Statement-Level Vulnerability Detection: Learning Vulnerability Patterns Through Information Theory and Contrastive Learning

VAN NGUYEN*, Monash University, Australia
TRUNG LE, Monash University, Australia
CHAKKRIT TANTITHAMTHAVORN, Monash University, Australia
MICHAEL FU, Monash University, Australia
JOHN GRUNDY, Monash University, Australia
HUNG NGUYEN, Adelaide University, Australia
SEYIT CAMTEPE, CSIRO Data61, Australia
PAUL QUIRK, Defence Science and Technology Group, Australia
DINH PHUNG, Monash University, Australia

Software vulnerabilities are a serious and crucial concern. Typically, in a program or function consisting of hundreds or thousands of source code statements, there are only a few statements causing the corresponding vulnerabilities. Most current approaches to vulnerability labelling are done on a function or program level by experts with the assistance of machine learning tools. Extending this approach to the code statement level is much more costly and time-consuming and remains an open problem. In this paper, we propose a novel end-to-end deep learning-based approach to identify the vulnerability-relevant code statements of a specific function. Inspired by the specific structures observed in real-world vulnerable code, we first leverage mutual information for learning a set of latent variables representing the relevance of the source code statements to the corresponding function’s vulnerability. We then propose novel clustered spatial contrastive learning in order to further improve the representation learning and the robust selection process of vulnerability-relevant code statements. Experimental results on real-world datasets of 200k+ C/C++ functions show the superiority of our method over other state-of-the-art baselines. In general, our method obtains a higher performance in VCP, VCA, and Top-10 ACC measures of between 3% to 14% over the baselines when running on real-world datasets in an unsupervised setting. Our released source code samples are publicly available at https://github.com/vannguyennd/livuitcl.

CCS Concepts: • Computing methodologies → Artificial intelligence; • Security and privacy;

Additional Key Words and Phrases: Statement-level vulnerability detection, Mutual information, and Contrastive learning.

1 INTRODUCTION

It is common for computer software to contain 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 lurking in software development and deployment processes can and do create severe security breaches,
leading to a total financial loss of over USD 1 trillion, a more than 50 percent increase from 2018 [McAfee and CSIS(2020)]. Modern computer software contains many thousands or even millions of lines of code and often follows diverse development processes that integrate code from different development teams. Thus finding such vulnerabilities is extremely challenging. Although many solutions have been proposed for software vulnerability detection (SVD), the number of SVs and the severity of the threats imposed by them have steadily increased and caused considerable damage to individuals and companies [Ghaffarian and Shahriari(2017)]. These threats create an urgent need for more effective automatic tools and methods to 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 rule-based, 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 approaches and require the knowledge of domain experts that can be outdated and biased [Zimmermann et al.(2009)]. To mitigate this problem, deep learning solutions have been used to conduct SVD and have shown great advantages over machine learning approaches that use hand-crafted features, notably [Li et al.(2018b), Lin et al.(2018), Dam et al.(2018), Li et al.(2018a), Duan et al.(2019), Cheng et al.(2019), Zhuang et al.(2020), Nguyen et al.(2019), Nguyen et al.(2020a), Li et al.(2021b), Nguyen et al.(2021b)]. Despite achieving promising performance for SVD, current state-of-the-art deep learning-based methods (e.g., [Li et al.(2018b), Dam et al.(2018), Li et al.(2021b), Nguyen et al.(2021b)]) are only able to detect software vulnerabilities at a function or program level. Particularly, these existing approaches detect whether a source code “section” denoted by $F$ (e.g., a C/C++ function or program) is vulnerable. Hereafter, we use “section” or “function” or “program” to denote a collection of code statements.

There have recently been some proposed approaches, notably [Li et al.(2020), Nguyen et al.(2021a), Li et al.(2021a), Fu and Tantithamthavorn(2022), Hin et al.(2022), Ding et al.(2022)], dealing with the statement-level vulnerability detection problem. This includes highlighting statements that are highly relevant to the corresponding function’s vulnerability $Y$ (i.e., $Y \in \{0, 1\}$ where 1: vulnerable and 0: non-vulnerable) and associated code statements. Although these recently introduced methods achieve promising results, they are limited in understanding and leveraging the relationships of hidden vulnerable patterns inside and between source code sections for facilitating statement-level vulnerability detection.

In this paper, we propose a novel end-to-end deep learning-based approach for statement-level software vulnerability detection that allows us to find and highlight code statements, in functions or programs, that are highly relevant to the presence of significant source code vulnerabilities. To determine key vulnerability-relevant code statements in each source code function $F$ consisting of $L$ code statements from $f_1$ to $f_L$ (in practice, each code statement is represented as a vector using a learnable embedding method, please refer to Section 4.3 for details), we (1) first introduce a learnable selection process $\epsilon$ that aims to learn a set of independent Bernoulli latent variables $z \in \{0, 1\}^L$ representing the relevance of the code statements to the corresponding function’s vulnerability $Y$, i.e., 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 construct $\tilde{F} = \epsilon(F)$ (i.e., the subset of code statements that actually lead to the vulnerability $Y$ of the function $F$) by $\tilde{F} = z(F) \odot F$, where $\odot$ represents the element-wise product. Inspired by [Chen et al.(2018), Nguyen et al.(2021a)], to ensure and enforce the selection process $\epsilon$ obtaining the most meaningful $\tilde{F}$ (i.e. $\tilde{F}$ can predict the vulnerability $Y$ of $F$ correctly), we then (2) maximize the mutual information between $\tilde{F}$ and $Y$.

From the real source code sections (i.e., functions or programs), we observe that there are often groups of several core statements that cause their vulnerability. If we group these core statements
together, we have vulnerability patterns shared across vulnerable source code sections. These hidden vulnerability patterns can be embedded into real-world source code sections at different spatial locations to form realistic vulnerable source code sections. Given a set of vulnerable source code sections, we need to devise a mechanism to guide the selection process $\varepsilon$ to select and highlight hidden vulnerability patterns. This is 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, to characterize a vulnerable source code section $F$, we consider $F_{\text{top}}$ that includes $K$ 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 $F_{\text{top}}$ which tend to form well-separated clusters. Based on this observation, we (3) propose a clustered spatial contrastive learning term, inspired by supervised contrastive learning [Khosla et al.(2020)], which ensures and enforces three important properties in the source code data. This includes (i) the vulnerable and non-vulnerable source code sections should have different representations of $F_{\text{top}}$, (ii) the vulnerable source code sections from different hidden vulnerability patterns are also encouraged to have different representations of $F_{\text{top}}$ while (iii) the vulnerable source code sections in the similar hidden vulnerability patterns are trained to have close representations of $F_{\text{top}}$. Ensuring these properties facilitates the selection process for helping boost the data representation learning in figuring out and selecting the code vulnerable statements of vulnerable source code sections.

The key contributions of this work include:

- We introduce an end-to-end deep learning-based approach for statement-level vulnerability detection based on an information-theoretic perspective in forming the model selection and guiding the whole training process. We propose a novel clustered spatial contrastive learning term inspired by contrastive learning [Khosla et al.(2020)] to model important properties in the relationship of vulnerable patterns between the source code sections. Our method shares the intuition with [Nguyen et al.(2021a)] of using mutual information in selecting the important subset of code statements in each source code function, however, in our method, we propose a novel clustered spatial contrastive learning to learn important properties from the source code data for boosting the data representations and learning hidden vulnerability patterns that facilitate the selection of vulnerable code statements in vulnerable source code sections (i.e., functions or programs).

- We conduct extensive experiments on real-world source code datasets, including CWE-119 and CWE-399 collected by [Li et al.(2018b)], and a big C/C++ source code dataset, namely Big-Vul, provided by [Fan et al.(2020)]. Our experiments on these three real-world datasets of 200k+ C/C++ functions show the superiority of our proposed method in selecting and highlighting the core vulnerable statements over state-of-the-art baselines.

