Information-theoretic Source Code Vulnerability Highlighting

Van Nguyen[1], Trung Le[1], Olivier De Vel[2], Paul Montague[2], John Grundy[1], Dinh Phung[1]
[1] Faculty of Information Technology, Monash University, Australia
[2] Defence Science and Technology Group, Australia

***Abstract*—Software vulnerabilities are a crucial and serious concern in the software industry and computer security. A variety of methods have been proposed to detect vulnerabilities in real-world software. Recent methods based on deep learning approaches for automatic feature extraction have improved software vulnerability identification compared with machine learning approaches based on hand-crafted feature extraction. However, these methods can usually only detect software vulnerabilities at a function or program level, which is much less informative because, out of hundreds (thousands) of code statements in a program or function, only a few core statements contribute to a software vulnerability. This requires us to find a way to detect software vulnerabilities at a fine-grained level. In this paper, we propose a novel method based on the concept of mutual information that can help us to detect and isolate software vulnerabilities at a fine-grained level (i.e., several statements that are highly relevant to a software vulnerability that include the core vulnerable statements) in both unsupervised and semi-supervised contexts. We conduct comprehensive experiments on real-world software projects to demonstrate that our proposed method can detect vulnerabilities at a fine-grained level by identifying several statements that mostly contribute to the vulnerability detection decision.**

I. Introduction

In the field of software security, software vulnerabilities (SVs) are specific potential flaws, glitches, weaknesses or oversights in parts of software. Attackers or vandals can leverage these vulnerabilities to carry out malicious actions, such as exposing or altering sensitive information, disrupting or destroying a system, or taking control of a program or computer system [1]. Owing to the rapid growth and dramatic diversity of software, a large amount of computer software potentially contains vulnerabilities, which can create severe threats to cyber-security, resulting in expenditure costs of about *USD 600 billion* globally each year [2]. These threats call for an urgent need of advanced approaches (i.e., automatic tools and methods) to efficiently and effectively deal with the large amount of vulnerable code with a minimal level of human intervention.

There are two typical approaches for software vulnerability detection (SVD) including methods based on either hand-crafted or automatic extraction of features. Most previous work in software vulnerability detection [3]–[9] has been developed based on hand-crafted features of data which are manually chosen by knowledgeable domain experts and may thus carry outdated experience, expertise and underlying biases [10]. To lessen the dependency on hand-crafted features, the use of automatically learned features for SVD has been recently studied, notably [11]–[13]. In particular, these works leverage deep learning models to automatically extract features, and have shown great advances over those based on hand-crafted features.

Despite showing promising performances, current deep learning-based methods are *only* able to detect software vulnerabilities at the *function* [12], [13] or *program* [11] levels. In real-world situations, programs or even functions are often very long and may consist of hundreds or thousands of lines of code. The source of most vulnerabilities arises from a significantly smaller scope, usually a few *core statements*. We thus want to be able to detect software vulnerabilities at a more fine-grained level, i.e., several code statements within functions or programs. This includes highlighting statements that are highly relevant to the corresponding vulnerability and associated code statements. In doing this, we can then significantly speed up the process of isolating and detecting software vulnerabilities, thereby reducing the time and cost involved.

In this paper, we propose a novel method that allows us to find and highlight code statements in *functions* or *programs* that are truly relevant to the presence of significant code vulnerabilities. Given vulnerable source code (i.e., functions or programs), we aim to highlight the *top-K* statements that are the most relevant to the vulnerable and non-vulnerable class labels. By referring to the vulnerable source code with a high probability, the highlighted statements contain the core statements that contribute to the overall code vulnerabilities. Moreover, our proposed method specifies and highlights the statements that *explain* the vulnerability intrinsic to the given source code. Our proposed method involves two key stages. In the first stage, we train a *reference* deep learning model, with the aim of approximating the true conditional distribution $p(y \mid F)$ (i.e., the true probabilistic labeling assignment mechanism) of label y (where $y = 1$ means a vulnerability and $y = 0$ means otherwise) w.r.t the source code F by $p_m(y \mid F)$ offered by the reference model. In the second stage, we learn another model that aims to explain the reference model by specifying the *top-K* statements in the given source code that mostly contribute to the vulnerability prediction decision for this model.

The idea is to select a subset of K statements that maximizes the mutual information to the corresponding label given

Acknowledgement: This research was supported under the Defence Science and Technology Group's Next Generation Technologies Program.

by the reference model. A similar information-theoretic metric has been employed in the L2X model [14] for instance-wise feature selection for the case of vectorial data. Although that model can be adopted to work for sequential data (e.g., source code), our proposed method is different from L2X in the following aspects: i) our formulation for mutual information takes into consideration the sequential nature of source code (in contrast to L2X) and ii) inspired from this formulation, we propose a novel architecture using a multi-Bernoulli distribution for selecting the *top-K* mostly relevant statements rather than employing a multinomial distribution as in L2X. The advantage of this approach is two-fold. First, this allows us to better control the random selection process. Second, it allows us to incorporate the information from ground truth in a semi-supervised context wherein we assume that a small portion of source code data might have annotations of core statements that cause a vulnerability. Our key contributions include:

- We propose a novel learn-to-explain model that is based on mutual information and takes into account the sequential nature of data to better evaluate mutual information. Using this theory, we propose a novel architecture based on multi-Bernoulli distribution for random subset of statements selection. Unlike the multinomial distribution used in L2X, our mechanism is more controllable and enables us to train the model in a semi-supervised context. In addition, our proposed model can be used to highlight the core statements that are a subset of the most relevant statements of vulnerable source code. It can also explain how the reference model works by identifying the most important statements that contribute to its prediction.
- We conduct experiments on the data sets collected by [12], that contain source code of vulnerable and non-vulnerable functions from two real-world software data sources and compare our proposed method to a state-of-the-art baseline L2X approach on these two data sets. We further investigate our proposed method in the semi-supervised context by comparing it to itself in an unsupervised context. We demonstrate that our proposed method can detect vulnerable code statements in functions much more effectively than L2X in unsupervised context and its semi-supervised variant can significantly boost the performance.

