Commit-Level, Neural Vulnerability Detection and Assessment

Yi Li
New Jersey Inst. of Technology
New Jersey, USA
yl622@njit.edu

Aashish Yadavally
University of Texas at Dallas
Texas, USA
aashish.yadavally@utdallas.edu

Jiaxing Zhang
New Jersey Inst. of Technology
New Jersey, USA
jz48@njit.edu

Shaohua Wang*
New Jersey Inst. of Technology
New Jersey, USA
davidshwang@ieee.org

Tien N. Nguyen
University of Texas at Dallas
Texas, USA
tien.n.nguyen@utdallas.edu

ABSTRACT

Software Vulnerabilities (SVs) are security flaws that are exploitable in cyber-attacks. Delay in the detection and assessment of SVs might cause serious consequences due to the unknown impacts on the attacked systems. The state-of-the-art approaches have been proposed to work directly on the committed code changes for early detection. However, none of them could provide both commit-level vulnerability detection and assessment at once. Moreover, the assessment approaches still suffer low accuracy due to limited representations for code changes and surrounding contexts.

We propose a Context-aware, Graph-based, Commit-level Vulnerability Detection and Assessment Model, VDA, that evaluates a code change, detects any vulnerability and provides the CVSS assessment grades. To build VDA, we have key novel components. First, we design a novel context-aware, graph-based, representation learning model to learn the *contextualized embeddings for the code changes* that integrate *program dependencies* and the surrounding *contexts* of code changes, facilitating the automated vulnerability detection and assessment. Second, VDA considers the mutual impact of learning to detect vulnerability and learning to assess each vulnerability assessment type. To do so, it leverages *multi-task learning* among the vulnerability detection and vulnerability assessment tasks, improving all the tasks at the same time. Our empirical evaluation shows that on a C vulnerability dataset, VDA achieves 25.5% and 26.9% relatively higher than the baselines in vulnerability assessment regarding F-score and MCC, respectively. In a Java dataset, it achieves 31% and 33.3% relatively higher than the baselines in F-score and MCC, respectively. VDA also relatively improves the vulnerability detection over the baselines from 13.4–322% in F-score.

CCS CONCEPTS

• **Computing methodologies → Neural networks**; • **Security and privacy → Software security engineering**.

KEYWORDS

Neural Networks; Deep Learning; Software Security; Vulnerability Detection; Vulnerability Assessment

ACM Reference Format:
Yi Li, Aashish Yadavally, Jiaxing Zhang, Shaohua Wang, and Tien N. Nguyen. 2023. Commit-Level, Neural Vulnerability Detection and Assessment. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616346

1 INTRODUCTION

Software Vulnerabilities (SVs) are security weaknesses and flaws that one can exploit in cyber-attacks. Such vulnerable code can be found and reported by scanning a version of a software project with the help of vulnerability detection tools [31, 48, 50]. However, it is crucial to identify SVs as soon as possible because late fixes and consequent damage due to affected systems are severe. Toward that, *commit-level vulnerability detection* (VD) approaches [12, 38, 49] have been proposed to catch SVs as soon as new code changes are committed to the code repositories during software development. They are useful in warning developers early and isolating vulnerability-inducing, fine-grained code change. An alternative solution as using a VD tool running on the snapshot after a commit would not be able to isolate vulnerability-inducing changes.

While addressing software vulnerabilities is crucial, providing developers with the assessment on the impacts of the detected vulnerability in the system under development is of equal interest. Such knowledge will help better prioritize code review tasks, while showing which areas need patching first. The assessment could be on the severity of the attack and the levels of damage regarding confidentiality, integrity, availability, etc. However, the commit-level VD approaches [12, 38, 49] do not support such automated assessment. Recognizing this, the state-of-the-art approaches [33] proposed commit-level software vulnerability assessment (VA) tools to assess a committed code change. They leveraged the Common Vulnerability Scoring System (CVSS) [3] ratings (i.e., numerical scores which can be transformed into qualitative representations such as low, medium, high, and critical) manually provided by security analysts to build a learning-based approach to predict assessment ratings for a code change possessing vulnerabilities. However, the existing VA tools still have limitations.

First, they work only on a committed code change that has already been identified as vulnerable by a VD tool. This strategy that

*Corresponding Author

runs VD and then VA (i.e., VD $\rightarrow$ VA) is prone to cascading errors. For instance, in commits that have wrongly been identified as vulnerable by a VD tool, the VA tool would give incorrect assessments, which should not be given at all. Moreover, the VA tool fails to predict assessment patterns in commits that are vulnerable, but missed by the VD tool. As a result, the VD $\rightarrow$ VA strategy is limited, and would not take advantage of the *mutual benefits of the learning of detection on the learning of assessment, and vice versa*. Our philosophy is driven by this inter-dependence, wherein the former could help isolate the part of the code change that is vulnerability-inducing, which, in turn, could help assess it better. Specifically, if a model learns that one of the assessment ratings is high (e.g., high severity or complete unavailability), the vulnerability detection outcome must be positive. If the detection outcome is negative, the assessment ratings for all the aspects must be at the lowest (e.g., none). Thus, it is more beneficial for a model to jointly learn to detect and assess vulnerabilities at the same time, leading to performance enhancement in both detection and assessment.

Second, the state-of-the-art VA approaches [33] are inept in their *representation of code changes*, which are not sufficiently context-aware. They learn an *n*-gram representation for code changes that captures limited local context, and is incapable of learning from VD/VA-relevant program statements that are further apart from the changed statements. As a result, the statements in the *n*-gram representation might not all contain important features for VD/VA. Such a sequential representation learning also enforces an order in source code, which might not be the order of execution (e.g., in the cases of loop, branch condition, or recursion), which contributes to detecting vulnerabilities involving execution and exception flows. Moreover, given that a vulnerability attack exploits the control and data flows, a VD/VA tool needs to consider program dependencies. This facet is ignored in the existing VA tools. Finally, the context of un-changed code is indistinguishable and not represented separately from the changes. As a result, it can be confused by two training instances that have the same combination of change and context overall, but with different changes and associated contexts.

In this work, we present VDA, a Context-aware, Graph-based, Commit-level Vulnerability Detection and Assessment model that evaluates a changed code to detect the presence of a vulnerability, while also providing corresponding CVSS assessment scores for the detected vulnerability. We build VDA via the novel integration of three key ideas. First, we *integrate vulnerability detection and assessment (VDA)* processes such that our tool can directly be built into a repository to evaluate committed code changes and provide just-in-time assistance. Due to the inter-dependence of these processes, we adopt a *multi-task learning* approach to propagate the learning between VD and VA, leading to better performance of both tasks. Moreover, the assessments for different aspects could also affect one another [33], e.g., between the availability and integrity of a system. Thus, our multi-task learning scheme includes the VD task and the assessment tasks for different security aspects.

Second, we address the limitation of code change representations described earlier via a novel *context-aware, graph-based, representation learning model* that integrates the *program dependencies*, and the surrounding *contexts* of code changes. We leverage both versions of the program dependence graphs (PDGs), i.e., before and after a commit via the multi-version δ-PDG [37]. To build such embeddings for code changes, VDA *explicitly represents the contexts* surrounding the changed statements via the sub-graphs in δ-PDG. It considers the impact of the context represented by a context vector on the building of the embeddings for the code changes. It uses the contextualized embeddings to predict if the changes have any vulnerability, and if yes, to provide assessment grades. Third, it utilizes the Label, Graph Convolution Network [7] to encode the program dependencies among the entities in the changed code and the ones in the surrounding un-changed code. This helps overcome the issues with *n*-grams and capture the statements that might be far apart, but still relevant to the vulnerability.

We have conducted experiments to evaluate VDA on real-world vulnerabilities. Our results on a C vulnerability dataset show that VDA achieves 25.5% and 26.9% relatively higher than the state-of-the-art VA tool DeepCVA [33] in F-score and MCC, respectively. The vulnerability detection result from VDA is improved over the state-of-the-art ML/DL-based VD approaches from 11.3-–146% in precision, 10.4-–553% in recall, and 13.4—322% in F-score. The results on a Java dataset with 1,229 vulnerabilities show that VDA also achieves 31.0% and 33.3% relatively higher than DeepCVA [33] in F-score and MCC, respectively.

To gain better insights, we conducted experiments to show that the better performance of VDA roots from our designed components. Our results show that *our novel code change embeddings* help VDA learn better class-separation, i.e., better in classifying the commits into the classes for vulnerability detection and assessment. Moreover, we used *explainable AI* to show that VDA indeed leverages the correct features in the dependencies among statements for its assessments. The key contributions of our work include

1) VDA: commit-level vulnerability detection and assessment model (VDA) that performs VD and VA in tandem, leveraging multi-task learning to improve both detection and assessment.

2) Our novel context-aware, graph-based embeddings for code changes integrate program dependencies and contexts. This embedding model is applicable for other down-stream tasks.

3) Empirical evaluation. We evaluated VDA against the state-of-the-art approach. Our model/code are available at [5].

