Fine-grained Commit-level Vulnerability Type Prediction by CWE Tree Structure

Shengyi Pan*, Lingfeng Bao*§, Xin Xia†, David Lo‡, Shanping Li*
*College of Computer Science and Technology, Zhejiang University, China
†Huawei, China
‡School of Information Systems, Singapore Management University, Singapore
{shengyi.pan,lingfengbao,shan}@zju.edu.cn, xin.xia@acm.org, davidlo@smu.edu.sg

***Abstract*—Identifying security patches via code commits to allow early warnings and timely fixes for Open Source Software (OSS) has received increasing attention. However, the existing detection methods can only identify the presence of a patch (i.e., a binary classification) but fail to pinpoint the vulnerability type. In this work, we take the first step to categorize the security patches into fine-grained vulnerability types. Specifically, we use the Common Weakness Enumeration (CWE) as the label and perform fine-grained classification using categories at the third level of the CWE tree. We first formulate the task as a Hierarchical Multi-label Classification (HMC) problem, i.e., inferring a path (a sequence of CWE nodes) from the root of the CWE tree to the node at the target depth. We then propose an approach named TreeVul with a hierarchical and chained architecture, which manages to utilize the structure information of the CWE tree as prior knowledge of the classification task. We further propose a tree structure aware and beam search based inference algorithm for retrieving the optimal path with the highest merged probability. We collect a large security patch dataset from NVD, consisting of 6,541 commits from 1,560 GitHub OSS repositories. Experimental results show that TreeVul significantly outperforms the best performing baseline, with an improvement of 5.9%, 25.0%, and 7.7% in terms of weighted F1-score, macro F1-score, and MCC, respectively. We further conduct a user study and a case study to verify the practical value of TreeVul in enriching the binary patch detection results and improving the data quality of NVD, respectively.**

***Index Terms*—Software Security, Vulnerability Type, CWE**

I. Introduction

Nowadays, software products heavily rely on open-source softwares (OSS). The rapidly increasing number of software vulnerabilities (SV) poses a significant threat to OSS. The National Vulnerability Database (NVD) recorded 20,061 new SVs in 2021, a 9.3% increase over the prior year [1].

Security patches play a crucial role in the SV remediation. Developers can apply the patch to fix an SV and estimate its impact (e.g., identify the affected software components) [2]. In practice, an SV is often (almost 70% in OSS [3]) publicly disclosed with relevant information (e.g., patches and metadata) after it has been patched for a while (as suggested by the coordinated disclosure policy [4]). Such a delay creates a *window of opportunity* for attackers, as given the public nature of OSS, attackers could probe for SVs by monitoring development activities, while downstream users may remain unaware and not take any mitigation [3], [5]. Thus, many approaches have been proposed to detect new SV fixes committed to OSS codebases, aiming to help OSS users be aware of patches in time, so that they can start remediation earlier [6]–[9].

However, the existing approaches only detect the existence of an SV-fixing commit (i.e., a binary classification of fixing or non-fixing), but do not provide any following analysis (e.g., type). Applying patches is not straightforward, e.g., the OSS components have usually been customized to meet certain requirements of the production system [10], requiring prioritization based on the assessment of the SV severity [11]. To guide the following remediation process, it is necessary to first analyze the type of the detected patch [11]–[13], which presents an overview of the SV that the patch is targeted to address. SV Type is leveraged by the security community as a standard (i.e., CWE [14]) to understand the root cause, possible impact, and types of mitigation to deploy. Well-known SV databases (e.g., NVD [1], vulnDB [15]) all adopt the CWE classification to provide users with essential insights of SVs.

Although the existing detection methods can automatically filter security patches, it still requires extensive manpower to analyze and categorize numerous detected patches. Sometimes, it may be difficult and time-consuming for practitioners to understand and analyze the detected patch, e.g., lacking experience with certain SVs, checking patches with lots of noise (i.e., non fixing-related code changes) [2]. The process of manual analysis also brings considerable delay before taking effective mitigation, compromising the value of early remediation.

Thus, we aim to propose an automated approach to further classify the detected patches into fine-grained SV types. With the insights illustrated by the SV type, practitioners can better analyze the detected patch (see Section II-B for a motivating example). Additionally, practitioners can retrieve well-summarized SV information (e.g., consequences, mitigation) on the CWE website using type as an index. This information can ease the following remediation process, e.g., taking temporary mitigation, assessing the severity. Moreover, as shown in Section VI-A, fine-grained SV types provide more actionable feedback and better help the practitioners.

The automated SV type analysis can also help to improve the quality of CWE metadata provided by NVD. NVD assigns each *Common Vulnerabilities and Exposures* (CVE) record with a CWE identifier to provide users with an overview of the SV nature and risk [16]. However, (1) The manual analysis of

[§] Corresponding author.

SV metadata brings delay to its publishment after disclosure. The timeliness of such analysis is important, as practitioners rely on these metadata to assess and react to new SVs [17]. (2) The CWE metadata suffers from quality issues. Roughly 31% of all CVE records miss valid CWE metadata (e.g., *NVD-CWE-Other*) [12]. We observe a new quality issue, i.e., part of the assigned CWE categories are not precise enough. We show in our preliminary study (see Section II-C) that 21% of the assigned CWE categories are at depth 1 or 2, which does not follow the good mapping practice suggested by CWE [18], i.e., mapping to the lowest entry in the CWE tree hierarchy (see Figure 1). These observations present the challenges NVD faces in providing in-time and high-quality analysis against the skyrocketing number of SVs, as well as the value of adopting automated approaches to reduce human efforts and biases on manual SV analysis (see Section VI-B for a case study).

In this study, we propose an approach named TreeVul to classify security patches into fine-grained SV types. TreeVul manages to leverage the structure information of the CWE tree as a prior knowledge of the classification task, aiming to improve the performance. Specifically, we first formulate the task as an Hierarchical Multi-label Classification (HMC) problem, i.e., inferring a path (a sequence of CWE nodes) from the root of the CWE tree to the target node (see Figure 1 for an example). Then, we design a hierarchical and chained model architecture, which exploits the internal relations between the classification tasks at different depths of the CWE tree. Finally, based on the defined *path probability*, we propose a tree-structure aware and beam-search based inference algorithm. Besides, we also propose to combine both coarse (hunk-level) and fine grained (token-level) change information for encoding commits. We collect a large and up-to-date security patch dataset from NVD, consisting of 6,541 commits from 1,560 GitHub OSS repositories. We conduct evaluations under the task of predicting categories at the *third* level of the CWE tree using our collected dataset. Experimental results show that TreeVul significantly outperforms the baselines, with an improvement of 5.9%, 25.0%, and 7.7% in terms of weighted F1-score, macro F1-score, and MCC, respectively. We further conduct a user study and a case study to verify the practical value of TreeVul in enhancing the early remediation workflow and improving the data quality of NVD, respectively. In summary, our contributions can be summarized as follows:

- To the best of our knowledge, we are the first to introduce the task of classifying security patches into fine-grained SV types. We collect a large and up-to-date patch dataset from NVD, consisting of 6,541 commits from 1,560 GitHub OSS. We label patches with categories at the third level of the CWE tree. We provide a replication package of our dataset and the proposed approach, which is available at [19].
- We formulate the task as an HMC problem and propose an automated approach named TreeVul. TreeVul has a hierarchical and chained architecture to incorporate the knowledge of the CWE tree structure.
- Experimental results show that TreeVul outperforms baselines substantially.

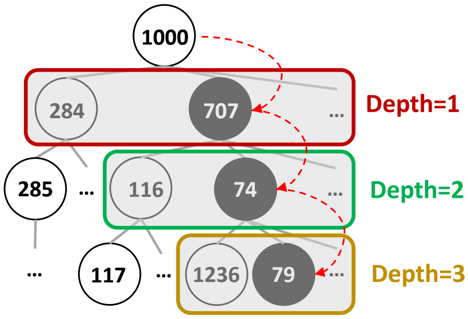


Fig. 1: Inference of the CWE path from the root node (CWE-1000) to the target node (CWE-79). Nodes out of the rectangles are pruned based on the parent node.

II. Motivation and Preliminaries

In this section, we first introduce the background of our paper. Then, we provide a motivating example, and conduct a preliminary study on CWE categories.

