FINE-GRAINED SOFTWARE VULNERABILITY DETECTION VIA INFORMATION THEORY AND CONTRASTIVE LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

Software vulnerabilities existing in a program or function of computer systems have been becoming a serious and crucial concern. In a program or function consisting of hundreds or thousands of source code statements, there are only few statements causing the corresponding vulnerabilities. Vulnerability labeling on a function or program level is usually done by experts with the assistance of machine learning tools; however, it will be much more costly and time-consuming to do that on a statement level. In this paper, to tackle this challenging problem, we propose a novel end-to-end deep learning-based approach to obtain the vulnerability-relevant code statements of a specific function. Inspired from previous approaches, we first leverage the mutual information theory for learning a set of latent variables that can represent the relevance of the source code statements to the corresponding function's vulnerability. We then propose a novel clustered spatial contrastive learning in order to further improve the representation learning and robust the selection process of vulnerability-relevant code statements. The experimental results on real-world datasets show the superiority of our proposed method over other state-of-the-art baselines.

1 INTRODUCTION

Due to the variety of computer software as well as the diversity in its development processes, a large amount of computer software faces software vulnerabilities (SVs), specific potential flaws, glitches, weaknesses or oversights, that can be exploited by hackers or vandals resulting in severe and serious economic damage (Dowd et al., 2006). Potential vulnerabilities presenting in software development and deployment processes can create severe threats to cyber-security, leading to expenditure costs of total over *USD 1 trillion*, a more than 50 percent increase from 2018 (McAfee & CSIS, 2020). Although much effort has been devoted and many solutions have been proposed for software vulnerability detection (SVD), the number of SVs and the severity of the threat imposed by them have gradually increased and caused considerable damage to individuals and companies (Ghaffarian & Shahriari, 2017). These threats create an urgent need for automatic tools and methods to effectively deal with a large amount of vulnerable code with a minimal level of human intervention.

There have been many methods proposed for SVD based on either machine learning or deep learning approaches. Most previous work in software vulnerability detection (Shin et al., 2011; Yamaguchi et al., 2011; Almorsy et al., 2012; Li et al., 2016; Grieco et al., 2016; Kim et al., 2017) belongs to the former involving the knowledge of domain experts that can be outdated and biased (Zimmermann et al., 2009). To mitigate this problem, deep learning approaches have been used to conduct SVD and have shown great advances, notably in (Li et al., 2018a; Lin et al., 2018; Dam et al., 2018; Li et al., 2018b; Duan et al., 2019; Cheng et al., 2019; Zhuang et al., 2020), over machine learning approaches based on hand-crafted features.

Despite achieving the promising performance, current state-of-the-art deep learning-based methods (e.g., (Li et al., 2018a; Dam et al., 2018; Li et al., 2018b)) are only able to detect software vulnerabilities at a function or program level. However, in real-world situations, programs or functions can consist of hundreds or thousands of source code statements where only few of them, usually few core statements, cause the corresponding vulnerabilities. Fig. 1 shows an example of a simple vulnerable

source code function. Among these many lines of code statements, there are only two statements highlighted in red actually lead to the function's vulnerability. The core statements underpinning a vulnerability are even much sparser in source code of real-world applications.

Recently, instead of detecting whether a source code section (i.e., a function or program. Hereafter, we use "section or function or program" to denote a collection of code statements) is vulnerable. There have been some approaches (Li et al., 2020; Nguyen et al., 2021) proposed to deal with the fine-grained software vulnerability detection problem. This includes highlighting statements that are highly relevant to the corresponding function's 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.

```
void func()
{
    char * data;
    data = NULL;
    if(staticReturnsTrue())
    {
        {
            char * dataBuffer = new char[100];
            memset(dataBuffer, 'A', 100-1);
            dataBuffer[100-1] = '\0';
            data = dataBuffer - 8;
        }
    }
    {
        size_t i;
        char source[100];
        memset(source, 'C', 100-1);
        source[100-1] = '\0';
        for (i = 0; i < 100; i++)
        {
            data[i] = source[i];
        }
        data[100-1] = '\0';
        printLine(data);
    }
}
```

Figure 1: An example of a buffer error vulnerability source code function. For demonstration purpose, we choose a short function. In this function, the statement *"data = dataBuffer - 8;"* is a vulnerability because we set data pointer to before the allocated memory buffer, and the statement "*data[i] = source[i];*" is a potential flaw due to possibly copying data to memory before the destination buffer.

In particular, Li et al. (2020) proposed a new deep learning framework named VulDeeLocator for fine-grained vulnerability detection. However, VulDeeLocator cannot work directly with the original source code. It requires to compile the source code to Lower Level Virtual Machine intermediate code and cannot be used if a function cannot be compiled. Furthermore, besides using the source code section F (e.g., a C/C++ function or program) and its corresponding label Y (i.e., $Y \in \{0, 1\}$ where *1*: vulnerable and *0*: non-vulnerable), VulDeeLocator requires the information relevant to vulnerable code statements for extracting tokens from a function according to a given set of vulnerability syntax characteristics, hence it cannot be operated in the unsupervised setting (i.e., where the training process does not require any information relevant to the labels at the code statement level). Nguyen et al. (2021) proposed a novel method, named Information-theoretic code vulnerability highlighting (ICVH), based on the concept of mutual information to detect software vulnerabilities at a fine-grained level. The authors use ICVH as an explaining model aiming to explain the reference model (i.e., the learning model approximating the true conditional distribution $p(Y \mid F)$) by specifying the K code statements in the given source code function, which mostly contribute to the vulnerability prediction decision of the reference model. ICVH can be implemented in the unsupervised setting and works directly on the original source code.

