2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)

Are We Learning the Right Features? A Framework for Evaluating DL-Based Software Vulnerability Detection Solutions

Satyaki Das, Syeda Tasnim Fabiha, Saad Shafiq, and Nenad Medvidović

University of Southern California
{satyakid, fabiha, sshafiq, neno}@usc.edu

***Abstract*—Recent research has revealed that the reported results of an emerging body of deep learning-based techniques for detecting software vulnerabilities are not reproducible, either across different datasets or on unseen samples. This paper aims to provide the foundation for properly evaluating the research in this domain. We do so by analyzing prior work and existing vulnerability datasets for the syntactic and semantic features of code that contribute to vulnerability, as well as features that falsely correlate with vulnerability. We provide a novel, uniform representation to capture both sets of features, and use this representation to detect the presence of both vulnerability and spurious features in code. To this end, we design two types of code perturbations: feature preserving perturbations (FPP) ensure that the vulnerability feature remains in a given code sample, while feature eliminating perturbations (FEP) eliminate the feature from the code sample. These perturbations aim to measure the influence of spurious and vulnerability features on the predictions of a given vulnerability detection solution. To evaluate how the two classes of perturbations influence predictions, we conducted a large-scale empirical study on five state-of-the-art DL-based vulnerability detectors. Our study shows that, for vulnerability features, only ~2% of FPPs yield the undesirable effect of a prediction changing among the five detectors on average. However, on average, ~84% of FEPs yield the undesirable effect of retaining the vulnerability predictions. For spurious features, we observed that FPPs yielded a drop in recall up to 29% for graph-based detectors. We present the reasons underlying these results and suggest strategies for improving DNN-based vulnerability detectors. We provide our perturbation-based evaluation framework as a public resource to enable independent future evaluation of vulnerability detectors.**

***Index Terms*—vulnerability detection, deep learning, software security, explainable AI**

I. Introduction

Identifying security vulnerabilities in software, and specifically in source code, has been an important focus of researchers and practitioners, prompted by numerous examples of high-profile security breaches [1]–[6]. Earlier research in this area concentrated on developing deterministic approaches for vulnerability detection that relied on predefined rules and patterns [7]–[10]. Since these approaches have suffered from a range of shortcomings [1], researchers have more recently turned to deep learning as a vehicle for vulnerability detection because DL offers a superior capacity to learn complex patterns from data [1], [11]–[14]. DL techniques have demonstrated their versatility in other software engineering tasks that involve source code datasets, such as code clone detection and authorship attribution [15], providing additional motivation for their use in software vulnerability detection.

Although the shift to DL has yielded promising results, it has also introduced new challenges. Specifically, these techniques operate as black boxes, making it difficult to understand the reasoning behind their predictions and decisions. They also suffer from a lack of generalizability [13], [16], performing poorly on unseen datasets and failing to adapt to new vulnerabilities. It is thus important to make these techniques more explainable, and this can be achieved by investigating the specific code features that influence their predictions. By doing so, we can uncover the underlying decision-making processes, expose potential biases and limitations, and pinpoint areas for refinement, ultimately leading to the development of more trustworthy, reliable, and effective vulnerability detection techniques.

On this front, existing literature has recognized the presence of spurious features in DL-based approaches [13], [17], [18]. These are code features that falsely correlate with the target label. Such spurious features can impact vulnerability detection tools and models, and they provide a helpful starting point for our work. However, to systematically advance the state-of-the-art in vulnerability detection, a three-pronged approach is necessary: (1) identify and disregard spurious features (*SF*) in code that can lead to inaccuracies; (2) pinpoint and leverage genuine features that contribute to vulnerabilities (*VF*); and (3) analyze and quantify the impact of *SF* and *VF* on a proposed vulnerability detection technique.

This paper presents our implementation of the above three-pronged approach. First, we conducted a rigorous analysis of the widely used SARD vulnerability dataset [19] to uncover the key features and patterns that contribute to the manifestation of the vulnerabilities in the dataset *VF*. Second, we expanded the list of *SF*s by exploring the assumptions made in the literature (e.g., those that do not hold true in our dataset samples). We have systematized the uncovered *VF* and *SF* and structured them into an expandable taxonomy of code features for vulnerability detection.

Third, we have developed *VIPer*, a novel perturbation-based approach for identifying the weaknesses in a given vulnerability detector's predictions. *VIPer* generates both feature preserving perturbations (FPP), which ensure that a feature (*VF* or *SF*) remains in a given code sample, and feature elim-

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00194

inating perturbations (FEP), which remove the feature from the code sample. *VIPer* comprises three phases: (1) *Feature detection* identifies the presence or absence of each feature from our taxonomy in a given source code sample. (2) *Targeted perturbation* modifies the code sample in a manner that either preserves (FPP) or removes (FEP) a detected feature. (3) *Solution evaluation* involve analyzing a vulnerability detector's response to the targeted perturbations and inferring the extent to which a given *VF* or *SF* impacts the detector's prediction.

We have applied *VIPer* on five state-of-the-art DL-based vulnerability detectors: DeepWukong [12], ReVeal [13], DeepDFA [20], LineVul [14], and SySeVR [11]. By analyzing the five detectors' responses to *VIPer*'s perturbations, we quantified the extent to which a given feature contributes to a prediction, thus providing valuable insights into the detectors' decision-making processes, the sensitivity and robustness of their predictions, and the potential biases and limitations of their vulnerability detection capabilities. Our findings indicate that, in case of *VF*s *VIPer*'s perturbations significantly impact the five detectors' performance, with precision decreasing by ~28% and recall by ~8%, on average. The detectors exhibit reasonable robustness to FPPs, with only ~2% of all FPPs yielding the inappropriate outcome of changed vulnerability predictions. However, ~84% of FEPs result in the inappropriate outcome of retained vulnerability predictions. Additionally, in the case of *SF*s, FPPs produce a decline in recall of up to 29%. Together, the latter two results mean that, in an overwhelming majority of cases, the state-of-the-art vulnerability detectors' original reasoning behind predictions was flawed as it was not actually based on the targeted features.

This paper makes the following contributions:

- an extendable taxonomy of vulnerability (*VF*) and spurious (*SF*) features;
- *VIPer*, a perturbation-based framework [21] to gauge the robustness of vulnerability detectors;
- a customizable wrapper for seamless integration of the framework in the evaluation pipeline of existing vulnerability detectors; and
- a comprehensive empirical evaluation of five state-of-the-art vulnerability detectors, assessing the impact of both *VF* and *SF* on their predictions.

In the paper's remainder, Section II introduces the novel taxonomy of code features. Section III details our approach, *VIPer*. Sections IV and V present the evaluation setup and results of our study. Section VI discusses our findings and their implications. Threats to validity are discussed in Section VII, related work in Section VIII, and conclusions in Section IX.

II. Taxonomy of Code Features

We initiated our study by examining which code features a given detector learns, by analyzing the widely-adopted vulnerability datasets: SARD [19], FFmpeg+Qemu [22], Draper [23], and BigVul [24]. All four datasets were instructive in our understanding of the problem and its different manifestations. However, only SARD provided annotations at critical points in the source code that describe how a vulnerability manifests itself in the vulnerable sample (e.g., see comment prefixed with "FLAW" on line 3 in Listing 1) and what changes one can apply to repair it in the corresponding non-vulnerable sample ("FIX" on line 3 in Listing 2). The absence of this information in other datasets makes it difficult, both, to identify *VF*s and to assess the accuracy of *VF* detection in samples. Along with the fact that SARD is the largest publicly available vulnerability dataset, containing many real-world security flaws (e.g., from Wireshark and GIMP) and actively supported by the National Institute of Standards and Technology [25], this led us to direct our focus to the SARD dataset. We will now delve into the process of identifying vulnerability (*VF*) and spurious (*SF*) features, followed by the development of the taxonomy.

```
1 int * data;
2 data = NULL;
3 /* FLAW: Allocate memory
      without using sizeof(int
      ) */
4 data = (int *)ALLOCA(10);
5 int source[10] = {0};
6 /* POTENTIAL FLAW: Possible
      buffer overflow if data
      was not allocated
      correctly in the source
      */
7 memcpy(data, source,
      10*sizeof(int));
8 printIntLine(data[0]);
```

Listing 1: Vul. Sample

```
1 int * data;
2 data = NULL;
3 /* FIX: Allocate memory
      using sizeof(int) */
4 data = (int *)ALLOCA(
      10*sizeof(int));
5 int source[10] = {0};
6 /* POTENTIAL FLAW: Possible
      buffer overflow if data
      was not allocated
      correctly in the source
      */
7 memcpy(data, source,
      10*sizeof(int));
8 printIntLine(data[0]);
```

Listing 2: Non-Vul. Sample

A. Identifying Vulnerability Features (VF)

We analyzed the annotated descriptions in the SARD dataset to identify properties of code that contribute to a vulnerability. Specifically, we focused on the 10 most frequent vulnerability categories, referred to as CWEs, out of 113 CWEs present in the SARD dataset. These 10 CWEs featured in 6,525 out of SARD's 22,080 source code files, as shown in Table I. We used as our cut-off point the fact that no other CWEs featured in at least 100 SARD files.

