Empirical Software Engineering (2024) 29:4
https://doi.org/10.1007/s10664-023-10346-3

AIBugHunter: A Practical tool for predicting, classifying and repairing software vulnerabilities

Michael Fu[1] · Chakkrit Tantithamthavorn[1] · Trung Le[1] · Yuki Kume[1] · Van Nguyen[1] · Dinh Phung[1] · John Grundy[1]

Accepted: 25 May 2023
© The Author(s) 2023

Abstract

Many Machine Learning(ML)-based approaches have been proposed to automatically detect, localize, and repair software vulnerabilities. While ML-based methods are more effective than program analysis-based vulnerability analysis tools, few have been integrated into modern Integrated Development Environments (IDEs), hindering practical adoption. To bridge this critical gap, we propose in this article AIBugHunter, a novel Machine Learning-based software vulnerability analysis tool for C/C++ languages that is integrated into the Visual Studio Code (VS Code) IDE. AIBugHunter helps software developers to achieve real-time vulnerability detection, explanation, and repairs during programming. In particular, AIBugHunter scans through developers' source code to (1) locate vulnerabilities, (2) identify vulnerability types, (3) estimate vulnerability severity, and (4) suggest vulnerability repairs. We integrate our previous works (i.e., LineVul and VulRepair) to achieve vulnerability localization and repairs. In this article, we propose a novel multi-objective optimization (MOO)-based vulnerability classification approach and a transformer-based estimation approach to help AIBugHunter accurately identify vulnerability types and estimate severity. Our empirical experiments on a large dataset consisting of 188K+ C/C++ functions confirm that our proposed approaches are more accurate than other state-of-the-art baseline methods for vulnerability classification and estimation. Furthermore, we conduct qualitative evaluations including a survey study and a user study to obtain software practitioners' perceptions of our AIBugHunter tool and assess the impact that AIBugHunter may have on developers' productivity in security aspects. Our survey study shows that our AIBugHunter is perceived as useful where 90% of the participants consider adopting our AIBugHunter during their software development. Last but not least, our user study shows that our AIBugHunter can enhance developers' productivity in combating cybersecurity issues during software development. AIBugHunter is now publicly available in the Visual Studio Code marketplace.

This article belongs to the Topical Collection: *Mining Software Repositories (MSR)*

✉ Chakkrit Tantithamthavorn
chakkrit@monash.edu

Extended author information available on the last page of the article

Published online: 20 November 2023

Keywords Vulnerability prediction · Vulnerability localization · Vulnerability classification · Vulnerability repair

1 Introduction

Software vulnerabilities are weaknesses in an information system, security procedures, internal controls, or implementations that could be exploited or triggered by a threat source Johnson et al. (2011). Such unresolved weaknesses result in extreme security or privacy risks. According to the research conducted by WhiteSource (2019) on open source vulnerabilities in the past 10 years (including multiple sources like the National Vulnerability Database (NVD), security advisories, GitHub issue trackers etc.), C has the highest number of vulnerabilities out of all seven reported languages (i.e., C, PHP, Java, JavaScript, Python, C++, Ruby), accounting for 47% of all reported vulnerabilities. Buffer errors (e.g., CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer) are the most common vulnerability in C and C++. It is worth noting that this group of vulnerabilities related to memory corruption could often have critical consequences such as system crashes and sensitive information disclosure. In particular, our proposed software vulnerability classification approach can correctly identify 79% of the CWE-119 buffer error as shown in Table 2 (see Rank 17).

Recently, the shift-left testing concept (i.e. move software testing earlier in project timelines) has been proposed to try to perform software testing at earlier stages of development, instead of testing applications during late phases of development. Thus, vulnerabilities related to fundamental features, such as buffer errors, could ideally be found and fixed earlier. DevSecOps has also been proposed to extend the idea of DevOps by integrating security into DevOps initiatives (Zettler, 2022). DevSecOps aims to examine application security from the start of development by automating some security gates and selecting the right tools to continuously integrate security in the DevOps workflow. For example, program analysis(PA)-based tools can be integrated into IDEs, such as Visual Studio Code (VS Code), to detect such vulnerabilities during coding. However, these methods usually rely on pre-defined vulnerability patterns and struggle to detect specific types of vulnerability. Croft et al. (2021) demonstrated that Machine Learning(ML)-based techniques are more accurate than PA-based tools in detecting file-level vulnerabilities. Our own previous study showed that our ML-based LineVul approach is more accurate than the PA-based Cppcheck tool (Marjamäki, 2007) on line-level vulnerability prediction (Fu and Tantithamthavorn, 2022b; Pornprasit and Tantithamthavorn, 2021, 2022). ML-based methods learn vulnerability patterns based on historical vulnerability data instead of relying on pre-defined patterns. Thus, ML-based approaches can capture more kinds of vulnerabilities and be more easily extended as new vulnerabilities emerge. PA-based tools, such as Checkmarx (Checkmarx, 2006), have been integrated into software development workflow to support security diagnosis during development. However, to date, ML-based tools have not been integrated as security tools to help detect security issues during software development.

In this article, we propose an ML-based software vulnerability analysis tool, AIBugHunter, to bridge the critical gap between ML-based security tools and software practitioners. AIBugHunter is integrated into a modern IDE (i.e., VS Code) – to fulfil the concept of shift-left testing and to support real-time security inspection during software development. In particular, given developers' source code written in C/C++, our AIBugHunter can (1) locate vulnerabilities, (2) classify vulnerability types, (3) estimate vulnerability severity, and (4) suggest repairs. We integrate our previous work LineVul (Fu and Tantithamtha-

vorn, 2022b) and VulRepair (Fu et al., 2022) for AIBugHunter to achieve automated vulnerability localization and repairs. In this article, we further propose a multi-objective optimization (MOO)-based approach to optimize the multi-task learning scenario and help our AIBugHunter accurately identify vulnerability types (i.e., CWE-IDs, and CWE-Types) and explain the detected vulnerabilities. In addition, a transformer-based approach is proposed to help AIBugHunter estimate the vulnerability severity (i.e., CVSS Score) which could be beneficial for the prioritization of security issues.

We evaluate our proposed MOO-based vulnerability classification and severity estimation approaches on a large dataset that consists of 188k+ C/C++ functions including various vulnerability types and severity. We found that our MOO-based vulnerability classification approach outperforms other baseline methods and achieves the accuracy of 65% (demonstrated in RQ1) and 74% (demonstrated in RQ2) for classifying CWE-ID and CWE-Types respectively. In addition, our transformer-based severity estimation approach outperforms other baseline methods and achieves the best mean squared error (MSE) and mean absolute error (MAE) measures (demonstrated in RQ3). We evaluate our AIBugHunter through qualitative evaluations including (1) a survey study to obtain software practitioners' perceptions of our AIBugHunter tool; and (2) a user study to investigate the impact that our AIBugHunter could have on developers' productivity in security aspects. Our survey study shows that predictions provided by AIBugHunter are perceived as useful by 47%-86% of participated software practitioners and 90% of participants will consider adopting our AIBugHunter. Moreover, our user study indicates that AIBugHunter could save developers' time spent on security analysis that could potentially enhance security productivity during software development (demonstrated in RQ4).

The main contributions of this work include:

1. AIBugHunter, a novel ML-based software security tool for C/C++ that is integrated into the VS Code IDE to bridge the gap between ML-based vulnerability prediction techniques and software developers and achieve real-time security inspection;
2. A quantitative evaluation of AIBugHunter on a large dataset showing its high precision and recall;
3. A qualitative survey study of AIBugHunter with 21 software practitioners demonstrating both its practicality and potential acceptance;
4. A qualitative user study of AIBugHunter with 6 software practitioners demonstrating AIBugHunter could enhance practitioners' productivity in combating security issues during software development;
5. A multi-objective optimization approach for vulnerability classification that optimizes the multi-task learning scenario for classifying the vulnerability types for vulnerable functions written in C/C++; and
6. A transformer-based approach to estimate vulnerability severity for vulnerable functions written in C/C++.

We make available our datasets, scripts including data processing, model training, model evaluation, and experimental results related to our approach in a GitHub repository: (https://github.com/awsm-research/AIBugHunter). Additionally, AIBugHunter is available at VS Code marketplace (https://marketplace.visualstudio.com/items?itemName=AIBugHunter.aibughunter).

The rest of this article is organized as follows. Section 2 presents a high-level overview of our AIBugHunter. Section 3 presents our approach to predicting vulnerability types and severity. Section 4 presents our studied datasets, our experimental setup, and our first three research questions along with their results. Section 5 presents a qualitative evaluation

of AIBugHunter including a survey study and a user study to answer the last research question. Section 6 discloses the threats to validity. Section 7 discusses the related works. Section 8 draws the conclusions.

2 AIBugHunter: Our Approach

We provide an overview of our AIBugHunter, an ML-based vulnerability prediction tool as a plug-in in Visual Studio Code (VS Code). The main purpose of our AIBugHunter is to bridge the gap between ML-based vulnerability prediction techniques and software developers by providing a security plug-in in IDE to present more security information during software development.

2.1 AIBugHunter Security Tool

As a security tool integrated into VS Code, AIBugHunter first scans the file opened by developers and parse the whole file into multiple separate functions. For each function, our AIBugHunter performs the following 4 steps:

1. *Localize* the vulnerable lines (LineVul);
2. *Classify* the vulnerability types (proposed in this paper);
3. *Estimate* the vulnerability severity (proposed in this paper); and
4. *Suggest* the repair patches (VulRepair).

where LineVul (Fu and Tantithamthavorn, 2022b) locates vulnerable lines; our approach predicts types and severity; and VulRepair (Fu et al., 2022) suggests repairs.

In AIBugHunter, we use LineVul and VulRepair from our previous works. These models were trained using the extensive Big-Vul dataset offered by Fan et al. (2020) and the CVEFixes dataset provided by Bhandari et al. (2021). We illustrate both of them as follows:

LineVul is among the first to predict line-level vulnerabilities using the transformer model and its self-attention mechanism. Given a C/C++ function as input, first, LineVul leverages a BPE tokenizer to tokenize the function into subword tokens and mitigate the out-of-vocab problem. Second, LineVul leverages transformer encoders (Vaswani et al., 2017) to learn the representation of those tokens, which can better tackle the long-term dependencies among tokens than previously proposed RNN-based methods (Li et al., 2021). Third, LineVul uses a linear classification head to predict function-level vulnerability prediction based on the learned representations. LineVul uses intrinsic model interpretation to localize line-level vulnerabilities. In particular, LineVul summarizes the self-attention scores of each line in the function and ranks the line scores to place potentially vulnerable lines on the top. Our previous work (Fu and Tantithamthavorn, 2022b) has demonstrated that LineVul achieves the best accuracy for both function-level and line-level vulnerability prediction and is the most cost-effective approach to localize line-level vulnerabilities when compared with other baseline methods.

VulRepair is among the first to leverage a large pre-trained language model for the automated vulnerability repair (AVR) problem. Given a vulnerable C/C++ function as input, instead of using word-level tokenization as previous work (Chen et al., 2021), VulRepair leverages a BPE tokenizer to tokenize the function into subword tokens and address the potential OOV problem. VulRepair uses a pre-trained encoder-decoder T5 architecture where encoders encode the representation of the vulnerable function and decoders generate the corresponding repair patches. In particular, the relative position encoding of T5 used by VulRepair improves

the absolute position encoding of the vanilla transformer used in previous work (Chen et al., 2021). VulRepair was evaluated using the human-written repairs as ground-truth labels where a repair generated by VulRepair is considered correct if it is identical to the labels. Our previous work (Fu et al., 2022) has demonstrated that VulRepair substantially improves the performance of previous works for the AVR problem.

2.2 Example Usage

Consider the situation where an opened file contains one function written in C++, shown in Fig. 1. This example uses a real-world "out-of-bounds write" vulnerability (CWE, 2009) that is considered the most dangerous vulnerability in 2021 (CWE, 2021a). Figure 1 shows the "*unPremulSkImageToPremul*" function. AIBugHunterhas analyzed this and considered it as a vulnerable function. This is due to the variable type "size_t" being misused, causing an "out-of-bounds write" vulnerability (i.e., CWE-787) at line number 9.