In this paper, we propose a novel end-to-end deep learning-based approach for fine-grained software vulnerability detection that allows us to find and highlight code statements, in *functions* or *programs*, truly relevant to the presence of significant source code vulnerabilities. In particular, inspired from Chen et al. (2018); Nguyen et al. (2021), we first leverage the mutual information theory in learning a set of independent Bernoulli latent variables that can represent the relevance of the source code statements to the corresponding function's vulnerability. We name this one as a random selection process ε picking out a subset $\tilde{F} = \varepsilon(F) \subset F$. Moreover, we observe that for vulnerable source code sections (i.e., functions or programs), there are several core statements causing their vulnerability. If we group these core statements together, we have vulnerability patterns shared across vulnerable source code sections. For example, the buffer overflow error can have two vulnerability patterns named "buffer copy without checking size of input" or "the improper validation of array index". Additionally, those hidden vulnerability patterns can be embedded into real-world source code sections at different spatial locations to form realistic vulnerable source code sections.

More specifically, given a set of vulnerable source code sections, we need to devise an elegant mechanism to guide the selection process ε to select and highlight hidden vulnerability patterns. This is evident a challenging task since vulnerability patterns are hidden and can be embedded into real vulnerable source code sections at different spatial locations. To this end, for characterizing a vulnerable source code section F, we consider F^{top} including K statements in F with the top

K highest selection probabilities. We further observe that a vulnerability type consists of several vulnerability patterns and vulnerable source code sections originated from the same vulnerability pattern possess very similar the top K statements F^{top} which tend to form well-separated clusters. Based on this observation, we propose clustered spatial contrastive learning term inspired from supervised contrastive learning (Khosla et al., 2020), which encourages F^{top} in the same cluster to have similar representations.

Finally, similar to Nguyen et al. (2021), we conduct experiments on two real-world source-code datasets CWE-119 and CWE-399. The extensive experiments on these two real-world datasets show the advancements of our proposed method in selecting and highlighting the core vulnerable statements more accurately indicated by its performance superiority over baselines by a wide margin.

2 Fine-grained Software Vulnerability Detection

We denote a source code section (e.g., a C/C++ function or program) as $F = [\boldsymbol{f}_1, \ldots, \boldsymbol{f}_L]$, which consists of L lines of code statements $\boldsymbol{f}_1, \ldots, \boldsymbol{f}_L$ (L can a be a large number, e.g., hundreds or thousands). In practice, each code statement is represented as a vector, which is extracted by some embedding methods. As those embedding methods are not the focus of this paper, we leave these details to the experiment section. We assume that F's vulnerability $Y \in \{0, 1\}$ (where *1*: vulnerable and *0*: non-vulnerable) is observed (labeled by experts). As previously discussed, there is usually a small subset with K code statements that actually lead to F being vulnerable, denoted as $\tilde{F} = [\boldsymbol{f}_{i_1}, \ldots, \boldsymbol{f}_{i_K}] = [\boldsymbol{f}_j]_{j \in S}$ where $S = \{i_1, \ldots, i_K\} \subset \{1, \ldots, L\}$ ($i_1 < i_2 < ... < i_K$). To select the vulnerability-relevant statements $\tilde{F}$ for each specific source code section F, we apply to use a learnable random selection process ε, i.e., $\tilde{F} = \varepsilon(F)$, whose training principle and construction are presented in Section 2.1. We then propose a novel clustered spatial contrastive learning, which can model important properties for the relationship of the source code sections, presented in Section 2.2 to further improve the representation learning and robust the selection process of $\tilde{F}$.

It is worth noting that most of available datasets only have the vulnerability label (i.e., Y) at the source code level (i.e., to know whether a function F is vulnerable) and they do not contain the information of the source code statements causing vulnerabilities. In the training process, our proposed method only require the vulnerability label at the source code level (i.e., Y) and are capable of pointing out the vulnerability-relevant statements. *In the context of fine-grained software vulnerability detection*, this setting is considered as the *unsupervised* one mentioned in Nguyen et al. (2021), meaning that the training process does not require labels at the code statement level (i.e., the ground truth of vulnerable code statements causing vulnerabilities). The ground truth of vulnerable code statements causing vulnerabilities is only used in the evaluation process.

2.1 Training principle and construction of the selection process

Training principle Inspired from Chen et al. (2018); Nguyen et al. (2021), we apply to use mutual information (i.e., a measure of the dependence between two random variables and it captures how much knowledge of one random variable reduces the uncertainty about the other) to the fine-grained software vulnerability detection problem. We implement mutual information as a training principle for obtaining the most vulnerability-relevant statements $\tilde{F}$ of each specific source code section F. If we view $\tilde{F}$ and Y as random variables, the selection process ε can be learned by maximizing the mutual information between $\tilde{F}$ and Y, formulated as follows:

$$\max_{\varepsilon} \mathbb{I}(\tilde{F}, Y). \tag{1}$$

We expand Eq. (1) further as the Kullback-Leibler divergence of the product of marginal distributions of $\tilde{F}$ and Y from their joint distribution:

$$\begin{aligned} \mathbb{I}(\tilde{F}, Y) &= \int p(\tilde{F}, Y) \log \frac{p(\tilde{F}, Y)}{p(\tilde{F})p(Y)} d\tilde{F} dY \\ &\geq \int p(Y, \tilde{F}) \log \frac{q(Y \mid \tilde{F})}{p(Y)} dY\, d\tilde{F} \end{aligned} \tag{2}$$

