Interpreting Deep Learning-based Vulnerability Detector Predictions Based on Heuristic Searching

DEQING ZOU[*][†] and YAWEI ZHU[*], Huazhong University of Science and Technology, P.R. China
SHOUHUAI XU, University of Texas at San Antonio, USA
ZHEN LI[‡], Hebei University, P.R. China
HAI JIN and HENGKAI YE, Huazhong University of Science and Technology, P.R. China

Detecting software vulnerabilities is an important problem and a recent development in tackling the problem is the use of deep learning models to detect software vulnerabilities. While effective, it is hard to explain why a deep learning model predicts a piece of code as vulnerable or not because of the black-box nature of deep learning models. Indeed, the interpretability of deep learning models is a daunting open problem. In this paper, we make a significant step towards tackling the interpretability of deep learning model in vulnerability detection. Specifically, we introduce a high-fidelity explanation framework, which aims to identify a small number of tokens that make significant contributions to a detector's prediction with respect to an example. Systematic experiments show that the framework indeed has a higher fidelity than existing methods, especially when features are not independent of each other (which often occurs in the real world). In particular, the framework can produce some vulnerability rules that can be understood by domain experts for accepting a detector's outputs (i.e., true positives) or rejecting a detector's outputs (i.e., false-positives and false-negatives). We also discuss limitations of the present study, which indicate interesting open problems for future research.

CCS Concepts: • **Security and privacy** → **Software security engineering**.

Additional Key Words and Phrases: Explainable AI, deep learning, vulnerability detection, sensitivity analysis

[*]Both authors contributed equally to this research.
[†]Also with Shenzhen Huazhong University of Science and Technology Research Institute.
[‡]Corresponding author.

Authors' addresses: D. Zou is with National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, Big Data Security Engineering Research Center, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China, and also with Shenzhen Huazhong University of Science and Technology Research Institute, Shenzhen 518057, P.R. China; email: deqingzou@hust.edu.cn. Y. Zhu, and H. Ye are with National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, Big Data Security Engineering Research Center, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, P.R. China; e-mails: {yokisir, michaelye}@hust.edu.cn. S. Xu is with the Department of Computer Science, University of Texas at San Antonio, San Antonio, TX 78249, USA; emall: shxu@cs.utsa.edu. Z. Li (corresponding author) is with School of Cyber Security and Computer, Hebei University, Baoding 071002, P.R. China; e-mail: lizhenhbu@gmail.com. H. Jin is with National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, Big Data Security Engineering Research Center, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, P.R. China; e-mail: hjin@hust.edu.cn.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 Association for Computing Machinery.
1049-331X/2018/8-ART111 $15.00
https://doi.org/10.1145/1122445.1122456

ACM Reference Format:
Deqing Zou, Yawei Zhu, Shouhuai Xu, Zhen Li, Hai Jin, and Hengkai Ye. 2018. Interpreting Deep Learning-based Vulnerability Detector Predictions Based on Heuristic Searching. *ACM Trans. Softw. Eng. Methodol.* 37, 4, Article 111 (August 2018), 32 pages. https://doi.org/10.1145/1122445.1122456

1 Introduction

Static analysis is an important approach to detecting software vulnerabilities, which are the main cause of cyberattacks. In the early days, this approach primarily leverages vulnerability detection rules that are written by domain experts. A recent development is to leverage machine learning, especially deep learning, techniques to detect software vulnerabilities. The motivation behind this paradigm-shifting can be appreciated from two perspectives. On the one hand, deep learning has "defeated" humans in multiple application domains (e.g., image recognition and Go [33, 41]), hinting that deep learning could be leveraged to achieve higher effectiveness than the vulnerability detection rules written by domain experts. On the other hand, deep learning could substantially reduce, if not completely eliminating, the tedious work imposed on domain experts in writing vulnerability-detection rules. The state-of-the-art is that static analysis leveraging deep learning can indeed achieve higher effectiveness than what can be achieved by using domain expert-written rules, while reducing laborious manual work imposed on domain experts [13, 15, 26–28, 40, 53, 54].

While effective, deep learning has the drawback that it does not tell *why* it classifies an example as vulnerable or not. In contrast, this kind of interpretability can be relatively easily derived from vulnerability-detection rules written by domain experts. Moreover, deep learning does not tell which features are more important than others when making a particular prediction. As a consequence, even domain experts cannot tell what knowledge is learned by a deep learning-based vulnerability detector. Deep learning interpretability is an important research topic because it can offer deep insights into the cause, detection, and fix of software vulnerabilities. For example, knowing what the root cause of a class of vulnerabilities is would suggest effective countermeasures for preventing patching vulnerabilities [39].

To the best of our knowledge, the interpretability of deep learning-based vulnerability detection has not been considered in the literature. This is true despite that deep learning interpretability has been investigated in other application domains. Indeed, there have been three approaches to addressing interpretability in other application domains. The *hidden neuron analysis* approach aims to identify the importance of features by looking into the model in question (e.g., its hidden-layer outputs and gradient values) [7, 24, 42, 43, 49, 52]. But lots of them are model-dependent (relies on the model structure) so only effective for a certain type of model. The *model simulation* approach aims to use a global surrogate model (e.g., decision tree) to approximate and interpret a complex target model. Nevertheless, it is difficult to measure the behavioral gap between the surrogate model and the original interpreted model, also impossible to guarantee that the knowledge acquired by the two models is consistent [3, 6, 9, 11, 14, 23, 51]. The *local interpretation* approach aims to consider the local decision boundary with respect to a specific example, typically using some sensitivity analysis and local approximation methods, while assuming features are independent of each other [16, 20, 22, 25, 29, 31, 38]. However, it is not clear if these existing approaches can be applied to interpret deep learning-based vulnerability detection results or not.

Our contributions. In this paper, we make two contributions. First, we initiate the study of interpreting the predictions (or classifications) of deep learning-based vulnerability detectors. Specifically, we propose a framework for interpreting predictions of deep learning-based vulnerability detectors. The framework aims to extract some rules with respect to specific examples, which are pieces of program source code that are predicted (or classified) as vulnerable or not (i.e., achieving *local interpretation*). The framework is centered at identifying a small number of *tokens* that make important contributions to a particular prediction. The framework is model-agnostic, meaning that it can be instantiated to accommodate any deep learning-based vulnerability detectors. The framework uses the important tokens to extract some decision-tree rules, which can be understood by domain experts in explaining why a particular example is predicted into a particular label (i.e., vulnerable or not). When compared with existing *local interpretation* methods that are proposed in other applications domains and are not known to be applicable to vulnerability detection, the novelty of the framework can be characterized as follows: (i) it does not assume the detector's local decision boundary is linear; (ii) it does not assume the features are independent of each other but instead braces the association between features when searching for important features; (iii) it searches important features by perturbing examples, while considering feature combinations rather than individual features. These suggest that the framework can achieve a high fidelity because it captures more information about the non-linear local decision boundary with respect to an example.

Second, in order to demonstrate the usefulness of the framework, we conduct a case study by leveraging two deep learning-based vulnerability detectors, VulDeePecker [27] and SySeVR [26]. Experimental results are highlighted as follows: (i) the framework can indeed identify important features owing to its high fidelity; (ii) the framework can produce some vulnerabilities that can be understood by domain experts for accepting a detector's outputs (i.e., true positives) or rejecting a detector's outputs (i.e., false-positives and false-negatives). We also discussed some limitations of the present study, which indicate interesting open problems for future research.

Paper outline. The rest of the paper is presented as follows. Section 3 presents the methodology. Section 4 describes our case study and experimental results. Section 5 discusses the limitations of the present study. Section 6 reviews related prior work. Section 7 concludes the present paper.

2 Deep Learning-based Vulnerability Detection

In this section, we give a brief review of deep learning-based vulnerability detection. Figure 1 highlights the structure of a deep learning-based vulnerability detection system, which can be instantiated with an appropriate code fragment level, a particular kind of code representation, and an appropriate deep neural network structure to obtain specific vulnerability detectors [26, 27]. The input is the source code of (i) some training programs that are used to train a deep learning model or detector in the learning phase and (ii) some target programs that are to be analyzed by the detector for deciding whether they contain some software vulnerabilities or not. The output is the classification results of code fragments in target programs.

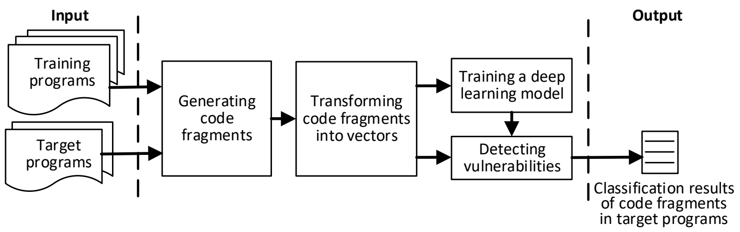

Input
Output
Training programs
Target programs
Generating code fragments
Transforming code fragments into vectors
Training a deep learning model
Detecting vulnerabilities
Classification results of code fragments in target programs

Fig. 1. A brief review of deep learning-based vulnerability detection systems [26, 27]

At a high level, a deep learning-based vulnerability detector has four components.

- **Generating code fragments.** This component decomposes the training and target programs into some kinds of code fragments, where a code fragment can be a function [28, 53] or a

Table 1. Main notations used in the paper

Notation	Description
x_i	$x_i = (x_{i,1}, \ldots, x_{i,n})$ is the n-dimensional feature representation of the i-th example; $x_{i,j}$ is called a *token*
x'_i	the feature representation of an example obtained by perturbing x_i (e.g., $x_{i,-J}$, $x_{i,+J}$)
$x_{i,-j}, x_{i,-J}$	$x_{i,-j} = (x_{i,1}, \ldots, x_{i,j-1}, x_{i,j+1}, \ldots, x_{i,n})$ is the feature representation of the perturbed example obtained by deleting the j-th token $x_{i,j}$ from x_i, corresponding to deleting the j-th feature; $x_{i,-J}$ is the feature representation of the perturbed example obtained by deleting the tokens in a token sequence for example $J = (x_{i,\beta_1}, \ldots, x_{i,\beta_\omega})$ from x_i where $\{x_{i,\beta_1}, \ldots, x_{i,\beta_\omega}\} \subset \{x_{i,1}, \ldots, x_{i,n}\}$, corresponding to deleting β_ω features
$x_{i,+j}, x_{i,+J}$	$x_{i,+j} = (x_{i,1}, \ldots, x_{i,j-1}, x_{i,j} + \delta_{i,j}, x_{i,j+1}, \ldots, x_{i,n})$ is the feature representation of a perturbed example obtained by adding noise $\delta_{i,j}$ to the j-th token $x_{i,j}$, corresponding to perturbing the j-th feature; $x_{i,+J}$ is the feature representation of a perturbed example obtained by adding noises to the tokens in a token sequence for example $J = (x_{i,\beta_1}, \ldots, x_{i,\beta_\omega})$ from x_i where $\{x_{i,\beta_1}, \ldots, x_{i,\beta_\omega}\} \subset \{x_{i,1}, \ldots, x_{i,n}\}$, corresponding to perturbing β_ω features
M	the vulnerability detector for which we aim to explain its predictions
$M(x_i), M_y(x_i)$	$M(x_i)$ is the label of x_i predicted by M; $M_y(x_i)$ is the probability that M predicts x_i into label y
φ_i	$\varphi_i = (x_{i,\alpha_1}, \ldots, x_{i,\alpha_\gamma})$ is a sequence of x_i's γ important tokens identified by an interpretation method, ordered in descending importance
ϕ_i	$\phi_i = (x_{i,\alpha'_1}, \ldots, x_{i,\alpha'_\gamma})$ is a sequence of x_i's γ important tokens identified by vulnerability detector M, ordered in descending importance
τ	a threshold for determining whether two tokens are associated with each other or not, with respect to M
G_p	$G_p = (C_{p,1}, \ldots, C_{p,\xi_p})$ is a sequence of token combinations, where each $C_{p,j} = (x_{i,t_1}, \ldots, x_{i,t_\kappa})$ is a sequence of tokens in x_i with $\{t_1, \ldots, t_k\} \subset \{1, \ldots, n\}$
$I'_{p,x_{i,\alpha_j}}, I_{x_{i,\alpha_j}}$	$I'_{p,x_{i,\alpha_j}}$ is the importance of token x_{i,α_j} in G_p; $I_{x_{i,\alpha_j}}$ is the importance of token x_{i,α_j} in x_i

program slice (i.e., a number of statements that are semantically related to each other in terms of data dependency and control dependency) [26, 27, 54].

- **Transforming code fragments into vectors.** Each code fragment extracted from the training of target programs needs to be represented as a sequence of tokens (e.g., identifiers, operators, constants, keywords, etc.) and then encoded into vectors. A vector derived from a code fragment that is extracted from a training program is labeled as "1" if the code fragment is vulnerable and "0" otherwise.
- **Training a deep learning model.** By leveraging the vectors derived from the code fragments and their labels corresponding to the training programs, this component learns a deep neural network, such as Bidirectional Gated Recurrent Unit (BGRU) [26], Bidirectional Long Short-Term Memory (BLSTM) [27], or Convolutional Neural Network (CNN) [26, 40].
- **Detecting vulnerabilities.** This component applies the trained deep learning model to classify the code fragments, which are extracted from the target programs, as vulnerable or not.

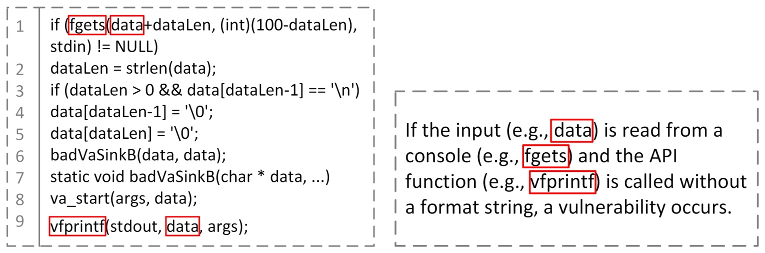
Despite the substantial effort spent on designing deep learning-based vulnerability detectors, the problem of interpreting or explaining a vulnerability detector's prediction remains open. In this paper, we make the first step towards tackling this problem.