As shown in Fig. 1, AIBugHunter first takes the whole function as an input and sends it to its backend models, LineVul (Fu and Tantithamthavorn, 2022b). The LineVul algorithm identifies that the 9th line of the "*unPremulSkImageToPremul*" function is a vulnerable line, as annotated by ①. Our approach further classifies this function as a vulnerability of CWE-787, shown as ②, with a Base type shown in ③.

This function is predicted as being of a high severity with a CVSS score of 7. This is shown as ④. Finally, we use our backend tool, VulRepair (Fu et al., 2022), to generate repair patches. This patch will be used to replace the vulnerable line. The developer can select this option by clicking on the "Quick Fix" button, shown as ⑤.

2.3 AIBugHunter Implementation

We developed our AIBugHunter extension using the VS Code Extension API provided by Microsoft to gain symbol information and utilize other VS Code IDE features. AIBugHunter is mainly written in TypeScript following the boilerplates provided by the VS Code extension generator. Being a plain VS Code extension, our package's operations are driven by a Node.js engine. In what follows, we introduce the front-end and back-end implementation details of AIBugHunter.

2.3.1 Front-End Implementation

The UI elements of AIBugHunter are defined by the VSCode API provided by Microsoft, the backend of which is Node.JS, and is interacted using the TypeScript language. When

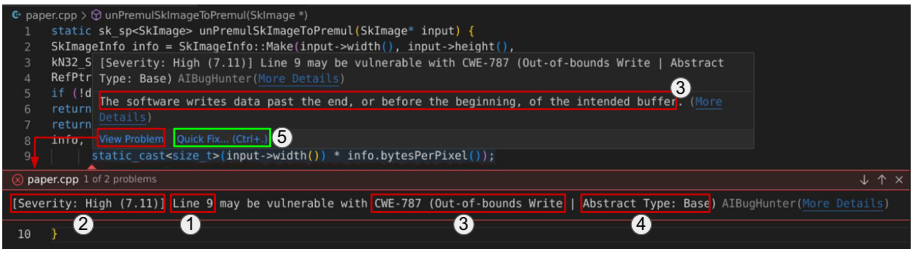


Fig. 1 The user interface of our AIBugHunter

a user opens a C/C++ file, AIBugHunter extracts each function from the source code using the "symbols" information available through VSCode API, and builds a list of parsed functions to be passed into the DL models introduced in the following section. The back-end will return the generated predictions using the API provided, and the relevant information is displayed on the UI as "diagnostics". This enables the extension to indicate the specific line to fix using underlines, display hover messages, provide a link to the CWE page, provide a "Quick Fix" button for repair candidates, and offer other error messages in the interface. AIBugHunter presents its vulnerability predictions and explanatory information as shown in Fig. 1.

2.3.2 Back-End Implementation

The back end consists of three main steps as summarized in Fig. 2. First, the data preparation step to construct data for DL models. Second, the DL models inference step for (1) locating line-level vulnerabilities, (2) classifying vulnerability types, (3) estimating vulnerability severity, and (4) suggesting repairs. Third, the post-prediction processing step is used to prepare information and present it in the UI.

Step 1: Data Preparation. When a C/C++ file is opened, VSCode automatically analyzes it and generates a "DocumentSymbol", which is a collection of symbols in the document such as variables, classes, and functions. We preserve only the collection of functions to construct a list of functions parsed from the document, where each parsed function undergoes formatting to remove comments. Note that all the modifications are recorded as a position delta to correctly map the prediction results to the original code.

Step 2: DL Model Inference. The model inference consists of two steps to obtain all the predictions to present in the front end as described below:

Step 2a. Send the list of functions from the data preparation step to the line-level vulnerability detection model's inference API endpoint (or flag in local inference mode). This will return a JSON which tells if individual functions are vulnerable or not (binary), and scores on each line of the function that determines which line the modifications are required to fix the vulnerability.

Step 2b. For functions that were predicted vulnerable in the previous step are now sent to three additional DL models. For each function, the first model will return a CWE-ID indicating the vulnerability type; the second model returns a CVSS score indicating the severity; and the third model returns an annotated piece of "patch code" as suggested repairs.

Step 3: Post-Prediction Processing. All the predictions from the model inference step are processed according to the user configuration. Additionally for functions predicted as vulnerable, we fetch the vulnerability description from MITRE ATT&CK Corporation (2022) based on the predictions to provide in-depth details of the predicted vulnerability and an accessible link to the official page of the specific CWE-ID. Finally, the organized information is displayed on the interface via the VSCode extension API.

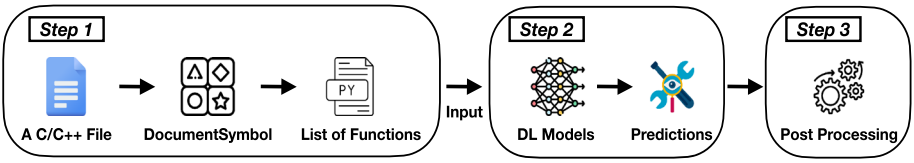


Fig. 2 The back-end implementation of our AIBugHunter

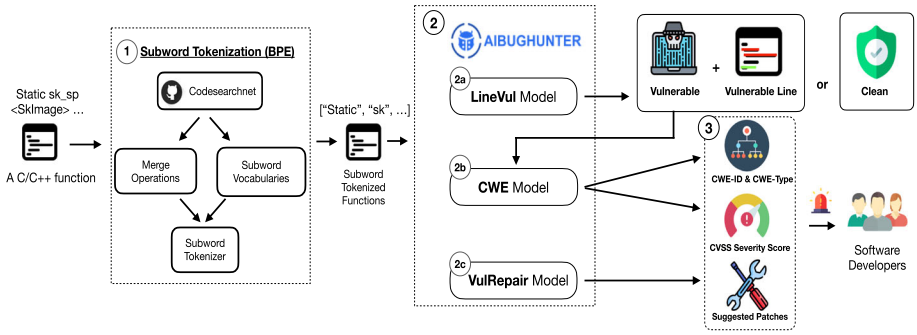


Fig. 3 An overview architecture of our approach

3 Learning to Predict Vulnerability Type and Severity

Our approach is a vulnerability prediction framework consisting of three different inference tasks. As shown in Fig. 3, given a C/C++ function, we first tokenize raw input into code tokens through Byte Pair Encoding (BPE) in Step ①. In Step ②, the tokenized function is then input to a LineVul model proposed in our previous work (Fu and Tantithamthavorn, 2022b) to predict vulnerable lines in the input function. If vulnerable lines exist in the function, our approach further predicts vulnerable types (i.e., CWE-ID and CWE-Type) and severity (i.e., CVSS severity score) of the vulnerable function as shown in step ②b. Furthermore, the vulnerable function is also input to the VulRepair (Fu et al., 2022) model to generate suggested repair patches as shown in step ②c. Finally in Step ③, AIBugHunter integrates the predictions from LineVul, our approach, and VulRepair models and present them to software developers in the IDE. We refer readers to our previous work (Fu and Tantithamthavorn, 2022b) for more technical details about BPE tokenization and the Transformer architecture of our approach.

In this section, we introduce key new components in our AIBugHunter approach over our prior works. Given a vulnerable function, we aim to predict its vulnerability types, where CWE-ID and CWE-Type are available categorizations provided by CWE (2006). CWE-Type is a higher-level of vulnerability category, where each CWE-Type may contain multiple similar CWE-IDs. Since CWE-ID and CWE-Type are highly correlated labels, we learn a shared CodeBERT model through multi-objective optimization as described in Section 3.1.

To predict the severity of vulnerabilities, we leverage a separate CodeBERT model instead of sharing the same model with the CWE classification task. This is due to (1) the CVSS severity score being a regression task that is different from the CWE classification; and (2) the CVSS severity score being determined using metrics provided by CVSS (2003) rather than based on vulnerability types. Thus the vulnerability types and severity scores are not necessarily highly correlated. In the following paragraphs, we describe in detail our approach for CWE classification, followed by severity regression.

3.1 Multi-Objective CWE Classification

In this section, we introduce our novel multi-objective approach that is used to predict the CWE-ID and CWE-Type of a vulnerable function.

3.1.1 Sequence Representation

As shown in Fig. 4, instead of using only one "[CLS]" token as a normal BERT model, our approach leverages two special tokens (one "[CLS]" token for CWE-ID classification and the other "[CLS_TYPE]" token for CWE-Type classification) along with a "[SEP]" token represents the end of a sequence. All of the special tokens are added during the subword tokenization process as described in our previous paper (Fu and Tantithamthavorn, 2022b).

The intuition behind using two special tokens for different tasks is the success of DeIT. Touvron et al. (2021) leveraged two special tokens to distill knowledge from a Transformer-based model for image classification tasks. In DeIT, one special token learns from the ground-truth labels while the other learns from the prediction generated by the teacher model to distill knowledge from it. Similarly, our CWE-ID class token is responsible for the CWE-ID prediction and learns from ground-truth labels of CWE-ID while our CWE-Type class token focuses on CWE-Type prediction and learns from ground-truth labels of CWE-Type.

3.1.2 Two Non-Shared Classification Heads

Similar to DeIT (Touvron et al., 2021), our approach uses two non-shared classification heads to generate predictions for two different tasks. Each classification head consists of two linear layers with dropout layers in between. Both heads rely on a softmax layer to activate the probabilities of each label which is the final prediction by our approach. The parameters of the two heads are non-shared, so they are able to map the representation of their own special token (i.e., the class token of CWE-ID and CWE-Type) to the prediction without conflicting with each other.

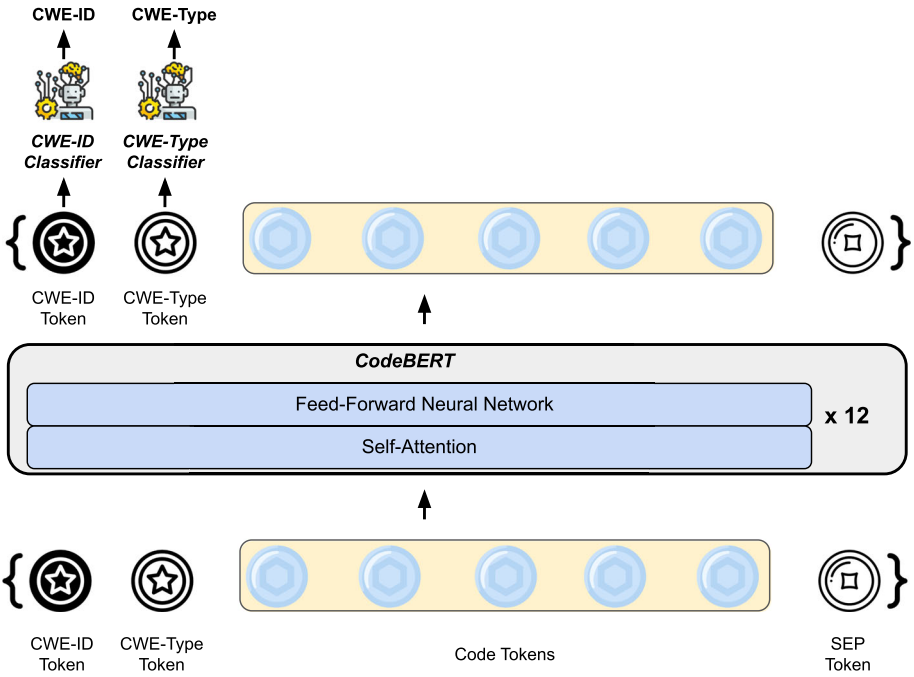


Fig. 4 An overview architecture of multi-objective CWE classification

The reasons for having two non-shared classification heads are (i) the number of classes for CWE-IDs is different from the number of classes for CWE-Type and (ii) we aim that each classification head can focus and specialize for each task (CWE-IDs or CWE-Type) to obtain better performances. Thus, we use separate non-shared heads to classify CWE-IDs and CWE-Type respectively. In concurring with our design, the experiment results in Fig. 6 show that our multi-objective method with a shared transformer architecture achieves the best performance among other baseline methods.