II. Motivating example

We give an example of a source code function obtained from the CWE-119 data set to demonstrate software vulnerability detection (SVD) at a fine-grained level, shown in Fig. 1. This function has a few core vulnerable code statements highlighted in red that are the main source of the vulnerability. The statement *"if (fgets (inputBuffer, CHAR_ARRAY_SIZE, stdin) != NULL)"* is a potential vulnerability because we read data from the console using *fgets()*. Likewise the two statements "*if (data >= 0)*" and "*buffer[data] = 1;*" cause another potential vulnerability because we attempt to write to an index of the array that exceeds the upper bound. Since the real-world source codes might contain hundreds of statements, we want

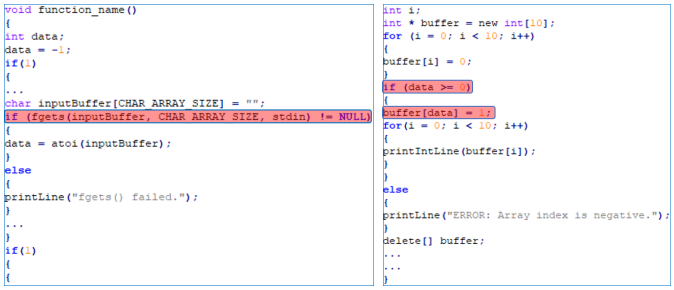

Fig. 1. An example of a source code function obtained from the CWE-119 data set. The left-hand and right-hand figures are the first and second parts of the function. For demonstration purpose, we choose a simple source code function, and some parts of the function are omitted for the brevity. The red lines specify the core vulnerable statements obtained from the ground truth.

to be able to highlight several statements in the function that are highly relevant to the presence of a vulnerability and contain the core vulnerable statements. In doing so, we can significantly speed up the process of isolating and detecting software vulnerabilities, and therefore reduce the cognitive load of the security analyst.

III. Information-theoretic code vulnerability highlighting

We begin with the problem statement of *Information-theoretic Code Vulnerability Highlighting* (ICVH), followed by the technical details of ICVH in the unsupervised and semi-supervised contexts.

A. The problem statement

Most of the publicly available data sets only have vulnerability labels (i.e., y) for the entire source codes (i.e., F) and have no information of code statements causing vulnerabilities. Our ICVH method only requires vulnerability labels at the source code level (i.e., Y obtained from the reference model) and is capable of pointing out the code statements highly relevant to these vulnerability labels. We hence call this setting as *unsupervised*, meaning that the training process does not require labels at the code statement level (i.e., ground-truth of vulnerable code statements causing vulnerabilities). In addition, in section semi-supervised context, we assume that a tiny portion of the source codes has labels at the code statement level. We hence name this setting as *semi-supervised*.

Consider a data set $D = \{(F_1, y_1), \ldots, (F_N, y_N)\}$ where $y_i \in \{0, 1\}$ (where 1: vulnerable code and 0: non-vulnerable code) and $F_i = [\boldsymbol{f}^i_1, \ldots, \boldsymbol{f}^i_{N_i}]$ is source code with a sequence of N_i statements (i.e., F_i is the i-th source code section in the data set D) while the lower-case f stands for a statement in the corresponding source code F (e.g., $\boldsymbol{f}^i_k$ is the k-th code statement in the source code F_i). Given a source code $F = [\boldsymbol{f}_1, \ldots, \boldsymbol{f}_L]$, we denote the subset $F_S = [\boldsymbol{f}_{i_1}, \ldots, \boldsymbol{f}_{i_K}] = [\boldsymbol{f}_j]_{j \in S}$ where $S = \{i_1, \ldots, i_K\} \subset \{1, \ldots, L\}$ ($i_1 < i_2 < \ldots < i_K$). In this work, we undertake vulnerability detection at a *fine-grained* level than at the function or program levels. In other words, we learn to emphasize the code blocks that are directly and highly relevant to the vulnerabilities. Specifically,

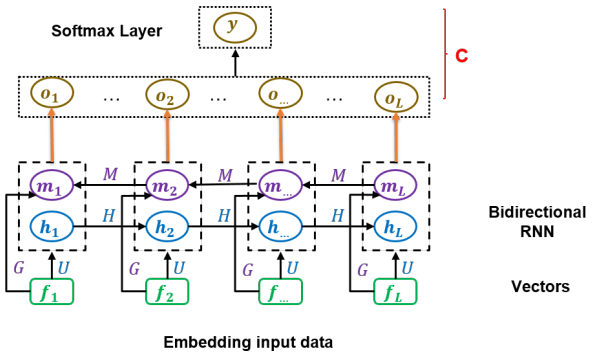

Fig. 2. The architecture of the reference model.

given a function F, our task is to select a subset F_S where $S = \{i_1, \ldots, i_K\} \subset \{1, \ldots, L\}$ in such a way that F_S is highly relevant to the presence of a vulnerability.

B. The reference model

The reference model aims to learn a model distribution $p_m(y \mid F)$ that can approximate the true distribution $p(y \mid F)$ where y is the label of the corresponding source code F and $y, F \sim p(y, F) = p(F)p(y \mid F)$. To obtain $p_m(y \mid F)$, we use a network architecture with a combination of a bidirectional recurrent neural network (Bi-RNN) to learn vector representations of source code and a deep feedforward neural network, which takes the outputs of the Bi-RNN as inputs, to model the distribution $p_m(y \mid F)$. The architecture of the reference model is depicted in Fig. 2 where $\boldsymbol{m}_i$, $\boldsymbol{h}_i$ and $\boldsymbol{o}_i = \text{concat}(\boldsymbol{m}_i, \boldsymbol{h}_i)$ with $i = 1, .., L$ are the hidden states and the output of the Bi-RNN respectively while C is the prediction layer, and M, H, U and G are model parameters. We note that without loss of generalization and to simplify the notion, we use $\boldsymbol{f}_i$ to represent both a symbolic code statement and its embedding vector that is fed to the networks. Data preprocessing and embedding is discussed in *data processing and embedding* section in the Supplementary material.

C. The explaining model: information-theoretic code vulnerability highlighting

1) Theoretical formulation with mutual information to capture the sequential nature of data: To select the most relevant subset F_S, we aim to maximize the mutual information: $\mathbb{I}(F_S, Y)$ where the random variable Y is characterized using $p_m(Y \mid F)$, which is previously trained using the whole training set $\mathcal{D}$. Mathematically, we aim to solve the following optimization problem:

$$\max \mathbb{E}_{p(F)}[\mathbb{E}_{p(S|F)}[\mathbb{I}(F_S, Y)]] \tag{1}$$

Eq. (1) means that given source code $F \sim p(F)$ (data distribution), we need to devise the random selection process characterized by $p(S \mid F)$ to select the subset F_S such that the mutual information of F_S and the label Y is maximized. The two main problems here are: i) how to design the random selection process $p(S \mid F)$ and ii) how to obtain $\mathbb{I}(F_S, Y)$.

