absolute values so that the values for every sample are between -1 and 1. While this normalization helps a human analyst to identify top-ranked features, the sheer amount of data in an explanation can still render an interpretation intractable. As a consequence, we expect a useful explanation method to assign high relevance scores to only a few features and keep the majority at a relevance of 0. To measure the sparsity we create a normalized histogram h of the relevance values to access the distribution of the values and calculate the *mass around zero* (MAZ) metric defined by $MAZ(r) = \int_{-r}^{r} h(x)dx$ for $r \in [0, 1]$. Sparse explanations that assign many features with relevances close to 0 will have a steep rise in MAZ at r close to zero and a flat slope when r is close to 1 since only few features are assigned high relevance.

Evaluation results. The second row in Figure 4 shows the the development of MAZ for the datasets and methods. We observe that white-box methods assign significantly more features a value close to zero, indicating that they are not relevant. This stands in contrast to black-box methods which assign relevance values to features with a broader range around zero resulting in flatter slopes of the MAZ close to 0.

LEMNA's relevance distribution for VulDeePecker is showing a steep rise close to 1 indicating that it assigns a lot of tokens a high relevance which is undesirable but results from the fused lasso constraint in the concept of LEMNA. For the DAMD network, we can see that IG has a massive peak at 0 showing that it marks almost all features as meaningless while, according to the experiments on the ARA, still choosing effective ones. This is particularly advantageous for human experts since the DAMD dataset contains up to 520,000 features per sample and investigating thousands of relevant bytes is tedious work.

We summarize the performance on the MAZ metric by calculating the *area under curve* again in table Table 4b. A higher AUC indicates that more features were assigned a relevance close to 0 and few features were assigned a relevance close to ±1. On average over all datasets and methods, we find that white-box methods are 19 % sparser compared to black-box methods.

Table 4: Conciseness and sparsity of explanations for different methods. Area under ARA curve when (a) removing up to 40 most important features and (b) calculating MAZ for histograms of the relevances.

(a) Area under curve for ARA curves from Figure 4.

Method	Drebin+	Mimicus+	DAMD	VulDeePecker
LIME	**0.5796**	**0.2569**	**0.9187**	**0.5714**
LEMNA	0.6560	0.4050	0.9827	0.7635
KernelSHAP	0.8909	0.5650	0.9661	0.8689
Gradient	0.4720	0.2126	0.8575	0.8557
IG	**0.4461**	**0.2059**	**0.4985**	**0.5737**
LRP	0.4743	0.2127	0.5035	0.6250
Brute-Force	0.4742	0.2079	0.5178	0.5682

(b) Area under curve for MAZ curves from Figure 4.

Method	Drebin+	Mimicus+	DAMD	VulDeePecker
LIME	0.7569	**0.7523**	**0.8326**	0.7445
LEMNA	0.6806	0.7271	0.6246	0.4160
KernelSHAP	**0.7832**	0.7164	0.7133	**0.8132**
Gradient	0.8459	0.8562	0.9487	0.8159
IG	**0.8473**	**0.8575**	**0.9983**	**0.8390**
LRP	0.8459	0.8561	0.9641	0.8271
Brute-Force	0.8487	0.8573	0.9979	0.8460

5.3 Completeness of Explanations

We additionally examine the completeness of explanations. The white-box methods we analyze can provide explanations for all samples that are processed by a neural network since they perform deterministic computation steps to calculate relevances, thus we do not analyze white-box methods in this section. For LIME, LEMNA and KernelSHAP the situation is different: In order to generate an explanation, it is necessary to create perturbations from the actual sample by setting features to zero at random and classify these perturbations with the neural network. In this setting, it is important that the perturbations have labels from the opposite class as otherwise none of these methods have a possibility to find relevant features for the classification.

During our experiments with 500 perturbations we find that the analyzed black-box methods produce meaningful results if at least 20 of the perturbations (about 4 % on average) come from the opposite class. We thus analyze how often we find this amount of perturbations for the four considered security systems.

Evaluation Results. Table 5 provides results on the problem of incompleteness. On average 29 % of the samples cannot be explained well, as the computed perturbations contain too few instances from the opposite class. In particular, we observe that creating malicious perturbations from benign samples is a severe problem for Drebin+ and DAMD, where only 32.6 % and 2.8 %, respectively, of the benign samples achieve sufficient malicious perturbations.