3.1.3 Multi-Objective Optimization

The problem solved by our approach can be considered as a multi-task learning (MTL) problem with an input space of X and a collection of task spaces $\{y^T\}$ where T is the number of tasks. Specifically, we have a large vulnerability dataset with data points $\{x_i, y_i^1, y_i^2\}_i \in [N]$ where x_i is a vulnerable function, y^1 is a CWE-ID label, y^2 is a CWE-Type label, and N is the number of data points.

To optimize the parameters of a multi-task model, we need to minimize both loss functions yielded by CWE-ID and CWE-Type labels so the model can infer both labels given the same input. Although the weighted summation is intuitively appealing as shown in (1), obtaining such weighted summation of loss functions for multi-task learning requires an expensive grid search over various scalings or the use of a heuristic such as Chen et al. (2018); Kendall et al. (2018) to find out the optimal values of W_1 and W_2.

$$\mathcal{L}_{Total} = W_1\mathcal{L}_{ID} + W_2\mathcal{L}_{Type} \tag{1}$$

Alternatively, our approach relies on the approach proposed by Sener and Koltun (2018) where the MTL problem is formulated as multi-objective optimization (MOO): optimizing a collection of possibly conflicting objectives. The training objective of our approach can be specified using a vector-valued loss L:

$$\min \ \ L(\theta^{sh}, \theta^1, \theta^2) = \min \ \ \left(\hat{\mathcal{L}}^1(\theta^{sh}, \theta^1), \hat{\mathcal{L}}^2(\theta^{sh}, \theta^2)\right) \tag{2}$$

where L is the combined cross-entropy (CE) loss (described in (4)) from both tasks computed by MOO, $\hat{\mathcal{L}}^1$ is the CE loss of the CWE-ID classification task, $\hat{\mathcal{L}}^2$ is the CE loss of the CWE-Type classification, θ^{sh} is parameters of shared 12-layer CodeBERT, θ^1 is parameters of the CWE-ID classification head, and θ^2 is parameters of the CWE-Type classification head as shown in Fig. 4. In short, we aim to minimize all of the parameters (i.e., $\theta^{sh}, \theta^1, \theta^2$) during gradient descent simultaneously.

To fulfill the objective (2) during the training phase of our approach, we leverage the same gradient update process as proposed by Sener and Koltun (2018). As shown in Algorithm 1, we first update the task-specific parameters (i.e., $\theta^1 \ and \ \theta^2$) through the gradient descent algorithm. We then apply the Frank-Wolfe solver (please refer to the original paper written by Sener and Koltun (2018) for details) to find a common descent direction to satisfy our training objective. We then apply the solution of the Frank-Wolfe solver to update the shared parameters (i.e., θ^{sh}) through the gradient descent algorithm. With such a gradient update process, all of the parameters (i.e., $\theta^{sh}, \theta^1, and \ \theta^2$) can be updated at the same time without conflicting with each other.

Algorithm 1 Gradient Update Equations for MTL.

for $t = 1$ to T **do**
  $\theta^t = \theta^t - \eta \nabla_{\theta^t} \hat{\mathcal{L}}^t(\theta^{sh}, \theta^t)$ ▷ Gradient descent on task-specific parameters(i.e., θ^1, θ^2)
end for
$\alpha^1, ..., \alpha^T = FRANKWOLFESOLVER(\theta)$ ▷ Solve to find a common descent direction
$\theta^{sh} = \theta^{sh} - \eta \sum_{t=1}^{T} \alpha^t \nabla_{\theta^{sh}} \hat{\mathcal{L}}^t(\theta^{sh}, \theta^t)$ ▷ Gradient descent on shared parameters(i.e., θ^{sh})

3.2 CVSS Severity Score Estimation

We used Version 3.1 of the CVSS score which has a range of 0-10. Below, we provide two concrete examples and present the difference between high and low severity scores. It can be seen that the CVSS scores were assigned based on different measures such as confidentiality impact, integrity impact, availability impact, access complexity, authentication, and gained access etc. A low CVSS score (see Example 1 in Fig. 5) usually has None or Partial impact to the confidentiality, integrity, and availability aspects of the software system. In contrast, a high CVSS score (see Example 2 in Fig. 5) usually corresponds to higher impact such as Complete impact where there could be total information disclosure, total compromise of system integrity, and total shutdown of the affected resource.

As the pre-trained CodeBERT model has been demonstrated its effectiveness for vulnerability-related tasks (Fu and Tantithamthavorn, 2022b; Hin et al., 2022), we rely on CodeBERT to obtain word embeddings for each vulnerable function. We add a linear layer as a regression head on top of CodeBERT, which returns one value for each vulnerable function as a severity score prediction. We minimize the Mean Square Error (MSE) loss as described in (3) to train the severity regression model:

$$\mathcal{L}_{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{3}$$

where y_i is a ground-truth severity score and $\hat{y}_i$ is a prediction of the model.

Example 1 - A vulnerable function with a low CVSS score.

Example Code	void ahci_uninit(AHCIState *s)
	{
	g_free(s->dev);
	}

CVSS Score	1.9
Confidentiality Impact	None (There is no impact to the confidentiality of the system.)
Integrity Impact	None (There is no impact to the integrity of the system)
Availability Impact	Partial (There is reduced performance or interruptions in resource availability.)
Access Complexity	Medium (The access conditions are somewhat specialized. Some preconditions must be satisfied to exploit)
Authentication	Not required (Authentication is not required to exploit the vulnerability.)
Gained Access	None
Vulnerability Type(s)	Denial Of Service

Example 2 - A vulnerable function with a high CVSS score.

Example Code	const Chapters::Display* Chapters::Atom::GetDisplay(int index) const
	{
	if (index < 0)
	return NULL;
	if (index >= m_displays_count)
	return NULL;
	return m_displays + index;
	}

CVSS Score	10.0
Confidentiality Impact	Complete (There is total information disclosure, resulting in all system files being revealed.)
Integrity Impact	Complete (There is a total compromise of system integrity. There is a complete loss of system protection, resulting in the entire system being compromised.)
Availability Impact	Complete (There is a total shutdown of the affected resource. The attacker can render the resource completely unavailable.)
Access Complexity	Low (Specialized access conditions or extenuating circumstances do not exist. Very little knowledge or skill is required to exploit.)
Authentication	Not required (Authentication is not required to exploit the vulnerability.)
Gained Access	None
Vulnerability Type(s)	Denial Of ServiceExecute CodeOverflowMemory corruption

Fig. 5 Two concrete examples of high and low CVSS severity scores

4 A Quantitative Evaluation of AIBugHunter

In this section we present a quantitative evaluation of AIBugHunter. We present our three research questions, our studied dataset, our experimental setup, and answers to our first three research questions along with their experimental results.

4.1 Research Questions

The empirical evaluation of LineVul and VulRepair backend components used in our AIBugHunter have been presented in our previous works. To evaluate our new proposed approach for vulnerability type and severity prediction, we conduct a new set of experiments to compare our proposed method with existing baseline approaches. Through an extensive evaluation of our approach on 8,783 C/C++ vulnerable functions including 88 different types of vulnerabilities, we answer the following three research questions:

RQ1: **How accurate is our approach for predicting vulnerability IDs (i.e., CWE-IDs)?** We focus on CWE-ID multi-class classification and compare our approach with four baseline models. Our approach achieves a multiclass accuracy of 0.65, which is 10%-141% more accurate than other baseline approaches with a median improvement of 86%.

RQ2: **How accurate is our approach for predicting vulnerability types (i.e., CWE abstract types)?** We focus on CWE-Type multiclass classification and compare our approach with the same four baseline models described in RQ1. Our approach achieves a multiclass accuracy of 0.74, which is 3%-45% more accurate than other baseline approaches with a median improvement of 23%.

RQ3: **How accurate is our approach for predicting vulnerability severity?**We focus on the CVSS severity score regression task and compare our approach with 3 baseline approaches. Our approach achieves an MSE of 1.8479 and an MAE of 0.8753, which are better than the baseline approaches.

4.2 Studied Dataset

To ensure a fair comparison with the previous work, we use the existing benchmark dataset (Fan et al., 2020). We did not further parse data from 2020 to 2022 as previous studies did not publish scripts to collect datasets. When implementing our data collection scripts, the collected data may not be the same as used by previous works, posing potential threats to internal validity. Nevertheless, we encourage future studies to evaluate our approach on more recent datasets once available.

As this article is an extended version of our previous work (Fu and Tantithamthavorn, 2022b), we use the same experimental dataset (i.e., Big-Vul (Fan et al., 2020)) to evaluate the performance of our approach on vulnerable functions. The Big-Vul dataset is collected from 348 open-source Github projects, which includes 91 different CWEs from 2002 to 2019, and nearly 11k of C/C++ vulnerable functions. Given a large number of vulnerable functions from diverse projects and timeframes, the Big-Vul dataset is a suitable dataset to evaluate whether our vulnerability classification and CVSS score estimation approaches can generalize well to the diverse samples. Other vulnerability datasets such as the Devign dataset (Zhou et al., 2019) are not selected because the CWE-ID and CVSS score information are not provided.

4.3 Experimental Setup

Data Splitting. Similar to our previous work (Fu and Tantithamthavorn, 2022b), we split the dataset into 80% of training data, 10% of validation data, and 10% of testing data. We randomly split the data into three similar distributions so different vulnerability types are equally represented in training, validation, and testing sets. We also ensure that CWE-IDs appearing in the testing set should also appear in the training set.

Data Preprocessing. To satisfy the scenario of CWE classification tasks and the severity score regression task, we only keep the vulnerable functions with known CWE-ID, CWE-Type, and CVSS scores. Table 1 presents the descriptive statistics of our studied dataset after removing non-vulnerable functions. After data filtering, we keep 8,783 C/C++ functions with 88 different CWE-IDs, 6 different CWE-Types, and CVSS scores (labelled based on CVSS version 3.1 (NVD, 2019)) ranging from 1.2-10.0. Note that CWE-IDs and CWE-Types are many-to-one mappings where each CWE-ID has one CWE-Type but each CWE-Type may correspond to many CWE-IDs.

Multi-objective Classification Model Implementation. We leverage the pre-trained CodeBERT model as a backbone encoder to generate the shared representation of CWE-ID and CWE-Type classification tasks using the Transformers library in Python. We then add two classification heads on top of the backbone, one predicting the CWE-ID and the other predicting the CWE-Type. Note that the parameters in the backbone are shared by both tasks, however, the parameters in each classification head are task-specific. We leverage two cross-entropy loss functions (i.e., CE_{ID} and CE_{Type}) and implement the multi-objective optimization process based on the implementation provided by Sener and Koltun (2018) to fine-tune the CodeBERT model under the multi-task setting of CWE-ID and CWE-Type. The multi-objective loss is implemented as described in Section 3.1.3 where each cross-entropy loss is implemented based on (4). We use the PyTorch library to update the model and optimize the loss functions.

$$\mathcal{L}_{CE}(p, q) = -\sum_{x} p(x) \ \log_q(x) \tag{4}$$

Severity Regression Model Implementation. We leverage the pre-trained CodeBERT model with a regression head for CVSS score regression. The model is implemented with the Transformers library and trained using the PyTorch library. We use the Mean Squared Error (MSE) loss to update the model during training as (3).

Hyperparameter Settings for Fine-Tuning. We use the default setting of CodeBERT, i.e., 12 Transformer Encoder blocks, 768 hidden sizes, and 12 attention heads. We follow the same fine-tuning strategy provided by Feng et al. (2020). During training, the learning rate is set to 2e-5 with a constant schedule. We use backpropagation with AdamW optimizer (Loshchilov and Hutter, 2018) which is widely adopted to fine-tune Transformer-based models to update

Table 1 Descriptive statistics of our studied datasets that describes the distribution of the severity score, and the distributions of cardinalities of CWE-ID and CWE-Type

	Mean	Median	Std.	1st Quantile	3rd Quantile	Min	Max
CWE-ID Cardinality	100	9	281	3	49	1	2127
CWE-Type Cardinality	1255	415	1491	138	1827	1	4437
Severity Score	6.18	6.8	1.95	4.6	7.5	1.2	10.0

the model and minimize the loss function. The best model is selected based on the validation data, which will perform inference on testing data as final evaluation results.
Execution Environment All of the experiments are run under Ubuntu 20.04 system with an AMD Ryzen 9 5950X CPU with 16C/32T, 64GB RAM, and an NVIDIA GTX 3090 GPU (24GB of memory).

