Pre-trained Model-based Actionable Warning Identification: A Feasibility Study

XIUTING GE, Department of Computer Science, China University of Geosciences (Wuhan), China
CHUNRONG FANG*, State Key Laboratory for Novel Software Technology, Nanjing University, China
QUANJUN ZHANG, Nanjing University of Science and Technology, China
DAOYUAN WU, Lingnan University, China
BOWEN YU, State Key Laboratory for Novel Software Technology, Nanjing University, China
QIRUI ZHENG, State Key Laboratory for Novel Software Technology, Nanjing University, China
AN GUO, State Key Laboratory for Novel Software Technology, Nanjing University, China
SHANGWEI LIN, College of Computing and Data Science, Nanyang Technological University, Singapore
ZHIHONG ZHAO, State Key Laboratory for Novel Software Technology, Nanjing University, China
YANG LIU, College of Computing and Data Science, Nanyang Technological University, Singapore
ZHENYU CHEN, State Key Laboratory for Novel Software Technology, Nanjing University, China

Actionable Warning Identification (AWI) plays a pivotal role in improving the usability of Static Code Analyzers (SCAs). Currently, Machine Learning (ML)-based AWI approaches, which mainly learn an AWI classifier from labeled warnings, are notably common. However, these approaches still face the problem of restricted performance due to the direct reliance on a limited number of labeled warnings to develop a classifier. Very recently, Pre-Trained Models (PTMs), which have been trained through billions of text/code tokens and have demonstrated substantial successful applications in various code-related tasks, could potentially address the above problem. Nevertheless, the performance of PTMs on AWI has not been systematically investigated, leaving a gap in understanding their pros and cons. In this paper, we are the first to explore the feasibility of applying various PTMs for AWI. By conducting an extensive evaluation on 12K+ warnings involving four commonly used SCAs (i.e., SpotBugs, Infer, CppCheck, and CSA) and three typical programming languages (i.e., Java, C, and C++), we (1) investigate the overall PTM-based AWI performance compared to the state-of-the-art ML-based AWI approach, (2) analyze the impact of three primary aspects (i.e., data preprocessing, model training, and model prediction) in the typical PTM-based

*Chunrong Fang is the corresponding author.

Authors' Contact Information: Xiuting Ge, Department of Computer Science, China University of Geosciences (Wuhan), Wuhan, Hubei, China, gext@cug.edu.cn; Chunrong Fang, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China, fangchunrong@nju.edu.cn; Quanjun Zhang, quanjunzhang@njust.edu.cn, Nanjing University of Science and Technology, Nanjing, China; Daoyuan Wu, Lingnan University, Hong Kong SAR, China, daoyuanwu@ln.edu.hk; Bowen Yu, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China, 201250070@smail.nju.edu.cn; Qirui Zheng, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China, 201250229@smail.nju.edu.cn; An Guo, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China, guoan218@smail.nju.edu.cn; Shangwei Lin, College of Computing and Data Science, Nanyang Technological University, Singapore, shang-wei.lin@ntu.edu.sg; Zhihong Zhao, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China, zhaozhih@nju.edu.cn; Yang Liu, College of Computing and Data Science, Nanyang Technological University, Singapore, yangliu@ntu.edu.sg; Zhenyu Chen, State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China, zychen@nju.edu.cn.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2025 Copyright held by the owner/author(s).
ACM 1557-7392/2025/11-ART
https://doi.org/10.1145/3777369

Just Accepted

AWI workflow, and (3) identify the reasons for the current underperformance of PTMs on AWI, thereby obtaining a series of findings. Based on the above findings, we further provide several potential directions to enhance PTM-based AWI.

CCS Concepts: • **General and reference → Empirical studies**; • **Software and its engineering** → *Software testing and debugging*;

Additional Key Words and Phrases: Actionable warning identification, pre-trained model, static analysis, feasibility study

1 INTRODUCTION

Static Code Analyzers (SCAs) can automatically scan software codebases and reveal potential defects without executing the program [11]. Despite the benefits of SCAs in software defect detection [23, 40], SCAs are still underused in practice due to reporting an overwhelming number of unactionable warnings, especially false positives [17, 61, 63]. Manually identifying warnings into actionable and unactionable ones is time-consuming and error-prone [29]. Thus, the tremendous unactionable warnings and the tedious manual inspection costs pose significant barriers to the usability of SCAs.