We develop the following relevant theory to efficiently derive $\mathbb{I}(F_S, Y)$ and solve Eq. (1). We have,

Lemma 1.

$$\mathbb{I}(F_S, Y) = \sum_{k=1}^{K} \mathbb{E}_{\boldsymbol{f}_{i_{1:k-1}}}[\mathbb{I}(\boldsymbol{f}_{i_k}, Y \mid \boldsymbol{f}_{i_{1:k-1}})]$$

Proof. We have with noting that $\boldsymbol{f}_{i_{1:0}} = \emptyset$:

$$\begin{aligned}
&\mathbb{I}(F_S, Y) \\
&= \overline{\mathbb{E}}\left[[\log \frac{p_m(Y, F_S)}{p_m(Y)\, p_m(F_S)}]\right] \\
&= \overline{\mathbb{E}}\left[\log \frac{p_m(Y, \boldsymbol{f}_{i_K} \mid \boldsymbol{f}_{i_{1:K-1}}) \prod_{k=1}^{K-1} p_m(\boldsymbol{f}_{i_k} \mid \boldsymbol{f}_{i_{1:k-1}})}{p_m(Y) \prod_{k=1}^{K} p(\boldsymbol{f}_{i_k} \mid \boldsymbol{f}_{i_{1:k-1}})}\right] \\
&= \overline{\mathbb{E}}\left[\log \frac{p_m(Y, \boldsymbol{f}_{i_K} \mid \boldsymbol{f}_{i_{1:K-1}}) \prod_{k=1}^{K-1} p_m(Y, \boldsymbol{f}_{i_k} \mid \boldsymbol{f}_{i_{1:k-1}})}{p_m(Y) \prod_{k=1}^{K} p(\boldsymbol{f}_{i_k} \mid \boldsymbol{f}_{i_{1:k-1}}) \prod_{k=1}^{K-1} p_m(Y \mid \boldsymbol{f}_{i_{1:k}})}\right] \\
&= \overline{\mathbb{E}}\left[\sum_{k=1}^{K} \log \frac{p_m(Y, \boldsymbol{f}_{i_k} \mid \boldsymbol{f}_{i_{1:k-1}})}{\prod_{k=1}^{K}\left[p(\boldsymbol{f}_{i_k} \mid \boldsymbol{f}_{i_{1:k-1}}) p_m(Y \mid \boldsymbol{f}_{i_{1:k-1}})\right]}\right] \\
&= \sum_{k=1}^{K} \overline{\mathbb{E}}\left[\log \frac{p_m(Y, \boldsymbol{f}_{i_k} \mid \boldsymbol{f}_{i_{1:k-1}})}{\prod_{k=1}^{K}\left[p(\boldsymbol{f}_{i_k} \mid \boldsymbol{f}_{i_{1:k-1}}) p_m(Y \mid \boldsymbol{f}_{i_{1:k-1}})\right]}\right]
\end{aligned}$$

where $\overline{\mathbb{E}} = \mathbb{E}_{P_m(Y, F_S)}$

$$\begin{aligned}
&\mathbb{I}(F_S, Y) \\
&= \sum_{k=1}^{K} \mathbb{E}_{p_m(Y, \boldsymbol{f}_{i_{1:k}})}\left[\log \frac{p_m(Y, \boldsymbol{f}_{i_k} \mid \boldsymbol{f}_{i_{1:k-1}})}{p_m(Y \mid \boldsymbol{f}_{i_{1:k-1}}) p(\boldsymbol{f}_{i_k} \mid \boldsymbol{f}_{i_{1:k-1}})}\right] \\
&= \sum_{k=1}^{K} \mathbb{E}_{\boldsymbol{f}_{i_{1:k-1}}}\left[\mathbb{I}(\boldsymbol{f}_{i_k}, Y \mid \boldsymbol{f}_{i_{1:k-1}})\right]
\end{aligned}$$

□

The following lemma tackles $\mathbb{E}_{\boldsymbol{f}_{i_{1:k-1}}}[\mathbb{I}(\boldsymbol{f}_{i_k}, Y \mid \boldsymbol{f}_{i_{1:k-1}})]$ and this term can be further derived as follows.

Lemma 2. We have

$$\begin{aligned}
&\mathbb{E}_{\boldsymbol{f}_{i_{1:k-1}}}[\mathbb{I}(\boldsymbol{f}_{i_k}, Y \mid \boldsymbol{f}_{i_{1:k-1}})] \\
&\quad \approx \mathbb{E}_{\boldsymbol{f}_{i_{1:k}}}[\mathbb{E}_{p_m(Y|\boldsymbol{f}_{i_{1:k}})}[\log p_m(Y \mid \boldsymbol{f}_{i_{1:k}})]] + \text{const}
\end{aligned}$$

Proof.

$$\begin{aligned}
&\mathbb{E}_{\boldsymbol{f}_{i_{1:k-1}}}\left[\mathbb{I}\left(\boldsymbol{f}_{i_k}, Y \mid \boldsymbol{f}_{i_{1:k-1}}\right)\right] \\
&= \mathbb{E}_{\boldsymbol{f}_{i_{1:k-1}}}\left[\widetilde{\mathbb{E}}\left[\log \frac{p_m(Y, \boldsymbol{f}_{i_k} \mid \boldsymbol{f}_{i_{1:k-1}})}{p_m(Y|\boldsymbol{f}_{i_{1:k-1}}) p(\boldsymbol{f}_{i_k} \mid \boldsymbol{f}_{i_{1:k-1}})}\right]\right] \\
&\approx \mathbb{E}_{\boldsymbol{f}_{i_{1:k-1}}}\left[\widetilde{\mathbb{E}}\left[\log \frac{p_m(Y, \boldsymbol{f}_{i_k} \mid \boldsymbol{f}_{i_{1:k-1}})}{p_m(Y \mid F) p(\boldsymbol{f}_{i_k} \mid \boldsymbol{f}_{i_{1:k-1}})}\right]\right] \\
&= \mathbb{E}_{\boldsymbol{f}_{i_{1:k-1}}}\left[\widetilde{\mathbb{E}}\left[\log \frac{p_m(Y \mid \boldsymbol{f}_{i_{1:k}}) p(\boldsymbol{f}_{i_k} \mid \boldsymbol{f}_{i_{1:k-1}})}{p_m(Y \mid F) p(\boldsymbol{f}_{i_k} \mid \boldsymbol{f}_{i_{1:k-1}})}\right]\right] \\
&= \mathbb{E}_{\boldsymbol{f}_{i_{1:k-1}}}\left[\widetilde{\mathbb{E}}\left[\log \frac{p_m(Y \mid \boldsymbol{f}_{i_{1:k}})}{p_m(Y|F)}\right]\right] \\
&= \mathbb{E}_{\boldsymbol{f}_{i_{1:k}}}\left[\mathbb{E}_{p_m(Y|\boldsymbol{f}_{i_{1:k}})}\left[\log p_m(Y \mid \boldsymbol{f}_{i_{1:k}})\right]\right] + \text{const}
\end{aligned}$$