Noting that in the above derivation, we use a variational distribution $q(Y|\tilde{F})$ to approximate the posterior $p(Y \mid \tilde{F})$, hence deriving a variational lower bound of $\mathbb{I}(\tilde{F}, Y)$ for which the equality holds if $q(Y \mid \tilde{F}) = p(Y \mid \tilde{F})$. This can be further expanded as:

$$\begin{aligned}\mathbb{I}(\tilde{F}, Y) \geq & \int p(Y, \tilde{F}, F) \log \frac{q(Y \mid \tilde{F})}{p(Y)} dY d\tilde{F} dF \\ = & \mathbb{E}_F \mathbb{E}_{\tilde{F}|F}[\sum_Y p(Y|F) \log q(Y|\tilde{F})] + \text{const} \end{aligned} \tag{3}$$

We note that $\tilde{F}|F := \tilde{F} \sim p(\cdot|F) := \varepsilon(F)$ is the same representation of the random selection process and $p(Y|F)$ as mentioned before could the ground-truth conditional distribution of the F's label on all of its features or probabilistic prediction from a reference model.

To model the conditional variational distribution $q(Y|\tilde{F})$, we introduce a classifier implemented with a neural network, which takes $\tilde{F}$ as input and outputs its corresponding label. Our objective is to learn the selection process as well as the classifier to maximize the mutual information:

$$\max_{\varepsilon, q}(\mathbb{E}_F \mathbb{E}_{\tilde{F}|F}[\sum_Y p(Y|F) \log q(Y|\tilde{F})]). \tag{4}$$

The mutual information facilitates a joint training process for the classifier and the selection process. The classifier is learned to identify a subset of the features leading to a data sample's label while the selection process is designed to select the best subset according to the feedback of the classifier.

Selection process For the selection process, we apply to use the multivariate Bernoulli distribution. Without loss of generality, we assume that all source code has the length of L statements (i.e., filling with **0** (s) for the shorter source code and truncating the longer source code). For each function, we use a binary latent vector $Z \in \{0, 1\}^L$ where each element z_i indicates whether f_i is related to the vulnerability of F. As Z depends on F, we denote $Z(F)$. With Z, we further construct $\tilde{F} = \varepsilon(F)$ by $\tilde{F} = Z(F) \odot F$, where $\odot$ represents the element-wise product.

To construct Z, we model $Z \sim \prod_{i=1}^{L} \text{Bernoulli}(p_i)$, which yields $\tilde{f}_i = f_i$ with probability p_i and $\tilde{f}_i = \mathbf{0}$ with probability $1 - p_i$. We then construct $p_i = \omega_i(F; \alpha)$ where ω is a neural network parameterized by α, taking F as input, and outputting a probability. We then employ another neural network $g(\tilde{F}; \beta)$ to define $q(Y|\tilde{F})$. Recall that $\tilde{F} = [z_i f_i]_{i=1}^{L}$ where $Z_i \sim \text{Bernoulli}(p_i)$ with $p_i = \omega_i(F; \alpha)$. We apply the Gumbel softmax distribution to do continuous relaxation that allows us to jointly train $\omega(\cdot; \alpha)$ and $g(\cdot; \beta)$. Let $a_i, b_i \overset{iid}{\sim} \text{Gumbel}(0, 1)$ and we sample $Z_i(F; \alpha) \sim \text{Concrete}(\log \omega_i(F; \alpha), \log(1 - \omega_i(F; \alpha)))$, we have:

$$Z_i(F; \alpha) = \frac{\exp\{\frac{\log \omega_i(F;\alpha) + a_i}{\tau}\}}{\exp\{\frac{\log \omega_i(F;\alpha) + a_i}{\tau}\} + \exp\{\frac{\log(1 - \omega_i(F;\alpha)) + b_i}{\tau}\}}$$

2.2 CLUSTERED SPATIAL CONTRASTIVE LEARNING

Motivation For each vulnerable function F, we observe that there are few statements causing vulnerability. If we group those core statements together, they form *vulnerability patterns*. The left-hand figure in Figure 2 shows a vulnerability pattern named the *improper validation of array index* flaw pattern for the *buffer overflow error* in which the software performs operations on a memory buffer, but it can read from or write to a memory location that is outside of the intended boundary of the buffer. More specifically, this aims to get a value from an array (i.e., *int *array*) via specific *index* and save this value into a variable (i.e., *value*). However, this only verifies that the given array *index* is less than the maximum length of the array using the statement "*if(index<len)*" but does not check for the minimum value, hence allowing a negative value to be accepted as the input array index, which will result in an out of bounds read and may allow access to sensitive memory.

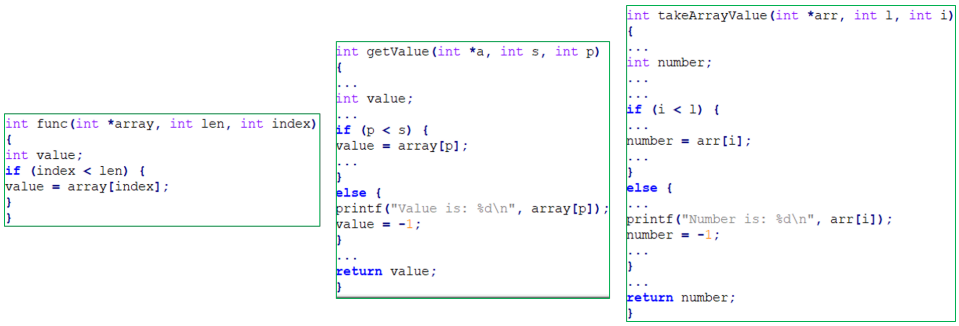

Figure 2: An example of the *improper validation of array index* flaw pattern (i.e., the left-hand figure) with two real-world source code functions (i.e., the middle and right-hand figures) containing this pattern. In each function, there are some parts omitted for the brevity.

