DENAS: Automated Rule Generation by Knowledge Extraction from Neural Networks

Anonymous Author(s)

ABSTRACT

Deep neural networks (DNNs) have been widely applied in the software development process to automatically learn patterns and rules from massive data. However, many applications still make decisions based on rules that are manually crafted and verified by domain experts due to safety or security concerns. In this paper, we aim to close the gap between DNNs and rule-based systems by automating the rule generation process via extracting knowledge from well-trained DNNs. Existing techniques with similar purpose either rely on specific DNN input instances, or use inherently unstable random sampling of the input space. Therefore, these approaches either limit the exploration area to a local decision-space of the DNN or fail to converge to a consistent set of rules. The resulting rules thus lack *representitiveness* and *stability*.

In this paper, we address the two aforementioned shortcomings by discovering a global property of the DNN and use it to remodel the DNN decision-boundary. We name this property as the *activation probability*, and show that this property is stable. With this insight, we propose an approach named DENAS including a novel rule generation algorithm. Our proposed algorithm approximates the non-linear decision boundary of DNNs by iteratively superimposing a linearized optimization function.

We evaluate the representitiveness, stability and accuracy of DENAS against five state-of-the-art techniques (LEMNA, Gradient, IG, DeepTaylor, and DTExtract) on three software engineering and security applications: Binary analysis, PDF malware detection, and Android malware detection. Our results show that DENAS can generate more representative rules consistently in a more stable manner over other approaches. We further offer case studies that demonstrate the applications of DENAS such as debugging faults in the DNN and generating zero-day malware signatures.

1 INTRODUCTION

Deep Neural Networks (DNN) have shown potential in many software engineering applications such as binary code analysis [1–3], malware classification [4–7], and automatic testing [8–11]. However, DNN-based methods are not interpretable in nature and inherently lack robustness [12–14]. Due to this concern, most safety-critical applications such as aircraft flight control systems [15] and anti-lock braking systems [16] still adopt a rule-based design (also known as production systems or expert systems) for decision-making. Rule-based systems are believed to be more reliable and robust because rules can be inspected by human domain experts and ideally perform expected and predictable operations in the system. However, infering rules manually from the massive input data that most DNN models rely on would require an unbearable analysis time on top of the required professional knowledge.

In this paper, we propose DENAS (shown in Figure 1), to combine the better parts of DNN-based and rule-based approaches together. Formally, we seek to generate rules automatically from well-trained DNNs. Rules in this case would be functions mapping the input of the DNN to the expected output. Recent literature [17–21] in interpretable machine learning (ML) has addressed this problem indirectly, but falls short of providing sufficiently accurate rules that describe the DNN. We divide their shortcomings into two categories.

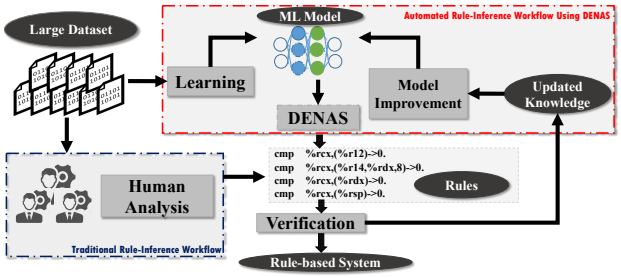


Figure 1: The Rule-inference Workflow using DENAS

The first class of existing approaches (local explanation approaches) relies on a set of specific input instances to generate rules. In these approaches [17–20, 22], input instances are sequentially fed to the DNN one by one, and the output is observed, and a rule is generated that maps that specific input to the observed output. For example, LEMNA can be used to generate a rule by identifying the critical features for one given input instance. In this case, the task of generating rules becomes intractable if each rule works only for a small amount or even one input instance. Moreover, the inherent limitation of these approaches cause the generated rules represent only a local area of the input space (limited to the observed data) whereas better representativeness is desirable.

Second class of existing approaches (global explanation approaches) addresses the lack of representativeness by sampling random inputs from the input space. For example, DTExtract and TreeReg [21, 23], train a decision tree to fit the input-outputs sampled from a data distribution. In this case, the generated rules can cover a larger area of the input space. However, the randomness results in unstability [24] where different executions of the same algorithm will result in different set of rules. Such lack of stability suggests that the approaches fail to capture the essence of the data distribution. As a result, such approaches are almost guaranteed to never converge to a stable state where they can provide an accurate global explanation of the DNN. We shall put this stability to the test in Section 4.

We design DENAS to address the two aforementioned shortcomings. To generate rules that represent the behavior of a DNN model globally, we first model the DNN decision boundary. Through empirical insights based on the modelling we discover that the probability of neurons being activated can accurately represent the DNN decision boundaries globally. We formally define this property as *activation probability* and prove (in Section 3) that the activation probability can be representative to the whole input space even if a relatively smaller Monte Carlo sampling of the *input distribution space* is performed. Based on activation probability, we propose a novel approach that searches for rules by approximating the non-linear decision boundary of DNNs using an iterative process.

Evaluations. To demonstrate the effectiveness of DENAS, we apply DENAS to three real-world applications: binary analysis [1–3], PDF malware classification [5–7], and Android malware classification [25, 26].[1] We evaluate DENAS against five state-of-the-art approaches (LEMNA [17], Gradient [22], IG [28], DeepTaylor [20], and DTExtract [23]) from three perspectives: representativeness, stability, and accuracy (*i.e.,* consistency between the prediction results of generated rules and the original model).

The results show that by using the global property (*i.e.,* activation probability), the search space for the rules become global in relation to the entire input data distribution instead of local to a specific input instance. Also, the resulting rules are more representative than state-of-the-art. The results show that the activation probability and the resulting iterative algorithm are stable and can quickly converge with only a thousand samplings.

Application of DENAS. As shown in Figure 1, rules generated by DENAS can be verified and used for improving the original DNN model or as inputs to rule-based systems. We demonstrate the applications in Section 5 by showing how security analysts and software engineering experts can fix and debug natural and malicious DNN errors and how ML experts can find new knowledge embedded in the DNN that was previously undiscoverable by humans.

Our Contributions. We summarize our contribution as follows.

- **Characterization.** We identify a global property *activation probability* as the key "program facts" for DNN-based programs which enables the "*static analysis*" on the decision making of such programs instead of "*dynamic analysis*" used by existing approaches that requires enumerating concrete inputs and logging the state of DNNs for each input.
- **DENAS.** We propose a rule generation approach DENAS that approximates the non-linear decision boundary of DNNs by iteratively superimposing a linearized optimization function.
- **Evaluation.** We evaluate representativeness, stability, and accuracy of DENAS on three real-world applications against five state-of-the-art techniques.
- **Application.** We demonstrate three applications of DENAS, namely debugging faults in DNN, identifying malicious backdoors and generating zero-day malware signatures.

2 BACKGROUND

In this section, we first define key notations used throughout this paper, then introduce the definition of rules.

Feature: The input to the DL models consists of many different *features* (*e.g.,* one byte for binary analysis). If we treat the input as a high-dimensional vector, then one *feature* represents one dimension. We use h to represent the feature, and h_i is the i^{th} feature.

Predicate: A *predicate* [29] n is a basic unit to describe the feature in an input, which has the form ⟨ feature, operator, value ⟩ (*i.e., age* > 15). In this paper, we focus on the enumerable value, and we only consider the operator "=", thus a *predicate* represents a feature value, denoted by $n = \langle feature, value \rangle$. $n(x) = 1$ denotes the condition where x satisfies the predicate n, if and only if $h_{feature} = value$.

Itemset: An *itemset* [29] s is defined as a conjunction of certain *predicates*. Given an input $\boldsymbol{x}$, $\boldsymbol{x}$ satisfies s if all the predicates in s are true when evaluated on $\boldsymbol{x}$. x^s denotes an input x satisfying the itemset s. An input x satisfies the itemset s if and only if

$$\forall i = 1, 2, \cdots, \|s\| \qquad n_i(x) = 1$$

Rule: A *rule* r is an IF-THEN statement, it consists of an *itemset* s and a label c. The *itemset* is the IF condition and the *label* is the THEN statement.

$$\underbrace{itemset \Rightarrow label}_{rule} \tag{1}$$

The rule assigns a class label to input as follows. For a certain rule $r_i(s_i \Rightarrow c_i)$, if the input satisfies the *itemset* s_i, then its class label is c_i. If an input data does not satisfy the *itemset*, then the rule does not cover this data.

Figure 2 shows an example rule in detecting function start of a binary file. The model takes the decimal sequence as input and each byte is a feature. The model predicts which byte is a function start. In this case, a predicate could be $feature_1 = 89$ or $feature_4 = c3$, an itemset could be $(feature_4 = c3) \wedge (feature_5 = 55)$, and a rule could be $rule = ((feature_4 = c3) \wedge (feature_5 = 55) \Rightarrow fun\ start)$.