where $\widetilde{\mathbb{E}} = \mathbb{E}_{P_m\left(Y, \boldsymbol{f}_{i_k} | \boldsymbol{f}_{i_{1:k-1}}\right)}$ □

Noting that we approximate $p_m(Y \mid \boldsymbol{f}_{i_{1:k-1}})$ by $p_m(Y|F)$. The next lemma gives a lower bound to $\mathbb{E}_{\boldsymbol{f}_{i_{1:k}}}[\mathbb{E}_{p_m(Y|\boldsymbol{f}_{i_{1:k}})}[\log\, p_m(Y \mid \boldsymbol{f}_{i_{1:k}})]]$.

Lemma 3. We can obtain a lower bound to $\mathbb{E}_{\boldsymbol{f}_{i_{1:k}}}[\mathbb{E}_{p_m(Y|\boldsymbol{f}_{i_{1:k}})}[\log\, p_m(Y \mid \boldsymbol{f}_{i_{1:k}})]$ by

$$\mathbb{E}_{\boldsymbol{f}_{i_{1:k}}}[\mathbb{E}_{p_m(Y|\boldsymbol{f}_{i_{1:k}})}[\log\, Q(Y \mid \boldsymbol{f}_{i_{1:k}})]]$$

for every $Q\,(Y \mid F)$.

Proof.

$$\begin{aligned}
&\mathbb{E}_{\boldsymbol{f}_{i_{1:k}}}\left[\mathbb{E}_{p_m(Y|\boldsymbol{f}_{i_{1:k}})}\left[\log\, p_m(Y \mid \boldsymbol{f}_{i_{1:k}})\right]\right] \\
&\quad = \mathbb{E}_{\boldsymbol{f}_{i_{1:k}}}\left[\mathbb{E}_{p_m(Y|\boldsymbol{f}_{i_{1:k}})}\left[\log\, Q(Y \mid \boldsymbol{f}_{i_{1:k}})\right]\right] \\
&\qquad + \mathbb{E}_{\boldsymbol{f}_{i_{1:k}}}\left[D_{KL}(p_m(Y \mid \boldsymbol{f}_{i_{1:k}}) \| Q(Y \mid \boldsymbol{f}_{i_{1:k}}))\right]
\end{aligned}$$

□

Combining Lemmas 1, 2 and 3, the optimization problem in Eq. (1) can be now rewritten:

$$\max\, \mathbb{E}_{p(F)}[\mathbb{E}_{p(S|F)}[\sum_{k=1}^{K} \mathbb{E}_{\boldsymbol{f}_{i_{1:k}}}[\mathbb{E}_{p_m(Y|\boldsymbol{f}_{i_{1:k}})}[Q_k]]]] \quad (2)$$

where $Q_k = \log\, Q(Y \mid \boldsymbol{f}_{i_{1:k}})$.

2) Network design in the unsupervised context: To design the random selection process for selecting $S = \{i_1, ..., i_K\}$ $(i_1 < i_2 < ... < i_K)$ to form $p(S \mid F)$ and formulate $Q(Y \mid \boldsymbol{f}_{i_{1:k}})$ for $k = 1, \ldots, K$, we employ a Bi-RNN with sequence length L. As shown in Fig. 3, for each statement $\boldsymbol{f}_k$ or its embedding vector, which is also denoted by $\boldsymbol{f}_k$, we use a Bernoulli random variable Z_k with $\mathbb{P}(Z_k = 1) = \mu_k$ (i.e., we apply the sigmoid activation over μ_k to convert it to a probability) to specify if $\boldsymbol{f}_k$ is selected or not (i.e., $Z_k = 1$ means $\boldsymbol{f}_k$ is selected). We compute $Z_k \boldsymbol{f}_k$ for all k and then feed them to another Bi-RNN where we make a prediction of the label at the hidden states with $Z_k = 1$ to mimic $Q(Y \mid \boldsymbol{f}_{i_{1:k}})$. In addition, we have approximated the sub-sequence of statements $[\boldsymbol{f}_{i_1}, ..., \boldsymbol{f}_{i_k}]$ by the sequence $[Z_1\boldsymbol{f}_1, ..., Z_L\boldsymbol{f}_L]$ in which the *inactive* (*not* selected) statements are substituted by the vector $\mathbf{0}$.

To render the above random selection process continuous and differentiable for training, we employ the Concrete (Gumbel-softmax) distribution [15], [16] to undertake relaxation on the Bernoulli random variable Z_k. In particular, we sample V_k from the Concrete distribution as: $[V_k, 1 - V_k] \sim \text{Concrete}(\mu_k, 1 - \mu_k)$.