As shown in Figure 2, this vulnerability pattern is embedded into real-world functions *getValue* and *takeArrayValue* in which the core statements in the vulnerability pattern are placed into different spatial locations under different variable names. We wish to guide the selection process so that the vulnerable source code sections originated from the same vulnerability pattern have similar selected and highlighted statements which commonly specify this vulnerability pattern. This is challenging because the common vulnerability pattern is embedded into those source code sections at different spatial locations. To address this issue, given a source code section F, we define F^{top} as a subset of F including its K statements with the top K selection probability $p_i = \omega_i(F; \alpha)$ and employ F^{top} to characterize the predicted vulnerability pattern of F. It is worth noting that the statements F^{top} preserves the order in F.

To enforce two vulnerable source code sections originated from the same vulnerability pattern having the same F^{top}, an initial naive solution is to employ the supervised contrastive learning (Khosla et al., 2020) to reach the following objective function based on the contrastive learning principle as follows:

$$\mathcal{L}_{scl} = \sum_{i \in I} 1_{y_i=1} \frac{-1}{|P(i)|} \sum_{p \in P(i)} \log \frac{\exp(\text{sim}(F_i^{top}, F_p^{top})/\tau)}{\sum_{a \in A(i)} \exp(\text{sim}(F_i^{top}, F_a^{top})/\tau)} \tag{5}$$

where $I \equiv \{1...m\}$ is a set of indices of input data in a specific mini-batch, sim is the cosine similarity, $\tau > 0$ is a scalar temperature parameter, $A(i) \equiv I \setminus \{i\}$, $P(i) \equiv \{p \in A(i) : y_p = 1\}$ is the set of indices of vulnerable source code sections with the label 1 (1 : *vulnerable* and 0: *non-vulnerable*) in the mini-batch except i, $|P(i)|$ is its cardinality, and 1_A represents the indicator function.

It can be observed that although the objective function in (5) encourages vulnerable source code sections sharing the same selected and highlighted vulnerability pattern, it seems to overdo this by forcing all vulnerable source code sections sharing the same vulnerability pattern. It what follows, we present an efficient workaround to mitigate this drawback.

Clustered spatial contrastive learning We observe that each different vulnerability type might have some different vulnerability patterns causing it. For example, the *buffer overflow error* can have *"buffer copy without checking size of input"* or "*the improper validation of array index*", and other vulnerability patterns. Please refer to the appendix section for more details. We further observe that the vulnerable source code sections originated from the same vulnerability pattern have the similar F^{top} and tend to form a well-separated cluster as shown in Figure 3. Therefore, we propose to do clustering analysis (e.g., k-means) on F^{top} to group vulnerable source code sections with the same vulnerability patterns and employs contrastive learning to force them to become more similar as follows:

$$\mathcal{L}_{cscl} = \sum_{i \in I} 1_{y_i=1} \frac{-1}{|C(i)|} \sum_{c \in C(i)} \log \frac{\exp(\text{sim}(F_i^{top}, F_c^{top})/\tau)}{\sum_{a \in A(i)} \exp(\text{sim}(F_i^{top}, F_a^{top})/\tau)} \tag{6}$$

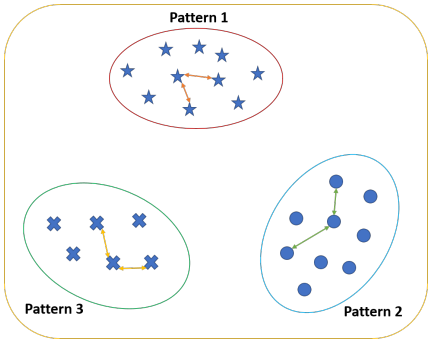

Figure 3: A demonstration of different vulnerability patterns forming different patterns in the latent space for the buffer overflow error. In this one, we assume that there are three different patterns causing the buffer overflow error (e.g., the CWE-119 dataset). Note that each data point in a pattern is a specific F^{top} of a corresponding function F. In reality, we can have more different vulnerability patterns in the the buffer overflow error.

where $I \equiv \{1...m\}$ is a set of indices of input data in a specific mini-batch, $A(i) \equiv I \setminus \{i\}$, $C(i) \equiv \{c \in A(i) : \widetilde{y}_c = \widetilde{y}_i \text{ and } y_c = 1\}$ is the set of indices of vulnerable source code sections labeled 1 which are in the same cluster as F_i except i, and $|C(i)|$ is its cardinality. Note that in (6), we apply k-means for the current mini-batch and denote $\tilde{y}_i$ as the cluster label of the source code section F_i.

Combining the objective functions mentioned in Eqs. (4 and 6), we arrive at the following objective function:

$$\max_{\varepsilon,q}(\mathbb{E}_F\mathbb{E}_{\tilde{F}|F}[\sum_{Y} p(Y|F)\log q(Y|\tilde{F})] - \alpha\mathcal{L}_{cscl}) \tag{7}$$

where $\alpha > 0$ is the trade-off hyper-parameter.

3 Experiments

3.1 Baseline approaches

We revised several possible baseline approaches and compared with our proposed method (i.e., named FinVulD-IC standing for Fine-grained Software Vulnerability Detection via Information Theory and Contrastive Learning). From machine learning and data mining perspectives, it seems that the existing methods in interpretable machine learning (Ribeiro et al., 2016; Shrikumar et al., 2017; Lundberg & Lee, 2017; Chen et al., 2018) with adoption are ready to apply. Unfortunately, besides Chen et al. (2018), none of others can be adopted to be applicable to the specific context of statement-grained vulnerability detection.