To alleviate the above problem, different approaches [2] have been proposed to optimize the precision of SCAs from the vendor's perspective, thereby reducing the number of false positives reported by SCAs. However, since the trade-off between precision and recall is non-trivial in the static analysis [50], it is inevitable for SCAs to report false positives. Despite retaining an initially high precision, SCAs could undergo a decline in defect detection performance as the nature of defects changes over time [30]. The continuous maintenance and updates to make SCAs overcome the concept drift could be an expensive endeavor [6]. As such, an alternative approach [44], i.e., *Actionable Warning Identification (AWI)*, has been proposed to postprocess warnings reported by SCAs from the user's perspective, thereby identifying actionable warnings from all reported warnings. Unlike the existing precision optimization approaches [2] that refine the complex static analysis techniques before the usage of SCAs, AWI focuses on leveraging various postprocessing techniques (e.g., clustering, ranking, pruning, or simplifying manual inspection) to classify or prioritize warnings after the usage of SCAs. It indicates that AWI is independent of specific SCAs with different static analysis techniques. More importantly, AWI can be equipped with various postprocessing techniques to augment SCAs, where these postprocessing techniques complement the capabilities of SCAs to report more precise results.

In the existing AWI approaches, Machine Learning (ML)-based AWI approaches are notably popular due to ML's strong ability to learn subtle and previously unseen patterns from historical data. The general process of ML-based AWI approaches is to utilize ML models to train the AWI classifier from labeled warnings and use this classifier to identify actionable warnings from unlabeled ones [18]. However, the performance of these approaches is still limited because the AWI classifier is generally established on a small number of labeled warnings [30, 33, 34, 38, 54, 64]. It indicates that the true power provided by ML techniques has not been fully unleashed on AWI.

The rapid development of ML techniques has spurred the emergence of Pre-Trained Models (PTMs). Different from the supervised learning of ML-based AWI approaches on a limited number of labeled warnings, PTMs are trained in a self-supervised fashion based on the tremendous unlabeled corpora and can be used for downstream tasks by fine-tuning labeled samples [24]. Currently, PTMs have exhibited remarkable performance in a variety of code-related tasks [47]. The unique characteristics and recent breakthroughs of PTMs inspire us to apply PTMs to alleviate the problem of existing ML-based AWI approaches [30]. However, the literature does not systematically investigate the actual power of modern PTMs on AWI, thereby failing to understand the pros and cons of PTM-based AWI.

To bridge the above gap, we perform the first extensive study to explore the feasibility of PTMs on AWI. Specifically, we first investigate the performance of PTM-based AWI by comparing the State-Of-The-Art (SOTA) ML-based AWI approach. Second, based on the typical PTM-based AWI workflow, we analyze the impact of

Just Accepted

the data preprocessing ways, model training components, and model prediction scenarios on the PTM-based AWI performance. Third, we identify the reasons for the underperformance of current PTMs on AWI. Based on more than 12K+ warnings involving four commonly used SCAs (i.e., SpotBugs [52], Infer [26], CppCheck [12], and CSA [13]) and three typical programming languages (i.e., Java, C, and C++), we conduct experiments of nine representative PTMs with encoder-decoder, encode-only, and decoder-only architectures. After that, we obtain the following findings, including (1) nine PTMs on AWI significantly outperform the SOTA ML-based AWI approach in terms of recall and F1, while maintaining very similar precision on Java warnings. In contrast, the average precision, recall, and F1 of the PTM-based AWI approach is consistently better than the SOTA ML-based AWI approach on C/C++ warnings; (2) in the data preprocessing, the warning context extraction consistently enhances the PTM-based AWI approach on Java and C/C++ warnings. However, the warning context abstraction improves the precision but hinders the recall of the PTM-based AWI approach on Java warnings, while increasing the precision and recall of the PTM-based AWI approach on C/C++ warnings; (3) in the model training, the code-related pre-training and fine-tuning components are beneficial for the PTM-based AWI approach; (4) in the model prediction, PTMs can generally perform better in the within-project AWI scenario than in the cross-project AWI scenario; and (5) PTMs struggle on AWI when involving non-distinct/missing warning contexts and the imbalanced warning dataset. In addition, we perform a rigorously qualitative analysis to discuss the practical applications of our findings on different SCAs and projects with different programming languages. To this end, we highlight three potential directions (e.g., the warning context refinement) for boosting the future PTM-based AWI approach.

In summary, we make the following contributions.

- **New perspective.** We incorporate recent advances of PTMs into the AWI community. Besides, we conduct a systematic evaluation to unveil the substantial improvement of PTMs on AWI. We believe that our study yields the best of current ML and static analysis fields, where PTMs augment the usability of existing SCAs.
- **Elaborate study.** We are the first to perform an extensive study to explore the feasibility of PTMs on AWI, including a detailed comparison between SOTA ML-based and PTM-based AWI approaches, a thorough investigation about the impact of primary aspects (i.e., data preprocessing, model training, and model prediction) in the typical PTM-based AWI workflow, and an in-depth analysis about the challenges of PTMs on AWI.
- **Extensive experiments.** We conduct extensive experiments on 12K+ warnings with four commonly used SCAs and three typical programming languages as well as nine representative PTMs with three architectures. Based on the experimental results, we obtain five findings.
- **Potential directions**. We highlight several potential directions (e.g., the warning context refinement) in future PTM-based AWI research based on our findings.
- **Available artifacts.** We release the studied warning dataset and the experimental scripts in a public repository [48] for replication and future research.