We now denote $V \odot F$ as $[V_k \boldsymbol{f}_k]_{k=1,...,L}$ and the sequence $V \odot F$ is injected into the second Bi-RNN. We denote S_K as the set of indices with the top K values for V_k. The optimization problem in Eq. (2) can be rewritten as follows:

$$\max\, \mathbb{E}_{p(F)}[\mathbb{E}_{p(S|F)}[\sum_{k \in S_K}[\sum_{y=0}^{1} p_m(Y = y \mid V \odot F) q_k]]] \quad (3)$$

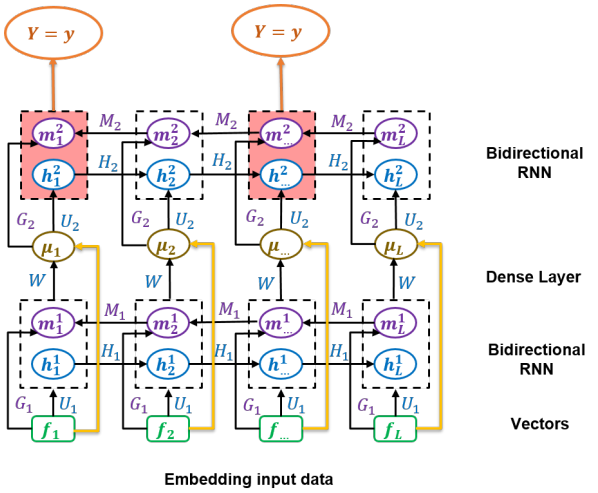

Fig. 3. The architecture of our Information-theoretic Code Vulnerability Highlighting (ICVH) network. The pink cells refer to those code statements in S_K and we formulate $p(Y = y \mid \boldsymbol{h}_k^2, \boldsymbol{m}_k^2)$ using the softmax function.

where $q_k = \log\, p(Y = y \mid \boldsymbol{h}_k^2, \boldsymbol{m}_k^2)$ relates to the log-likelihood of the second Bi-RNN. The architecture of our proposed Information-theoretic Code Vulnerability Highlighting applied to code vulnerability identification is depicted in Fig. 3 where $\boldsymbol{m}_i^1$, $\boldsymbol{h}_i^1$, $\boldsymbol{m}_i^2$ and $\boldsymbol{h}_i^2$ with $i = 1, .., L$ are the hidden states of two Bi-RNNs while M_j, H_j, U_j, G_j with $j = 1, 2$ and W are model parameters. Our ICVH model can be applied to both source code and binary code vulnerability identification.

In the testing phase, we choose the code statements whose indices lie in S_K with top K values for μ_k. We can interpret μ_k as the probability to select the k-th code statement in our subset.

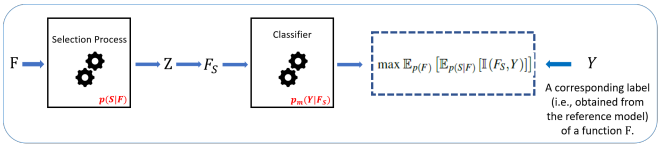

Fig. 4. The training phase of our ICVH model.

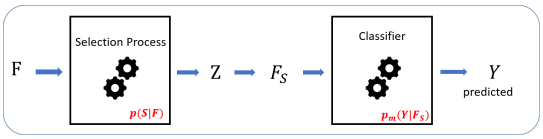

Fig. 5. The testing phase of our ICVH model using the trained model obtained from the training phase.

For our explaining ICVH model, we aim to obtain high performances not only on approaching close to (or have a good explanation of) the reference model (i.e., a high F1-score for the label prediction Y obtained from the reference model), but also on the selecting and highlighting process of vulnerable code statements in vulnerable functions (i.e., the VCP and VCA measures which are both mentioned in the Experiment section). The working processes of our ICVH model in the

training (i.e., in the unsupervised context) and testing phases are visualized in Fig. 4 and Fig. 5 respectively.

3) Network design in the semi-supervised context: It is very convenient to incorporate the annotations of the core vulnerable statements in the ground truth to our network design. Specifically, let $F_c = [\boldsymbol{f}_{i_1}, ..., \boldsymbol{f}_{i_m}]$ be the core vulnerable statements for a given source code. We maximize the probabilities of *selecting* the core statements and *not selecting* other statements in the first Bi-RNN as follows:

$$\max \sum_{k \in I_c} \log \mu_k + \sum_{k \notin I_c} \log(1 - \mu_k),$$

where $I_c = [i_1, ..., i_m]$. We then add the above objective function to the main objective function in Eq. (3) with the trade-off parameter $\lambda > 0$.

IV. Experiment

First, we compare our Information-theoretic Code Vulnerability Highlighting (ICVH) method with L2X introduced in [14] and the random selection method (RSM) where we randomly choose statements from functions in order to compare with ground truths of vulnerable code statements in the *unsupervised* context. Second, we investigate ICVH in the *semi-supervised* context in which there is a small portion of data having ground truth of core vulnerable code statements. Finally, we inspect the explanatory capability of ICVH by identifying example misclassifications by the reference model and then analysing the reason for these (details in the Supplementary material). Data preprocessing and model configuration are mentioned in the Supplementary material.

From machine learning and data mining perspectives, it seems that the existing methods in interpretable machine learning [14], [17], [18] with adoption are ready to apply. Unfortunately, besides [14], none of others can be adopted to be applicable to the specific context of statement-grained vulnerability detection.

To the best of our knowledge, there is one deep learning-based method, VulDeeLocator [19] posted on ArXiv, for fine-grained level vulnerability detection. We do not compare with VulDeeLocator because: i) it cannot work directly with source code (i.e., it requires to compile source codes to Lower Level Virtual Machine intermediate code), and ii) it requires information relevant to vulnerable statements for extracting tokens from program code according to a given set of vulnerability syntax characteristics, hence it cannot be operated in the unsupervised setting.

A. Experimental setup

1) Experimental data sets: We used the real-world data sets collected by [12] which contain the source code of vulnerable functions (vul-funcs) and non-vulnerable functions (non-vul-funcs) obtained from two real-world software data sets, containing buffer error vulnerabilities (CWE-119 has 5,582 vul-funcs and 5,099 non-vul-funcs) and resource management error vulnerabilities (CWE-399 has 1,010 vul-funcs and 1,313 non-vul-funcs). For both CWE-119 and CWE-399, we remove functions that are identical. The minimum, mean, and maximum length of functions in CWE-399 and CWE-119 are (4; 51; 177) and (4; 21; 164) respectively. For the 1,010 vulnerable functions of CWE399 and 5,582 vulnerable functions of CWE119, the percentage of vulnerable statements and non-vulnerable statements is 5.50% and 8.13% respectively. These percentages between vulnerable and non-vulnerable statements demonstrate that our proposed VCP and VCA measures are reasonable.

2) Labeling core vulnerable statements for evaluation: The CWE-399 and CWE-119 data sets have only vulnerable and non-vulnerable labels for their source codes. Our aim in this work is to detect the statements responsible for causing the vulnerability. Although our proposed method *does not need* the information of vulnerable statements *at all* in *the training phase*, this information is *necessary* in *evaluating its performance*. To obtain this information regarding the location of vulnerable statements in source code for the CWE-399 and CWE-119 data sets, we further processed these data sets. To obtain the ground truth of vulnerable code statements, we used the description of vulnerability information (i.e., the comments and annotations) in the original source code as well as the differences between the vulnerable versions and the fixed versions (i.e., non-vulnerable versions) of the source code.