4.4 Experimental Results

RQ1: How accurate is our approach for predicting vulnerability IDs (i.e., CWE-IDs)?

Approach. To answer this RQ, we focus on CWE-ID multi-class classification and compare our approach with four baseline models, described as follows:

1. BERT models pre-trained on natural language (i.e., BERT-base (Devlin et al., 2019)), which have been adopted for CWE classification tasks (Das et al., 2021; Wang et al., 2021).
2. BERT models pre-trained on programming language (i.e., CodeBERT (Feng et al., 2020)), which have been applied to software vulnerability prediction (Fu and Tantithamthavorn, 2022b; Thapa et al., 2022; Yuan et al., 2022).
3. BoW+RF uses bag of words as features together with a Random Forest model for CWE-ID classification (Aota et al., 2020; Wang et al., 2020).
4. BoW+NB uses bag of words as features together with a Naive Bayes model for CWE-ID classification (Na et al., 2016).

The pre-trained BERT-based language models are selected because previous studies such as Wang et al. (2021) and Zhu et al. (2022) have used them to achieve promising results on the CWE-ID classification tasks. The Random Forest and Naive Bayes models are selected because they are important machine learning-based methods for CWE-ID classification tasks proposed in previous studies.

We evaluate our approach based on the multiclass accuracy which is computed as $\frac{\text{Correctly Predicted Testing Data}}{\text{Total Testing Data}}$.

Results. Figure 6 presents the experimental results of our approach and the four baseline approaches according to the multiclass accuracy.

Our approach achieves an accuracy of 0.65, which is 10%-141% more accurate than other baseline approaches with a median improvement of 86%. These results confirm that our approach is more accurate than other baseline approaches for CWE-ID classification.

We use CodeBERT as our backbone architecture, however, our approach outperforms the CodeBERT model by 6%. Our approach can learn knowledge from two perspectives based on the class of CWE-ID and the class of CWE-Type where both classes describe the same vulnerable function. The correlated information between the two kinds of labels further benefits our method. On the other hand, the CodeBERT method only learns from CWE-ID labels. In other words, the comparison between our approach and CodeBERT highlights the advancement of using labels from both tasks (i.e., CWE-ID and CWE-Type) with multi-objective optimization. In short, our results demonstrate that **the multi-task learning with multi-objective optimization using both CWE-ID and CWE-Type labels outperforms other baselines that are only trained using CWE-ID label.**

We analyze the performance of our approach on 879 testing samples. First, 567 of 879 (65%) are correctly predicted. On the other hand, 312 samples are misclassified. Among the 312 misclassified samples, we find that 89 of 312 (35%) were predicted as close to the

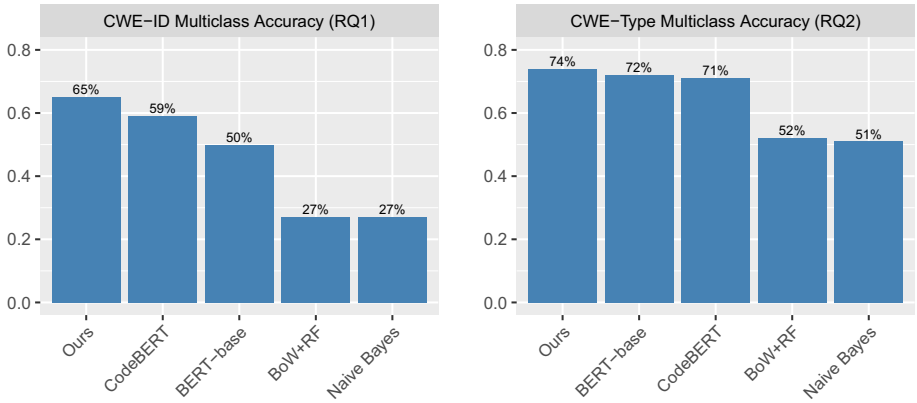


Fig. 6 (RQ1 and RQ2) The Multiclass Accuracy of our approach and four other baselines. (↗) Higher Multiclass Accuracy = Better

ground truth (i.e., incorrectly predict the CWE-ID, but correctly predict the CWE-Type). This means our approach can at least correctly predict the vulnerability type for 75% $\frac{(567+89)}{879}$ of testing samples (outperform all other baselines), highlighting the potential usefulness of our approach in practice.

To further investigate whether our approach can classify dangerous real-world vulnerabilities, we further evaluate our approach on Top-25 most dangerous CWE-IDs (CWE, 2021a) in the testing set to understand the significance of our approach in the practical usage scenario. Table 2 presents the accuracy of our approach on Top-25 most dangerous CWE-IDs. **We find**

Table 2 (RQ1 Discussion) The Accuracy of our approach for the Top-25 Most Dangerous CWEs (https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html)

Rank	CWE-ID	Name	Accuracy	Proportion
1	CWE-787	Out-of-bounds Write	43%	9/21
2	CWE-79	Cross-site Scripting	29%	2/7
3	CWE-125	Out-of-bounds Read	67%	44/66
4	CWE-20	Improper Input Validation	66%	71/107
7	CWE-416	Use After Free	52%	15/29
8	CWE-22	Path Traversal	0%	0/4
9	CWE-352	Cross-Site Request Forgery	0%	0/1
12	CWE-190	Integer Overflow	68%	21/31
14	CWE-287	Improper Authentication	0%	0/2
15	CWE-476	NULL Pointer Dereference	41%	7/17
17	CWE-119	Improper Restriction	79%	180/228
18	CWE-862	Missing Authorization	0%	0/1
20	CWE-200	Exposure of Sensitive Info	62%	26/42
22	CWE-732	Incorrect Permission Assignment	86%	6/7
23	CWE-611	Improper Restriction	50%	1/2
25	CWE-77	Improper Neutralization	0%	0/1
			67%	382/566

that our approach can correctly predict 67% of the vulnerable functions affected by the Top-25 most dangerous CWE-IDs, which is better than the average performance of our approach(i.e., 65%).

In addition, Fig. 7 presents our method's accuracy for each CWE-ID in the testing set. It can be seen that the accuracy of our approach is not highly correlated to training or testing data frequencies. Our approach performs well on some of the CWE-IDs with low frequencies such as CWE-754 while having challenges generalizing to other low frequencies CWE-IDs such as CWE-94. However, those CWE-IDs that cannot be identified by our approaches are all CWE-IDs rarely occur in the dataset. This highlights the challenge of imbalanced data in the CWE-ID classification task where some CWE-IDs are common (e.g., CWE-119) and easy to collect while other CWE-IDs can be rare (e.g., CWE-369) and difficult to collect. Those rare CWE-IDs are more prone than common CWE-IDs to be misclassified by our approach due to not-enough training samples. Thus, future researchers may explore new techniques to solve this imbalance problem.

Last but not least, we found that the complexity of vulnerabilities may also affect the performance of our approach. In particular, our approach achieves an accuracy of 86% for the least complex CWE-IDs that are under the CWE-Type of the "class weakness". Class weaknesses typically describe issues in terms of 1 or 2 of the following dimensions: behavior, property, and resource (CWE, 2021b). For instance, the class weakness of "Uncontrolled Resource Consumption" (CWE-400) describes an issue (Uncontrolled) with a behavior (Consumption) associated with any type of resource. However, our approach only achieves an accuracy of 51% for the most complex CWE-IDs that are under the CWE-Type of the "variant weakness". Variant weaknesses typically describe issues in terms of 3 to 5 of the following dimensions: behavior, property, technology, language, and resource (CWE, 2021c). For instance, the variant weakness of "Use After Free" (CWE-416) describes an issue (Referencing memory after it has been freed) with a specific resource (Memory) with specific languages (C/C++). These results highlight the challenge of classifying those complex vulnerability types such as the variant weakness.

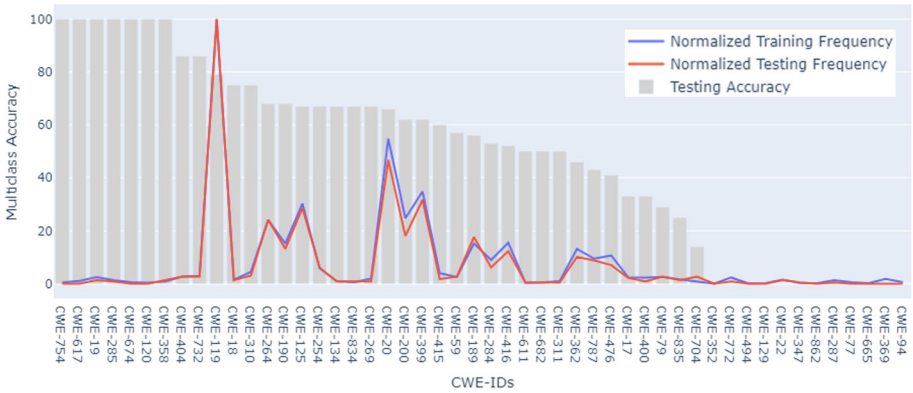


Fig. 7 (RQ1 Discussion) Our method's Multiclass Accuracy of CWE-ID classification for each CWE-ID in the testing set. The accuracy is shown in percentage

RQ2: How accurate is our approach for predicting vulnerability types (i.e., CWE abstract types)?

Approach. To answer this RQ, we focus on CWE-Type multiclass classification and compare our approach with the same four baseline models described in RQ1. We adopt the same measure as mentioned in RQ1 to evaluate our approach.

Results. Figure 6 presents the experimental results of our approach and the four baseline approaches according to the multiclass accuracy.

Our approach achieves an accuracy of 0.74, which is 3%-45% more accurate than other baseline approaches with a median improvement of 23%. These results confirm that our approach is more accurate than other baseline approaches for CWE-Type classification.

Our approach performs the best, which is the only model that leverages both CWE-Type and CWE-ID labels during training. The improvement of our approach compared with other baseline approaches is 3%-45% which is not as significant as the improvement demonstrated in RQ1 (10%-141%). The difference in improvements implies that while leveraging both labels can benefit performance for both CWE-ID and CWE-Type classification tasks, our method is more beneficial for the CWE-ID classification.

In addition, Table 3 presents detailed accuracy for each CWE-Type. It can be seen that the performance depends on the number of samples and varies for each type. Nevertheless, our approach has the best overall accuracy and is the only approach that achieves at least 50% of accuracy for each CWE-Type.

In Section 3.1.3, we proposed leveraging Multi-Objective Optimization (MOO) instead of taking the weighted summation of loss functions for gradient descent. We now further evaluate whether MOO can help our approach learn better on multi-task learning. Specifically, we compare our approach (with using MOO) with a variant method (without using MOO) that leverages a weighted summary of the loss function during gradient descent. The loss function of the weighted summary version of our approach is described as (1). We set $W1$ and $W2$ to 0.5 so both tasks contribute equally to the total loss. To ensure a fair comparison, we only switch the MOO component of our approach and adopt the same model architecture, hyperparameters, and training strategy for both approaches.