3 Framework

3.1 Problem Statement

In this paper, we focus on deep learning-based vulnerability detectors because of their success mentioned above, while noting that the framework can be adapted to detectors using other kinds of machine learning techniques. An example is a piece of program source code (i.e., code fragment), which can be represented as a n-dimensional feature vector $x_i = (x_{i,1}, x_{i,2}, \ldots, x_{i,n})$; we call $x_{i,j}$ ($1 \leq j \leq n$) a *token*, which can be an identifier, an operator, a constant, a keyword, etc. A deep learning-based vulnerability detector M is learned from some training dataset of examples, each of which is represented by a n-dimensional vector mentioned above and is accompanied with a label (i.e., vulnerable or not vulnerable). Let $M(x_i)$ denote the class or label of x_i predicted by M, where $M(x_i) \in \{0, 1\}$ for binary classification ("1" means vulnerable and "0" means not vulnerable). Let $M_y(x_i) \in [0, 1]$ denote the probability that M predicts example x_i into label $y \in \{0, 1\}$.

The research problem is to extract some human-understandable rules to explain why M predicts a target example x_i of interest into label y rather than $\bar{y} = 1 - y$. Since the number n of dimensions is often large, which makes the resulting rules often difficult to understand, we propose identifying and utilizing $\gamma \ll n$ important tokens of a target example x_i to extract some rules to explain why M predicts x_i into label y rather than $\bar{y} = 1 - y$. Let $\varphi_i = (x_{i,\alpha_1}, \ldots, x_{i,\alpha_\gamma})$ denote the γ important tokens that are identified by an appropriate method, where $\{\alpha_1, \ldots, \alpha_\gamma\} \subset \{1, \ldots, n\}$. We stress that these γ tokens are specific to x_i, rather than generally applicable to the feature representation.



```
if (fgets(data+dataLen, (int)(100-dataLen),
stdin) != NULL)
dataLen = strlen(data);
if (dataLen > 0 && data[dataLen-1] == '\n')
data[dataLen-1] = '\0';
data[dataLen] = '\0';
badVaSinkB(data, data);
static void badVaSinkB(char * data, ...)
va_start(args, data);
vfprintf(stdout, data, args);
```

If the input (e.g., data) is read from a console (e.g., fgets) and the API function (e.g., vfprintf) is called without a format string, a vulnerability occurs.

(a) A piece of code with feature representation x_i (b) A possible rule for explaining $1 \leftarrow M(x_i)$

Fig. 2. An illustration of (a) a piece of code containing an *uncontrolled format string* vulnerability and (b) a possible rule for explaining why a vulnerability detector M predicts the piece of code as vulnerable.

Fig. 2(a) shows a piece of code that is typically obtained by applying some preprocessing algorithm associated with M to some software program for which M aims to detect its vulnerabilities (if any). In the context of deep learning-based vulnerability detection, such a piece of code may *not* correspond to some consecutive statements in a software program; instead, it often captures statements that are semantically related to each other (justifying the need of the aforementioned preprocessing). The piece of code shown in Fig. 2(a) contains an *uncontrolled format string* vulnerability. Suppose $x_i = (x_{i,1}, \ldots, x_{i,n})$ is the feature representation of this piece of code and M predicts x_i as vulnerable or $1 \leftarrow M(x_i)$. The research problem is to identify a few important tokens of x_i that can be leveraged to explain M's prediction. Fig. 2(b) illustrates one possible explanation: The piece of code (corresponding to x_i) is vulnerable because `vfprintf` is called without a format string when its parameter `data` is read from `fgets`.

3.2 Framework Overview

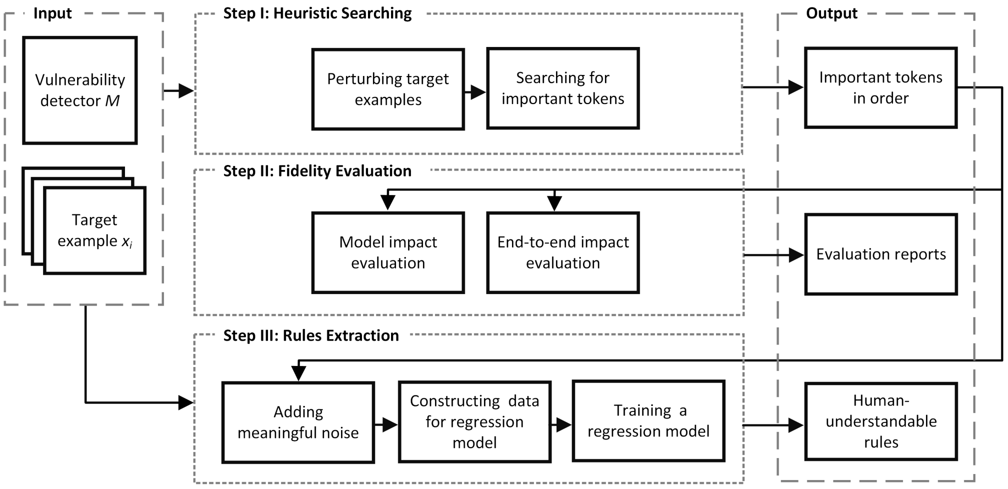

Input
Vulnerability detector M
Target example x_i
Step I: Heuristic Searching
Perturbing target examples
Searching for important tokens
Step II: Fidelity Evaluation
Model impact evaluation
End-to-end impact evaluation
Step III: Rules Extraction
Adding meaningful noise
Constructing data for regression model
Training a regression model
Output
Important tokens in order
Evaluation reports
Human-understandable rules

Fig. 3. A framework for explaining why $M(x_i)$ returns y rather than $\bar{y}$ in three steps

Fig. 3 highlights the framework for explaining why $y \leftarrow M(x_i)$ rather than $\bar{y} \leftarrow M(x_i)$. The framework has three steps.

- Step I: Heuristic searching. This step aims to search for a sequence $\varphi_i = (x_{i,\alpha_1}, \ldots, x_{i,\alpha_\gamma})$ of γ important tokens of example x_i using an interpretation method. These perturbed tokens are ranked according to their contribution in leading to the flipping of the prediction label.
  A concrete method for this purpose is to (i) perturb x_i into some variant examples near the decision boundary, and (ii) identify important tokens as the ones whose perturbations lead to the variant examples have significant impact on the prediction of M.
- Step II: Fidelity evaluation. This step aims to evaluate the fidelity of an interpretation method with respect to M. This can be done by "comparing" the sequence $\varphi_i = (x_{i,\alpha_1}, \ldots, x_{i,\alpha_\gamma})$ of γ important tokens identified by an interpretation method and the sequence $\phi_i = (x_{i,\alpha'_1}, \ldots, x_{i,\alpha'_\gamma})$ of γ important tokens identified by M itself.
  A concrete method is to "compare" the impact of deleting the r important tokens in φ_i on the output of vulnerability detection model M, namely contrasting $M(x_i)$ and $M(x_{i,-\alpha_1-\ldots-\alpha_r})$.
- Step III: Rules extraction. This step aims to extract human-understandable rules for explaining why $y \leftarrow M(x_i)$ rather than $\bar{y} \leftarrow M(x_i)$.
  A concrete method is to add some "meaningful" noises to the afore-identified important tokens of x_i to further generate multiple perturbed examples, train a decision tree-based regression model based on these perturbed examples, and extract rules from this regression model.

It should be noted that our framework focuses on deep learning-based vulnerability detectors with binary classification (i.e., vulnerable vs. not vulnerable) and that the input target examples are extracted from program source code. In the next subsections, we elaborate the three steps of the framework.

3.3 Step I: Heuristic Searching

This step is further divided into two sub-steps: *perturbing target examples* and *searching for important tokens.*

3.3.1 Perturbing Target Examples. Given vulnerability detector M and target example x_i where $y \leftarrow M(x_i)$, this step aims to heuristically search for a sequence $\varphi_i = (x_{i,\alpha_1}, \ldots, x_{i,\alpha_\gamma})$ of γ important tokens such that their perturbation leads to a new example x_i' and $\bar{y} \leftarrow M(x_i')$. There are many methods for perturbing examples, such as the following.

- Perturbation by adding noise to tokens: This is to add some noise $\delta_{i,j}$ to the j-th token in $x_i = (x_{i,1}, \ldots, x_{i,n})$, leading to the feature representation of a new (or variant) example, denoted by $x_{i,+j} = (x_{i,1}, \ldots, x_{i,j-1}, x'_{i,j}, x_{i,j+1}, \ldots, x_{i,n})$, where the perturbation is $x'_{i,j} = x_{i,j} + \delta_{i,j}$. This corresponds to perturbing the j-th feature. We say a noise $\delta_{i,j}$ is *meaningful* if $x'_{i,j}$ preserves the type of $x_{i,j}$, such as API function calls, keywords, operators, delimiters, and constants. For example, a meaningful noise may perturb an operator $x_{i,j} = +$ to operator $x'_{i,j} = -$. This perturbation can be extended to accommodating a sequence of tokens $J = (x_{i,\beta_1}, \ldots, x_{i,\beta_\omega})$, where $\{\beta_1, \ldots, \beta_\omega\} \subset \{1, \ldots, n\}$. By respectively adding noises to the tokens in J, we would obtain the feature representation of a new example, denoted by $x_{i,+J}$.
- Perturbation by deleting tokens: This is to delete the j-th token $x_{i,j}$ from $x_i = (x_{i,1}, \ldots, x_{i,n})$, leading to the feature representation of a new example, denoted by $x_{i,-j} = (x_{i,1}, \ldots, x_{i,j-1}, x_{i,j+1}, \ldots, x_{i,n})$. This perturbation can also be extended to multiple tokens, say a sequence of tokens $J = (x_{i,\beta_1}, \ldots, x_{i,\beta_\omega})$ where $\{\beta_1, \ldots, \beta_\omega\} \subset \{1, \ldots, n\}$. Deleting the tokens in J leads to a new example, denoted by $x_{i,-J}$.

For concise description, we may use x_i' to denote a new example that is obtained by perturbing x_i but *without* specifying what the perturbation is. This notation is useful especially when the specific perturbation method does not matter; for example, when we discuss the degree of perturbations. The degree of perturbation can be measured by using a standard norm, denoted by $||x_i' - x_i||$. In the case of adding noise, the ℓ_1 norm means $||x_i' - x_i|| = \sum_{j=1}^{n} x_{i,j} \oplus x'_{i,j}$, indicating the number of tokens that have been perturbed; in the case of deletion, it means the number of tokens that are deleted.

Definition 3.1 (degree of token association). Given a vulnerability detector M and a target example $x_i = (x_{i,1}, \ldots, x_{i,n})$, we define the degree that the h-th token $x_{i,h}$ is associated to a token sequence J (of one or more tokens) as

$$|M_1(x_{i,-J}) - M_1(x_{i,-J-h})| - |M_1(x_i) - M_1(x_{i,-h})|. \tag{1}$$

Intuitively, the degree of token association given by Eq. (1) captures that the *extra* "damage" to the classification accuracy that is incurred by deleting a token together with other token(s), when compared with the "damage" to the classification accuracy that is incurred by deleting the specific token alone.

3.3.2 Searching for Important Tokens. Given example x_i and vulnerability detector M, in order to search for γ *important tokens*, there are $n \times (n-1) \times \cdots \times (n-\gamma+1)$ candidates, meaning that searching is feasible only for small constant γ. Therefore, we need some heuristic strategies. On the other hand, a competent search algorithm should take into consideration the afore-defined *token association* because the tokens may not be independent of each other. This leads to the following notion of *token combinations.*

Definition 3.2 (token combination). Given vulnerability detector M and target example $x_i = (x_{i,1}, \ldots, x_{i,n})$, a *token combination* C is a sequence of associated tokens in x_i, denoted by $C = (x_{i,t_1}, \ldots, x_{i,t_\kappa})$, meaning that x_{i,t_j} is possibly associated to x_{i,t_h} for $1 \leq t_1 \leq t_j, t_h \leq t_\kappa \leq n$.

Algorithm 1 is a concrete method for searching for important tokens by perturbing x_i to x_i' while bounding the degree of perturbation from above by $||x_i' - x_i|| \leq \Theta$, where $|| \cdot ||$ is an appropriate norm (e.g., the ℓ_1-norm as mentioned in Section 3.3.1) and Θ is the perturbation upper bound. In order to increase the chance for the algorithm to identify truly important tokens, we search for N sequences of token combinations, denoted by $G_1, \ldots, G_N$, where $G_p = (C_{p,1}, \ldots, C_{p,\xi_p})$ is a sequence of token combinations as defined in Definition 3.2 and $1 \leq p \leq N$. Different sequences of token combinations start with different token combinations, meaning $C_{p,1} \neq C_{q,1}$ for $1 \leq p, q \leq N$ and $p \neq q$. The algorithm first initializes N sequences of token combinations as empty sequences (Lines 1-3 of Algorithm 1). For each sequence of token combinations G_p, the algorithm searches for a token combination C by using function *searchTokenCombination*, which will be detailed as Algorithm 2. We use a flag Z to mark the target of the searching process in a loop (Line 7), where "Z = True" means that the algorithm finds a token combination whose perturbation can cause the flipping of the predicted label. In the next loop, function *searchTokenCombination* searches for the token combination that can: (i) flip the predicted label $M(x_{i,-J})$ after deleting a small number of tokens while leading to the largest change to $M_1(x_{i,-J})$; or (ii) lead to the largest change to $M_1(x_{i,-J})$ after reaching the perturbation upper bound, where J is the sequence of tokens corresponding to the token combinations of G_p. When a loop ends with Z = False, the algorithm does not find any token combination whose perturbation causes the flipping of the label. As an alternative, function *searchTokenCombination* searches for the token combination whose perturbation can cause the largest change to $M_1(x_{i,-J})$. Then, the token combination C is appended to the end of G_p (Lines 9-16 of Algorithm 1). The process of searching for token combinations of G_p repeats until there are at least Ω important tokens in the token combinations of G_p (Lines 8-22 of Algorithm 1), When each of the N sequences of token combinations has at least Ω important tokens, the search process ends (Lines 5-23 of Algorithm 1). Finally, the algorithm computes the token importance of each token in φ_i (Lines 24-26 of Algorithm 1) according to the following Definition 3.3; this allows to sort the important tokens by their importance in descending order, leading to the sequence of important tokens φ_i for x_i as the output of the heuristic searching (Lines 27-28 of Algorithm 1).