2 MOTIVATION

2.1 Motivating Example

Let us present an example from an HTML parser, named *Jsoup*, and our observations. Figure 1 displays the information on the vulnerability CVE-2021-37714 that was reported on *Jsoup*, and published on 08/18/21. The change that was deemed to contribute to the vulnerability were committed at version 1.12.1 to the method `process(Token,HtmlTreeBuilder)` of the `HtmlTreeBuilderState` class (lines 10–12 of Figure 2). That change directly uses the value returned from `resetInsertionMode()` as the condition to insert `startTag` (line 13). With this change, certain input HTML code with a specific start tag could make the program go to line 16 with a recursive call to the method `process(...)`. That call resulted in an NullPointerException at line 3. In other cases, the parser can get stuck, i.e., *"loop indefinitely until canceled"* as described in the official description of CVE-2021-37714. Figure 1 also shows the Common Vulnerability Scoring System grades (CVSS) given by security experts for various vulnerability assessment types (VATs) for that CVE. Due to the

Vulnerability Details: CVE-2021-37714
1. Description: *jsoup is a Java library for working with HTML. Those using jsoup versions prior to 1.14.2 to parse untrusted HTML or XML may be vulnerable to DOS attacks. If the parser is run on user supplied input, an attacker may supply content that causes the parser to get stuck (loop indefinitely until cancelled), to complete more slowly than usual, or to throw an unexpected exception. This effect may support a denial of service attack. The issue is patched in version 1.14.2. There are a few available workarounds. Users may rate limit input parsing, limit the size of inputs based on system resources, and/or implement thread watchdogs to cap and timeout parse runtimes.*
Publish Date : 2021-08-18 Last Update Date : 2022-02-07
2. Vulnerability Type(s): Denial Of Service
3. CVSS Score: ...
4. Detailed CVSS Grades:

Vulner. Assess. Type	Value	Description
Confidentiality Impact	**None**	No impact to the confidentiality
Integrity Impact	**None**	No impact to the integrity
Availability Impact	**Complete**	There is reduced performance or interruptions in availability
Access Complexity	**Low**	Specialized access conditions or extenuating circumstances do not exist Little knowledge is required to exploit
Authentication	**Not Req**	Authentication is not required to exploit the vulnerability
Gained Access	**None**	No gained access with the vulnerability
Access Vector	**Local**	The vulnerability is in the local parser

Figure 1: Vulnerability Details for Jsoup: CVE-2021-37714

```
1  // .../jsoup/parser/HtmlTreeBuilderState.java
2  boolean process(Token t, HtmlTreeBuilder tb) { ...
3    if (t.isCharacter()&& inSorted(
             tb.currentElement().normalName(),InTableFoster)){
4        ...
5        return tb.process(t);
6    }
7    ...
8    } else {
9        tb.popStackToClose(name);
10 -     tb.resetInsertionMode();
11 -     if (tb.state() == InTable) {
12 +     if (!tb.resetInsertionMode()) {
13          tb.insert(startTag);
14          return true;
15        }
16        return tb.process(t, InHead);
17        ...
18 }
```

Figure 2: Code Change at Version 1.12.1 for CVE-2021-37714

above effects, the availability impact for this vulnerability is rated as *Complete* (i.e., for some inputs, there will be reduced performance and interruptions in available services).

The above vulnerability is potentially damaging. However, in the existing workflow, it was detected and reported late, and finally assessed by the security experts in the CVSS system [3]. In the meantime, the similar vulnerabilities with the same assessments were also reported in other applications from *Netapp*, *Oracle*, and *Quarkus*, i.e., more widespread damages were done.

In contrast, by taking advantage of the records of vulnerabilities and their corresponding assessments in CVSS via a *learning-based approach*, this could have been addressed as soon as the code change is committed. Such a *commit-level VDA not only allows an early detection and assessment, but also isolates the vulnerability-inducing code change*, which cannot be achieved if a VD tool is run on the project's snapshot after changes. For example, the vulnerability-inducing code change in *Jsoup* at lines 10–12 needs to be pinpointed among several other benign changes in the same commit.

In this work, toward enabling such a learning process, we make the following observations.

2.1.1 *Observation 1 [Mutual Impact of Vulnerability Detection and Assessment].*

In Figure 2, if a model learns that the availability assessment of this vulnerability is *Complete* (i.e., system could be completely unavailable), it could learn that this change is a vulnerability-introducing one (i.e., the detection outcome is positive). On the other hand, if a model learns that this is a vulnerability-introducing change, it could learn more about the semantics of the code change, and consequently how the code change gives hints to the assessment. In contrast, if a model decides that this does not possess any vulnerability, it can learn that all the VAT outcomes must be *None*. Unfortunately, none of existing VD and VA approaches take advantage of this mutual impact. The ML-based vulnerability detection approaches [12, 38, 49] focus only on VD, without any VA support. In contrast, the existing ML-based VA approaches [33] works only on the vulnerability-introducing change with the work flow that a VD tool is used first to detect such a commit.

2.1.2 *Observation 2 [Program Dependencies].*

To detect and assess a vulnerability, a model needs to consider the program dependencies among the statements. For example, to assess Availability, one needs to check the potential infinite loop or null-pointer exception, and examine the control and data dependencies between the changed line 12 and the line 16. That is where the method `process` is recursively called, which leads to the null-pointer exception at line 3 (`currentElement()` returns null). Unfortunately, the state-of-the-art vulnerability assessment models, e.g., DeepCVA [33] learn only n-gram representations (n=1,3,5), and do not consider long-range dependencies. However, program statements that can help with automated assessment (e.g., line 12 $\longrightarrow$ line 16 $\longrightarrow$ line 3) can be far apart. DeepCVA is incapable of capturing such a flow.

2.1.3 *Observation 3 [Context].*

By examining *only the tokens involving in the changes* (e.g., the tokens `tb`, `resetInsertionMode`, `state`, and `InTable` in the deleted lines 10–11, and the inserted line 12), a model can not decide on the vulnerability or its impacts on the system's availability or not. Generally, *the same/similar changes occurring in different surrounding contexts might cause different effects*. For example, adding a null check: `if p != null` is a common change in many places. However, it could prevent a null-pointer exception in some context, while does not in the others. Unfortunately, DeepCVA [33] does not capture well the contexts of the changes. First, the n-gram representation learning is limited as explained. Second, DeepCVA does not model the inter-relationships between the context, i.e., the un-changed code elements and the changed ones (both pre-change, and post-change). This is essential in capturing code changes better – resulting in improved detection and assessment.

2.2 Key Ideas

We develop VDA, a Context-aware, Graph-based, Commit-level Vulnerability Detection and Assessment Model that detects any vulnerability in a committed change and provides the CVSS assessment grades for it if any. VDA is designed with the following ideas:

2.2.1 *Key Idea 1 [Vulnerability Detection and Assessments with Joint Learning].*

From the Observations, a commit-level VDA tool is desired. We leverage multi-task learning to propagate the mutual benefits of learning of vulnerability detection to that of assessment, and vice versa. Moreover, the learning of one type of

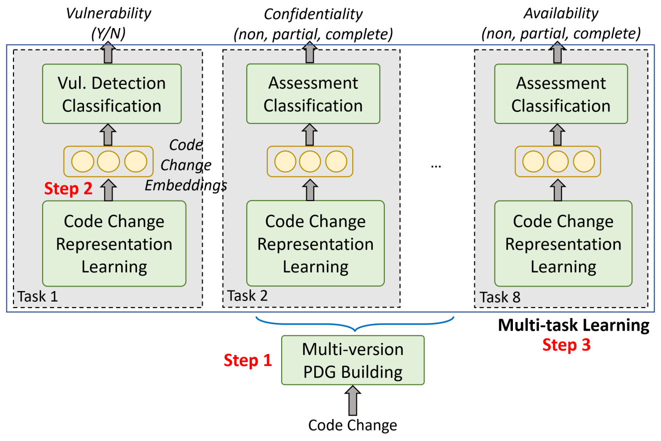

Vulnerability (Y/N)
Confidentiality (non, partial, complete)
Availability (non, partial, complete)
Vul. Detection Classification
Assessment Classification
Assessment Classification
Code Change Embeddings
Step 2
Code Change Representation Learning
Task 1
Task 2
Task 8
...
Multi-task Learning
Step 3
Step 1
Multi-version PDG Building
Code Change

Figure 3: VDA: Context-aware, Graph-based, Commit-level Vulnerability Detection and Assessment

vulnerability assessment can affect the learning of another (e.g., between Availability and Access Complexity). Thus, our multi-task learning scheme is designed with one task for VD, and one task for each of the assessment types. In training, we optimize the joint loss function for all the tasks. When the VD's outcome is positive, VDA provides assessments. Otherwise, non-impact scores are given.

2.2.2 Key Idea 2 [Contextualized Embeddings for Code Changes with Graph-based Representation Learning]. To overcome the limitation of code change representation, we introduce a graph-based model to build the *contextualized embeddings for code changes* that integrate both *program dependencies* and *surrounding contexts*.

Unlike existing code change embedding approaches [9, 23] where code changes are represented as sequences, we explicitly represent code changes and the context of a change via a graph representation, called multi-version PDG [37]. The graph consists of the program entities of both versions before and after the changes, and their dependencies. The context is defined as the surrounding, un-changed nodes of the changed node. We use Label, Graph Convolution Network (Label-GCN) [7] to build the contextualized embeddings for code changes, considering the contexts as the weights in computation. We use past vulnerabilities and experts' ratings to train VDA to build such embeddings, and use them for classification.

2.2.3 Key Idea 3 [Program Dependencies in Code Change Representation via Graph-based Neural Network]. We leverage Label-GCN [7] to incorporate the program dependencies among the changed code elements and the surrounding un-changed code elements into VDA. The graph enables a partial order among program entities in a PDG rather than enforcing a total order as in n-gram learning. This enables VDA overcome the aforementioned issues, and capture dependencies on distant yet vulnerability-relevant statements.