Figure 8 presents the accuracy of our approach, the variant approach, and the single-task CodeBERT. **We find that the multi-task learning framework is always better than the CodeBERT which only learns from a single task, and our approach performs the best on both tasks.** Our approach can achieve an accuracy of 63%-65% and 73%-74% on CWE-ID and CWE-Type classification respectively while single-task CodeBERT only achieves an accuracy of 59% and 71%. This result confirms that (1) leveraging multi-task learning on two correlated tasks may benefit the model performance on both tasks and (2) the MOO approach

Table 3 (RQ2) The Multiclass Accuracy of each CWE type for all of the approaches evaluated in RQ2

Methods	CWE Abstract Types						
	Class	Category	Variant	Base	Pillar	Deprecated	Overall
Ours	85.55%	67.01%	62.86%	60.8%	52.94%	50%	74%
BERT-base	84.42%	68.53%	34.29%	55.11%	64.71%	50%	72%
CodeBERT	78.78%	63.95%	42.86%	71.02%	52.94%	30%	71%
BoW+RF	74.49%	29.44%	14.29%	33.52%	5.88%	10%	52%
Naive Bayes	99.77%	3.55%	–	–	–	–	51%

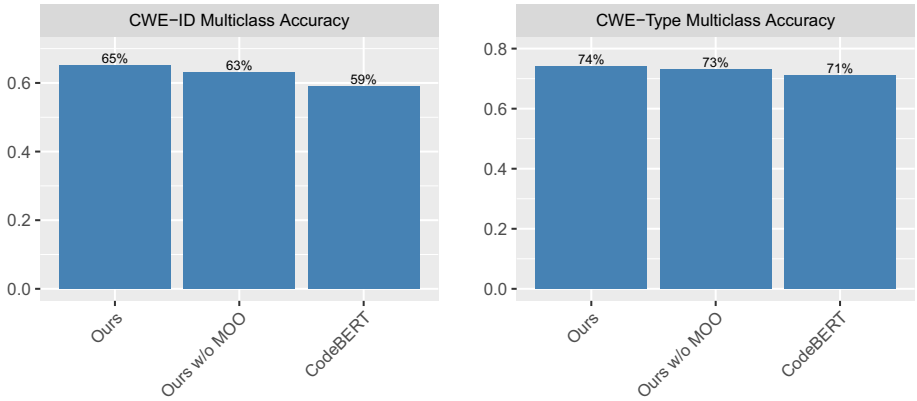


Fig. 8 The Multiclass Accuracy of our approach, our approach w/o MOO, and single-task CodeBERT. (↗) Higher Multiclass Accuracy = Better

used by our approach can learn a model with higher accuracy than the weighted summary approach.

Furthermore, we analyze the impact of function length on our approach for CWE-ID and CWE-Type classification. According to Table 4, when the function length is short, e.g., consisting of 0-100 tokens, our tool can have better 84% and 85% accuracy respectively. However, the performance decreases as functions become longer, for functions consisting of more than 500 tokens, the accuracy becomes 57% and 72% respectively. These results highlight the challenge of tackling long sequences for vulnerability classification tasks. Thus, future researchers should further explore techniques that can classify difficult longer vulnerable functions.

RQ3: How accurate is our approach for predicting vulnerability severity?

Approach. To answer this RQ, we focus on the CVSS severity score regression task and compare our approach with 3 baseline approaches as follows:

1. BERT models pre-trained on natural language (i.e., BERT-base (Devlin et al., 2019)).
2. BoW+RF uses bag of words as features together with a Random Forest model for severity score regression (Aota et al., 2020; Wang et al., 2020).

Table 4 (RQ2 Discussion) The analysis of how different function lengths affect the multi-class accuracy of our approach for CWE-ID and CWE-Type prediction tasks

Function Length (Tokens)	CWE-ID Accuracy	CWE-Type Accuracy
0-100	84%	85%
101-200	72%	81%
201-300	69%	71%
301-400	60%	69%
401-500	61%	70%
>500	57%	72%

Note. Function lengths counted by number of tokens in a tokenized function

3. BoW+LR uses bag of words as features together with a Linear Regression model for severity score regression.

We evaluate our approach based on Mean Squared Error (MSE) 3 and Mean Absolute Error (MAE) where MSE penalizes the predictions that are far from true values through the square of Euclidean distance and MAE measures the exact distance between predicted values and ground-truth values as $MAE = \frac{1}{n}\sum_{i=1}^{n} \mid y_i - \hat{y}_i \mid$.

Results. Figure 9 presents the experimental results of our approach and the three baseline approaches according to the MSE and MAE loss.

Our approach achieves an MSE of 1.8479 and an MAE of 0.8753, which are better than all of the baseline approaches. These results confirm that our approach can predict the most accurate severity scores.

Our approach **achieves 0.1440 and 0.7674 less MSE than the baselines using traditional Bag of Words and Machine Learning algorithms (i.e., BoW+RF, BoW+LR)**. This result highlights the advancement of leveraging a BPE tokenization and pre-trained word embedding with a Transformer-based architecture. The word embeddings with the self-attention mechanism (Vaswani et al., 2017) in the Transformer model can learn the semantic features of input source code while the traditional BoW approach only considers the word frequencies when representing source code. Thus, our approach learns a more accurate mapping between a vulnerable function and its corresponding severity score.

Our approach **achieves 0.0141 less MSE and 0.0193 less MAE than the BERT-base (pre-trained on natural language) model**. This result confirms that leveraging a BERT architecture pre-trained using programming languages (our approach) can improve the one pre-trained using natural language.

To investigate whether each approach can accurately predict the severity of vulnerable functions, we map the CVSS score into four classes of severity based on the CVSS protocol, i.e., low, medium, high, and critical as detailed in Fig. 10. Our approach achieves an accuracy of 0.71, which is 6%-18% better than other baselines. The result confirms that our approach can correctly predict the severity class for 71% of vulnerable functions in testing data.

To further investigate our approach's performance, we present our approach's confusion matrix in Fig. 10. It can be seen that our approach neither estimates low severity as critical (last row, first column) nor estimates critical severity as low (first row, last column). Furthermore,

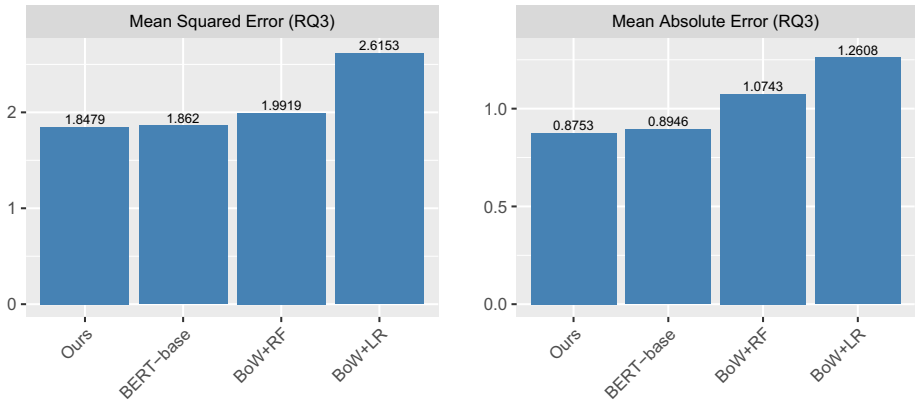
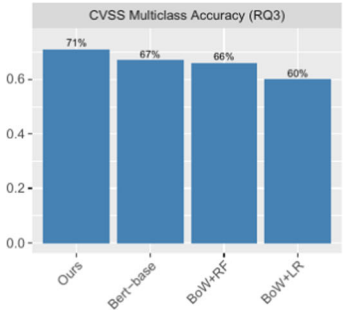


Fig. 9 (RQ3) The Mean Squared Error (MSE) and Mean Absolute Error (MAE) of our approach and three other baselines. (↘) Lower MSE, MAE = Better



Confusion Matrix of Our Approach

	Low	Medium	High	Critical
Low	**33**	22	6	0
Medium	6	**374**	99	3
High	1	88	**144**	4
Critical	0	14	14	**71**

Fig. 10 (RQ3 Discussion) The left part is the multi-class accuracy of the CVSS score for each approach evaluated in RQ3. The right part is the confusion matrix of our approach. Note that each class of CVSS is directly mapped from the CVSS score as shown at the bottom of the confusion matrix table

the last column shows that when our approach predicts a critical severity, the accuracy is 91%. Nevertheless, the most common error of our approach is predicting samples to the close class such as estimating a medium severity as high severity, which highlights the challenge of the CVSS severity estimation task.

5 Qualitative Evaluations of AIBugHunter

We conducted qualitative evaluations including (1) a survey study to obtain software practitioners' perceptions of our AIBugHunter tool; and (2) a user study to investigate the impact that our AIBugHunter could have on developers' productivity in security aspects, to answer the following research question:

RQ4: According to our survey study, each kind of vulnerability prediction provided by our AIBugHunter is perceived as useful by 47%-86% of participated software practitioners. Furthermore, 90% of participants consider adopting our AIBugHunter if it is freely available in an IDE without conditions. Moreover, our user study shows that our AIBugHunter could save developers' time spent on security analysis that could enhance security productivity during software development.

5.1 A Qualitative Survey Study

Following (Kitchenham and Pfleeger, 2008), we conduct our study according to the following steps: (1) design and develop a survey, (2) recruit and select participants, and (3) verify data and analyze data. We explain the detail of each step below.

5.1.1 Survey Design

Step 1: Design and development of the survey: We design our survey as a cross-sectional study where participants provided their responses at one fixed point in time. The survey consists of 6 closed-ended questions and 5 open-ended questions. For closed-ended questions, we use multiple-choice questions and a Likert scale from 1 to 5. Our survey consists of two parts: preliminary questions and developers' perceptions of AI-based software vulnerability predictions.

Part I: Demographics. The survey starts with a question, ("(D1) What is your role in your software development team?"), to ensure that our survey results are obtained from the right target participants. Then, the survey is followed by a demographics question, ("(D2) What is the level of your professional experience?"), to ensure our survey is distributed across software practitioners with different levels of professional experience.

Part II: Vulnerability predictions generated by our AIBugHunter. We then ask about software practitioners' perceptions of AI-based vulnerability predictions. Specifically, we present an example visualization of a prediction generated by AIBugHunter as shown in Fig. 1. Then, we ask four questions, i.e., ("(Q1) How do you perceive the usefulness of the recommended location of the vulnerability (i.e., line number)?"), ("(Q2) How do you perceive the usefulness of the vulnerability severity prediction?"), ("(Q3) How do you perceive the usefulness of the vulnerability type prediction (i.e., CWE-ID and CWE-Type)?"), and ("(Q4) How do you perceive the usefulness of the "Quick Fix" button which will replace a vulnerable line with the suggested repair on click?") Each question is followed by an open question for the rationale.

We use Google Form to conduct our survey in an online setting. Each participant is provided with an explanatory statement on the landing page that describes the purpose of the study, why the participant is chosen for this study, possible benefits and risks, and confidentiality. The survey takes approximately 10 minutes to complete and is completely anonymous. Our survey has been rigorously reviewed and approved by the Monash University Human Research Ethics Committee (MUHREC ID: 35047).

Step 2: Recruit and select participants: We recruit developers that have software development experience through LinkedIn and Facebook platforms. We send a survey invitation to the target groups via direct message. To mitigate potential bias introduced by the participant groups, we selected participants with different software engineering-related professions, different lengths of professional experience, and different organizations. Finally, we obtained a total of 22 responses over a two-week period of recruitment.

Step 3: Verify data and analyze data: To verify the completeness of the response in our survey (i.e., whether all questions were appropriately answered), we manually review all of the open-ended questions. Finally, we obtain a set of 21 valid responses. We present the results of closed-ended responses in a Likert scale with stacked bar plots. We manually analyze the responses to the open-ended questions to better understand the in-depth insights.