For each token in combination C, Algorithm 1 computes its importance in the sequence of token combinations G_p and appends these tokens to the end of sequence φ_i. The algorithm divides the important tokens in φ_i into two groups: *positive tokens* and *negative tokens*, depending on the *direction* of $M_1(x_i')$ incurred by the perturbation (rather than depending on whether the label is flipped or not). Specifically, positive tokens are the ones whose perturbations make $M_1(x_i')$ change in the direction towards the opposite of the predicted label $M(x_i)$, namely $1 - M(x_i)$; negative tokens are the tokens whose perturbations can make $M_1(x_i')$ change in the direction towards the predicted label $M(x_i)$. Note that positive tokens have odd indices in the sequence G_p of token combinations and negative tokens have even indices in G_p. If x_i is perturbed to x_i' by the tokens in token combination C that contains x_{i,α_j}, the token importance of x_{i,α_j} in G_p depends on (i) the importance of token combination C, denoted by I_C, and (ii) the number of tokens in C, denoted by s. The more important C is and the fewer tokens C contains, the more important the x_{i,α_j} is. Formally, we have

Definition 3.3 (token importance). Denote by $I'_{p,x_{i,\alpha_j}}$ the importance of x_{i,α_j} in G_p, which is defined as:

$$I'_{p,x_{i,\alpha_j}} = \begin{cases} \lambda \frac{I_C}{s}, & \text{if } x_{i,\alpha_j} \text{ is in } G_p \text{ and the index of } C \text{ is odd} \\ -\lambda \frac{I_C}{s}, & \text{if } x_{i,\alpha_j} \text{ is in } G_p \text{ and the index of } C \text{ is even} \\ 0, & \text{if } x_{i,\alpha_j} \text{ is not in } G_p \end{cases} \tag{2}$$

where I_C is the importance of token combination C, namely the change $|M_1(x_{i,-J-C}) - M_1(x_{i,-J})|$ incurred by C, s is the number of tokens in C, and λ is the attenuation factor to balance the

Algorithm 1 Heuristic searching with token deletion

```
Input: $M$ (vulnerability detector); $x_i$ (example of interest); $N$ (the number of sequences of token
    combinations); $\Omega$ (the number of tokens in each sequence of token combinations); $\Theta$ (per-
    turbation upper bound); $k$ (the number of reserved token combinations during each loop of
    searching)
Output: A sequence of $\gamma$ important tokens $\varphi_i = (x_{i,\alpha_1}, \ldots, x_{i,\alpha_\gamma})$ for $x_i$ with respect to our inter-
    pretation method
 1: for $p \leftarrow 1$ to $N$ do
 2:     Initialize each sequence of token combinations $G_p$ as an empty sequence;
 3: end for
 4: Initialize $\varphi_i$ as an empty sequence; (for storing $\gamma$ important tokens in descending order)
 5: for $p \leftarrow 1$ to $N$ do
 6:     Initialize $J$ as an empty sequence; (for storing important tokens recorded by $G_p$)
 7:     $Z \leftarrow$ True; (a flag for marking the searching target)
 8:     while $|J| < \Omega$ do
 9:         $C \leftarrow$ searchTokenCombination($M$, $x_{i,-J}$, $\Theta$, $Z$, $k$), where $C$ could change $M_1(x_{i,-J})$ the
    most
10:         if $M(x_{i,-J}) = M(x_{i,-J-C})$ and $Z$ = True then
11:             $Z \leftarrow$ False
12:         else
13:             $Z \leftarrow$ True
14:         end if
15:         Append all tokens in $C$ to the end of $J$;
16:         Append $C$ to the end of $G_p$ (the first token combination in each $G_p$ should be different);
17:         Compute the importance of token combination $I_C$ according to the change in $M_1(x'_i)$ as
    caused by $C$;
18:         for each token $f \in C$ do
19:             Compute the token importance of $f$ in $G_p$, denoted by $I'_{p,x_{i,f}}$;
20:             Append $f$ to the end of $\varphi_i$;
21:         end for
22:     end while
23: end for
24: for each token $x_{i,\alpha_j} \in \varphi_i$ do
25:     Compute the token importance $I_{x_{i,\alpha_j}}$ by the token importance of $x_{i,\alpha_j}$ in $N$ sequences of
    token combinations (i.e., $I'_{1,x_{i,\alpha_j}}, \ldots, I'_{N,x_{i,\alpha_j}}$);
26: end for
27: Sort the tokens in $\varphi_i$ by their importance in the descending order;
28: return $\varphi_i$;
```

importance of different token combinations (e.g., $\lambda = 1/q$ with q being the index of the token combination in a sequence).

Intuitively, a token's importance is independent of the G_p's to which it belong. This suggests the following method for computing a token's importance. For token $x_{i,\alpha_j} \in \varphi_i$ where $1 \leq \alpha_j \leq n$, its importance, denoted by $I_{x_{i,\alpha_j}}$, is defined as

$$I_{x_{i,\alpha_j}} = sign\left(\sum_{p=1}^{N}\left|I'_{p,x_{i,\alpha_j}}\right|\right), \tag{3}$$

where the *sign* function is meant to make the sign (i.e., positive or negative) of $I_{x_{i,\alpha_j}}$ the same as the sign of the largest $|I'_{p,x_{i,\alpha_j}}|$ among the N sequences of token combinations $G_1, \ldots, G_N$.

Recall that a positive token has a positive importance and has an odd index in some sequence of token combinations, and that a negative token has a negative importance and has an even index in some sequence of token combinations. Note also that when a token x_{i,α_j} is in the token combination with odd (resp. even) index in G_p and also in the token combination with even (resp. odd) index in G_q, the sign of token importance depends on the sign of the one who has a larger absolute value.

Now we present the details of function *searchTokenCombination* in Algorithm 1 as the following Algorithm 2. Denote by F_i the set of all tokens in x'_i, by $A = (a_1, \ldots, a_\theta)$ a sequence of token combinations $a_1, \ldots, a_\theta$, and by C the token combination that is being searched. For each token combination in A', the algorithm treats each token in F_i as a token combination a and append it to A (Lines 8-12) in the first loop of Lines 10-23. If multiple token combinations lead to the flipping of the predicted label $M(x'_i)$, namely $M(x'_i) \neq M(x'_{i,a})$, then C is updated to be the token combination that leads to the largest change to $M_1(x'_i)$ (Lines 13-22), meaning $|M_1(x'_{i,-C}) - M_1(x'_i)|$ is maximized. In the loop of Lines 10-23, the algorithm appends each token f that has yet to be perturbed into a, which is then appended to A (Lines 8-12). If (i) "Z = True" and there are token combinations whose perturbation can cause the flipping of $M(x'_i)$ or (ii) "Z = False", C is updated to the token combination whose perturbation can cause the flipping of $M_1(x'_i)$ (Lines 13-22). Then, the token combinations in A are sorted in the descending order when $M(x'_i) = 1$ and in the ascending order when $M(x'_i) = 0$, and the first k token combinations are retained (Lines 25-31). The algorithm repeats the *while* loop (Lines 5-32) until identifying a token combination that flips the predicted label $M(x'_i)$ or the perturbation upper bound has been reached (i.e., the number of tokens in each token combination in A equals to Θ). If Θ tokens of x_i are deleted and $M(x'_i)$ is not flipped, C is updated to a_1, which leads to the largest changes to $M_1(x'_i)$. The time complexity of Algorithm 2 is $O(k * |F_i| * \theta)$, where $|F_i|$ is the size of set F_i.

Illustrating Algorithms 2. Fig. 4 uses a specific x_i related to the uncontrolled memory allocation vulnerability mentioned above to illustrate the searching process for obtaining G_1, where the parameters include $\Theta = 5$ and $k = 4$. In this case, the predicted label $0 \leftarrow M(x_i)$ and $0.01 \leftarrow M_1(x_i)$. The Algorithm 1 initializes G_1 as an empty sequence and uses Algorithm 2 to search for the first token combination as follows. The algorithm treats each token in x_i as a token combination and identifies one or multiple token combinations that lead to the flipping of the predicted label, namely $1 \leftarrow M(x_{i,-93})$. Since token $x_{i,93}$ (i.e., &&) leads to the flipping of the the predicted label, namely $1 \leftarrow M(x_{i,-93})$ whereas $0 \leftarrow M(x_i)$, and leads to the largest change to $M_1(x_i)$ (i.e., 0.86-0.01=0.85), the token combination "`(&&)`" is appended to G_1. Then, the algorithm searches for the second token combination for G_1 based on $x_{i,-93}$, where "93" is the index of "&&" in x_i. The algorithm traverses the tokens in $x_{i,-93}$ individually, and finds that no token can lead $M_1(x_{i,-93})$ to flip the model prediction. Therefore, the algorithm retains the k most important token combinations, namely "`(myString)`", "`(size_t)`", "`([)`" and "`(;)`". The algorithm continues the search based on each of these token combinations. Consequently, the algorithm identifies several token combinations that lead to the flipping. Among these token combinations, the token combination "(`myString`, `HELLO_STRING`)" leads to the largest change $\left|M_1(x_{i,-93-98-110}) - M_1(x_{i,-93})\right| = |0.28 - 0.86| = 0.58$ and is therefore appended to G_1.

Algorithm 2 Function *searchTokenCombination* in Algorithm 1 (for the purpose of searching for a token combination)

```
Input: $M$ (vulnerability detector); $x'_i$ (example); $\Theta$ (perturbation upper bound); $Z$ (indicating the
    search target); $k$ (number of reserved token combinations at each loop of searching)
Output: The token combination $C$ which could (i) flip the predicted label $M(x'_i)$ by deleting the
    fewest tokens while changing $M_1(x'_i)$ the most or (ii) change $M_1(x'_i)$ the most if $\Theta$ tokens of $x_i$
    are deleted and $M(x'_i)$ is not flipped
 1: function searchTokenCombination($M$, $x'_i$, $\Theta$, $k$)
 2:     $F_i \leftarrow$ the set of all tokens in $x'_i$;
 3:     Initialize the sequence of token combinations $A = (a_1, \ldots, a_\theta)$ as a sequence that contains
    only an empty token combination;
 4:     Initialize token combination $C$ as an empty sequence;
 5:     while ($C$ is an empty sequence) and (the number of tokens in each token combination in
    $A < \Theta$) do
 6:         $A' \leftarrow A$;
 7:         $A \leftarrow$ an empty sequence;
 8:         for each token combination $a \in A'$ do
 9:             $F'_i \leftarrow$ the set of all tokens in $a$;
10:             for each token $f \in F_i - F'_i$ do
11:                 Append $f$ to token combination $a$;
12:                 Append $a$ to the sequence of token combination $A$;
13:                 if $Z$ = True and $M(x'_i) \neq M(x'_{i,-a})$ then
14:                     if ($C$ is an empty sequence) or ($|M_1(x'_{i,-a}) - M_1(x'_i)| > |M_1(x'_{i,-C}) - M_1(x'_i)|$)
    then
15:                         $C \leftarrow a$;
16:                     end if
17:                 end if
18:                 if $Z$ = False then
19:                     if ($C$ is an empty sequence) or ($M(x'_i)$=1 and $M_1(x'_{i,-a}) > M_1(x'_{i,-C})$) or
    ($M(x'_i)$=0 and $M_1(x'_{i,-a}) < M_1(x'_{i,-C})$) then
20:                         $C \leftarrow a$;
21:                     end if
22:                 end if
23:             end for
24:         end for
25:         if $Z$ = True and $M(x'_i) = 1$ then
26:             Sort the token combinations in $A$ by $M_1(x'_{i,-a})$ in descending order;
27:         end if
28:         if $Z$ = True and $M(x'_i) = 0$ then
29:             Sort the token combinations in $A$ by $M_1(x'_{i,-a})$ in ascending order;
30:         end if
31:         $A \leftarrow$ the sequence of token combinations $(a_1, \ldots, a_k)$;
32:     end while
33:     if $C$ is an empty sequence then
34:         $C \leftarrow a_1$;
35:     end if
36:     return $C$;
37: end function
```

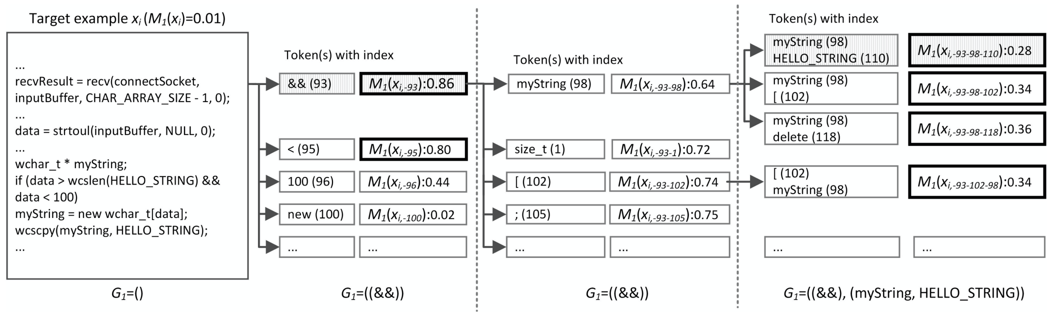

```
...
recvResult = recv(connectSocket,
inputBuffer, CHAR_ARRAY_SIZE - 1, 0);
...
data = strtoul(inputBuffer, NULL, 0);
...
wchar_t * myString;
if (data > wcslen(HELLO_STRING) &&
data < 100)
myString = new wchar_t[data];
wcscpy(myString, HELLO_STRING);
...
```

Fig. 4. Illustrating the search of token combinations in G_1, where boxes with texture represent the token combinations in G_1, boxes without texture do not belong to G_1, bold boxes highlight label flipping (with threshold $\tau = 0.5$), and a number in parentheses indicates a token's location in the target example of interest.

Fig. 5 illustrates the calculation of token importance $I_{x_{i,98}}$ involving two sequences of token combinations for target example x_i in Fig. 4. According to Eq. (3), the importance of token $x_{i,98}$ (i.e., `myString`) is $I_{x_{i,98}} = sign\left(I'_{1,x_{i,98}} + I'_{2,x_{i,98}}\right)$. According to Eq. (2), we have $I'_{1,x_{i,98}} = -\lambda\frac{I_{C_{1,2}}}{s}$. Since the index of the token combination involving $x_{i,98}$ $C_{1,2}$ is 2, $C_{1,2}$ has 2 tokens, and $I_{C_{1,2}} = 0.58$ $(|M_1(x_{i,-93-98-100}) - M_1(x_{i,-93})|)$, the importance of $x_{i,98}$ in G_1 is $I'_{1,x_{i,98}} = -\frac{1}{2} \times \frac{0.58}{2} = -0.145$. Similarly, the importance of $x_{i,98}$ in G_2 is $I'_{2,x_{i,98}} = -\lambda\frac{I_{C_{2,2}}}{s} = -\frac{1}{2} \times \frac{0.66}{2} = -0.165$. Therefore, $I_{x_{i,98}} = -|I'_{1,x_{i,98}} + I'_{2,x_{i,98}}| = -0.31$ and $x_{i,98}$ is a negative token.