3 APPROACH OVERVIEW

VDA has three key components working in three steps (Figure 3).

Step 1. Representing Code Changes and Contexts with Multi-version PDG. Program Dependence Graph (PDG) [16] is a directed graph with a set N of nodes and a set E of edges, in which a node $n \in N$ represents a program statement or a conditional expression; an edge $e \in E$ represents the data or control flow among the statements. We adopt *the multi-version PDG (δ-PDG^{x,y}) [37] to represent the changes between two versions x and y, before and after the commit.* A multi-version PDG δ-PDG^{x,y} [37] is a directed graph generated from the disjoint union of all nodes and edges in the PDGs at versions x and y. δ-PDG^{x,y} (Figure 4, Section 4) allows us to capture the program dependencies including data/control flows that are crucial in vulnerability detection and assessment (Key idea 2).

Step 2. Contextualized Embeddings for Code Changes with Graph-based Representation Learning. We develop our representation learning model to learn the contextualized embeddings for the code changes in a commit that integrate both program dependencies and the contexts. To build the contextualized embeddings, we leverage the Label, Graph-based Convolution Network [7] to learn the vector v for each node n in the graph whose nodes can have labels. We use labels to denote the nodes at either the versions x (before changes) or y (after changes), or at both versions.

For a changed node n_c, we collect all the un-changed nodes in the context of n_c, which is defined as the set of all the un-changed nodes that are k-hop neighbors of n_c. In Figure 4, for the changed statement at line 4, if $k = 1$, the context of that change includes the statements at lines 2, 5, and 7 (i.e., 1-hop neighbors from line 4).

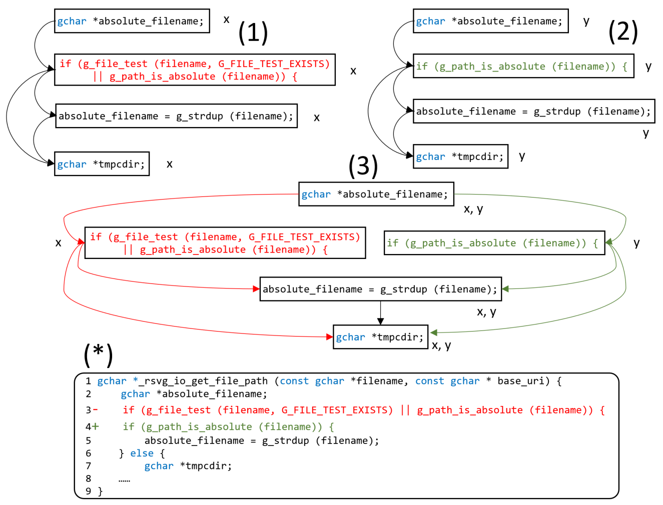
From the context nodes, we compute the vector for the context for a changed node n_c and use it as a weight to represent the impact of context to build the contextualized embedding for n_c. From those embeddings, we compute the vector for the entire commit and feed it to a SoftMax layer acting as a classifier to perform vulnerability classification or assessment classification for each assessment type. The softmax function is often used as the last activation function of a neural network to normalize the output of a network to a probability distribution over the predicted output classes.

Step 3. Multi-Task Learning for Classification. For each assessment type, we have a SoftMax layer working as an assessment classification model on the embedding of the entire commit. We also have another SoftMax layer as the classifier for vulnerability detection. To propagate the impact of the classification for one assessment type on one another, we leverage multi-task learning among all the classification models. We use the uncertainty weighted multi-task loss [13] for each classification task as the final multi-task learning loss function and use the maximum of the average F-score from all the classification tasks as the training target.

Training and Predicting Processes. The training and predicting processes share the above steps, except that in training, the classification labels for the vulnerable commit and vulnerability assessment types (VATs) are known. For a benign commit, the output labels are negative and non-impact. When predicting, VDA takes a code change, predicts vulnerability and provides the assessment classifications for seven VATs. Next, we will explain VDA in details.

4 BUILDING MULTI-VERSION PDG (δ-PDG^{x,y})

The first step in VDA is to build the multi-version PDG. Figure 4(*) shows the vulnerable method `rsvg_io_get_file_path`. The change at line 3 into line 4 was deemed as a vulnerability-introducing change by a detection tool, in which `g_file_test(filename, G_FILE_TEST_EXISTS)`


(1)
gchar *absolute_filename;
x
if (g_file_test (filename, G_FILE_TEST_EXISTS) || g_path_is_absolute (filename)) {
x
absolute_filename = g_strdup (filename);
x
gchar *tmpcdir;
x
(2)
gchar *absolute_filename;
y
if (g_path_is_absolute (filename)) {
y
absolute_filename = g_strdup (filename);
y
gchar *tmpcdir;
y
(3)
gchar *absolute_filename;
x, y
x
if (g_file_test (filename, G_FILE_TEST_EXISTS) || g_path_is_absolute (filename)) {
if (g_path_is_absolute (filename)) {
y
absolute_filename = g_strdup (filename);
x, y
gchar *tmpcdir;
x, y
(*)

```
1 gchar *_rsvg_io_get_file_path (const gchar *filename, const gchar * base_uri) {
2     gchar *absolute_filename;
3-    if (g_file_test (filename, G_FILE_TEST_EXISTS) || g_path_is_absolute (filename)) {
4+    if (g_path_is_absolute (filename)) {
5         absolute_filename = g_strdup (filename);
6     } else {
7         gchar *tmpcdir;
8     ……
9 }
```

Figure 4: Multi-Version Program Dependence Graph

was removed from the condition at line 3. Figure 4(1) and Figure 4(2) display the PDGs of the method `rsvg_io_get_file_path` before and after the change. All the nodes of the PDG before the change are marked with x, and those of the PDG after the change are marked with y. Figure 4(3) shows the multi-version PDG (δ-PDG^{x,y}) built from the two versions x and y of that method before and after the change. In δ-PDG^{x,y}, the nodes labeled with either x or y appear only in the PDG of the version x or y, respectively. The nodes labeled with (x, y) appear in the PDGs at both versions.

We adopt the multi-version graph building algorithm in Flexeme [37]. Specifically, we generate the PDGs for both versions x and y. We run Git diff tool on the source code to determine the changed and unchanged nodes for the statements. The added nodes are kept in δ-PDG^{x,y} with the labels y as they appear in the newer version y. We also retain the deleted nodes and use the label x for them. The unchanged nodes between the versions are matched by using string similarity among the respective statements to filter the candidates and line-span proximity to rank them. When considering the edge changes, we back-propagate the delete nodes to the edges flowing into them. We also add all unmatched edges in the newer version y to the PDG^{x,y} as the edges relevant to the added nodes.

After building δ-PDG^{x,y}, for each changed node in the graph, VDA collects all the unchanged nodes within the k-hops and the inducing edges among them to build a sub-graph as the context for the changed node. δ-PDG^{x,y} and the context for each changed node are used as the input of the Label-GCN. Building δ-PDG^{x,y} and contexts is needed in both training and predicting.

5 GRAPH-BASED, CONTEXTUALIZED EMBEDDINGS FOR CODE CHANGES

Let us first explain how we extract the feature vectors for the nodes in δ-PDG^{x,y} and build the *code change contextualized embeddings* that integrate both *program dependencies and contexts* via Label-GCN. From those embeddings for code changes, we build the embedding for the entire commit, which will be later used in the classification tasks for vulnerability detection and assessment.

5.1 Feature Vectors v_n^f for δ-PDG^{x,y} Nodes

After the previous step, we obtain δ-PDG^{x,y} and the context sub-graphs for the changed nodes in δ-PDG^{x,y}. We leverage Label-GCN [7] to model δ-PDG^{x,y} as follows. We first build the node feature vector for each node n in δ-PDG^{x,y}. To do so, we use the sequence of the code tokens t_n of the statement s corresponding to n. We use a word embedding model to learn the vector v_t for each token when we consider the code token sequence for s as a sentence. The *node content vector* for n is computed as the average vector v_{avg} of all the vectors v_t of all the tokens in the statement s. To integrate the labels x, y, and (x, y) into the node feature, we use a one-hot vector with the length of three for those labels. By concatenating the node content vector with the one-hot vector for the labels, we have the *node feature vector* v_n^f for the node n.

5.2 Contextualized Embedding v_n for Node n

Next, we replace each node n in δ-PDG^{x,y} with the node feature vector v_n^f. Similar to the traditional GCN [25], Label-GCN [7] takes the graph with the node feature vectors as the input and generates the embeddings for each node in the graph. In addition, for the current node, in the first layer, *Label-GCN considers the version labels (x,y, (x, y)) of the neighboring nodes as part of the feature vectors, which helps distinguish the old/new nodes.* Label-GCN generates the vectors (embeddings) for the nodes in each layer as follows:

$$H_{(l)} = \begin{cases} \sigma[(\hat{A}X - diag(\hat{A})\sum_{j=1}^{K} e_j e_j^T)W^0] & l = 1 \\ \sigma(\hat{A}H^{(l-1)}W^{(l-1)}) & l \geq 1 \end{cases} \quad (1)$$

$$\hat{A} = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}} \quad (2)$$

$$\tilde{A} = A + I \quad (3)$$

Where H is the output for hidden layers; A is the adjacency matrix and I is the identity matrix; $\tilde{D}$ is the diagonal node degree matrix; W is the weight matrix; X is the input and $X \in R^{num \times (d+K)}$; num is the number of nodes; d is the dimension of node features; K is the number of types of node labels in the input (K=3 for x, y, and (x, y)); and $-diag(\hat{A})\sum_{j=1}^{K} e_j e_j^T$ is used to eliminate the self-loops for the components of the feature vectors for the labels.