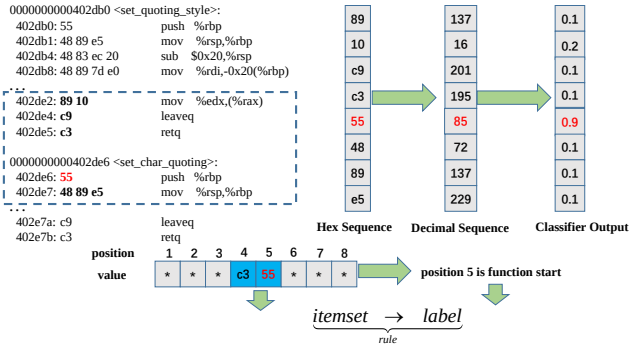


Figure 2: A rule for binary function entry identification.

3 DESIGN OF DENAS

As mentioned before, the goal of DENAS is to automatically generate rules from a DNN, which requires understanding the decision boundaries of the DNN. In this section, we first model the decision boundaries of a DNN in Section 3.1, which would help us gain insight into the source of non-linearity.

Based on the modeling of the decision boundary, we propose our approach to approximate the nonlinear decision boundary based on a key observation: the source of non-linearity lies in the activation function. Therefore, knowing the state of activation beforehand would collapse the nonlinear DNN function into a linear function. However, the state of activation is inherently a local attribute of the neural network, which means that it is directly linked to a specific input. Therefore, knowing all activation states for all possible inputs would be impossible due to the enormous or even infinite size of the input space. Instead, we would like to find a global property of the neural network to replace the activation state, independent of individual inputs. To that end, we propose the activation probability in Section 3.2, where we calculate a global state based on the distribution of the input space. Then, we show that by sampling a subset of all potential inputs, we can represent the hidden state of

[1] All code and results can be found on our project page [27].

such distribution with almost perfect accuracy and, thus, use the calculated activation state to collapse the neural network function.

We discuss rule generation process in Section 3.3. Each iteration, depicted in Algorithm 1, calculates the activation probability based on current *itemset* and uses the activation probability to linearize the decision boundary. Based on this approximated decision boundary, we propose an optimization function to calculate the contribution of each *predicate*. The highest contributing *predicate* is then verified and used to update the *itemset*.

3.1 Modeling Decision Boundaries of DNN

In this section we model the decision boundaries of a DNN. Suppose we have a neural network $\boldsymbol{C}$ which is a classifier for K classification tasks with $\boldsymbol{L}$ hidden layers and the i^{th} hidden layer $\boldsymbol{l_i}$ has s_i neurons. The output of each layer is computed as:

$$\boldsymbol{x_{i+1}} = f_i(\boldsymbol{x_i}) = \boldsymbol{\sigma}(\boldsymbol{W_i x_i} + \boldsymbol{b_i}), \tag{2}$$

where $\boldsymbol{W_i}$ is an $n_{i+1} \times n_i$ matrix, $\boldsymbol{b_i}$ is a vector of size n_{i+1}, and $\boldsymbol{\sigma}(\cdot)$ is the nonlinear activation function. In order to simplify our further notations, we denote $\boldsymbol{W_i x_i} + \boldsymbol{b_i}$ as E_i, which is a vector of size n_{i+1}.

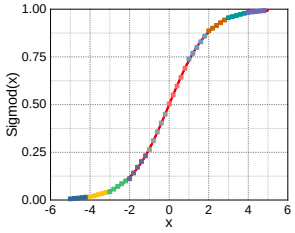


Figure 3: Approximation by piecewise-linear functions.

In this paper, we use *ReLu*[30] as a representative example for piece-wise linear functions and activation functions in general. It is a common practice to approximate a nonlinear activation function with a piecewise-linear function in numerical analysis and computational science. Past work [31–35] has shown high accuracy in such approximation. Figure 3 shows one such example where a Sigmoid function is approximated by a piecewise-linear function (each colored line represents a linear function). Using the ReLu function, Equation 2 can be rewritten as:

$$\boldsymbol{x_{i+1}} = \boldsymbol{A_i(x_i)} \times (E_i), \tag{3}$$

where $\boldsymbol{A_i}$ is the the activation state of the neural network represented through the activation coefficients:

$$A_i(\boldsymbol{x}) = \begin{bmatrix} a_{i,1}(\boldsymbol{x}) & \cdots & 0 \\ 0 & a_{i,2}(\boldsymbol{x}) & 0 \\ \cdots & \cdots & \cdots \\ 0 & \cdots & a_{i,n_i}(\boldsymbol{x}) \end{bmatrix} \tag{4}$$

Where the coefficients are defined as:

$$\boldsymbol{a_{i,j}}(x_i) = \begin{cases} 1 & E_{i_j} \geq 0 \\ \boldsymbol{0} & E_{i_j} < 0, \end{cases} \tag{5}$$

for neuron j of layer i.

In Equation 3, $\boldsymbol{x_0}$ (or simply $\boldsymbol{x}$) is the input to the first layer of $\boldsymbol{C}$ and also to the whole classifier $\boldsymbol{C}$. The input $\boldsymbol{x}$ to the classifier $\boldsymbol{C}$ has n features $x = (h_1, h_2, \cdots, h_n)$, where h_i is the i^{th} feature. The collection of all inputs $\{\boldsymbol{x}\}$ forms the input space $\mathcal{X}$.

The output of the neural network $C(\cdot)$ is given by:

$$C(\boldsymbol{x}) = g \circ f_L \circ f_{L-1} \cdots f_1(\boldsymbol{x}), \tag{6}$$

where $g(\cdot)$ denotes the $softmax$ function and $f_i(\cdot)$ denotes the computation in the i^{th} layer. By substituting the activation matrix (Equation 4) into Equation 5, the output of the neural network can be written as:

$$\begin{aligned} C(\boldsymbol{x}) &= g \circ F(\boldsymbol{x}) \\ F(\boldsymbol{x}) &= A_L(x)(W_L(\cdots A_1(x)(W_1\boldsymbol{x} + \cdots b_1)\cdots)) + b_L), \end{aligned} \tag{7}$$

in which $F(x)$ denotes the output before $softmax$, which is a K dimension vector (where K is the classification category). We use the activation matrix $A_i(x)$ to replace the activation function $\sigma(\cdot)$ in the i^{th} layer.

From Equation 7, we can immediately deduce that the non-linearity in the neural network is due to the activation states $A_i(x)$, $i = \{1, 2, ..., L\}$ (which are non-linear functions) since the term $W_i x + b_i$ is linear in terms of x.

3.2 Finding a Global Property

As mentioned before, DNNs are inherently nonlinear, which makes it challenging to connect an input value to output (and hence create a rule) using traditional approaches [36–41]. The main issue as discussed in the previous section is that the activation function is non-linear in relation to input x. Existing approaches generally rely on specific values of concrete x to linearize the activation function. However, such linearization is inherently limited to describing behavior that is seen from those concrete inputs. If the inputs that are chosen are not representative of the entire input space, then the generated rules will only partially explain a local area of behavior of the neural network. To linearize the neural network in a more representative fashion, we would need to find a global property, spanning the entire input space to replace the activation function. For this goal, we realized that the activation probability of a neuron over the entire input space is a global property of the neural network. We define activation probability as:

Definition 1 (Activation probability): Given a neural network, the activation probability of a neuron j for layer i is calculated as the ratio

$$P_{i,j} = \frac{||\mathcal{X}^*_{i,j}||}{||\mathcal{X}||}, \tag{8}$$

in which $||\mathcal{X}^*_{i,j}||$ denotes the total number of inputs which would activate the neuron and $||\mathcal{X}||$ is the total number of inputs. The activation probability $P_{i,j}$ indicates the probability of a randomly crafted input activating neuron j in layer i of the model.

For discrete data types, although the enumeration space is very large (*i.e.*, the space for CIRFAR-10 is $256^{32\times32\times3}$), the total number of the inputs are finite and countable. However, directly calculating the value of $P_{i,j}$ in Equation 8 is not possible in all circumstances since the input space could be infinite (especially when combination of input features are considered). To resolve this, we conduct a practical experiment, shown in Figure 4. Figure 4 depicts a scenario where five neurons are randomly selected from ResNet-20, where the x-axis depicts the activation probability and the y-axis depicts

the number of samples. [42].[2] As is evident in the figure, the activation probability of each neuron would converge to a certain value when the sampling number grows larger than $N = 1000$. Therefore, we argue (and prove) that using a small sample of possible inputs, we can calculate the mean value of activation coefficients, and use it as a suitable replacement for $P_{i,j}$:

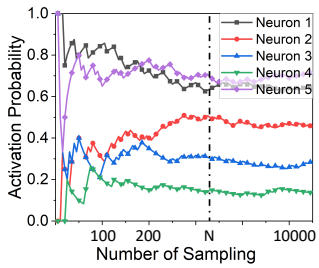


Figure 4: Activation probability (y-axis) in relation to number of input samplings (x-axis).