This problem arises from the fact that only few features make a sample malicious, whereas there exists a large variety of benign features. As a consequence, setting malicious features to 0 for a

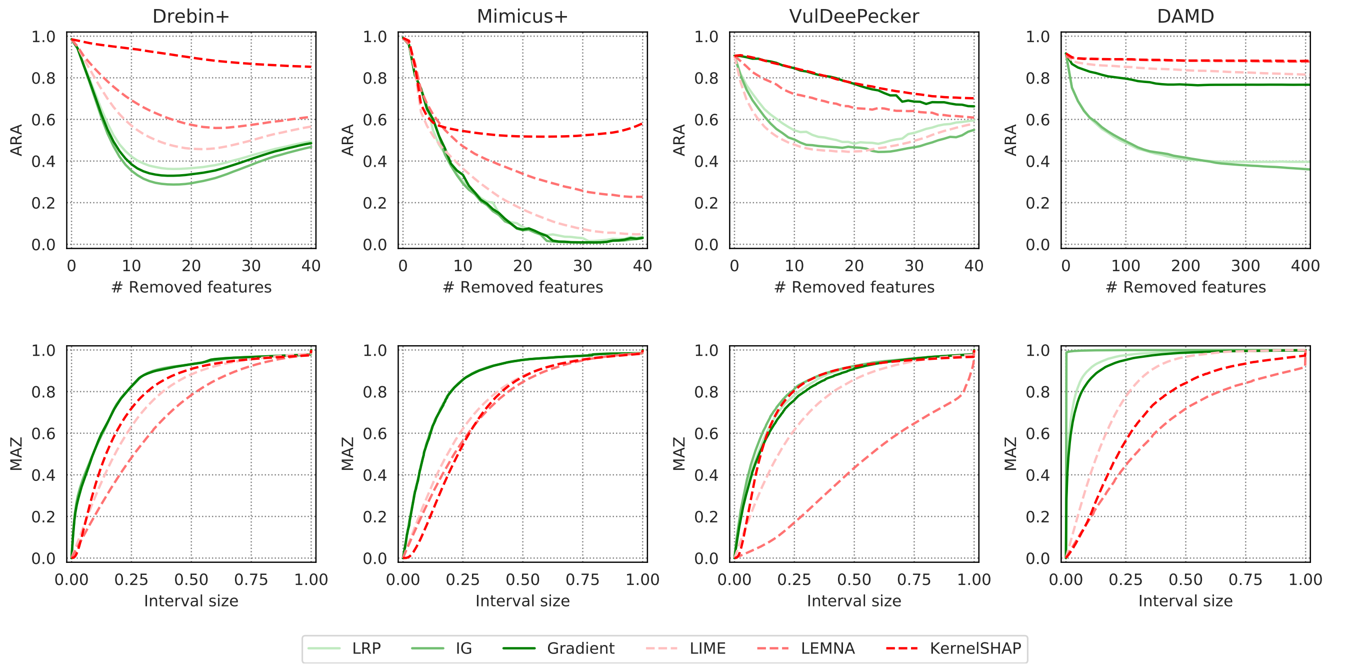


Figure 4: Conciseness of explanations for black-box and white-box methods. Top row: ARA when removing relevant features; bottom row: MAZ for histograms of relevance values.

Table 5: Incomplete explanations of black-box methods. First two columns: samples remaining when enforcing at least 4 % perturbations of opposite class, last column: incomplete explanations.

System	Class +	Class −	Incomplete
Drebin+	32.6 %	99.1 %	58.6 %
Mimicus+	87.8 %	99.5 %	6.1 %
DAMD	2.8 %	97.2 %	44.7 %
VulDeePecker	95.0 %	95.8 %	7.5 %
Average	**54.6 %**	**97.9 %**	**29.2 %**

perturbation usually leads to a benign classification. Setting benign features to zero, however, often does not impact the classification result. For example, the smallest byte-sequence in the DAMD dataset is eight bytes long and classified as benign. No matter which feature-combinations are set to zero for the perturbations, the sample will never be classified as malicious by the network.

Table 6: Run-time per explanation of the different explanation methods.

Method	Drebin+	Mimicus+	DAMD	VulDeePecker
LIME	**30.54 ms**	**28.32 ms**	**736.21 ms**	**30.04 ms**
LEMNA	4.59 s	2.56 s	685.90 s	6.14 s
KernelSHAP	9.05 s	424.80 ms	44.55 s	4.95 s
Gradient	**4.43 ms**	**0.07 ms**	26.77 ms	**1.21 ms**
IG	134.00 ms	0.79 ms	26.70 ms	**1.01 ms**
LRP	**4.34 ms**	**0.08 ms**	**15.29 ms**	254.55 ms
Brute-Force	621.12 ms	202.84 ms	517.18 s	210.97 ms

5.4 Efficiency of Explanations

When dealing with large amounts of data, it might be desirable for the user to create explanations for every sample to identify features that are important for an entire class. We therefore compare the run-time of white-box and black-box methods by processing all datasets and measuring the average run-time per sample. Since black-box methods are non-deterministic and the run-time depends on the initialization of the parameters, we repeat the black-box algorithms 10 times for every sample and use the median value of the 10 runs for the calculation of the mean run-time.

Evaluation Results. Table 6 shows the average run-time for generating explanations for the considered systems. We observe that white-box methods are at least an order of magnitude faster than black-box methods on all datasets. This advantage arises from the fact that data can be processed *batch-wise* for white-box methods, that is, explanations for a set of samples can be calculated at the same time. The Mimicus+ dataset, for example, can be processed in one batch resulting in a speed-up factor of more than 400 over the fastest black-box method LIME. The runtime of the black-box methods increases for high dimensional datasets, especially DAMD, since the regression problems for them have to be solved in higher dimensions. While the speed-up factors are already enormous, we have not even included the creation of perturbations and their classification for the black-box algorithms, which require additional run-time as well.