The main baseline to our proposed FinVulD-IC approach is the Information-theoretic code vulnerability highlighting (ICVH) introduced by Nguyen et al. (2021). We did not compare with VulDeeLocator (Li et al., 2020) because: i) it cannot work directly with source code (i.e., the source code needs to be compiled to the Lower Level Virtual Machine intermediate code and the method cannot be used when a program source code cannot be compiled), and ii) VulDeeLocator requires information relevant to vulnerable code 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 (i.e., where we do not use the information about the ground truth of vunerable code statements in the fine-grained software vulnerability detection problem).

3.2 Experimental setup

Experimental datasets We used two real-world datasets including the resource management error vulnerabilities (i.e., CWE-399 consisting of 1,010 vulnerable functions and 1,313 non-vulnerable functions) and the buffer error vulnerabilities (CWE-119 consisting of 5,582 vulnerable functions

and 5,099 non-vulnerable functions) collected by Li et al. (2018a). The minimum, mean, and maximum length of functions in CWE-399 and CWE-119 are (4; 51; 177) and (4; 21; 164) respectively. Note that to the identical functions in both CWE-119 and CWE-399 datasets, we keep one and remove the rest.

Labeling core vulnerable statements for evaluation The CWE-399 and CWE-119 datasets only have vulnerability labels at the function level (i.e., the function is vulnerable or non-vulnerable). In the training process, we do not use the information of vulnerable statements (i.e., the vulnerability labels at the statement level); however, this information is necessary to evaluate the models' performance. To obtain the ground truth of vulnerable code statements, inspired from Nguyen et al. (2021), we used the description of vulnerability information (i.e., the comments and annotations) in the original source code functions as well as the differences between the vulnerable versions and the fixed versions (i.e., non-vulnerable versions) of the source code functions.

Measures and evaluation The main purpose of our proposed FinVulD-IC method is to support programmers and developers to narrow down the vulnerable scope for seeking vulnerable statements. This would be helpful in the context that they need to identify several vulnerable statements from hundreds or thousands of lines of code. We aim to specify lines of statements (e.g., *top K=5*) so that with a high probability those lines cover most or all vulnerable statements. Bearing this incentive, and inspired from Nguyen et al. (2021), to evaluate the performance of the our proposed method and baselines, we use two measures introduced in Nguyen et al. (2021) including: *vulnerability coverage proportion (VCP)* (i.e., the proportion of correctly detected vulnerable statements over all vulnerable statements in a dataset) and *vulnerability coverage accuracy (VCA)* (i.e., the ratio of the successfully detected functions, having all vulnerable statements successfully detected, over all functions in a dataset). In addition to VCP and VCA measures, we also reported the label (i.e., Y) classification accuracy (ACC) on CWE-399 and CWE-119 datasets for the mentioned methods.

For the data processing and embedding, and the models' configuration, please refer to the appendix section.

3.3 Experimental results

Code vulnerability highlighting with selected code statements in the unsupervised setting We compared the performance of our proposed FinVulD-IC method with baselines including RSM (i.e., the random selection method, we first randomly chose K code statements from each function in the CWE-119 and CWE-399 datasets, and then we compute the VCP and VCA measures of the method for each dataset), L2X (Chen et al., 2018), and ICVH (Nguyen et al., 2021) in the unsupervised setting (i.e., we do not use any information about ground truth of vulnerable code statements in the training process) for highlighting the vulnerable code statements. We aim to find out the top K statements that mostly cause the vulnerability of each function. The number of selected code statements for each function is fixed equal to 5 or 10 as mentioned in Table 1.

Table 1: Performance results in terms of two main measures including VCP and VCA on the testing set of the CWE-399 and CWE-119 datasets for RSM, L2X, ICVH and FinVulD-IC methods with $K = 5$ and $K = 10$ (best performance is shown in **bold**).

Dataset	K	Method	VCP	VCA	ACC	Dataset	K	Method	VCP	VCA	ACC
CWE-399	5	RSM	16.2%	15.0%	NA	CWE-399	10	RSM	48.7%	38.0%	NA
		L2X	77.7%	68.0%	95.8%			L2X	80.4%	71.0%	94.7%
		ICVH	69.5%	54.7%	95.3%			ICVH	84.5%	77.0%	96.4%
		FinVulD-IC	**80.4%**	**72.0%**	**96.0%**			FinVulD-IC	**87.2%**	**81.0%**	**96.9%**
CWE-119	5	RSM	36.7%	27.2%	NA	CWE-119	10	RSM	49.9%	46.9%	NA
		L2X	89.2%	84.5%	93.1%			L2X	93.2%	90.3%	93.7%
		ICVH	77.2%	67.9%	92.6%			ICVH	93.5%	91.1%	**94.0%**
		FinVulD-IC	**90.9%**	**87.8%**	**93.8%**			FinVulD-IC	**97.5%**	**95.5%**	93.9%

The experimental results in Table 1 show that *our proposed FinVulD-IC method achieved a much higher performance for the VCP and VCA measures compared with the RSM, L2X, and ICVH methods on both CWE-399 and CWE-119 datasets in both cases $K = 5$ and $K = 10$*. For example, to the CWE-399 dataset with $K = 10$, our proposed FinVulD-IC method achieved 87.2% for VCP and 81.0% for VCA while (RSM, L2X, and ICVH) achieved (48.7%, 80.4%, and 84.5%) for VCP and (38.0%, 71.0%, and 77.0%) for VCA respectively. We also observed that the higher value of

selected code statement K was used, the higher performance for the VCP and VCA measures that the models obtained.