A. Background

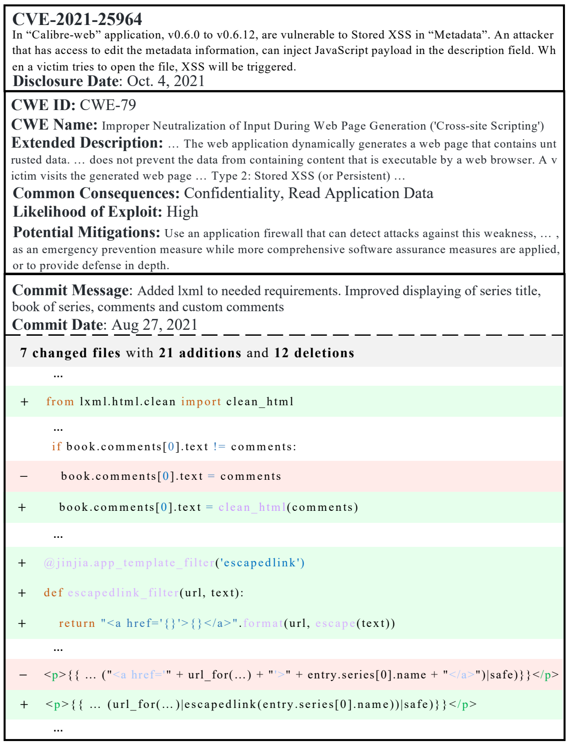
Here, we describe some important terms used in our study: **Hierarchical Multi-label Classification (HMC)** Multi-label classification deals with tasks where each sample x is associated with a set of lables Y, where $Y \subseteq L$. In a Hierarchical Multi-label Classification (HMC) task, the labels are required to be ordered in a predefined structure (e.g., tree) [20].

Common Weakness Enumeration (CWE) is a list of common software and hardware weakness types [21]. Each CWE entry represents a single SV type. CWE entries are organized in a tree hierarchy of multiple levels of abstraction (Figure 1). There are three views of CWE hierarchies available. VIEW-1000 (Research Concepts) [22] is for research of inter-dependencies between CWE entries. Both VIEW-699 (Software Development) [23] and VIEW-1194 (Hardware Design) [24] organize entries from development perspectives. Specifically, VIEW-699 is only 2-levels deep, with the top level containing categories of developer-friendly concepts (e.g., API/Function Errors), which should not be mapped but only help developers quickly navigate [18]. In this study, we focus on VIEW-1000, which is also adopted by previous works [25]–[27]. Specifically, TreeVul utilizes VIEW-1000's deep tree structure to perform the top-down search from abstract categories to specific ones.

B. Motivating Example

Considering an application scenario where OSS users apply the existing patch detection techniques [6], [7] to monitor the new fixes committed to target OSS codebases, and checks the detected patches regularly, to facilitate early remediation. The user may be interested or assigned to first check common and dangerous SVs, such as *cross-site scripting (XSS)* (CWE-79 [28]). Automated SV type analysis is essential to support such an application scenario, which groups similar SVs to facilitate management and the reuse of known security practices.

Moreover, SV type can help users analyze the detected patches. Figure 2 presents an example from CVE-2021-25964 [29]. This SV was fixed over one month before being



CVE-2021-25964
In "Calibre-web" application, v0.6.0 to v0.6.12, are vulnerable to Stored XSS in "Metadata". An attacker that has access to edit the metadata information, can inject JavaScript payload in the description field. When a victim tries to open the file, XSS will be triggered.
Disclosure Date: Oct. 4, 2021

CWE ID: CWE-79
CWE Name: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
Extended Description: … The web application dynamically generates a web page that contains untrusted data. … does not prevent the data from containing content that is executable by a web browser. A victim visits the generated web page … Type 2: Stored XSS (or Persistent) …
Common Consequences: Confidentiality, Read Application Data
Likelihood of Exploit: High
Potential Mitigations: Use an application firewall that can detect attacks against this weakness, … , as an emergency prevention measure while more comprehensive software assurance measures are applied, or to provide defense in depth.

Commit Message: Added lxml to needed requirements. Improved displaying of series title, book of series, comments and custom comments
Commit Date: Aug 27, 2021

7 changed files with **21 additions** and **12 deletions**

```
      ...
+     from lxml.html.clean import clean_html
      ...
      if book.comments[0].text != comments:
−       book.comments[0].text = comments
+       book.comments[0].text = clean_html(comments)
      ...
+     @jinjia.app_template_filter('escapedlink')
+     def escapedlink_filter(url, text):
+       return "<a href='{}'>{}</a>".format(url, escape(text))
      ...
−     <p>{{ ... ("<a href='" + url_for(...) + "'>" + entry.series[0].name + "</a>")|safe)}}</p>
+     <p>{{ ... (url_for(...)|escapedlink(entry.series[0].name))|safe)}}</p>
      ...
```

Fig. 2: A motivating example from CVE-2021-25964

publicly disclosed in NVD. The patch detection techniques can early inform OSS users of the occurrence of such a commit. However, before taking further remediation, users still need to manually analyze this commit to identify the type of SV that is fixed, understand the logic of code revisions and assess the severity. It can be difficult for OSS users who lack skills and experience to understand this patch [30] at the first glance, let alone making further analysis. According to the commit message, this commit is about feature enhancing (i.e., improving the display of the website). The major code changes within this commit are about cleaning the overburdened web page using `LXML` [31]. The hunks that related to SV-fixing are buried in the middle of this large commit (i.e., adding *escapedlink* to filter invalid inputs in the *name* attribute). Such a commit can confuse the OSS users.

However, if OSS users are informed that this commit fixes a SV of *XSS*, they can refer to the typical causes of such SVs, thus locating the hunks that are related to the SV-fixing and understanding the rationale behind code changes more quickly. Also, considering it is an *XSS* SV in the web application, OSS users will likely prioritize its remediation as such SVs can be easily exploited by attackers. Additionally, OSS users can retrieve the well-summarized SV information from CWE website based on the predicted SV type, which benefits further analysis: First, they can know the typical behaviours and causes of this type of SV. Second, CWE summarizes the common consequences of the SV, which can help OSS users quickly assess the potential impacts (e.g., exploitability), thus deciding the priorities of fixing it. Third, CWE usually gives suggestions to mitigate this type of SV. For instance, the webpage of CWE-79 provides several potential mitigations, e.g., using an application firewall. Thus, OSS users can take temporary actions to remediate this SV as the official release with the security patches is not ready.

C. Preliminary Study

We perform a preliminary study to understand the characteristics of CWE categories in NVD. Based on the observations, we find a new type of quality issue that part of the assigned CWE categories are not precise enough (i.e., classified at a coarse-grained level). We also explain the motivation of leveraging the CWE tree structure in fine-grained classification.

We get 8,275 security patches after preprocessing our collected data (see Section IV-A). These security patches are classified at different depths of the CWE tree (see the distribution of the depth of CWE categories in Figure 3.) A large number of security patches (i.e., 1,734) are classified at a coarse-grained level, i.e., depth 1 or 2. Among them, 86 security patches are even classified at depth-1, which is the most abstract type of weakness in CWE. The remaining 6,541 security patches are classified at a more fine-grained level, i.e., the CWE category is located at depth$\geq$3.

The official mapping guidance of CWE encourages analysts to map to the lowest-level CWE entry (i.e., as specific as possible), since precise mappings offer better-quality data and help to coalesce community standards [32]. However, CWE category of each CVE record is assigned by different security experts, whose skill and experience differences can introduce biases. The granularity of the classification directly affects the usefulness of the type information, as well as the practical value of the automated tool. The fine-grained classification provides developers with more detailed information, putting them in a better position to mitigate risks most effectively [18].

Therefore, we aim to propose an automated approach to classify security patches at a fine-grained level. Moreover, such an approach can also be used to fill the fine-grained CWE categories for the existing CVE records, improving the overall data quality (see the case study in Section VI-B). Specifically, we choose to assign CWE categories at depth-3. Although CWE suggests practitioners to assign the lowest category they can, only a few practitioners prefer or are capable of mapping at depth>3 [18]. We rely on mappings conducted by NVD analysts to supervise the training of TreeVul. Besides, some depth-3 categories (e.g., CWE-1236) do not have children.

However, the fine-grained classification also brings the following challenges: The 6,541 security patches belong to 78 CWE categories (at depth-3). Additionally, the distribution of these CWE categories in the dataset is highly imbalanced. CWE-119 has the most instances (i.e., 1,641), while several categories have less than 10 instances (e.g., CWE-838). Thus, the task of classifying these security patches into 78 categories is extremely challenging.

Inspired by the good mapping practice, we observe some opportunities in utilizing the CWE tree structure information to tackle this challenging task. In practice, as suggested by CWE [18], analysts should navigate the hierarchy to understand the relation between weaknesses and perform a top-down search. Specifically, the model should exploit the relation

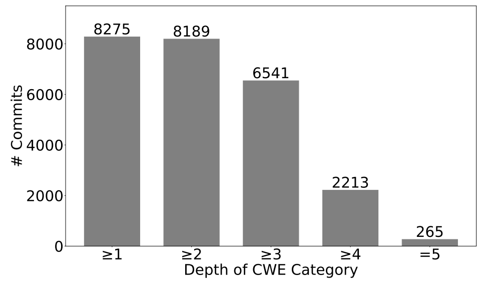


Fig. 3: Distribution of the depth of CWE categories

between categories defined by the CWE tree and follow the same top-down way to infer the fine-grained CWE categories (at the lower levels of the tree). That is, we start from the root CWE node, progressively predicting one CWE node at the lower level until reaching the target depth (see Figure 1). At the coarse-grained level, classification is considerably easier with fewer categories (the number of CWE categories at depth 1 and 2 in our dataset are 7 and 28, respectively) and more generic concepts. Thus, we can utilize the classification results of the upper levels to ease the difficulties of the lower ones, i.e., prune invalid branches from the CWE tree. Furthermore, since the classification tasks at different levels of the CWE tree are highly correlated (i.e., classify at different granularities), we can benefit from the multitask learning paradigm, i.e., training related tasks simultaneously using a shared model, which has been proven effective for various applications [33].