2 MOTIVATION

Real-world source code programs or functions can consist of many hundreds or thousands of source code statements, and only a few of them (usually a few core statements) cause the corresponding vulnerabilities. Figure 1 shows an example of a simple vulnerable C/C++ source code function. Among many lines of code statements, only two statements, highlighted in red, actually lead to the function’s vulnerability. The core statements underpinning a vulnerability are even much sparser in the source code of real-world applications. In Figure 1, for demonstration purposes, we choose a short C/C++ source code function in which the statement "data = dataBuffer - 8;" is a vulnerability because we set the 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.
It is worth noting that most of the existing state-of-the-art SVD approaches can only detect whether a source code “section” $F$ (e.g., a C/C++ function or program) is vulnerable. In contrast, by doing statement-level vulnerability detection (i.e., learning to select the vulnerable statements in each vulnerable source code section), e.g., [Li et al.(2020), Nguyen et al.(2021a), Li et al.(2021a), Fu and Tantithamthavorn(2022), Hin et al.(2022), Ding et al.(2022)], we can help speed up the process of isolating and detecting software vulnerabilities thereby reducing the time and cost involved. Doing statement-level vulnerability detection is the main target of our proposed approach.

3 OUR APPROACH

3.1 The problem setting

We denote a source code section (e.g., a C/C++ function or program) as $F = [f_1, \ldots, f_R]$, which consists of $L$ code statements from $f_1$ to $f_R$ ($L$ can 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 $\tilde{F} = [f_{i_1}, \ldots, f_{i_K}] = [f_i]_{i \in S}$ where $S = \{i_1, \ldots, i_K\} \subset \{1, \ldots, L\}$ ($i_1 < i_2 < \ldots < i_K$). To select the vulnerability-relevant statements $\tilde{F}$ for each specific source code section $F$, we apply to use a learnable selection process $\epsilon$, i.e., $\tilde{F} = \epsilon (F)$, whose training principle and construction are presented in the following subsection. We then propose novel clustered spatial contrastive learning, which can model important properties for the relationship of the source code sections, presented in the following sections to further improve the representation learning and the robust selection process of $\tilde{F}$.

The focus of our proposed method is to tackle the most challenging reality setting where most available datasets only have the vulnerability label (i.e., $Y$) at the source code function level i.e., only denote whether a function $F$ is vulnerable, by experts with the assistance of machine learning tools. They do not contain information of which particular source code statement(s) cause vulnerabilities. In the training process, our method only requires a vulnerability label at the source code function level (i.e., $Y$) and is capable of pointing out the vulnerability-relevant statements. In the context of statement-level vulnerability detection, this setting is considered as the unsupervised one as mentioned in [Nguyen et al.(2021a)], 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
Fig. 2. The overall architecture of our LEAP method. Given a mini-batch of source code sections (i.e., from $F^1$ to $F^i$), to each code section, e.g., $F^i$, the selection process $\varepsilon$ learns a set of independent Bernoulli latent variables $z \in \{0, 1\}^L$ representing the relevance of the code statements to the corresponding function’s vulnerability $Y^i$. For demonstration purposes, we assume that there are six code statements in $F^i$. However, in reality, this number can be in the hundreds. We then construct $\tilde{F}^i$ (i.e., the subset of code statements that actually lead to the vulnerability $Y^i$) by $\tilde{F}^i = z^i (F^i) \odot F^i$. Importantly, for ensuring and enforcing $\varepsilon$ obtaining the most meaningful $\tilde{F}^i$ (i.e., $\tilde{F}^i$ can predict the vulnerability $Y^i$ of $F^i$ correctly), we maximize the mutual information between $\tilde{F}^i$ and $Y^i$. The proposed clustered spatial contrastive learning helps to learn and enforce important properties in the source code data for boosting the data representation learning in figuring out and selecting vulnerable patterns and vulnerable statements in each vulnerable source code section.

The ground truth of vulnerable code statements causing vulnerabilities in source code sections is only used in the evaluation process.

In Sections 3.2 and 3.3 as follows, we present the methodology of our proposed approach for identifying source code sections causing the vulnerability of each source code section. In particular, in Section 3.2, we first present the way we construct a learnable selection process $\varepsilon$ and use mutual information to ensure and enforce the selection process $\varepsilon$ obtaining the most important source code statements, denoted by $\tilde{F}$, which causes the corresponding label $Y$ as well as forms the corresponding vulnerability pattern of the source code section $F$. In Section 3.3, we then present our novel clustered spatial contrastive learning in guiding the selection process $\varepsilon$ to model important properties in the relationships of vulnerable patterns between the source code sections to boost the data representation learning in figuring out and selecting the source code vulnerable statements.

3.2 Learning to select vulnerable-relevant statements and the training principle

3.2.1 Vulnerability-relevant statements selection process. Giving a source code function $F$ consisting of $L$ code statements from $f_1$ to $f_L$ (i.e., in practice, each code statement $f_i$ is represented as a vector using a learnable embedding method, please refer to Section 4.3 for details), to figure
To construct \( z = \{z_i\}_{i=1}^{L} \), we model \( z \sim \text{MultiBernoulli}(p) = \prod_{i=1}^{L} \text{Bernoulli}(p_i) \), which indicates \( f_i \) is related to the vulnerability \( Y \) of \( F \) (i.e., if \( z_i \) is equal to 1, the statement \( f_i \) plays an important role causing the vulnerability \( Y \)).

As \( z \) depends on \( F \), we denote \( z(F) \). With \( z \), we construct \( \tilde{F} = \epsilon(F) \) (i.e., the subset of code statements that actually lead to the vulnerability \( Y \) of the function \( F \)) by \( \tilde{F} = z(F) \odot F \), where \( \odot \) represents the element-wise product. Please refer to Figure 2 for the corresponding visualization.

### 3.2.2 Gumbel-Softmax trick for continuous optimization

Recall that the vulnerability-relevant selected code statements \( \tilde{F} \) are constructed by \( \tilde{F} = z(F) \odot F \) where \( z \sim \text{MultiBernoulli}(p) \) with \( p \) is the output of the selection process \( \epsilon \) parameterized by a neural network \( \omega(., \alpha) \). To make this computational process (having a sampling operation) continuous and differentiable during training, we apply the temperature-dependent Gumbel-Softmax trick [Jang et al.(2016)] for relaxing each Bernoulli variable \( z_i \). Particularly, we sample \( z_i(F; \alpha) \sim \text{Concrete}(\omega_i(F; \alpha), 1 - \omega_i(F; \alpha)) \):

\[
z_i(F; \alpha) = \frac{\exp((\log \omega_i(F; \alpha) + a_i)/\nu)}{\exp((\log \omega_i(F; \alpha) + a_i)/\nu) + \exp((\log (1 - \omega_i(F; \alpha)) + b_i)/\nu)}
\]

where \( \nu \) is a temperature parameter (i.e., that allows us to control how closely a continuous representation from a Gumbel-Softmax distribution approximates this from the corresponding discrete representation from a discrete distribution (e.g., the Bernoulli distribution), random noises \( a_i \) and \( b_i \) independently drawn from Gumbel distribution \( G = -\log(-\log u) \) with \( u \sim \text{Uniform}(0, 1) \).

### 3.2.3 Mutual information for guiding the selection process

**Mutual information.** In information theory [Shannon(1998), Cover and Thomas(2006)], mutual information (MI) is used to measure the mutual dependence between two random variables. In particular, MI quantifies the amount of information obtained about one random variable by observing the other random variable. To illustrate, consider a scenario where \( A \) denotes the outcome of rolling a standard 6-sided die, and \( B \) represents whether the roll results in an even number (0 for even, 1 for odd). Evidently, the information conveyed by \( B \) provides insights into the value of \( A \), and vice versa. In other words, these random variables possess mutual information.