The vectors $H(l)$ at the output layer are used as the vectors v_is for the nodes in the δ-PDG^{x,y} graph after Label-GCN in Figure 5.

5.3 Context Integration

For each changed node n_c in δ-PDG^{x,y}, we build the sub-graph containing all the un-changed nodes in the k-hop neighbors of n_c and use that sub-graph as the context for n_c. We merge the vectors v_is in the context into a matrix. We then use a fully connected layer on the matrix to build the *context vector* v_{ctx} for the context.

To integrate the context into the embeddings, we compute the final vector v'_{n_c} for the changed node n_c by performing the cross-product between v_{ctx} and the vector v_{n_c} for n_c computed by the Label-GCN model as described in Section 5.2: $v'_{n_c} = v_{ctx} \times v_{n_c}$.

5.4 Classification for Each Task

To compute the vector for the entire commit, we collect all the vectors v'_{n_c}s of the changed nodes n_c in δ-PDG^{x,y} into a matrix. We apply a fully connected layer to learn the vector v_{M_i} for a changed

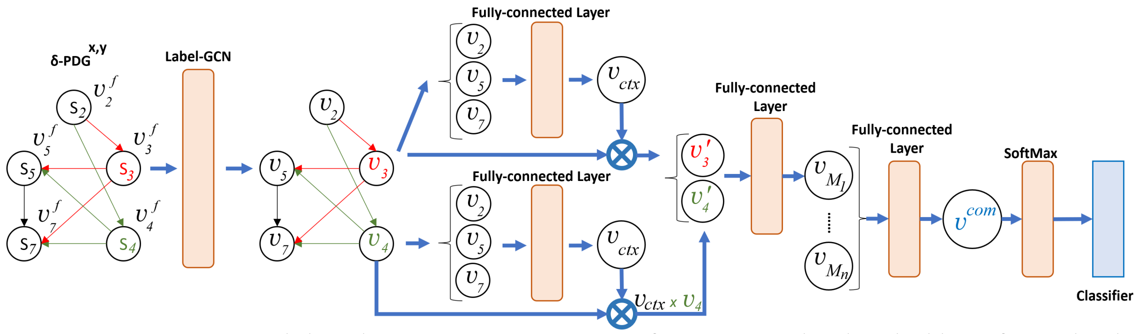


Figure 5: Context-aware, Graph-based Representation Learning for Contextualized Embeddings for Code Changes

method M_i. The vectors for all changed methods are passed through another fully connected layer to get the vector v^{com} for the commit. This vector is fed into each SoftMax classifier for each task.

5.5 Illustrating Example

Figure 5 illustrates the vector building for the code changes in Figure 4. From δ-PDG^{x,y}, we compute the node feature vectors v_n^f (Section 5.1) for all statement nodes S_2,...,S_7 in that graph. Label-GCN takes that graph with node feature vectors to generate the embeddings v_2,..., v_7 for the nodes (Section 5.2).

Let us consider the changed nodes, n_3 and n_4. The context for n_3 with one-hop distance includes n_2, n_5, and n_7. Similarly, the context for n_4 includes the same nodes. After merging those vectors into a matrix, which is passed through a fully-connected layer, we obtain the context vectors v_{ctx} for n_3. Similarly, we obtain v_{ctx} for n_4. Then, the final contextualized vector for the changed node n_3 taking the context into account is computed as the cross-product: $v_3' = v_{ctx} \times v_3$. Similarly, $v_4' = v_{ctx} \times v_4$. We then put the vectors v_3' and v_4' together in a matrix and use a fully-connected layer (FCL) to produce the method vector v_{M1}. All changed method vectors are passed through another FCL to get v^{com} for the commit. Finally, we pass v^{com} to a SoftMax layer to perform vulnerability detection or the classification for an assessment type (e.g., *None*, *Partial*).

6 MULTI-TASK LEARNING FOR PREDICTION AND ASSESSMENT CLASSIFICATION

In the previous sections, we have explained how VDA performs each classification task for detection or assessment. In this section, we will explain our multi-task learning mechanism to perform the classification for vulnerability prediction and the classifications for seven VATs with the following prediction classes for each VAT:

(1) **Confidentiality**: None; Partial; Complete
(2) **Integrity**: None; Partial; Complete
(3) **Availability**: None; Partial; Complete
(4) **Access Vector**: Local; Network
(5) **Access Complexity**: Low; Medium; High
(6) **Authentication**: None; Single
(7) **Severity**: Low; Medium; High

As explained in Section 5.4, the vector v^{com} representing the entire commit is passed through a SoftMax layer for the classification for vulnerability detection or for a specific VAT. In VDA, the multi-task learning mechanism uses the uncertainty weighted multi-task loss [13] to learn all classification tasks at the same time. Specifically, for each classification task, VDA uses a cross-entropy loss function to do the classification as follows:

$$L = -log(Softmax(y, f(x))) \tag{4}$$

Where $f(x)$ is the output of a classification task f; y is the ground truth. To get the joint loss function for all the tasks with uncertainty weighting, following Kendall *et al.*'s [13], we have

$$L_i(W) = -log(Softmax(y_i, f^W(x_i))) \tag{5}$$

$$L(W, \sigma_1, \sigma_2, ..., \sigma_8) = \sum_{i=1}^{8} \frac{1}{2\sigma_i^2} L_i(W) + log\sigma_i^2 \tag{6}$$

Where W is the weight adding to the input, σ_i is the i^{th} noise scalar, and W and σ_i are both trainable in the model.

With Formula (6), VDA uses the multi-task learning to train all classification models together with the features for each task. For training, we set as the objective the highest average F-score (Section 7.2) for all tasks. For prediction, the trained model produces the classification results for vulnerability detection and for all VATs.

7 EMPIRICAL EVALUATION

To evaluate VDA, we seek to answer the following questions:

RQ1. Comparison with State-of-the Art ML-based Vulnerability Detection Approaches on C dataset. How well does VDA perform compared with the existing DL-based VD approaches?

RQ2. Comparison in Vulnerability Assessment on a C Dataset. How well does VDA perform compared to the state-of-the-art model in vulnerability assessment on a C dataset?

RQ3. Contextualized Embeddings for Code Changes. Do VDA's embeddings help it improve over the baseline in classifications?

RQ4. Explainable AI to Study Relevant Features on Program Dependencies. Does VDA use program dependencies in vulnerability detection and assessment?

In RQ3 and RQ4, we aim to evaluate the extent of contributions of two VDA's key design choices, i.e., *contextualized embeddings for code changes* and *program dependencies* to its performance.

RQ5. Ablation Study on Multi-Task Learning and Context. How do multi-task learning and context affect VDA's performance in vulnerability detection and assessment?

RQ6. Comparison in Vulnerability Assessment on a Java Dataset. How does VDA perform compared to the state-of-the-art model in vulnerability assessment on a Java dataset?

Table 1: Statistics of BigVul and CVAD Datasets

Datasets	BigVul (C)	CVAD (Java)
# of Projects	303	246
# of Vulnerabilities	3336	542
# of Vulnerability Introducing Commits	7851	1229

7.1 Datasets

We used two vulnerability datasets in C and Java: BigVul 2.0 [15] and CVAD [33] (Table 1). Both were manually checked and used in prior research [15, 29, 33]. Our experiments were conducted on a server with 16 core CPU and a single Nvidia A100 GPU.

7.2 Experimental Methodology

7.2.1 Comparison on Vulnerability Detection on BigVul (RQ1).

Baselines. We include **VCCFinder** [38] (a commit-level ML-based VD), and other ML VD tools: **VulDeePecker** [50], **Devign** [48], **SySeVR** [31], **Russell** *et al.* [41], **Reveal** [11], and **IVDetect** [29]. Except VCCFinder, we ran the others on the code after commit.

Procedure. We used all vulnerable methods and randomly selected the same number of non-vulnerable methods from the fixed version projects, to build a dataset with the vul:non-vul ratio of 1:1. We also evaluated the tools with the real-world ratio of 9:1. We randomly split the data 80%, 10%, 10% on the project basis without changing the vul:non-vul ratio for training, tuning, and testing.

Parameter Tuning. For VDA, we used autoML [4] for tuning the following hyper-parameters to have the best performance: (1) Epoch size (100, 200, 300); (2) Batch size (64, 128, 256); (3) Learning rate (0.001, 0.003, 0.005, 0.010); (4) word embeddings length (150, 200, 250, 300). We tuned DeepCVA's parameters from its documentation.

Evaluation Metrics. We use the following metrics to measure the effectiveness of a model: $Recall = \frac{TP}{TP+FN}$, $Precision = \frac{TP}{TP+FP}$, and $F\text{-}score = \frac{2*Recall*Precision}{Recall+Precision}$. where TP = True Positives; FP = False Positives; FN = False Negatives; TN = True Negatives.

7.2.2 Comparison on Vulnerability Assessment on BigVul (RQ2).

Baseline. We compare VDA with DeepCVA [33].

Procedure. We used BigVul dataset with the same longitudinal setting in [14, 33] for training, validation, and testing to mimic the real-world scenario in which the older vulnerabilities are used for training to assess the newer. Specifically, we sorted all the commits in a chronological order based on their absolute time. We divided the commits into 10 equal folds from oldest to newest. For a fold k ($k \leq 8$), we used all the folds 1 to k for training, the $(k+1)^{th}$ and $(k+2)^{th}$ folds for validation and testing. Models are tuned as in RQ1.