These observations motivate us to recast the classification task of fine-grained type prediction into a Hierarchical Multi-label Classification (HMC) task, and further incorporate the information of CWE tree structure as prior knowledge of the task to build the prediction model.

III. Approach

In this section, we introduce our approach, named TreeVul. We first define the task of identifying the fine-grained CWE category for security patches as an HMC problem. We then present the details of our proposed model. Finally, we introduce the steps of inference.

A. Task Definition

This work aims to automatically predict the fine-grained CWE category for an input security patch (i.e., a GitHub code commit in this work). Since the CWE categories are organized in a tree-like hierarchy, the abstraction level of the predicted CWE category is decided by the given depth d, i.e., predicting CWE category y_d at the d^{th} level of the tree ($y_d \in L_d$). Specially, for a given commit c, we do not directly map it to y_d as a typical multi-class classification task. Instead, we formulate this task as an HMC problem (see Section II-A), i.e., map c to a sequence of labels $(y_1, \ldots, y_d)$ which should align with the CWE tree structure (motivations are discussed in Section II-C). The goal of this work is to find a method F:

$$\begin{aligned} & F(c,d) = Y \\ s.t., \; & Y = (y_1, \ldots, y_d), \forall i \in \{1, \ldots, d\}, \\ & y_i \in L_i \; and \; y_i \in children(y_{i-1}) \end{aligned} \tag{1}$$

F takes the commit c and the target depth d (controls the granularity of the predicted category) as inputs and outputs a sequence of d CWE categories, denoted as Y. Y should be a valid *path* starting from the root node of the CWE tree to the node at the target depth d, i.e., each y_i should be one of the CWE categories at depth i ($y_i \in L_i$) and one of the children of the pre-node y_{i-1}.

B. Model Architecture

We propose an automated approach, named TreeVul, to tackle the HMC problem defined in Section III-A. The overview of TreeVul is presented in Figure 4. Generally, it is composed of a shared *commit embedding module*, together with d *depth-specific prediction heads*. Each head is for the classification task at depth i, i.e., $f_i(c) = y_i$. These tasks are highly co-related, which motivates us to train them simultaneously with a shared model using multi-task learning [33]. Besides, we organize the depth-specific prediction heads in a hierarchy and apply chain classifiers (i.e., the output of the parent classifier is used as input to the child classifier) [34].

Compared with a model that directly maps the commit c into y_d (CWE category at the target depth), the design of TreeVul under the HMC setting brings the following benefits:

- **TreeVul leverages the structure information of the classification schema (i.e., CWE tree).** CWE tree illustrates the expert-refined relations (i.e., similarities and differences) between various weaknesses. Thus, its structure information is valuable prior knowledge for the classification task.
- **TreeVul fully utilizes the label information to provide hierarchical supervision during the training.** TreeVul not only uses the CWE category at the target depth y_d, but also its ancestors $\{y_1, \ldots, y_{d-1}\}$ along the path. The model is trained in a hierarchical way to introduce an inductive bias by supervising low-level tasks at the bottom layers and more complex ones at the top layers [35].
- **TreeVul explicitly exploits the relation of the CWE category with its ancestors in the hierarchy.** The category at each depth y_i should align with the parent to conform to the CWE tree structure. TreeVul explicitly incorporates the information from the parent using chain classifiers, which benefits the lower-level classifications (invalid branches can be pruned based on the parent node).

1) Commit Embedding Module: The commit embedding module is used to represent the input commit. The implementation of this module can be arbitrary as long as it takes a commit as input and outputs its embedding. We implement our embedding module based on CodeBERT [36], a Transformer-based pretrained language model, which has been proven effective in the latest work regarding security patch detection [6]. First, the hunk-level code changes are extracted from the `diff` file. Then, the *rem-code* and *add-code* segments from the extracted code changes are tokenized using the CodeBERT tokenizer, separately. The CodeBERT is naturally (transformer-based) a powerful encoder for a pair of sequences. The input is constructed as `[CLS]` *rem-code* `[SEP]` *add-code* `[EOS]` (see Figure 4). The `[CLS]` and the

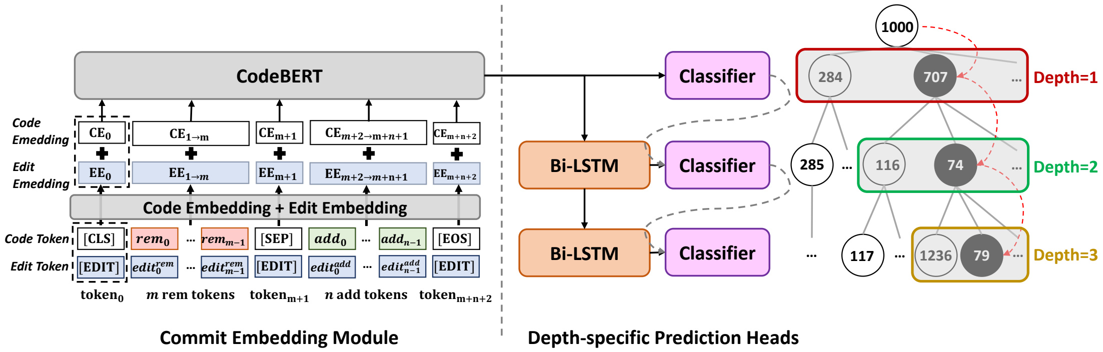


Fig. 4: Overview of TreeVul (with the target depth set to 3)

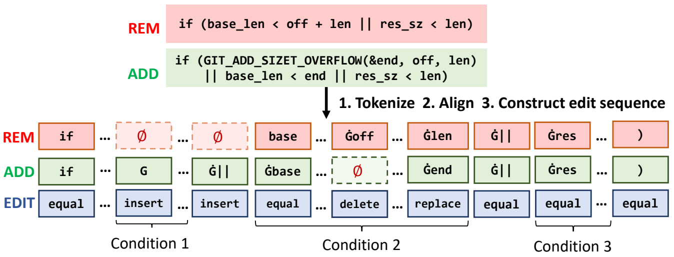

REM
ADD
if (base_len < off + len || res_sz < len)
if (GIT_ADD_SIZET_OVERFLOW(&end, off, len)
|| base_len < end || res_sz < len)
1. Tokenize 2. Align 3. Construct edit sequence
REM
ADD
EDIT
if
∅
∅
base
Ġoff
Ġlen
Ġ||
Ġres
)
if
G
Ġ||
Ġbase
∅
Ġend
Ġ||
Ġres
)
equal
insert
insert
equal
delete
replace
equal
equal
equal
Condition 1
Condition 2
Condition 3

Fig. 5: Construct token-level code changes. This patch (CVE-2018-10887) updates the *if* statement by adding condition 1, updating condition 2, and leaving condition 3 unchanged.

`[EOS]` are special tokens used to represent the start and end of the input sample, respectively. The `[SEP]` token is used to separate the pair of *rem-code* and *add-code*. Such input is constructed at the **hunk-level**, as the model is only explicitly informed that the sequence before the middle `[SEP]` token is replaced by the one after in the updated code segments.

We argue that the model can benefit from more fine-grained change information. The **token-level** change information is proven effective in the task of just-in-time comment updating [37], [38], which also requires to capture the semantics of code change. We believe the token-level change information is also effective regarding security patch classification, as security patches often introduce changes on operators, operands, and condition statements [39], [40]. For example, typical fixes for SVs regarding *Improper Restrictions of Buffer Boundary* (CWE-119 [41]) include changing an operator $<$ into $\leq$ or an operand n into $n-1$. Thus, we further introduce the **token-level** change information as an extra input to the CodeBERT. The implementation consists of two steps:

❶ **Extract token-level change information.** Figure 5 presents the overview of this step. We use the approach proposed by Liu *et al.* [37] to extract the token-level change information. First, the extracted *rem-code* and *add-code* sequences are tokenized using CodeBERT tokenizer. Then, the token sequences are automatically aligned using a `diff` tool [42] and some heuristics following Liu *et al.* [37]. Finally, the token-level change (i.e., edit) can be inferred from the pair $\langle rem_i, add_i \rangle$. There are four types of edits, i.e., `equal`, `replace`, `insert` and `delete`. Figure 5 presents an example of security patch for CVE-2018-10887 [43], which is an *Out-of-bounds Read* (CWE-125 [44]) SV. The patch modifies the `If` statement by adding a new condition, updating an existing one, and leaving the left one unchanged. Explicitly providing the token-level change information can help the model better capture and focus on these change details.