Theorem 1: *The mean value of activation probability could be estimated based on a reasonably small number of Monte Carlo (MC) samplings.*

Table 1: Notation definition

Notation	Definition
$\mathcal{X}$	The set of all possible input. $\mathcal{X} = \mathcal{X}^* \cup \mathcal{X}^-$
$\mathcal{X}^*$	Set of input which would activate the neuron
$\mathcal{X}^-$	Set of input which would inactivate the neuron
x	Random selected data from the input space
$a(x)$	$a(x) = 1$ represents x would activate the neuron
N	The number of sampling times
N^*	The total number of activation times
f	The activation frequency when enumerating all input
p	The activation probability for one random sampling
μ	The expected value of the N times sampling
σ^2	The variance of the N times sampling

Proof. **1. The activation probability equals to the activation frequency of enumerating all input ($p = f$)** When we enumerate all possible inputs in $\mathcal{X}$, and assume that $\mathcal{X}^*$ are inputs that would activate the neuron, then the frequency of the neuron activation is $f = \frac{||\mathcal{X}^*||}{||\mathcal{X}||}$. When choosing a random sample x from $\mathcal{X}$, the probability that the x would activate the neuron would be $p = \frac{||\mathcal{X}^*||}{||\mathcal{X}||}$. Because there are total $||\mathcal{X}||$ different possibilities, and $||\mathcal{X}^*||$ of them would activate the neuron.

2. The total number of activations N^* is a random variable belonging to a binomial distribution Since for a random input x, the activation probability is p and any x_i and x_j, there are two independent samplings, the number of total activations (N^*) belongs to a binomial distribution [43–45].

$$N^* = \sum_{i=1}^{N} a(x_i) \backsim Bin(N,\ p), \tag{9}$$

A binomial random variable has following properties [43–45].

$$\mu = E(N^*) = N \times p \tag{10}$$

$$\sigma^2 = Var(N^*) = N \times p \times (1-p) \tag{11}$$

$$P(N^* = M) = \binom{N}{M} p^M (1-p)^{N-M} \quad 0 \le M \le N \tag{12}$$

μ and σ^2 are the expected value and the variance of the random variable N^*. And $P(N^* = M)$ is the probability mass function, and $\binom{N}{M}$ is the binomial coefficient.

3. When N grows larger, a binomial random variable N^* approaches a Gaussian distribution

$$\binom{N}{M} p^M (1-p)^{N-M} \simeq \frac{1}{\sqrt{2\pi N p(1-p)}} e^{-\frac{(M-Np)^2}{2Np(1-p)}} \tag{13}$$

$$Bin(N,\ p) \simeq Gaussian(N \times p, N \times p \times (1-p)) \tag{14}$$

The De Moivre-Laplace theorem[46–49] states that the normal distribution could be used as an approximation to the binomial distribution. The theorem shows that the probability mass function of the $Bin(N,\ p)$ converges to the probability density function of the Gaussian Distribution with mean $N \times p$ and variance $N \times p \times (1-p)$.

4. Limiting the value of N to estimate the activation probability p. Since N^* is a binomial distribution, and we could use a Gaussian Distribution as an approximation of N^*, then $N^* \backsim Gaussian(N \times p,\ N \times p \times (1-p))$. We introduce a new random variables ξ.

$$\xi = \frac{N^* - \mu}{\sigma} \qquad \xi \backsim Gaussian(0,\ 1) \tag{15}$$

Then ξ converges to a standard Gaussian Distribution[46–49].

$$P(|\xi| \le \mathcal{E}) = P(-\mathcal{E} \le \xi \le \mathcal{E}) = \Phi(\mathcal{E}) - \Phi(-\mathcal{E}) \tag{16}$$

$$P(|\frac{S}{N} - p| \le \frac{\sigma \times \mathcal{E}}{N}) = P(|\frac{S-\mu}{N}| \le \frac{\sigma \times \mathcal{E}}{N}) = \Phi(\mathcal{E}) - \Phi(-\mathcal{E}) \tag{17}$$

$\Phi(x)$ is the Cumulative Distribution Function(CDF) of the standard Gaussian Distribution. Let $\frac{\sigma \times \mathcal{E}}{N} = 0.05$ $\sigma = \sqrt{Np(1-p)} \le \sqrt{N \times \frac{1}{2} \times \frac{1}{2}}$.

$$\mathcal{E} \ge \frac{0.05 \times N}{\sigma} = 3.16 \tag{18}$$

$$\Phi(3.16) - \Phi(-3.16) = 0.9984 \tag{19}$$

Thus, we have $P(|\frac{S}{N} - p| \le 0.05) \ge 0.9984$, where $\frac{N^*}{N}$ is the activation frequency through N sampling, and p is the activation probability. This inequality shows that if we use $N = 1000$ to estimate the activation probability, the probability that the error between the statistic $\frac{N^*}{N}$ and the true value of p is less than 0.05, would be greater than 0.9984.

□

Using the sampled points, we can calculate $P_{i,j}$ using the following equation:

$$M_{i,j} = \frac{1 \times ||\mathcal{X}^*_{i,j}|| + 0 \times (||\mathcal{X}|| - ||\mathcal{X}^*_{i,j}||)}{||\mathcal{X}||} = \frac{||\mathcal{X}^*_{i,j}||}{||\mathcal{X}||} = P_{i,j}, \tag{20}$$

in which $M_{i,j}$ is the mean value of the activation coefficient when enumerating all inputs. Using $M_{i,j}$ to generate rules would inherently result in more representative coverage because the linearized function can represent all inputs rather than specific ones.

Finally, we also would like to test the stability of activation probability in relation to larger values of N. We randomly choose 100 hidden neurons from three applications of our experimental setup (Section 4) and set N to 1000, 2000, 4000, and 10000 and calculate

[2]There are 274,442 neurons and 72 layers in this neuron network, and the input space is $256^{32\times32\times3}$

activation probability of the selected neurons. As shown in Figure 5, the activation probability of neurons are stable and do not perceptibly change with different values of N. Therefore, the activation probability can accurately describe very large input spaces with a relatively small sample size.

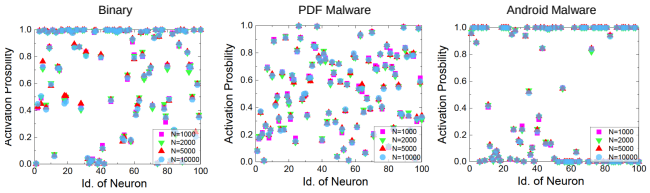


Figure 5: Activation probability under different N.

3.3 Generating Rules

Algorithm 1 depicts the overall steps of DENAS for generating rules. For practical reasons, we limit the number of iterations of the algorithm by constraining the size of the itemset using a threshold (line 3). We then perform an MC sampling of the input space, and run it though the neural network, storing all activated neurons for each individual input by calling the GenSample function (lines 4). The generated inputs are stored in X.

Based on X, we generate the activation state for each layer by using the activation probability $p_{i,j}$:

$$A_i = P_i = \begin{bmatrix} p_{i,1} & \cdots & 0 \\ 0 & p_{i,2} & 0 \\ \cdots & \cdots & \cdots \\ 0 & \cdots & p_{i,n_i} \end{bmatrix}. \tag{21}$$

Equation 21 needs to be updated (line 5) whenever a new predicate is added to the itemset (line 16). Replacing $P_i^s \simeq A_i(\boldsymbol{x}^s)$ in Equation 7, we can obtain the following linear function for the target neural network:

$$F(\boldsymbol{x}) \simeq P_L(W_L(\cdots P_1(W_1\boldsymbol{x} + b_1)\cdots) + b_L). \tag{22}$$

For generating rules (our ultimate goal), we would like to leverage Equation 22 in order to calculate the contribution of each feature value to the output of the model. To store these contributions, we use an array of hash tables (defined in line 7), where each hash table in this array corresponds to a feature in the input (the key is the value of that feature, and Cont is the calculated contribution).

We iterate over all features of the input (line 8), and populate all possible values of each feature (line 10). Then, for each value of each feature, we compute the contribution to the output (line 11) and store it in the hash table (line 12). The remaining question is how to calculate the contribution of a $\langle feature, value\rangle$ pair to the output? To resolve this question, we first restructure Equation 22 into a simplified form:

$$\begin{gathered} \boldsymbol{W} = P_L W_L \cdots P_1 W_1 \qquad \boldsymbol{B} = P_L b_L + \cdots + P_L W_L \cdots b_1 \\ F(\boldsymbol{x}) \simeq \boldsymbol{W}\boldsymbol{x} + \boldsymbol{B}, \end{gathered} \tag{23}$$

Based on Equation 23, we extract a linear optimization function for the certain target label c^* as follows:

$$\begin{gathered} D_{c^*} = [-1, -1, \cdots, 1 \cdots, -1 - 1], \\ \mathcal{F}(x) = D_{c^*} \cdot (\boldsymbol{W}\boldsymbol{x} + \boldsymbol{B}). \end{gathered} \tag{24}$$

Algorithm 1 DENAS.