Table 7: Mean results of the explanation methods wrt. the four different metrics. The last column summarizes these metrics in a rating comprising three levels: strong (●), medium (◐), and weak (○).

Explanation Method	Conciseness	Sparsity	Completeness	Efficiency	Overall Rating
LIME	0.5817	0.7716	–	2.06×10^{-1} s	◐ ◐ ○ ◐
LEMNA	0.7018	0.6121	–	1.75×10^{2} s	○ ○ ○ ○
KernelSHAP	0.8227	0.7565	–	1.47×10^{1} s	○ ◐ ○ ○
Gradient	0.5999	0.8667	✓	1.26×10^{-2} s	◐ ● ● ●
IG	0.4311	0.8855	✓	4.06×10^{-2} s	● ● ● ●
LRP	0.4539	0.8733	✓	6.92×10^{-2} s	● ● ● ●
Brute-Force	0.4420	0.8875	✓	1.30×10^{2} s	● ● ● ○

5.5 Summary

We evaluated the four metrics conciseness, sparsity, completeness and efficiency in the last four subsections. However, a good explanation method should achieve good results in each metric and on every dataset. For example, we have seen that the Gradient method computes sparse results in a decent amount of time but the features are not concise on the DAMD and VulDeePecker dataset. Equally, the relevances of KernelSHAP for the Drebin+ dataset are sparser than those from LEMNA but less concise at the same time. We summarize the results of all explanations methods on the four data-sets by computing the mean of the four metrics to enable a comparison of all methods.

Evaluation Results. Table 7 Shows the average performance of the methods for the four metrics. For each metric we assign each method one of the categories ●, ◐, and ○. The ● category is assigned to the best method and all other metrics that lie in a decent area around the best method. The ○ category is assigned to the worst method and methods close to it. Finally, the ◐ category is assigned to methods that lie between the best and worst methods.

We see that only white-box methods achieve a ● ranking in all metrics. They can compute results in less than 70 ms, mark only few features as relevant and those features have great impact on the decision of the classifier.

6 QUALITATIVE EVALUATION

Finally, we turn to a qualitative analysis of white-box and black-box explanations. To this end, we visualize and discuss explanations for the four security systems in detail. As a result of Section 5 we focus on the results from IG and LIME since these are the best white-box and black-box methods respectively.

To visualize the relevance vector of an explanation, we normalize the scores to −1 and 1, and highlight features according to whether they support the decision (green color) or contradict the decision (red color). The brightness reflects the importance of the features.

6.1 Drebin+

We start the qualitative analysis with an Android application from the Drebin dataset, which the neural network correctly classifies as malware. The considered sample[2] belongs to the *FakeInstaller* family [31], whose members are known to steal money from smartphone users by secretly sending text messages (SMS) to premium services. In order to trick users into installing this kind of malware, the malicious code is often repackaged into legitimate applications. Table 8 depicts the explanations from IG and LIME side by side.

Explanations. The considered malware sample exhibits low complexity, allowing us to present all extracted features along with their relevance scores in Table 8. Both methods rate features as highly relevant (highlighted in green), that can be linked to the SMS functionality of the device. For instance, the requested permission `SEND_SMS` is pointed out by both methods. Moreover, features related to accessing sensitive information, such as the `READ_PHONE_STATE` permission and the API call `getSimCountryIso` have also obtained high scores. In line with the malware's objective, the accentuation of these features seems valid.

Nonetheless, the explanations also bring features to light that contradict the neural network's classification as malware (highlighted in red). Both methods assign high relevance scores to the hardware feature `touchscreen`, the `launcher` intent filter, and the (used) `INTERNET` permission. However, these features generally occur in most Android applications and are thus not particularly descriptive for benignity. Naturally, the interpretation of features speaking for benign application is more challenging due to the broader scope and the difficulty in defining benignity. In the worst case, however, the missing context of the selected features may indicate that the underlying classifier considers artifacts of the data at hand rather than learning the underlying task.

Summary. Overall, we find that IG and LIME both provide similar results when explaining the decisions of the neural network employed by Drebin+. We therefore conclude that both methods are on par with each other in this setting. While they often point to features that are helpful to identify characteristics of malicious applications, features speaking for benignity are less specific and even suggest that artifacts might have been learned.

6.2 Mimicus+

Next, we discuss the results of both explanation methods for a malicious PDF document[3] from the Mimicus+ dataset. The document tries to exploit vulnerabilities in the JavaScript engine to download and install additional malicious code to the system [15].

Explanations. Inspecting the explanations of IG and LIME for Mimicus+, we observe that the features for classification of malware are dominated by the features `count_javascript` and `count_js` which both stand for the number of JavaScript object markers in

[2]MD5: `7c5d7667c0734b2faf9abd68a882b146`

[3]MD5: `33539f91407615622618b83c453f9541`

Table 8: Explanations for the Android malware FakeInstaller generated for Drebin+ using IG and LIME.