For each vulnerability in this set, we manually analyzed its annotated descriptions (comments containing prefixes "FLAW", "POTENTIAL FLAW", or "FIX", such as those in Listings 1 and 2) and corresponding source code segments. To systematically map the annotated descriptions with the corresponding features in the code, we represent each code sample in a code property graph (CPG) [26]. A CPG is

TABLE I: Top 10 CWEs in the SARD dataset

CWE ID	Description	# Files
CWE805	Buffer Access with Incorrect Length Value	1506
CWE806	Buffer Access Using Size of Source Buffer	1037
CWE124	Buffer Underwrite	907
CWE127	Buffer Under-read	784
CWE193	Off-by-one Error	748
CWE126	Buffer Over-read	550
CWE415	Double Free	421
CWE839	Numeric Range Comparison Without Minimum Check	314
CWE131	Incorrect Calculation of Buffer Size	129
CWE416	Use After Free	129

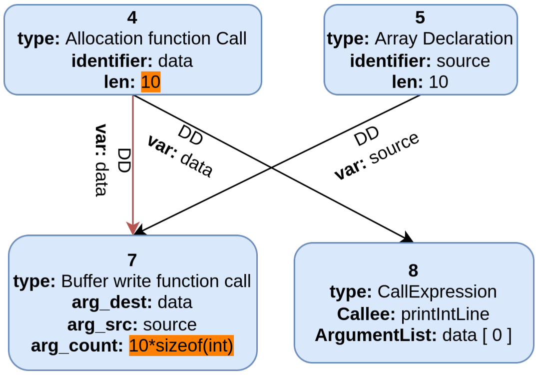

Fig. 1: Abridged CPG constructed from sample in Listing 1

constructed from a program's abstract syntax tree (AST), control flow graph (CFG), and program dependence graph (PDG), combining their labeling and property functions. This combination allows a CPG to leverage information from all three sources to conduct better vulnerability analysis [26].

For example, Fig. 1 depicts an abridged version of the CPG that is constructed from the sample in Listing 1, where each node represents properties that characterize the statement in the corresponding line of code, and each edge represents data, control, and call dependencies between the nodes. The annotated descriptions in lines 3 and 6 in Listing 1 describe the vulnerability present in lines 4 and 7, respectively. They are encoded in the CPG by the leftmost data dependence (DD) edge (in red) for the buffer data between nodes 4 and 7, illustrating an overflow scenario by performing write operation (10*sizeof(int) bytes) on a 10-byte buffer. We encoded the description of each of the 10 selected CWE vulnerability categories into a CPG. We elaborate on these rules in Section III.

B. Identifying Spurious Features (SF)

*SF*s required a different approach since all vulnerability datasets focus on features relevant to vulnerabilities and not on those that should be avoided. As our starting point, we used several existing studies, which provide valuable insights into how features such as variable and method names [17] and formatting tokens [18] falsely correlate with vulnerabilities. These features are examples of *SF*s we aim to study, and they can have an especially negative impact on the robustness and performance of *token-based* vulnerability detectors, such as SySeVR [11] and LineVul [14].

To address this, researchers have recently developed *graph-based* vulnerability detectors, such as ReVeal [13] and DeepWukong [12], which replace these *SF*s with symbolic names (e.g., VAR1, FUN1) and incorporate additional information from program graphs (e.g., control and data flow, call dependencies, etc.) to make predictions. This also means that the *SF*s observed in existing literature [17], [18] for token-based detectors do not apply to the graph-based detectors, requiring further exploration of *SF*s that may influence the latter.

To this end, we focused on the assumptions made by graph-based detectors [12], [13] regarding features that may contribute to a vulnerability. For instance, one common assumption is that a vulnerability is defined strictly by the set of nodes in the graph of the vulnerable sample. However, we observe that the same vulnerability would still exist if a mock node (e.g., a printf("Benign") statement) is added to the graph. Another common assumption is that the set of edges strictly defines the vulnerability in the graph of the vulnerable sample. However, a mock edge (e.g., if (5!=5) return;) can be added to the graph without affecting its vulnerability. The idea of introducing changes that should not impact vulnerability to the sets of nodes and edges was inspired by the existing literature that discussed how graph neural networks learn spurious correlations between sets of nodes and edges [27]. Our examination of the SARD dataset revealed that the above two assumptions in particular do not always hold and can lead to spurious correlations that impact detectors' predictions. We demonstrate the extent of this impact in Section V.

C. Developing the Taxonomy

To capture the dichotomy between *VF*s and *SF*s, and to provide a comprehensive understanding of their characteristics, we classified them into a taxonomy of *Code Features*. Our hierarchy-based taxonomy [28] is shown in Fig. 2. The taxonomy is not intended to be comprehensive. We expect that follow-on work will add further categories to the taxonomy, which will in turn be encoded as further *VIPer* rules.

We partitioned *Code Features* into two mutually exclusive sub-classes *Vulnerability Features* (*VF*s) and *Spurious Features* (*SF*s). All 10 identified *VF*s from Table I are classified

- Code Features
  - Vulnerability Features
    - Overflow
      - Incorrect Calculation of Buffer Size
      - Buffer Access Using Size of Source Buffer
      - Buffer Access Using Size of Destination Buffer
      - Off-by-one Error
    - Deallocated Buffer Use
      - Use-After-Free
      - Double-Free
    - Numeric Range Comparison Without Minimum Check
      - Buffer Underwrite
      - Buffer Under-read
    - Sensitive API Call
      - Read API
      - Write API
  - Spurious Features
    - Token-based
      - Identifier Names
      - Code Formatting
    - Graph-based
      - Node Set
        - Print instructions without variable
        - Unused variable declarations
      - Edge Set
        - Unreachable conditional operation without variables
        - Dependency altering uses of variables

Fig. 2: Taxonomy of Code Features

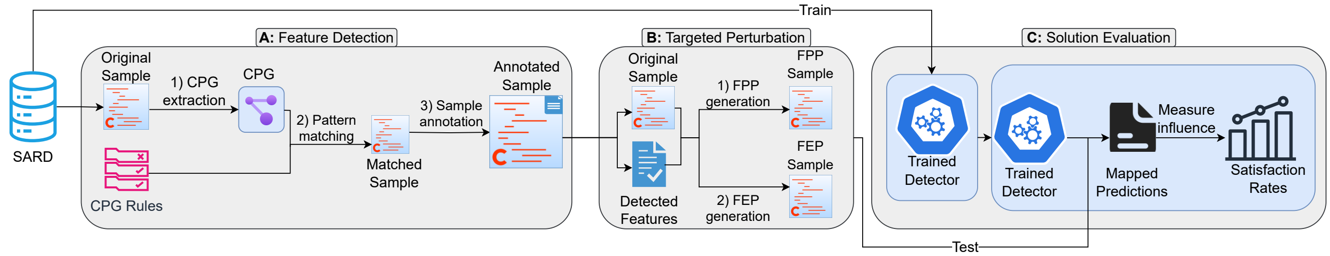

Fig. 3: High-level *VIPer* workflow

under the *Vulnerability Features* sub-class, which is subdivided into four further sub-classes based on feature characteristics:

1) *Overflow* vulnerabilities occur when a buffer is accessed outside its allocated memory. We further categorize overflow vulnerabilities based on the nature of the out-of-bounds access: (a) Incorrect Calculation of Buffer Size, (b) Buffer Access Using Size of Source Buffer, (c) Buffer Access Using Size of Destination Buffer, and (d) Off-by-one Error.
2) *Deallocated Buffer Use* vulnerabilities arise when a buffer is accessed after it has been deallocated. We further categorize these vulnerabilities based on the type of post-deallocation use: (a) Use-After-Free and (b) Double-Free.
3) *Numeric Range Comparison* vulnerabilities involve an array indexing operation where the index variable never gets checked for a minimum value. Based on the type of array indexing operation (i.e., read or write), we subdivide these vulnerabilities into (a) Buffer Underwrite and (b) Buffer Under-read.
4) *Sensitive API Call* vulnerabilities are characterized by the use of system APIs that are well-known for causing vulnerabilities [1], [29], [30]. We categorize these vulnerabilities into (a) Read APIs and (b) Write APIs.