**How mutual information guides the selection process.** Leveraging the important property of mutual information (quantifying the amount of information obtained about one random variable by observing the other random variable) and inspired by [Chen et al.(2018)], we maximize the mutual information between \( \tilde{F} \) and \( Y \) as mentioned in Eq. (1). Specifically, we aim that via Eq. (1), by using the information from \( Y \), the selection process \( \epsilon \) will be enforced to obtain the most meaningful \( \tilde{F} \) (i.e., \( \tilde{F} \) can predict the vulnerability \( Y \) of \( F \) correctly). If we view \( \tilde{F} \) and \( Y \) as random variables, the selection process \( \epsilon \) is learned by maximizing the mutual information between \( \tilde{F} \) and \( Y \) as follows:

\[
\max_{\epsilon} I(\tilde{F}, Y).
\]

We expand Eq. (1) further as the Kullback-Leibler divergence (i.e., it measures the relative entropy or difference in information represented by two distributions) of the product of marginal
distributions of \( \hat{F} \) and \( Y \) from their joint distribution:

\[
\mathbb{I}(\hat{F}, Y) = \int p(\hat{F}, Y) \log \frac{p(\hat{F}, Y)}{p(\hat{F})p(Y)} d\hat{F}dY \\
\geq \int p(Y, \hat{F}) \log \frac{q(Y | \hat{F})}{p(Y)} dYd\hat{F}
\]

In practice, estimating mutual information is challenging as we typically have access to samples but not the underlying distributions. Therefore, in the above derivation, we use a variational distribution \( q(Y | \hat{F}) \) to approximate the posterior \( p(Y | \hat{F}) \), hence deriving a variational lower bound of \( \mathbb{I}(\hat{F}, Y) \) for which the equality holds if \( q(Y | \hat{F}) = p(Y | \hat{F}) \). This can be further expanded as:

\[
\mathbb{I}(\hat{F}, Y) \geq \int p(Y, \hat{F}, F) \log \frac{q(Y | \hat{F})}{p(Y)} dYd\hat{F}dF \\
= \mathbb{E}_{\hat{F}} \mathbb{E}_{F|\hat{F}} \left[ \sum_Y p(Y|F) \log q(Y|\hat{F}) \right] + \text{const}
\]

We note that \( \hat{F}|F := \hat{F} \sim p(\cdot|F) := e(F) \) is the same representation of the selection process and \( p(Y|F) \) as mentioned before is the ground-truth conditional distribution of the \( F \)'s label on all of its features.

To model the conditional variational distribution \( q(Y|\hat{F}) \), we introduce a classifier implemented with a neural network \( g(\hat{F}; \beta) \), which takes \( \hat{F} \) 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:

\[
\max_{\alpha, q} \mathbb{E}_{\hat{F}} \mathbb{E}_{F|\hat{F}} \left[ \sum_Y p(Y|F) \log q(Y|\hat{F}) \right].
\]

The mutual information facilitates a joint training process for the classifier and the selection process. The classifier learns to identify a subset of 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.

### 3.3 Clustered spatial contrastive learning

**Motivation.** For each vulnerable function \( F \), we observe that there are few statements causing a vulnerability. If we group those core statements together, they form vulnerability patterns. For example, the top-left-hand figure in Figure 3 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., \( \text{int} * \text{array} \)) via specific index and save this value into a variable (i.e., \( \text{value} \)). 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.

As shown in Figure 3, this vulnerability pattern is embedded into real-world functions \( \text{getValue} \) and \( \text{takeArrayValue} \) 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 \( F^{\text{top}} \) as a subset of \( F \) including its \( K \) statements with the top \( K \) selection probability \( p_i = \omega_i (F; \alpha) \).
Fig. 3. An example of the improper validation of array index flaw pattern (i.e., the top-left-hand figure) with two real-world source code functions (i.e., `takeArrayValue` and `getValue`) containing this pattern. In each function, there are some parts (i.e., denoted by “...”) omitted for brevity.

and employ \( F_{\text{top}} \) to characterize the predicted vulnerability pattern of \( F \). It is worth noting that the statements \( F_{\text{top}} \) preserves the order in \( F \).

Fig. 4. A graphic showing two vectors with cosine similarities close to 1, close to 0, and close to -1. The similarity of two vectors is measured by the cosine of the angle between them. The similarity can take values between -1 and +1. Smaller angles between vectors produce larger cosine values, indicating greater cosine similarity.

To enforce two vulnerable source code sections originated from the same vulnerability pattern having the same \( F_{\text{top}} \), a possible 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:

\[
L_{\text{scl}} = \sum_{i=1}^{m} 1_{y_i=1} \sum_{p \in P(i)} \log \frac{\exp(\text{sim}(F_{i_{\text{top}}}, F_{p_{\text{top}}})/\tau)}{\sum_{a \in A(i)} \exp(\text{sim}(F_{i_{\text{top}}}, F_{a_{\text{top}}})/\tau)}
\]

(5)

where \( I \equiv \{1...m\} \) is a set of indices of input data in a specific mini-batch; \( \text{sim} \) is the cosine similarity (i.e., a metric used to measure the similarity of two vectors. Specifically, it measures the similarity in the direction or orientation of the vectors ignoring differences in their magnitude or scale. The similarity of two vectors is measured by the cosine of the angle between them (please refer to Figure 4 for details). Note that to form the corresponding vector for each \( F_{\text{top}} \) of each source code section \( F \) in the latent space for calculating the cosine similarity, we simply concatenate all vectors where
each vector stands for a representation of a code statement in $F_{top}$; $\tau > 0$ 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 (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 to share the same selected and highlighted vulnerability pattern, it seems to overdo this by forcing all vulnerable source code sections to share the same vulnerability pattern. **In 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 access with incorrect length", "improper validation of array index", or "expired pointer dereference" as mentioned in Figure 5. We further observe that the vulnerable source code sections originated from the same vulnerability pattern tend to have the similar $F_{top}$ and form a well-separated cluster as shown in Figure 5.

![Figure 5](image_url)

**Fig. 5.** A demonstration of different vulnerability patterns forming different patterns in the latent space for the buffer overflow error. In particular, Pattern 1 stands for the expired pointer dereference flaw in which the program dereferences a pointer containing a location for memory that was previously valid, but it is no longer valid. Pattern 2 represents the improper validation of the array index flaw in which the product uses untrusted input when using an array index, but the product does not validate or incorrectly validates the index to ensure the index references a valid position within the array while Pattern 3 presents the buffer access with an incorrect length value flaw in which the software uses a sequential operation to read or write a buffer, but it may use an incorrect length value resulting in accessing memory that is outside of the bounds of the buffer. Note that each data point in a pattern is a specific $F_{top}$ (e.g., the colored background lines) of a corresponding function $F$. In this demonstration, we assume that there are three different patterns causing the buffer overflow error. In reality, the number of vulnerability patterns causing the buffer overflow error can be higher.
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:

\[
L_{csl} = \sum_{i \in I} 1_{Y^t_i = 1} \frac{-1}{|C(i)|} \sum_{c \in C(i)} \log \frac{\exp(\text{sim}(F_{itop}, F_{ctop})/\tau)}{\sum_{a \in A(i)} \exp(\text{sim}(F_{itop}, F_{atop})/\tau)}
\]  

where $I \equiv \{1...m\}$ is a set of indices of input data in a specific mini-batch; $A(i) \equiv I \setminus \{i\}$, $C(i) \equiv \{c \in A(i) : \bar{Y}^c = \bar{Y}^i$ 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 Eq. (6), we apply $k$-means for the current mini-batch and denote $\bar{Y}^i$ as the cluster label of the source code section $F^i$.

It is worth noting that our proposed clustered spatial contrastive learning helps to enforce three important properties in the source code data including (i) the vulnerable and non-vulnerable source code sections should have different representations of $F_{top}$, (ii) the vulnerable source code sections from different hidden vulnerability patterns are also encouraged to have different representations of $F_{top}$ while (iii) the vulnerable source code sections in the similar hidden vulnerability patterns are trained to have close representations of $F_{top}$. Ensuring these properties facilitates the selection process for helping boost the data representation learning in figuring out and selecting the source code vulnerable statements of vulnerable source code sections (e.g., functions).

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

\[
\max_{\epsilon, q} \{E_F E_{\tilde{F}} | F \left[ \sum_Y p(Y|F) \log q(Y|\tilde{F}) \right] - \alpha L_{csl} \}
\]  

where $\alpha > 0$ is the trade-off hyper-parameter. The overall architecture of our proposed method is depicted in Figure 2.

4 EXPERIMENTAL DESIGN

4.1 Research Questions

We wanted to evaluate our LEAP approach and compare it with other baselines for statement-level vulnerability detection. Below, we present the key research questions to answer in our study.

(RQ1) Can our LEAP approach identify vulnerable code statements in vulnerable functions in an unsupervised setting (where there is no ground truth of vulnerable code statements required in the training process)? This is the main focus of our proposed method where we aim to tackle the most challenging reality setting in which most publicly available datasets only have the vulnerability label (i.e., $Y$) at the source code function level, i.e., only denote whether a function $F$ is vulnerable, by experts with the assistance of machine learning tools. They do not contain information on which source code statement(s) cause vulnerabilities. Therefore, proposing a method for statement-level vulnerability detection that can be trained without using any ground truth of vulnerable code statements play an important role in this challenging reality context.