3) Data processing and embedding: We preprocess the data sets before injecting them into the deep networks. First, we standardize the source code by: removing comments and non-ASCII characters, mapping user-defined variables to symbolic names (e.g., "*var1*", "*var2*") and user-defined functions to symbolic names (e.g., "*func1*", "*func2*"), and replacing strings with a generic <*str*> token. Second, we embed statements in source code into vectors. For instance, in the following statement (C programming language) *"if(func3(func4(2,2),&var2)!=var11)*": to embed this code statement, we tokenize it to a sequence of tokens (e.g., *if,(,func3,(,func4,(,2,2,),&,var2,),!=,var11,)),* construct the frequency vector of the statement, and multiply this frequency vector by the statement embedding matrix. The statement embedding matrix represents the learnable variables in our model.

4) Model configuration: We implemented ICVH and L2X in Python using Tensorflow [20], an open-source software library for Machine Intelligence developed by the Google Brain Team. We ran our experiments on a server with an Intel Xeon Processor E5-1660 which had 8 cores at 3.0 GHz and 128 GB of RAM. The length of each function is padded or cut to 100 code statements. For the reference model (i.e., the learning model), we used a bidirectional recurrent neural network (Bi-RNN) using LSTM cells, where the size of the hidden states is in $\{128, 256\}$, combined with a deep feedforward neural network having two hidden layers with the size of each hidden layer in $\{100, 200, 300\}$. For L2X, we used the structure with parameters as mentioned in [14] and for each data set, we used 10 epochs as suggested in [14] for the training process. For our ICVH method, regarding the first and second Bi-RNN, we used LSTM cells where the size of hidden states is in $\{128, 256\}$. The deep feedforward neural networks consisted of two hidden layers with the size of each

hidden layer in $\{100, 200, 300\}$. The trade-off parameter λ is in $\{10^{-1}, 10^{-2}\}$.

We employed the Adam optimizer [21] with an initial learning rate in $\{0.001, 0.003\}$, while the mini-batch size is 100 and the temperature τ for the Gumbel-softmax distribution is in $\{0.1, 0.5\}$, for both L2X and ICVH. For the reference (learning) model and explaining model (L2X and our ICVH method), we split the data of each data set into three random partitions. The first partition contains 80% for training, the second partition contains 10% for validation and the last partition contains 10% for testing. We additionally apply gradient clipping regularization to prevent over-fitting when training the model.

5) Measures and evaluation: To evaluate the performance of the proposed method and baselines in detecting the core vulnerable code statements, we proposed two measures: *vulnerability coverage proportion* (VCP) and *vulnerability coverage accuracy* (VCA).

The VCP aims to measure the proportion of correctly detected vulnerable statements over all vulnerable statements in a given data set. The VCP hence is mathematically defined as $\frac{\#detectedVCS}{\#allVCS}$ where $\#detectedVCS$ is the number of vulnerable code statements detected correctly and $\#allCVS$ is the number of all vulnerable code statements in a data set.

The VCA is considered more strictly, because it measures the ratio of the successfully detected functions over all functions in a data set. In addition, a function is considered *successfully detected* by a method if this method can detect successfully all vulnerable statements in this function. Mathematically, the VCA can be expressed as $\frac{\#detectedVFunc}{\#allVFunc}$ where $\#detectedVFunc$ is the number of successfully detected functions and $\#allVFunc$ is the number of functions in a data set.

In addition to VCP and VCA measures, we also reported the label (i.e., Y) classification F1-score on CWE-399 and CWE-119 data sets for our proposed method and baselines.

B. Experimental results

1) Learning process (the reference model): We aim to learn a model distribution $p_m(y \mid F)$ that can approximate the true distribution $p(y \mid F)$ where y is the label corresponding to the source code F. To obtain $p_m(y|F)$, we use the network architecture as depicted in Fig. 2. We measure the F1-score of the reference model (learning model) on CWE-119 and CWE-399 real-world data sets. Using this architecture *we obtained a high predictive performance for learning the approximate distribution* $p_m(y|F)$. In particular, the learning model obtained 99.25% and 94.29% F1-score for CWE-399 and CWE-119 respectively.

In the explaining process described in the next section, we aim to explain the reference model by specifying the most important code statements in each function F that have the most significant role for the reference model to make its decision about the corresponding label y.

2) Explaining code vulnerability highlighting with selected code statements in the unsupervised context: We compared the performance of our ICVH method with L2X [14] and RSM in the *unsupervised* context for explaining the reference model and highlighting the vulnerable code statements. We wanted to find out the top K statements that mostly influence the decision of the vulnerability of each function. The number of selected code statements for each function used in each method is fixed equal to 10 (i.e., $K = 10$). When comparing L2X and ICVH in the explainable or interpretable model, we not only aim to obtain a high F1-score for a good explanation, but also aim to measure how the selected and highlighted statements cover the core vulnerable statements.

The experimental results in Table I show that *our proposed method (ICVH) achieved a higher performance for both VCP and VCA measures, and F1-score compared with L2X on the CWE-399 and CWE-119 data sets*. In particular, for CWE-119, our proposed method (ICVH) achieved 89.13% for VCP and 86.27% for VCA while L2X achieved 83.21% and 77.74% for VCP and VCA respectively.

The higher F1-score that was achieved by ICVH shows that it can approximate (or achieve better explainability of) the reference model compared with L2X. The higher VCP and VCA measures show that our method can detect vulnerable code statements in vulnerable functions much more accurately and effectively compared with L2X.

TABLE I
PERFORMANCE RESULTS ON THE TESTING SET OF CWE-399 AND CWE-119 FOR RSM, L2X AND OUR ICVH METHODS (BEST PERFORMANCE AMONG METHODS FOR EACH DATA SET IN **BOLD**).