We categorized the *Spurious Features* based on the two primary approaches for DL-based vulnerability detection:

1) *Token-based SF*s comprise features of individual code tokens that may lead to spurious correlations, such as (a) Identifier Names that may not be relevant to the vulnerability and (b) Code Formatting. *SF*s previously identified in literature [17], [18] fall under this category (grayed out in Fig. 2).
2) *Graph-based SF*s are features of nodes and their relationships (edges) that exhibit spurious correlations but do not inherently contribute to vulnerabilities. We categorize these features into (a) *Node Set* and (b) *Edge Set*. Node Set refers to the presence of one or more single-line instructions that have no impact on the vulnerability. Specific instances of these *SF*s include (i) *Print instructions without variables*, i.e., "empty" print statements in the code, and (ii) *Unused variable declarations*, i.e., variables that are not used after declaration. Edge Set refers to the presence of multi-line instructions defining data or control dependencies that do not affect vulnerability. Specific instances of Edge Set vulnerabilities are (i) *Unreachable conditional operation without variables*, i.e., a condition that will always be false and may change the control dependency, and (ii) *Dependency-altering uses of variables*, i.e., usage of variables that may alter dataflow or control dependency but that do not contribute to vulnerability. Note that the goal of the study reported in this paper is to observe the impact of *SF* categories that require minimal change. For this reason, we exclude the *SF*s that are greyed out in Fig. 2 from this study.

The transition from individual structured annotations in the SARD dataset to the development of the taxonomy was carried out iteratively. We analyzed the annotated descriptions of each sample containing VFs. During each iteration, an author suggested a category and potential sub-categories, which were then discussed and refined. This process continued until all authors reached a consensus. Additionally, we incorporated categories for SFs by consulting relevant literature [12], [13], [27] and applying the same iterative approach.

III. Approach

Our approach to **V**ulnerability **I**dentification via **Per**turbations, *VIPer*, comprises three phases, as illustrated in Fig. 3: (1) *Feature Detection* uses CPG rules to identify the presence of *VFs* in samples and annotate them as candidates for perturbation. Since *SF*s are inherent properties of code, this phase does not pertain to them. (2) *Targeted Perturbation* systematically applies FPPs (alterations in code with no change to a feature) and FEPs (alteration in code that eliminates the feature) to the tagged samples, generating a perturbed dataset for each of the 10 *VF*s. For *SF*s, *VIPer* only generates FPPs because *SF*s are inherent properties of code that cannot be eliminated. We used the structured annotations in the SARD dataset as ground truth for evaluating the correctness of the CPG rules to detect features. We manually compared the CPG rule-detected features in 363 out of the 6,525 files (95% confidence with a 5% margin of error) with the annotations in SARD. When devising the FPPs and FEPs, we ensured that the perturbation targeted the property mentioned in the annotations and later manually analyzed the same 363 files to determine if the FPPs retained the vulnerability and the FEPs removed the vulnerability. (3) *Solution Evaluation* assesses the detectors' responses to the perturbed datasets, examining how perturbations affect the predictive performance of the detectors and their ability to react satisfactorily. A satisfactory reaction is defined as no prediction change for FPPs and a prediction change for

FEPs. With this information, *VIPer* quantifies the influence of these features on the detectors' predictions. The following subsections detail each phase.

A. Feature Detection

The goal of the *Feature Detection* phase is to systematically identify the suitable candidate samples for perturbation. The whole process is broken down into three main steps: 1) CPG Extraction, 2) Pattern Matching, and 3) Sample Annotation. We elaborate each step next:

1) CPG extraction: We use joern [26] to construct the CPG for each sample in the dataset. To facilitate our analysis, we abridge the graph by retaining only edges that represent the most relevant relationships: Data Dependence (DD), Control Dependence (CD), Post-Dominance (PD), DEF, and USE. These edge types are selected because they are widely recognized as essential relationships for vulnerability analysis [13].

A CPG is formally defined as a graph $G = (V, E, \mu)$, where V is the set of all nodes and E the set of all edges in the CPG, and $\mu : \ (V \cup E) \times K \to S$ is a function that sets or retrieves the property of a node or edge. K is the key that denotes which property would be retrieved or set, and S is the value for property K. To facilitate our implementation of *VIPer*, we extended the property function μ's capabilities to retrieve and set additional properties of nodes and edges. Due to space limitation, we list these additional property keys and values (as acronyms) used by property function μ in the online appendix [31].

2) Pattern matching: Once the abridged CPGs are constructed from the dataset samples, we iterate through each CPG to identify which ones satisfy any of our predefined CPG rules for the *VFs*. If a CPG satisfies a rule, we infer that the corresponding dataset sample exhibits the *VF* associated with that rule and is a suitable candidate for perturbation. The CPG rules along with their corresponding *VF* are given in Table II.

To assess if a CPG satisfies a rule, we develop corresponding detection algorithms for each of the rules listed in Table II. Due to the page limitation, we will only describe the detection algorithm for Rule ID: 2.1 for *VF Incorrect Calculation of Buffer Size* which is the CPG rule that the sample in Listing 1 satisfies. We provide the algorithms for the remaining detection algorithms in the online appendix [31].

We will now discuss the algorithm used to check if a CPG satisfies the CPG rule with ID 2.1 - *Incorrect Calculation of Buffer Size*. The description for its rule in Table II states that, for this vulnerability feature to exist in a sample, the CPG constructed from the sample must have a node v representing a buffer write function call that writes n number of bytes to a buffer d with a defined length of LEN_d that is smaller than n (i.e., $LEN_d < n$). To detect this feature, *VIPer* checks if the constructed CPG has a node v representing a buffer write function call where the number of bytes to write (n) is larger than the defined length of the destination buffer (LEN_d) by traversing the constructed CPG to determine the static values for LEN_d and n. Algorithm 1 describes how *VIPer* determines these static values and checks if the sample satisfies Rule 2.1.

Algorithm 1: Incorrect Calculation of Buffer Size

Input: $G = (V, E, \mu)$ representing the CPG constructed from the sample code
Output: Boolean value indicating whether the feature exists in the sample
Property keys for μ
- arg_dest: Destination buffer in a WF
- arg_count: Number of bytes to write in a WF
- type: Type of Node or Edge
- len: Length of a defined buffer
- var: Variable associated with data dependence

Begin
1 Let $V' \subset V \leftarrow \{v$ for $v \in V$ if $\mu(v, type)$ = WF$\}$
2 **for** *node* v *in* V' **do**
3   Let $d \leftarrow \mu(v, arg\_dest)$
4   Let $n \leftarrow \mu(v, arg\_count)$
5   Let $IN_v^{DD} \subset V \leftarrow \{u$ for $u \in V$ if $(u, v) \in E$ and $\mu(u, v, type) =$ DD and $\mu(u, v, var) = d$ $\}$
6   **for** u *in* IN_v^{DD} **do**
7   **if** $\mu(u, type)$ = *AF or* $\mu(u, type)$ = *AD* **then**
8   Let $LEN_d \leftarrow \mu(u, len)$
9   **if** $n > LEN_d$ **then**
10   **return** *true*
11 **return** *false*

In Algorithm 1, Line 2 iterates through every node v in V' where V' is the set of nodes representing a buffer write function (WF) call (e.g., a call to `memcpy`) (Line 1). Lines 3 and 4 in the algorithm retrieve the destination buffer (d) and the number of bytes to write (n) in the function call, respectively. Line 5 retrieves the start nodes of all the incoming data dependence (DD) edges of node v with respect to the destination buffer d (i.e., IN_v^{DD}). Lines 6-10 iterate through the retrieved data dependence edges and check if an edge's start node u represents an allocation function (AF) or an array declaration (AD). If so, it further checks whether the number of bytes to write (n) according to node v exceeds the length defined at node u (Line 9). If true, it concludes that the sample contains the vulnerability *Incorrect Calculation of Buffer Size*. We do not develop separate detection algorithms for *SF*s because the purpose of these algorithms is to identify relevant samples for perturbation, which is not necessary for *SF*s. Unlike *VF*s, *SF*s (including the ones we focused on i.e., Identifier Names, Code Formatting, Set of Nodes, and Set of Edges) are inherent properties of every sample.