(RQ2) Can our LEAP approach identify vulnerable code statements in vulnerable functions more accurately in a semi-supervised setting than in an unsupervised setting? In some scenarios, a small portion of the training data may have the ground truth of vulnerable code statements (i.e., the semi-supervised setting). Therefore, in this research question, we aim to investigate the performance of our proposed method (and the baselines) when we incorporate the annotations of the vulnerable statements of a small portion of the training data in the training
process. In particular, we assume that 10% of the training data has the ground truth of vulnerable code statements.

(RQ3) **What are the contributions of the components of our LEAP approach (mutual information and our proposed cluster spatial contrastive learning)?** We aim to investigate the effectiveness of our proposed cluster spatial contrastive learning in Eq. (6) in terms of facilitating the data representation learning process and helping figure out the hidden vulnerability pattern for improving the model’s ability to select the vulnerable statements in each source code data compared to the cases of using the normal contrastive learning in Eq.(5) and without using contrastive learning (i.e., in this case, we only use the mutual information for guiding the whole training process).

4.2 **Studied datasets**

We used three real-world datasets including the **CWE-399 dataset** with 1,010 and 1,313 vulnerable/non-vulnerable functions for resource management error vulnerabilities, the **CWE-119 dataset** with 5,582 and 5,099 vulnerable/non-vulnerable functions for the buffer error vulnerabilities, and a big C/C++ dataset, namely Big-Vul, provided by [Fan et al.(2020)] containing many types of vulnerabilities such as Out-of-bounds Write, Improper Input Validation, and Path Traversal.

For the CWE-399 and CWE-199 datasets collected by [Li et al.(2018b)], we used the ones processed by [Nguyen et al.(2021a)]. Additionally, the Big-Vul dataset, one of the largest vulnerability datasets having the ground truth at the statement level, was collected from 348 open-source Github projects from 2002 to 2019. It consists of 188,636 C/C++ source code functions where a ratio of vulnerable functions is 5.7% (i.e., 10,900 vulnerable functions).

For the training process of the unsupervised setting, 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.

4.3 **Data processing and embedding**

We preprocessed the datasets before injecting them into deep neural networks. In particular, we standardized the source code by removing comments and non-ASCII characters for the Big-Vul dataset while for the CWE-399 and CWE-199 datasets, we used the ones preprocessed by [Nguyen et al.(2021a)]. We then embedded source code statements into vectors. For instance, consider the following statement (C programming language) 

```
for(i=0;i<10;i++)
```

We tokenized this code statement into a sequence of tokens (e.g., `for(i=0,i<10,i++)`) to encode each statement in a function. Finally, a mini-batch of functions consisting of L encoded statements was fed to the models.

Different from the L2X [Chen et al.(2018)], ICVH [Nguyen et al.(2021a)], and our LEAP methods, the LineVul method [Fu and Tantithamthavorn(2022)] based on the CodeBERT tokenizer and CodeBERT pre-trained model from [Feng et al.(2020)] to tokenize and generate vector representations of source code functions.

4.4 **Measures and evaluation**

The main purpose of our LEAP method is to support security analysts and code developers to narrow down the vulnerable scope in their search for 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 identify source code statements (e.g., top K=10) so that with a high
probability these statements cover most or all vulnerable statements in the corresponding source code section.

To evaluate the performance of our LEAP method and baselines, we use two main measures introduced in [Nguyen et al.(2021a)] 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 addition to the VCP and VCA measures, we also report two other measures including Top-10 Accuracy (i.e., it measures the percentage of vulnerable functions where at least one actual vulnerable lines appear in the top-10 ranking) and Initial False Alarm (IFA) (i.e., it measures the number of incorrectly predicted lines (i.e., non-vulnerable lines incorrectly predicted as vulnerable or false alarms) that security analysts need to inspect until finding the first actual vulnerable line for a given function) used in [Fu and Tantithamthavorn(2022)].

4.5 Baseline Methods

We used several baseline approaches to compare to our proposed method LEAP (Statement-Level Vulnerability Detection: Learning Vulnerability Patterns Through Information Theory and Contrastive Learning). The main baselines to our method are ICVH [Nguyen et al.(2021a)] and LineVul [Fu and Tantithamthavorn(2022)]. From the interpretable machine-learning perspective, it seems that the existing methods [Ribeiro et al.(2016), Shrikumar et al.(2017), Lundberg and Lee(2017), Chen et al.(2018)] with adaptations can be applied. However, besides L2X [Chen et al.(2018)], none of the others is applicable to the context of statement-level vulnerability detection. It is worth noting that unlike L2X, ICVH, and our proposed methods, LineVul does not work straightforwardly at the statement level. In particular, it works at the token level to find out the important weight (the attention score) of each code token, from the source code section input, contributing to the model’s prediction. After obtaining the attention scores for all code tokens, the authors integrate those scores into statement scores to find out the vulnerable statements.

Similar to ICVH, we did not compare our method to VulDeeLocator [Li et al.(2020)] because: i) it cannot work directly with the source code, which needs to be compiled to the Lower Level Virtual Machine code, and ii) it cannot be operated in the unsupervised setting because of requiring information relevant to vulnerable code statements. About two other methods for statement-level vulnerability detection including LineVD [Hin et al.(2022)] and VELVET [Ding et al.(2022)], we also did not compare our method with these baselines because they cannot be operated in the unsupervised setting (i.e., this is a challenging setting and the main focus of our paper).

4.6 Model Configuration

For the L2X [Chen et al.(2018)] and ICVH [Nguyen et al.(2021a)] 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 statement-level 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 LEAP 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 temperature $\nu$ for the Gumbel softmax distribution is equal to $0.5$. The number of chosen clusters guiding the computation of the proposed clustered spatial contrastive learning mentioned in Eq. (6) is in $\{3, 5, 7\}$. The trade-off hyper-parameter $\alpha$ representing the weight of the proposed clustered spatial contrastive learning term in the final objective function (mentioned in Eq. (7)) is in $\{10^{-2}, 10^{-1}, 10^0\}$ while the scalar temperature $\tau$ is equal to $0.5$. It is worth noting that the length $(L)$ of each function is padded or truncated to 100 code statements for the CWE-119 and CWE-399 datasets while to the Big-Vul dataset, $L$ is set to 150. We chose these values based on the fact that for the CWE-119 and CWE-399 datasets, over 90% of the source code functions, have the number of code statements less than or equal to 100 while this value is equal to 150 for the Big-Vul dataset. Furthermore, almost all important information relevant to the vulnerability of each source code function lies in the 100 and 150 first code statements for the (CWE-119 and CWE-399) and Big-Vul datasets respectively.

For our LEAP method and baselines, we employed the Adam optimizer [Kingma and Ba(2014)] with an initial learning rate in $\{10^{-3}, 10^{-4}\}$, while the mini-batch size is 100. We split the data of each dataset 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, VCA, Top10 ACC, and IFA measures. We ran our experiments in Python using Tensorflow [Abadi et al.(2016)] for the used methods on an Intel E5-2680, having 12 CPU Cores at 2.5 GHz with 128GB RAM, integrated NVIDIA Tesla K80.

For the LineVul method, we kept the proposed architecture (i.e., a Transformer-based model) and used the shared source code written in Python using Pytorch [Paszke et al.(2019)] from the authors [Fu and Tantithamthavorn(2022)].

5 EXPERIMENTAL RESULTS

RQ1: Can our LEAP approach identify vulnerable code statements in vulnerable functions in an unsupervised setting (where there is no ground truth of vulnerable code statements required in the training process)?

Approach. We compared the performance of our LEAP method with baselines including L2X [Chen et al.(2018)], ICVH [Nguyen et al.(2021a)], and LineVul [Fu and Tantithamthavorn(2022)] in the unsupervised setting (i.e., we do not use any information about the ground truth of vulnerable code statements in the training process) for localizing the vulnerable code statements. We aim to find out the top $K$ statements that mostly cause the vulnerability of each function. In this experiment, the number of selected code statements for each function is fixed equal to 10 shown in Table 1.

Result. The results in Table 1 show that our LEAP method achieved a much higher performance on statement-level vulnerability detection measures, including VCP, VCA, and Top-10 ACC, compared to the L2X, ICVH, and LineVul methods on the CWE-399, CWE-119, and Big-Vul datasets. To the IFA measure, our LEAP method also achieved a higher performance on the CWE-399 and Big-Vul datasets while for the CWE-119 dataset, our LEAP method obtained a comparable value to the highest one (i.e., 2.0) from the LineVul baseline.