Id	IG	LIME
0	permission::android.permission.SEND_SMS	permission::android.permission.SEND_SMS
1	activity::.FirstActivity	call::sendSMS
2	call::sendSMS	activity::.FirstActivity
3	permission::android.permission.READ_PHONE_STATE	feature::android.hardware.telephony
4	feature::android.hardware.telephony	intent::android.intent.action.DATA_SMS_RECEIVED
5	intent::android.intent.action.DATA_SMS_RECEIVED	permission::android.permission.INTERNET
6	service_receiver::.services.SMSSenderService	permission::android.permission.READ_PHONE_STATE
7	service_receiver::.sms.BinarySMSReceiver	service_receiver::.services.SMSSenderService
8	permission::android.permission.INTERNET	call::getSimCountryIso
9	call::getSimCountryIso	api_call::android/../TelephonyManager;->getLine1Number
10	api_call::android/../TelephonyManager;->getLine1Number	api_call::android/app/Activity;->startActivity
11	api_call::android/app/Activity;->startActivity	service_receiver::.sms.BinarySMSReceiver
12	permission::android.permission.RECEIVE_SMS	real_permission::android.permission.SEND_SMS
13	real_permission::android.permission.SEND_SMS	api_call::org/apache/http/impl/client/DefaultHttpClient
14	intent::android.intent.action.MAIN	permission::android.permission.RECEIVE_SMS
15	call::printStackTrace	call::printStackTrace
16	api_call::org/apache/http/impl/client/DefaultHttpClient	api_call::android/telephony/SmsManager;->sendTextMessage
17	api_call::android/telephony/SmsManager;->sendTextMessage	real_permission::android.permission.READ_PHONE_STATE
18	real_permission::android.permission.READ_PHONE_STATE	intent::android.intent.action.MAIN
19	call::getSystemService	call::getSystemService
20	real_permission::android.permission.INTERNET	real_permission::android.permission.INTERNET
21	intent::android.intent.category.LAUNCHER	intent::android.intent.category.LAUNCHER
22	feature::android.hardware.touchscreen	feature::android.hardware.touchscreen

the document. Table 10 shows the explanations of both methods for the selected malicious document. For better representation, we only show the 10 most important features for both classes and refer the reader to Table 12 in the appendix for a full listing.

The coloring of the features confirms observations from the sparsity analysis in Figure 4, that is, IG chooses more features close to zero and only marks a few features with strong relevance. In contrast, LIME distributes the relevance across more features and provides a less concise explanation. The strong impact of JavaScript object markers identified by both method is meaningful, as it is well known that JavaScript is used in malicious PDF documents frequently [26].

We also identify relevant features in the explanations that are non-intuitive, especially since Guo et al. [20] train the model by setting non-zero features to one to make the representation usable for LEMNA. For example, features like `count_trailer` that measures the number of trailer markers in the document or `count_box_letter` that counts the number of US letter sized boxes can be hardly related to security and rather constitute an artifact of the binary conversion.

To investigate which features are most often indicative for benign and malicious samples, we count how often feature appear in the top 10 features of all samples in the dataset and report the 5 most prominent candidates for both classes in Table 9. Furthermore, we specify in which fraction of the samples from the two classes these samples occur in the entire dataset to see whether these features are well chosen by the explanation methods.

For the benign class we find that the features `count_font` (count of font object markers), `producer_mismatch` (count of differences in producer values) and `title_num` (count of numeric characters in title) occur in the top features of both methods and those features are appearing rarely in the malicious samples. LIME finds the `pos_eof_min` (normalized position of last EOF marker) which is frequent in both classes. IG uses the `title_uc` (count of upper case letters in title) features often which is assigned significantly more often in the benign samples. Given these observations it might be easy for malware authors to evade a detection of Mimicus+ by using font object markers in the document and numeric, upper case letters in the author name and document title or avoid the usage of JavaScript.

Summary. LIME and IG both use the count of JavaScript object markers as the most important features for classification. JavaScript

Table 9: Most prominent features from the Mimicus+ dataset from the explanation methods for every class and fraction of appearance of the feature in the benign and malicious class in the entire dataset.

Class	Top 5 Feature	Method	Benign	Malicious
−	count_font	Both	98.4 %	20.8 %
−	producer_mismatch	Both	97.5 %	16.6 %
−	title_num	Both	68.6 %	4.8 %
−	pdfid1_num	Both	81.5 %	2.8 %
−	title_uc	IG	68.6 %	4.8 %
−	pos_eof_min	LIME	100.0 %	93.4 %
+	count_javascript	Both	6.0 %	88.0 %
+	count_js	Both	5.2 %	83.4 %
+	count_trailer	Both	89.3 %	97.7 %
+	pos_page_avg	IG	100.0 %	100.0 %
+	count_endobj	IG	100.0 %	99.6 %
+	createdate_tz	LIME	85.5 %	99.9 %
+	count_action	LIME	16.4 %	73.8 %

appears in 88.0 % of the malicious documents where only about 6 % of the benign samples use it. This makes JavaScript an extremely discriminating feature for the dataset. From a security perspective, however, this is a less satisfying result, as the neural network of Mimicus+ relies on a single indicator for detecting the malicious code in the selected document.