❷ **Incorporate the token-level change information into CodeBERT.** Supplying CodeBERT with token-level change information is implemented by adding an additional edit embedding for each input token (See Figure 4). Specifically, based on the constructed edit sequence, the input token becomes a tuple of code token (i.e., the original CodeBERT token) and the corresponding edit token. We add an extra embedding module to transform the edit tokens, where the embeddings are randomly initialized and optimized during the training. The token embedding becomes the element-wise addition of code embedding (i.e., original CodeBERT token embedding) and the edit embedding. In this way, we keep the power of CodeBERT in encoding hunk-level $\langle rem, add \rangle$ pairs (i.e., the input pair structure is unchanged) while providing token-level change information. We argue that the encoder can benefit from both coarse and fine grained change information.

2) Depth-specific Prediction Heads: There are d depth-specific heads attached upon the shared commit embedding module, each corresponds to a classification task at depth i, i.e., $f_i(c) = y_i$. The heads are composed of:

❶ **Hierarchical Bi-LSTM encoders.** The CWE categories are organized in a tree hierarchy for multiple levels of abstraction. Categories at deeper levels of the CWE tree require more fine-grained information to differentiate. For example, both CWE-74 and CWE-116 are related to *Improper Neutralization* as they belong to the same parent (CWE-707). What differentiates these two categories is the target that being improper neutralized, i.e., the output of the current component for CWE-116 and the input from an upstream component for CWE-74. The features used for predictions at lower levels are based on those from upper levels and further add more details.

Thus, we organize the Bi-LSTM encoders into a hierarchy to capture depth-specific features, i.e., each encoder further encodes beyond the representation produced by the previous one (see Figure 4). We regard the representation produced by the *Commit Embedding Module* as the most basic feature, and directly use it for the lowest-level classification. We average the output of the Bi-LSTM encoder at every time steps as the final representation of the input hunk. Since one commit may compose of multiple hunks, we further average the representation of each hunk to get the final representation of input commit. We choose AVG pooling instead of sophisticated Seq2Vec modules based on the opinion that the order of hunks within a commit should not affect the output embedding.

❷ **Chain classifiers.** Since the category at each depth should align with the parent to conform to the CWE tree hierarchy, TREEVUL explicitly exploits this relation by incorporating the information of the parent node in prediction. Specifically, the predicted CWE category at the upper level is transformed into embeddings and merged with the depth-specific commit embeddings using element-wise addition. Each CWE has a short title (attribute *Name*) that summarizes the core behaviour of this type of SVs. Inspired by the *label embedding* (how to describe a class) [45], [46], we utilize this side information from label (i.e., textual descriptions) in classification by transforming the short title into embedding of each CWE category using CodeBERT. Compared with randomly initialized embedding, this embedding already contains semantic information (similar categories share similar embeddings).

We implement each classifier using a one-layer fully connected feed forward module, which takes the merged embedding and predicts the CWE category at the corresponding depth. The loss used to train TREEVUL is defined as the average of the cross-entropy losses of d tasks:

$$loss = \sum_{i=1}^{d} \text{cross-entropy}(\widehat{y}_i, y_i) \tag{2}$$

C. Path Inference

The goal of TREEVUL is to find an optimal *path* from the root of the CWE tree to the target node (see Eq. 1). We first define the probability of a given path $(y_1, \ldots, y_d)$ using a merging rule, i.e., the sum of the logarithms of the predicted probabilities (transferred from the product of probabilities) on the CWE node along the path:

$$\log P(y_1, \ldots, y_d \mid c) = \sum_{t=1}^{d} \log P(y_t \mid y_1, \ldots, y_{t-1}, c) \tag{3}$$

Based on the defined path probability, we perform a *top-down* inference (see Figure 4). We start from the root CWE node, progressively choosing one node at the next depth until reaching the target depth. Our designed inference algorithm has two key points: ❶ *Tree structure aware*: the node at the next depth is only chosen from the children of the current node. We make sure the inferred path is valid, i.e., aligns with the CWE tree hierarchy. ❷ *Beam search based*: to alleviate the possible error propagation from the upper levels to the lower ones, we apply beam search (i.e., consider top k nodes at each depth) instead of greedy search and choose the path with the highest probability in the final. The pseudo-code of our inference algorithm can be found in an online Appendix [19].

The designed path inference algorithm further demonstrates the following advantages of incorporating CWE tree structure information with a HMC task setting: ❶ Classifications at the upper levels of the hierarchy are considerably less error-prone. During inference, TREEVUL makes good use of the upper-level classification results to reduce the error of lower ones, i.e., prunes invalid branches from the CWE tree. ❷ The chained architecture makes TREEVUL naturally more interactive, which is essential with the scenario of human-in-the-loop (e.g., partial path is available). Specifically, TREEVUL can better utilize the expert-curated classification results at the upper levels to perform the predictions at lower levels.

IV. EXPERIMENT SETUP

In this section, we first introduce our data collection procedure. Then, we describe our experiment settings.

A. Data Collection

To develop models for classifying security patches into fine-grained SV types, we build a dataset of security patches and label them with the corresponding CWE ID from NVD. We also collect the CWE entries with relevant information from the CWE website and organize them into a tree-like hierarchy.

Step 1: Collecting vulnerability-relevant information. We first collect all CVE records from NVD (on Oct. 19, 2021). We then crawl the CWE entries from the CWE website based on the the CWE IDs assigned to these records. Each CVE record has a *references* field, which lists the external links related to the vulnerability. We try to retrieve the links tagged with *patch* and filter out those not from GitHub.

Step 2: Collecting commit data. For OSS repositories hosted on GitHub, the patch (i.e., commit) is identified with a unique *hash* value, and can be retrieved using the URL: *https://github.com/{owner}/{repo}/commit/{hash}.patch*. We mainly focus on three types of patch-related URL links (i.e., commit, issue and pull request) and filter them out using the regular expressions. For links of issues and pull requests, we write custom crawling scripts to further retrieve the related commits. Finally, we crawl the commit data based on the extracted patch links, including the diff file and the metadata (e.g., commit date and commit message). We use the corresponding CWE ID to label the collected commits.

As a result, we collect 10,037 security patches, spanning across 2,260 OSS projects and corresponding to 6,384 CVE records. Different from the existing datasets [6], [47], [48] that assign binary labels (for distinguishing security patches from non-security ones), we label security patches with the CWE category of SVs being fixed for our type classification task. Besides, every existing dataset is restricted to one certain programming language (i.e., C/C++, Java or Python), while ours is not restricted and is thus much larger.

B. Data Preparation

Filtering the patch data. First, we remove duplicate commits and commits whose CWEs are invalid (e.g., missing, discarded). Then, we remove large commits with more than 100 files and 10,000 lines of code following the practice of [49], [50]. Next, we infer the file type based on the extension. We remove files without source code (e.g., data, documentation) or written in programming languages that appear less than 1% times (i.e., 301 files) in our collected dataset. The top 3 programming languages in our dataset are C/C++, PHP and Java, respectively. We further decide the CWE *path* for each commit and remove those associated with multiple CWEs. After this cleaning step, there are 8,275 commits left. Finally, since we aim to perform the fine-grained classification using categories at the third level of the CWE tree (see Section II-C), we filter out commits assigned with CWE categories at depth$<$3. As a result, the dataset used for experiments contains 6,541 commits (14,658 changed files), spanning across 1,560 OSS projects and corresponding to 4,253 CVEs.

Processing of code changes. we parse the `diff` file of each collected commit using a Python tool named `unidiff` [51]. Based on the parsed results, we extract the code revision at the hunk level. Specifically, for each hunk within a commit, we extract the removed and added code lines, respectively. We perform the same code segment preprocessing as CodeBERT.

Building the CWE tree. We utilize two attributes of each CWE entry: ❶ *Name*, short descriptions of the core behaviours of the SV type. It is used by TREEVUL to generate the label embedding. ❷ *Related Weaknesses*, relations with other CWE entries. We use the parent-child relation to organize the collected CWE entries into the tree hierarchy. Other types of relations (e.g., PeerOf) are not used in our study (see Section VI-C for more discussions). Based on the built CWE tree, we generate the ground truth *path* for each CWE entry. Specifically, we decide the unique (i.e., the most commonly-used) parent/path for CWE categories with multiple parents/-paths (account for 22.6% of CWEs and 16.5% of commits in our dataset). VIEW-1000 is intended to be theoretically comprehensive, but only a subset of categories are frequently used in practice [16], [52], [53]. Taking CWE-425 [54] as an example, it has three parents (i.e., CWE-288, CWE-424, and CWE-862) in VIEW-1000 with minor differences. Only CWE-862→CWE-425 are listed in the commonly-used CWEs suggested by the CWE team [52], VIEM-699 (Software Development) [23], or Top-25 weaknesses [53].

C. Experiment Setting

The experimental environment is a server with the NVIDIA GTX 3090 GPU, Intel Xeon 6226R CPU, running Ubuntu OS.

Implementation Details. The target depth is set to three (see Section II-C), i.e., we build and evaluate the model to classify security patches into categories at the third level of the CWE tree. We use the pre-trained CodeBERT model from the Hugging Face Transformer library [55]. We use 768-dimensional embeddings (same as CodeBERT [36]) for *edit* token. The hidden states of all Bi-LSTM modules are 384 dimensions; thus, the output is 768-dimensional. All the Bi-LSTMs have only one layer. Classifiers are implemented using one layer fully connected feed-forward module with the hidden layer size set to 512.