3) Sample annotation: Once a feature is detected in a sample, *VIPer* annotates the sample with the detected feature, relevant line numbers, and variable names, which differ across the 10 *VF*s. For example, when *VIPer* detects *Incorrect Calculation of Buffer Size* in a sample (like in Listing 1), it first lists the name of the detected feature as *Incorrect Calculation of Buffer Size* and lists the line number that defines LEN_d (line 4 in Listing 1) and the line number where the value of n is defined (line 7 in Listing 1) and the variable name used for the destination buffer d (`data` for Listing1). We provide the annotations used for the remaining *VF*s in the online appendix [31]. After completing the sample annotations, *VIPer* creates a dataset comprising only the annotated samples. This

TABLE II: CPG Rules

Rule ID	Vulnerability Feature	CPG Rule (condition under which vulnerability exists in a sample)
2.1	Incorrect Calculation of Buffer Size (IBS)	CPG constructed from the sample must have a node v representing a buffer write function call that writes n number of bytes to a buffer d with a defined length of LEN_d that is smaller than n (i.e., $LEN_d < n$).
2.2	Buffer Access Using Size of Source Buffer (BSB)	CPG constructed from the sample must have a node v representing a buffer write function call that writes n number of bytes from a source buffer s with a defined length of LEN_s equaling n to a destination buffer d with a defined length of LEN_d that is smaller than n (i.e., $(LEN_d < n) \wedge (n == LEN_s)$).
2.3	Off-by-one Error (OE)	CPG constructed from the sample must have a node v representing a buffer write function call that writes n number of bytes to a buffer d with a defined length of LEN_d that is smaller than n by exactly one. (i.e., $n == LEN_d + 1$).
2.4	Buffer Over-read (BO)	CPG constructed from the sample must have a node v representing a buffer copy function call that reads n number of bytes from a buffer s with a defined length of LEN_s that is smaller than n (i.e., $LEN_s < n$).
2.5	Double-Free (DF)	CPG constructed from the sample must have two nodes u and v who call `free` on the same buffer b and there exists no node w between u and v that uses an allocation function (e.g., `malloc`) on b.
2.6	Use-After-Free (UAF)	CPG constructed from the sample must have two nodes u and v where node v uses a buffer b after node u already deallocates buffer b.
2.7	Buffer Underwrite (BUW)	CPG constructed from the sample must have a node v that writes to a buffer b using an index value idx where idx is never checked to ensure that it does not hold a negative value.
2.8	Buffer Under-read (BUR)	CPG constructed from the sample must have a node v that reads from a buffer b using an index value idx where idx is never checked to ensure that it is not a negative number.
2.9	Read API (RA)	CPG constructed from the sample must have a node v representing a function call to a sensitive Read API (e.g., `fgets`) where the location $\mathcal{L}$ of node v is a vulnerable line in the sample.
2.10	Write API (WA)	CPG constructed from the sample must have a node v representing a function call to a sensitive Write API (e.g., `memcpy`) where the location $\mathcal{L}$ of node v is a vulnerable line in the sample.

annotated dataset is utilized by the subsequent phase of *VIPer*, *Targeted Perturbation*, which we will describe next.

B. Targeted Perturbation

The goal of this phase is to understand how robust the models are to changing input and whether the detectors are able to learn from the *VF*s instead of the SFs. We posit that the detectors should be able to correctly predict the outcome solely based on the presence of *VF* in the code sample and should remain unchanged if the code sample changes without losing the *VF*. To achieve this, *VIPer* perturbs samples in two ways: preserving the feature (FPP) or eliminating it (FEP). The underlying principle is that if a perturbation leaves the *VF* intact (FPP), the detector's prediction should remain unchanged. Conversely, if a perturbation eliminates the *VF* (FEP), the detector's prediction should change accordingly.

Fig. 3 provides a high-level overview of how *VIPer* generates perturbations. The input for this phase is the annotated dataset generated by the previous phase, *Feature Detection*. From this dataset, *VIPer* extracts the three essential elements required for perturbation: (1) the original sample in the dataset where the *VF* was detected, (2) the name of the detected *VF* and (3) the relevant line numbers and variable names for generating perturbations. Based on the detected feature's name, *VIPer* selects one of 10 tailored perturbation generation algorithm sets. Each set involves generating one or more FPPs and FEPs. Due to page limitations, we will only elaborate on the perturbation generation algorithm set for *Incorrect Calculation of Buffer Size*. The remaining nine algorithm sets are provided in the online appendix [31]. Recall the CPG rule for *VF Incorrect Calculation of Buffer Size* is $LEN_d < n$. The algorithms used for generating FPPs and FEPs for this CPG rule are given as follows:

1) FPP generation: When generating FPPs, we want to make sure that the perturbation still retains the rule $LEN_d < n$. There are two ways this can be achieved, (1) decreasing the value of LEN_d or (2) increasing the value of n. To apply these two types of perturbations, *VIPer* uses Algorithm 2. In lines 1 & 2 of the algorithm, it extracts the values of LEN_d and n from line numbers extracted from the feature annotated samples. In lines 3 & 5, it creates two clones of the original sample, and in the first clone applies perturbation (1) (i.e., decreasing the value of LEN_d) (see line 4) while in the second clone, it applies perturbation (2) (i.e., increasing the value of n) (see line 6).

Algorithm 2: FPP: Incorrect Calc. of Buffer Size

Input:
G = CPG constructed from the sample code
$feat\_name$ = name of the detected feature
u = line defining the destination buffer
v = line representing WF
Output: Boolean value indicating whether the feature exists in the sample
Begin
1 Let $LEN_d \leftarrow \mu(u, len)$
2 Let $n \leftarrow \mu(v, arg\_count)$
3 Let $G_1 = (V_1, E_1, \mu_1) \leftarrow$ G.clone()
4 $\mu_1(u, len) \leftarrow LEN_d - 1$
5 Let $G_2 = (V_2, E_2, \mu_2) \leftarrow$ G.clone()
6 $\mu_2(v, arg\_count) \leftarrow n + 1$
7 **return** G_1, G_2

2) FEP generation: When generating FEPs, we want to achieve the opposite goal and ensure that the perturbation no longer satisfies the rule $LEN_d < n$. Again, there are two ways this can be achieved, (1) increasing the value of LEN_d to match the value of n or (2) decreasing the value of n to match the value of LEN_d. To apply these two types of perturbations, *VIPer* uses Algorithm 3. Similar to the previous algorithm, in lines 1 & 2 of Algorithm 3, *VIPer* extracts the values of LEN_d and n from line numbers extracted from the feature annotated samples and in lines 3 & 5, it creates two clones of the original sample. However, unlike the previous algorithm, to generate FEPs, *VIPer* increases the value of LEN_d to match n for the

Algorithm 3: FEP: Incorrect Calc. of Buffer Size

```
Input:
  G = CPG constructed from the sample code
  feat_name = name of the detected feature
  u = line defining the destination buffer
  v = line representing WF
Output: Boolean value indicating whether the feature exists
        in the sample
Begin
1 Let LEN_d ← μ(u, len)
2 Let n ← μ(v, arg_count)
3 Let G_1 = (V_1, E_1, μ_1) ← G.clone()
4 μ_1(u, len) ← n
5 Let G_2 = (V_2, E_2, μ_2) ← G.clone()
6 μ_2(v, arg_count) ← LEN_d
7 return G_1, G_2
```

first clone (see line 4), and for the second clone, it decreases the value of n to match LEN_d (see line 6).