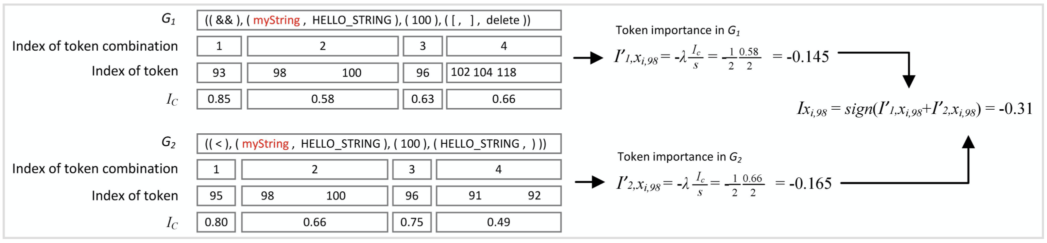

G_1
((&&), (myString , HELLO_STRING), (100), ([,] , delete))
Index of token combination
1 | 2 | 3 | 4
Index of token
93 | 98 100 | 96 | 102 104 118
I_C
0.85 | 0.58 | 0.63 | 0.66
G_2
((<), (myString , HELLO_STRING), (100), (HELLO_STRING ,)))
Index of token combination
1 | 2 | 3 | 4
Index of token
95 | 98 100 | 96 | 91 92
I_C
0.80 | 0.66 | 0.75 | 0.49
Token importance in G_1
$I'_{1,x_{i,98}} = -\lambda\frac{I_c}{s} = -\frac{1}{2}\frac{0.58}{2} = -0.145$
$I_{x_{i,98}} = sign(I'_{1,x_{i,98}}+I'_{2,x_{i,98}}) = -0.31$
Token importance in G_2
$I'_{2,x_{i,98}} = -\lambda\frac{I_c}{s} = -\frac{1}{2}\frac{0.66}{2} = -0.165$

Fig. 5. Illustrating the calculation of the importance of token "myString" $I_{x_{i,98}}$ by two sequences of token combinations (i.e., G_1 and G_2) for target example x_i in Fig. 4.

3.4 Step II: Fidelity Evaluation

In the context of interpretability, the notion of *fidelity* [37] has been used to describe the degree at which an *interpretation method* approximates the behavior of the target model (i.e., M in the context of the present paper). For vulnerability detection, we use the important tokens identified by Algorithm 1 for model interpretation and define fidelity as follows.

Definition 3.4 (Fidelity). Intuitively, fidelity is the degree at which the sequence of the α_γ important tokens identified by an interpretation method, namely $\varphi_i = (x_{i,\alpha_1}, \ldots, x_{i,\alpha_\gamma})$, is "similar" to the sequence of the α'_γ important tokens considered by the target vulnerability detector M, denoted by $\phi_i = (x_{i,\alpha'_1}, \ldots, x_{i,\alpha'_\gamma})$, where $1 \leq \alpha'_1, \ldots, \alpha'_\gamma \leq n$ and the α'_γ important tokens are also sorted according to their importance in descending order. Moreover, the higher the similarity between φ_i and ϕ_i, the higher fidelity the interpretation method. Because of the black-box nature of deep learning models, ϕ_i is often unknown. Nevertheless, important tokens ϕ_i would affect the decision-making of the model. The similarity between φ_i and ϕ_i can be approximated by the impact that is incurred when deleting the tokens in φ_i on the output of the vulnerability detection model, as follows: (i) the difference between the classification results of target models fine-tuned by training examples using vs. not using the important tokens in φ_i; and (ii) the difference between the classification results of target examples when using vs. not using the important tokens in φ_i. The bigger these differences, the greater the impact on the output of the classification model, and the higher the similarity between φ_i and ϕ_i.

3.4.1 Model Impact Evaluation In order to characterize the impact of important tokens on the vulnerability detection model, we quantify the fidelity of the interpretation method from the change in a vulnerability detector's output. Given a set of examples D, let M' be the vulnerability detector obtained by *fine-tuning* M using the examples in D except the important tokens identified by Algorithm 1 and M^+ be the vulnerability detector obtained by fine-tuning M using the examples in D, where "fine-tuning M" means to use the examples in D, which is different from the data set that was used for training M. More specifically, in order to fine-tune the vulnerability detector M, we first randomly select a set of examples D from the data set. For each example $x_d \in D$, after obtaining the important tokens from Steps I, we remove the first r important positive tokens to obtain $x_{d,-\varphi_d^r}$, where φ_d^r is the set of first r important tokens, and obtain a new data set D'. Then we use D' to fine-tune vulnerability detector M to obtain M' and use D to fine-tune vulnerability detector M to obtain M^+. For each target example $x_u \in U$, where $U \neq D$, we obtain $M_1'(x_u)$, $M_1^+(x_u)$, $M'(x_u)$, and $M^+(x_u)$. Finally, we get the difference between the outputs of the two models on U. The greater the difference between the M' and M^+, the better the vulnerability detector understood by the interpretation method.

Definition 3.5 (Class Change). Given a vulnerability detector M and a dataset $U = \{x_1, x_2, \ldots, x_T\}$, where T is the number of examples in U, the *Class Change* (CC) is defined as the differences between the classification results of M^+ and the classification results of M' on the examples in U. Intuitively, the greater the difference between the outputs of two detectors for the same input examples, the more important the deleted tokens, and the higher the fidelity.

For a target example x_u ($1 \leq u \leq T$), let $|TP'|$ be the number of examples that satisfy $M'(x_u) = M^+(x_u) = 1$, $|FP'|$ be the number of examples that satisfy $M'(x_u) = 1$ and $M^+(x_u) = 0$, $|FN'|$ be the number of examples that satisfy $M'(x_u) = 0$ and $M^+(x_u) = 1$, and $|TN'|$ be the number of examples that satisfy $M'(x_u) = M^+(x_u) = 0$. The following four metrics are used to evaluate the CC. (i) false-positive rate of M' with respect to M^+, denoted by $FPR' = \frac{|FP'|}{|FP'|+|TN'|}$; (ii) false-negative rate of M' with respect to M^+, denoted by $FNR' = \frac{|FN'|}{|TP'|+|FN'|}$; (iii) accuracy of M' with respect to M^+, denoted by $A' = \frac{|TP'|+|TN'|}{|TP'|+|FP'|+|TN'|+|FN'|}$; (iv) the overall effectiveness F1-measure of M' with respect to M^+, denoted by $F1' = \frac{2|TP'|}{2|TP'|+|FP'|+|FN'|}$. Note that a larger FPR', larger FNR', lower A', and lower $F1'$ indicate that the important tokens identified by the interpretation method are more faithful to the important tokens that are implicitly recognized by M.

Definition 3.6 (Vulnerable Probability Change). Given a vulnerability detector M and a test set $U = \{x_1, x_2, \ldots, x_T\}$ where T is the size of the test set), the *Vulnerable Probability Change* (VPC) of an interpretation method is defined as the root mean square error for the probability that vulnerability detector M' predicts target example x_u ($1 \leq u \leq T$) into vulnerable ($y = 1$) and the probability that vulnerability detector M^+ predicts target example x_u into vulnerable ($y = 1$), namely

$$VPC = \sqrt{\frac{1}{T}\sum_{u=1}^{T}(M_1'(x_u) - M_1^+(x_u))^2}. \tag{4}$$

Recall that vulnerability detector M' uses the examples without important tokens to fine-tune M. If the important tokens identified by the interpretation method are indeed important, M' would learn patterns corresponding to the unimportant tokens (i.e., the tokens that are not deleted), which enlarges the difference between M' and M. On the other hand, vulnerability detector M^+ is trained by using the same examples with the same tokens (including the important tokens) as what were used for training M. For target example $x_u \in U$, the greater the difference between $M_1'(x_u)$ and $M_1^+(x_u)$, the greater the difference between the fine-tuned vulnerability detectors M' and M^+, and the more important the tokens identified by the interpretation method. Therefore, the larger VPC, the better important tokens that identified by the interpretation method.

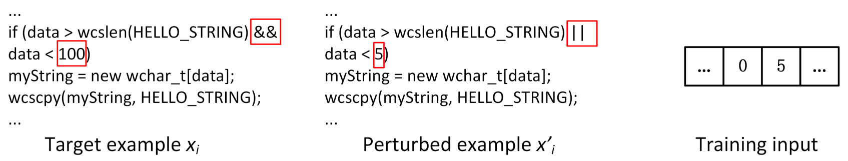
3.4.2 End-to-end Impact Evaluation. In order to evaluate the impact of important tokens on the target examples, we conduct an end-to-end impact evaluation using the idea of *token deduction test* [20]. The basic idea is the following. For each target example x_i, we construct an example $x_{i,-\varphi_i^r}$ by deleting the tokens in φ_i^r from x_i, where φ_i^r is the set of first r important tokens for x_i. Then we obtain $M(x_{i,-\varphi_i^r})$ and calculate the indicator Positive Classification Rate (PCR) [20], which measures the proportion of target examples which satisfy $M(x_{i,-\varphi_i^r}) = M(x_i)$ to the total target examples. If the tokens obtained by Step I mentioned above are accurately selected, deleting φ_i^r from the example x_i will flip the predicted label. Therefore, if the interpretation method has high fidelity, a low PCR will be returned during the token deduction test.

3.5 Step III: Rules Extraction

After obtaining some important tokens, we aim to extract some human-understandable rules for each target example. For this purpose, we use a decision tree-based regression model to find the impact of important tokens on model decision making, which involves the following three steps.

Step III.1 Adding meaningful noise. In order to assure efficiency, we add meaningful noise to target examples by replacing important tokens, where meaningful noise can make the program remain correct syntax when added to the token. For each target example x_i, we structure $x_{i,+j}$ where $x_{i,j}$ corresponds to the important tokens obtained in Step I. We propose the following strategies to automatically add noise while trying to follow the syntax of the program. For constants that take continuous values, we perform intra-region random replacements (i.e., replacing it with a random value in the domain of token). For tokens that are not constants but take discrete (or categorical) values (e.g., API function calls, keywords, operators, and delimiters), we replace them with a token that is randomly selected from the token's domain. perform the random replacement with tokens of the same type. For example, a token that is a number "`10`" can be replaced with "`5`", "`9`", "`11`", and "`20`"; a relational operator "`>`" can be replaced with another relational operator such as "`<`", "`!=`", and "`==`".

Step III.2 Constructing data for regression model. The training data for the decision tree-based regression model include training inputs and expected outputs. Each training input is a noise addition condition for the perturbed example obtained from the previous step, thus each dimension corresponds to a change in an important token. and the corresponding expected output is the probability that the vulnerability detector M predicts the perturbed example into vulnerable. For constants, the dimension value is $x_{i,j}'$ resulting from noise addition; for other tokens, we use "1" or "0" for the dimension value, where "1" means that the type of token $x_{i,j}'$ is the same as the type of token $x_{i,j}$ and "0" otherwise. Fig. 6 illustrates an example of decision tree-based regression model for target example x_i in Fig. 4, where meaningful noise is added to token $x_{i,93}$ (i.e., "&&") and token $x_{i,96}$ (i.e., "100"). Specifically, token "&&" is replaced by "||" in the perturbed example x'_i, meaning that the dimension value of the training input is 0; token "100" is replaced by "5", meaning that the dimension value in the training input is 5.


```
...
if (data > wcslen(HELLO_STRING) &&
data < 100)
myString = new wchar_t[data];
wcscpy(myString, HELLO_STRING);
...
```
Target example x_i

```
...
if (data > wcslen(HELLO_STRING) ||
data < 5)
myString = new wchar_t[data];
wcscpy(myString, HELLO_STRING);
...
```
Perturbed example x'_i

...	0	5	...

Training input

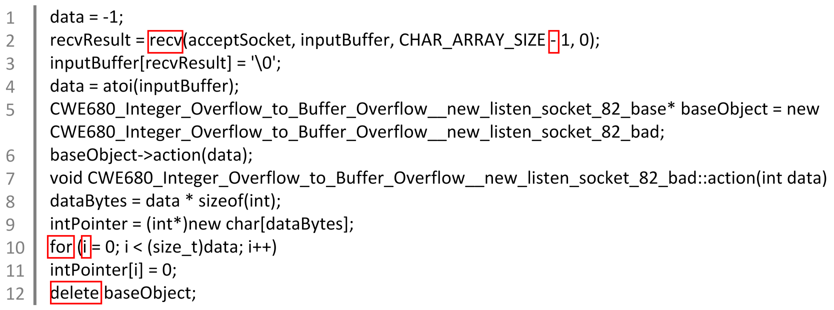
Fig. 6. An example to illustrate the training input for rules extraction for target example x_i in Fig. 4. The token "&&" is replaced by "||", thus the value of corresponding dimension in the training input is 0; the token "100" is replaced by "5", thus the value of corresponding dimension in the training input is 5.

Step III.3 Training a regression model. For each target example x_i, we use the training data from the previous step to train a decision tree-based regression model $Tree_i$ which can fit the vulnerability detector M as much as possible. In a decision tree, each non-leaf node corresponds to an `if` condition with an important token which determines whether the important token is used in the example x_i, each leaf node corresponds to a prediction, and each path from the root to a leaf represents a decision. By synthesizing the `if` conditions corresponding to the nodes on a root-to-leaf path, we can extract an `if-then-else` rule corresponding to the path. Since a decision tree is easy to understand, we propose leveraging the decision tree to extract the `if-then-else` rules to explain the prediction of a target example. Recall that the *Program Dependency Graph* (PDG) is a graphical representation of the program whose edges represent the data dependency and control dependency of statements in the program. Algorithm 3 shows the process of extracting an understandable vulnerability rule. Based on the regression values corresponding to different branches in the decision tree, we select the tokens corresponding to the intersections of the branches with large differences as the tokens for generating the rules. According to the tokens in the `if-then-else` rule extracted from the decision tree, we can find the corresponding statements and the data dependency and control dependency in PDG. Then domain experts can summarize human-understandable rules by synthesizing the if-then-else rules composed of important tokens and the data dependency and control dependency in the PDG.