Evaluation Metrics. We use the same metrics in DeepCVA [33]: F-score and Matthews Correlation Coefficient (MCC). F-score ranges from 0 to 1 (the best), to evaluate classification tasks and to handle the class imbalance prevalent in some of the VATs. MCC ranges from -1 to 1 (the best). Since we evaluate the classification models with multiple classes, we used the macro F-score [44] and the multi-class version of MCC [18]. The overall MCC is computed as the average of the MCCs for all classification tasks as in DeepCVA [33].

7.2.3 Code Change Embedding Analysis (RQ3). We aim to evaluate the impact of our novel graph-based, contextualized code change embedding model on VDA's ability in class separation.

Table 2: Vulnerability Detection on C Dataset (RQ1)

Approach	Precision	Recall	F-score
VCCFinder [38]	0.28	0.13	0.18
VulDeePecker [50]	0.55	0.77	0.64
SySeVR [31]	0.54	0.74	0.63
Russell *et al.* [41]	0.54	0.72	0.62
Devign [48]	0.56	0.73	0.63
Reveal [11]	0.62	0.69	0.65
IVDetect [29]	0.54	0.77	0.65
VDA	**0.69**	**0.85**	**0.76**

7.2.4 Program Dependencies (RQ4). We employ Explainable AI (XAI) to demonstrate the utilization of program dependencies by our VDA model in vulnerability detection and assessment. An XAI model enables us to analyze the essential features within the input code that influence the model's predictive outcomes. If XAI highlights program dependencies within the input code, we can subsequently investigate whether VDA appropriately employs these program dependency-related features for accurate predictions.

7.2.5 Ablation Study on Multi-Task Learning and Context (RQ5). We evaluate the impacts of the following factors in VDA: (1) multi-task learning, and (2) change context. We conducted an analysis by removing each factor from VDA and made a comparison.

7.2.6 Comparison on Assessment on CVAD Java dataset (RQ6).

Baseline and Procedure. We compared VDA with the baseline DeepCVA [33]. We used the same procedure, tuning, and longitudinal setting as in RQ1, but on the Java dataset.

8 EXPERIMENTAL RESULTS

8.1 Comparison on Vulnerability Detection on BigVul C Dataset (RQ1)

As seen in Table 2, to detect vulnerability, VDA improves over all the baselines in all the metrics. Specifically, VDA relatively improves over the baseline models from **11.3%– 146%** in Precision, from **10.4%–553%** in Recall, and from **13.4%–322%** in F-score.

VDA performs much better than commit-level VCCFinder [38], possibly because VCCFinder only uses traditional SVM. We can also see that VDA performs better than the snapshot baselines (i.e., Reveal [11] and IVDetect [29]), in the cases where the code changes are relevant to the vulnerability. Consider an example in which a statement (e.g., null check) was removed leading to vulnerable code (e.g., Null-Pointer Exception). Due to its ability to examine changes, as opposed to the baselines which only look at the modified version, VDA can detect the vulnerability-inducing changes better.

In Section 8.4, we used *Explainable AI to display the changed statements* in the code that contributes to the detected vulnerability (Figure 10). This is another advantage from commit-level VD (pointing to the *fine-grained vulnerability-inducing change*) that the VD tools working on the new version only do not have.

We also use a real-world vulnerability setting with a 9:1 non-vulnerability to vulnerability ratio. We randomly selected 10% of the vulnerable instances in the test set ten times, and finally took the average of the F-scores. We report that the F-scores for VDA and the top baseline IVDetect are 45.3% and 25.4%, respectively. VDA

Table 3: Vulnerability Assessment on C Dataset (RQ2)

CVSS Metric	Evaluation Metric	Model	
		DeepCVA [33]	VDA
Confidentiality	macro F-score	0.50	0.65
	MCC	0.23	0.31
Integrity	macro F-score	0.42	0.55
	MCC	0.24	0.33
Availability	macro F-score	0.47	0.63
	MCC	0.28	0.34
Access Vector	macro F-score	0.58	0.69
	MCC	0.22	0.31
Access Complexity	macro F-score	0.49	0.66
	MCC	0.26	0.35
Authentication	macro F-score	0.67	0.72
	MCC	0.36	0.39
Severity	macro F-score	0.44	0.58
	MCC	0.23	0.28
Average	macro F-score	0.51	0.64 ($\Uparrow$**25.5%**)
	MCC	0.20	0.33 ($\Uparrow$**26.9%**)

exhibits a consistent trend with IVDetect while outperforming it by 78.1%. F-score is lower than in Table 2 due to unbalanced data.

8.2 Comparison on Vulnerability Assessment on BigVul C Dataset (RQ2)

In Table 3, VDA relatively improves over DeepCVA [33] by *25.5% in macro F-score and 26.9% in multi-class MCC* on the overall multi-class classification. For individual VATs, VDA improves upon DeepCVA by *7.5–34.7% in macro F-score and 8.3–40.9% in multi-class MCC*. We can see that VDA consistently outperforms DeepCVA on all VAT types, thus corroborating with the design choices in Key Ideas 1–3 (see Section 2.2). Moreover, we can see that the largest relative improvement in macro F-score and multi-class MCC happens for *Access Complexity* and *Access Vector*, respectively. Such gains in performance can possibly be attributed to the nature of Access VAT, the access information for which, is more often than not, extensively checked in the changed code context, which is well represented in VDA but not in DeepCVA.

8.3 Class Separability with Code Change Embeddings (RQ3)

In this study, we aim to show that our *embeddings for code changes helps VDA have better class-separation*, i.e., *better classification measures* for detection and assessment than that of the baseline.

For each class C regarding a vulnerability assessment type (VAT), we selected 366 commits that are labeled with the class C in the oracle. This was chosen based on the population of the data, such that the sample size corresponded to a 95% confidence level, with a confidence interval of 5% for each VAT. For example, for *Confidentiality*, we randomly selected an equal number of commits (i.e., 366), that are marked as *None*, *Partial*, and *Complete*. For each type, we took those 366 × 3 = 1,098 commits and used *VDA's code change representation learning model* and *DeepCVA's n-gram-based embedding model* to produce the embeddings for those commits. We projected the embeddings from these approaches into the vector space using t-SNE [2] technique, based on which we can visualize high-dimensional data by giving each data point its projected location in a two-dimensional vector space. Next, in the silhouette plots [1], we succinctly present the data points for these embeddings, which represent how well they have been classified.

Figure 6 shows the comparison between the silhouette plots for the embeddings produced by VDA and DeepCVA regarding 3 classes of Confidentiality. Figures 6a. and c. display the t-SNE visualizations for the embeddings, while figures 6b. and d. display the silhouette plots for the data in these visualizations. The silhouette coefficient value (X-axis in Figures 6b., d., e.) is a measure of how similar an object is to its own class compared to other classes, which ranges from [-1, 1]. Here, a higher value indicates that an object is well matched to its own class and poorly matched to neighboring classes. If most objects have high values, the class configuration is proper. That corresponds to better-formed classes, facilitating a model to group the commits into the correct classes (i.e., *None, Partial,* and *Complete*). If many points have low or negative values, the class configuration is ill-formed, i.e. does not help with classification.

Let us consider the commits in the *Complete* class in Figures 6a. and b. Each line in *Complete* class in Figure 6b. corresponds to a point in *Complete* class in the vector space in Figure 6a. Length of a line is equal to the silhouette coefficient for a point. These lines are sorted from largest to smallest, and drawn from top to bottom, creating a knife shape. As seen, the knife shapes from DeepCVA have longer and thicker tails than those from VDA, which actually have no tail for the classes *Partial* and *None*. Thus, DeepCVA produces embeddings that overlap more with their neighboring classes. In Figure 6e., we place two silhouette plots in an overlay image. The plot from VDA is thicker than that from DeepCVA: VDA produces more points with positive values than DeepCVA. In brief, Figures 6a–e show that the *embeddings for code changes from VDA facilitates better classification for vulnerability assessment than the embeddings produced by DeepCVA*.

Figure 7 and Figure 8 show the comparison among the silhouette plots for the embeddings from VDA and DeepCVA on *Integrity* and *Availability* (remaining VAT types are not shown here due to space limit). We can see that the comparisons for all VATs have the same trend, i.e., the knife shapes from VDA have no or shorter tails, and are thicker than those of DeepCVA. In brief, the silhouette plots indicate that VDA produces embeddings that have more cohesion with the ones in the same class and more separation with the ones in the different classes. Thus, our *embeddings with more class-separability help VDA perform better classification*.

We also performed the same plotting for the classification task for vulnerability detection. Considering the overlap between two plots in Figure 9, the knife shapes from VDA for both classes (vulnerability and benign) are wider and have less negative values than those from the best baseline IVDetect. Specifically, the average silhouette score in VDA is 0.027, while that of IVDetect is 0.0072. Thus, VDA has better class-separability, leading to better performance.