The remainder of our study is organized as follows. Section 2 describes the background and related work. Section 3 introduces the experiment design. Section 4 shows the experimental result analysis. Section 5 outlines the practical application of our findings, future potential research directions, and threats to validity. Section 6 concludes the paper.

2 BACKGROUND AND RELATED WORK

2.1 Static Analysis Warnings

SCAs can detect various defects in the codebase, e.g., security issues and code smells. The existing AWI studies [41, 57, 63, 65] denote such defects as static analysis warnings, alerts, alarms, or violations. In our study, such

Just Accepted

defects are simply denoted as warnings. To help developers quickly locate and understand defects, each warning is generally equipped with category, priority, message, and location. Of these, the location often consists of the class and method information containing a warning as well as the warning line numbers. As an example, Fig. 2 (1) shows a simplified warning reported by SpotBugs [52].

Based on whether warnings are acted on and fixed by developers, warnings can be divided into actionable and unactionable ones [41, 57, 63, 65]. An actionable warning, including a true defect or a warning concerned by SCA users, is acted on and fixed by developers via the warning-related source code changes. Conversely, an unactionable warning might be a false positive warning due to the inherent problem (i.e., over-approximation) of SCAs [50], an unimportant warning for SCA users, or just an incorrectly reported warning due to bugs of SCAs [56, 66]. Thus, an unactionable warning is not acted on or fixed by developers.

Formally, given a set of commits $C = \{c_1, ..., c_i, ..., c_n\}$ in a project (c_n is the latest commit), a SCA is used to scan the source code of c_i and a set of warnings $W_i = \{w_{i1}, ..., w_{ij}, ..., w_{im}\}$ (m is the number of warnings in c_i) is obtained. If w_{ij} disappears via the warning-related source code change in any commit from c_{i+1} to c_n, w_{ij} is denoted as **an actionable warning**. If w_{ij} persists from c_{i+1} to c_n, w_{ij} is denoted as **an unactionable warning**.

2.2 ML-based AWI Approaches

In general, ML-based AWI approaches first extract features from the warning report or the warning-related source code, learn an AWI classifier from these extracted features of labeled warnings via traditional ML models or Deep Learning (DL) models, and utilize this classifier to identify unlabeled warnings into actionable and unactionable ones. Based on the extracted warning features, ML-based AWI approaches can be mainly divided into hand-engineered, token-based, and text-based approaches [19]. Based on the adopted ML models, ML-based AWI approaches can be roughly divided into traditional ML-based and DL-based AWI ones [18]. However, an effective AWI classifier relies extremely on labeled warnings, and there is a limited number of labeled warnings. As such, the performance of existing ML-based AWI approaches is still restricted.

2.3 Pre-trained Models

PTMs pre-train transformer-based models on large-scale and unlabeled corpora to distill the generic representation and then employ such a generic representation to handle downstream tasks by fine-tuning a limited number of labeled samples [24]. According to different PTM architectures, PTMs can be classified into encoder-only, decoder-only, and encoder-decoder models [62]. The encoder-only model, focusing solely on transforming the input data into the latent representation, is good at understanding tasks like text/code classification. The decoder-only model, aiming to decode output sequences from a given representation of the input data, is good at generating tasks like text/code completion. The encoder-decoder model combines both an encoder and a decoder into a single architecture, which is capable of handling sequence-to-sequence tasks.

2.4 Related Work

There are two studies [39, 60], which aim to leverage the potential of LLMs (e.g., ChatGPT) for AWI by designing the proper prompts. However, our study is different from the two studies in three aspects. First, without modifying the model parameters, the two studies focus on mining the model ability for AWI via prompt engineering. In contrast, our study explores the feasibility of PTMs in AWI by incorporating pre-training and fine-tuning of PTMs. In other words, by using pre-training and fine-tuning to adjust the model parameters, our study focuses on enhancing the model ability for AWI. Second, our study scrutinizes the impact of different data preprocessing ways and model prediction scenarios on the PTM-based AWI performance, which can provide researchers and practitioners with a holistic understanding of applying PTMs for AWI. Third, our study analyzes the reasons