Table 10: Explanations for a malicious PDF document generated for Mimicus+ using IG and LIME.

Id	IG	LIME
0	count_javascript	count_js
1	count_js	count_javascript
2	count_trailer	count_page
3	count_endobj	ratio_size_stream
4	box_other_only	createdate_version_ratio
5	pos_page_avg	size
6	createdate_tz	createdate_tz
7	count_stream	pos_acroform_avg
8	pos_page_min	pos_box_min
9	ratio_size_stream	pos_page_min
10	...	...
34	moddate_tz	pos_page_avg
35	size	pos_image_avg
36	pos_eof_max	pos_eof_avg
37	count_endstream	pos_image_min
38	pos_eof_avg	ratio_size_page
39	moddate_version_ratio	count_eof
40	count_xref	len_stream_max
41	count_eof	moddate_tz
42	pos_eof_min	count_xref
43	len_stream_max	pos_eof_min

6.3 DAMD

We resume with the interpretation of explanations for the Android malware detector DAMD. To this end, we analyze 9 malicious applications from three popular Android malware families: GoldDream [25], DroidKungFu [24], and DroidDream [17]. These families exfiltrate sensitive user data and can install exploits on the device to take over full control.

Explanations. In our analysis of the raw Dalvik bytecode that the neural network processes, the black-box method's tendency of considering large numbers of long sequences becomes apparent. This confirms results from experiments in Section 5. White-box methods, on the contrary, highlight only very few sequences. Consequently, analyzing all "relevant" features of LIME's explanation, unfortunately, is infeasible in practice, while it seems perfectly doable for IG. To further examine whether this reduced selection indeed points to essential characteristics of the malicious samples, we analyze the opcodes with the highest IG scores and find that these indeed are directly linked to the malicious functionality of these applications.

As an example, Table 11 depicts the opcode sequence, that is found in all samples of the GoldDream family[4]. Taking a closer look, the opcode sequence can be identified to only occur in the `onReceive` method of the `com.GoldDream.zj.zjReceiver` class. In this function, the malware intercepts incoming SMS and phone calls and stores the information in local files before sending them to an external server.

Similar results are achieved by IG for the other malware families: members of the DroidDream family are known to root infected device by running different exploits. If the attack has been successful, the malware installs itself as a service with the name `com.android.root.Setting` [17]. The top-ranked features determined by IG indeed lead to two methods of this very service, namely `com.android.root.Setting.getRawResource()` together with `com.android.root.Setting.cpFile()`. Likewise, the highest ranked opcode sequence of the DroidKungFu family point to the class, in which the decryption routine for the root exploit is stored in. Note that for all members of each family, the identified opcode sequences are identical, thus proving that (a) DAMD has learned a crucial characteristic of these families, and (b) IG to the points explains what the neural network has learned.

Table 11: Explanations of IG and LIME for a sample of the GoldDream family from the DAMD dataset. The selected opcodes belong to the *onReceive* method of the malware.

Id	IG	LIME
0	invoke-virtual	invoke-virtual
1	move-result-object	move-result-object
2	if-eqz	if-eqz
3	const-string	const-string
4	invoke-virtual	invoke-virtual
5	move-result-object	move-result-object
6	check-cast	check-cast
7	array-length	array-length
8	new-array	new-array
9	const/4	const/4
10	array-length	array-length
11	if-ge	if-ge
12	aget-object	aget-object
13	check-cast	check-cast

Summary. In all analyzed samples, IG points to only a few but highly relevant features, thus enabling us to understand the decisions made by the classifier. In contrast, LIME marks a larger number of long opcode sequences as relevant, making it difficult to find the relevant places in the opcode sequence.

6.4 VulDeePecker

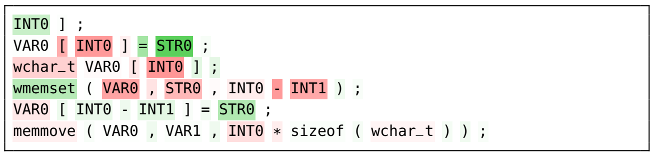
Finally, we turn to the field of vulnerability discovery where we look at explanations for neural network learned by VulDeePecker [29]. In contrast to the previous case studies, the neural network classifies samples in the form of program slices, that is, a sequence of lexical tokens of program code in the order of its execution. Figure 5(a) shows the source code of such a program slice, while Figures 5(b) and 5(c) depict the same piece of code as lexical tokens, including the relevance of each token for IG and LIME, respectively. Note that VulDeePecker by design restricts its inputs to the last 50 tokens in the program slice.