5.1.2 Survey Results

Figure 11 summarizes the survey results, we describe each question in detail in the following.
Respondent demographics. Figure 12 presents the overall respondent demographic. In terms of the profession of the participants, 48% ($\frac{10}{21}$) of them are full-stack software engineers, 19% ($\frac{4}{21}$) of them are security analysts, while the other 33% are software engineering researchers, machine learning engineers, etc. In terms of the level of their professional experience, 52% ($\frac{11}{21}$) of them have less than 5 years of experience, 24% ($\frac{5}{21}$) have 6-10 years of experience, while the other 24% have more than 10 years of experience.
(Q1) How do you perceive the usefulness of the recommended location of the vulnerability (i.e., line number)?
Findings. 86% of the respondents perceived that the prediction of the vulnerability location is useful due to various reasons:

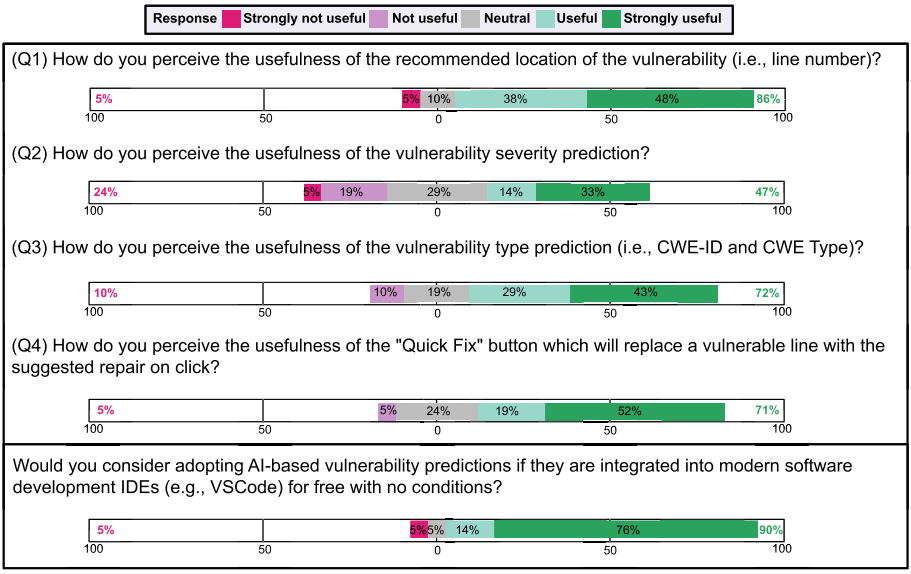


Fig. 11 (Q1-Q4) A summary of the survey questions and the results obtained from 21 participants

- Explicitly localize the vulnerability (R1: *I think it is useful to know which line does the vulnerability locate.*, R7: *It can help you quickly identify where the vulnerability is.*, R15: *It is helpful to know the reason for vulnerability and line number.*)
- Reduce time spent on code review (R4: *Speed to fix for developers.*, R11: *This would decrease code review time when I want to check for security breaches.*, R12: *This is something it would take me a lot of time debugging to figure out.*)
- Support debugging process (R6: *Been useful in debugging code*, R9: *Helps with quick resolution of bugs/vulnerabilities.*)

(Q2) How do you perceive the usefulness of the vulnerability severity prediction?
Findings. 47% of the respondents perceived that the prediction of the vulnerability severity score is useful due to various reasons:

- Prioritization of vulnerability repairs (R3: *Having it will allow me to prioritize fixing high-impact vulnerabilities before looking at things that don't matter as much.*, R8:

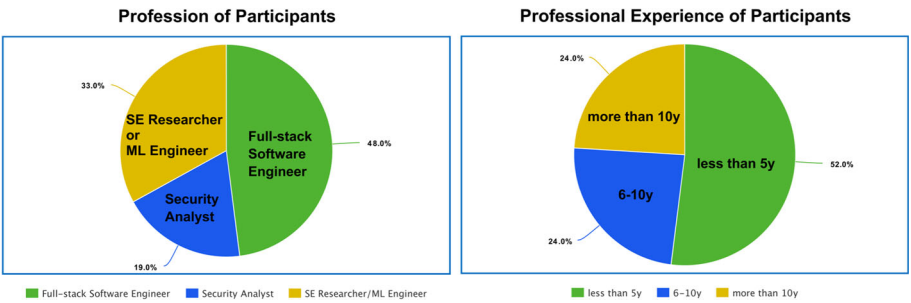


Fig. 12 The demographics of our survey participants in terms of their profession and professional experience

Vulnerability score will help me prioritize the fix., R11: *This would help me prioritize which part of the code I should fix first.*)
- Risk management (R4: *Modeling business risk is very useful for overall software development planning.*, R9: *Determines the magnitude of the vulnerability against the risks involved.*, R13: *Just like vulnerability scanning tools, it is helpful to know how bad it is and decide the further steps, so yes, it is useful.*)

(Q3) How do you perceive the usefulness of the vulnerability type prediction (i.e., CWE-ID and CWE-Type)?
Findings. 72% of the respondents perceived that the predictions of CWE-ID and CWE-Type are useful due to various reasons:

- Help understands the vulnerability (R3: *It is important to understand what the vulnerability is before you fix it.*, R9: *Helps to identify common weaknesses and resolve them easily.*, R11: *This would help me understand which kind of security breach I might be facing.*, R15: *It helps understand the problem.*, R19: *Easier to know what the problem is.*)
- Align with security practices (R4: *Aligns well with security practices.*, R13: *A quick classification would help solve the problem more efficiently.*)

(Q4) How do you perceive the usefulness of the "Quick Fix" button which will replace a vulnerable line with the suggested repair on click?
Findings. 71% of the respondents perceived that the "Quick Fix" button that suggests the vulnerability repair is useful due to various reasons:

- Reduce time spent on vulnerability repairs (R6, R15: *It saves time.*, R9: *Save hours to resolve an issue.*, R11: *This would save a lot of time. I can also modify the suggested codes if I want to.*)
- Help with vulnerability repair implementation (R12: *Having a potential fix helps me think through the fix I would like to implement even if I do not use the suggested fix.*)

Summary. Our survey study with 21 software practitioners found that all kinds of vulnerability predictions provided by our AIBugHunter are perceived as useful. Specifically, the vulnerable line prediction reduces the time required to locate vulnerability while severity score prediction helps developers prioritize their workloads. Moreover, the prediction of the vulnerability type helps developers understand the vulnerability and the repair recommendation suggested by the "Quick Fix" button helps developers come up with repair implementation. Finally, we found that **90% of the respondents consider adopting an AI-based vulnerability prediction approach such as AIBugHunter** if it is publicly available for free in a modern IDE (e.g., Visual Studio Code), highlighting the practical need for our AI-based vulnerability prediction approach.

5.2 A Preliminary User Study

We conducted a preliminary user study to assess the impact that AIBugHunter may have on developers' productivity in security analysis. To do so, we choose *single-subject experimental designs* as our research methodology—a type of research methodology characterized by repeated assessment of a particular phenomenon (often a behavior) over time. The single-subject experiment is useful when researchers are attempting to observe the behavior of an individual or a small group of individuals and wishes to document that observation. In

particular, we run the user study with two groups, i.e., a control group (i.e., the group of participants that do not have access to AIBugHunter) and a treatment group (i.e., the group of participants that have access to AIBugHunter). First, we assign a vulnerable C/C++ function to a participant to perform given tasks. The tasks are to locate, estimate severity, explain its type, and suggest repairs. In a well-designed experiment, all variables apart from the treatment should be kept constant between the two groups, allowing us to correctly measure the entire effect of the treatment without interference from confounding variables. With this methodology, our results will not be affected by different participants' expertise and task difficulty (which is commonly affected by randomized control trials). In what follows, we illustrate our user study design followed by the results.

5.2.1 User Study Design

(Step 1) Design and develop a user study. Our user study is face-to-face where each participant participated individually. Our user study consists of three parts, (1) demographic questions, (2) user study, and (3) survey questions to seek feedback after using AIBugHunter.

In the demographic questions, we asked about the participants' education and experience in software engineering and software security to ensure that we approach the right target group of participants.

In the user study, we used a real-world vulnerable C function in our experiment (Renaud, 2018). The *jpeg_size* function from a PDF generation library caused a buffer overread vulnerability (i.e., CWE-125) due to an inappropriate data bounding check. In particular, we designed our main experiment into two parts. The participants were asked to diagnose the vulnerable C function without using our AIBugHunter tool in the first part while they were asked to diagnose the vulnerable function with the help of our AIBugHunter tool in the second part. In each part, the participants were required to complete four tasks within 15 minutes, i.e., (1) locate vulnerability, (2) explain the vulnerability type, (3) estimate vulnerability severity, and (4) suggest repairs.

In the survey questions, we asked about the participants' satisfaction with our AIBugHunter using a Likert scale ranging from 1 to 5 followed by an open-ended question for justification. Last but not least, our experiment has been rigorously reviewed and approved by the Monash University Human Research Ethics Committee (MUHREC ID: 36037).

(Step 2) Recruit and select participants. We recruited software developers and researchers that have software engineering and/or software security expertise. To ensure the diversity of our participants, we select participants from a diverse set of professional experiences and occupations. Finally, we recruited a total of 6 participants to participate in our user study. Each participant will receive a gift card of $20 as a token of appreciation.

(Step 3) Conduct the user study. We conducted the user study as mentioned in Step 1. We also video-recorded during the user study with permission from the participants. Finally, for each participant, we analyzed the time spent on each task between the two groups (i.e., control vs treatment). Then, we manually analyzed the responses to the open-ended questions to better understand the in-depth insights from the participants.

5.2.2 User Study Results

Participant Demographics. The education level of our participants varies from bachelor, master, to Ph.D. degrees, while the professional experience in software engineering and

software security varies from a few months to 10 years, ensuring that the results are not bounded to specific groups of participants.

Main Findings. Our AIBugHunter can reduce the time spent on detecting, locating, estimating, explaining, and repairing vulnerabilities from 10-15 minutes to 3-4 minutes (see Fig. 13). Without using AIBugHunter, the results show that the majority of the participants cannot provide accurate answers to the given tasks, which indicates that the vulnerability analysis task is challenging and time-consuming. With the use of AIBugHunter, the results show that all of the participants were able to provide accurate answers to the given tasks within 4 minutes. This finding implies that AIBugHunter could possibly enhance developers' productivity in combating cybersecurity issues during the software development lifecycle. Last but not least, all of the participants rated our AIBugHunter as satisfied or highly satisfied due to reasons as follows

- P1: *It is seamlessly integrated into my development environment.*
- P3: *It exceeds my expectations for automated tools.*
- P4: *Detect the vulnerability down to line-level and provide CWE information.*
- P5: *Identify the vulnerability fast.*

5.3 The implications of AIBugHunter to researchers and practitioners

In this section, we discuss the broader implications of our AIBugHunter tool for researchers and practitioners. For practitioners, our AIBugHunter tool can help security practitioners locate vulnerabilities, identify vulnerability types, estimate vulnerability severity, and suggest vulnerability repairs. These AI-powered security intelligence features can produce significant benefits to practitioners. This includes potentially increasing developers' productivity, increasing the security of their software systems, and reducing overall software development costs. For researchers, our AIBugHunter tool is among the first proof-of-concept AI-powered security intelligence tool with numerous features combined into one tool. Many static analysis tools can only perform vulnerability detection, not repairs. Instead, we present how such important features could be integrated into a VS Code Extension. The results of

	Demographics			Test	W/O AIBugHunter				With AIBugHunter				
	Years in SE	Years in SS	Degree	CWE	Task 1	Task 2	Task 3	Task 4	Task 1	Task 2	Task 3	Task 4	Satisfaction
P1	6	0	Bachelor	Don't know	8	9	10	12	1	2	3	3	5
P2	10	1	Master	Knows	5	7	9	10	1	1	2	2	5
P3	10	3	Bachelor	Knows	5	5	8	11	1	1	2	2	5
P4	4	0.5	Bachelor	Knows	N/A	14	14	N/A	1	1	1	3	4
P5	10	3	PhD	Knows	5	5	7	10	3	1	3	4	5
P6	5	2	Bachelor	Knows	12	12	13	15	1	1	1	4	4
SE: Software Engineering				Correct Answer				Incorrect Answer					
SS: Software Security				CWE: Common Weakness Enumeration									

Fig. 13 The experimental results of our user study with six participants. Wherein the first task was to locate the vulnerability, the second task was to explain the vulnerability type, the third task was to estimate the vulnerability severity, and the fourth task was to suggest repairs. The time was measured in minutes and the satisfaction ranges from 1 (highly dissatisfied) to 5 (highly satisfied)

our user study also highlight the usability of our tool and its substantial potential benefits for the software engineering community.

6 Threats to Validity

6.1 Construct Validity

Threats to the construct validity relate to the potential bias of our survey study and user study. In our survey study and user study, we recruited 21 and 6 participants respectively from different professions such as software engineers and security analysts. However, the results of our two studies could still be biased towards our participants and the results do not necessarily generalize to other audiences. To mitigate this threat for our survey study, we spread our survey on social platforms such as Facebook and LinkedIn to ensure diverse participant demographics. To mitigate this threat for our user study, we recruited software practitioners with different backgrounds and professional experiences for our user study.