Just Accepted

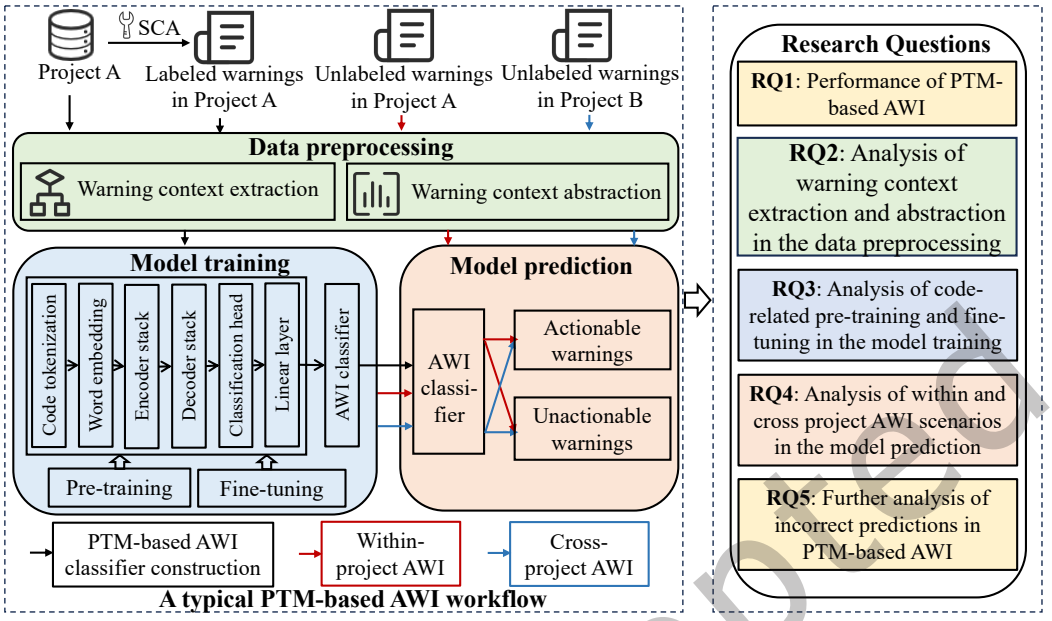


Fig. 1. Overview of our study.

why PTMs incorrectly identify some warning cases, which can help researchers and practitioners provide the potential research directions (e.g., the PTM-based AWI model improvement) for the future AWI field.

Another work is called DeepInferEnhance [30], which relies on the CodeBERTa architecture to train a new PTM on unlabeled source code and fine-tune this PTM on a few labeled warnings for AWI. Yet, there are three aspects distinguishing DeepInferEnhance from our study. First, DeepInferEnhance discards the original knowledge of CodeBERTa and retraining a new PTM for AWI. In contrast, our study focuses on fine-tuning the off-the-shelf PTMs for AWI. Such an operation can use the generic knowledge of off-the-shelf PTMs and learn the unique warning knowledge from the fine-tuning component to enhance AWI. Besides, such an operation can greatly reduce the PTM-based AWI model construction time and fully explore the impact of PTM characteristics (i.e., the fine-tuning component) on AWI. Second, CodeBERTa is inspired by the success of CodeBERT [16], which is a typical and popular encode-only PTM. In addition to CodeBERT, our study considers the other eight PTMs, which encompass three typical PTM architectures (i.e., encoder-decoder, encode-only, and decode-only). Such an operation enables researchers and practitioners to thoroughly explore the impact of various PTM architectures on AWI. Third, our study relies on the typical PTM-based AWI workflow to systematically investigate the impact of primary aspects in the data preprocessing, model training, and model prediction stages, thereby helping researchers and practitioners conduct a thorough exploration of applying various PTMs on AWI.

3 STUDY OVERVIEW

3.1 Typical PTM-based AWI Workflow

In the PTM-based AWI field, given a targeted warning with associated context $X = \{x_1, ..., x_k\}$, x_k is the k_{th} code token in the warning context. Taking X as the input, PTM-based AWI relies on $Pr(X;\theta)$ to output a class label y. The weight θ is obtained from the transformer that makes up the encoder and decoder. y is a binary value,