Data sets	K	Methods	VCP	VCA	F1-score
CWE-399	10	RSM	36.36%	30.87%	NA
		L2X	80.41%	71.00%	99.10%
		ICVH (ours)	**86.82%**	**80.46%**	**99.40%**
CWE-119	10	RSM	40.37%	33.28%	NA
		L2X	83.21%	77.74%	97.30%
		ICVH (ours)	**89.13%**	**86.27%**	**99.23%**

3) Explaining code vulnerability highlighting in the semi-supervised context with the variation of K: We investigated the performance of ICVH for two different contexts, including the unsupervised (ICVH) and semi-supervised (S2-ICVH) contexts for explaining the reference model and highlighting the vulnerable statements. In the semi-supervised context, we assume that there is a small portion of the training set (i.e., 5% or 10%) having ground truth of vulnerable code statements. We investigated the performance of ICVH from both unsupervised and semi-supervised contexts with some different values of K (i.e., $K = 5, 10$ code statements that are highly relevant to the presence of a vulnerability).

The experimental results in Table II show that *by using a small portion of data having ground truth (i.e., 5% or 10%) of vulnerable code statements, the model performance is significantly increased*. For example, for CWE-399, in the case of $K = 10$, the model performance in the unsupervised context (ICVH) achieved 86.82% and 80.46% for VCP and VCA respectively while the model performance in the semi-supervised context for S2-ICVH-5 (5% of data in the training process having ground truth of vulnerable code statements) and S2-ICVH-10 (10% of data in the training process having ground truth of vulnerable code statements) obtained (91.72%

TABLE II
PERFORMANCE RESULTS ON THE TESTING SET OF CWE-399 AND CWE-119 FOR OUR PROPOSED METHOD IN THE UNSUPERVISED CONTEXT (ICVH) AND SEMI-SUPERVISED CONTEXT (S2-ICVH) (BEST PERFORMANCE AMONG METHODS FOR EACH VALUE OF K IN **BOLD**).

Data sets	K	Methods	VCP	VCA	F1-score
CWE-399	5	ICVH	69.46%	54.65%	99.21%
		S2-ICVH-5	89.05%	85.42%	**100%**
		S2-ICVH-10	**90.67%**	**88.57%**	99.76%
	10	ICVH	86.82%	80.46%	99.40%
		S2-ICVH-5	91.72%	88.54%	**100%**
		S2-ICVH-10	**95.11%**	**92.86%**	99.76%
CWE-119	5	ICVH	67.53%	58.72%	99.39%
		S2-ICVH-5	90.53%	86.84%	**99.52%**
		S2-ICVH-10	**94.24%**	**91.73%**	99.51%
	10	ICVH	89.13%	86.27%	99.23%
		S2-ICVH-5	95.10%	92.85%	**99.51%**
		S2-ICVH-10	**98.63%**	**98.03%**	99.50%

for VCP and 88.54% for VCA) and (95.11% for VCP and 92.86% for VCA) respectively.

The experimental results in Table II show that *the more selected code statements we have, the higher performance in two main measures including VCP and VCA we obtain*. For instance, to data set CWE-119, in the case with $K = 5$, ICVH obtained 67.53% for VCP and 58.72% for VCA while with $K = 10$, ICVH obtained 89.13% for VCP and 86.27% for VCA respectively.

The high values of F1-score (over 99% for all cases mentioned in Table II) show that our proposed methods can approach very close to (or have a good explanation of) the learning model while the high values of VCP and VCA show that our proposed methods can effectively and efficiently detect vulnerable code statements in vulnerable functions.

4) Visualization of detected and highlighted code statements: Here we illustrate how we can visualize the highlighted code statements in vulnerable functions, in order to demonstrate the ability of our method to detect and highlight core vulnerable code statements in vulnerable functions to aid security auditors and code developers. We set $K = 5$ for the function in Fig. 6 and $K = 10$ for the functions in Fig. 7. In these figures, the colored lines (i.e., the green and red lines) highlight the detected code statements obtained when using our ICVH in the unsupervised learning context. In addition, each red line specifies the core vulnerable statement obtained from the ground truth, and these lines are detected by our method.

For example, in Fig. 6, the corresponding function has two core vulnerable statements including "*memset (var1 , str , 100 - 1) ;*" and "*memmove (var3 , var1 , strlen (var1) * sizeof (char)) ;*", which lead to a vulnerability, because in this case we initialize *var1* as a large buffer that is larger than the small buffer used in the sink (i.e., *var1* is larger than *var3)*. Our ICVH method with $K = 5$ can detect these core vulnerable statements that make the corresponding function vulnerable. In Fig. 7, the function has some core vulnerable code statements including "*if (fgets (var2 , var3 , stdin) != NULL)*", which is a potential vulnerability because we read data from the console using *fgets()*, and "*if (var1 >= 0); var7 [var1] = 1*" which are also a potential vulnerability in the case we attempt to write to an index of the array that is

True label: vulnerable and predicted label: vulnerable

```
void func1 ( )
{
char * var1 ;
char var2 [ 100 ] ;
var1 = var2 ;
if ( 5 == 5 )
{
memset ( var1 , str , 100 - 1 ) ;
var1 [ 100 - 1 ] = str ;
}
{
char var3 [ 50 ] = str ;
memmove ( var3 , var1 , strlen ( var1 ) * sizeof ( char ) ) ;
var3 [ 50 - 1 ] = str ;
func2 ( var1 ) ;
}
}
```

Fig. 6. The predicted label from the model and the true label are shown in the first row. The source code function and selected code statements highlighted relevant to vulnerabilities are shown with $K = 5$. The colored lines (i.e., the green and red lines) highlight the detected code statements while red lines specify the core vulnerable statements obtained from the ground truth, and these lines are detected by our method.

True label: vulnerable and predicted label: vulnerable

```
void func1 ( )
{
int var1 ;
var1 = - 1 ;
if ( 1 )
{
...
char var2 [ var3 ] = str ;
if ( fgets ( var2 , var3 , stdin ) != NULL )
{
var1 = atoi ( var2 ) ;
}
else
{
func2 ( str ) ;
}
...
}
if ( 1 )
{
{
```

```
int var6 ;
int * var7 = new int [ 10 ] ;
for ( var6 = 0 ; var6 < 10 ; var6 ++ )
{
var7 [ var6 ] = 0 ;
}
if ( var1 >= 0 )
{
var7 [ var1 ] = 1 ;
for ( var6 = 0 ; var6 < 10 ; var6 ++ )
{
func3 ( var7 [ var6 ] ) ;
}
}
else
{
func2 ( str ) ;
}
delete [ ] var7 ;
...
...
}
```

Fig. 7. The predicted label from the model and the true label are shown in the first row. The source code function and selected code statements highlighted relevant to vulnerabilities are shown with $K = 10$. The left-hand and right-hand figures are the first and second parts of the function, respectively. For demonstration purpose, there are some parts of the function omitted for the brevity. The colored lines (i.e., the green and red lines) highlight the detected code statements while red lines specify the core vulnerable statements obtained from the ground truth, and these lines are detected by our method.

above the upper bound. Our ICVH method with $K = 10$ can detect all of these core potential vulnerable code statements that make the corresponding function vulnerable.