Generally, our LEAP method obtained a higher performance on the VCP, VCA, and Top-10 ACC measures – from 4% to 14% for the CWE-119 dataset, from 3% to 7% for the CWE-399 dataset, and 3% to 11% for the Big-Vul dataset compared with the baselines. For example, to the CWE-399 dataset
Table 1. Performance results in terms of the main measures for statement-level vulnerability detection including VCP, VCA, Top-10 accuracy (Top-10 ACC), and IFA on the testing set of the CWE-399, CWE-119, and Big-Vul datasets for the L2X, ICVH, LineVul and LEAP methods with $K = 10$ (the best performance is shown in **bold**).

| Dataset    | K  | Method                          | VCP  | VCA  | Top-10 ACC | IFA |
|------------|----|---------------------------------|------|------|------------|-----|
| CWE-399    | 10 | L2X [Chen et al.(2018)]        | 88.5%| 83.0%| 83.0%      | 3.8 |
|            |    | ICVH [Nguyen et al.(2021a)]   | 84.5%| 77.0%| 81.0%      | 5.5 |
|            |    | LineVul [Fu and Tantithamthavorn(2022)] | 92.0%| 89.0%| 91.0%      | 3.8 |
|            |    | LEAP (ours)                    | 96.6%| 95.0%| 95.0%      | 2.4 |
| CWE-119    | 10 | L2X [Chen et al.(2018)]        | 93.2%| 90.3%| 94.1%      | 3.3 |
|            |    | ICVH [Nguyen et al.(2021a)]   | 93.5%| 91.1%| 94.5%      | 2.2 |
|            |    | LineVul [Fu and Tantithamthavorn(2022)] | 93.0%| 89.0%| 91.0%      | 2.0 |
|            |    | LEAP (ours)                    | 97.5%| 96.5%| 97.6%      | 2.1 |
| Big-Vul    | 10 | L2X [Chen et al.(2018)]        | 65.5%| 60.9%| 69.7%      | 2.2 |
|            |    | ICVH [Nguyen et al.(2021a)]   | 73.8%| 69.6%| 76.8%      | 2.2 |
|            |    | LineVul [Fu and Tantithamthavorn(2022)] | 62.0%| 60.0%| 74.0%      | 3.7 |
|            |    | LEAP (ours)                    | 80.5%| 77.6%| 80.6%      | 1.5 |

with $K = 10$, our LEAP method achieved 96.6% for VCP, 95.0% for both VCA and Top-10 ACC, and 2.4 for IFA while (the L2X, ICVH, and LineVul methods) achieved (88.5%, 84.5%, and 92.0%) for VCP, (83.0%, 77.0%, and 89.0%) for VCA, (83.0%, 81.0%, and 91.0%) for Top-10 ACC, and (3.8, 5.5, and 3.8) for IFA respectively.

In addition to the main measures for statement-level vulnerability detection including VCP, VCA, Top-10 accuracy, and IFA, we also computed the function classification accuracy (ACC) for our LEAP method and baselines (i.e., the L2X, ICVH, and LineVul methods) for the used datasets. With $K = 10$, our LEAP method and baselines all obtained an ACC higher than 96%, 93%, and 91% for CWE-399, CWE-119, and Big-Vul datasets respectively.

**In conclusion for RQ1**: The experimental results (in Table 1) on the statement-level vulnerability detection measures (i.e., VCP, VCA, Top-10 accuracy (Top-10 ACC), and IFA) along with the function classification accuracy (ACC) show the superiority of our LEAP method in achieving a high performance in terms of making the label predictions and localizing the vulnerable code statements on the used real-word datasets over the baselines.

**RQ2**: Can our LEAP approach identify vulnerable code statements in vulnerable functions more accurately in a semi-supervised setting than in an unsupervised setting?

**Approach**. By using the multivariate Bernoulli distribution in the selection process, the ICVH and LEAP methods can be operated in the semi-supervised setting [Nguyen et al.(2021a)] (i.e., where we assume that the core vulnerable statements in a small proportion of functions are manually annotated) in addition to the unsupervised setting. We can leverage such ground-truth information by adding the maximization of a log-likelihood as an additional training objective:

$$
\max \left\{ \sum_{k \in I_c} \log p_k + \sum_{k \not\in I_c} \log(1 - p_k) \right\}
$$

where $I_c = [i_1, ..., i_m]$. We then add the above additional objective function to the final objective function (mentioned in Eq. (7)) with the trade-off parameter $\eta > 0$ (i.e., we set the value of $\eta$ in $\{10^{-3}, 10^{-2}, 10^{-1}\}$).
Here, we investigated the performance of our LEAP method in the semi-supervised setting compared with its performance in the unsupervised setting for highlighting vulnerable statements. We also compared the performance of our LEAP method with the ICVH method in the semi-supervised setting. We conducted these experiments on the CWE-119, CWE-399, and Big-Vul datasets. In the semi-supervised setting, we assume that there is a small portion of the training set (i.e., 10%) having the ground truth of vulnerable code statements.

Table 2. Performance results of the ICVH and LEAP methods with K=10 for the VCP, VCA, TopK ACC, and IFA measures on the testing set of the CWE-399, CWE-119, and Big-Vul datasets in the unsupervised and semi-supervised settings (with 10% of the training set having the ground truth of vulnerable code statements). In the semi-supervised setting, we denote our LEAP method as LEAP-S10 while the ICVH method is denoted as ICVH-S10 (the best performance is shown in bold).

| Dataset  | K  | Method                        | VCP  | VCA  | Top10 ACC | IFA |
|----------|----|-------------------------------|------|------|-----------|-----|
| CWE-119  | 10 | ICVH [Nguyen et al.(2021a)]  | 93.5%| 91.1%| 94.5%     | 2.2 |
|          |    | ICVH-S10 [Nguyen et al.(2021a)] | 99.4%| 99.3%| 100%      | 1.2 |
|          |    | LEAP                          | 97.5%| 96.5%| 97.6%     | 2.1 |
|          |    | LEAP-S10                      | 99.9%| 99.8%| 100%      | 1.9 |
| CWE-399  | 10 | ICVH [Nguyen et al.(2021a)]  | 84.5%| 77.0%| 81.0%     | 5.5 |
|          |    | ICVH-S10 [Nguyen et al.(2021a)] | 90.5%| 86.0%| 100%      | 5.0 |
|          |    | LEAP                          | 96.6%| 95.0%| 95.0%     | 2.4 |
|          |    | LEAP-S10                      | 99.3%| 99.0%| 99.0%     | 2.0 |
| Big-Vul  | 10 | ICVH [Nguyen et al.(2021a)]  | 73.8%| 69.6%| 76.8%     | 2.2 |
|          |    | ICVH-S10 [Nguyen et al.(2021a)] | 79.3%| 76.1%| 80.6%     | 3.5 |
|          |    | LEAP                          | 80.5%| 77.6%| 80.6%     | 1.5 |
|          |    | LEAP-S10                      | 82.9%| 80.6%| 83.6%     | 1.5 |

**Result.** The experimental results in Table 2 show that by using a small portion of data having the ground truth (i.e., 10%) of vulnerable code statements, the performance of our LEAP method in the semi-supervised setting significantly increased compared to its performance in the unsupervised setting for all the used datasets.

The results in Table 2 also show that the model’s performance of the ICVH method in the semi-supervised setting increased compared to its performance in the unsupervised setting. However, our LEAP method still obtained a higher performance on three used datasets (i.e., the CWE-119, CWE-399, and Big-Vul datasets) for most of the used metrics, especially for the VCP and VCA measures.

**In conclusion for RQ2:** The experimental results (in Table 2) show a considerable increase in the model’s performance of our LEAP method in the semi-supervised setting compared to itself in the unsupervised setting. We also observed an improvement in the model’s performance of the ICVH method. However, compared to ICVH, our proposed method still obtains a higher performance on the used datasets for most metrics, especially for the VCP and VCA measures.

**RQ3:** What are the contributions of the components of our LEAP approach (mutual information and our proposed cluster spatial contrastive learning)?

**Approach.** We investigated the performance of our proposed method in three different cases related to contrastive learning including i) using clustered spatial contrastive learning mentioned
in Eq. (6) (LEAP -with-CSCL), ii) using normal contrastive learning mentioned in Eq. (5) (LEAP -with-CL, and iii) not using contrastive learning (LEAP -without-CL) (i.e., in this case, we only use the mutual information for guiding the whole training process).