Just Accepted

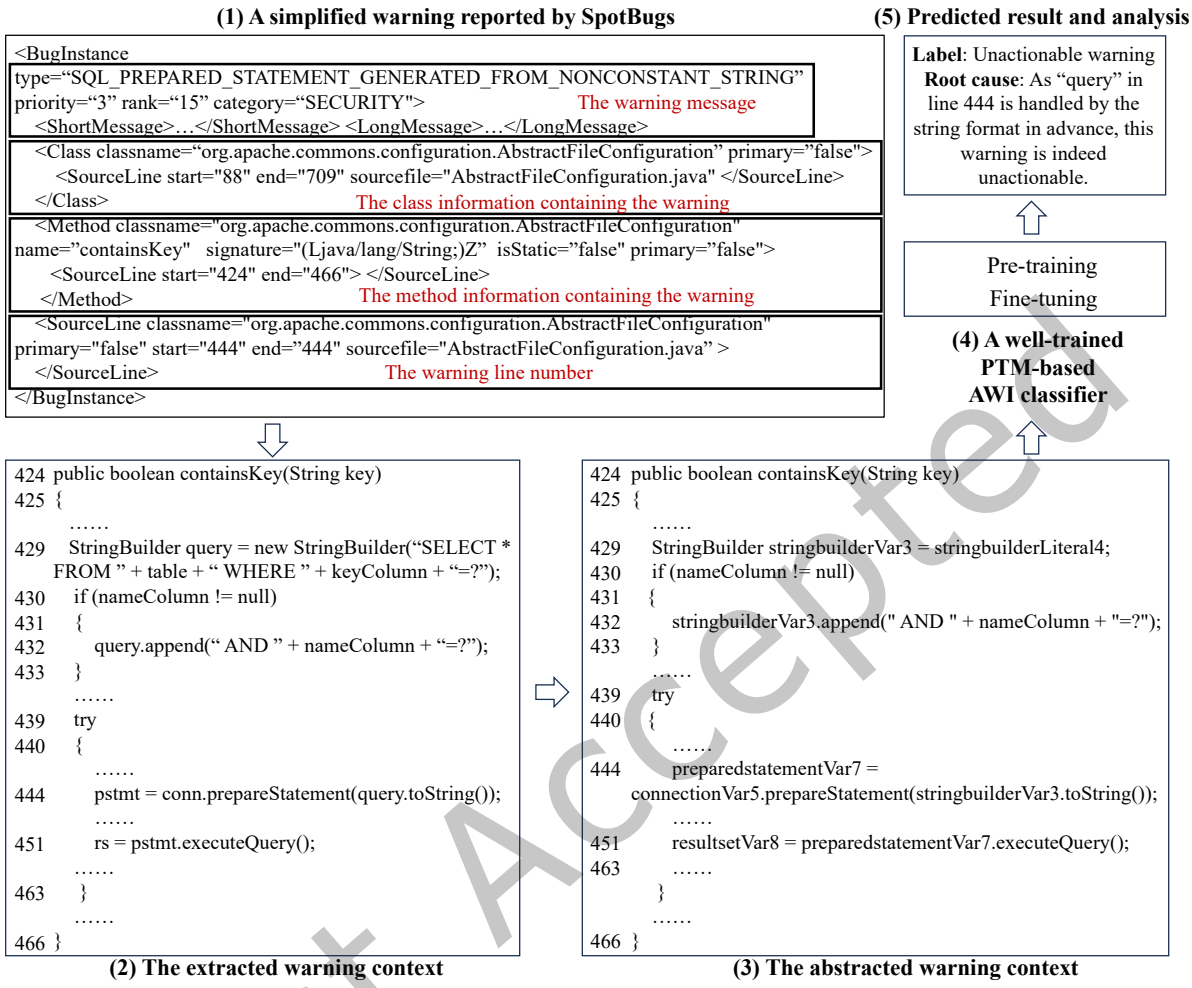


Fig. 2. An example to illustrate our approach.

where *y = 0/1* denotes an unactionable/actionable warning respectively. Fig. 1 shows a typical PTM-based AWI workflow with data preprocessing, model training, and model prediction stages.

Data preprocessing. Given a warning reported by a SCA as input, the processed context of this warning is returned. According to the existing ML-based AWI studies [30, 33, 34, 38, 54, 64], the data preprocessing stage mainly involves the warning context extraction and abstraction. The warning context extraction acquires the warning-inducing source code. The warning context abstraction is to rename some special words (e.g., identifiers and literals) in the warning context to a pool of predefined code tokens.

The warning context extraction process is shown as follows. Given a warning reported by a SCA, the warning location can be obtained, including the class/method information containing this warning as well as the warning line numbers. Naturally, based on the warning location, there are three warning context ways, i.e., the source code from the class containing a warning *ClassContext*, the method containing a warning *MethodContext*, and the

Just Accepted

warning line numbers *LineContext*. Compared to *MethodContext*, *ClassContext* brings too much information that is irrelevant to the warning, and *LineContext* is too coarse-grained due to missing the detailed context information that generates a warning. To this end, our approach considers *MethodContext* as the warning context. It is noted that there are other warning context extraction techniques, e.g., the program slicing [19, 59]. Instead of these techniques, our approach still considers *MethodContext* as the warning context. The main reasons are shown as follows. First, a method serves as a natural semantic unit in the source code, which can typically encapsulate localized behavior. Additionally, many warnings are closely tied to the logic of an individual method in practice [20]. Second, due to being well-bounded and readily extractable, *MethodContext* strikes a balance between the context richness and the extraction efficiency. Third, the existing studies [10, 31, 33] showcase that defects are generally revealed by analyzing source code in the method scope. It implies that *MethodContext* is a reasonable starting point for AWI. Given a warning in Fig. 2(1), the extracted warning context is shown in Fig. 2(2).