8.4 Key Features in Program Dependencies for Classification with Explainable AI (RQ4)

We aim to evaluate whether VDA uses the vulnerable statements and their dependencies in its vulnerability prediction and assessment. This also allows us to evaluate its ability to pinpoint the

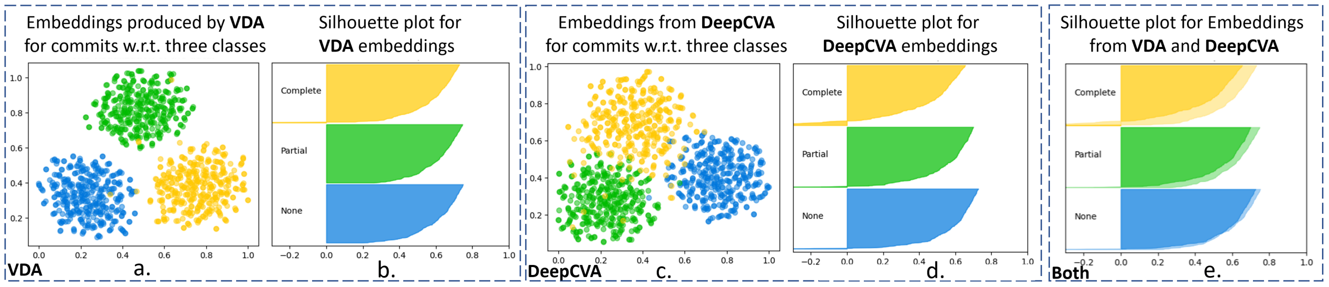


Figure 6: Silhouette Plots for the Embeddings of Commits produced by VDA and DeepCVA on CONFIDENTIALITY (RQ3)

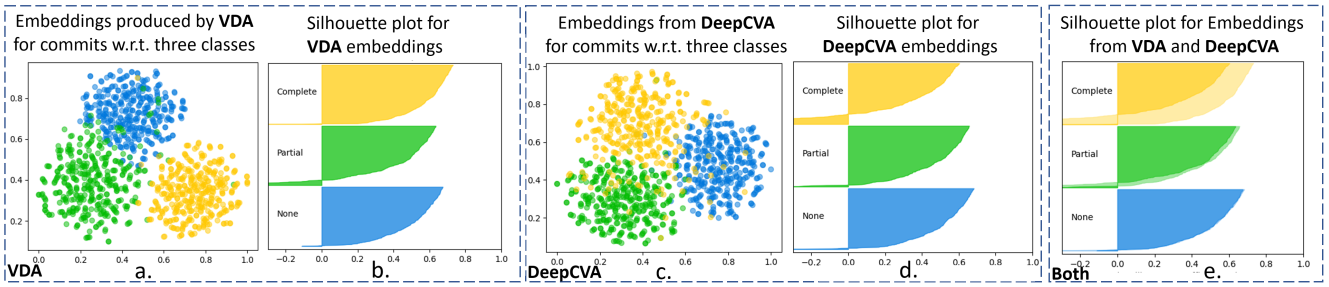


Figure 7: Silhouette Plots for the Embeddings of Commits produced by VDA and DeepCVA on INTEGRITY (RQ3)

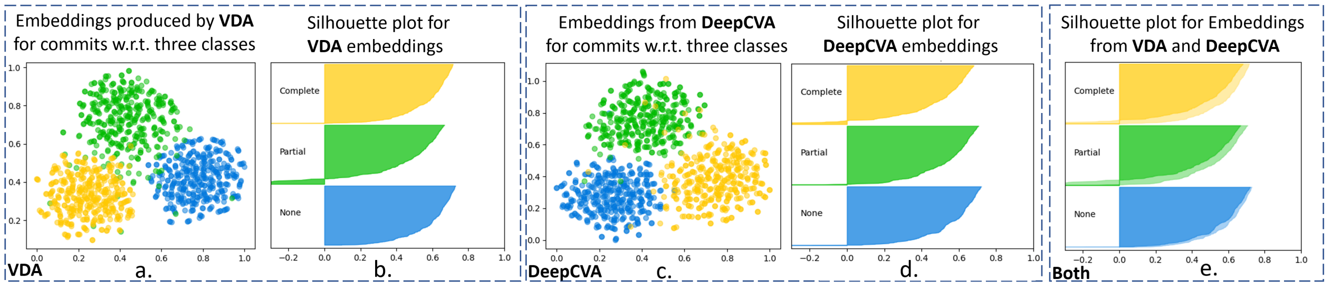


Figure 8: Silhouette Plots for the Embeddings of Commits produced by VDA and DeepCVA on AVAILABLITY (RQ3)

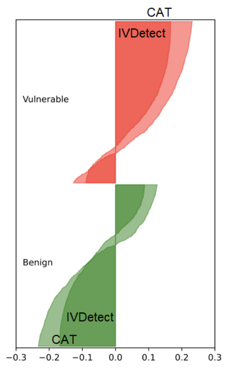


Figure 9: Silhouette Plots for the Embeddings of Commits produced by VDA and IVDetect for Vulnerability Detection: VDA has better Class Separability (RQ3)

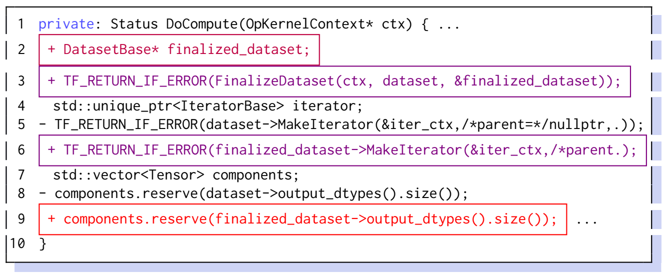
changed statements relevant to the detected vulnerability. Specifically, for each vulnerability assessment type (VAT), we randomly selected 366 samples of the commits in our dataset that VDA predicts the correct classes of the vulnerability detection and VATs (e.g., *None*, *Partial*, *Complete*). That sample size gives us the confidence level of 95% and the confidence interval of 5% for each VAT.

We use an explainable Artificial Intelligence (XAI) model, called *GNNExplainer* [47], which takes a GNN-based classification model $\mathcal{M}$ and a specific input I of $\mathcal{M}$, and produces an explanation on why $\mathcal{M}$ arrives at its prediction O for the input I. We fed VDA and each commit C of those sample commits as the input for GNNExplainer. The explanation produced by GNNExplainer is in the form of a sub-graph Δ in δ-PDG that was built from the input commit C. The sub-graph Δ is referred to as the *explanation sub-graph* for C regarding the classification of C for the current assessment type. The explanation sub-graph Δ is defined as the minimal sub-graph in the input graph δ-PDG that minimizes the prediction scores between using the entire graph δ-PDG and using Δ as the input for VDA. Δ is minimal in the sense that if any node and edge is removed from it, the decision of VDA is affected, i.e., VDA will produce a different class for the input commit C. That is, *the explanation sub-graph* Δ

Table 4: Vunerable Statements/Dependencies as Key Features

Confidence	*Integrity*	*Avail*	*AccessVec*	*AccCompl*	*Auth*	*Severity*	**Avg**
63	84	81	72	93	93	81	81.4

%Commits VDA correctly uses vulnerable statements/dependencies in Vulnerability Detection and Assessment



```
1  private: Status DoCompute(OpKernelContext* ctx) { ...
2  + DatasetBase* finalized_dataset;
3  + TF_RETURN_IF_ERROR(FinalizeDataset(ctx, dataset, &finalized_dataset));
4    std::unique_ptr<IteratorBase> iterator;
5  - TF_RETURN_IF_ERROR(dataset->MakeIterator(&iter_ctx,/*parent=*/nullptr,.));
6  + TF_RETURN_IF_ERROR(finalized_dataset->MakeIterator(&iter_ctx,/*parent.);
7    std::vector<Tensor> components;
8  - components.reserve(dataset->output_dtypes().size());
9  + components.reserve(finalized_dataset->output_dtypes().size());  ...
10 }
```

Figure 10: Contributions of Different Statements in Correct Vulnerability Prediction and Classification of VATs by VDA

contains the statements and dependencies that are most decisive for VDA to determine the class O for the commit C.

To evaluate whether VDA via Label-GCN can capture the crucial statements and dependencies in deciding the class for an input commit, we compared the explanation sub-graph Δ with the vulnerability-inducing statements and dependencies in the ground truth of those commits. If Δ contains one of such statements (nodes) and dependencies (edges), we consider that VDA uses the correct vulnerable statements and dependencies as the features for its correct classification (detection and assessment).

Table 4 shows the percentages of the cases in which VDA correctly uses the vulnerable statements and their dependencies in correctly predicting the VATs. For example, among the 366 commits that VDA successfully classified into (*None*, *Single*) for *Authentication*, GNNExplainer determines that in 93% of them, VDA uses at least one actual vulnerable statement or dependency as key features in its prediction. As seen, the degree of VDA's reliance on vulnerable statements/dependencies for its assessment across different types is different. While in 84%, VDA relies on vulnerable statements/dependencies to assess the impact of *Integrity*; in 81% samples, it relies on them for assessing that of *Severity*. For *vulnerability detection*, in 84% of samples, VDA uses the right statements/dependencies in its correct prediction (not shown). On an average, in 81.4% samples, VDA correctly relies on the vulnerable statements/dependencies. In brief, this result shows that *program dependencies among vulnerable statements are key features to VDA in its correct vulnerability detection and assessment*, which corroborates with our design choice with program dependencies.

Example. Figure 10 shows an example of the vulnerability-introducing change for CVE-2021-37650. The change introduced the variable `finalized_dataset` at line 2, representing a dataset that was populated at line 3 and used at lines 6 and 9. However, the input was not validated, and the code at line 9 assumed only string inputs and interpreted numbers as valid strings. When computing the CRC of the record, this resulted in heap buffer overflow. VDA correctly predicted this vulnerability and its assessment *Severity=Medium*. GNNExplainer pointed out that VDA used the changed statements and their dependencies at lines 2, 3, 6, 9 and another line (not shown), to perform classification. This is correct because despite line 9 being a fixed line (in a later version), lines 2, 3, 6 are parts of the control/data flows leading to line 9. Thus, VDA correctly used the vulnerability-relevant statements and dependencies in correct prediction.