The sample illustrates a buffer overflow originating from the definition of the fixed size buffers on line 2 and 4. Later on, in line 7,

[4]e.g., MD5: a4184a7fcaca52696f3d1c6cf8ce3785

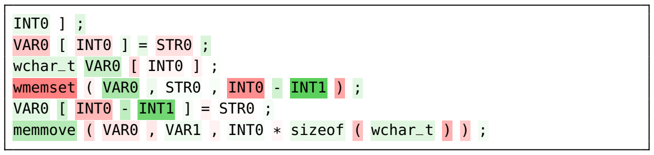
```
data = NULL;
data = new wchar_t[50];
data[0] = L'\\0';
wchar_t source[100];
wmemset(source, L'C', 100-1);
source[100-1] = L'\\0';
memmove(data, source, 100*sizeof(wchar_t));
```

(a) Original code



```
INT0 ] ;
VAR0 [ INT0 ] = STR0 ;
wchar_t VAR0 [ INT0 ] ;
wmemset ( VAR0 , STR0 , INT0 - INT1 ) ;
VAR0 [ INT0 - INT1 ] = STR0 ;
memmove ( VAR0 , VAR1 , INT0 * sizeof ( wchar_t ) ) ;
```

(b) Integrated Gradients



```
INT0 ] ;
VAR0 [ INT0 ] = STR0 ;
wchar_t VAR0 [ INT0 ] ;
wmemset ( VAR0 , STR0 , INT0 - INT1 ) ;
VAR0 [ INT0 - INT1 ] = STR0 ;
memmove ( VAR0 , VAR1 , INT0 * sizeof ( wchar_t ) ) ;
```

(c) LIME

Figure 5: Explanations for a code sample from the VulDeePecker dataset using (a) LEMNA and (b) LRP.

data from the one buffer, `source`, is copied to the other, `data`, using the `memmove` function. The third argument of the function determines the number of bytes that are copied, which is set to the size of 100 wide characters of type `wchar_t`. Consequently, twice as many bytes are moved to the destination buffer than it may contain.

Explanations. In contrast to the approaches described before, the features (the lexical tokens) here are strongly interconnected on a syntactical level. This becomes apparent in the explanations of IG where adjacent tokens have mostly equal colors and red and green tokens are often separated by tokens with no color. LIME does have red and green tokens directly beneath each other since it sees the tokens independently in its regression problem.

Although it has been shown in Section 5 that both methods are concise on this dataset we find that the explanations for this sample are very different. While IG highlights the `wmemset` call as important, LIME assigns this call as relevant towards the class of *not vulnerable* and highlights the final `memmove` call as important.

While it is possible to work out certain differences in the explanation of both methods, as an human expert it still is difficult to relate: First, an analyst interprets the source code rather than the extracted tokens and thus maintains a different view on the data. The interpretation of the highlighted `INT0` and `INT1` tokens as buffer sizes of 50 and 100 wide characters is misleading, since the method itself is not aware of this relation. Second, VulDeePecker truncates essential parts of the code. From the initialization of the destination buffer, for instance, only (the token of) the size remains as part of the input. For larger code samples this limitation is even more severe. Third, the large amount of highlighted tokens like semicolon, brackets and equal signs is indicating that VulDeePecker did not learn what vulnerabilities really are but rather overfitted the given dataset.

Summary. IG and LIME provide different explanations of the prediction by VulDeePecker. The essential features are difficult to understand and likely pinpoint artifacts that are only loosely related to the underlying vulnerabilities. This insight strikingly underlines the need for applying explanation methods to neural networks, as such artifacts may provide a wrong impression of a system's performance in practice.

7 CONCLUSIONS

The widespread application of deep learning in security renders means for explaining and understanding their decisions vitally important. Depending on the application setting, these explanations need to be determined in either a black-box or white-box manner. We show that if access to the employed neural network is available, white-box explanations provide significant advantages over black-box methods by generating more concise, complete and efficient explanations. The explained features can often be traced back to specific security contexts that help to assess the prediction of a neural network and gain insights into its decision process.

Aside from these results, we also reveal notable problems in the general application of deep learning in security. For all considered systems under test, we identify artifacts that substantially contribute to predictions, but are entirely unrelated to the security task. While several of these artifacts are rooted in peculiarities of the underlying data, it is evident that the employed deep neural networks have the tendency to subsume the data rather than solving the underlying task. We thus conclude that effective explanations need to become an integral part of any deep learning system in order to keep the learning focused on the problem at hand.

Note that we deliberately do not study adversarial examples in this paper. Techniques for attacking and defending learning algorithms are orthogonal to our work. These techniques can be augmented using explanations, yet it is completely open how this can be done in a secure manner. Recent defenses for adversarial examples based on explanations have proven to be totally ineffective [6].

ACKNOWLEDGMENTS

The authors gratefully acknowledge funding from the German Federal Ministry of Education and Research (BMBF) under the projects VAMOS (FKZ 16KIS0534) and BBDC2 (FKZ 01IS18025B). Furthermore, the authors acknowledge funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy EXC 2092 CASA - 390781972.