We use AdamW [56] as the optimizer. The learning rate (lr) is set to $5e^{-5}$ for the CodeBERT encoder following [36] and $1e^{-3}$ for other modules. During the training, the learning rate linearly warm-ups over the first 3,000 steps (roughly five epochs) and decays in the remaining steps. In addition, to avoid overfitting, we apply dropout [57] with the drop rate set to 0.1 and early stopping with patience set to 10. The beam size used in inference is set to five. The hunk-level removed and added code sequences are both truncated by 128 tokens, which covers 85% cases in our dataset. The maximum number of hunks to consider within a commit is set to eight since 85% of commits in our dataset change less than eight hunks.

Baselines. We adopt the following machine learning classifiers as our baselines: Random Forest (RF), LR (Linear Regression), SVM (Support Vector Machine), XGB (XGBoost), KNN (K-Nearest Neighbour). These methods are adopted in the latest work regarding security patch detection [6] and commit-level SV assessment [49]. For the above machine learning baselines, we follow [6], [49] to apply bag-of-words (BoW) as features and limit the vocabulary size to 10K.

We also include the following deep learning (DL) based approaches as our neural baselines: CodeBERT [36] and Bi-LSTM [58]. Zhou *et al.* [6] fine-tuned the CodeBERT to model code-changes, which achieved the best performance in identifying security patches. They also included a Bi-LSTM model as a baseline. We adopt the CodeBERT variant proposed by them in our experiments. For Bi-LSTM, we also set the vocabulary size to 10K following [6], [49]. We use the pretrained 300-dimensional Glove word embedding [59].

Evaluation Metrics. To evaluate the performance of classifying security patches into fine-grained CWE categories (i.e., a multi-class classification problem), we utilize the metrics including weighted F1-score, macro F1-score, and Matthews Correlation Coefficient (MCC) [60]. These metrics are commonly used in the literature regarding SV severity assessment (classifying security patches into different severity levels) [49], [61], [62]. Also, these metrics are suitable for our data where the classes are highly imbalanced (refer to section II-C for more details) [63]. Weighted F1-score is the average F1-score of all classes weighted by their support, i.e., the number of samples of each class in the test set. Macro F1-score is the unweighted mean of F1-score of all classes. MCC can also be regarded as a balanced measure despite the very different class sizes. Both F1-scores range from 0 to 1, while MCC ranges from -1 to 1. All three metrics have the best value of 1. However, there is no direct proportional relationship between F1-scores and MCC.

Although our task is to predict the CWE categories at depth-3, we also present the metrics of upper levels (i.e., depth 1 and 2) to provide more comprehensive insights of model performance. For example, assuming the correct category at depth 3 of the input sample is CWE-79, model *A* and *B* predict it as CWE-1236 and CWE-117, respectively. Although both models make wrong predictions, we argue that model *A* is better since CWE-1236 and CWE-79 share the same parent (CWE-74), suggesting it has a closer relation to the ground truth (see Figure 4). To explicitly consider the relation between categories based on the CWE hierarchy in the evaluation, we define a new metric named *Path Fraction (PF)*. For each test sample, we calculate the fraction of the predicted CWE path $\widehat{Y_i}$ (i.e., a sequence of CWE categories) which are actually part of the true path Y_i.

$$PF = \frac{1}{N}\sum_{i=1}^{N}\frac{\left|\widehat{Y_i}\cap Y_i\right|}{|Y_i|} \quad (4)$$

V. Experiment Results

In this paper, we aim to answer these two RQs:

- **RQ1: How effective is TreeVul compared to baselines for fine-grained SV type (i.e., depth-3 CWE) prediction?**
- **RQ2: How effective are the key designs of TreeVul?**

TABLE I: Description of datasets used in experiments

	# Commits	# Files	# Projects
Training Set	5,233	11,729	1,367
Validation Set	654	1,492	355
Testing Set	654	1,437	364

A. RQ1. The Effectiveness of TreeVul

Method. We compare the performance of TreeVul with both ML and DL baselines that are commonly adopted in the relevant tasks (see Section IV-C for more details). We split the dataset into the train set, validation set, and test set with a ratio of 8:1:1. Specifically, the dataset is divided using stratified random sampling, with each subset preserving the original distribution of CWE categories (at depth-3). Table I presents the statistics of the datasets used in the experiments.

Results. Table II presents the performance comparisons between TreeVul and baselines for depth-3 CWE category prediction (see Section IV-C for the motivation of including depth-1&2 metrics). The best results are highlighted in **bold**. Regarding ML baselines, KNN, as an unsupervised approach, performs much worse than the supervised counterparts. Among the supervised baselines, the performances of LR and XGB are close and much better than RF and SVM. The performance of DL baselines are generally better than the ML baselines. However, we observe a drop in the macro F1-score. We suspect that Bi-LSTM and CodeBERT are prone to the large categories, thus achieving better weighted performance while suffering on macro metrics.

TreeVul yields the best performance on all metrics for classifications at all three depths. At depth 3, TreeVul achieves the best weighted F1-score, macro F1-score and MCC of 0.72, 0.50, and 0.70, improving the best-performing baselines by 5.9%, 25.0%, and 7.7%, respectively. These results verify the effectiveness of TreeVul on fine-grained commit-level SV type prediction. Furthermore, we find that the improvements on macro F1-score are generally more significant than those on weighted F1-score. That is because the weighted F1-score is mainly determined by the model performances on large categories. As discussed in Section II-C, the CWE categories in our dataset are highly imbalanced. Performance improvement on small categories does not have much impact on the weighted F1-score. A classifier that is addicted to the large categories may still achieve satisfactory weighted F1-scores while lacking discriminative power. Thus, we argue the macro F1-score is more indicative in measuring the discriminative power of approaches. In the literature regarding commit-level SV assessment, which has a similar task setting with ours (i.e., multiclass classification with imbalanced class distribution), the macro F1-score is also preferred over the weighted version for model evaluation [49], [62]. Moreover, when comparing the performance improvements at different depths, we find the improvements for lower levels are more significant than those for upper levels, e.g., the improvement of macro F1-score at depth-3 is 25.0% (from 0.40 to 0.50) while is only 8.7% at depth-1 (from 0.69 to 0.75). This is because (1) the classification tasks at upper levels are easier with much fewer categories and more generic concepts; (2) the power of the hierarchical and chained model design of TreeVul lies in the classifications at lower levels of the CWE tree, where we can exploit more structural information. Considering that fine-grained categories offer more specific insights and actionable feedback about SVs (see Section II-C), TreeVul has larger practical values over the baselines. Besides, TreeVul also achieves the best performance (0.79) on the *PF* metric, indicating that TreeVul is capable of correctly predicting 2.4/3 CWE nodes along the correct path on average.

RQ-1: TreeVul *outperforms baselines for fine-grained SV type (i.e., CWE categories at depth-3) prediction, with an improvement of 5.9%, 25.0%, and 7.7% in terms of weighted F1-score, macro F1-score, and MCC (each compared to the best-performing baseline), respectively.*

B. RQ2. The Key Designs of **TreeVul**

Method. In RQ1, we have verified that TreeVul boosts the performance of baselines by a large margin. RQ2 aims to further provide insights into the effectiveness of the key designs of TreeVul: ❶ the design of incorporating the token-level change information in commit embedding (Section III-B1). ❷ the design of a hierarchical and chained model architecture, which exploits the CWE tree structure information (Section III-B2). We compare the performance of TreeVul with two variants (i.e., TreeVul-t and TreeVul-h) for depth-3 CWE category prediction, each lacking one of the aforementioned key designs. Specifically, TreeVul-t removes the token-level change information in commit embedding. TreeVul-h replaces the *Depth-specific Prediction Heads* in Figure 4 with a single classification layer, which directly predicts the CWE category at depth-3 instead of inferring a path from the root. TreeVul-h adopts the same *Commit Embedding Module* with TreeVul.

Results. Table III presents the performance comparisons between TreeVul and two variants for depth-3 CWE category prediction. The best results are highlighted in **bold**. TreeVul achieves the best performance across all metrics for classification tasks at all three depths. Comparing the performance of TreeVul with TreeVul-t, the weighted F1-score, macro F1-score, and MCC at depth-3 are improved by 1.4%, 8.7%, and 2.9%, respectively. As discussed before, the macro F1-score is preferred regarding the measurement of the model's discriminative power. The results verify that explicitly incorporating the fine-grained code change information can benefit the prediction of certain SV types (see Section III-B1).