The warning context abstraction process is shown as follows. Our approach first tokenizes the *MethodContext* via the lexical analysis. Then, by constructing an abstract syntax tree, our approach identifies identifier and literal types from *MethodContext*. Finally, our approach replaces each identifier/literal in the stream of tokens with a distinct number, which denotes the type and role of this identifier/literal in *MethodContext*. For example, the source code "int a = 1;" is abstracted into "int intVar1 = intLiteral1;". Based on the above warning context abstraction, Fig. 2(3) describes the abstracted warning context.

Model training. A PTM-based AWI classifier is first established on the top of the transformer [55] and the mapping from warning context to warning label is optimized by updating the parameters of the designed classifier. Similar to the vanilla transformer architecture [55], PTMs often initiate with an encoder stack and a decoder stack, and culminate with a linear layer equipped with softmax activation. In AWI, taking the processed warning context as input, PTM first splits the input into words via code tokenization. Second, PTM performs the word embedding to yield the representation vectors for the tokenized warning context. Third, PTM feeds these vectors into the encoder and decoder stacks to output a last hidden state. Fourth, PTM adds a classification head for this last hidden state to obtain the logits. Fifth, PTM employs a linear layer with softmax activation for the obtained logits to acquire the probability distribution of binary warning labels.

Generally, PTM involves two essential components, i.e., pre-training and fine-tuning [24]. In AWI, the pre-training component is related to whether a PTM-based AWI classifier is obtained by pre-training over unlabeled and large-scale codebases. In contrast, the fine-tuning component is related to whether a PTM-based AWI classifier is obtained by fine-tuning on a limited number of labeled warnings. After the above model training, a well-trained PTM-based AWI classifier can be obtained in Fig. 2(4).

Model prediction. The well-trained PTM-based AWI classifier is used to classify unlabeled warnings into actionable and unactionable ones during the model prediction. Based on different project sources between labeled warnings in the model training and unlabeled warnings in the model prediction, there are within- and cross-project AWI scenarios. When labeled warnings used for the model training and unlabeled warnings used for the model prediction are from the same project, it is called the within-project AWI scenario. By contrast, when labeled warnings used for the model training and unlabeled warnings used for the model prediction are from different projects, it is called the cross-project AWI scenario.

After the model prediction, Fig. 2(5) shows that the given warning in Fig. 2(1) is correctly classified as an unactionable warning by the well-trained PTM-based AWI classifier. Correspondingly, Fig. 2(5) describes the root cause of such an unactionable warning. That is, as "query" in line 444 is handled by the string format in advance, this warning is ignored by the developer and is indeed an unactionable warning.

3.2 Research Questions

Inspired by the typical PTM-based AWI workflow, we investigate the following Research Questions (RQs).

Just Accepted

Table 1. Dataset information. #C. commits, #UW, #AW, #W, #Category, and #type are the number of commits compilabled by SpotBugs, unactionable warnings, actionable warnings, all warnings, distinct categories, and distinct types respectively.

No.	Project	Time period	#LoC	#Commits	#C. commits	#UW	#AW	#W	#Category	#Type
1	bcel	2001/10/29 ~ 2023/02/11	10k+~168k+	2400	1913	595	30	625	7	41
2	codec	2003/04/26~2022/11/26	5k+~55k+	2296	1966	595	70	665	6	31
3	collections	2001/04/14~2022/11/02	1k+~136k+	3810	1144	642	3	645	6	29
4	configuration	2003/12/23 ~ 2022/12/24	20k+~134k+	3743	3169	2843	62	2905	10	56
5	dbcp	2001/04/15~2023/02/10	8k+~55k+	2791	638	150	15	165	9	33
6	digester	2001/05/03 ~ 2023/02/04	3k+~54k+	2233	1622	620	30	650	9	40
7	fileupload	2002/03/24~ 2022/10/25	2k+~16k+	1284	1064	528	40	568	6	26
8	mavendp	2006/04/10~2022/10/30	5k+~37k+	1165	748	900	18	918	7	37
9	net	2002/04/03~2022/11/08	50k+~57k+	2683	1672	687	31	718	8	50
10	pool	2001/04/15~2023/02/10	6k+~34k+	2656	1638	2106	175	2281	8	39
All	/	/	110k+~746k+	25061	15574	9666	474	10140	10	137

RQ1: How is the performance of PTMs on AWI in comparison to the SOTA ML-based AWI approach?
RQ2: How do the data preprocessing ways affect the performance of PTMs on AWI?

- **RQ2.1**: What is the impact of warning context extraction?
- **RQ2.2**: What is the impact of warning context abstraction?

RQ3: How do the model training components affect the performance of PTMs on AWI?

- **RQ3.1**: What is the role of a code-related pre-training component?
- **RQ3.2**: What is the role of a fine-tuning component?

RQ4: How do the model prediction scenarios affect the performance of PTMs on AWI?
RQ5: What are the reasons for incorrect predictions in the current PTM-based AWI approach?