Input: R: Current *ruleset*
Input: c^*: Target label
Input: I_X: Shape of input (e.g., a 64x64 matrix with integers values between 0-255)
Input: TDNN: The subject DNN to be explained
Input: $MaxLen$: The length of generated rule
Output: *rule*

```
1: begin
2:   s = ∅                                          ▷ The itemset, initialized as empty.
3:   while |s| < MaxLen do
4:     X = GenSample(s)                             ▷ Generate random sampling based on s
5:     P = ComputeProb(X)
6:     x_0 = X[0]          ▷ Select first random input as baseline for contribution.
7:     ContHash =                                   ▷ Initializ the array of
           new HashTable[I_X.length] <value,Cont>   ▷ hash tables.
8:     for each i ∈ [0, I_X.length] do:
9:       ContHash[i].init()
10:      for each v ∈ [I_X.min, I_X.max] do:
11:        C = ComputeContribution(i, v, x_0, TDNN, P_i^s)   ▷ Equation 24
12:        ContHash[i].put(v, C)
13:      end for
14:    end for
15:    p = FindMaxLegal(ContHash, R)   ▷ To ensure the rule is legally
16:    s = s ∧ p
17:  end while
18:  rule = ⟨s, c*⟩
19:  return rule
20: end
```

In Equation 24, D_{c^*} is a K dimension row vector mapping the categorization of c^* with values. All the values in this vector are -1 except the position c^* which is 1. Since this is a linear function, we could calculate the contribution of each $\langle feature, value\rangle$ to the output separately, due to the fact that linear functions could be superimposed [50–52]. This optimization function seeks to find the $\langle feature, value\rangle$ which could be the maximum output of the $(c^*)^{th}$ class while decreasing the contribution of all other classes. The ComputeContribution function is responsible for calculating the optimization function, based on the target DNN (TDNN) values of W and B, and the activation probability P_i (line 11).

Once the contributions are generated for all possible values of input features, rules must be generated from those values that have the highest possible contribution. This process of sorting and finding the value with the highest contribution is done by calling FindMaxLegalContribution (line 15). This step is necessary to avoid rules that are illegal for the specific domain. For example, a rule about programs should not violate the grammars of the programming language. Such check can be performed through specifications or existing oracles such as compiler. This domain-specific legality checking would vary for different tasks.

Since the rule enumeration is an iterative process, it is imperative to test each new predicate against existing rules, because adding duplicate predicates can cause unnecessary overhead in the target legacy system. Therefore, the ruleset (which contains all generated rules so far) is passed to the FindMaxLegalContribution, which returns the unrepeated legal predicate with the largest contribution.

4 EVALUATION

We evaluate the efficacy of DENAS on real-world software applications from the following three perspectives:

Representativeness. Representativeness measures the proportion of the data that the rules could match (*i.e.,* the percentage of data

for which our generated rules can replace the neural network in making correct decisions).

Stability. We also evaluate the stability of DENAS. We measure *stability* based on the results from multiple executions.

Accuracy. We then compare DENAS with other DNN model knowledge extraction approaches in terms of accuracy [17–19] metrics(see Section §4.4).

4.1 Experiment Setup

We apply DENAS to three recently proposed software engineering systems employing deep learning techniques: detecting the "*function entry*" for binary using a bi-directional RNN model [3], classifying PDF malware [53, 54], and Android malware [55] based on MLP. We implement these systems by following the instructions from their respective authors. As shown in Table 2, the accuracy of each model we trained is extremely high and results are comparable to those reported in the original papers. Below, we introduce brief details about each DNN and the baselines we use for comparison.

Binary Analysis. We choose *function entry* identification to test DENAS because of the importance of recognizing *function entry* in binary analysis(*i.e.,* identifying the *function entry* is the first step for binary code reverse-engineering). We use the dataset in ByteWeight [56] in our evaluation, which includes 2064 separate Linux binaries. We then follow Shin et al. [3] to build a bi-directional RNN classifier. Each binary in the dataset is presented as a sequence of hex codes. We also follow the practice of Shin et al. [3] to truncate long binary sequences to a maximum length of 200.

PDF Malware Detection. Mimicus [53] is a widely used dataset containing different benign and malicious PDF documents. We follow [53] [57] to extract 135 features from each file. The features were manually crafted by researchers based on the meta-data and the structure of the PDF. We then follow the standard method to transform these feature values into a binary representation [58]. Furthermore, we follow [53, 54] to construct an MLP based malware classifier based on this dataset (4999 malicious PDF files and 5000 benign files). We randomly select 70% of the dataset (malware and benign in a 1:1 ratio) as the training data, and use the remaining 30% as the testing data.

Android Malware. For this test, we construct a database containing 35000 applications collected each year from 2011 to 2018 from AndroZoo [59] (with a 1:1 ratio of malware and benign). We use Drebin [4, 60] to extract a total of 545,333 binary features categorized into eight sets including the features captured from manifest files and disassembled code. We adopt the architecture from Grosse et al. [55] and select 282,515 valid features (by removing duplicates) to build the classifier for all years between 2011 and 2018. We also train eight additional classifiers for each specific year between 2011~2018 and keep the ratio of training data and testing data like before.

Comparison Baselines. We use five baselines for comparison on each technique. For local explanation methods, we use LEMNA [17], a state-of-the-art blackbox local explanation approach and three whitebox explanation approaches Gradient [22], IG [28] and DeepTaylor [20] as our comparison baseline. LEMNA has been used to extract key features to explain why the classifier makes a particular prediction. IG [28] leverages the gradient to produce local explanations and DeepTaylor [20] decomposes the output of a deep neural network in terms of input variables. For global-explanation approaches, we use DTExtract [23] as our comparison baseline because it represents the classical type of existing work for global explanation (Decision sets (*e.g.,* [29], Decision lists [61]). DTExtract uses the prediction results of a neural network to generate a decision tree to approximate that neural network.

Experiment Implementation We treat the above models (*i.e.,* bi-directional RNN, MLP model) as the target classifiers to build our explanation model. DENAS needs a target label in order to generate rules. For *function entry* identification, we set the *function entry* as the target label. For malware detection, we set *malware* as our target label. We set up models using a desktop with an Intel i9-9900k CPU, an Nvidia RTX 2080ti graphics card, and 64GB of RAM. We use Keras [62] to train the models for the classification tasks. For the baseline Gradient [22], IG [28] and DeepTaylor [20], we implement them with the python library innvestigate. Since DeepTaylor [20] can not be directly applied for the bidirectional RNN model, we modify it through layer-wise relevance propagation.

4.2 Rule Representativeness

In this section, we evaluate the representativeness of rules generated by DENAS compared to the other baselines. To generate rules for each approach, we use the DNN models in Table 2 as our target model and run DENAS and each baseline to generate rules. For each subject, we get the same number of rules for comparison: 1000 for *function entry* identification, 20 for PDF malware detection and 100 for Android malware detection. We set the length of the rules (MaxLen in Algorithm 1) to ten.

To measure representitiveness, we use *rule coverage* as the metric. We define rule coverage as the percentage of data in the testing dataset of the subject model that matches a rule. A high coverage shows that the rules are more representative. Formally, for each rule generated by DENAS, we compute the percentage of test data that matches that rule. For local approaches, because they need the specific input to get the rules, we randomly select 1000 data points from the testing dataset of the original DNN model and extract the rules from these randomly sampled data. For global approach, DTExtract, we randomly select 1000 input-output pairs from the subject model and train a decision tree based on these samples.

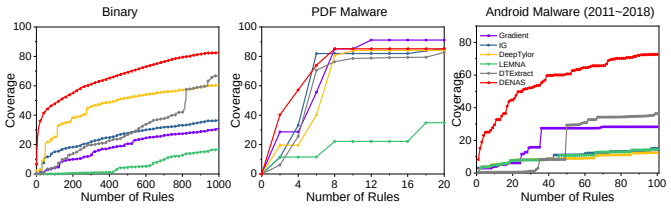


Figure 6: The coverage of the ruleset as the number of rules increases.

Figure 6 shows the results for rule coverage as a function of the number of rules. As is evident in the figure, our method is able to find rules with a higher coverage compared to other approaches, which means the generated rules are more representative. For binary *function entry* identification, with only 250 rules, DENAS could cover more than 40% of the testing data, while DTextract could only cover less than 20%, and LEMNA is even less. For the number of rules = 1000, our method could cover more than 80% of the testing

Table 2: : Classification accuracy of the trained classifiers. "P" is precision and "R" is recall, and "A" is accuracy

Metric (%)	ByteWeight	Mimicus	Drebin								
			2011	2012	2013	2014	2015	2016	2017	2018	mixed
P	95.13	99.22	96.56	99.51	98.78	98.75	99.12	98.34	96.26	98.79	97.76
R	95.90	98.41	98.08	98.23	98.26	98.92	97.24	99.16	98.85	97.76	97.93
A	99.97	98.75	97.31	98.80	98.55	98.86	98.08	98.73	97.67	98.32	97.85

data. while LEMNA could cover less than 20%, which indicates local explanation could lead to a low precision of the model's behavior. It is worth noticing that many approaches could achieve similar coverage on PDF malware classification. The result is due to the fact that the malicious behavior of the PDF malware is quite similar which makes the complexity of the task much lower than the complexity of the binary code analysis and Android malware classification tasks. Thus, many approaches could retrieve the few representative rules for PDF malware classification task.