TABLE II: The performance comparisons between TreeVul and baselines for depth-3 CWE category prediction

Model	Depth-1			Depth-2			Depth-3			PF
	Weighted F1	**Macro F1**	**MCC**	**Weighted F1**	**Macro F1**	**MCC**	**Weighted F1**	**Macro F1**	**MCC**	
RF	0.77	0.62	0.63	0.64	0.47	0.60	0.59	0.35	0.57	0.69
LR	0.76	0.66	0.62	0.67	0.48	0.61	0.63	0.40	0.59	0.69
SVM	0.71	0.55	0.54	0.56	0.30	0.51	0.49	0.21	0.47	0.62
XGB	0.77	0.68	0.64	0.67	0.46	0.62	0.63	0.39	0.61	0.71
KNN	0.63	0.50	0.40	0.48	0.32	0.40	0.44	0.26	0.38	0.53
Bi-LSTM	0.79	0.66	0.67	0.70	0.46	0.64	0.64	0.36	0.61	0.71
CodeBERT	0.81	0.69	0.69	0.72	0.43	0.68	0.68	0.37	0.65	0.74
TreeVul	**0.85**	**0.75**	**0.76**	**0.76**	**0.58**	**0.73**	**0.72**	**0.50**	**0.70**	**0.79**

TABLE III: The performance comparisons in the ablation study

Model	Depth-1			Depth-2			Depth-3			PF
	Weighted F1	**Macro F1**	**MCC**	**Weighted F1**	**Macro F1**	**MCC**	**Weighted F1**	**Macro F1**	**MCC**	
TreeVul-t	0.84	**0.75**	0.74	**0.76**	0.50	0.72	0.71	0.46	0.68	0.77
TreeVul-h	0.81	0.71	0.69	0.71	0.44	0.67	0.69	0.42	0.66	0.74
TreeVul	**0.85**	**0.75**	**0.76**	**0.76**	**0.58**	**0.73**	**0.72**	**0.50**	**0.70**	**0.79**

The core novelty of TreeVul is the design of adopting an HMC task setting, and further proposing a hierarchical and chained model architecture. This design leverages the CWE tree structure information as the prior knowledge of the classification task. The experiment results also verify the importance of this design as a key factor contributing to the performance improvement. TreeVul outperforms TreeVul-h by 4.3%, 19%, and 6.1% in terms of weighted F1-score, macro F1-score, and MCC at depth-3.

RQ-2: *Both designs of incorporating token-level change information and CWE tree structure information benefit the* TreeVul. *The latter one is the key factor contributing to performance improvement.*

VI. Discussion

In this section, we discuss two practical applications of our approach and the threats to the validity of our work.

A. User Study

We conduct a small-scale user study to investigate the practical value of including fine-grained SV type information to enhance the workflow of early remediation. Considering a practical application pipeline, developers first apply the existing patch detection techniques [6], [7] to filter commits. Then, for the detected patch-related commits, TreeVul further predicts fine-grained SV types, providing more insights and information to help following analysis.

Experiment Tasks. We randomly select nine commits from three types (three per type): a) patches whose true CWEs are within the top five categories recommended by TreeVul. b) patches whose true CWEs are not within the top five recommended categories. c) non SV-fixing commits that are *falsely* predicted as patches in the proceeding binary detection step. We use the same commits from [6] where they sampled false positives (FP) of the proposed patch detection technique to conduct a manual analysis. For each commit, we provide the top five CWE categories recommended by TreeVul as hints. For each CWE category, we present its *Name*, *Description* and direct URL to the CWE website. We ask following questions:

- Q1: Is this commit a vulnerability-fixing commit?
- Q2: How difficult is it to make the above judgement?
- Q3: If the answer to Q1 is *yes*, what type of vulnerability does this commit fix? (briefly describe the reasons)

For Q1 and Q2, we aim to investigate the usefulness of providing CWE categories to help participants understand and verify the patch detection results. We use 5-point likert scale to measure the difficulties. With Q3, we further evaluates participants' deeper understandings of the detected patch by asking them to assign a specific SV type. We provide 11 options corresponding to the top 10 frequent SVs [3], [6] (plus an *Other* option). A correct understanding of the SV nature can better guide the following remediation process.

In practice, FP patches from the preceding detection step are inevitable. Providing irrelevant CWE categories to these non SV-fixing commits may mislead the participants. We include FP patches in user study to explore the possible side effects (Type *C*). Besides, we also include patches with wrongly recommended categories (Type *B*). We want to investigate when does our approach fail and how will the wrong categories impact participants' analysis. The top-1/3/5 accuracy of TreeVul on our test set are 0.73, 0.84 and 0.87.

Participants. We invite nine security experts from a prominent IT company with three to five years of experience in software security as our participants. We divide participants into three groups (three per group): ❶ the experimental group provided with the CWE categories (at depth 3) recommended by TreeVul as hints. We present the complete *path* from the root of the CWE tree to the predicted node. ❷ the control group provided with the parent (at depth 1) of the predicted CWE categories. ❸ the blank control group with no hints.

From Group *3* to *1*, we gradually add more detailed SV type information. By comparing these three groups, we can have a comprehensive view of gains and losses of providing fine-grained SV categories in real applications.

Results. Table IV presents the correctness (Q1 and Q3) and difficulties (Q2) for each task (i.e., sum of the Likert scores given by the three participants) of the experimental group and two control groups. For patches with correctly recommended

TABLE IV: Results of our user study

Tasks		Type A			Type B			Type C		
		T1	T7	T9	T3	T5	T8	T2	T4	T6
Q1 (# Correct)	G1	2	3	3	3	2	0	2	2	2
	G2	1	2	1	3	1	1	3	2	2
	G3	0	2	2	3	2	1	3	2	3
Q2 (Difficulty)	G1	13	5	8	4	12	5	9	11	8
	G2	12	8	11	7	8	8	12	9	7
	G3	11	7	14	7	10	9	11	7	9
Q3 (# Correct)	G1	2	3	3	1	1	0	-	-	-
	G2	0	2	1	3	1	0	-	-	-
	G3	0	1	2	3	2	1	-	-	-

CWE categories (Type *A*), the experimental group generally has higher correctness and reports less difficulties for completing the tasks. We collected feedback from participants in the blank control group who struggled to make correct judgements. They stated that they were unfamiliar with or failed to recall the typical features of certain SV types. Regarding the control group with CWE categories at depth one, we find its performance is worse than the experimental group, and is almost similar to the the blank control group. This observation indicates that providing coarse-grained type information may not be helpful. For example (*T1*), the patch of CVE-2020-23995 [64] fixes an *Information Exposure* [65] SV caused by the improper generation of error message which leaks the upload data path. The patch changes 4 files with 7 additions and 12 deletions, while only one hunk buried in the middle is directly related to the SV. TreeVul recommends the correct CWE, the website of which [65] clearly lists the *"Generation of Error Message Containing Sensitive Information"* as one of the specific cases and provides a similar demonstrative example. However, the parent CWE at depth one is too abstractive (*Improper Control of a Resource Through its Lifetime*) [66] to provide useful hints for this specific case.

For patches with wrongly recommended CWE categories (Type *B*), we have two observations: (1) The wrong categories may closely relate to the correct one, thus misleading the experimental group in understanding the SV details. For example (*T3*), the patch of CVE-2019-17177 [67] fixes a *Memory Leak* [68] SV by adding `free` statements, while the predicted CWE category is *Use After Free* [69]. (2) Some of these patches are really difficult to understand, even for human analysts (e.g., *T5*, *T8*).

For FP patches (Type *C*), providing irrelevant CWE categories does not necessarily mislead the participants. These commits are sometimes deceptive as they confused the detection model. For example (*T6*), the commit [70] updates the verification mechanism and the recommended CWE is *Improper Certificate Validation* [71]. However, the commit actually adds support to allow to skip verification under certain circumstances. We observed that participants who were likely to be misled by the irrelevant CWE categories were those not careful enough (i.e., they felt the tasks were easier). For those not affected, they said that after examining the commit details, they could understand why the model was deceived and thus were confident with their judgements.

B. Improve the Quality of CWE Metadata in NVD

We conduct a case study to present the application of TreeVul in updating the existing upper-level CWEs into more fine-grained categories. We discuss in Section III-C that the design of TreeVul brings particular advantages in application scenarios with human-in-the-loop (i.e., parent categories are curated by analysts). We crawl the latest SV data from NVD (on Aug. 10th, 2022) and match them with those used in our experiments (on Oct. 19th, 2021). We find six CVE records (see Table V) whose assigned CWE categories have been updated from depth 1 or 2 into lower ones. The updates are made 427 days later on average, and the severity of these CVEs are all above *High* with an average CVSS score of 8.7/10. These results show that NVD tries to fix low-quality CWE mappings with high-severity SVs as priorities, and the current practice faces significant delay in updates.

TABLE V: CVEs with updated CWEs in the case study

CVE ID	Old CWE	New CWE	Initial - Update Date	Severity(CVSS)
CVE-2020-15255	CWE-74	CWE-1236	2020.10.28 - 2021.11.18	HIGH (7.3)
CVE-2021-21305	CWE-74	CWE-94	2021.02.12 - 2022.04.26	HIGH (8.8)
CVE-2021-27185	CWE-74	CWE-77	2021.02.16 - 2022.04.29	CRITICAL (9.8)
CVE-2021-28122	CWE-287	CWE-306	2021.03.18 - 2022.07.12	CRITICAL (9.8)
CVE-2021-32620	CWE-285	CWE-863	2021.05.28 - 2022.08.05	HIGH (8.8)
CVE-2021-34825	CWE-311	CWE-319	2021.06.21 - 2022.07.12	HIGH (7.5)

We apply TreeVul to automatically update the CWE category of the six CVE records (at depth 1 or 2), i.e., continue the inference of the complete path (to depth 3) based on the old CWE (see Section III-C). In all six cases, the correctly updated CWEs can be retrieved within the top two CWE categories recommended by TreeVul. In case 3/4/6, the first recommended CWE is correct. These results show the potential of applying TreeVul to reduce human efforts and biases on manual SV type analysis.