Result. The experimental results in Table 3 show that our method using clustered spatial contrastive learning obtains a much higher performance compared to its performance when using normal contrastive learning, especially compared to the case not using contrastive learning on the VCP, VCA, Top-10 ACC, and IFA measures. These results demonstrate the efficiency and superiority of our proposed clustered spatial contrastive learning in modeling the important properties for the relationship of vulnerable patterns between the source code sections in order to improve the selection process of $\tilde{F}$.

Table 3. Performance results on VCP, VCA, Top-10 ACC, and IFA measures for the testing set of the CWE-399, CWE-119, and Big-Vul datasets for our method in three different cases including LEAP -with-CSCL, LEAP -with-CL, and LEAP -without-CL with $K = 10$ (the best performance is shown in **bold**).

| Dataset   | K  | Method          | VCP  | VCA  | Top-10 ACC | IFA  |
|-----------|----|-----------------|------|------|------------|------|
| CWE-399   | 10 | LEAP -without-CL| 85.8%| 79.0%| 80.0%      | 4.5  |
|           |    | LEAP -with-CL   | 91.2%| 87.0%| 87.0%      | 3.5  |
|           |    | LEAP -with-CSCL | **96.6%**| **95.0%**| **95.0%**| **2.4**|
| CWE-119   | 10 | LEAP -without-CL| 92.1%| 89.2%| 93.6%      | 4.1  |
|           |    | LEAP -with-CL   | 94.5%| 92.3%| 94.2%      | 2.6  |
|           |    | LEAP -with-CSCL | **97.5%**| **96.5%**| **97.6%**| **2.1**|
| Big-Vul   | 10 | LEAP -without-CL| 74.3%| 70.1%| 76.1%      | 2.3  |
|           |    | LEAP -with-CL   | 75.9%| 70.6%| 76.5%      | 2.1  |
|           |    | LEAP -with-CSCL | **80.5%**| **77.6%**| **80.6%**| **1.5**|

**In conclusion for RQ3:** The experimental results (in Table 3) prove the effectiveness of our proposed cluster spatial contrastive learning described in Eq.(6) in boosting the data representation learning and in figuring out the hidden vulnerability pattern. In particular, our method using clustered spatial contrastive learning obtains a much higher performance compared to its performance when using normal contrastive learning, especially compared to the case of not using contrastive learning on the VCP, VCA, Top-10 ACC, and IFA measures.

6 DISCUSSION

6.1 Effect of the number of clusters

We investigated the correlation between the number of chosen clusters guiding the computation of the proposed clustered spatial contrastive learning term mentioned in Eq. (6) and the VCP and VCA measures for our LEAP method on the testing set of CWE-119, CWE-399, and Big-Vul datasets.

As shown in Figure 6, we observe that our LEAP method obtains a higher performance for the VCP and VCA measures 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. In 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 is lower than the case when the chosen cluster varies in \{3, 5, 7\} which can reflect more appropriate values of the true number of vulnerability patterns.
6.2 Parameter sensitivity

We investigated the correlation between the hyper-parameter $\alpha$ which represents the weight of the proposed clustered spatial contrastive learning term in the final objective function (7) and the VCP and VCA measures for our LEAP method on the testing set of the CWE-399 and CWE119 datasets in the unsupervised setting.

The results in Figure 7 show that we can obtain a better model’s performance when $\alpha$ varies in $\{10^0, 2 \times 10^0, 3 \times 10^0\}$, compared to the case in which $\alpha$ varies in $\{10^{-3}, 10^{-2}, 10^{-1}\}$. That indicates the importance of the clustered spatial contrastive learning term in the training process to model important properties for the relationship of vulnerable patterns between the source code sections. That helps boost the selection process of $\tilde{F}$ to improve the model’s performance significantly. Furthermore, the results in Figure 7 also indicate that we should set the value of the trade-off hyper-parameter $\alpha$ 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.
6.3 Explanatory capability of our proposed method

In order to demonstrate the ability of our proposed method in detecting the vulnerable code patterns and statements in the vulnerable functions to support security analysts and code developers, in this section, we show a visualization of the selected code statements for a vulnerable function in the unsupervised setting.

As shown in Figure 8, the two functions in the left-hand figure (a) share some similar flaws. These include (i) a potential flaw of reading data from the console, i.e., in the statement "if ( fgets ( var2 , var3 , stdin ) != NULL )" of the first function and in the statement fscanf ( var3 , str , & var1 ) ; of the second function, and (ii) a potential flaw due to no maximum limitation for memory allocation in the statements "if ( var1 > wcslen ( var9 ) )" and "if ( var1 > wcslen ( var5 ) )" of the first and second functions, respectively. The two functions on the right-hand figure (b) also share some similar vulnerabilities including (i) a potential flaw of reading data from an environment, i.e., in the statement "wcsncat ( var1 + var2 , var3 , 100 - var2 - 1 ) ;" of the first function and in the statement "if ( fgetws ( var1 + var2 , ( int ) ( 100 - var2 ) , var3 ) != var4 )" of the second function, and (ii) a flaw of incrementing the pointer in the loop which will cause us to free the memory block not at the start of the buffer in the statement "for ( ; * var1 != str ; var1 ++ )" of both these functions.

In these functions, the selected code statements from the model relevant to vulnerabilities are shown with \( K = 5 \) (i.e., the green background lines) from our LEAP method. Note that the selected statements in each function with the red border specify the core vulnerable statements obtained from the ground truth and also detected by our method. Two functions on the left-hand figure (a) consist of two similar potential flaws which are "reading data from the console" and "no maximum limitation for memory allocation". Two functions on the right-hand figure (b) share a similar potential flaw and flaw including "reading data from an environment" and "incrementing the pointer in the loop", respectively. For the demonstration purpose and simplicity, we choose simple and short vulnerable source code functions in which some statements are omitted and replaced by "... " for brevity.

In these functions, the selected code statements from the model relevant to vulnerabilities are shown with \( K = 5 \). The green background lines in each function highlight the detected code statements while the green background lines with the red border specify the core vulnerable statements obtained from the ground truth, and these lines are also detected by our method. Our LEAP method with \( K = 5 \) can detect all of these potentially vulnerable code statements that make the corresponding functions vulnerable. It is worth noting that to the source code sections having similar vulnerability patterns, the subset of vulnerability-relevant selected statements (i.e., \( F_{\text{top}} \)) from our LEAP method are quite similar. That further demonstrates...
the ability of our method in detecting vulnerable code patterns and statements in the vulnerable functions.

6.4 Additional experiments

**Auxiliary measure.** As mentioned in the experiments section of the paper, the main purpose of our LEAP method is to support security analysts and code 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=10) so that with a high probability those lines cover most or all vulnerable statements. Bearing this incentive, and inspired by [Nguyen et al.(2021a)], to evaluate the performance of our proposed method and baselines, we use two main measures introduced in [Nguyen et al.(2021a)] 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). We also apply using two other measures including Top-10 Accuracy (i.e., it measures the percentage of vulnerable functions where at least one actual vulnerable lines appear in the top-10 ranking) and Initial False Alarm (IFA) (i.e., it measures the number of incorrectly predicted lines (i.e., non-vulnerable lines incorrectly predicted as vulnerable or false alarms) that security analysts need to inspect until finding the first actual vulnerable line for a given function) used in [Fu and Tantithamthavorn(2022)].

In practice, security analysts and code developers can set their preferable top-K for the used methods, so 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.

We computed the auxiliary VCE measure for our LEAP method and baselines (i.e., L2X, ICVH, and LineVul) with top-K=5 in the unsupervised setting. The experimental results mentioned in Table 4 show that our LEAP method obtained the highest VCE measure in both CWE-399 and CWE-119 datasets compared to the baselines. In particular, to the CWE-399 dataset, our LEAP method gained 45.1% for the auxiliary VCE measure. It means that in this case, we can detect 5×0.451 = 2.255 vulnerable statements out of 5 spotted lines.