Table 5: Impact of Multi-Task Learning and Context (RQ5)

Detection	VDA w/o Multi-Task	VDA w/o Context	VDA
F-score	0.68	0.70	0.76

Table 6: Impact of Multi-Task Learning and Context (RQ5)

Assessment	VDA w/o Multi-Task	VDA w/o Context	VDA
F-score	0.54	0.57	0.64

Table 7: Impact of Num. of Hops k for Context Size (RQ5)

	macro F-Score	MCC
VDA (k = 1)	0.62	0.32
VDA (k = 2)	0.63	0.33
VDA (k = 3)	0.64	0.33
VDA (k = 4)	0.62	0.31
VDA (k = 5)	0.61	0.30

8.5 Ablation Study (RQ5)

As seen in Tables 5 and 6, without multi-task learning, the performance decreases 10.5% in F-score in detection and 15.6% in macro F-score in assessment. Without context, it decreases 7.9% in F-score in detection and 10.9% in macro F-score in assessment. This result confirms our hypotheses:

(1) Multi-task learning helps improve both vulnerability detection and assessment: adding multi-task learning, both VD and VA improve (0.68 to 0.76, 0.54 to 0.64). Note: without multi-task learning, VDA also improves over the baselines (Tables 2– 3) in both VD and VA due to VDA's code change embeddings (Section 8.3).

(2) both multi-task learning and context have positive contributions, in which multi-task learning contributes more.

Multi-task learning model performs better than cascading VD to VA, which has the cascading error due to false positives in VD.

Table 7 shows the impact of the context size k (the number of hops from a changed node). As seen, when k increases from 1–3, the macro F-score increases to its highest value of 0.64. However, when k continues to increase $k \geq 3$, both macro F-score and multi-class MCC decrease. The rationale is that as the context size is too small, the limited number of surrounding nodes cannot capture well the relevant statements for assessment. As the context size gets larger, the increasing number of the irrelevant statements will bring in biases. Thus, we selected k=3 in other studies on the C dataset.

8.6 Comparative Study on Java Dataset (RQ6)

We aim to show that our approach also works for vulnerabilities in a different programming language. Table 8 shows that it relatively improves DeepCVA [33] by *31.0% in macro F-score and 33.3% in multi-class MCC* on the overall multi-class classification in Java dataset. For specific VATs, VDA improves DeepCVA by *3.0–25.6% in macro-F-score and 30.8% in multi-class MCC.* Moreover, the largest relative improvement in macro F-score happens for *Availability* and the largest one in multi-class MCC happens for *Access Vector*. The

Table 8: Vulnerability Assessment on Java Dataset (RQ6)

CVSS Metric	Evaluation Metric	Model	
		DeepCVA [33]	VDA
Confidentiality	macro F-score	0.44	0.55
	MCC	0.27	0.32
Integrity	macro F-score	0.43	0.52
	MCC	0.25	0.27
Availability	macro F-score	0.43	0.54
	MCC	0.27	0.27
Access Vector	macro F-score	0.55	0.59
	MCC	0.13	0.17
Access Complexity	macro F-score	0.46	0.53
	MCC	0.24	0.26
Authentication	macro F-score	0.66	0.68
	MCC	0.35	0.38
Severity	macro F-score	0.42	0.51
	MCC	0.21	0.22
Average	macro F-score	0.45	0.59 (⇑**31.0%**)
	MCC	0.24	0.32 (⇑**33.3%**)
Vulnerability Detection		VCCFinder	CAT
	F-score	0.24	0.76

absolute macro F-score value (0.68) for *Authentication* is highest among all the VATs. The F-score for detection from VDA is also higher than that of the commit-level VCCFinder, which uses SVM. Other baselines in Table 2 do not work on Java. In brief, this result is consistent with the trend as VDA being run on the C dataset.

8.7 Threats to Validity and Limitations

The threats come from the following aspects: (1) *Programming languages (PLs)*. Our approach has been tested on Java and C commits. However, the techniques used in VDA are not tied to Java or C. In principle, our approach can applied to other PLs. (2) *Generalization of the results.* Our comparisons with DeepCVA were only carried out on the publicly available C and Java datasets. Further comparisons with the baselines on other datasets should be done.

Our approach also has room for further improvements. First, VDA does not work well for the code changes that are common but have impacts on the far-apart, un-changed parts of the project. Second, VDA fails in the assessment for complex changes that program dependencies cannot capture, e.g., event-driven programs. Finally, the detection component could be improved further with a more dedicated model on vulnerability detection.

9 RELATED WORK

ML-based Vulnerability Prediction. Machine Learning has been applied in commit-level vulnerability detection [12, 38, 49]. VCCFinder [38] trains a SVM classifier to flag suspicious commits. We used only code change features for our experiment. Zhou and Sarma [49]'s works on commit messages and bug reports. It uses an ensemble model to combine multiple classifiers. VDA supports both vulnerability detection and assessment.

Deep learning (DL) has been applied to detect vulnerabilities [11, 22, 28–30, 35, 36, 42, 43, 45, 46, 48]. Harer *et al.* [21] leverages RNN model. Lin *et al.* [32] learns function repreentations via AST for VD. Russell *et al.* [41] combine the neural features of functions with random forest as a classifier. Harer *et al.* [20] compare the effectiveness in VD of using source code and the compiled code. VulDeePecker [50] uses a RNN trained on program slices from API calls for VD. SySeVR [31] expands VulDeePecker by including the program slices from syntactic units. Devign [48] uses Gated Graph Recurrent Layers on program graphs. Reveal [10] uses CPG with GGNN. IVDetect [29] focuses on interpretation and directly uses PDG with GCN. LineVul [17] use BigVul dataset to train a transformer-based model which has over 150K training instances. To avoid under-training of LineVul and an unfair comparison (given that it has over 110M parameters), we chose to not compare with it.

Automated Vulnerability Assessment. Distinct software vulnerabilities can have different levels of threats and severity, and require assessment [24, 27, 34]. The automated approaches have been recently proposed [6, 8, 33]. Bozorgi *et al.* [8] propose a SVM-based approach to predict whether a vulnerability will be exploited or not. Lamkanfi *et al.* [26] predict the severity of a reported bug using text mining algorithms on bug reports. Han *et al.* [19] propose a multi-class text classification DL-based model that is based on the description to predict the severity level of a vulnerability.

Georgios *et al.* [44] adopt a multi-target classification coupled with text analysis on vulnerability descriptions to predict their characteristics and scores. Le *et al.* [27] propose a ML-based approach to learn the word features in vulnerability description, and handle the extended concepts in the description. Other studies [39, 40] leverage code patterns in fixing commits of third-party libraries to assess vulnerabilities. In comparison, VDA supports commit-level vulnerability detection and assessment using code changes.

Code Change Embeddings. Our work is also related to code change embedding models [9, 23]. Those approaches mainly treat code as sequences and do not consider structures and/or program dependencies. The key departure points of VDA include the use of graph representation to model the changes and dependencies, as well as the surrounding context to build the embeddings.

10 CONCLUSION

This paper proposes VDA, a Context-aware, Graph-based, Commit-level Vulnerability Detection and Assessment Model that evaluates a commit, detects any vulnerability and provides the CVSS assessment grades. The key advances in VDA over the existing approaches include 1) multi-task learning between vulnerability detection and assessments of different aspects, 2) code change embedding model that integrates program dependencies and contexts, and 3) graph-based representations of dependencies and contexts. Our evaluation shows that on a C dataset, VDA achieves 25.5% and 26.9% relatively higher than the baselines in vulnerability assessment in F-score and MCC. In a Java dataset, VDA achieves 31% and 33.3% relatively higher in F-score and MCC. VDA also relatively improves the vulnerability detection over the baselines from 13.4–322% in F-score.

11 DATA AVAILABILITY

Our data and code are available at [5].

ACKNOWLEDGMENTS

This work was supported in part by the US NSF grant CNS-2120386 and the NSA grant NCAE-C-002-2021.

REFERENCES

[1] [n. d.]. Silhouette (clustering). https://en.wikipedia.org/wiki/Silhouette_(clustering). Last Accessed March 15, 2022.

[2] [n. d.]. T-SNE. https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html. Last Accessed March 9, 2022.

[3] 2021. Common Vulnerability Scoring System. https://www.first.org/cvss/

[4] 2021. *The NNI autoML tool*. https://github.com/microsoft/nni

[5] 2023. CAT. https://github.com/vulnerability-assessment-cat/vulnerability-assessment-cat

[6] Luca Allodi and Fabio Massacci. 2014. Comparing Vulnerability Severity and Exploits Using Case-Control Studies. *ACM Trans. Inf. Syst. Secur.* 17, 1, Article 1 (aug 2014), 20 pages. https://doi.org/10.1145/2630069

[7] Claudio Bellei, Hussain Alattas, and Nesrine Kaaniche. 2021. Label-GCN: An Effective Method for Adding Label Propagation to Graph Convolutional Networks. *CoRR* abs/2104.02153 (2021). arXiv:2104.02153 https://arxiv.org/abs/2104.02153

[8] Mehran Bozorgi, Lawrence K. Saul, Stefan Savage, and Geoffrey M. Voelker. 2010. Beyond Heuristics: Learning to Classify Vulnerabilities and Predict Exploits. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (Washington, DC, USA) *(KDD '10)*. Association for Computing Machinery, New York, NY, USA, 105–114. https://doi.org/10.1145/1835804.1835821

[9] Rocío Cabrera Lozoya, Arnaud Baumann, Antonino Sabetta, and Michele Bezzi. 2021. Commit2Vec: Learning Distributed Representations of Code Changes. *SN Comput. Sci.* 2, 3 (mar 2021), 16 pages. https://doi.org/10.1007/s42979-021-00566-z

[10] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2020. Deep Learning based Vulnerability Detection: Are We There Yet? *arXiv preprint arXiv:2009.07235* (2020).