To generate perturbations targeting Spurious Features (SFs) in vulnerable samples, we employ separate approaches for token-based and graph-based detectors. Since *VIPer* evaluates token-based approaches for *SF*s that are established in previous literature, we use existing methods of perturbation for these *SF*s. Specifically, we adopt the symbolization mechanism from *Li et al.* [1] to perturb identifier names and leverage the auto-indentation feature of the CLion IDE [32] to introduce indentations into code samples.

In contrast, for graph-based *SF*s, we develop new perturbations. Importantly, since modifications to *SF*s do not impact the sample's vulnerability ground truth, all generated perturbations for *SF*s are FPPs. The goal for *SF* perturbations is to modify the nodes and edges in a sample with minimal possible change, without affecting the vulnerability ground truth. Therefore, for *Node Set*, the corresponding perturbation is inserting a *printf("");* statement at the start of each function (i.e., the *Print instructions without variables SF*) and for *Edge Set*, the corresponding perturbation is inserting *if(0==1) return;* at the start of each function (i.e., the *Unreachable conditional operation without variables SF*). [1] Note that these perturbations are generic and are applied to all samples in the dataset.

C. Solution Evaluation

The goal of this phase is to measure the effect of the perturbations generated in the previous phase on the predictions of the detectors. By analyzing the predictions of the detectors on the perturbed dataset, *VIPer* measures how the *VF*s and *SF*s influence the detector's prediction. Fig. 3 depicts a high-level overview of the *Solution Evaluation* phase. First, we train the detectors on the SARD dataset. Next, we use the dataset samples annotated in the *Feature Detection* phase and their corresponding FPPs and FEPs generated in the *Targeted Perturbation* phase to retrieve the predictions of the detectors. Using these predictions, *VIPer* analyzes the detector's response to the perturbations. Specifically, it measures how satisfactory are the detectors' responses. Recall that a satisfactory reaction is when a detector retains its predictions for FPP perturbations or when it changes predictions for FEP perturbations. Based on this information, we calculate the satisfaction rate SR_f of the detectors on the perturbations targeting a feature f for a dataset X to measure the influence of f on the detector's prediction. Formally, the satisfaction rate is defined as:

$$SR_f = \left(\frac{T'_{FPP} + T'_{FEP}}{T_{FPP} + T_{FEP}}\right) \times 100$$

where T_{FPP} is the total number of FPPs generated from X for f, T_{FEP} is the total number of FEPs generated from X for f, T'_{FPP} is the total number of FPPs that retain the detector's prediction (expected outcome), T'_{FEP} is the total number of FEPs that change the prediction of the detector (expected outcome).

Additionally, the aim is to understand how FPPs and FEPs individually impact the detectors. *VIPer* measures this impact by calculating the satisfaction rate of FPPs SR_f^{FPP} and FEPs SR_f^{FEP} for a feature f individually as shown below:

$$SR_f^{FPP} = \left(\frac{T'_{FPP}}{T_{FPP}}\right) \times 100 \qquad SR_f^{FEP} = \left(\frac{T'_{FEP}}{T_{FEP}}\right) \times 100$$

IV. Evaluation Setup

A. Research Questions

Our evaluation aims to answer three research questions.

- ***RQ1*** – How do the targeted perturbations affect the reported prediction accuracy of the vulnerability detectors?
- ***RQ2*** – What are the detectors' responses to perturbations targeting different features?
- ***RQ3*** – To what extent do the FPPs and FEPs influence the detectors' predictions?

B. Classifying Evaluation Results

For the purpose of our analysis and discussion, specifically RQ3, we classify detector responses into one of four SR_f^{FPP}–SR_f^{FEP} combinations: high–high (HH), high–low (HL), low–high (LH), and low–low (LL). We consider the satisfaction rate high if the value is within 3% (chosen value ϵ) of the average satisfaction rate for a perturbation type targeting feature f. This ensures that detectors performing close to the average are still recognized as effective.[2] Detectors in the *HH* category exhibit a desired understanding of feature f since they retain predictions for FPPs and change predictions for FEPs. These detectors will not experience significant drop in precision or recall in the presence of perturbations. A detector in the *HL* category exhibits an understanding of feature f, but that understanding does not align with the ground-truth characteristics of f (i.e., the relevant CPG rule). These detectors will experience a noticeable drop in precision in the presence of perturbations. A detector in the *LH* category changes its

[1]For better readability, we are referring to the *SF*s *Print instructions without variables* and *Unreachable conditional operation without variables* as *Node Set* and *Edge set*, respectively, for the rest of the paper.

[2]While our analysis would be carried out the same way regardless of the selected value ϵ, we selected this value based on established conventions, as an ϵ of $0.01-0.03$ is frequently used in surveys, such as the American Statistical Association [33] and the American Community Survey's methodology for population and housing estimates [34].

predictions given any perturbation involving feature f. Such detectors will experience a significant drop in recall. Lastly, detectors in the *LL* category will exhibit a significant drop in both precision and recall in the presence of perturbations.

Researchers and engineers can use *VIPer*'s results to assess the vulnerability detectors and decide which option is best suited for their purpose. Ideally, one would always prefer a detector falling under the HH category. If none of the available detectors fall under that category, their users will have to assess whether a detector that falls within one of the other categories is suitable for their tasks. Generally, vulnerability detection tasks prioritize recall over precision, i.e., tolerating false alarms while fixing as many vulnerabilities as possible. In such cases, the engineer may prefer detectors falling under the HL category. However, there may be certain tasks where precision is preferred over recall. An example is as a large and stable system in which vulnerable code is less prevalent and the cost of handling false alarms is significant. In that case, the LH category may be considered.

C. Evaluation Subjects

We investigated whether five state-of-the-art vulnerability detectors learn from *VF*s or *SF*s: DeepWukong [12], Reveal [13], DeepDFA [20], LineVul [14], and SySeVR [11].

DeepWukong leverages an advanced graph neural network (GNN) to embed code fragments into a low-dimensional representation [12].

ReVeal generates function-level prediction by using gated GNNs that are intended to make the model capable of understanding complex code semantics and dependencies [13].

DeepDFA detects function-level vulnerabilities by extracting abstract dataflow information from functions and applying a Gated Graph Sequence Neural Network to learn vulnerabilities in the extracted dataflow [20].

LineVul predicts software vulnerabilities at the line level leveraging the BERT architecture [35] with self-attention layers. It first predicts the vulnerable functions and then locates the specific vulnerable lines within those functions.

SySeVR uses syntax-based vulnerability candidates (SyVCs) from code and semantic-based candidates (SeVCs) from control and data dependencies, by representing them into vectors suitable for marking the vulnerabilities in code [11].

D. Evaluation Dataset

To assess the five evaluation subjects, *VIPer* uses the largest vulnerability dataset, SARD, containing production, synthetic, and academic programs (a.k.a. test cases). We utilized the curated version of SARD from SySeVR as it is the largest dataset used among all five vulnerability detectors [36]. The dataset includes 22,080 C/C++ files. The vulnerable programs in SARD provide precise locations of each vulnerability, enabling effective analysis. In total, the dataset had 366,419 C/C++ functions that we used for training the five evaluation subjects.

V. Results

In this section, we present the results corresponding to the three posed research questions in the previous section.

A. RQ1: Analysis of Accuracy

RQ1 investigates the impact of targeted perturbations on the detectors' predictive performance. We assess the detectors' performance using two widely adopted evaluation metrics [37] - Precision and Recall. Table III presents the changes between the original and the perturbed dataset for each identified *VF* and *SF* in terms of predictive performance. A negative value indicates a decline, whereas a positive value indicates an increase in the corresponding metric. For perturbations targeting *VF*s that belong to *Overflow* vulnerability sub-class in Fig. 2, overall, we observe that all five detectors suffer a noticeable decrease in precision. However in case of SySeVR, the drop is 48% on average, higher than the other four detectors. As for recall, DeepWukong shows a drop up to 25.78% while the rest of the detectors show minimal drop with LineVul retaining its original recall.