Fig. 7 shows a decision tree and a subgraph of PDG for a target example. According to Fig. 7(b), when at least one of '`recv`' and '`for`' is used in x_i (i.e., the corresponding `if` condition is `True`), the regression value range is 0.82-0.96; when '`recv`' and '`for`' is not used in x_i (i.e., the corresponding `if` condition is `False`), the regression value range is 0.69-0.82. Since these branches intersect with '`recv`' and '`for`', the sequence of tokens J=('`recv`', '`for`'). The `if-then-else` rule *BranchRule* is "if '`recv`' or '`for`', then vulnerable". These tokens are located in Line 2 and Line 10 of this example respectively in Fig. 7(a), which has a data dependency from variable '`inputBuffer`' in Line 2 to '`data`' in Line 10 (denoted by *Depend*), as shown in Fig. 7(c). *BranchRule* and *Depend* can help domain experts to summarize a understandable rule as follows: *If the external input from* `recv` *in* `for` *loop (e.g., the "data" in* `for` *loop is data-dependent on parameter "inputBuffer" in* `recv`*) is used, the example is vulnerable.*

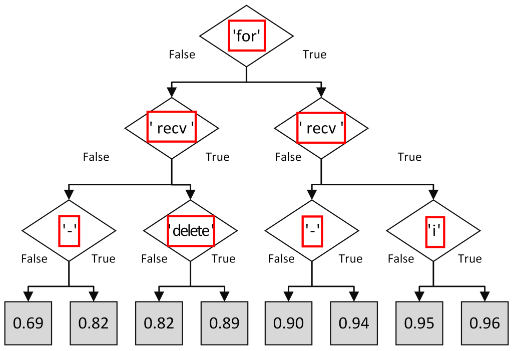
Algorithm 3 Extract an understandable vulnerability rule

```
Input: $Tree_i$ (decision tree of example $x_i$); $PDG_i$ (PDG of example $x_i$)
Output: An understandable vulnerability rule $Rule_i$ of example $x_i$
1: $J \leftarrow$ a sequence of tokens corresponding to the intersection of the branches with large differences between regression values in $Tree_i$;
2: $BranchRule \leftarrow$ a set of if-then-else rules according to the branches of $Tree_i$;
3: $S \leftarrow$ a set of statements where the tokens in $J$ are located;
4: $Depend \leftarrow$ the data dependency and control dependency of statements in $S$ according to $PDG_i$;
5: $Rule_i \leftarrow$ summarize an understandable rule for $x_i$ by synthesizing $branchRule$ and $Depend$;
6: return $Rule_i$;
```

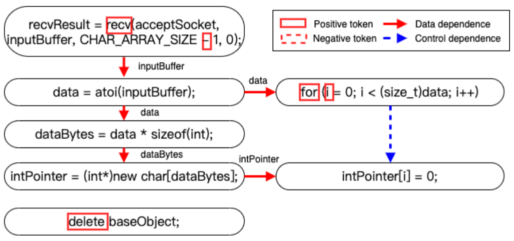


```
data = -1;
recvResult = recv(acceptSocket, inputBuffer, CHAR_ARRAY_SIZE - 1, 0);
inputBuffer[recvResult] = '\0';
data = atoi(inputBuffer);
CWE680_Integer_Overflow_to_Buffer_Overflow__new_listen_socket_82_base* baseObject = new
CWE680_Integer_Overflow_to_Buffer_Overflow__new_listen_socket_82_bad;
baseObject->action(data);
void CWE680_Integer_Overflow_to_Buffer_Overflow__new_listen_socket_82_bad::action(int data)
dataBytes = data * sizeof(int);
intPointer = (int*)new char[dataBytes];
for (i = 0; i < (size_t)data; i++)
intPointer[i] = 0;
delete baseObject;
```

(a) A target example



(b) Trained decision tree


recvResult = recv(acceptSocket, inputBuffer, CHAR_ARRAY_SIZE - 1, 0);
Positive token
Negative token
Data dependence
Control dependence
inputBuffer
data = atoi(inputBuffer);
data
for (i = 0; i < (size_t)data; i++)
data
dataBytes = data * sizeof(int);
dataBytes
intPointer = (int*)new char[dataBytes];
intPointer
intPointer[i] = 0;
delete baseObject;

(c) Subgraph of program dependency graph

Fig. 7. A trained decision tree and a subgraph of the PDG for a target example: the first 5 important tokens identified by our interpretation method are highlighted, which involve 5 positive tokens that are respectively highlighted by solid boxes. The vulnerability rule can be summarized as: *If the external input from '*`recv`*' in '*`for`*' loop (e.g., the '*`data`*' in '*`for`*' loop is data-dependent on parameter '*`inputBuffer`*' in '*`recv`*') is used, the example is vulnerable.*

4 Case Study

4.1 Vulnerability Detectors and Dataset

In order to demonstrate the usefulness of the framework, we conduct a case study on two deep learning-based vulnerability detectors, VulDeePecker [27] and SySeVR [26]. We choose these two detectors because they operate at the fine granularity of program slices. VulDeePecker is based on the BLSTM model; SySeVR is a framework for using deep learning to detect vulnerabilities and we instantiate it with the CNN model. Their implementations are based on Keras [10] with Tensorflow [2] as the backend.

For VulDeePecker, we use the dataset released by [27], which includes buffer error vulnerabilities and resource management error vulnerabilities in C/C++ programs. The unit for vulnerability detection is called *code gadget*, which is a small code fragment (i.e., some program statements that are related to each other in terms of data dependency) [27]. Each code gadget is an example, which is vulnerable ("1") or not vulnerable ("0"). The dataset has 61,638 examples, including 17,725 vulnerable examples and the others are not vulnerable. We randomly divide the dataset into a training set and a test set with an 80:20 ratio. We randomly select one half of the examples in the test set as the dataset D for interpretation purposes, and use the other half as the dataset U for testing the effectiveness of a vulnerability detector.

For SySeVR, we use the dataset released by [26], which contains 126 types of vulnerabilities in C/C++ programs. An example contains the semantic information induced by data dependency and/or control dependency, while recalling that VulDeePecker only accommodates data dependency. The unit for vulnerability detection is *code gadget*, and a code gadget is vulnerable ("1") or not vulnerable ("0"). The dataset has 420,627 examples and we randomly divide the dataset into a training set and a test set with an 80:20 ratio. We randomly select 3,000 examples in the test set as the dataset D for interpretation purposes, and use the other examples as the dataset U to test the effectiveness of a vulnerability detector.

4.2 Research Questions for Evaluating the Usefulness of the Framework

Our experiments focus on answering the following three research questions.

- RQ1: Can the framework identify important tokens, and how effective is the framework in evaluating the model impact on fidelity?
- RQ2: How effective is the framework in evaluating the end-to-end impact of fidelity?
- RQ3: Can the framework extract human-understandable rules to explain why $y \leftarrow M(x_i)$ rather than $\bar{y} \leftarrow M(x_i)$?

Evaluation Metrics. We use the following metrics to evaluate the effectiveness of a vulnerability detector. Let $|TP|$ be the number of examples with vulnerabilities detected correctly, $|FP|$ be the number of examples with false vulnerabilities detected, $|FN|$ be the number of examples with true vulnerabilities undetected, and $|TN|$ be the number of examples with no vulnerabilities undetected. The metric $FPR = \frac{|FP|}{|FP|+|TN|}$ represents the proportion of false-positive examples in the not vulnerable example set. The metric $FNR = \frac{|FN|}{|TP|+|FN|}$ represents the proportion of false-negative examples in the vulnerable example set. The metric $A = \frac{|TP|+|TN|}{|TP|+|FP|+|TN|+|FN|}$ represents the proportion of correctly predicted examples. The metric $F1 = \frac{2|TP|}{2|TP|+|FP|+|FN|}$ measures the overall effectiveness of the vulnerability detector. These four metrics show the effectiveness of vulnerability detection from different perspectives. The closer FPR and FNR are to 0, the better; the closer A and F1 are to 1, the better.

For fidelity evaluation, we choose LEMNA [20] and Kernel SHAP [30] methods for comparison. These two methods are able to give tokens sorted by importance and distinguish between positive and negative tokens. They have been used for PDF malware classification, function start detection in binary reverse-engineering, and image classification, but their performance on vulnerability detection is unclear. In addition, we also construct a random token selection method for comparison. Given a target example, the random method randomly selects tokens as the important tokens.

4.3 Experiments and Results

4.3.1 Experiments for Answering RQ1. In order to show the effectiveness of our interpretation method on model fidelity, we first fine-tune the interpreted vulnerability model, then evaluate the model impact of our interpretation method and its variants, and finally compare it with typical interpretation methods.

We use Algorithm 1 to search for important tokens for each target example, while instantiating Algorithm 2 based on the *beam search* method [34]. We set the number of reserved token combinations at each loop of searching for k, dubbed "beam width" in the *beam search* method, to 5. We search for $N = 3$ sequences of token combinations with each sequence having at least $\Omega = 8$ tokens. We set the perturbation upper bound $\Theta = 5$. Therefore, we can extract at least 8 important tokens for each example. We obtain important tokens for one half of the examples in dataset D, then delete the first $r = 5$ important positive tokens from those of each example in D so as to form a new dataset D'. Based on the vulnerability detector, we get BLSTM$^+$ by using dataset D to train 1 epoch, and by using D' to train 1 epoch with same learning rate to obtain BLSTM'. Similarly, we construct CNN$^+$ and CNN'.

Tables 2 and 3 compare the effectiveness of multiple vulnerability detectors. We observe that the vulnerability detector using BLSTM (denoted by BLSTM) achieves a relatively high effectiveness (i.e., an 89.57% accuracy and a 83.07% F1). When we use one half of the examples in the test set to fine-tune the BLSTM detector to obtain a new model BLSTM$^+$, we observe that there is no much difference between the effectiveness of the BLSTM detector and that of the BLSTM$^+$ detector. Similarly, there is no much difference between the effectiveness of the CNN detector and that of the CNN$^+$ detector. This indicates that the examples without modification do not significantly affect the effectiveness of the model regardless the presence or absence of fine-tuning.

Table 2. Model impact evaluation results of our interpretation method and its variants on BLSTM

Model	Detection				Fidelity Evaluation				
	FPR(%)	FNR(%)	A(%)	F1(%)	FPR'(%)	FNR'(%)	A'(%)	F1'(%)	VPC
BLSTM	6.90	18.20	89.57	83.07	-	-	-	-	-
BLSTM$^+$	5.96	19.01	89.96	83.46	-	-	-	-	-
BLSTM' (our method)	1.67	75.65	75.19	38.04	0.33	71.02	78.87	44.66	0.29
BLSTM' with Factor 1	5.34	23.29	89.04	81.41	1.43	9.44	96.21	93.37	0.10
BLSTM' with Factor 2	2.69	63.04	78.44	51.74	0.58	55.84	83.16	60.68	0.27

After using our interpreted method to identify the important tokens, we delete these important tokens and use them to fine-tune the BLSTM model and the CNN model to obtain a BLSTM' detector and a CNN' detector. When compared with BLSTM$^+$, the effectiveness of BLSTM' is significantly reduced, leading to a 14.77% lower accuracy and a 45.03% lower F1. This suggests that these important tokens identified by our interpretation method are indeed important, which justifies the high fidelity of our interpretation method. The F1 of the CNN' model drops by 49.39%, so we can draw a similar conclusion.

In order to show the importance of some factors in our interpretation method, we vary the following two factors: (i) involving the token combinations that consider the associations between the tokens in the heuristic searching process (denoted by "Factor 1") and (ii) distinguishing the positive tokens from the negative tokens (denoted by "Factor 2"). When our interpretation method does not involve Factor 1, Algorithm 1 degenerates to the method of [38]. That is, we directly perturb the tokens in a target example x_i one-by-one, and search for the top-5 tokens that can change $M_1(x_i)$ to the largest extent. From Table 2, we observe that the effectiveness of the BLSTM' model with Factor 1 or Factor 2 only is significantly reduced, which indicates that some of the deleted tokens are indeed important.

Table 2 shows the model impact evaluation results of our interpretation method and its variants on BLSTM. We observe that when compared with the interpretation method with Factor 2, our interpretation method can significantly improve the FNR' by 15.18%, reduce the A' by 4.29% and the F1' by 16.02%. This justifies the importance of considering token associations in the process of identifying important tokens. When our interpretation method does not involve Factor 2 (i.e., considering Factor 1 only), the first 5 tokens with the highest absolute token importance are selected for deletion. When compared with the interpretation method considering Factor 1 only, our interpretation method can significantly improve the FNR' by 60.58%, reduce the A' by 17.34% and the F1' by 48.71%. The VPC of our method is 2.90 times of its counterpart when considering Factor 1 only. This may be caused by the fact that the positive tokens contribute more to the prediction of an example. Since the F1' of the BLSTM' model with Factor 2 is less than its counterpart of the BLSTM' model with Factor 1, and the VPC of the BLSTM model with Factor 2 is larger than its counterpart of the BLSTM' model with Factor 2, we conclude that Factor 2 plays a more important role than Factor 1 does.

Table 3. Model impact evaluation results of our interpretation method and its variants on CNN

Model	Detection				Fidelity Evaluation				
	FPR(%)	FNR(%)	A(%)	F1(%)	FPR'(%)	FNR'(%)	A'(%)	F1'(%)	VPC
CNN	1.25	18.69	96.36	85.94	-	-	-	-	-
CNN$^+$	1.20	18.60	96.43	86.16	-	-	-	-	-
CNN' (our method)	0.03	74.40	90.93	40.69	0.03	74.40	90.93	40.69	0.19
CNN' with Factor 1	0.14	64.72	91.04	51.83	0.04	59.64	92.72	57.40	0.16
CNN' with Factor 2	0.00	75.62	89.58	39.01	0.06	72.34	91.15	43.19	0.18

Table 3 summarizes the model impact evaluation results of our interpretation method and its variants on CNN. The target detector using CNN, denoted by CNN model, achieves a 96.36% accuracy and an 85.94% F1. The detection effectiveness of the CNN model is similar to its counterpart of the CNN$^+$ models. We observe that our interpretation method can improve the F1' by 16.71% when compared with the interpretation method with Factor 1, and can reduce the F1' by 2.50% when compared with the interpretation method with Factor 2.

Insight 1. *The framework can effectively identify important tokens. Moreover, the notions of* token association *(Definition 3.1) and* positive tokens *(Section 3.3.2) play important roles.*

In order to compare the effectiveness of BLSTM' (and CNN') using different interpretation methods, we consider the random token selection method as the baseline while the number of deleted tokens is also set to be 5. For the Kernel SHAP method [30], we use 500 perturbed examples to train the linear model for each target example. For LEMNA [20], we set the number of perturbed examples to be 500, the total number of mixture components (i.e., the number of linear regression models in the mixture regression model) to be 6, and the threshold for fused lasso (the penalty term for accommodating feature dependencies) to be 1e-4. The time spent by our method on identifying important tokens of 3,000 examples is 44,048s, which is shorter than that of the LEMNA's (49,864s) but longer than that of the Kernel SHAP's (2,532s).