The goal of our survey study and user study is to investigate the usefulness of the tool. Thus, we only focus on correct predictions when designing our survey study and user study. However, our AIBugHunter could also return incorrect predictions. Thus, an extended user study is also required to fully evaluate the impact of our AIBugHunter by including both correct and incorrect predictions. Since this research question requires a rigorous user study and a different methodology than we use in this article, we plan to investigate this in future work.

Furthermore, the maturity of AIBugHunter is still at the early stage of development and is not yet ready for commercialization. Our user study experiment was conducted as a preliminary analysis. Thus, the findings are only limited to our studied group, and may not be generalized to other participants, users, software systems, and organizations. Therefore, an extensive evaluation of AIBugHunter is still needed.

6.2 Internal Validity

Threats to the internal validity relate to our choice of hyperparameter settings (i.e., optimizer, scheduler, learning rate, etc.) of our models to classify vulnerability types and estimate vulnerability severity. Finding a set of optimal hyperparameter settings of the CodeBERT model is extremely expensive due to a large number of trainable parameters in CodeBERT and the large search space of the Transformer architecture. Thus, we leverage the default setting of CodeBERT as reported by Feng et al. (2020). Hence, our results serve as a lower bound for our approach, which can be further improved through hyperparameter optimization Tantithamthavorn et al. (2016, 2019). To mitigate this threat, we report the hyperparameter settings in the replication package to support future replication studies.

6.3 External Validity

Threats to the external validity relate to the generalizability and applicability of our AIBugHunter. The models used in AIBugHunter were trained using Big-Vul Fan et al. (2020) and CVEFixes Bhandari et al. (2021) datasets consisting of C/C++ source code. Thus, our models do not necessarily generalize to other data and programming languages. However, the AIBugHunter tool could be used with other programming languages as it is designed

to adopt any deep learning models. Nevertheless, future work could explore the effectiveness of the AIBugHunter tool in other programming languages when other models are used.

7 Related Work

We discuss key previous studies of ML-based vulnerability prediction and multi-task learning for software vulnerability prediction. We compare our approach with previous methods and illustrate the difference.

7.1 ML-Based Vulnerability Type Classification

Multiple ML-based approaches have been proposed to automate the CWE-ID classification task (Aota et al., 2020; Na et al., 2016; Shuai et al., 2013). Shuai et al. (2013) constructed a Huffman Tree SVM , Na et al. (2016) used a Naive Bayes model , and Aota et al. (2020) leveraged a Random Forest model to automate the CWE-ID classification task. All of these approaches rely on the Bag of Words technique, while such a method can embed textual input features into numeric vector space, such embedding based on word counting can not capture enough semantic information of input.

Instead of using CVE entries as input, Wang et al. (2020) leveraged ML-based models to classify CWE-IDs for vulnerability security patches based on the features extracted from security patches. However, defining such hand-crafted features is time-consuming and may require much effort.

Recently, researchers have proposed DL-based models that learn the input representation through neural networks to better capture the semantic features of the input. Aghaei et al. (2020) proposed ThreatZoom, a Hierarchical Neural Network that considers the hierarchical nature of CWE-ID. Wang et al. (2021) leveraged the BERT architecture to learn textual features through the self-attention mechanism.

Previous studies focus on mapping either CVE entries (i.e., vulnerability description) or security patches into CWE-ID, however, such input features are not available during the software development stage, thus they are not compatible with our AIBugHunter, where it requires an ML model to predict based on the source code written by developers. In contrast, our approach only takes vulnerable source code without any description as input and predicts the corresponding CWE-ID. Therefore, it can support our AIBugHunter to generate vulnerability predictions based on the code written by developers.

7.2 Multi-Task Learning for Software Vulnerability Prediction

Spanos and Angelis (2018) used three ML ensemble classifiers to predict CVSS characteristics based on vulnerability description. Le et al. (2021) proposed DeepCVA which uses multiple GRUs and a shared embedding layer as a multi-task learning framework for commit-level vulnerability assessment. Gong et al. (2019) leveraged a Bi-LSTM as a shared feature extractor with multiple classifiers to predict different Common Vulnerability Scoring System (CVSS) characteristics based on vulnerability description. Babalau et al. (2021) used a shared BERT architecture with two prediction heads to learn a multi-task model which supports CVSS severity score classification and regression.

Some of these studies leveraged a shared architecture (Babalau et al., 2021; Gong et al., 2019; Takerngsaksiri et al., 2022) that can learn from labels of different tasks that are cor-

related, hence may help improve the model performance. Nevertheless, all of these studies relied on the weighted summation of loss functions during gradient descent, i.e., (1) averaging the loss of each task (Le et al., 2021), (2) tuning loss weights of each task (Babalau et al., 2021), (3) summarizing loss of each task (Gong et al., 2019). Such a weighted summation approach may not find the optimal solution when updating the shared model, for instance, the updated parameters are better for one task but not the other as discussed in Section 3.1.3. In contrast, our approach finds an optimal collection of parameters that benefits all tasks simultaneously during gradient descent that can optimize a collection of possibly conflicting objectives. To the best of our knowledge, this paper is among the first to leverage multi-objective optimization to learn a DL model for the software vulnerability classification task.

7.3 Explainable AI for Cybersecurity

Explainability is now becoming a critical concern in software engineering. Many researchers often employed AI/ML techniques for defect prediction (Pornprasit and Tantithamthavorn, 2021, 2022; Pornprasit et al., 2021; Rajapaksha et al., 2021; Wattanakriengkrai et al., 2020), malware detection (Liu et al., 2022, 2023), and effort estimation (Fu and Tantithamthavorn, 2022a). Yet, little is focused on explaining the vulnerability predictions, which is the focus of this paper. While these AI/ML techniques can greatly improve developers' productivity, software quality, and end-user experience, practitioners still do not understand why such AI/ML models made those predictions (Jiarpakdee et al., 2020, 2021; Pornprasit et al., 2021; Rajapaksha et al., 2021; Tantithamthavorn et al., 2021; Tantithamthavorn and Jiarpakdee, 2021b). In particular, the survey study by Jiarpakdee et al. (2021) found that explaining the predictions is as equally important and useful as improving the accuracy of defect prediction. However, their literature review found that 91% (81/96) of the defect prediction studies only focus on improving the predictive accuracy, without considering explaining the predictions, while only 4% of these 96 studies focus on explaining the predictions.

Although Explainable AI is still a very under-researched topic within the software engineering community (Cito et al., 2023; Tantithamthavorn and Jiarpakdee, 2021a; Tantithamthavorn et al., 2021, 2023), very few existing XAI studies have shown some successful usages e.g., in defect prediction. In one example, Wattanakriengkrai et al. (2020) and Pornprasit and Tantithamthavorn (2021) employed model-agnostic techniques (e.g., LIME) for line-level defect prediction (e.g., predicting which lines will be defective in the future), helping developers to localize defective lines in a cost-effective manner. For example, Jiarpakdee et al. (2020) and Khanan et al. (2020) employed model-agnostic techniques (e.g., LIME) for explaining defect prediction models, helping developers better understand why a file is predicted as defective. Rajapaksha et al. (2021) and Pornprasit et al. (2021) proposed local rule-based model-agnostic techniques to generate actionable guidance to help managers chart the most effective quality improvement plans.

In contrast to the prior studies, but in the same vein of research in explainable AI for software engineering, this paper aims to go beyond vulnerability prediction but provides explanations on the types, the severity, and the suggested repairs.

8 Conclusions

In this article, we propose AIBugHunter, an integration of our proposed software vulnerability classification (multi-objective optimization approach) and estimation (a transformer-based

approach) approaches and our previous works. Our AIBugHunter is an ML-based vulnerability prediction tool to (1) localize vulnerabilities, (2) classify vulnerability types, (3) estimate vulnerability severity, and (4) suggest repairs. To the best of our knowledge, this article is among the first to deploy an ML-based vulnerability prediction tool for C/C++ to the VS Code IDE. Our AIBugHunter realizes real-time vulnerability prediction during software development, which helps integrate security approaches into the software development life cycle. Our empirical survey study with 21 software practitioners confirms that our AIBugHunter is perceived as useful; and our user study indicates that our AIBugHunter could help reduce developers' time spent on security analysis, which could enhance developers' productivity in combating security issues during software development.

Acknowledgements Chakkrit Tantithamthavorn was partly supported by the Australian Research Council's Discovery Early Career Researcher Award (DECRA) (DE200100941). John Grundy is supported by ARC Laureate Fellowship (FL190100035).

Funding Open Access funding enabled and organized by CAUL and its Member Institutions.

Data Availibility The data, model training and evaluation scripts that support the findings of this study are available at: (https://github.com/awsm-research/AIBugHunter). Our proposed AIBugHunter is available at: (https://marketplace.visualstudio.com/items?itemName=AIBugHunter.aibughunter).

Declarations

Conflict of Interest Statement The authors of this article declared that they have no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

Aghaei E, Shadid W, Al-Shaer E (2020) Threatzoom: Hierarchical neural network for cves to cwes classification. In: International Conference on Security and Privacy in Communication Systems, Springer, pp 23–41

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

Babalau I, Corlatescu D, Grigorescu O, Sandescu C, Dascalu M (2021) Severity prediction of software vulnerabilities based on their text description. In: 2021 23rd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), IEEE, pp 171–177

Bhandari G, Naseer A, Moonen L (2021) Cvefixes: automated collection of vulnerabilities and their fixes from open-source software. In: Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering, pp 30–39

Checkmarx (2006) Checkmarx. https://checkmarx.com/

Chen Z, Badrinarayanan V, Lee CY, Rabinovich A (2018) Gradnorm: Gradient normalization for adaptive loss balancing in deep multitask networks. In: International conference on machine learning, PMLR, pp 794–803

Chen Z, Kommrusch S, Monperrus M (2021) Neural transfer learning for repairing security vulnerabilities in c code. IEEE Transactions on Software Engineering

Cito J, Chandra S, Tantithamthavorn C, Hemmati H (2023) Expert perspectives on explainability. IEEE Software 40(3):84–88. https://doi.org/10.1109/MS.2023.3255663
Corporation TM (2022) Att&ck. https://attack.mitre.org/
Croft R, Newlands D, Chen Z, Babar MA (2021) An empirical study of rule-based and learning-based approaches for static application security testing. In: In the Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), pp 1–12
CVSS (2003) Common vulnerability scoring system (cvss). https://nvd.nist.gov/vuln-metrics/cvss
CWE (2006) Common weakness enumeration (cwe). https://cwe.mitre.org/index.html
CWE (2009) Cwe-787. https://cwe.mitre.org/data/definitions/787.html
CWE (2021a) 2021 cwe top 25 most dangerous software weaknesses. https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html
CWE (2021b) Cwe abstract type - class weakness. https://cwe.mitre.org/documents/glossary/#Class%20Weakness
CWE (2021c) Cwe abstract type - variant weakness. https://cwe.mitre.org/documents/glossary/#Variant%20Weakness
Das SS, Serra E, Halappanavar M, Pothen A, Al-Shaer E (2021) V2w-bert: A framework for effective hierarchical multiclass classification of software vulnerabilities. In: 2021 IEEE 8th International Conference on Data Science and Advanced Analytics (DSAA), IEEE, pp 1–12
Devlin J, Chang MW, Lee K, Toutanova K (2019) Bert: Pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), pp 4171–4186
Fan J, Li Y, Wang S, Nguyen TN (2020) A c/c++ code vulnerability dataset with code changes and cve summaries. In: Proceedings of the 17th International Conference on Mining Software Repositories, pp 508–512
Feng Z, Guo D, Tang D, Duan N, Feng X, Gong M, Shou L, Qin B, Liu T, Jiang D et al (2020) Codebert: A pre-trained model for programming and natural languages. Findings of the Association for Computational Linguistics: EMNLP 2020:1536–1547
Fu M, Tantithamthavorn C (2022a) GPT2SP: A Transformer-Based Agile Story Point Estimation Approach. IEEE Transactions on Software Engineering
Fu M, Tantithamthavorn C (2022b) Linevul: A transformer-based line-level vulnerability prediction. In: 2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR), IEEE
Fu M, Tantithamthavorn C, Le T, Nguyen V, Dinh P (2022) Vulrepair: A t5-based automated software vulnerability repair. In: In the Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)
Gong X, Xing Z, Li X, Feng Z, Han Z (2019) Joint prediction of multiple vulnerability characteristics through multi-task learning. In: 2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS), IEEE, pp 31–40
Hin D, Kan A, Chen H, Babar MA (2022) Linevd: Statement-level vulnerability detection using graph neural networks. In: 2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR), IEEE
Jiarpakdee J, Tantithamthavorn C, Dam HK, Grundy J (2020) An Empirical Study of Model-Agnostic Techniques for Defect Prediction Models. IEEE Transactions on Software Engineering (TSE) p To Appear
Jiarpakdee J, Tantithamthavorn C, Grundy J (2021) Practitioners' Perceptions of the Goals and Visual Explanations of Defect Prediction Models. In: Proceedings of the International Conference on Mining Software Repositories (MSR), p To Appear
Johnson A, Dempsey K, Ross R, Gupta S, Bailey D et al (2011) Guide for security-focused configuration management of information systems. NIST special publication 800(128):16–16
Kendall A, Gal Y, Cipolla R (2018) Multi-task learning using uncertainty to weigh losses for scene geometry and semantics. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp 7482–7491
Khanan C, Luewichana W, Pruktharathikoon K, Jiarpakdee J, Tantithamthavorn C, Choetkiertikul M, Ragkhitwetsagul C, Sunetnanta T (2020) Jitbot: An explainable just-in-time defect prediction bot. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, pp 1336–1339
Kitchenham BA, Pfleeger SL (2008) Personal opinion surveys. In: Guide to Advanced Empirical Software Engineering, Springer, pp 63–92
Le THM, Hin D, Croft R, Babar MA (2021) Deepcva: Automated commit-level vulnerability assessment with deep multi-task learning. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, pp 717–729