Furthermore, the higher classification accuracy (ACC) in almost cases on the CWE-399 and CWE-119 datasets with $K = 5$ and $K = 10$ shows *that our proposed FinVulD-IC method achieved a better highlighting performance in terms of making label predictions and highlighting the vulnerable code statements compared to the baselines.*

3.4 EXPLANATORY CAPABILITY OF OUR PROPOSED METHOD

In order to demonstrate the ability of our proposed method in detecting and highlighting the vulnerable code statements in the vulnerable functions to support security auditors and code developers, in this section, we show some visualizations of the selected code statements in some vulnerable functions. Note that, for demonstration purpose and simplicity, we choose simple and short vulnerable source code functions. We set $K = 5$ for these functions as mentioned in Figs. (4 and 5). In these figures, the colored lines (i.e., the green and red lines) highlight the detected code statements obtained when using our proposed method in the unsupervised learning setting. In addition, red lines specify the core vulnerable statements obtained from the ground truth, and these lines are detected by our method.

```
void func1 ( )
{
char * var1 ;
char * var2 = ( char * ) func2 ( 100 * sizeof ( char ) ) ;
var1 = var2 ;
switch ( 6 )
{
case 6 :
memset ( var1 , str , 100 - 1 ) ;
var1 [ 100 - 1 ] = str ;
break ;
var3 :
func3 ( str ) ;
break ;
}
{
char var4 [ 50 ] = str ;
strcpy ( var4 , var1 ) ;
func3 ( var1 ) ;
}
}
```

Figure 4: The source code function and selected code statements highlighted relevant to vulnerabilities are shown with $K = 5$. 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. For demonstration purpose, we choose a simple and short vulnerable source code function.

For example, in Fig. 4, the vulnerable function has two vulnerable code statements including "*memset (var1 , str , 100 - 1) ;*", which is a flaw because we initialize *var1* as a large buffer that is larger than the small buffer used in the sink, and "*strcpy (var4 , var1) ;*", which is a potential flaw because of the possible buffer overflow if data is larger than dest, which lead to a vulnerability. Our proposed method with $K = 5$ can detect these vulnerable statements which cause the corresponding function vulnerable.

In Fig. 5, 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 > wcslen (var7))*" which is potential flaw due to no maximum limitation for memory allocation. Our method with $K = 5$ can also detect all of these potential vulnerable code statements that make the corresponding function vulnerable.

```
void func1 ( )
{
size_t var1 ;
var1 = 0 ;
if ( 1 )
{
{
char var2 [ var3 ] = str ;
if ( fgets ( var2 , var3 , var4 ) != var5 )
{
var1 = func2 ( var2 , var5 , 0 ) ;
}
else
{
func3 ( str ) ;
}
}
}
```

```
if ( 1 )
{
{
wchar_t * var6 ;
if ( var1 > wcslen ( var7 ) )
{
var6 = new wchar_t [ var1 ] ;
wcscpy ( var6 , var7 ) ;
func4 ( var6 ) ;
delete [ ] var6 ;
}
else
{
func3 ( str ) ;
}
}
}
}
```

Figure 5: The left-hand and right-hand figures are the first and second parts of the function respectively. The source code function and selected code statements highlighted relevant to vulnerabilities are shown with $K = 5$. The 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. For demonstration purpose and simplicity, we choose a simple and short vulnerable source code function.

3.5 ABLATION STUDIES

In this section, we investigate the correlation between the number of chosen clusters guiding the computation of the proposed clustered spatial contrastive learning mentioned in Eq. (6), the trade-off hyper-parameter α representing for the weight of the proposed clustered spatial contrastive learning term in the final objective function 7, and the VCP and VCA measures for our proposed FinVulD-IC method. As mentioned in the experiments section, the number of chosen

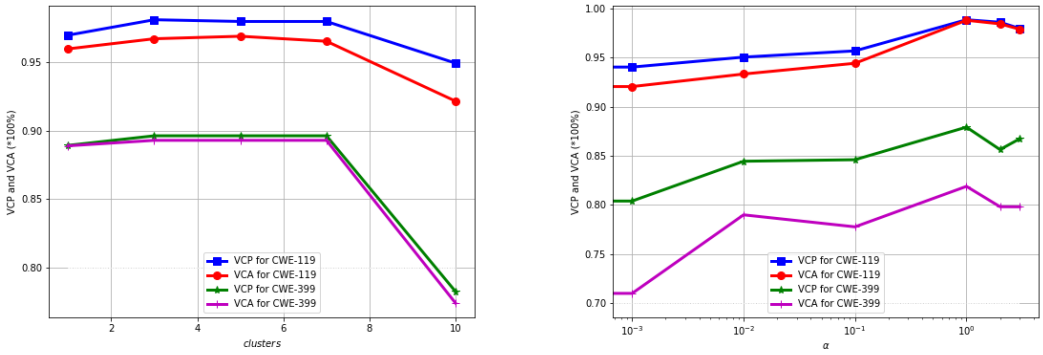

Figure 6: The correlation between the number of chosen clusters and the trade-off hyper-parameter α, and the VCP and VCA measures.

clusters used in our proposed clustered spatial contrastive learning is set equal to 3 while the trade-off parameter α is in $\{10^{-2}, 10^{-1}, 10^{0}\}$. In this ablation study, the number of chosen clusters varies in $\{1, 3, 5, 7, 10\}$ whilst the trade-off hyper-parameter α varies in $\{0, 10^{-3}, 10^{-2}, 10^{-1}, 10^{0}, 2 \times 10^{0}, 3 \times 10^{0}\}$.