Table 4. Experimental results in terms of the auxiliary VCE measure on the testing set of the CWE-399 and CWE-119 datasets for the L2X, ICVH, LineVul, and LEAP methods with K = 5 in the unsupervised setting (the best performance is shown in **bold**).

| Dataset  | K   | Method                                      | VCE   |
|----------|-----|---------------------------------------------|-------|
| CWE-399  | 5   | L2X [Chen et al.(2018)]                    | 43.6% |
|          |     | ICVH [Nguyen et al.(2021a)]                | 39.1% |
|          |     | LineVul [Fu and Tantithamthavorn(2022)]    | 29.2% |
|          |     | LEAP (ours)                                | **45.1%** |
| CWE-119  | 5   | L2X [Chen et al.(2018)]                    | 30.5% |
|          |     | ICVH [Nguyen et al.(2021a)]                | 26.4% |
|          |     | LineVul [Fu and Tantithamthavorn(2022)]    | 23.9% |
|          |     | LEAP (ours)                                | **31.1%** |
6.5 Threats to Validity

**Construct Validity.** Key construct validity threats are if our assessments of the methods demonstrate the ability for detecting vulnerable code patterns and statements in the vulnerable functions in unsupervised and semi-supervised settings. The main purpose of our LEAP method is to support security analysts and code 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 in functions or programs. To evaluate the performance of our LEAP method and baselines, we use four main measures including: vulnerability coverage proportion (VCP), vulnerability coverage accuracy (VCA), Top-10 Accuracy, and Initial False Alarm (IFA) as mentioned in Section 4.4.

**Internal validity.** Key internal validity threats are relevant to the choice of hyper-parameter settings (i.e., optimizer, learning rate, number of layers in deep neural networks, etc.) as described in Section 4.6. It is worth noting that finding a set of optimal hyperparameter settings of deep neural networks is expensive due to a large number of trainable parameters. To train our method, we only use the common and default values of hyper-parameters such as using Adam optimizer and the learning rate in \( \{10^{-3}, 10^{-4}\}\). We also report the hyperparameter settings in the released reproducible source code to support future replication studies.

**External validity.** Key external validity threats include whether our proposed method will generalize to other vulnerabilities and whether they will work on other source code datasets. We mitigated this problem by using big and common but different vulnerabilities on three real-world source code datasets, namely, the CWE-399, CWE-199, and Big-Vul datasets.

7 RELATED WORK

Deep learning has been applied successfully to source code and binary software vulnerability detection (SVD) [Dam et al.(2017), Lin et al.(2018), Li et al.(2018b), Le et al.(2019), Zhuang et al.(2020), Nguyen et al.(2020c), Nguyen et al.(2020b), Li et al.(2021b), Nguyen et al.(2021b), Lou et al.(2021), Nguyen et al.(2022), Fu et al.(2022), Fu et al.(2023a), Fu et al.(2024)]. However, most of the current approaches only detect vulnerabilities at either the function or program level, not at the more fine-grained code statement level. From the interpretable machine-learning perspective, it seems that the existing methods [Ribeiro et al.(2016), Shrikumar et al.(2017), Lundberg and Lee(2017), Chen et al.(2018)] with adaptations can be ready to apply. However, besides L2X [Chen et al.(2018)], none of the others is applicable to the context of statement-level vulnerability detection.

Recently, there have been several approaches [Li et al.(2020), Nguyen et al.(2021a), Li et al.(2021a), Fu and Tantithamthavorn(2022)] proposed to solve the statement-level SVD problem. In particular, [Li et al.(2020)] proposed VulDeeLocator, a deep learning-based method, requiring compiling the source code to Lower Level Virtual Machine code (this method cannot be used if a function cannot be compiled) and the information relevant to vulnerable code statements (the method cannot work in the unsupervised setting). Hoppity [Dinella et al.(2020)] is a learning-based approach that uses a graph neural network (GNN) [Scarselli et al.(2009)] to detect (and fix bugs) at the token level in Javascript programs. However, the bug detection part in Hoppity needs to use the information relevant to the vulnerable code tokens and cannot work in the unsupervised setting.

[Nguyen et al.(2021a)] proposed ICVH based on mutual information and used it as an explaining model to explain the reference model (i.e., the learning model approximating the true conditional distribution \( p(Y \mid F) \)). [Li et al.(2021a)] introduced the IVDetect method using a Feature-attention Graph Convolution Network approach to predict function level vulnerabilities and a GNNEXplainer to identify which sub-graph contributed the most to the predictions to locate the fine-grained location of vulnerabilities. However, such sub-graphs still contain many lines of code.
[Fu and Tantithamthavorn(2022)] proposed LineVul based on BERT [Devlin et al.(2018)] to locate vulnerable lines via attention weights. Recently, [Hin et al.(2022)] propose a deep learning framework, named LineVD, using the graph attention network (GAT) model [Velickovic et al.(2018)], formulating statement-level vulnerability detection as a node classification task for the statement level vulnerability detection. [Ding et al.(2022)] introduce an ensemble learning approach, named VELVET, to locate vulnerable statements. In particular, the model combines graph-based and sequence-based neural networks to capture the context of a program graph and understand code semantics and vulnerable patterns. While ICVH [Nguyen et al.(2021a)] and LineVul [Fu and Tantithamthavorn(2022)] can be operated in the unsupervised setting, LineVD and VELVET need to use the ground truth of vulnerable code statements during the training process via the supervised setting.

8 CONCLUSION

We have proposed a novel end-to-end deep learning-based approach for tackling the statement-level source code vulnerability detection problem. In particular, we first leverage mutual information 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 then proposed novel clustered spatial contrastive learning in order to further improve the representation learning and the robust selection process $\epsilon$. Specifically, our novel clustered spatial contrastive learning guides the selection process $\epsilon$ to select the 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 similarly selected vulnerability-relevant code statements. Our experimental results on three real-world datasets show the superiority of our LEAP method over other state-of-the-art baselines in detecting the vulnerable code statements in source code functions in both the unsupervised and semi-supervised settings.

REFERENCES

[Abadi et al.(2016)] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). 265–283.

[Almorsy et al.(2012)] M. Almorsy, J.C. Grundy, and A. Ibrahim. 2012. Supporting Automated Vulnerability Analysis Using Formalized Vulnerability Signatures. In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012). 100–109.

[Chen et al.(2018)] J. Chen, L. Song, M. J. Wainwright, and M. I. Jordan. 2018. Learning to Explain: An Information-Theoretic Perspective on Model Interpretation. CoRR abs/1802.07814 (2018).

[Cheng et al.(2019)] Xiao Cheng, Haoyu Wang, Jiayi Hua, Miao Zhang, Guoai Xu, Li Yi, and Yulei Sui. 2019. Static Detection of Control-Flow-Related Vulnerabilities Using Graph Embedding. In 2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS).

[Cover and Thomas(2006)] Thomas M. Cover and Joy A. Thomas. 2006. Elements of Information Theory. John Wiley and Sons, Inc.

[Dam et al.(2017)] Hoa K. Dam, Truyen Tran, Trang Pham, Shien W. Ng, John Grundy, and Aditya Ghose. 2017. Automatic feature learning for vulnerability prediction. CoRR abs/1708.02368 (2017).

[Dam et al.(2018)] Hoa K. Dam, Truyen Tran, Trang Pham, Shien W. Ng, John Grundy, and Aditya Ghose. 2018. Automatic feature learning for predicting vulnerable software components. IEEE Transactions on Software Engineering (2018).

[Devlin et al.(2018)] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. CoRR abs/1810.04805 (2018).

[Dimella et al.(2020)] Elizabeth Dimella, Hanjun Dai, Zi Yang Li, Mayur Naik, Le Song, and Ke Wang. 2020. Hoppity: learning graph transformations to detect and fix bugs in programs. In International Conference on Learning Representations.

[Ding et al.(2022)] Yangruibo Ding, Sahil Suneja, Yunhui Zheng, Jim Laredo, Alessandro Morari, Gail Kaiser, and Baishakhi Ray. 2022. VELVET: a noVel Ensemble Learning approach to automatically locate VulnErable sTatements. In 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). 959–970. https://doi.org/10.1109/SANER53432.2022.00114
[Dowd et al.(2006)] Mark Dowd, John McDonald, and Justin Schuh. 2006. The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities. Addison-Wesley Professional.

[Duan et al.(2019)] Xu Duan, Jingzheng Wu, Shouling Ji, Zhiqing Rui, Tianyue Luo, Mutian Yang, and Yanjun Wu. 2019. VulSniper: Focus Your Attention to Shoot Fine-Grained Vulnerabilities. In Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19. 4665–4671.