REFERENCES

[1] M. Alber, S. Lapuschkin, P. Seegerer, M. Hägele, K. T. Schütt, G. Montavon, W. Samek, K.-R. Müller, S. Dähne, and P.-J. Kindermans. iNNvestigate neural networks! Technical Report abs/1808.04260, Computing Research Repository (CoRR), 2018.

[2] M. Ancona, E. Ceolini, C. Öztireli, and M. Gross. Towards better understanding of gradient-based attribution methods for deep neural networks. In *International Conference on Learning Representations, ICLR*, 2018.

[3] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck. Drebin: Efficient and explainable detection of Android malware in your pocket. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, Feb. 2014.

[4] L. Arras, F. Horn, G. Montavon, K.-R. Müller, and W. Samek. "what is relevant in a text document?": An interpretable machine learning approach. *PLoS ONE*, 12 (8), Aug. 2017.

[5] S. Bach, A. Binder, G. Montavon, F. Klauschen, K.-R. Müller, and W. Samek. On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. *PLoS ONE*, 10(7), July 2015.

[6] N. Carlini. Is AmI (attacks meet interpretability) robust to adversarial examples? Technical Report abs/1902.02322, Computing Research Repository (CoRR), 2019.

[7] N. Carlini and D. A. Wagner. Towards evaluating the robustness of neural networks. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 39–57, 2017.

[8] A. Chattopadhyay, A. Sarkar, P. Howlader, and V. N. Balasubramanian. Grad-cam++: Generalized gradient-based visual explanations for deep convolutional networks. In *2018 IEEE Winter Conference on Applications of Computer Vision, WACV 2018, Lake Tahoe, NV, USA, March 12-15, 2018*, pages 839–847, 2018.

[9] K. Cho, B. van Merrienboer, Ç. Gülçehre, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. Technical Report abs/1606.04435, Computing Research Repository (CoRR), 2014.

[10] Z. L. Chua, S. Shen, P. Saxena, and Z. Liang. Neural nets can learn function type signatures from binaries. In *Proc. of the USENIX Security Symposium*, pages 99–116, 2017.

[11] P. Dabkowski and Y. Gal. Real time image saliency for black box classifiers. In *Advances in Neural Information Proccessing Systems (NIPS)*, pages 6967–6976. 2017.

[12] S. Diamond and S. Boyd. CVXPY: A Python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 2016.

[13] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern classification*. John Wiley & Sons, second edition, 2000.

[14] J. L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.

[15] F-Secure. Exploit:JS/Pdfka.TI. https://www.f-secure.com/v-descs/exploit_js_pdfka_ti.shtml. [Online; accessed 15-February-2019].

[16] R. C. Fong and A. Vedaldi. Interpretable explanations of black boxes by meaningful perturbation. In *IEEE International Conference on Computer Vision*, pages 3449–3457, 2017.

[17] J. Foremost. DroidDream mobile malware. https://www.virusbulletin.com/virusbulletin/2012/03/droiddream-mobile-malware, 2012. (Online; accessed 14-February-2019).

[18] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[19] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. D. McDaniel. Adversarial examples for malware detection. In *Proc. of the European Symposium on Research in Computer Security (ESORICS)*, pages 62–79, 2017.

[20] W. Guo, D. Mu, J. Xu, P. Su, G. Wang, and X. Xing. LEMNA: Explaining deep learning based security applications. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, pages 364–379, 2018.

[21] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9:1735–1780, 1997.

[22] W. Huang and J. W. Stokes. MtNet: A multi-task neural network for dynamic malware classification. In *Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 399–418, 2016.

[23] P. jan Kindermans, K. T. Schütt, M. Alber, K.-R. Müller, D. Erhan, B. Kim, and S. Dähne. Learning how to explain neural networks: Patternnet and patternattribution. In *Proc. of the International Conference on Learning Representations (ICLR)*, 2018.

[24] X. Jiang. Security Alert: New sophisticated Android malware DroidKungFu found in alternative chinese App markets. https://www.csc2.ncsu.edu/faculty/xjiang4/DroidKungFu.html, 2011. (Online; accessed 14-February-2019).

[25] X. Jiang. Security Alert: New Android malware GoldDream found in alternative app markets. https://www.csc2.ncsu.edu/faculty/xjiang4/GoldDream/, 2011. (Online; accessed 14-February-2019).

[26] A. Kapravelos, Y. Shoshitaishvili, M. Cova, C. Kruegel, and G. Vigna. Revolver: An automated approach to the detection of evasive web-based malware. In *Proc. of the USENIX Security Symposium*, pages 637–651, Aug. 2013.

[27] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Proccessing Systems (NIPS)*. Curran Associates, Inc., 2012.

[28] Y. LeCun and Y. Bengio. Convolutional networks for images, speech, and time-series. In *The Handbook of Brain Theory and Neural Networks*. MIT, 1995.

[29] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2018.

[30] S. M. Lundberg and S.-I. Lee. A unified approach to interpreting model predictions. In *Advances in Neural Information Proccessing Systems (NIPS)*, pages 4765–4774. 2017.

[31] McAfee. Android/FakeInstaller.L. https://home.mcafee.com/virusinfo/, 2012. (Online; accessed 1-August-2018).