Li Y, Wang S, Nguyen TN (2021) Vulnerability detection with fine-grained interpretations. In: 29th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2021, Association for Computing Machinery, Inc, pp 292–303

Liu Y, Tantithamthavorn C, Li L, Liu Y (2022) Explainable AI for android malware detection: Towards understanding why the models perform so well? In: IEEE 33rd International Symposium on Software Reliability Engineering, ISSRE 2022, Charlotte, NC, USA, October 31 - Nov. 3, 2022, IEEE, pp 169–180. https://doi.org/10.1109/ISSRE55969.2022.00026

Liu Y, Tantithamthavorn C, Li L, Liu Y (2023) Deep learning for android malware defenses: A systematic literature review. ACM Comput Surv 55(8):153:1–153:36. https://doi.org/10.1145/3544968

Loshchilov I, Hutter F (2018) Decoupled weight decay regularization. In: International Conference on Learning Representations

Marjamäki D (2007) Cppcheck. https://cppcheck.sourceforge.io/

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

NVD (2019) Cvss version 3.1. https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator

Pornprasit C, Tantithamthavorn C (2021) JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction. In: Proceedings of the International Conference on Mining Software Repositories (MSR)

Pornprasit C, Tantithamthavorn C (2022) DeepLineDP: Towards a Deep Learning Approach for Line-Level Defect Prediction. IEEE Transactions on Software Engineering

Pornprasit C, Tantithamthavorn C, Jiarpakdee J, Fu M, Thongtanunam P (2021) Pyexplainer: Explaining the predictions of just-in-time defect models. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, pp 407–418

Rajapaksha D, Tantithamthavorn C, Bergmeir C, Buntine W, Jiarpakdee J, Grundy J (2021) Sqaplanner: Generating data-informed software quality improvement plans. IEEE Transactions on Software Engineering

Renaud A (2018) A vulnerable c function. https://github.com/AndreRenaud/PDFGen/commit/8f9b3202f67feb386c9974520d%9bcc4531350fff

Sener O, Koltun V (2018) Multi-task learning as multi-objective optimization. Advances in neural information processing systems 31

Shuai B, Li H, Li M, Zhang Q, Tang C (2013) Automatic classification for vulnerability based on machine learning. In: 2013 IEEE International Conference on Information and Automation (ICIA), IEEE, pp 312–318

Spanos G, Angelis L (2018) A multi-target approach to estimate software vulnerability characteristics and severity scores. Journal of Systems and Software 146:152–166

Takerngsaksiri W, Tantithamthavorn C, Li YF (2022) Syntax-aware on-the-fly code completion. arXiv preprint arXiv:2211.04673

Tantithamthavorn C, Jiarpakdee J (2021a) Explainable AI for Software Engineering. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, pp 1–2

Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K (2016) Automated Parameter Optimization of Classification Techniques for Defect Prediction Models. In: ICSE, pp 321–332

Tantithamthavorn C, McIntosh S, Hassan AE, Matsumoto K (2019) The Impact of Automated Parameter Optimization on Defect Prediction Models. TSE

Tantithamthavorn C, Jiarpakdee J, Grundy J (2021) Actionable analytics: Stop telling me what it is; please tell me what to do. IEEE Software 38(4):115–120

Tantithamthavorn C, Cito J, Hemmati H, Chandra S (2023) Explainable ai for se: Challenges and future directions. IEEE Software 40(3):29–33. https://doi.org/10.1109/MS.2023.3246686

Tantithamthavorn CK, Jiarpakdee J (2021b) Explainable ai for software engineering. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, pp 1–2

Thapa C, Jang SI, Ahmed ME, Camtepe S, Pieprzyk J, Nepal S (2022) Transformer-based language models for software vulnerability detection: Performance, model's security and platforms. arXiv preprint. arXiv:2204.03214

Touvron H, Cord M, Douze M, Massa F, Sablayrolles A, Jégou H (2021) Training data-efficient image transformers & distillation through attention. In: International Conference on Machine Learning, PMLR, pp 10,347–10,357

Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, Kaiser Ł, Polosukhin I (2017) Attention is all you need. In: Advances in neural information processing systems (NeurIPS), pp 5998–6008

Wang T, Qin S, Chow KP (2021) Towards vulnerability types classification using pure self-attention: A common weakness enumeration based approach. In: 2021 IEEE 24th International Conference on Computational Science and Engineering (CSE), IEEE, pp 146–153

Wang X, Wang S, Sun K, Batcheller A, Jajodia S (2020) A machine learning approach to classify security patches into vulnerability types. In: 2020 IEEE Conference on Communications and Network Security (CNS), IEEE, pp 1–9

Wattanakriengkrai S, Thongtanunam P, Tantithamthavorn C, Hata H, Matsumoto K (2020) Predicting defective lines using a model-agnostic technique. IEEE Transactions on Software Engineering (TSE)

WhiteSource (2019) What are the most secure programming languages? https://www.mend.io/most-secure-programming-languages/

Yuan X, Lin G, Tai Y, Zhang J (2022) Deep neural embedding for software vulnerability discovery: Comparison and optimization. Security and Communication Networks 2022

Zettler K (2022) The devsecop tools that secure devops workflows. https://www.redhat.com/en/topics/devops/what-is-devsecops

Zhou Y, Liu S, Siow J, Du X, Liu Y (2019) Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In: Proceedings of the 33rd International Conference on Neural Information Processing Systems, pp 10,197–10,207

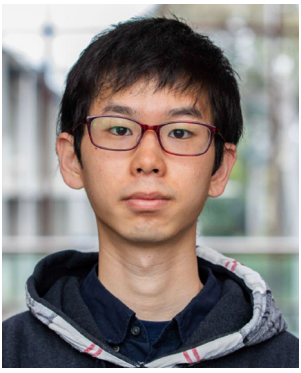
Zhu C, Du G, Wu T, Cui N, Chen L, Shi G (2022) Bert-based vulnerability type identification with effective program representation. In: Wireless Algorithms, Systems, and Applications: 17th International Conference, WASA 2022, Dalian, China, November 24–26, 2022, Proceedings, Part I, Springer, pp 271–282

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Michael Fu is a PhD student in the department of software systems and cybersecurity at Monash University, Australia. His research focuses on developing deep learning-based automated approaches to (1) locate vulnerability down to the statement level, (2) identify vulnerability types, and (3) suggest vulnerability repairs. More about him at https://michaelfu1998-create.github.io/.

Dr. Chakkrit (Kla) Tantithamthavorn is a Senior Lecturer and a Director of Engagement and Impact at the Faculty of Information Technology, Monash University, Australia. He is pioneering an emerging research area of Explainable AI for Software Engineering, inventing many AI-based technologies to improve developers' productivity and make software systems more reliable and more secure while being explainable to practitioners. He has made several major advances in explainable AI for software engineering and published the first online book on Explainable AI for Software Engineering (http://xai4se.github.io), attracting 13,000+ pageviews from 83 countries worldwide and receiving positive responses from the SE community. His publications, books, and tutorials have informed many other studies and educated the SE community on the importance of explainability and its applications to software engineering. More about him at http://chakkrit.com.

Dr. Trung Le is a Lecturer in the Department of Data Science and AI at Monash University, Australia. He specialises in topics such as optimal transport theory, machine learning, optimization, probabilistic inference, generative models, transfer learning, continual learning, adversarial/trustworthy machine learning, and cyber-security. Dr. Le has published over 100 papers in prestigious conferences and journals and secured over 1 million dollars in funding for research in areas such as generative models, adversarial/trustworthy machine learning, transfer learning, and cyber-security.



Yuki Kume is an undergraduate student pursuing Bachelor's degree in Computer Science Advanced (Honours) at Monash University, Australia. His interests are in software engineering and their application in enhancing security and privacy. His interest in cybersecurity also extends to the cryptography and network security domain, including post-quantum cryptography (PQC) and applied cryptography in networked systems.

Van Nguyen is a Research Fellow at Monash University in Australia, specifically in the Department of Software Systems & Cybersecurity. Additionally, he is an Affiliate at CSIRO's Data61, also located in Australia. In February 2021, he completed his PhD degree in Computer Science from Monash University. Van Nguyen was acknowledged as a global talent through Australia's global talent independent program (GTI) in 2021. His research focuses on advancing machine learning and deep learning techniques to address topical issues in Cybersecurity. His objective is to enhance the design, construction, and protection of security systems, ultimately mitigating security breaches. Van Nguyen also demonstrates interest in the fields of computer vision and natural language processing. His contributions to these areas have resulted in the publication of over 16 papers in esteemed conferences and journals that specialise in machine learning and software engineering.

Dinh Phung is a Professor in the Department of Data Science and AI, Monash University, Australia. He received his B.Sc (first-class honour) and PhD degrees in computer science from Curtin University in 2001 and 2005 respectively. He is best known for his contributions in optimal transport theory for machine learning, Bayesian nonparametric models, deep generative models and trustworthy AI. He has won numerous research awards, published 250+ papers and attracted over 15 million fundings in these areas and application domains such as NLP. computer vision, cybersecurity, digital health, and AI-enabled autism research. He has been actively serving the academic community, including TPC members for 50+ top-ranked conferences in AI and machine learning, being associate editor for the Journal of Artificial Intelligence Research since 2021 and was the Finalist for the prestigious Australian Museum Eureka Prize for Excellence in Data Science in 2020.

John Grundy is Australian Laureate Fellow and Professor of Software Engineering at Monash University. His interests include automated software engineering, software engineering methods and tools, human aspects and their impact on software engineering, requirements engineering and software engineering education. He is Fellow of Automated Software Engineering and Fellow of Engineers Australia.

Authors and Affiliations

Michael Fu[1] · Chakkrit Tantithamthavorn[1] · Trung Le[1] · Yuki Kume[1] · Van Nguyen[1] · Dinh Phung[1] · John Grundy[1]

Michael Fu
yeh.fu@monash.edu

Trung Le
trunglm@monash.edu

Yuki Kume
yuki.kume@monash.edu

Van Nguyen
Van.Nguyen1@monash.edu

Dinh Phung
Dinh.Phung@monash.edu

John Grundy
John.Grundy@monash.edu

1 Monash University, Clayton, VIC, Australia