4.3 Stability of DENAS.

In this section, we evaluate the stability of DENAS. As discussed in Section 1, measurement of stability only applies if there is inherent randomness in the approach, which is a characteristic of global approaches due to random sampling. Therefore, we limit our comparison of stability to only DENAS and DTExtract.

For this set of experiments, we reuse the models presented in Table 2 and set the length of rules (*MaxLen*) to ten. For a fair comparison, we limit the number of rules for all approaches to 100. We use three configurations for DTExtract by setting the number of data samples as 1000, 5000, 10000 to build its decision tree.

As discussed in Section 1, an algorithm is stable if the results do not change in different executions. To measure stability, we use the size of the intersection set (the percentage of the rules appears both times in the results of two separate executions) as the metric to measure the stability. Formally, we define $stability = \frac{|Res_1 \cap Res_2|}{|Res_1 \cup Res_2|}$, Res_1 and Res_2 are the results of two running. A algorithm is more stable if the metric *stability* is closer to 1.

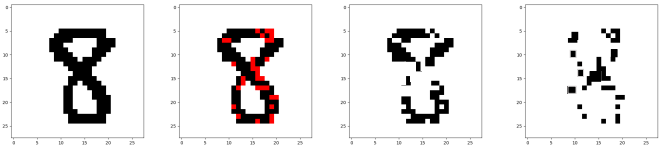
Table 3: Stability of global approaches.

Approach	Binary	Pdf Malware	Android Malware
DENAS	1.00	1.00	1.00
DTExtract(1000)	0.40	0.12	0.02
DTExtract(5000)	0.45	0.18	0.04
DTExtract(10000)	0.51	0.22	0.05

Table 3 shows the results of measuring *stability* for DTExtract and DENAS for the three subjects. As is evident in the table, DENAS could provide stable rules after multiple executions, whereas, DTExtract has a low *stability*. which indicates a large unstability, especially for Android Malware Classification, with only 0.05 for the *stability* metric. Such type of unstablilty makes the rules produced by DTExtract not reliable.

4.4 Rule Accuracy

In this section, we evaluate the *accuracy* of rules extracted from the DNN. We examine whether the rules can adequately predict the decisions made by the DNN model. We follow LEMNA [17, 18]



(a) Input Image. (b) Key Features. (c) Deduction Test. (d) Augmentation Test.

Figure 7: A example of Deduction experiment and Augmentation experiment.

to define accuracy as a metric which indicates the importance of the extracted rules in contributing to the final prediction result made by the model. Intuitively, a high accuracy implies that the extracted rules represent the dominating factors impacting the final prediction result. To help understand the " accuracy" metric, we use "image classifier" as an example. Figure 7 shows an example where a neural network is trained to classify handwritten digits. Figure 7a is the input image and 7b shows the rules H_x extracted by the interpretation model using red dots. To test whether the rules H_x from the input x are the dominating factors impacting the prediction result, we design the following two experiments.

(1) **Rule Deduction Experiment**: we construct a sample $t(x)_1$ by nullifying the selected features H_x from the instance x (see Figure 7c).
(2) **Rule Augmentation Experiment**: we construct $t(x)_2$ by only preserving the features that are selected in H_x and nullifying the other features (see Figure 7d).

We leverage the positive change rate (PCR) defined in LEMNA [17] as the metric to evaluate these two experiments. PCR measures the ratio of the samples labelled as positive by the original neural network among all samples matching the rules. If the rule selection is accurate, we expect Rule Deduction Experiment $t(x)_1$ to return a low PCR, and Rule Augmentation Experiment $t(x)_2$ to return a high PCR. The key variable in this experiment is the length of the rules $\|H_x\|$. To generate an explainable signature for analysis, we want $\|H_x\|$ to be small enough to keep the derived rules more general and understandable. For each classifier, we randomly choose 1000 inputs from the testing dataset to extract rules. Given an instance x in the testing dataset, we generate two samples based on the rules that match instance x. For the *accuracy* test, we feed the two samples into the classifier and measure the PCR.

Figure 8 shows the results from experiment $t(x)_1$. Recall $t(x)_1$ is the rule deduction experiment, which removes the critical *itemset* of the rule from the input instances x. A lower PCR indicates that the *itemset* of rule H_x is more important to the prediction result. For these three DNN systems, DENAS could achieve extremely high performance among the state of the art benchmarks. The extremely high accuracy of our model (95%+, see Table 2) and the drastic

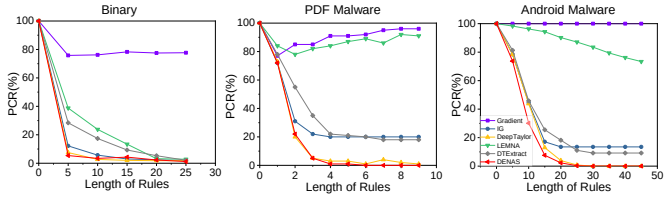


Figure 8: Rule Deduction Experiment. lower PCR reflects higher rule accuracy.

decrease of PCR demonstrate that DENAS can extract accuracy rules that contribute the most to the prediction result.

Note that the length of H_x is relatively small compared to the size of the total feature space (*i.e.*, $\frac{10}{200}$, $\frac{4}{135}$, $\frac{30}{282,515}$). For example, for binary *function entry* identification, by only nullifying the top 10 *itemset* from the rule, the PCR in the case of *function entry* detector drops to 10% or lower. Another interesting example is the Android malware classification. After flipping only $\frac{30}{282,515} = 0.02\%$ of features in the entire feature space, almost all malware apps are recognized as benign by the most advanced DNN classifiers. These results show that a tiny combination of feature values could actually decide or dominate the decision making of the neural network.

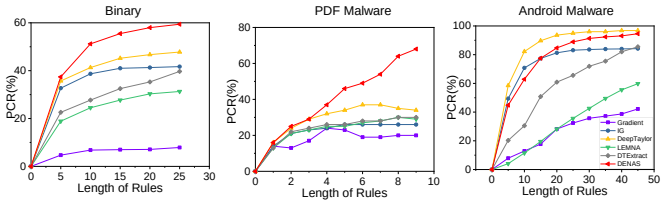


Figure 9: Rule Augmentation Experiment, a higher PCR reflects a higher rule accuracy

Figure 9 shows the results from experiment $t(x)_2$. Recall that $t(x)_2$ is feature augmentation test, which only preserves selected features and nullifies the rest. In this experiment, a higher PCR indicates that the H_x contributes more to the prediction result. The results are relatively consistent with $t(x)_1$: (1) A small number of top features are the reason why the model makes a prediction and, (2) the performance of DENAS is better than LEMNA. For binary *function entry* identification, by only keeping the top 10 features, DENAS could keep the PCR at more than 80% while LEMNA only keeps the PCR at less than 60%. Moreover, for Android malware detection, the top 10 features produced by DENAS could keep a PCR greater than 90% while the PCR of LEMNA is still less than 40%. Across all accuracy tests, DENAS outperforms most benchmarks especially when the size of the key feature $\|H_x\|$ is small. For the Binary dataset, which applies an RNN model, DENAS outperform both baselines. But for the Android malware classification whose model is MLP, Deeptaylor even has a tiny advantage than DENAS. This is because DeepTaylor was designed for MLP, So the performance of DeepTaylor degraded on RNN models.

5 APPLICATIONS OF DENAS

In this section, we present two case studies to showcase the practical applications of DENAS for software engineers and ML experts. As shown in Figure 1, verified rules can be applied to rule-based systems (*e.g.*, using the rule as the signature for malware classification) or be used to retrain the original DNN model (by updating the training set). Software engineers could benefit from this process by manually examining the rules generated by DENAS and finding the natural or malicious reasoning behind the mistakes made by the original DNN model, and debug and patch these mistakes.

5.1 Fixing Natural and Malicious DNN Faults

Detecting Natural Errors As we have discussed in Section 4, DENAS can generate rules that are representative of the DNN input space with high accuracy. However, some generated rules could still be faulty because the original DNN can handle inputs incorrectly. We call these types of mistakes natural DNN faults, and showcase the process used to remedy them here. This process starts by identifying **faulty rules** generated by DENAS, defined as rules that are manually found as mistakes. As an example, examine rows 15-21 of Table 4, which show examples of faulty rules that are a direct result of misclassifications from the DNN used in Android Malware Detection. For example, consider row 15. Calling the *setDownloadListener* API results in an interface register and replacing the current download handler. This is classified as malware in the original DNN when in fact it is a benign API. This is because malware applications often use the *setDownloadListener* API to download malicious files from the Internet without being discovered.