For *VF*s that belong to *Deallocated Buffer Use* vulnerability sub-class, we observe that all graph-based detectors (DeepWukong, ReVeal, and DeepDFA) noticeably outperform the token-based detectors (LineVul and SySeVR). DeepWukong achieves the smallest drop in precision (3.3% on average) and a slight increase in recall (1.99% on average), while DeepDFA exhibits the highest drop in precision (up to 24%). Since both token-based approaches demonstrate relatively poor performance (i.e., over 50% drop in precision on average for LineVul and similar drop in recall for SySeVR), this could be due to the fact that *Deallocated Buffer Use* vulnerabilities are characterized mainly by the control flow of a program [26]. Since graph-based detectors, by design, are able to better capture context information from graphs, they are expected to outperform token-based approaches. However, DeepDFA only focuses on the dataflow of a function which may explain its drop in precision. For *VF*s that belong to *Numeric Range Comparison* vulnerabilities, LineVul shows the highest drop in precision (33%) among the five detectors.

For *Sensitive API Call VF*s, LineVul and ReVeal show the biggest drop in precision by 78% on average with LineVul also showing the largest drop in recall (50%). SySeVR gets the second-largest drop in precision with 65% on average. Results for *VF*s under this sub-class are surprising since this *VF* focuses on calls to specific system APIs; token-based approaches are by design expected to effectively leverage tokens containing system API names. We elaborate on this particular aspect in Section VI.

For the *SF Code Formatting*, we observe that both SySeVR and LineVul are reasonably resilient with the drop in precision and recall never exceeding 1%. For the *SF Identifier Name*, LineVul shows a significant drop in precision with over 67%, while SySeVR's precision drops only by 6.7%. This is likely because SySeVR preprocesses raw code tokens (e.g., symbolizing code elements to prevent learning from custom naming conventions) while LineVul converts the raw code tokens directly into vectors thus exposing itself to this *SF*. For two *SF*s that belong to the graph-based sub-class, we observe that DeepWukong only suffers a noticeable drop in recall for

TABLE III: Accuracy Metrics Change (Original→ Perturbed) [least values are in bold]

	Feature Name	Graph-based						Token-based			
		DeepWukong		ReVeal		DeepDFA		LineVul		SySeVR	
		PREC	REC	PREC	REC	PREC	REC	PREC	REC	PREC	REC
VF	IBS	-33.38	**-9.55**	-35.42	0.00	-9.94	-2.57	-35.42	0.00	**-49.70**	0.00
	BSB	-52.40	**-25.78**	-54.39	0.00	-8.31	-0.76	-54.39	0.00	**-69.70**	0.00
	OE	-12.78	4.87	-24.55	-1.97	-7.71	-4.20	-21.94	0.00	**-32.33**	**-5.95**
	BO	-11.85	4.31	-22.71	**-1.21**	-14.40	-6.60	-22.53	0.00	**-41.76**	2.72
	DF	0.00	0.00	-18.71	-4.04	-24.23	-1.38	**-60.00**	0.00	0.00	**-57.71**
	UAF	-6.69	3.99	-18.65	-3.42	-23.02	-1.31	**-47.25**	0.00	1.29	**-51.05**
	BUW	-34.02	**-16.27**	-9.41	-4.84	-10.94	-3.39	**-45.88**	-1.43	-30.43	1.38
	BUR	-10.45	**-6.69**	-22.13	-6.12	-21.44	-4.81	-20.15	-1.24	**-28.01**	-3.44
	RA	0.00	0.00	**-100.00**	0.00	9.92	1.66	**-100.00**	0.00	**-100.00**	0.00
	WA	-30.48	**-4.61**	**-57.04**	0.01	1.51	0.47	**-57.04**	0.01	-29.99	0.95
SF	Identifier Name	-	-	-	-	-	-	-67.69	-0.35	-6.70	**-10.41**
	Code Formatting	-	-	-	-	-	-	-0.41	-0.03	-0.01	0.00
	Node Set	**-3.90**	-0.74	-3.44	**-18.42**	0.16	0.03	-	-	-	-
	Edge Set	-1.08	**-29.04**	**-3.87**	-21.38	0.06	-0.01	-	-	-	-

the *SF Edge Set* (29.04%), while ReVeal's recall drops for both *SF*s. This reduction in recall in both cases suggests that graph-based approaches tend to generate false negatives whenever encountering a behavior-preserving perturbation, such as introducing a mock edge to the graph. However, in case of *Node Set* perturbations, DeepWukong is less influenced, possibly due to the fact that it only considers control and data dependence edges from the CPG; since *Node Set* perturbations do not introduce any of these dependencies, they do not influence DeepWukong's predictions. In contrast, ReVeal considers all the nodes from CPG, and is thus inherently exposed to *Node Set* perturbations. The precision and recall drops for DeepDFA are negligible ($< 0.5\%$). Similarly to the above discussion of DeepWukong, this is due to the fact that DeepDFA only works on dataflow information and *SF* perturbations introduce no changes to a program's dataflow. At the same time, it may be worth investigating whether DeepDFA retains the same robustness against *SFs* that impact dataflow information, e.g., the use of intermediate variables during a mathematical calculation.

B. RQ2: Analysis of Feature Satisfaction Rate

RQ2 investigates the detectors' robustness to perturbations targeting individual features using the SR_f metric introduced in Section III-C. Fig. 4 presents a heatmap illustrating the SR_f of the detectors for feature-specific perturbations – The larger the SR_f, the more intense the color in the cell. Among the five detectors, DeepWukong has the highest average SR_f for perturbations targeting *VF*s that belong to *Overflow* vulnerability sub-class with 88% while SySeVR has the lowest average SR_f with 61%. For *VF*s that belong to *Deallocated Buffer Use* vulnerability sub-class, we observe an opposing scenario from RQ1 where both token-based detectors demonstrate a better SR_f (15% higher) than the graph-based detectors.

For *VF*s that belong to *Numeric Range Comparison* vulnerability sub-class, SySeVR displays the lowest SR_f with 57% on average while rest of the detectors demonstrate higher SR_f, i.e., $86\% - 87\%$. For *Sensitive API Call VF*s, DeepWukong achieves the highest SR_f with 95% on average. It is worth noting that, for the feature *Read API*, *VIPer* does not generate FPPs since we could not find replacement for system read functions from the list of vulnerability-causing system APIs [1], [29], [30] with matching argument list and syntax. Therefore, *VIPer* only generates FEPs for this *VF*. Since all the FEPs targeting this feature retain predictions for ReVeal and LineVul, their SR_f is 0. We will discuss this particular detectors' behavior in the next Section VI.

For *SF*s that belong to the *Token-based* sub-class, both LineVul and SySeVR produce roughly the same SR_f. For *SF*s that belong to graph-based sub-class, DeepWukong achieves a slightly higher SR_f than ReVeal, while DeepDFA achieves near perfect SR_f for both *SFs*.

C. RQ3: Analysis of FPP and FEP Satisfaction Rate

The goal of RQ3 is to determine which type of perturbations (FPPs or FEPs) have the most significant impact on the detectors' predictions. Table IV shows the SR_f^{FPP} and SR_f^{FEP} for individual *VFs*. As can be seen in Table IV, for perturbations targeting *VF*s that belong to *Overflow* vulnerability sub-class, the average SR_f^{FPP} is 99.3% and since all five detectors' SR_f^{FPP} are around ~3% of the average we consider all their SR_f^{FPP} to be high. However, the average SR_f^{FEP} is 9.8%, which is lower than the recommended minimum threshold for metrics measuring the intelligence of AI systems [38] (i.e., 51%). Therefore, if the SR_f^{FEP} is lower than 51% we