C. Threats to Validity

Threats to internal validity refer to the experiment bias and errors. The first threat comes from the collection of security patch dataset. A patch may contain noise (i.e., non fixing-related code changes) [2]. For files within a commit, we infer its type based on extension. We remove files that do not contain source code (e.g., documentation) or are written in less commonly used programming languages. Threats related to our approach are ❶ The construction of token-level code changes. The removed and added code sequences may not be perfectly aligned, leading to wrongly inferred token change information. ❷ TreeVul only leverages parent-child relations of CWE to perform top-down search from abstract categories to specific ones. Parent-child is the most important relation in the CWE hierarchy [27]. There are other types of relations (e.g., PeerOf) that TreeVul may benefit from. However, including these relations will turn CWE tree into a graph. While the idea of inferring a path under the HMC task setting is still applicable (Section III-A), specific implementations need adjustments and will become more complicated. ❸ Our task is to assign depth-3 CWE categories, however, some important categories may be located at depth>3, we plan to optimize TreeVul to allow it automatically decide the appropriate level to end the top-down search (e.g., confidence is below a threshold) for each specific input in the future work.
Threats to external validity refer to the generalizability of our approach. We collect SV-fixing commits only from

the projects hosted on GitHub, which might not represent all security patches. Nevertheless, the security patches in our dataset cover various OSS projects written in different programming languages. Besides, we only consider patch as the model input following [6], [40]. The commit message may also contain useful information, though according to the coordinated disclosure, commit messages should hide the intention of SV fixing [6], [40]. For example, Apache suggests commit messages should not make any reference to any security-related nature [72]. However, we argue the key unique design of TREEVUL (i.e., leveraging the CWE tree structure using a hierarchical and chained model architecture) is generalizable to automate fine-grained type analysis for other SV-related artifacts (e.g., commit message, SV-inducing commit, and report). The only necessary change is to replace the *Commit Embedding Module* of TREEVUL (see Figure 4) with the corresponding encoders, while the *Depth-specific Prediction Heads* are generic. Another threat is that TREEVUL is designed to identify CWE categories by leveraging its tree structure. Thus, TREEVUL might not be compatible with other classification schemes (i.e., without hierarchy). However, CWE is the most widely used standard for classifying SVs.

Threats to construct validity refer to the suitability of evaluation measures. We mainly adopt the same metrics following a recent work regarding commit-level SV assessment [49], which shares similar task settings with ours. Besides, we propose a new metric named *PF* (Section IV-C), which may have some latent limitations, since the traversal path to some CWEs are not distinct. We try to minimize this bias by choosing the most commonly-used *path* for CWEs with multiple parents. In most cases (like CWE-425 discussed in Section IV-B), the *PF* is not affected as not-selected parents (i.e., CWE-288 and CWE-424) do not appear in our dataset, i.e., we can regard the selected parent (CWE-862) as a merged node of all parents.

VII. Related Work

In this section, we describe two aspects of the related work:

Security Patch Detection. Many approaches have been proposed to detect security patches to allow early warnings and timely remediation of SVs [6], [7], [9], [40]. Sabetta and Bezzi [9] consider the code changes as bags of words (BoW) and build an SVM model to identify SV fixes. Zhou et al. [7] implement a DL-based patch identification system, utilizing both commit message and code revision with two separate network components. Wang *et al.* [40] point out the importance of detecting silent SV fixes (i.e., without explicit indications) to prevent 0-day attack. They manually identify 61 code features and further propose a machine learning based classification mechanism. Recently, Zhou et al. [6] propose VulFixMiner, leveraging CodeBERT to represent commit-level code changes, to identify silent SV fixes. Different from the existing studies targeting at patch detection, we are the first focus on enriching the practical value of the binary detection results by providing fine-grained SV type information.

SV Type Prediction. Most of the existing studies focus on classifying experts-curated SV descriptions into CWE categories to better understand the SV nature and risk [13], [73], [74]. Na *et al.* [74] utilize Naïve Bayes classifier to categorize CVE descriptions into the 10 most frequent CWE categories. Ruohonen *et al.* [73] propose an information retrieval technique for mapping the SV description from NVD and Synk to the most similar CWE category. They apply cosine similarity based on the tf-idf feature of the text description. To the best of our knowledge, there is only one approach, namely μVulDeePecker [25], predicts SV type by analyzing the source code (i.e., vulnerable functions). However, this model is only capable of classifying C/C++ functions into 40 selected CWE categories. Different from the existing studies, we are the first to 1) automate type prediction by analyzing commit-level code revisions, 2) leverage the CWE tree structure to perform fine-grained classifications (e.g., depth-3 CWE categories).

VIII. Conclusion and Future Work

In this paper, we take the first step to categorize the detected security patches into fine-grained SV types. We leverage the CWE tree structure to propose an approach named TREEVUL. TREEVUL recasts the prediction task into an HMC problem, i.e., inferring a path (a sequence of CWE nodes) from the root of the CWE tree to the target category (at depth-3) of the input patch. TREEVUL exploits the relation between categories in the CWE tree hierarchy by the design of a hierarchical and chained model architecture. We evaluate the effectiveness of TREEVUL on a dataset containing 6,541 security patches from 1,560 GitHub OSS repositories. The experimental results show that TREEVUL outperforms the best performing baseline, and verify the effectiveness of the key designs. Finally, we conduct a user study and a case study to verify the practical value of TREEVUL in enhancing the workflow of early remediation and improving the data quality of NVD, respectively.

We plan to extend TREEVUL to other SV-related artifacts (e.g., SV-inducing commit) to provide an all-in-one solution to automate fine-grained type analysis throughout SV lifecircle.

IX. Acknowledgments

This project is supported by the National Key Research and Development Program of China (No. 2021YFB2701102), the National Science Foundation of China (No. U20A20173, No. 61902344, and No. 62141222), the Fundamental Research Funds for the Central Universities (No. 226-2022-00064), Alibaba-Zhejiang University Joint Institute of Frontier Technologies and Alibaba Group through Alibaba Innovative Research Program, the National Research Foundation, Singapore and National University of Singapore through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) office under the Trustworthy Computing for Secure Smart Nation Grant (TCSSNG) award no. NSOE-TSS2020-02. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and National University of Singapore (including its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) office).

References

[1] "National vulnerability database." [Online]. Available: https://nvd.nist.gov/

[2] A. Sejfia, Y. Zhao, and N. Medvidović, "Identifying casualty changes in software patches," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 304–315.

[3] F. Li and V. Paxson, "A large-scale empirical study of security patches," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2201–2215.

[4] "Github documents about coordinated disclosure of security vulnerabilities," https://docs.github.com/en/code-security/repository-security-advisories/about-coordinated-disclosure-of-security-vulnerabilities, accessed: 2022-09-01.

[5] N. Imtiaz, A. Khanom, and L. Williams, "Open or sneaky? fast or slow? light or heavy?: Investigating security releases of open source packages," *IEEE Transactions on Software Engineering*, 2022.

[6] J. Zhou, M. Pacheco, Z. Wan, X. Xia, D. Lo, Y. Wang, and A. E. Hassan, "Finding a needle in a haystack: Automated mining of silent vulnerability fixes," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 705–716.

[7] Y. Zhou, J. K. Siow, C. Wang, S. Liu, and Y. Liu, "Spi: Automated identification of security patches via commits," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–27, 2021.

[8] Y. Chen, A. E. Santosa, A. M. Yi, A. Sharma, A. Sharma, and D. Lo, "A machine learning approach for vulnerability curation," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 32–42.

[9] A. Sabetta and M. Bezzi, "A practical approach to the automatic classification of security-relevant commits," in *2018 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 2018, pp. 579–582.

[10] S. Keßler and P. Alpar, "Customization of open source software in companies," in *IFIP International Conference on Open Source Systems*. Springer, 2009, pp. 129–142.

[11] T. H. Le, H. Chen, and M. A. Babar, "A survey on data-driven software vulnerability assessment and prioritization," *arXiv preprint arXiv:2107.08364*, 2021.

[12] A. Anwar, A. Abusnaina, S. Chen, F. Li, and D. Mohaisen, "Cleaning the nvd: Comprehensive quality assessment, improvements, and analyses," *IEEE Transactions on Dependable and Secure Computing*, 2021.

[13] M. Aota, H. Kanehara, M. Kubo, N. Murata, B. Sun, and T. Takahashi, "Automation of vulnerability classification from its description using machine learning," in *2020 IEEE Symposium on Computers and Communications (ISCC)*. IEEE, 2020, pp. 1–7.