[11] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray. 2022. Deep Learning Based Vulnerability Detection: Are We There Yet? *IEEE Transactions on Software Engineering* 48, 09 (sep 2022), 3280–3296. https://doi.org/10.1109/TSE.2021.3087402

[12] Xiang Chen, Yingquan Zhao, Zhanqi Cui, Guozhu Meng, Yang Liu, and Zan Wang. 2020. Large-Scale Empirical Studies on Effort-Aware Security Vulnerability Prediction Methods. *IEEE Transactions on Reliability* 69, 1 (2020), 70–87. https://doi.org/10.1109/TR.2019.2924932

[13] R. Cipolla, Y. Gal, and A. Kendall. 2018. Multi-task Learning Using Uncertainty to Weigh Losses for Scene Geometry and Semantics. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 7482–7491. https://doi.org/10.1109/CVPR.2018.00781

[14] Davide Falessi, Jacky Huang, Likhita Narayana, Jennifer Fong Thai, and Burak Turhan. 2020. On the Need of Preserving Order of Data When Validating Within-Project Defect Classifiers. *Empirical Softw. Engg.* 25, 6 (nov 2020), 4805–4830. https://doi.org/10.1007/s10664-020-09868-x

[15] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*. Association for Computing Machinery, New York, NY, USA, 508–512. https://doi.org/10.1145/3379597.3387501

[16] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (jul 1987), 319–349. https://doi.org/10.1145/24039.24041

[17] Michael Fu and Chakkrit Tantithamthavorn. 2022. LineVul: A Transformer-based Line-Level Vulnerability Prediction. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. 608–620. https://doi.org/10.1145/3524842.3528452

[18] J. Gorodkin. 2004. Comparing Two K-Category Assignments by a K-Category Correlation Coefficient. *Comput. Biol. Chem.* 28, 5–6 (dec 2004), 367–374. https://doi.org/10.1016/j.compbiolchem.2004.09.006

[19] Zhuobing Han, Xiaohong Li, Zhenchang Xing, Hongtao Liu, and Zhiyong Feng. 2017. Learning to Predict Severity of Software Vulnerability Using Only Vulnerability Description. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 125–136. https://doi.org/10.1109/ICSME.2017.52

[20] Jacob A Harer, Louis Y Kim, Rebecca L Russell, Onur Ozdemir, Leonard R Kosta, Akshay Rangamani, Lei H Hamilton, Gabriel I Centeno, Jonathan R Key, Paul M Ellingwood, et al. 2018. Automated software vulnerability detection with machine learning. *arXiv preprint arXiv:1803.04497* (2018).

[21] Jacob A. Harer, Onur Ozdemir, Tomo Lazovich, Christopher P. Reale, Rebecca L. Russell, Louis Y. Kim, and Peter Chin. 2018. Learning to Repair Software Vulnerabilities with Generative Adversarial Networks. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 7944–7954.

[22] David Hin, Andrey Kan, Huaming Chen, and M. Ali Babar. 2022. LineVD: Statement-Level Vulnerability Detection Using Graph Neural Networks. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 596–607. https://doi.org/10.1145/3524842.3527949

[23] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. CC2Vec: Distributed Representations of Code Changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 518–529. https://doi.org/10.1145/3377811.3380361

[24] Saad Khan and Simon Parkinson. 2018. Review into state-of-the-art of vulnerability assessment using artificial intelligence. In *Guide to Vulnerability Analysis for Computer Networks and Systems*. Springer, 3–32.

[25] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=SJU4ayYgl

[26] Ahmed Lamkanfi, Serge Demeyer, Emanuel Giger, and Bart Goethals. 2010. Predicting the severity of a reported bug. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. 1–10. https://doi.org/10.1109/MSR.2010.5463284

[27] Triet Huynh Minh Le, Bushra Sabir, and M. Ali Babar. 2019. Automated Software Vulnerability Assessment with Concept Drift. In *Proceedings of the 16th International Conference on Mining Software Repositories* (Montreal, Quebec, Canada) *(MSR '19)*. IEEE Press, 371–382. https://doi.org/10.1109/MSR.2019.00063

[28] Xin Li, Lu Wang, Yang Xin, Yixian Yang, and Yuling Chen. 2020. Automated vulnerability detection in source code using minimum intermediate representation learning. *Applied Sciences* 10, 5 (2020), 1692.

[29] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Vulnerability Detection with Fine-Grained Interpretations. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, New York, NY, USA, 292–303. https://doi.org/10.1145/3468264.3468597

[30] Z. Li, D. Zou, S. Xu, Z. Chen, Y. Zhu, and H. Jin. 2022. VulDeeLocator: A Deep Learning-Based Fine-Grained Vulnerability Detector. *IEEE Transactions on Dependable and Secure Computing* 19, 04 (jul 2022), 2821–2837. https://doi.org/10.1109/TDSC.2021.3076142

[31] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen. 2022. SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities. *IEEE Transactions on Dependable and Secure Computing* 19, 04 (jul 2022), 2244–2258. https://doi.org/10.1109/TDSC.2021.3051525

[32] Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, and Yang Xiang. 2017. POSTER: Vulnerability Discovery with Function Representation Learning from Unlabeled Projects. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2539–2541. https://doi.org/10.1145/3133956.3138840

[33] Triet Huynh Minh Le, David Hin, Roland Croft, and M. Ali Babar. 2021. DeepCVA: Automated Commit-level Vulnerability Assessment with Deep Multi-task Learning. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 717–729. https://doi.org/10.1109/ASE51524.2021.9678622

[34] Kartik Nayak, Daniel Marino, Petros Efstathopoulos, and Tudor Dumitraş. 2014. Some vulnerabilities are different than others. In *International Workshop on Recent Advances in Intrusion Detection*. Springer, 426–446.

[35] Stephan Neuhaus and Thomas Zimmermann. 2009. The Beauty and the Beast: Vulnerabilities in Red Hat's Packages. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference* (San Diego, California) *(USENIX'09)*. USENIX Association, USA, 30.

[36] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. 2007. Predicting Vulnerable Software Components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) *(CCS '07)*. Association for Computing Machinery, New York, NY, USA, 529–540. https://doi.org/10.1145/1315245.1315311

[37] Profir-Petru Pârundefinedachi, Santanu Kumar Dash, Miltiadis Allamanis, and Earl T. Barr. 2020. Flexeme: Untangling Commits Using Lexical Flows. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 63–74. https://doi.org/10.1145/3368089.3409693

[38] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 2015. VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 426–437. https://doi.org/10.1145/2810103.2813604

[39] S. Ponta, H. Plate, and A. Sabetta. 2018. Beyond Metadata: Code-Centric and Usage-Based Analysis of Known Vulnerabilities in Open-Source Software. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE Computer Society, Los Alamitos, CA, USA, 449–460. https://doi.org/10.1109/ICSME.2018.00054

[40] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. 2020. Detection, Assessment and Mitigation of Vulnerabilities in Open Source Dependencies. *Empirical Softw. Engg.* 25, 5 (sep 2020), 3175–3215. https://doi.org/10.1007/s10664-020-09830-x

[41] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability

detection in source code using deep representation learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 757–762.

[42] Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. 2014. Predicting Vulnerable Software Components via Text Mining. *IEEE Transactions on Software Engineering* 40, 10 (2014), 993–1006. https://doi.org/10.1109/TSE.2014.2340398

[43] 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. https://doi.org/10.1109/TSE.2010.81

[44] Georgios Spanos and Lefteris Angelis. 2018. A multi-target approach to estimate software vulnerability characteristics and severity scores. *Journal of Systems and Software* 146 (2018), 152–166. https://doi.org/10.1016/j.jss.2018.09.039

[45] 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*. 13–13.

[46] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. 2012. Generalized Vulnerability Extrapolation Using Abstract Syntax Trees. In *Proceedings of the 28th Annual Computer Security Applications Conference* (Orlando, Florida, USA) *(ACSAC '12)*. Association for Computing Machinery, New York, NY, USA, 359–368. https://doi.org/10.1145/2420950.2421003

[47] Rex Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. 2019. *GNNExplainer: Generating Explanations for Graph Neural Networks*. Curran Associates Inc., Red Hook, NY, USA.

[48] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. *Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks*. Curran Associates Inc., Red Hook, NY, USA.

[49] Yaqin Zhou and Asankhaya Sharma. 2017. Automated Identification of Security Issues from Commit Messages and Bug Reports. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 914–919. https://doi.org/10.1145/3106237.3117771

[50] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin. 2021. VulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *IEEE Transactions on Dependable and Secure Computing* 18, 05 (sep 2021), 2224–2236. https://doi.org/10.1109/TDSC.2019.2942930

Received 2023-02-02; accepted 2023-07-27