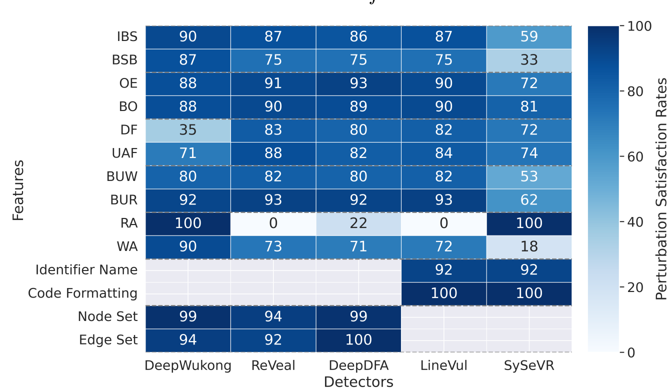

Fig. 4: Perturbation Satisfaction Rates

consider the value to be low. This means that all five detectors receive very low SR_f; specifically, LineVul never changes its prediction for FEPs. Therefore, all five of them fall under the category HL. For *VF*s falling that belong to *Deallocated Buffer Use* vulnerability sub-class, using the same principle as before, we selected thresholds for SR_f^{FPP} and SR_f^{FEP} as 93.88% and 51% respectively. We observe that DeepDFA, DeepWukong, LineVul, and ReVeal remain in category HL while SySeVR falls under category LH for deallocated buffer use vulnerabilities. For *VF*s that belong to *Numeric Range Comparison* vulnerability sub-class, the selected thresholds for SR_f^{FPP} and SR_f^{FEP} are 97.66% and 51% respectively. Therefore, all five detectors fall under the category HL. As mentioned when discussing RQ2, *VIPer* does not generate FPPs for *Read API VF*, therefore, we only consider SR_f^{FEP}. We observe that DeepWukong and SySeVR have a 100% SR_f^{FEP} while LineVul and ReVeal demonstrate 0% SR_f^{FEP}, with DeepDFA achieving 22% SR_f^{FEP}. Note that since we do not have any FPPs for this specific *VF*, we cannot characterize the detectors under one of the four categories. For Write API *VF*s, the selected thresholds for SR_f^{FPP} and SR_f^{FEP} are 99.68% and 51% respectively. For this *VF* also, all five detectors fall under the category HL.

VI. Discussion and Implications

The evaluation of five detectors by *VIPer* yielded valuable insights, revealing the detectors' respective strengths and weaknesses. In the case of DeepWukong, *VIPer* revealed that its lowest SR_f comes from *VF*s *Double-Free* and *Use-After-Free*, which both belong to the sub-class *Deallocated Buffer Use*. Given that these vulnerabilities are primarily characterized by program control flow [26], *VIPer*'s findings suggest that DeepWukong struggles to comprehend control dependencies and their contribution to vulnerability. *VIPer* shows that DeepWukong is also influenced by the *SF Edge Set*, which may be a key factor underlying its limited understanding of control dependencies.

VIPer's findings indicate that ReVeal struggles to understand the impact of system APIs on vulnerabilities as it displays the lowest SR_f for *VF Write API*. ReVeal also shows a drop in recall for *Node Set* and *Edge Set SF*s. *VIPer*'s findings suggest that these *SF*s may impede ReVeal's ability to recognize crucial dependencies related to system API calls.

Similarly to ReVeal, DeepDFA struggles to understand how system APIs impact vulnerabilities, evidenced by it demonstrating the lowest SR_f for the *VF Write API*. This is because DeepDFA does not capture the system APIs in the dataflow analysis during the learning process. On the other hand, unlike ReVeal, DeepDFA is not affected by graph-based *SF*s.

VIPer shows that graph-based detectors ReVeal and DeepWukong experience a drop in recall from the *SF Edge Set*. This may be attributed to SARD's fix generation process for certain vulnerable code. Specifically, we observed that SARD often fixes vulnerabilities involving sensitive system APIs by introducing control dependencies with unsatisfiable conditions (e.g., *if(5!=5)*). We observed this in 326 source files. *VIPer*'s findings suggest that these control dependencies may lead graph-based detectors to mistakenly associate them with fixes for the vulnerability. There could be two possible remedies to improve the accuracy of vulnerability detection models: 1) we recommend avoiding such spurious control dependencies when generating fixes in synthetic vulnerability datasets, or 2) preventing vulnerability detectors from learning from the spurious control dependencies by eliminating them using an edge filtering algorithm during preprocessing. Since DeepDFA incorporates the above-mentioned remedies by only extracting dataflow information, it does not get influenced by these *SF*s.

VIPer's findings also indicate that LineVul achieves a 100% SR_f^{FPP} but a 0% SR_f^{FEP}. This disparity suggests that LineVul severely overfits when trained on SARD, excelling in false positive reduction but failing to generalize effectively. *VIper* also shows that the *SF*, *Identifier Name* impacts LineVul's predictions by reducing its precision. This *SF* is one of the contributors to LineVul's overfitting to SARD. Therefore, LineVul may benefit from using some preprocessing technique on the tokens (e.g., token symbolizing) to avoid exposing itself to the *SF Identifier Name*. For SySeVR, it gets lower SR_f compared to the four other detectors for most of the features. However, the lowest value is from the *VF*, *Write API*, suggesting that SySeVR's token symbolization may inadvertently capture system APIs, hindering its ability to learn from these API names and understand their contribution to vulnerabilities. SySeVR may address this by modifying its token symbolization to include only user-defined functions.

VII. Threats to validity

1) External validity: This threats refers to the generalizability of the experiments and *VIPer*. We mitigated this threat by conducting a large-scale study including five recent and representative vulnerability detectors following token-based and graph-based approaches. [3] Although, the dataset employed in this study only contains C/C++ code samples, *VIPer*'s primary goal is to ensure accurate representation of vulnerabilities that should be learnt by the detectors, thus, it is language agnostic and dataset independent. Another threat may arise from our reliance on the SARD dataset when devising *SF*s. However, SARD is the largest available dataset that is widely used and contains many real-world security flaws. To further confirm that our results are not inadvertently impacted by SARD, we also examined how the graph-based *SF*s influence ReVeal by utilizing the FFmpeg+Qemu dataset. Our findings indicate that both *SF*s also affect ReVeal's predictions on FFmpeg+Qemu, in a manner consistent with our SARD-based results. This supports the *SF*s' broad applicability.

2) Internal validity: This threat may arise from a weak research protocol or selection bias. We overcome this threat by strictly following the reproducible guidelines provided by the authors of the employed vulnerability detectors. Also, the

[3] We additionally investigated the possibility of using a sixth vulnerability detector, DeepVD [39]. However, we were unable to reproduce DeepVD's published results and did not receive a response from its authors when we asked for clarification. For this reason, we excluded DeepVD from our study.

TABLE IV: Satisfaction Rates for FPP and FEP

VF	Graph-based						Token-based			
	DeepWukong		ReVeal		DeepDFA		LineVul		SySeVR	
	FPP	FEP	FPP	FEP	FPP	FEP	FPP	FEP	FPP	FEP
IBS	99.33	32.13	100.00	0.00	97.96	0.00	100.00	0.00	100.00	0.00
BSB	99.25	47.50	100.00	0.00	98.94	1.61	100.00	0.00	100.00	0.00
OE	99.63	5.30	98.18	19.51	98.20	48.84	100.00	0.00	96.92	20.63
BO	99.97	10.26	99.43	4.96	99.44	0.76	100.00	0.00	95.89	17.14
DF	100.00	0.00	94.68	30.95	94.15	16.67	100.00	0.00	64.29	91.67
UAF	100.00	1.19	98.20	35.66	95.66	12.40	100.00	0.00	93.86	44.30
BUW	91.45	34.92	98.52	5.23	97.55	3.36	100.00	0.00	94.43	8.58
BUR	99.29	5.88	98.87	4.71	98.54	5.82	100.00	0.00	98.72	2.08
RA	-	100.00	-	0.00	-	22.22	-	0.00	-	100.00
WA	98.73	43.93	100.00	0.00	97.82	2.50	100.00	0.00	100.00	18.37

decision to opt for SARD dataset and specific vulnerability detectors was made based on a comprehensive literature analysis. SARD is relatively the largest dataset available and the chosen detectors are widely employed in prior work.

3) Construct validity: One of the critical design decisions made in this study is to restrict *SF*s for graph-based approaches to two *Node Set* and *Edge Set* in the taxonomy, however, there might be other spurious features that the detectors could be learning from. That said, the taxonomy can be further extended as needed. Another construct validity threat may arise due to the employed evaluation metrics. When reproducing the vulnerability tools, we utilized their adopted evaluation metrics. In contrast, the metric "satisfaction rate" is unique to this study which is employed to measure the extent to which the vulnerability detectors deviate from the desired outcomes when FPP and FEP are applied to the code samples.