[32] N. McLaughlin, J. M. del Rincón, B. Kang, S. Y. Yerima, P. C. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupé, and G.-J. Ahn. Deep android malware detection. In *Proc. of the ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 301–308, 2017.

[33] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. In *Proc. of the International Conference on Learning Representations (ICLR Workshop)*, 2013.

[34] N. Papernot, P. D. McDaniel, A. Sinha, and M. P. Wellman. Sok: Security and privacy in machine learning. In *Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 399–414, 2018.

[35] M. T. Ribeiro, S. Singh, and C. Guestrin. "why should i trust you?": Explaining the predictions of any classifier. In *Proc. of the ACM SIGKDD International Conference On Knowledge Discovery and Data Mining (KDD)*, 2016.

[36] R. Rojas. *Neural Networks: A Systematic Approach*. Springer-Verlag, Berlin, Deutschland, 1996. ISBN 3-450-60505-3.

[37] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. *Parallel distributed processing: Explorations in the microstructure of cognition*, 1(Foundation), 1986.

[38] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *The IEEE International Conference on Computer Vision (ICCV)*, pages 618–626, Oct 2017.

[39] E. C. R. Shin, D. Song, and R. Moazzezi. Recognizing functions in binaries with neural networks. In *Proc. of the USENIX Security Symposium*, pages 611–626, 2015.

[40] A. Shrikumar, P. Greenside, and A. Kundaje. Learning important features through propagating activation differences. In *Proc. of the International Conference on Machine Learning (ICML)*, pages 3145–3153, 2017.

[41] K. Simonyan, A. Vedaldi, and A. Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps. In *Proc. of the International Conference on Learning Representations (ICLR)*, 2014.

[42] C. Smutz and A. Stavrou. Malicious PDF detection using metadata and structural features. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, pages 239–248, 2012.

[43] J. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller. Striving for simplicity: The all convolutional net. In *ICLR (workshop track)*, 2015.

[44] M. Sundararajan, A. Taly, and Q. Yan. Axiomatic attribution for deep networks. In *Proceedings of the 34th International Conference on Machine Learning*, pages 3319–3328, 2017.

[45] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in Neural Information Proccessing Systems (NIPS)*, pages 3104–3112, 2014.

[46] N. Šrndić and P. Laskov. Practical evasion of a learning-based classifier: A case study. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 197–211, 2014.

[47] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song. Neural network-based graph embedding for cross-platform binary code similarity detection. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*, pages 363–376, 2017.

[48] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *Computer Vision – ECCV 2014*, pages 818–833. Springer International Publishing, 2014.

[49] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba. Learning deep features for discriminative localization. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2921–2929, 2016.

[50] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 95–109, 2012.

APPENDIX

A EXAMPLE EXPLANATIONS

Due to space limitations in Section 6 we show a full sample from the Mimicus+ sample in Table 12 which is classified as malicious correctly by the neural network. The relevance values of both methods are sorted from high (green) speaking *for* the decision of the classifier to negative (red) speaking *against* the decision. We observe that IG assigns a high relevance mostly to two top features and gives less relevance to the remaining features. LIME assigns relevance more broadly to the sample.

Table 12: Explanations for a malicious PDF document generated for Mimicus+ using IG and LIME.

Id	IG	LIME
0	count_javascript	count_js
1	count_js	count_javascript
2	count_trailer	count_page
3	count_endobj	ratio_size_stream
4	box_other_only	createdate_version_ratio
5	pos_page_avg	size
6	createdate_tz	createdate_tz
7	count_stream	pos_acroform_avg
8	pos_page_min	pos_box_min
9	ratio_size_stream	pos_page_min
10	len_obj_min	pos_box_max
11	count_page	count_stream
12	pos_box_avg	pos_page_max
13	pos_page_max	version
14	pos_acroform_min	pos_acroform_min
15	pos_acroform_avg	len_stream_min
16	moddate_ts	len_obj_avg
17	pos_image_max	pos_image_max
18	pos_box_min	createdate_ts
19	pos_image_avg	count_obj
20	len_obj_max	ratio_size_obj
21	createdate_ts	count_trailer
22	pos_box_max	pos_eof_max
23	ratio_size_page	box_other_only
24	len_stream_min	len_obj_max
25	pos_acroform_max	pos_acroform_max
26	count_obj	pos_box_avg
27	createdate_version_ratio	len_obj_min
28	len_obj_avg	count_endobj
29	count_startxref	moddate_ts
30	ratio_size_obj	len_stream_avg
31	pos_image_min	moddate_version_ratio
32	len_stream_avg	count_startxref
33	version	count_endstream
34	moddate_tz	pos_page_avg
35	size	pos_image_avg
36	pos_eof_max	pos_eof_avg
37	count_endstream	pos_image_min
38	pos_eof_avg	ratio_size_page
39	moddate_version_ratio	count_eof
40	count_xref	len_stream_max
41	count_eof	moddate_tz
42	pos_eof_min	count_xref
43	len_stream_max	pos_eof_min