Table 4. Effectiveness for vulnerability detection using BLSTM with different interpretation methods

Model	Detection				Fidelity Evaluation				
	FPR(%)	FNR(%)	A(%)	F1(%)	FPR'(%)	FNR'(%)	A'(%)	F1'(%)	VPC
Random	6.21	18.94	89.80	83.25	2.42	4.58	96.95	83.24	0.07
Kernel SHAP	1.00	46.21	84.87	68.98	0.00	40.53	88.07	74.59	0.21
LEMNA	6.27	36.27	84.34	71.80	2.70	24.09	91.01	83.24	0.22
Our method (BLSTM)	1.67	75.65	75.19	38.04	0.33	71.02	78.87	44.66	0.29

Table 4 summarizes the effectiveness of BLSTM' models using different interpretation methods. The effectiveness of BLSTM' for our method is significantly reduced compared with other interpretations methods, e.g., the F1 is 36.64% lower than other methods on average. Table 4 summarizes the model impact results of BLSTM' using different interpretation methods. We observe that the VPC of our interpretation method is 1.38 times that of Kernel SHAP and 1.32 times that of LEMNA, and the Class Change metrics of our interpretation method is significantly better than these methods. For example, our interpretation method can improve F1' by 29.93% compared with Kernel SHAP and 38.58% compared with LEMNA.

Table 5. Effectiveness for vulnerability detection using CNN with different interpretation methods.

Model	Detection				Fidelity Evaluation				
	FPR(%)	FNR(%)	A(%)	F1(%)	FPR'(%)	FNR'(%)	A'(%)	F1'(%)	VPC
Random	1.18	19.52	96.32	85.66	0.54	5.12	98.90	95.44	0.05
Kernel SHAP	0.10	56.35	92.21	60.50	0.04	50.47	93.83	66.13	0.15
LEMNA	1.87	15.85	96.22	85.88	1.44	2.31	98.48	94.00	0.06
Our method (CNN)	0.05	77.31	89.39	36.90	0.03	74.40	90.93	40.69	0.19

Table 5 summarizes the effectiveness of CNN' using different interpretation methods. We observe that the VPC of our interpretation method is 1.27 times of the Kernel SHAP's and 3.17 times of the LEMNA's. For Class Change metrics like F1', our interpretation method is better than the other methods. The low fidelity of the existing methods is mainly due to the following two reasons. (i) The association among tokens learned by the deep learning model may mask the importance of tokens, which may lead to an incorrect ranking of important tokens. (ii) These methods are shackled by considering the output of model ($M_1(x')$) as the cumulative sum of contributions from tokens. However, the boundary of the interpreted example is highly non-linear, thus the contribution of one token in slightly dissimilar contexts may be quite different.

Insight 2. *The framework has a higher fidelity than existing methods, because it neither assumes the local decision boundary is linear nor assumes the tokens are independent of each other.*

4.3.2 Experiments for Answering RQ2. In order to investigate the effectiveness of our interpretation method on the end-to-end impact evaluation of fidelity, we conduct the token deduction test on the same set of interpreted target examples D as model impact evaluation, where the number of select tokens r varies from 1 to 5.

Fig. 8 plots the experimental results of the four interpretation methods (i.e., our method, Kernel SHAP, LEMNA, and Random). We observe that our interpretation method is more effective than the existing methods. The PCR of each of the our interpretation methods is decreases when the

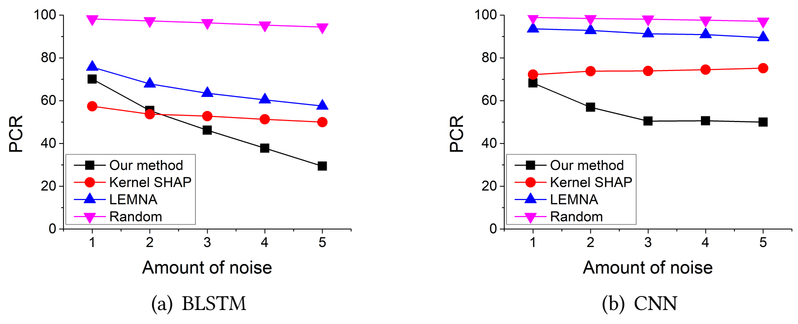


Fig. 8. End-to-end impact evaluation of fidelity for different interpretation methods

amount of noise increases. In the case of BLSTM with the amount of noise being 5, the PCR of our interpretation method reduces to 29.4% while the average PCR of the existing methods is 53.7%. This can be explained by the fact that we focus on the important tokens that contribute to $M(x_i)$ for x_i.

Insight 3. *The framework is significantly more effective than existing methods in the* token deduction test *(proposed in [20]) for end-to-end impact evaluation of fidelity, especially when the amount of noise is large.*

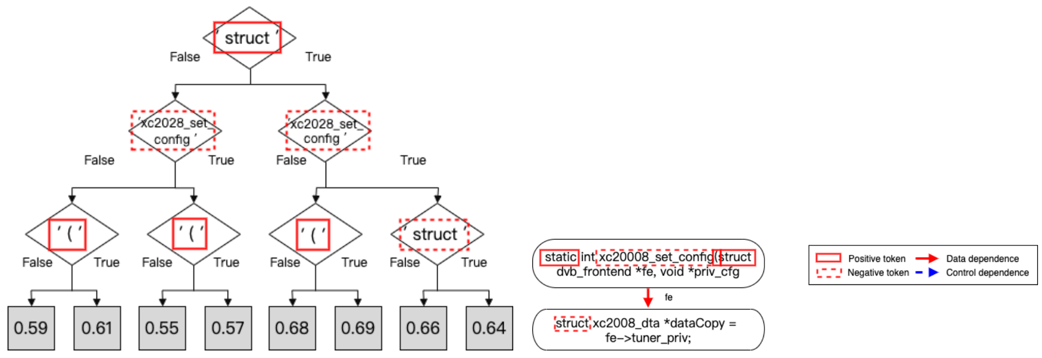
4.3.3 Experiments for Answering RQ3. In order to investigate whether or not our interpretation method can extract human-understandable rules for explaining $y \leftarrow M(x_i)$, we use Step III of the framework to obtain rules from the examples. For each target example, the first 5 important tokens are selected for noise addition and each important token is perturbed with meaningful noise 5 times. In what follows, we first show some examples and their important tokens that are obtained by our interpretation method, then extract the vulnerability rules based on these important tokens. We also compare these rules with that of a commercial vulnerability detection tool, Checkmarx [1].

We use a true-positive example, a true-negative example, a false-positive example, and a false-negative example to illustrate the first 5 important tokens identified by our interpretation method as well as the trained decision tree-based regression model and a subgraph of the PDG. These examples are selected from the datasets of VulDeePecker and SySeVR. We limit the maximum height of the decision tree to 3 for a convenience of display.

Fig. 7 (a) shows a true-positive example which involves a integer overflow to buffer overflow vulnerability (CWE-680) [5] (we also use this example to show the process of extracting rules in Section 3.5). The vulnerability is caused by the following: The index `‘i’` of the array `‘intpointer’` ranges from 0 to `‘data’`. If the integer overflow occurs for `‘dataBytes’` in Line 8, the size of `‘intpointer’` may be an unexpected value. Consequently, the assignment of `‘intPointer[i]’` may be beyond the assigned size of `‘intPointer’` (Line 11). We observe that our interpretation method identifies the important tokens, such as the unreliable external input API call `‘recv’`, the loop `‘for’` statement used for writing, and the temporary variable `i` used for traversal, though it also identifies the unexpected operation `‘-’` and the API call `delete`. Fig. 7 (b) shows the trained decision tree. We observe that the `‘for’` loop statement and the unreliable external input API call `‘recv’` determine that the example is more likely to be vulnerable. Fig. 7 (c) shows a subgraph of the PDG of this example. The intersection of the external input data stream received by the `‘recv’` and the `‘for’` statement is the vulnerability trigger position (Line 11). Therefore, the vulnerability rule is: If the external input from ‘recv’ in ‘for’ loop (e.g., the ‘data’ in ‘for’ loop is data-dependent on parameter ‘inputBuffer’ in ‘recv’) is used, the example is vulnerable. This rule makes sense because the vulnerability is caused by unreliable external input without checking the boundary of the buffer. We also observe that even if the API call ‘recv’ or the ‘for’ statement is not used, this example is still likely predicted as vulnerable because the unreliable external input ‘data’ is not checked in the example. For the commercial vulnerability tool Checkmarx [1], this vulnerability is a false-negative. That is, Checkmarx cannot detect this vulnerability.

```
1   static int xc2028_set_config(struct dvb_frontend *fe, void *priv_cfg)
2   struct xc2028_data *dataCopy = fe->tuner_priv;
3   struct xc2028_data *priv = dataCopy;
4   struct xc2028_ctrl *p    = priv_cfg;
5   int              rc   = 0;
6   tuner_dbg("%s called\n", __func__);
7   mutex_lock(&priv->lock);
8   kfree(priv->ctrl.fname);
9   priv->ctrl.fname = NULL;
10  memcpy(&priv->ctrl, p, sizeof(priv->ctrl));
11  if (p->fname) {
12  priv->ctrl.fname = kstrdup(p->fname, GFP_KERNEL);
```

(a) A false-positive example



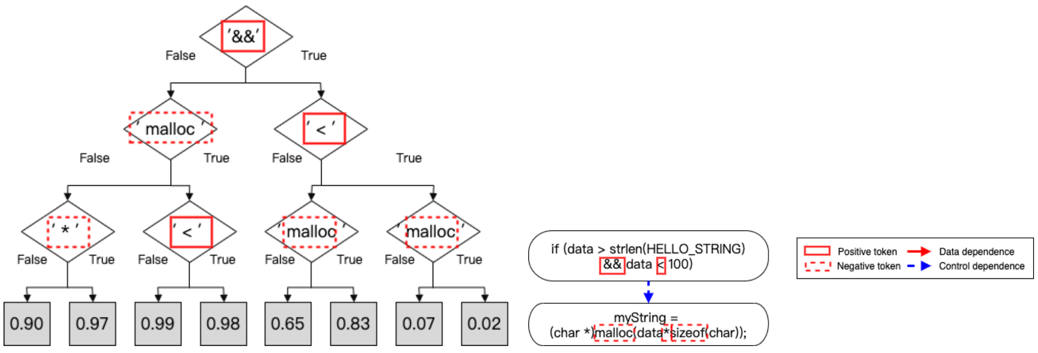
(b) Trained decision tree (c) Subgraph of program dependency graph

Fig. 9. A false-positive example, the trained decision tree, and a subgraph of the program dependency graph, where the first 5 important tokens identified by our interpretation method are highlighted (3 positive tokens are highlighted by solid box and 2 negative tokens are highlighted by dashed boxes. The vulnerability rule is: *If the type of the first parameter in function* ‘xc2028_set_config’ *is a user-defined* ‘struct’, *the example is vulnerable.*

Fig. 9(a) shows a false-positive example whose code is from open-source software OpenSSL. According to the results of explanation, we observe that the detector considers the example vulnerable mainly based on the function signatures (Line 1). Note that the detector does not consider the memory copy API ‘memcpy’ as dangerous (Line 10). The reason may be that the length of the copy is limited by the length of the copied buffer. Fig. 9(b) shows the trained decision tree. We observe that the function signature determines that the example is likely to be vulnerable. Fig. 9(c) shows the subgraph of the PDG of this example. We summarize the vulnerability rule as: if the type of the first parameter in function ‘xc2028_set_config’ is a user-defined ‘struct’, then the example is likely to be vulnerable. This rule does not make sense since the function signatures have nothing to do with vulnerabilities. Therefore, it may be an effective way to identify false-positive examples by determining the rules extracted from the decision tree does not make sense. For Checkmarx [1], this example is also a false-positive. The corresponding vulnerability rule defined by human experts is simply matching the API call name, regardless of data flow, i.e., ‘`memcpy`’ is a dangerous function.

```
1   size_t data;
2   data = 0;
3   goodB2GSource(data);
4   static void goodB2GSource(size_t &data)
5   fscanf(stdin, "%ud", &data);
5   char * myString;
6   if (data > strlen(HELLO_STRING) && data < 100)
7   myString = (char *)malloc(data*sizeof(char));
8   strcpy(myString, HELLO_STRING);
9   printLine(myString);
10  free(myString);
```

(a) A true-negative example



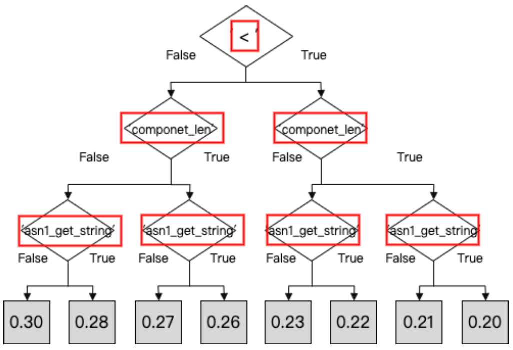
(b) Trained decision tree (c) Subgraph of program dependency graph

Fig. 10. The trained decision tree and subgraph of program dependency graph for a true-negative example: the first 5 important tokens identified by our interpretation method are highlighted, involving 2 positive tokens highlighted by solid line boxes and 3 negative tokens highlighted by dotted boxes. This example is not vulnerable because *the parameter of memory allocation API ‘*`malloc`*’ has been checked (using ‘*`<`*’) in an ‘*`if`*’ conditional statement with ‘*`&&`*’*.