[14] "Cwe overview," https://cwe.mitre.org/about/index.html, accessed: 2022-09-01.

[15] "Vuldb." [Online]. Available: https://vuldb.com/?doc.sources

[16] "Nvd - categories." [Online]. Available: https://nvd.nist.gov/vuln/categories

[17] C. Elbaz, L. Rilling, and C. Morin, "Automated keyword extraction from" one-day" vulnerabilities at disclosure," in *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2020, pp. 1–9.

[18] "Cve to cwe mapping guidance," https://cwe.mitre.org/documents/cwe_usage/guidance.html, accessed: 2022-09-01.

[19] "Replication package," https://doi.org/10.6084/m9.figshare.19727050, accessed: 2023-02-10.

[20] C. N. Silla and A. A. Freitas, "A survey of hierarchical classification across different application domains," *Data Mining and Knowledge Discovery*, vol. 22, no. 1, pp. 31–72, 2011.

[21] "Common weakness enumeration." [Online]. Available: https://cwe.mitre.org/index.html

[22] "Cwe-1000: Research concepts," https://cwe.mitre.org/data/definitions/1000.html, accessed: 2022-09-01.

[23] "Cwe-699: Software development," https://cwe.mitre.org/data/definitions/699.html, accessed: 2022-09-01.

[24] "Cwe-1194: Hardware design," https://cwe.mitre.org/data/definitions/1194.html, accessed: 2022-09-01.

[25] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μvuldeepecker: A deep learning-based system for multiclass vulnerability detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2224–2236, 2019.

[26] S. Pan, J. Zhou, F. R. Cogo, X. Xia, L. Bao, X. Hu, S. Li, and A. E. Hassan, "Automated unearthing of dangerous issue reports," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 834–846.

[27] Z. Han, X. Li, H. Liu, Z. Xing, and Z. Feng, "Deepweak: Reasoning common software weaknesses via knowledge graph embedding," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2018, pp. 456–466.

[28] "Cwe-79." [Online]. Available: https://cwe.mitre.org/data/definitions/79.html

[29] "Cve-2021-25964," https://nvd.nist.gov/vuln/detail/CVE-2021-25964, accessed: 2022-09-01.

[30] "patch of cve-2021-25964," https://github.com/janeczku/calibre-web/commit/32e27712f0f71fdec646add20cd78b4ce75acfce, accessed: 2022-09-01.

[31] "Official doc of lxml," https://lxml.de/lxmlhtml.html#cleaning-up-html, accessed: 2022-09-01.

[32] "Cwe mapping tips," https://cwe.mitre.org/documents/cwe_usage/quick_tips.html, accessed: 2022-09-01.

[33] M. Crawshaw, "Multi-task learning with deep neural networks: A survey," *arXiv preprint arXiv:2009.09796*, 2020.

[34] J. Read, B. Pfahringer, G. Holmes, and E. Frank, "Classifier chains for multi-label classification," in *Joint European conference on machine learning and knowledge discovery in databases*. Springer, 2009, pp. 254–269.

[35] V. Sanh, T. Wolf, and S. Ruder, "A hierarchical multi-task approach for learning embeddings from semantic tasks," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, 2019, pp. 6949–6956.

[36] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[37] Z. Liu, X. Xia, D. Lo, M. Yan, and S. Li, "Just-in-time obsolete comment detection and update," *IEEE Transactions on Software Engineering*, 2021.

[38] S. Panthaplackel, P. Nie, M. Gligoric, J. J. Li, and R. J. Mooney, "Learning to update natural language comments based on code changes," *arXiv preprint arXiv:2004.12169*, 2020.

[39] K. Lu, A. Pakki, and Q. Wu, "Detecting {Missing-Check} bugs via semantic-and {Context-Aware} criticalness and constraints inferences," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1769–1786.

[40] X. Wang, K. Sun, A. Batcheller, and S. Jajodia, "Detecting" 0-day" vulnerability: An empirical study of secret security patch in oss," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019, pp. 485–492.

[41] "Cwe-119." [Online]. Available: https://cwe.mitre.org/data/definitions/119.html

[42] "Python difflib," https://docs.python.org/3/library/difflib.html, accessed: 2022-09-01.

[43] "Cve-2018-10887." [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-10887

[44] "Cwe-125." [Online]. Available: https://cwe.mitre.org/data/definitions/125.html

[45] Z. Akata, F. Perronnin, Z. Harchaoui, and C. Schmid, "Label-embedding for image classification," *IEEE transactions on pattern analysis and machine intelligence*, vol. 38, no. 7, pp. 1425–1438, 2015.

[46] G. Wang, C. Li, W. Wang, Y. Zhang, D. Shen, X. Zhang, R. Henao, and L. Carin, "Joint embedding of words and labels for text classification," *arXiv preprint arXiv:1805.04174*, 2018.

[47] X. Wang, S. Wang, P. Feng, K. Sun, and S. Jajodia, "Patchdb: A large-scale security patch dataset," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2021, pp. 149–160.

[48] 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.

[49] T. H. M. Le, D. Hin, R. Croft, and M. A. Babar, "Deepcva: Automated commit-level vulnerability assessment with deep multi-task learning," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 717–729.

[50] S. McIntosh and Y. Kamei, "Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction," *IEEE Transactions on Software Engineering*, vol. 44, no. 5, pp. 412–428, 2017.

[51] "Python unidiff," https://pypi.org/project/unidiff/, accessed: 2022-09-01.

[52] "Weaknesses for simplified mapping of published vulnerabilities," https://cwe.mitre.org/data/definitions/1003.html, accessed: 2022-09-01.

[53] "2022 cwe top 25 most dangerous software weaknesses," https://cwe.mitre.org/data/definitions/1387.html, accessed: 2022-09-01.

[54] "Cwe-425: Direct request (forced browsing)," https://cwe.mitre.org/data/definitions/425.html, accessed: 2022-09-01.

[55] "Microsoft codebert model from huggingface transformer library," https://huggingface.co/microsoft/codebert-base, accessed: 2022-09-01.

[56] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," *arXiv preprint arXiv:1711.05101*, 2017.

[57] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[58] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *2013 IEEE international conference on acoustics, speech and signal processing*. Ieee, 2013, pp. 6645–6649.

[59] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[60] J. Gorodkin, "Comparing two k-category assignments by a k-category correlation coefficient," *Computational biology and chemistry*, vol. 28, no. 5-6, pp. 367–374, 2004.

[61] Z. Han, X. Li, Z. Xing, H. Liu, and Z. Feng, "Learning to predict severity of software vulnerability using only vulnerability description," in *2017 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 2017, pp. 125–136.

[62] G. Spanos and L. Angelis, "A multi-target approach to estimate software vulnerability characteristics and severity scores," *Journal of Systems and Software*, vol. 146, pp. 152–166, 2018.

[63] A. Luque, A. Carrasco, A. Martín, and A. de Las Heras, "The impact of class imbalance in classification performance metrics based on the binary confusion matrix," *Pattern Recognition*, vol. 91, pp. 216–231, 2019.

[64] "Patch of cve-2020-23995," https://github.com/ILIAS-eLearning/ILIAS/commit/94d9b16010ec3abeae8d2cbb05622ccd999119ad, accessed: 2022-09-01.

[65] "Cwe-200: Exposure of sensitive information to an unauthorized actor (information exposure)," https://cwe.mitre.org/data/definitions/200.html, accessed: 2022-09-01.

[66] "Cwe 664: Improper control of a resource through its lifetime," https://cwe.mitre.org/data/definitions/664.html, accessed: 2022-09-01.

[67] "Patch of cve-2019-17177," https://github.com/FreeRDP/FreeRDP/commit/9fee4ae076b1ec97b97efb79ece08d1dab4df29a, accessed: 2022-09-01.

[68] "Cwe 401: Cwe-401: Missing release of memory after effective lifetime (memory leak)," https://cwe.mitre.org/data/definitions/401.html, accessed: 2022-09-01.

[69] "Cwe-672: Operation on a resource after expiration or release (use after free)," https://cwe.mitre.org/data/definitions/672.html, accessed: 2022-09-01.

[70] "False positive patch," https://github.com/urllib3/urllib3/commit/dfd9e0dc7ecbe4bb7abe723b5fd0e01688096d02, accessed: 2022-09-01.

[71] "Cwe-295: Improper certificate validation," https://cwe.mitre.org/data/definitions/295.html, accessed: 2022-09-01.

[72] "Asf project security for committers (apache.org)." [Online]. Available: https://www.apache.org/security/committers.html

[73] J. Ruohonen and V. Leppänen, "Toward validation of textual information retrieval techniques for software weaknesses," in *International Conference on Database and Expert Systems Applications*. Springer, 2018, pp. 265–277.

[74] S. Na, T. Kim, and H. Kim, "A study on the classification of common vulnerabilities and exposures using naïve bayes," in *International Conference on Broadband and Wireless Computing, Communication and Applications*. Springer, 2016, pp. 657–662.