4) Conclusion validity: This threat concerns with the authenticity and significance of the findings reported in this study. We mitigated this threat by rigorously following authors' guidelines while reproducing the results for their vulnerability detectors. Also, due to absence of replacement for system read APIs, we do not generate FPPs for the *VF Read API* and hence did not include in our results. Another threat may be the use of Joern to extract the CPG from each dataset sample. Joern has been observed to occasionally miss dependencies. However, its developers have actively tried to address this issue in recent updates. In addition to using an updated version, we mitigated this threat by filtering out the invalid CPGs constructed by Joern.

VIII. Related work

Understanding the features relevant to vulnerability prediction (i.e., *VF*s) is critical for improving the DL-based vulnerability detectors' reliability and trustworthiness. Recent studies have, therefore, attempted to explore the *VF*s in DL-based vulnerability detectors. Risse et al. [18] studied the vulnerability detectors' ability to learn fixes for vulnerable patches. Suneja et al. [40] probed the signal-awareness of models used for vulnerability detection by reducing the input source code to the minimum tokens required to retain the prediction. Meanwhile, our approach for devising *VF*s is primarily focused on properties related to code dependencies. It is also the first to evaluate whether vulnerability detection techniques understand these code dependencies.

Another way researchers pursue explaining the predictions of DL-based models is by examining the unintended pattern in the dataset that the models might learn erroneously (i.e., *SF*s). On that note, recent studies have explored the influence of variable names [17], [41] and method names [17], [42] in the prediction of the models of code. Risse et al. [18] analyzed the behavior of models of code on logic-preserving transformations. Our work is the first to categorize *SF*s based on how they impact token-based and graph-based approaches, and to develop *SF*s based on graph properties of code. Additionally, this paper presents the dichotomy between *VF*s and *SF*s through an illustrative taxonomy as well as an evaluation framework that allows for the assessment of both *VF*s and *SF*s within the same platform.

IX. Conclusion and Future Work

Previous studies have shown that existing vulnerability detectors fail to generalize well to unseen datasets, suggesting they may be learning irrelevant code features. To address this, we introduce *VIPer*, a framework for accurately evaluating vulnerability detectors by providing a comprehensive view of the features that truly contribute to vulnerabilities and how they impact the predictions of vulnerability detectors. *VIPer* employs feature-preserving and feature-eliminating perturbations to assess a detector's performance. Our results reveal that, in the case of vulnerability features, approximately ~2% of feature-preserving perturbations and ~84% of feature-eliminating perturbations have an adverse impact on detector outcomes. In the case of spurious features, we observe that feature-preserving perturbations produced a drop in recall up to 29% for graph-based detectors. To facilitate correct evaluation and improvement of vulnerability detectors, we have made *VIPer* publicly available to the research community. For future work, we plan to discover more vulnerability and spurious features and explore ways to ensure that vulnerability detectors handle them properly.

Acknowledgment

This work is supported in part by the National Science Foundation under grant CCF-2106871.

References

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

[2] N. Bhatt, A. Anand, V. S. S. Yadavalli, and V. Kumar, "Modeling and characterizing software vulnerabilities," *International Journal of Mathematical, Engineering and Management Sciences*, 2017.

[3] B. Grobauer, T. Walloschek, and E. Stocker, "Understanding cloud computing vulnerabilities," *IEEE Security & privacy*, vol. 9, no. 2, pp. 50–57, 2010.

[4] F. Piessens, "A taxonomy of causes of software vulnerabilities in internet software," in *Supplementary Proceedings of the 13th International Symposium on Software Reliability Engineering*. Citeseer, 2002, pp. 47–52.

[5] J. C. Santos, A. Peruma, M. Mirakhorli, M. Galstery, J. V. Vidal, and A. Sejfia, "Understanding software vulnerabilities related to architectural security tactics: An empirical investigation of chromium, php and thunderbird," in *2017 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2017, pp. 69–78.

[6] A. Sejfia, S. Das, S. Shafiq, and N. Medvidović, "Toward improved deep learning-based vulnerability detection," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–12.

[7] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Detection of recurring software vulnerabilities," in *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, 2010, pp. 447–456.

[8] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE transactions on software engineering*, vol. 37, no. 6, pp. 772–787, 2010.

[9] 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*, 2016, pp. 201–213.

[10] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *2017 IEEE symposium on security and privacy (SP)*. IEEE, 2017, pp. 595–614.

[11] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "Sysevr: A framework for using deep learning to detect software vulnerabilities," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 4, pp. 2244–2258, 2021.

[12] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, "Deepwukong: Statically detecting software vulnerabilities using deep graph neural network," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 3, pp. 1–33, 2021.

[13] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet," *IEEE Transactions on Software Engineering*, 2021.

[14] M. Fu and C. Tantithamthavorn, "Linevul: A transformer-based line-level vulnerability prediction," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 608–620.

[15] J. Cito, I. Dillig, V. Murali, and S. Chandra, "Counterfactual explanations for models of code," in *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, 2022, pp. 125–134.

[16] 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*, 2016, pp. 1135–1144.

[17] M. M. Rahman, I. Ceka, C. Mao, S. Chakraborty, B. Ray, and W. Le, "Towards causal deep learning for vulnerability detection," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–11.

[18] N. Risse and M. Böhme, "Limits of machine learning for automatic vulnerability detection," *arXiv preprint arXiv:2306.17193*, 2023.

[19] U. D. of Commerce, "Sard's website." [Online]. Available: https://samate.nist.gov/SARD/

[20] B. Steenhoek, H. Gao, and W. Le, "Dataflow analysis-inspired deep learning for efficient vulnerability detection," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024, pp. 1–13.

[21] "Viper's replication package." [Online]. Available: https://zenodo.org/records/14838016

[22] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[23] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 2018, pp. 757–762.

[24] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, "Ac/c++ code vulnerability dataset with code changes and cve summaries," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 508–512.

[25] U. D. of Commerce, "Nist's website." [Online]. Available: https://www.nist.gov/

[26] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *2014 IEEE Symposium on Security and Privacy*. IEEE, 2014, pp. 590–604.

[27] S. Fan, X. Wang, C. Shi, P. Cui, and B. Wang, "Generalizing graph neural networks on out-of-distribution graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2023.

[28] M. Usman, R. Britto, J. Börstler, and E. Mendes, "Taxonomies in software engineering: A systematic mapping study and a revised taxonomy development method," *Information and Software Technology*, vol. 85, pp. 43–59, 2017.

[29] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks." in *USENIX security symposium*, vol. 98. San Antonio, TX, 1998, pp. 63–78.

[30] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer overflows: Attacks and defenses for the vulnerability of the decade," in *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, vol. 2. IEEE, 2000, pp. 119–129.

[31] "Viper's online appendix." [Online]. Available: https://sites.google.com/view/viper-framework

[32] JetBrains, "Clion: A cross-platform ide for c and c++ by jetbrains." [Online]. Available: https://www.jetbrains.com/clion/

[33] "American statistical association." [Online]. Available: https://www.amstat.org/docs/default-source/amstat-documents/pol-seeingthroughstats_spring_2013.pdf

[34] "American community survey." [Online]. Available: https://www.census.gov/programs-surveys/acs

[35] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[36] "Sysevr's sard." [Online]. Available: https://github.com/SySeVR/SySeVR/blob/master/Program%20data/SARD/SARD.7z

[37] D. M. Powers, "Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation," *arXiv preprint arXiv:2010.16061*, 2020.

[38] F. Cabitza and A. Campagner, "Who wants accurate models? arguing for a different metrics to take classification models seriously," *arXiv preprint arXiv:1910.09246*, 2019.

[39] W. Wang, T. N. Nguyen, S. Wang, Y. Li, J. Zhang, and A. Yadavally, "Deepvd: Toward class-separation features for neural network vulnerability detection," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 2249–2261.

[40] S. Suneja, Y. Zheng, Y. Zhuang, J. A. Laredo, and A. Morari, "Probing model signal-awareness via prediction-preserving input minimization," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 945–955.

[41] Z. Yang, J. Shi, J. He, and D. Lo, "Natural attack for pre-trained models of code," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1482–1493.

[42] M. V. Pour, Z. Li, L. Ma, and H. Hemmati, "A search-based testing framework for deep neural networks of source code embedding," in *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2021, pp. 36–46.