As shown in Fig. 6 (the left-hand figure), we observe that our proposed FinVulD-IC method obtains a higher performance for the VCP and VCA measures on both CWE-119 and CWE-399 datasets when the chosen cluster is in $\{3, 5, 7\}$ compared to the case in which the chosen cluster is equal to 1 or 10. In the case of the chosen cluster equal to 1, we assume that to each vulnerability type (e.g., the buffer overflow error), there is only one dynamic pattern causing the corresponding vulnerability; however, in reality, for each vulnerability type, there are some vulnerability patterns as mentioned in section 2.2. To the case when we set the chosen cluster equal to 10, we may set the number of vulnerability patterns higher than the true one. These are the reasons why the model's performance in these cases are lower than the case when the chosen cluster varies in $\{3, 5, 7\}$ which can reflect more appropriate values of the true number of patterns in each dataset (i.e., CWE-399 or CWE-119).

To the case of the trade-off hyper-parameter α, the results in Fig. 6 (the right-hand figure) show that we can obtain a better model's performance when α varies in $\{10^{0}, 2 \times 10^{0}, 3 \times 10^{0}\}$ compared to the case in which α varies in $\{0, 10^{-3}, 10^{-2}, 10^{-1}\}$. It means that the clustered spatial contrastive learning term plays an important role in the training process because the model's performance is significantly improved, and the model's performance is much higher than the case without using this term (i.e., the value of α is set to 0). Furthermore, the results in Fig. 6 (the right-hand figure) also indicate that we should set the value of the trade-off hyper-parameter α higher than 10^{-1} to make sure that we use enough information of the clustered spatial contrastive learning term to enhance the representation learning and robust the selection process of vulnerability-relevant code statements.

4 Conclusion

In this paper, we have successfully proposed a novel end-to-end deep learning-based method for tackling the fine-grained software vulnerability detection problem. In particular, we first leverage the mutual information theory in learning a set of independent Bernoulli latent variables that can represent the relevance of the source code statements to the corresponding function's vulnerability. This one is named as a random selection process ε. We then propose a novel clustered spatial contrastive learning in order to further improve the representation learning and robust the random selection process ε. Specifically, our novel clustered spatial contrastive learning guides the random selection process ε to select and highlight hidden vulnerability pattern characterized by F^{top} in source code sections so that the vulnerable source code sections originated from the same vulnerability pattern are encouraged to have similar selected and highlighted vulnerability-relevant code statements. The experimental results show the superiority of our proposed FinVulD-IC method compared with other state-of-the-art baselines in selecting and highlighting the vulnerable code statements in source code functions.

REFERENCES

M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, Geoffrey 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)*, pp. 265–283, 2016.

M. Almorsy, J.C. 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*, ASE 2012, pp. 100–109, 2012. ISBN 978-1-4503-1204-2.

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

Xiao Cheng, Haoyu Wang, Jiayi Hua, Miao Zhang, Guoai Xu, Li Yi, and Yulei Sui. Static detection of control-flow-related vulnerabilities using graph embedding. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2019.

H. K. Dam, T. Tran, T. Pham, N. S. Wee, J. Grundy, and A. Ghose. Automatic feature learning for predicting vulnerable software components. *IEEE Transactions on Software Engineering*, 2018.

M. Dowd, J. McDonald, and J. Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley Professional, 2006. ISBN 0321444426.

Xu Duan, Jingzheng Wu, Shouling Ji, Zhiqing Rui, Tianyue Luo, Mutian Yang, and Yanjun Wu. Vulsniper: Focus your attention to shoot fine-grained vulnerabilities. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pp. 4665–4671, 2019.

Seyed M. Ghaffarian and Hamid R. Shahriari. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys (CSUR)*, 50 (4):56, 2017.

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*, CODASPY '16, pp. 85–96, 2016. ISBN 978-1-4503-3935-3.

Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. Supervised contrastive learning. *CoRR*, abs/2004.11362, 2020.

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*, pp. 595–614. IEEE Computer Society, 2017.

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

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*, ACSAC '16, pp. 201–213, 2016. ISBN 978-1-4503-4771-6.

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*, abs/1801.01681, 2018a.

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.

Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, Zhaoxuan Chen, Sujuan Wang, and Jialai Wang. Sysevr: A framework for using deep learning to detect software vulnerabilities. *CoRR*, abs/1807.06756, 2018b.

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.

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

McAfee and CSIS. Latest report from mcafee and csis uncovers the hidden costs of cybercrime beyond economic impact. 2020.

Van Nguyen, Trung Le, Olivier de Vel, Paul Montague, John Grundy, and Dinh Phung. Information-theoretic source code vulnerability highlighting. In *International Joint Conference on Neural Networks (IJCNN)*, 2021.

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

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*, 37(6):772–787, 2011.

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

N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15: 1929–1958, 2014.

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*, pp. 13–23, 2011.

Yuan Zhuang, Zhenguang Liu, Peng Qian, Qi Liu, Xiang Wang, and Qinming He. Smart contract vulnerability detection using graph neural network. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, pp. 3283–3290, 2020.

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*, ESEC/FSE '09, pp. 91–100, 2009. ISBN 978-1-60558-001-2.