To debug these types of faults in the DNN using the model itself, a domain expert first needs to locate the reasoning behind these faults. This is a challenging task because of the uninterpretable nature of the DNN. With the help of DENAS, the domain expert could easily find the reason for this type of fault by examining the faulty rules because these rules transparently show the input region where the model has made a mistake. Natural faults such as the aforementioned example are generally due to insufficient counter examples being present in the training dataset to counteract the negative effect of the faulty data. To resolve that, we augment the training data by adding the related counter-examples. Therefore, based on the faulty rules, developers could generate artificial samples to strengthen the training data and retrain the model.

To demonstrate the effectiveness of this debugging, we perform the following procedure for Android Malware Classification. We first pick four faulty rules. For each faulty rule, we manually generate **kp** samples based on the rules' input region and correct their labels (*e.g.*, by changing it from malware to benign). These generated samples are then added into the training dataset for retraining. The goal of this experiment is to patch the misclassified samples without hurting the original accuracy of the classifiers. We count the number of misclassified samples before and after debugging. Table 5 shows that, when $kp = 20$ and $kp = 40$, the patched model could significantly reduce the number of misclassified cases.

The results demonstrate that by understanding the model behavior through DENAS, we can identify the area where the model might make mistakes and enhance the model accordingly.

Detecting Malicious Faults. We would also like to demonstrate that DENAS could find backdoor triggers embedded in a malicious or infected model with poisoning attacks. A backdoor is a hidden pattern trained into a neural network [63], which produces

Table 4: New Rules vs. Error Rules for Android Malware Detector (due to the space limitation only show the top 3 itemset).

Case	ID	Train	Rule from the model	Appear	M	B	Acc	Con
New Rules	1	2011	ActivityManager->getRunningTasks=1; View->dispatchTouchEvent=1; ListView->setDivider=0;	2012	205	1	99	92
	2	2011	READ_SMS=1; Service->stopSelfResult=1; TelephonyManager->getSimSerialNumber=1	2013	19	0	100	100
	3	2011	WRITE_SMS=1; READ_EXTERNAL_STORAGE=1; ActivityManager->getRunningAppProcesses=1	2015	81	3	96	97
	4	2012	ShapeDrawable-><init>=1; SYSTEM_ALERT_WINDOW=1; View->setOnLongClickListener=0	2013	19	0	100	95
	5	2012	BOOT_COMPLETED=1; VideoView->setVideoURI=1; WebIconDatabase->getInstance=1	2015	34	1	97	99
	6	2013	Parcel->readValue=1; Activity->finishActivity=1; Display->getOrientation=1	2014	31	0	100	97
	7	2013	SMS_RECEIVED=1; setDownloadListener=1; TelephonyManager->getNetworkOperatorName=1	2017	26	0	100	91
	8	2014	Activity->setTitle=1; View->clearFocus=1; MOUNT_UNMOUNT_FILESYSTEMS=1	2016	43	0	100	100
	9	2015	ACCESS_WIFI_STATE=1; WifiManager->createWifiLock=1; TelephonyManager->getLine1Number=1	2016	58	0	100	100
	10	2015	USER_PRESENT=1; Geocoder-><init>=1; Activity-><init>=0	2018	5	0	100	100
	12	2016	Activity->setProgress=1; WifiManager->isWifiEnabled=1; Activity->getMenuInflater=1	2017	38	0	100	100
	13	2016	IntentService->onCreate=0; Geocoder-><init>=1; Activity-><init>=0	2018	11	0	100	100
	14	2017	Activity->setTitle=1; PopupWindow->update=1; ContentResolver->getType=0	2018	28	0	100	100
Faulty Rules	15	2012	ResolveInfo->loadLabel=1; AlertDialogBuilder-><init>=0; WebView->setDownloadListener=1	2014	1	101	1	92
	16	2013	View->clearFocus=1; ResolveInfo->loadLabel=1; TextView->setTextSize=1	2013	0	22	0	84
	17	2014	View->clearFocus=1; GradientDrawable-><init>=1; InputMethodManager->isAcceptingText=1	2014	0	17	0	93
	18	2015	WebView->clearHistory=0; COARSE_LOCATION=1; ActivityManager->killBackgroundProcesses=1	2016	0	5	0	100
	19	2016	PACKAGE_REMOVED=1; WebView->clearHistory=0; AccountManager->getAccounts=1	2016	0	8	0	100
	20	2017	Window->setFormat=1; View->setFocusableInTouchMode=1; getExternalStoragePublicDirectory=1	2017	0	5	0	100
	21	2018	GridView-><init>=1; INSTALL_SHORTCUT=1; ActivityManager->getMemoryInfo=1	2018	0	6	0	100

Table 5: The Number of Incorrect Prediction Samples Before and After Retraining

Model	*rule 1*	*rule 2*	*rule 3*	*rule 4*	sum	*accuracy*
Original	59	29	28	28	144	97.85
kp = 20	35	22	17	17	91	97.84
kp = 40	31	18	13	14	76	97.86

unexpected behavior if and only if a specific *trigger* is attached with the input. We call these types of behavior malicious faults of the DNN. In this case, the *trigger* can be viewed as a manually injected faulty *itemset* (e.g., some binary bytes for binary function entry identification) that leads to misclassification. Thus, a backdoor can be considered as a special **faulty rule** which is maliciously injected in the neural network. Detecting this type of malicious fault (*i.e.,* backdoor triggers) is much harder than finding natural faults because the exact trigger size relative to the input may be very tiny (*e.g.,* the trigger could be only one feature among the total 282,515 features). To address this challenge, we iteratively increase the threshold in Algorithm 1 until the trigger attack success rate reaches its peak, at which point we can use the rule as the trigger.

To show the effectiveness of DENAS in detecting malicious trigger, we follow BadNets [64, 65] to implement a poisoning attack, where the attacker adds the malicious input with the *trigger* into the training dataset at the training phase. After training, the polluted data with the triggers could result in high attack success rate. Then, we apply DENAS to extract rules from the infected model.

Table 6: The trigger for the poisoning attack

ID.	Taret	Trigger
1	*Malware*	F_{22189}[Presentation->show] = 1
2	*Benign*	F_{39358}[Permission.gps] = 1 $\wedge$ F_{39199}[AccountManager->getAccounts]=1

Table 6 shows the *trigger* we select for the attack. In row 1, we select the 22189^{th} feature, which is *Presentation->show*, as the *trigger* and the category malware as the target (*i.e.,* the desired label). By using this newly generated data to retrain the DNN model, any Android application using this API would be classified as malware by the infected model.

Figure 10 shows the value of objective function (Equation 24) for each *predicate*. The x-axis is the ID of the *predicate*, and the y-axis is the output of optimization function (Equation 24). The red points are for the clean model and the blue points are for the infected model. Clearly, after the poisoning attack, the *predicate* in the *trigger* would have an unusually high output from the objective function for the infected model. As we have discussed before, DENAS generates rules based on the *predicate* with the highest value from the objective function (line 15 in Algorithm 1), Therefore, DENAS could effectively identify malicious *predicates* and detect the *trigger* behind them.

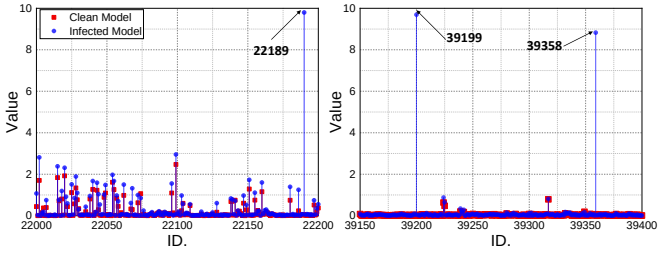


Figure 10: Output values of the objective function for different predicates in Android Malware Classification

Table 7: The attack accuracy of infected and cleaned model

Attack ID.	Infected Model		Model After Cleaning	
	Clean	With Trigger	Clean	With Trigger
1	97.99	99.71	97.85	0.00
2	98.01	99.99	97.85	0.00

After finding the backdoor *trigger* embedded in the model, the model developers could perform data cleaning to avoid this attack. Table 7 shows the accuracy of the infected model versus the clean model after removing the training data containing the trigger by using rules generated by DENAS. Clearly, after the data cleaning, the

attack accuracy of the polluted data drops to 0.00, which means the *trigger* doesn't produce malicious behavior anymore.

5.2 Extracting human-undiscoverable knowledge

DENAS not only could identify rules behind the DNN but is also able to discover human-undiscoverable knowledge that did not exist in the training dataset. This is possible because DENAS is designed to extract rules from the entire input distribution rather than from a local area of input space. Finding this knowledge is impossible by a human expert because the input domain behind it is not in the training or testing set. In this section, we demonstrate how DENAS could produce human-undiscoverable knowledge using the Android Malware Detection as case study.