Interestingly, we can use the *vulnerability relevance probability* μ_k associated with each statement to visualize a heat map over the source code as shown in Fig. 8. This is intuitive and informative as it shows which statements or blocks of statements are highly relevant to the vulnerabilities.

5) Investigation of misclassification of the non-vulnerable functions: In this section, we investigate the case when some non-vulnerable functions are predicted as vulnerable functions as depicted in Fig. 9. These functions appear as non-vulnerable in the ground truth. However, the reference model predicted them as vulnerable. Using $K = 5$, for the left-hand function shown in Fig. 9, the green selected code statements "*for (var4 = 0 ; var4 < var5 ; var4 ++)*" and "*var3 [var4] = var2 [var4] ;*" can in some cases (e.g., if *var2* is larger than *var3*) lead to a potential vulnerability. For the right-hand function shown in Fig. 9, the green selected code statement "*wmemset (var1 , str , 50 - 1) ;*" will be a vulnerable code statement if we change "*50 - 1*" into "*100 - 1*", because in this case we would initialize the source buffer as a buffer that

Fig. 8. An example heat map that represents the vulnerable relevance probabilities over the given source code function.

is larger than the buffer used in the sink (i.e., "*wcsncat (var4 , var1 , wcslen (var1)) ;*"). These are some typical examples for the case when non-vulnerable functions are predicted as vulnerable functions. The main reasons are likely due to: i) the key contributed code statements for marking the label of a function can be a potential vulnerability in some specific cases (e.g., depending on behaviour in the calling functions), or ii) the key contributed code statements for marking the label of a function can be a potential vulnerability if we have a minor code change effected in them.

True label: non-vulnerable and predicted label: vulnerable

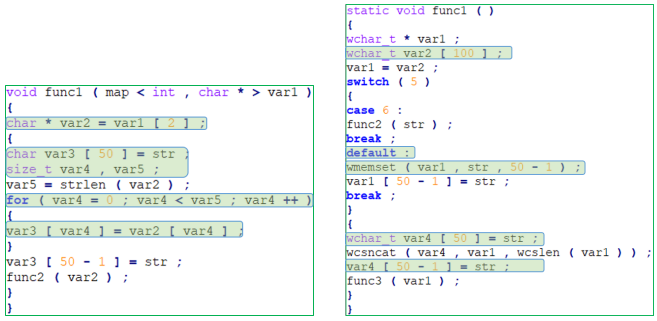

Fig. 9. The predicted label from the model and the true label are shown in the first row. The two source code functions and selected code statements in each function are shown with $K = 5$. The colored lines (i.e., the green lines) highlight the detected code statements by our method.

V. Conclusions

We have proposed a new method to detect software vulnerabilities at a fine-grained level than the function or program levels in both unsupervised and semi-supervised contexts. Our proposed method aims to maximize the mutual information between selected code statements and the response variable of a function or program offered by the reference model trained in the learning phase. By maximizing this mutual information, the selected statements are expected to strongly correlate with the existence of a vulnerability, hence potentially containing the core vulnerable statements. In addition, our proposed model is able to play the role of an explanatory model that explains which statements in a given source mostly contribute to the prediction of the reference model. Our experimental results on real-world data sets showed that by using our proposed method we can detect software vulnerabilities at a fine-grained level effectively and accurately.

References

[1] M. Dowd, J. McDonald, and J. Schuh, *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley Professional, 2006.

[2] McAfee and CSIS, "There's nowhere to hide from the economics of cybercrime," 2017.

[3] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07, 2007, pp. 529–540.

[4] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Transactions on Software Engineering*, vol. 37, no. 6, pp. 772–787, 2011.

[5] F. Yamaguchi, F. Lindner, and K. Rieck, "Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning," in *Proceedings of the 5th USENIX conference on Offensive technologies*, 2011, pp. 13–23.

[6] M. Almorsy, J. Grundy, and A. Ibrahim, "Supporting automated vulnerability analysis using formalized vulnerability signatures," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012, 2012, pp. 100–109.

[7] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "Vulpecker: An automated vulnerability detection system based on code similarity analysis," in *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ser. ACSAC '16, 2016, pp. 201–213.

[8] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward large-scale vulnerability discovery using machine learning," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '16, 2016, pp. 85–96.

[9] S. Kim, S. Woo, H. Lee, and H. Oh, "VUDDY: A scalable approach for vulnerable code clone discovery," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2017, pp. 595–614.

[10] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: A large scale experiment on data vs. domain vs. process," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09, 2009, pp. 91–100.

[11] H. K. Dam, T. Tran, T. Pham, N. S. Wee, J. Grundy, and A. Ghose, "Automatic feature learning for vulnerability prediction," *CoRR*, vol. abs/1708.02368, 2017.

[12] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," *CoRR*, vol. abs/1801.01681, 2018.

[13] G. Lin, J. Zhang, W. Luo, L. Pan, Y. Xiang, O. De Vel, and P. Montague, "Cross-project transfer representation learning for vulnerable function discovery," in *IEEE Transactions on Industrial Informatics*, 2018.

[14] J. Chen, L. Song, M. J. Wainwright, and M. I. Jordan, "Learning to explain: An information-theoretic perspective on model interpretation," *CoRR*, vol. abs/1802.07814, 2018.

[15] E. Jang, S. Gu, and B. Poole, "Categorical reparameterization with gumbel-softmax," *arXiv preprint arXiv:1611.01144*, 2016.

[16] C. J. Maddison, A. Mnih, and Y. W. Teh, "The concrete distribution: A continuous relaxation of discrete random variables," *arXiv preprint arXiv:1611.00712*, 2016.

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

[18] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Advances in Neural Information Processing Systems*, 2017, pp. 4765–4774.

[19] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin, "Vuldeelocator: A deep learning-based fine-grained vulnerability detector," *arXiv preprint arXiv:2001.02350*, 2020.

[20] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

[21] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.