3.3 Evaluation Metrics

We adopt the following evaluation metrics. These metrics are mainly calculated by the confusion matrix, which includes True Positive (i.e., the number of actionable warnings accurately predicted as actionable warnings), False Positive (i.e., the number of unactionable warnings falsely predicted as actionable warnings), True Negative (i.e., the number of unactionable warnings accurately predicted as unactionable warnings), and False Negative (i.e., the number of actionable warnings falsely predicted as unactionable warnings). The definitions of selected evaluation metrics are shown below.

- $Precision = \frac{TP}{TP+FP}$ presents that the ratio of correctly predicted actionable warnings over all warnings predicted as being actionable ones.
- $Recall = \frac{TP}{TP+FN}$ describes that the ratio of correctly predicted actionable warnings over all actionable warnings.
- $F1 = \frac{2*Precision*Recall}{Precision+Recall}$, aka F1-Score, is the weighted harmonic mean of precision and recall.
- T_time, aka training time, is the time overhead that trains an AWI classifier. The unit of T_time is the second(s).
- I_time, aka inference time, is the time overhead that relies on a well-trained classifier to predict the targeted warnings. The unit of I_time is the second(s).

3.4 Selection of PTMs

We focus on open-source PTMs instead of unreleased PTMs (e.g., Codex [9]). On the one hand, as shown in Section 3.1, it is a core subtask to investigate the role of the fine-tuning component in the PTM-based AWI

Just Accepted

Table 2. Details of selected PTMs for AWI.

No.	Model	Architecture	No. of parameters	Organization
1	CodeT5-small [58]	Encoder-decoder	60M	Salesforce
2	CodeT5-base [58]	Encoder-decoder	220M	Salesforce
3	CodeT5-large [36]	Encoder-decoder	770M	Salesforce
4	PLBART [1]	Encoder-decoder	140M	UCLANLP
5	CodeBERT [16]	Encoder-only	125M	Microsoft
6	GraphCodeBERT [22]	Encoder-only	125M	Microsoft
7	UnixCoder [21]	Encoder-only	125M	Microsoft
8	QwenCoder [3]	Decoder-only	500M	Qwen Team
9	CodeGPT [43]	Decoder-only	124M	Microsoft

approach. In comparison to unreleased PTMs, open-source PTMs, whose source code is publicly available, can support the fine-tuning component for AWI. On the other hand, unreleased PTMs make it difficult to replicate and validate the results. In contrast, open-source PTMs facilitate transparency and accessibility, making it easier to conduct reproducible evaluations.

Table 2 shows nine selected PTMs for AWI. The nine PTMs present the following characteristics. First, each PTM is well-trained on unlabeled and large-scale codebases because AWI is a code-related task. Second, as shown in Section 2.3, such PTMs encompass three typical architectures (i.e., encoder-decoder, encoder-only, and decoder-only models). Third, such PTMs span different numbers of parameters. In particular, CodeT5 contains multiple versions with different numbers of parameters (i.e., 60M ~ 770M), which helps understand the impact of the parameter amount on the PTM-based AWI approach. Fourth, such PTMs originate from different authoritative organizations (e.g., Salesforce and Microsoft). Fifth, such PTMs are publicly accessible from an up-to-now largest open-source large language model community called Hugging Face [15]. In addition, such PTMs have already been widely used for software engineering tasks [25]. Due to the limitations of our computing resources (i.e., two Tesla V100-SXM2 GPUs), we select PTMs with a maximum of 770M parameters.

3.5 Dataset

We adopt two typical warning datasets for the experimental evaluation.

D1. The first dataset is obtained by using SpotBugs to scan open-source and large-scale Java projects. Based on the work of Ge et al. [20], we obtain 10140 distinct SpotBugs warnings (i.e., 9666 unactionable and 474 actionable warnings) from 10 open-source and large-scale Java projects. Specifically, Ge et al. [20] incorporate manual inspection and verification latency into postprocess labels from an advanced closed-warning heuristic and thereby acquire credible labels. Based on the 10 projects with 25K+ revisions and 2087K+ SpotBugs warnings, Ge et al. construct a qualified warning dataset with 11975 distinct warnings. The more warning dataset collection details can be seen in the work of Ge et al. [20]. Subsequently, based on the warning details provided by Ge et al., we leverage JGit[1] to extract the source code from the class/method containing a warning and the warning line numbers. However, the inherent issues in JGit make it unfeasible to trace and extract the source code of 1835 warnings. Finally, we obtain 10140 distinct warnings for the experimental evaluation. The specific warning dataset information is shown in Table 1. Noted, SpotBugs involves ten warning categories (e.g., MALOCIOUS_CODE), of which one contains multiple warning types (e.g., MS_PKGPROTE).