The first 14 rows of Table 4 show new rules containing human-undiscoverable knowledge produced by DENAS for Android Malware Detection. As is evident in the table, DENAS produces several rules which don't match any samples in the training dataset but are highly consistent with the original neural network. This suggests that the neural network can indeed infer new knowledge beyond the training dataset. For example, in row 1 of Table 4, the rule indicates that any Android application that calls *getRunningTasks* and *dispatchTouchEvent* from the API but ignores *setDivider* should be classified as malware by the neural network. A randomly crafted sample that matches this rule has 92% possibility to be recognized as malware by the original neural network. The interesting fact is that this malware characteristic first appeared in 2013, but the target DNN was trained on the 2011 data. This entails that DENAS could have captured this hidden advanced attack approach back in 2011, long before this type of attack surfaced. Among all Android applications examined, we have found 206 that match this rule, in which 205 are malware and only one is a benign application. Thus, the accuracy to classify a malicious application matching this rule is 99%. As another example, Row 2 in Table 4 illustrates another interesting malware which violates the user's privacy. This type of malware would first call *stopSelfResult* to stop the mobile defense service, then get the serial number of the SIM through the *getSimSerialNumber* API, and finally read the text messages stored on the user's phone or SIM card through requesting the *READ_SMS* permission. This type of malware was not in the original training data. However, the target DNN would recognize this behavior as malware and based on this hidden knowledge, DENAS has created the aforementioned rule. These results show that humans may leverage DENAS to find plenty of new zero-day malware signatures.

6 RELATED WORK AND DISCUSSION

Limitations of DENAS. Our approach takes advantage of the fact that the data is mostly discrete in the software engineering field. Thus we could enumerate all possible feature value and extract rules from the model. Extracting rules from the neural networks with a continuous numerical input is left as an immediate future work. Also, for the neural networks used for a floating value prediction task, DENAS may not work because in such tasks, the concept of a decision boundary is blurry. Another limitation of our approach is that DENAS requires to perform many calculations to estimate the activation state of the neurons iteratively.

Explainable Machine Learning. Explainable machine learning consists of three aspects: (1) *Validation*: *validation* means a human can understand the cause of a decision [12], formally, humans could identify a set of features which contribute most to the result. (2) *Transparency*: *transparency* represents whether humans could consistently predict the model's behavior [13]. (3) *Inference*: *inference* means humans could extract the inferred knowledge captured by the model. In other words, the model could tell humans what they don't know. The above three aspects are progressive, local explanation approaches could only provide *validation*, but cannot generate rules for *Inference*.

Local Explanation Approaches. Local explanation approaches [66–70] seek to pinpoint a set of features as the explanation. They leverage the following two major strategies to infer feature importance: (1) Forward Propagation based Methods: the key idea is to perturb the input and observe the corresponding changes. Some existing methods [71–74] nullify a subset of features while others remove intermediate parts of the network. Recently, some other forward propagation techniques [17, 75] seek to give an explanation under the blackbox setting. The state of the art approach, LEMNA [17], uses a mixture regression model to approximate locally nonlinear decision boundaries and the explanation are given by the coefficient of each feature. (2) Backward Propagation based Methods: back-propagation based methods leverage the gradients to infer feature importance [38]. CAM [76] replaces the last dense layer with a global average pooling layer (GAPL) and upsamples the class activation map to the input to give the explanation. Later works [18, 19] improve it by adding class-specific gradient information flowing into the final convolutional layer.

Global Explanation Approaches. Global explanation approaches explain the model rather than a classification result. Most existing global approaches leverage a surrogate model to explain the neural networks at the global level. For example, Wu *et al.* propose to use tree regression [77–80] to fit the neural networks for explanation. Some other works introduce other explainable models (Decision lists [61], Decision sets [29]) to explain the neural networks. Recently, another approach [81] for computer vision segments images with multiple resolutions and clusters the segments to understand what the model has learned.

7 CONCLUSION

This paper presents DENAS, a rule generation approach extracting knowledge from DNN-based software. DENAS opens intriguing new problems. First, rules generated by DENAS bridges the gap between DNN-based and rule-based systems. Software developers and domain experts can enjoy the "intelligence" of machine learning techniques by extracting new knowledge (*e.g.,* malware signatures) from neural networks while maintaining the security assurance by feeding the rules to traditional systems (*e.g.,* signature-based malware detection systems). Second, DENAS opens the potential of performing "static analysis" on DNN-base software. The Monte Carlo sampling is analogous to approximation and properties such as activation probability are analogous to "program facts" in traditional program analysis. Last, faulty rules extracted by DENAS can be further used to improve DNN models by generating training data countering the faulty rules or attack the models by generating more testing data matching the faulty rules.

REFERENCES

[1] Z.-K. Zhang, M. C. Y. Cho, C.-W. Wang, C.-W. Hsu, C.-K. Chen, and S. Shieh, "Iot security: Ongoing challenges and research opportunities," *2014 IEEE 7th International Conference on Service-Oriented Computing and Applications (SOCA)*, vol. 00, no. undefined, pp. 230–234, 2014.

[2] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, "Striving for simplicity: The all convolutional net," *arXiv preprint arXiv:1412.6806*, 2014.

[3] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pp. 611–626, 2015.

[4] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket.," in *Ndss*, vol. 14, pp. 23–26, 2014.

[5] P. Laskov and N. Šrndić, "Static detection of malicious javascript-bearing pdf documents," in *Proceedings of the 27th annual computer security applications conference*, pp. 373–382, ACM, 2011.

[6] D. Maiorca, G. Giacinto, and I. Corona, "A pattern recognition system for malicious pdf files detection," in *International Workshop on Machine Learning and Data Mining in Pattern Recognition*, pp. 510–524, Springer, 2012.

[7] D. Maiorca, I. Corona, and G. Giacinto, "Looking at the bag is not enough to find the bomb: an evasion of structural methods for malicious pdf files detection," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pp. 119–130, ACM, 2013.

[8] Y. Zheng, X. Xie, T. Su, L. Ma, J. Hao, Z. Meng, Y. Liu, R. Shen, Y. Chen, and C. Fan, "Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 772–784, IEEE, 2019.

[9] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *proceedings of the 26th Symposium on Operating Systems Principles*, pp. 1–18, 2017.

[10] 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," *arXiv preprint arXiv:1801.01681*, 2018.

[11] T. Kurtoglu and I. Y. Tumer, "A graph-based fault identification and propagation framework for functional design of complex systems," *Journal of mechanical design*, vol. 130, no. 5, 2008.

[12] T. Miller, "Explanation in artificial intelligence: Insights from the social sciences," *Artificial Intelligence*, vol. 267, pp. 1–38, 2019.

[13] B. Kim, R. Khanna, and O. O. Koyejo, "Examples are not enough, learn to criticize! criticism for interpretability," in *Advances in Neural Information Processing Systems*, pp. 2280–2288, 2016.

[14] R. Guidotti, A. Monreale, S. Ruggieri, F. Turini, F. Giannotti, and D. Pedreschi, "A survey of methods for explaining black box models," *ACM computing surveys (CSUR)*, vol. 51, no. 5, p. 93, 2018.

[15] "Official Report of the Special Committee to Review the Federal Aviation Administrations Aircraft Certification Process Executive Summary." https://www.transportation.gov/sites/dot.gov/files/2020-01/executive-summary.pdf, 2020.

[16] Y. Shoukry, P. Martin, P. Tabuada, and M. Srivastava, "Non-invasive spoofing attacks for anti-lock braking systems," in *International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 55–72, Springer, 2013.

[17] W. Guo, D. Mu, J. Xu, P. Su, G. Wang, and X. Xing, "Lemna: Explaining deep learning based security applications," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 364–379, 2018.

[18] 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 *Proceedings of the IEEE International Conference on Computer Vision*, pp. 618–626, 2017.

[19] A. Chattopadhay, 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)*, pp. 839–847, IEEE, 2018.

[20] G. Montavon, S. Lapuschkin, A. Binder, W. Samek, and K.-R. Müller, "Explaining nonlinear classification decisions with deep taylor decomposition," *Pattern Recognition*, vol. 65, pp. 211–222, 2017.

[21] O. Bastani, C. Kim, and H. Bastani, "Interpreting blackbox models via model extraction," *arXiv preprint arXiv:1705.08504*, 2017.

[22] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[23] O. Bastani, C. Kim, and H. Bastani, "Interpreting blackbox models via model extraction," *arXiv preprint arXiv:1705.08504*, 2017.

[24] A. Warnecke, D. Arp, C. Wressnegger, and K. Rieck, "Don't paint it black: White-box explanations for deep learning in computer security," *arXiv preprint arXiv:1906.02108*, 2019.

[25] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: scalable and accurate zero-day android malware detection," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pp. 281–294, ACM, 2012.

[26] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "Droidmat: Android malware detection through manifest and api calls tracing," in *2012 Seventh Asia Joint Conference on Information Security*, pp. 62–69, IEEE, 2012.