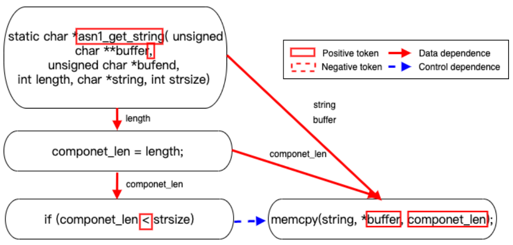
Fig. 10(a) shows a true-negative example which is a patched example of uncontrolled memory allocation vulnerability (CWE-789) [5]. According to the results of explanation, we observe that the detector considers the example is not vulnerable mainly based on the memory allocation ‘`malloc`’ statement and the conditional ‘`if`’ statement in Lines 6-7. Fig. 10(b) shows the trained decision tree and Fig. 10(c) shows a graph of the PDG of this example. We observe that this example is not vulnerable because the parameter of memory allocation API ‘`malloc`’ has been checked (using ‘`<`’) in an ‘`if`’ conditional statement with ‘`&&`’. For Checkmarx [1], the corresponding vulnerability rule defined by human experts is: if the memory size that memory allocation API ‘`malloc`’ uses is derived from the external input and has not been sanitized, it is vulnerable. This example is a false-positive for Checkmarx which does not recognize the complete data flow checking in the example.

```
1   static char *
2   asn1_get_string(
3   unsigned char **buffer,
4   unsigned char *bufend,
5   int         length,
6   char        *string,
7   int         strsize)
8   componet_len = length;
9   if (componet_len < strsize)
10  memcpy(string, *buffer, componet_len);
```

(a) A false-negative example



(b) Trained decision tree



static char *asn1_get_string(unsigned char **buffer, unsigned char *bufend, int length, char *string, int strsize)
length
componet_len = length;
componet_len
if (componet_len < strsize)
memcpy(string, *buffer, componet_len);
string
buffer
componet_len
Positive token
Negative token
Data dependence
Control dependence

(c) Subgraph of program dependency graph

Fig. 11. The trained decision tree and subgraph of program dependency graph for a false-negative example: the first 5 important tokens identified by our interpretation method are highlighted, involving 5 positive tokens highlighted by solid line boxes and no negative tokens. This example is not vulnerable because *the parameter* '`componet_len`' *in* '`memcpy`' *has been checked using* '`<`' *in the* '`if`' *statement.*

Fig. 11 (a) shows a false-negative example (CVE-2007-5849) which is a numeric errors vulnerability (CWE-189), in CUPS 1.2 through 1.3.4. It allows remote attackers to execute arbitrary code via a crafted SNMP response that triggers a stack-based buffer overflow [4]. Specifically, the vulnerability is caused by the following: the value of variable '`componet_len`' is not checked whether it is negative before it is passed to the parameter of '`memcpy`' (Line 10). According to the results of explanation, we observe that the detector considers the example is not vulnerable mainly based on the parameters ('`buffer`' and '`componet_len`') used for memory copy and corresponding data checking in Lines 9-10. Fig. 11(b) shows the trained decision tree and Fig. 11 (c) shows the subgraph of the PDG of this example. We observe that the detector classifies this example as not vulnerable because the parameter of memory copy '`componet_len`' has been checked using '`<`'. But the detector does not realize that the example fails to check whether the integer variable '`componet_len`' is a non-negative number. Therefore, the rule learned by the detector is not complete. It may be necessary to supplement more data similar to this example to help model learning. For Checkmarx [1], similar to Fig. 9, this example is a true-positive because Checkmarx simply matches the API name '`memcpy`' without checking the data flow.

Table. 6 gives brief explanations on the prediction results of another four examples. We observe that the rules corresponding to the TP and TN examples are reasonable, while the rules corresponding to the FP and FN examples are unreasonable. This means that potential false-negative examples and false-positive examples can be identified by analyzing whether the rules are convincing.

In summary, most important tokens of examples identified by our interpretation model may meet our expectations. However, there are also some unexpected behaviors that may due to the bias in the training data and as a result the model can not correctly understand the semantics of the program. For the vulnerability detection tool Checkmarx [1] which relies on human experts to define vulnerability rules, the vulnerability detection effectiveness is not good for our examples and many vulnerability rules are one-sided. By contrast, the deep learning-based vulnerability detector can detect the vulnerabilities from the semantic level. This leads to:

Insight 4. *The framework can extract vulnerability rules that can be understood and interpreted by a domain expert, and can help a domain expert identify some false-positive and false-negative examples when the extracted vulnerability rules do not make a security sense.*

4.3.4 User Study for Human-understandability. In order to evaluate the human-understandability of the rules derived from our interpretation method, we conduct a user study by defining the following 4 attributes:

- Conclusiveness: Whether or not a rule is conclusive in showing an example is vulnerable or non-vulnerable. For instance, the rule described in Figure 7 is conclusive on that the example is vulnerable.
- Conditionality: Whether or not a rule is conditioned on something that has a cybersecurity meaning.
- Association: Whether or not the objects mentioned in a rule correspond to some code elements in the example (e.g., variables, expressions, and statements).
- Inference: Whether or not a rule's conclusion can be inferred from on the conditions and/or the association attributes mentioned above (when applicable).

In principle, these attributes can be quantified in a continuous fashion (e.g., quantities in $[0, 1]$) but their exact measurements are challenging to define and obtain. For simplicity, we focus on their discrete definitions in $\{0, 1\}$, where "0" means that a rule is hard for a human evaluator to understand and "1" means that a rule is easy for a human evaluator to understand.

We randomly select 5 examples respectively for the 4 categories of examples (i.e., true-positive, false-positive, true-negative and false-negative), which leads to 20 examples (i.e., 20 rules) in total. We ask 8 computer science students to measure a given rule's human-understandability according to the preceding 4 attributes, without telling them which examples belong to which category. The human-understandability of a rule is averaged over 8×4 scores (as each student evaluator gives 4 scores on each rule). Intuitively, a higher score means a higher human-understandability. In order to draw insights into the impact of example category, we also "zoom into" the examples by computing their attribute-wise average scores in each category.

Figure 12(a) plots the average score of each example, which is grouped by category and sorted with each category. The average score is 0.88 for the true-positive examples, 0.98 for the true-negative examples, 0.46 for the false-positive examples, and 0.73 for the false-negative examples. We observe that the rules for interpreting true-positive examples and true-negative examples are more human-understandable than the rules for interpreting false-positive and false-negative examples. This is actually intuitive because true-positives and true-negatives are "natural" examples that are more "reasonable" to the *interpreter*; in contrast, false-positives and false-negatives are "unnatural" examples that are less "reasonable" to the *interpreter*.

Table 6. Explanation results for other examples, where positive tokens are highlighted with red solid boxes and negative tokens are highlighted with red dotted boxes.

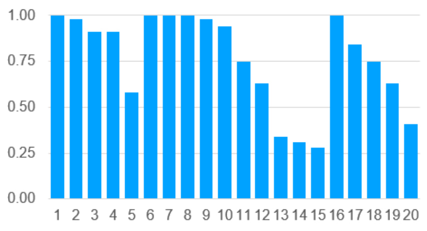
Type	Example	Rule

TP	static void badSource(wchar_t * &data)
	data[0] = L'\0';
	void (*funcPtr)(wchat_t *) = badSource;
	(*badSource)(data);
	void badSource(wchar_t * &data)
	size_t dataLen = wcslen(data);
	if (fgetws(data+dataLen, (int)(100-dataLen), pFile) == NULL)
	data[dataLen] = L' \0 ';
	for (; *data != L'\0'; data++)
	free(data);
		The detector predicts this example as vulnerable because the pointer is '++' in the 'for' loop and the pointer is not at the beginning of the buffer when the buffer is released (CWE-761). (*Reasonable rule*)

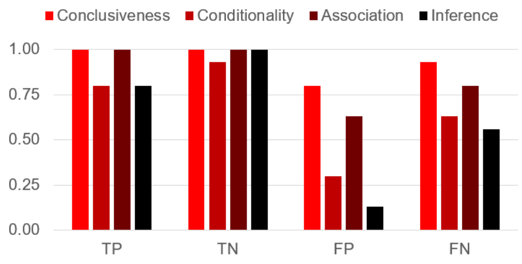
FP	void CWE78_OS_Command_Injection__char_connect_socket_system_44_bad()
	char * data ;
	char data_buf [100] = FULL_COMMAND ;
	data = data_buf;
	WSADATA wsaData ;
	int recvResult ;
	struct sockaddr_in service ;
	size_t dataLen = strlen (data) ;
	if (WSAStartup (MAKEWORD (2 , 2) , & wsaData) != NO_ERROR)
	connectSocket = socket (AF_INET , SOCK_STREAM , IPPROTO_TCP);
	if (connectSocket == INVALID_SOCKET)
	memset (& service , 0 , sizeof (service));
	service . sin_family = AF_INET;
	service . sin_addr . s_addr = inet_addr (IP_ADDRESS);
	service . sin_port = htons (TCP_PORT);
	if (connect (connectSocket , (struct sockaddr *) & service ,
	sizeof (service)) == SOCKET_ERROR)
	recvResult = recv (connectSocket , (char *) (data + dataLen) , sizeof (char) * (100 - dataLen - 1) , 0);
	if (recvResult == SOCKET_ERROR || recvResult == 0)
	data [dataLen + recvResult / sizeof (char)] = '\0';
	replace = strchr (data , '\r');
	if (replace)
	* replace = '\0';
	replace = strchr (data , '\n');
	if (replace)
	* replace = '\0;
	while (0)
	if (connectSocket != INVALID_SOCKET)
	CLOSE_SOCKET (connectSocket);
	funcPtr (data);
		The detector predicts this example as vulnerable because it ends with ');'. (*Unreasonable rule*).

TN	size_t data;
	data = 0;
	goodG2BSource(data);
	static void goodG2BSource(size_t &data)
	data = 20;
	char * myString;
	if (data > strlen(HELLO_STRING))
	myString = (char *)malloc (data* sizeof (char));
	strcpy (myString, HELLO_STRING);
	printLine(myString);
	free(myString);
		The detector predicts this example as not vulnerable because a constant '20' is used to restrict the size of system resource, which will not cause uncontrollable memory allocation problems. (*Reasonable rule*).

FN	static int Read(stream_t *s, void *p_read, unsigned int i_read)
	if(!p_read)
	return 0;
	if(Fill(s))
	return -1;
	stream_sys_t *p_sys = s->p_sys;
	int i_len = __MIN(i_read, p_sys->i_len - p_sys->i_pos);
	memcpy(p_read, p_sys->psz_xspf + p_sys->i_pos, i_len);
		The detector predicts this example as not vulnerable because it ends with ';' and the type of parameter is 'unsigned'. (*Unreasonable rule*).



(a) The score of each example grouped by category



(b) The attribute-wise average sores in each category

Fig. 12. (a) The score of the example that is averaged over the 8 evaluators and the 4 categories, grouped by the category it belongs to, and sorted within each category (examples 1-5 are true-positives, examples 6-10 are true-negatives, examples 11-15 are false-positives, and examples 16-20 are false-negatives). (b) The attribute-wise scores averaged over the examples in each category.

Figure 12(b) "zooms into" the attribute-wise average scores in each category. We observe that true-positive examples and true-negative examples have similar scores in terms of the 4 attributes. However, false-positive examples have very low inference score. By looking into the raw scores, we find that 3 (out of the 5) false-positive examples have inference score 0. This means that false-positives are especially difficult for their corresponding rules to make a cybersecurity sense, hinting that poor interpretability could be leveraged as an indicator of false alarm.

5 Discussion

The present study has several limitations, which need to be addressed in future studies. First, we cannot explain why a vulnerability detector thinks a token is more important than others. In order to answer this question, we will have to completely "open the box" of a deep learning model, which is a big challenge. Moreover, the selected tokens do not appear to have obvious cybersecurity interpretations that may help explain why they are more important. This may be intertwined with the fact that the important tokens identified by the heuristics we use may not be the most important factors. These explain why our work only represents a first step towards the ultimate goal of interpretability. Second, the generated rules are subjective because we only achieve semi-automated, rather than fully automated, interpretation. This means that the resulting human-understandable vulnerability rules are manually summarized based on decision trees and PDGs. It is an outstanding open problem to fully automate the extraction of human-understandable rules. This explains why our work only represents a first step towards the ultimate goal of automated interpretability. Third, the explanatory capability of our framework is demonstrated by only evaluating the fidelity of important tokens, but the fidelity of the rules. How to evaluate the fidelity of rules is an interesting future work. This explains why our work only represents a first step towards the ultimate goal of high-fidelity interpretability. Fourth, our framework is a local interpretation one because each interpretation result corresponds to an example. Therefore, the rules generated for one example is not applicable to interpreting other examples. It is an outstanding open problem to investigate global interpretation methods that can produce universal vulnerability rules. This explains why our work only represents a first step towards the ultimate goal of generic interpretability.

In terms of the validity of our evaluation, there are two important factors that need to be considered in future studies. First, we use VulDeePecker and SySeVR to demonstrate the feasibility of our interpretation method because their details are available to us. In principle, our interpretation method should be equally applied to other deep learning-based vulnerability detectors. Nevertheless, more experiments need to be conducted with other detectors. Second, we focus on using tokens as the unit when conducting code perturbation. On one hand, there can be other units (e.g., program statements and statement blocks) on which perturbations can be conducted. The granularity of units may affect the fidelity and readability of the interpretations. On the other hand, we perturb source code by deleting tokens and adding noise to tokens. There can be other methods for code perturbation, which is left as an interesting problem for future research. Therefore, more experiments need to be conducted to validate the interpretation method.

6 Related Work

It is worth mentioning that the problem of interpreting deep learning models is fundamentally different from the problem of feature selection. This is because the former aims to identify key feature values of an example (i.e., feature vector) to explain why the example is classified as such, meaning that tokens corresponding to different features may be selected for different examples. In contrast, feature selection algorithms (e.g., Info Gain [32] and Chi-Squared [18]) select features based on the contribution of a feature (i.e., the corresponding tokens of all examples), and therefore cannot be applied to explain the specific classification of an example. The purpose of model interpretation is different from the purpose of generating adversarial examples [48, 50], as interpretability is no concern in the later case. Nevertheless, there are related prior studies on model interpretation, which can be divided into three approaches: *hidden neuron analysis*, *model simulation*, and *local interpretation*.

Hidden Neuron Analysis. This approach aims to understand what concepts are learned by a neural network by transforming hidden layers into some human-understandable formats. This can be achieved by using multiple methods. The first method (e.g., [24, 42]) estimates feature importance by leveraging gradients and therefore suffers from the problem of vanishing gradients. The second method estimates feature importance by leveraging the attention mechanism [43, 44, 47]. AutoFocus [8] analyzes the important areas in the code in the context of code functionality classification and is not applicable to the models investigated in the present paper. The third method uses visualization to identify the most valuable neurons with respect to a class (e.g., using Global Max Pooling and ranked softmax weight in CNN models [52]). It is not clear how this method can be extended to accommodate other models (e.g., RNNs). The fourth method estimates feature importance by transforming the intermediate results of CNNs into the original input space, so that humans can observe the contours, color features, texture features, and local parts of the recognition across layers (e.g., DeConvNet [49]). It is not clear how this method can be extended to accommodate the models investigated in the present paper. The fifth method maps hidden layer variables to concept representations and quantifies the degree of matching (e.g., Network Dissection [7]). This method is not applicable to the present context because programs have no semantic hierarchies (e.g., color vs. material vs. object).

Model Simulation. This approach uses surrogate models with a higher explainability (e.g., linear regression, logistic regression, and decision tree [3, 6, 11, 12, 14, 17, 23, 51]) to approximate and interpret models with a lower explainability. For example, it is now known that piecewise linear neural network is mathematically equivalent to, and therefore can be explained by, a series of local linear classifiers [11]. However, this approach inevitably incurs distortion on the content learned by a complex model when using simple model to interpret it.

Local interpretation. This approach leverages the following insight: The decision boundary of a model may be complex, but the decision boundary with respect to a particular instance can be simple or even linear. The methods fall into this approach: *sensitivity analysis* and *local approximation*.