[Fan et al.(2020)] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. The 17th International Conference on Mining Software Repositories (2020).

[Feng et al.(2020)] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaoacheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. CoRR abs/2002.08155 (2020).

[Fu et al.(2024)] Michael Fu, Van Nguyen, Chakkrit Tantithamthavorn, Dinh Phung, and Trung Le. 2024. Vision transformer inspired automated vulnerability repair. ACM Transactions on Software Engineering and Methodology 33, 3 (2024), 1–29.

[Fu et al.(2023a)] Michael Fu, Van Nguyen, Chakkrit Kla Tantithamthavorn, Trung Le, and Dinh Phung. 2023a. VulExplainer: A Transformer-based Hierarchical Distillation for Explaining Vulnerability Types. IEEE Transactions on Software Engineering (2023).

[Fu and Tantithamthavorn(2022)] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: A Transformer-based Line-Level Vulnerability Prediction. In the International Conference on Mining Software Repositories (MSR).

[Fu et al.(2022)] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. 2022. VulRepair: a T5-based automated software vulnerability repair. In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 935–947.

[Fu et al.(2023b)] Michael Fu, Chakkrit Kla Tantithamthavorn, Van Nguyen, and Trung Le. 2023b. Chatgpt for vulnerability detection, classification, and repair: How far are we? In 2023 30th Asia-Pacific Software Engineering Conference (APSEC). IEEE, 632–636.

[Ghaffarian and Shahriari(2017)] Seyed M. Ghaffarian and Hamid R. Shahriari. 2017. Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey. ACM Computing Surveys (CSUR) 50, 4 (2017), 1–36.

[Grieco et al.(2016)] Gustavo Grieco, Guillermo Luis Grinblat, Lucas Uzal, Sanjay Rawat, Josselin Feist, and Laurent Mounier. 2016. Toward Large-Scale Vulnerability Discovery Using Machine Learning. In Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy (New Orleans, Louisiana, USA) (CODASPY ’16). 85–96.

[Hin et al.(2022)] David Hin, Andrey Kan, Huaming Chen, and M. Ali Babar. 2022. LineVD: Statement-level Vulnerability Detection using Graph Neural Networks. https://doi.org/10.48550/ARXIV.2203.05181

[Jang et al.(2016)] E. Jang, S. Gu, and B. Poole. 2016. Categorical reparameterization with gumbel-softmax. arXiv preprint arXiv:1611.01144 (2016).

[Khosla et al.(2020)] Prannay Khosla, Piotr Teterwak, Chen Wang, Aaron Sarna, Yonglong Tian, Phillip Isola, Aaron Maschinot, Ce Liu, and Dilip Krishnan. 2020. Supervised Contrastive Learning. CoRR abs/2004.11362 (2020).

[Kim et al.(2017)] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In IEEE Symposium on Security and Privacy. IEEE Computer Society, 595–614.

[Kingma and Ba(2014)] D. P. Kingma and J. Ba. 2014. Adam: A Method for Stochastic Optimization. CoRR abs/1412.6980 (2014).

[Le et al.(2019)] Tue Le, Tuan Nguyen, Trung Le, Dinh Phung, Paul Montague, Olivier De Vel, and Lizhen Qu. 2019. Maximal Divergence Sequential Autoencoder for Binary Software Vulnerability Detection. In In International Conference on Learning Representations.

[Li et al.(2021a)] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021a. Vulnerability Detection with Fine-grained Interpretations. CoRR abs/2106.10478 (2021).

[Li et al.(2020)] Zhen Li, Deqing Zou, Shouhui Xu, Zhaoxuan Chen, Yawei Zhu, and Hai Jin. 2020. VulDeeLocator: A Deep Learning-based Fine-grained Vulnerability Detector. arXiv preprint arXiv:2001.02350 (2020).

[Li et al.(2016)] Zhen Li, Deqing Zou, Shouhui Xu, Hai Jin, Hanchao Qi, and Jie Hu. 2016. VulPecker: An Automated Vulnerability Detection System Based on Code Similarity Analysis. In Proceedings of the 32Nd Annual Conference on Computer Security Applications (Los Angeles, California, USA) (ACSCA ’16). 201–213.

[Li et al.(2021b)] Zhen Li, Deqing Zou, Shouhui Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. 2021b. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. IEEE Transactions on Dependable and Secure Computing (2021).

[Li et al.(2018a)] Zhen Li, Deqing Zou, Shouhui Xu, Hai Jin, Yawei Zhu, Zhaoxuan Chen, Sujuan Wang, and Jialai Wang. 2018a. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. CoRR abs/1807.06756 (2018).

[Li et al.(2018b)] Zhen Li, Deqing Zou, Shouhui Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018b. VulDeePecker: A Deep Learning-Based System for Vulnerability Detection. CoRR abs/1801.01681 (2018).
Statement-Level Vulnerability Detection Through Information Theory and Contrastive Learning

[Lin et al.(2018)] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, Yan Zhang, Olivier De Vel, and Paul Montague. 2018. Cross-Project Transfer Representation Learning for Vulnerable Function Discovery. In *IEEE Transactions on Industrial Informatics*, Vol. 14.

[Lou et al.(2021)] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

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

[McAfee and CSIS(2020)] McAfee and CSIS. 2020. Latest Report from McAfee and CSIS Uncovers the Hidden Costs of Cybercrime Beyond Economic Impact. (2020).

[Nguyen et al.(2020a)] Van Nguyen, Trung Le, Olivier De Vel, Paul Montague, John Grundy, and Dinh Phung. 2020a. Dual-Component Deep Domain Adaptation: A New Approach for Cross Project Software Vulnerability Detection. *Pacific-Asia Conference on Knowledge Discovery and Data Mining* (2020).

[Nguyen et al.(2021a)] Van Nguyen, Trung Le, Olivier de Vel, Paul Montague, John Grundy, and Dinh Phung. 2021a. Information-theoretic Source Code Vulnerability Highlighting. In *International Joint Conference on Neural Networks (IJCNN)*.

[Nguyen et al.(2020c)] Van Nguyen, Trung Le, Tue Le, Khanh Nguyen, Olivier de Vel, Paul Montague, John Grundy, and Dinh Phung. 2020c. Code action network for binary function scope identification. *Advances in Knowledge Discovery and Data Mining (PAKDD)* (2020).

[Nguyen et al.(2020b)] Van Nguyen, Trung Le, Tue Le, Khanh Nguyen, Olivier de Vel, Paul Montague, and Dinh Phung. 2020b. Code Pointer Network for Binary Function Scope Identification. *International Joint Conference on Neural Networks (IJCNN)* (2020).

[Nguyen et al.(2019)] Van Nguyen, Trung Le, Tue Le, Khanh Nguyen, Olivier DeVel, Paul Montague, Lizhen Qu, and Dinh Phung. 2019. Deep Domain Adaptation for Vulnerable Code Function Identification. In *The International Joint Conference on Neural Networks (IJCNN)*.

[Nguyen et al.(2022)] Van Nguyen, Trung Le, Chakkrit Tantithamthavorn, John Grundy, Hung Nguyen, and Dinh Phung. 2022. Cross Project Software Vulnerability Detection via Domain Adaptation and Max-Margin Principle. (2022). arXiv:2209.10406 [cs.CR]

[Scarselli et al.(2009)] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. The Graph Neural Network Model. *IEEE Transactions on Neural Networks (TNN)* (2009).

[Shannon(1998)] Claude Elwood Shannon. 1998. *The mathematical theory of communication*. Warren Weaver. University of Illinois Press, Urbana.

[Shin et al.(2011)] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A. Osborne. 2011. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering* 37, 6 (2011), 772–787.

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

[Srivastava et al.(2014)] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research* 15 (2014), 1929–1958.

[Veličković et al.(2018)] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rJXMpikCZ

[Yamaguchi et al.(2011)] Fabian Yamaguchi, Felix Lindner, and Konrad Rieck. 2011. Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning. In *Proceedings of the 5th USENIX conference on Offensive technologies*. 13–23.
[Zhuang et al.(2020)] Yuan Zhuang, Zhenguang Liu, Peng Qian, Qi Liu, Xiang Wang, and Qinming He. 2020. Smart Contract Vulnerability Detection using Graph Neural Network. In Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20. 3283–3290.

[Zimmermann et al.(2009)] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. 2009. 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 (Amsterdam, The Netherlands) (ESEC/FSE ’09). 91–100.