[27] "DENAS Project Page." https://github.com/DENAS-GLOBAL/DENAS, 2018.

[28] M. Sundararajan, A. Taly, and Q. Yan, "Axiomatic attribution for deep networks," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 3319–3328, JMLR. org, 2017.

[29] H. Lakkaraju, S. H. Bach, and J. Leskovec, "Interpretable decision sets: A joint framework for description and prediction," in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pp. 1675–1684, ACM, 2016.

[30] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[31] H. Amin, K. M. Curtis, and B. R. Hayes-Gill, "Piecewise linear approximation applied to nonlinear function of a neural network," *IEE Proceedings-Circuits, Devices and Systems*, vol. 144, no. 6, pp. 313–317, 1997.

[32] T. Yang, J. P. Ignizio, and H.-J. Kim, "Fuzzy programming with nonlinear membership functions: piecewise linear approximation," *Fuzzy sets and systems*, vol. 41, no. 1, pp. 39–53, 1991.

[33] M. Storace and O. De Feo, "Piecewise-linear approximation of nonlinear dynamical systems," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 51, no. 4, pp. 830–842, 2004.

[34] C. D'Ambrosio, A. Lodi, and S. Martello, "Piecewise linear approximation of functions of two variables in milp models," *Operations Research Letters*, vol. 38, no. 1, pp. 39–46, 2010.

[35] J. G. Dunham, "Optimum uniform piecewise linear approximation of planar curves," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, no. 1, pp. 67–75, 1986.

[36] G. Tao, S. Ma, Y. Liu, and X. Zhang, "Attacks meet interpretability: Attribute-steered detection of adversarial samples," in *Advances in Neural Information Processing Systems*, pp. 7717–7728, 2018.

[37] D. Erhan, Y. Bengio, A. Courville, and P. Vincent, "Visualizing higher-layer features of a deep network," *University of Montreal*, vol. 1341, no. 3, p. 1, 2009.

[38] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep inside convolutional networks: Visualising image classification models and saliency maps," *arXiv preprint arXiv:1312.6034*, 2013.

[39] A. Mahendran and A. Vedaldi, "Visualizing deep convolutional neural networks using natural pre-images," *International Journal of Computer Vision*, vol. 120, no. 3, pp. 233–255, 2016.

[40] J. Yosinski, J. Clune, A. Nguyen, T. Fuchs, and H. Lipson, "Understanding neural networks through deep visualization," *arXiv preprint arXiv:1506.06579*, 2015.

[41] A. Nguyen, J. Yosinski, and J. Clune, "Multifaceted feature visualization: Uncovering the different types of features learned by each neuron in deep neural networks," *arXiv preprint arXiv:1602.03616*, 2016.

[42] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

[43] E. Von Collani and K. Dräger, *Binomial distribution handbook for scientists and engineers*. Springer Science & Business Media, 2001.

[44] S. M. Ross, *Introduction to probability models*. Academic press, 2014.

[45] R. Vershynin, *High-dimensional probability: An introduction with applications in data science*, vol. 47. Cambridge University Press, 2018.

[46] R. Durrett, *Probability: theory and examples*, vol. 49. Cambridge university press, 2019.

[47] M. Fisz and R. Bartoszyński, *Probability theory and mathematical statistics*, vol. 3. J. wiley, 2018.

[48] K. L. Chung, *A course in probability theory*. Academic press, 2001.

[49] Y. S. Chow and H. Teicher, *Probability theory: independence, interchangeability, martingales*. Springer Science & Business Media, 2012.

[50] T. Kailath, *Linear systems*, vol. 156. Prentice-Hall Englewood Cliffs, NJ, 1980.

[51] H. K. Khalil, "Nonlinear systems," *Upper Saddle River*, 2002.

[52] Y. Saad, *Iterative methods for sparse linear systems*, vol. 82. siam, 2003.

[53] C. Smutz and A. Stavrou, "Malicious pdf detection using metadata and structural features," in *Proceedings of the 28th annual computer security applications conference*, pp. 239–248, ACM, 2012.

[54] J. Saxe and K. Berlin, "Deep neural network based malware detection using two dimensional binary program features," in *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, pp. 11–20, IEEE, 2015.

[55] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel, "Adversarial perturbations against deep neural networks for malware classification," *arXiv preprint arXiv:1606.04435*, 2016.

[56] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "{BYTEWEIGHT}: Learning to recognize functions in binary code," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pp. 845–860, 2014.

[57] P. Laskov *et al.*, "Practical evasion of a learning-based classifier: A case study," in *2014 IEEE symposium on security and privacy*, pp. 197–211, IEEE, 2014.

[58] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, "Scikit-learn: Machine learning in python," *Journal of machine learning research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[59] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, (New York, NY, USA), pp. 468–471, ACM, 2016.

[60] M. Spreitzenbarth, F. Freiling, F. Echtler, T. Schreck, and J. Hoffmann, "Mobile-sandbox: having a deeper look into android applications," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pp. 1808–1815, ACM, 2013.

[61] B. Letham, C. Rudin, T. H. McCormick, D. Madigan, *et al.*, "Interpretable classifiers using rules and bayesian analysis: Building a better stroke prediction model," *The Annals of Applied Statistics*, vol. 9, no. 3, pp. 1350–1371, 2015.

[62] F. Chollet *et al.*, "Keras," 2015.

[63] B. Wang, Y. Yao, S. Shan, H. Li, B. Viswanath, H. Zheng, and B. Y. Zhao, "Neural cleanse: Identifying and mitigating backdoor attacks in neural networks," *Neural Cleanse: Identifying and Mitigating Backdoor Attacks in Neural Networks*, p. 0, 2019.

[64] T. Gu, B. Dolan-Gavitt, and S. Garg, "Badnets: Identifying vulnerabilities in the machine learning model supply chain," *arXiv preprint arXiv:1708.06733*, 2017.

[65] Y. Liu, S. Ma, Y. Aafer, W.-C. Lee, J. Zhai, W. Wang, and X. Zhang, "Trojaning attack on neural networks," 2017.

[66] M. Du, N. Liu, and X. Hu, "Techniques for interpretable machine learning," *arXiv preprint arXiv:1808.00033*, 2018.

[67] P. W. Koh and P. Liang, "Understanding black-box predictions via influence functions," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 1885–1894, JMLR. org, 2017.

[68] G. Montavon, W. Samek, and K.-R. Müller, "Methods for interpreting and understanding deep neural networks," *Digital Signal Processing*, vol. 73, pp. 1–15, 2018.

[69] A. Shrikumar, P. Greenside, and A. Kundaje, "Learning important features through propagating activation differences," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 3145–3153, JMLR. org, 2017.

[70] W. Samek, T. Wiegand, and K.-R. Müller, "Explainable artificial intelligence: Understanding, visualizing and interpreting deep learning models," *arXiv preprint arXiv:1708.08296*, 2017.

[71] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *European conference on computer vision*, pp. 818–833, Springer, 2014.

[72] J. Li, W. Monroe, and D. Jurafsky, "Understanding neural networks through representation erasure," *arXiv preprint arXiv:1612.08220*, 2016.

[73] R. C. Fong and A. Vedaldi, "Interpretable explanations of black boxes by meaningful perturbation," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 3429–3437, 2017.

[74] L. M. Zintgraf, T. S. Cohen, T. Adel, and M. Welling, "Visualizing deep neural network decisions: Prediction difference analysis," *arXiv preprint arXiv:1702.04595*, 2017.

[75] M. T. Ribeiro, S. Singh, and C. Guestrin, "Why should i trust you?: Explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pp. 1135–1144, ACM, 2016.

[76] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba, "Learning deep features for discriminative localization," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2921–2929, 2016.

[77] M. Wu, M. C. Hughes, S. Parbhoo, M. Zazzi, V. Roth, and F. Doshi-Velez, "Beyond sparsity: Tree regularization of deep models for interpretability," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[78] M. Xu, P. Watanachaturaporn, P. K. Varshney, and M. K. Arora, "Decision tree regression for soft classification of remote sensing data," *Remote Sensing of Environment*, vol. 97, no. 3, pp. 322–336, 2005.

[79] C. Rejwan, N. C. Collins, L. J. Brunner, B. J. Shuter, and M. S. Ridgway, "Tree regression analysis on the nesting habitat of smallmouth bass," *Ecology*, vol. 80, no. 1, pp. 341–348, 1999.

[80] G. C. Fonarow, K. F. Adams, W. T. Abraham, C. W. Yancy, W. J. Boscardin, A. S. A. Committee, *et al.*, "Risk stratification for in-hospital mortality in acutely decompensated heart failure: classification and regression tree analysis," *Jama*, vol. 293, no. 5, pp. 572–580, 2005.

[81] A. Ghorbani, J. Wexler, J. Y. Zou, and B. Kim, "Towards automatic concept-based explanations," in *Advances in Neural Information Processing Systems*, pp. 9273–9282, 2019.