A Appendix

Data processing and embedding

We preprocessed the datasets before injecting them into deep neural networks. We standardized 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. We then embedded source code statements into vectors. For instance, considering the following statement (C programming language) *"if(func3(func4(2,2),&var2)!=var11)*", to embed this code statement, we tokenized it to a sequence of tokens (e.g., *if,(,func3,(,func4,(,2,2,),&,var2,),!=,var11,)*), and then we used a 150-dimensional token embedding followed by a Dropout layer with a dropped fixed probability $p = 0.2$ and (a 1D convolutional layer with the filter size 150 and kernel size 3, and a 1D max pooling layer) or (a 1D max pooling layer) to encode each statement in a function F. Finally, a mini-batch of functions in which each function consisting of L encoded statements was fed to the models.

Model configuration

For the L2X (Chen et al., 2018) and ICVH (Nguyen et al., 2021) methods, they were proposed to work as explaining models aiming to explain the output of a learning model (i.e., which approximates the true conditional distribution $p(Y \mid F)$). To use these methods directly to deal with the problem of fine-grained software vulnerability detection, we keep their principles and apply them directly to approximate $p(Y \mid F)$ using $p(Y \mid \tilde{F})$ where $\tilde{F}$ consists of the selected vulnerability-relevant source code statements. To these methods, for the architecture of the random selection

network obtaining $\tilde{F}$ as well as the classifier working on $\tilde{F}$ to mimic $p(Y \mid F)$, we follow the structures mentioned in the corresponding original papers.

To our proposed method, for the $\omega(\cdot;\alpha)$ and $g(\cdot;\beta)$ networks, we used deep feed-forward neural networks having three and two hidden layers with the size of each hidden layer in $\{100, 300\}$. The dense hidden layers are followed by a ReLU function as nonlinearity and Dropout (Srivastava et al., 2014) with a retained fixed probability $p = 0.8$ as regularization. The last dense layer of the $\omega(\cdot;\alpha)$ network for learning a discrete distribution is followed by a sigmoid function while the last dense layer of the $g(\cdot;\beta)$ network is followed by a softmax function for predicting. The number of chosen clusters guiding the computation of the proposed clustered spatial contrastive learning mentioned in Eq. (6) is set to 3 while the trade-off hyper-parameter α representing for the weight of the proposed clustered spatial contrastive learning term in the final objective function 7 is in $\{10^{-2}, 10^{-1}, 10^{0}\}$, and the scalar temperature τ is in $\{0.5, 1.0\}$. The length of each function is padded or truncated to $L = 100$ code statements.

To our proposed method and baselines, we employed the Adam optimizer (Kingma & Ba, 2014) with an initial learning rate of 10^{-3}, while the mini-batch size is 100 and the temperature τ for the Gumbel softmax distribution is equal to 0.5. 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. For each dataset, we used 10 epochs for the training process. We additionally applied gradient clipping regularization to prevent over-fitting. For each method, we ran the corresponding model 5 times and reported the averaged VCP and VCA measures as well as the ACC. We ran our experiments in Python using Tensorflow (Abadi et al., 2016) on an Intel Xeon Processor E5-1660 which has 8 cores at 3.0 GHz and 128 GB RAM.

ADDITIONAL EXPERIMENTS

Auxiliary measure As mentioned in the experiments section in the paper, the main purpose of our proposed FinVulD-IC method is to support programmers and developers to narrow down the vulnerable scope for seeking vulnerable statements. This would be helpful in the context that they need to identify several vulnerable statements from hundreds or thousands of lines of code. We aim to specify lines of statements (e.g., *top K=5*) so that with a high probability those lines cover most or all vulnerable statements. Bearing this incentive, and inspired from Nguyen et al. (2021), to evaluate the performance of the our proposed method and baselines, we use two measures introduced in Nguyen et al. (2021) including : *vulnerability coverage proportion (VCP)* (i.e., the proportion of correctly detected vulnerable statements over all vulnerable statements in a dataset) and *vulnerability coverage accuracy (VCA)* (i.e., the ratio of the successfully detected functions, having all vulnerable statements successfully detected, over all functions in a dataset).

In practice, programmers and developers can set their preferable *top K* for the used methods, so that we need a measure that penalizes large *top K*. To this end, we propose an auxiliary measure named VCE (i.e., vulnerable coverage efficiency) which measures the percentage of vulnerable statements detected over the number of selected statements. For example, if a source code section has 3 core vulnerable code statements and using *top K*=5, we can successfully detect 2 vulnerable statements. The VCE measure in this case is equal to 2/5 = 0.4. This additional measure would offer a helpful measure of efficiency to users.

Table 2: Experimental results in terms of the auxiliary VCE measure on the testing set of the CWE-399 and CWE-119 datasets for RSM, L2X, ICVH and FinVulD-IC methods with $K = 5$.

Dataset	K	Method	VCE
CWE-399	5	RSM	9.1%
		L2X	43.6%
		ICVH	39.1%
		FinVulD-IC	**45.1%**
CWE-119	5	RSM	12.5%
		L2X	30.5%
		ICVH	26.4%
		FinVulD-IC	**31.1%**

We have computed the auxiliary VCE measure for our proposed FinVulD-IC method and baselines (i.e., RSM, L2X, and ICVH) with top $K = 5$ in the unsupervised setting. The experimental results mentioned in Table 2 show that our proposed FinVulD-IC method obtained the highest VCE measure in both CWE-399 and CWE-119 datasets compared with the baselines. In particular, to the CWE-399 dataset, the FinVulD-IC method gained 45.1% for the auxiliary VCE measure. It means that in this case, we can detect $5 \times 0.451 = 2.255$ vulnerable statements out of 5 spotted lines.