Sensitivity analysis aims to characterize how a model's output is affected by varying the model's input [16, 25, 29, 38]. The present study belongs to this method with the innovation that of considering the association between features. Local approximation uses a self-explanatory machine learning model to learn the importance of the features with respect to an example [19, 20, 22, 31, 35]. For example, LIME [36] perturbs a given example to obtain a set of examples and feed these examples to a linear regression model. This method cannot be applied to the setting of the present paper because it assumes that the local decision boundary is linear and features are independent, which would not hold for RNN because it aims to learn feature dependencies in sequential data. On the other hand, we have used experiments to compare our methods with LEMNA [20] and Kernel SHAP [31]. It is worth mentioning that when perturb an example, we accommodate features associations (rather than assuming they are independent of each other), which may affect the local decision boundary – a matter that has been encountered in a different context [21, 45, 46].

7 CONCLUSION

We have presented a high-fidelity model interpretation framework for explaining the predictions of deep learning-based source code vulnerability detectors. Systematic experiments show that the framework indeed has a higher fidelity than prior methods known as Kernel SHAP and LEMNA methods, especially when features are not independent of each other (which occurs often in the real world). In particular, the framework can produce some vulnerability rule that can be understood by domain experts for accepting a detector's outputs (i.e., true positives) or rejecting a detector's outputs (i.e., false-positives and false-negatives). We also discussed some limitations of the present study, which indicate interesting open problems for future research.

Acknowledgments

We thank the reviewers for their comments that guided us in revising and improving the paper; in particular, Section 4.3.4 was inspired by a reviewer's comment. The authors affiliated with Huazhong University of Science and Technology and Hebei University were supported in part by the National Key Research and Development Plan of China under Grant No. 2017YFB0802205, in part by the National Natural Science Foundation of China under grant No. U1936211 and No. 61802106, in part by the Natural Science Foundation of Hebei Province under grant No. F2020201016, and in part by the Shenzhen Fundamental Research Program under Grant No. JCYJ20170413114215614. Shouhuai Xu was supported in part by National Science Foundation Grants #1814825 and #1736209. The opinions expressed in the paper are those of the authors' and do not reflect the funding agencies' policies in any sense.

References

[1] Checkmarx 2020. *Checkmarx - Application Security Testing and Static Code Analysis.* Checkmarx, Israel. https://www.checkmarx.com/

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR* abs/1603.04467 (2016).

[3] Reza Abbasi-Asl and Bin Yu. 2017. Interpreting Convolutional Neural Networks Through Compression. *CoRR* abs/1711.02329 (2017).

[4] American Information Technology Laboratory 2020. *National Vulnerability Database.* American Information Technology Laboratory. https://nvd.nist.gov/.

[5] American Information Technology Laboratory 2020. *Software Assurance Reference Dataset.* American Information Technology Laboratory. https://samate.nist.gov/SRD/.

[6] Osbert Bastani, Carolyn Kim, and Hamsa Bastani. 2017. Interpreting Blackbox Models via Model Extraction. *CoRR* abs/1705.08504 (2017).

[7] David Bau, Bolei Zhou, Aditya Khosla, Aude Oliva, and Antonio Torralba. 2017. Network Dissection: Quantifying Interpretability of Deep Visual Representations. In *Proceedings of the 2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017.* IEEE Computer Society, 3319–3327. https://doi.org/10.1109/CVPR.2017.354

[8] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2019. AutoFocus: Interpreting Attention-Based Neural Networks by Code Perturbation. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019.* IEEE, 38–41. https://doi.org/10.1109/ASE.2019.00014

[9] Jianbo Chen, Le Song, Martin J. Wainwright, and Michael I. Jordan. 2018. Learning to Explain: An Information-Theoretic Perspective on Model Interpretation. In *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018.* 882–891. http://proceedings.mlr.press/v80/chen18j.html

[10] François Chollet et al. 2015. Keras. https://keras.io.

[11] Lingyang Chu, Xia Hu, Juhua Hu, Lanjun Wang, and Jian Pei. 2018. Exact and Consistent Interpretation for Piecewise Linear Neural Networks: A Closed Form Solution. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*, Yike Guo and Faisal Farooq (Eds.). ACM, 1244–1253. https://doi.org/10.1145/3219819.3220063

[12] Mark W. Craven and Jude W. Shavlik. 1995. Extracting Tree-Structured Representations of Trained Networks. In *Advances in Neural Information Processing Systems 8, NIPS, Denver, CO, USA, November 27-30, 1995*, David S. Touretzky, Michael Mozer, and Michael E. Hasselmo (Eds.). MIT Press, 24–30. http://papers.nips.cc/paper/1152-extracting-tree-structured-representations-of-trained-networks

[13] Hoa Khanh Dam, Trang Pham, Shien Wee Ng, Truyen Tran, John Grundy, Aditya Ghose, Taeksu Kim, and Chul-Joo Kim. 2018. A deep tree-based model for software defect prediction. *CoRR* abs/1802.00921 (2018). http://arxiv.org/abs/1802.00921

[14] Amit Dhurandhar, Karthikeyan Shanmugam, Ronny Luss, and Peder A. Olsen. 2018. Improving Simple Models with Confidence Profiles. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada.* 10317–10327. http://papers.nips.cc/paper/8231-improving-simple-models-with-confidence-profiles

[15] Xu Duan, Jingzheng Wu, Shouling Ji, Zhiqing Rui, Tianyue Luo, Mutian Yang, and Yanjun Wu. 2019. VulSniper: Focus Your Attention to Shoot Fine-grained Vulnerabilities. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence (IJCAI), Macao, China.* 4665–4671.

[16] Ruth C. Fong and Andrea Vedaldi. 2017. Interpretable Explanations of Black Boxes by Meaningful Perturbation. In *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017.* IEEE Computer Society, 3449–3457. https://doi.org/10.1109/ICCV.2017.371

[17] Nicholas Frosst and Geoffrey E. Hinton. 2017. Distilling a Neural Network into a Soft Decision Tree. In *Proceedings of the First International Workshop on Comprehensibility and Explanation in AI and ML 2017 co-located with 16th International Conference of the Italian Association for Artificial Intelligence (AI*IA 2017), Bari, Italy, November 16th and 17th, 2017.* http://ceur-ws.org/Vol-2071/CExAIIA_2017_paper_3.pdf

[18] Karl Pearson F.R.S. 1900. X. On the criterion that a given system of deviations from the probable in the case of a correlated system of variables is such that it can be reasonably supposed to have arisen from random sampling. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 50, 302 (1900), 157–175. https://doi.org/10.1080/14786440009463897 arXiv:https://doi.org/10.1080/14786440009463897

[19] Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Dino Pedreschi, Franco Turini, and Fosca Giannotti. 2018. Local Rule-Based Explanations of Black Box Decision Systems. *CoRR* abs/1805.10820 (2018).

[20] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. 2018. LEMNA: Explaining Deep Learning based Security Applications. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018.* 364–379. https://doi.org/10.1145/3243734.3243792

[21] Olivier Habimana, Yuhua Li, Ruixuan Li, Xiwu Gu, and Ge Yu. 2020. Sentiment analysis using deep learning approaches: an overview. *Sci. China Inf. Sci.* 63, 1 (2020), 111102. https://doi.org/10.1007/s11432-018-9941-6

[22] Mahdi Hajiaghayi and Ehsan Vahedi. 2018. Code Failure Prediction and Pattern Extraction using LSTM Networks. *CoRR* abs/1812.05237 (2018).

[23] Bo-Jian Hou and Zhi-Hua Zhou. 2018. Learning with Interpretable Structure from RNN. *CoRR* abs/1810.10708 (2018).

[24] Jiwei Li, Xinlei Chen, Eduard H. Hovy, and Dan Jurafsky. 2016. Visualizing and Understanding Neural Models in NLP. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, San Diego California, USA, June 12-17, 2016.* 681–691. https://doi.org/10.18653/v1/n16-1082

[25] Jiwei Li, Will Monroe, and Dan Jurafsky. 2016. Understanding Neural Networks through Representation Erasure. *CoRR* abs/1612.08220 (2016).

[26] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, Zhaoxuan Chen, Sujuan Wang, and Jialai Wang. 2018. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *CoRR* abs/1807.06756 (2018). arXiv:1807.06756 http://arxiv.org/abs/1807.06756

[27] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018.* http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_03A-2_Li_paper.pdf

[28] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, and Yang Xiang. 2017. POSTER: Vulnerability Discovery with Function Representation Learning from Unlabeled Projects. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017.* 2539–2541. https://doi.org/10.1145/3133956.3138840

[29] Lingqiao Liu and Lei Wang. 2012. What has my classifier learned? Visualizing the classification rules of bag-of-feature model by support region detection. In *2012 IEEE Conference on Computer Vision and Pattern Recognition, Providence, RI, USA, June 16-21, 2012.* IEEE Computer Society, 3586–3593. https://doi.org/10.1109/CVPR.2012.6248103

[30] Scott M. Lundberg, Gabriel G. Erion, and Su-In Lee. 2018. Consistent Individualized Feature Attribution for Tree Ensembles. *CoRR* abs/1802.03888 (2018). arXiv:1802.03888 http://arxiv.org/abs/1802.03888

[31] Scott M. Lundberg and Su-In Lee. 2017. A Unified Approach to Interpreting Model Predictions. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA.* 4765–4774. http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions

[32] J. Ross Quinlan. 1986. Induction of Decision Trees. *Mach. Learn.* 1, 1 (1986), 81–106. https://doi.org/10.1023/A:1022643204877

[33] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. 2019. Regularized Evolution for Image Classifier Architecture Search. In *Proceedings of the 33rd AAAI Conference on Artificial Intelligence, AAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019.* 4780–4789. https://doi.org/10.1609/aaai.v33i01.33014780

[34] D Raj Reddy et al. 1977. Speech understanding systems: A summary of results of the five-year research effort. *Department of Computer Science. Camegie-Mell University, Pittsburgh, PA* 17 (1977).

[35] Marco Túlio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. Nothing Else Matters: Model-Agnostic Explanations By Identifying Prediction Invariance. *CoRR* abs/1611.05817 (2016).

[36] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016.* 1135–1144.

[37] Marko Robnik-Sikonja and Marko Bohanec. 2018. Perturbation-Based Explanations of Prediction Models. In *Human and Machine Learning - Visible, Explainable, Trustworthy and Transparent.* 159–175. https://doi.org/10.1007/978-3-319-90403-0_9

[38] Marko Robnik-Sikonja and Igor Kononenko. 2008. Explaining Classifications For Individual Instances. *IEEE Trans. Knowl. Data Eng.* 20, 5 (2008), 589–600. https://doi.org/10.1109/TKDE.2007.190734

[39] Abhik Roychoudhury and Yingfei Xiong. 2019. Automated program repair: a step towards software automation. *Sci. China Inf. Sci.* 62, 10 (2019), 200103:1–200103:3. https://doi.org/10.1007/s11432-019-9947-6

[40] Rebecca L. Russell, Louis Y. Kim, Lei H. Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In *17th IEEE International Conference on Machine Learning and Applications, ICMLA 2018, Orlando, FL, USA, December 17-20, 2018.* 757–762. https://doi.org/10.1109/ICMLA.2018.00120

[41] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, and Adrian Bolton. 2017. Mastering the game of Go without human knowledge. *Nature* 550, 7676 (2017), 354–359. https://doi.org/10.1038/nature24270

[42] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. 2014. Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps. In *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Workshop Track Proceedings.* http://arxiv.org/abs/1312.6034

[43] Hendrik Strobelt, Sebastian Gehrmann, Michael Behrisch, Adam Perer, Hanspeter Pfister, and Alexander M. Rush. 2019. Seq2seq-Vis: A Visual Debugging Tool for Sequence-to-Sequence Models. *IEEE Trans. Vis. Comput. Graph.* 25, 1 (2019), 353–363.

[44] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron C. Courville, Ruslan Salakhutdinov, Richard S. Zemel, and Yoshua Bengio. 2015. Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015.* 2048–2057.

http://proceedings.mlr.press/v37/xuc15.html

[45] Li Xu, Zhenxin Zhan, Shouhuai Xu, and Keying Ye. 2013. Cross-layer detection of malicious websites. In *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy, CODASPY'13, San Antonio, TX, USA, February 18-20, 2013*. 141–152. https://doi.org/10.1145/2435349.2435366

[46] Li Xu, Zhenxin Zhan, Shouhuai Xu, and Keying Ye. 2014. An evasion and counter-evasion study in malicious websites detection. In *Proceedings of IEEE Conference on Communications and Network Security, CNS 2014, San Francisco, CA, USA, October 29-31, 2014*. 265–273. https://doi.org/10.1109/CNS.2014.6997494

[47] Zichao Yang, Diyi Yang, Chris Dyer, Xiaodong He, Alexander J. Smola, and Eduard H. Hovy. 2016. Hierarchical Attention Networks for Document Classification. In *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, San Diego California, USA, June 12-17, 2016*. 1480–1489. https://doi.org/10.18653/v1/n16-1174

[48] Noam Yefet, Uri Alon, and Eran Yahav. 2019. Adversarial Examples for Models of Code. *CoRR* abs/1910.07517 (2019). arXiv:1910.07517 http://arxiv.org/abs/1910.07517

[49] Matthew D. Zeiler and Rob Fergus. 2014. Visualizing and Understanding Convolutional Networks. In *Proceedings of the 13th European Conference on Computer Vision, ECCV 2014, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part I*. 818–833. https://doi.org/10.1007/978-3-319-10590-1_53

[50] Huangzhao Zhang, Zhuo Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. 2020. Generating Adversarial Examples for Holding Robustness of Source Code Processing Models. In *Proceedings of the 34th AAAI Conference on Artificial Intelligence, AAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 1169–1176. https://aaai.org/ojs/index.php/AAAI/article/view/5469

[51] Quanshi Zhang, Yu Yang, Ying Nian Wu, and Song-Chun Zhu. 2018. Interpreting CNNs via Decision Trees. *CoRR* abs/1802.00121 (2018).

[52] Bolei Zhou, Aditya Khosla, Àgata Lapedriza, Aude Oliva, and Antonio Torralba. 2016. Learning Deep Features for Discriminative Localization. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 2921–2929. https://doi.org/10.1109/CVPR.2016.319

[53] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Proceedings of Annual Conference on Neural Information Processing Systems 2019 (NeurIPS)*. 10197–10207.

[54] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. 2019. μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *IEEE Transactions on Dependable and Secure Computing* (2019). https://doi.org/10.1109/TDSC.2019.2942930