[1]JGit [27] is a tailored library used for operating Git repositories in Java projects.

Just Accepted

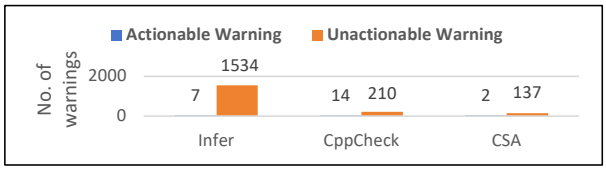


Fig. 3. The distribution of 1904 warnings reported by Infer, CppCheck, and CSA.

D2. The second dataset, containing 1904 warnings, is obtained by using Infer, CppCheck, and CSA to scan C/C++ projects. Specifically, Wen et al. [60] collect 2643 warnings by using Infer, CppCheck, and CSA to scan 14 real-world C/C++ projects. When we extract the contexts of these warnings for AWI, it is observed that there are two invalid project links (i.e., "m4" and "TecentOS-tiny") and a few missing warning locations. In the end, we gather 1904 warnings with associated contexts for the experimental evaluation. In particular, as the small number of warnings may cause meaningless results during the experimental process, we conduct experiments by gathering all warnings reported by three SCAs, rather than separately conducting experiments on the warnings reported by each SCA. Fig. 3 shows the distribution of the 1904 warnings. It is noted that these warnings involve six categories, including NULL_ POINTER_DEREFERENCE, UNINITIALIZED_VARIABLE, USE_AFTER_FREE, DIVIDE_BY_ZERO, MEMORY_LEAK, and BUFFER_OVERFLOW.

3.6 Implementation Details

In the SOTA ML-based AWI approach, we use PyTorch [49] to implement CNN and LSTM for AWI. The architecture design of CNN and LSTM follows the work of Lee. et al. [38] and Koc et al. [34] respectively. However, due to different numbers of warnings and different sources of warnings [34, 38] between our study and the existing studies, we reset the parameters in CNN and LSTM. For CNN, we set the word embedding dimension to 128, the batch size to 16, the dropout rate to 0.5, and use the SGD optimizer [42] with 0.005 learning rate. For LSTM, we set the word embedding dimension to 128 and the batch size to 16. In the PTM-based AWI approach, we use the implementation version in Hugging Face. We set the input sequence length to 256 and use the Adam optimizer [32] with $5e{-}5$ learning rate. Particularly, due to the limited computing resource in our sever and different amounts of parameters in PTMs, we separately set the batch size for different PTMs, thereby making PTMs successfully run on our server. The specific batch size can be seen in the repository [48]. All experiments are conducted with one Ubuntu 18.04.3 server with two Tesla V100-SXM2 GPUs.

4 RESULTS AND ANALYSIS

4.1 RQ1: Performance of PTM-based AWI

Motivation. This RQ aims to explore the performance of PTM-based AWI in terms of effectiveness and efficiency. Besides, this RQ investigates the performance differences of nine PTMs on AWI.

Design. To answer this RQ, we compare the SOTA ML-based with PTM-based AWI approaches. Based on the classification of ML-based AWI approaches in Section 2.2, we identify the SOTA ML-based AWI approach. Specifically, as shown in Section 3.1, the PTM-based AWI approach extracts source code from the method containing a warning as the warning context, thereby representing warnings for AWI. It indicates that the PTM-based AWI approach falls under the text-based AWI approach. As such, we consider selecting the SOTA ML-based AWI approach from the existing text-based AWI approaches. By further investigating the existing text-based AWI approaches, it is observed that the DL-based AWI approach generally performs better than the traditional ML-based AWI approach [30, 34, 64]. In the end, we select the text-based and DL-based AWI approach as the SOTA ML-based AWI approach.

Just Accepted

Table 3. The performance of the DL-based and PTM-based AWI approaches.

Approach	No. of parameters	D1 (10140 Java warnings from SpotBugs)					D2 (1904 C/C++ warnings from Infer, CppCheck, and CSA)					Architecture
		Precision	Recall	F1	T_time	I_time	Precision	Recall	F1	T_time	I_time	
CNN	/	**0.66**	0.15	0.24	20350	190	0.45	0.36	0.40	1800	20	/
LSTM	/	0.54	0.25	0.34	22580	230	0.45	0.35	0.40	2080	25	/
CodeT5-small	60M	0.50	0.28	0.36	**4500**	**70**	0.65	0.41	0.50	**400**	10	Encoder-decoder
CodeT5-base	220M	0.60	0.33	0.43	19600	250	**1.00**	0.21	0.35	630	20	
CodeT5-large	770M	0.65	0.34	0.44	84500	715	0.45	0.36	0.40	2290	30	
PLBART	140M	0.59	0.36	**0.45**	23950	285	0.71	0.36	0.48	1090	15	
CodeBERT	125M	0.62	0.32	0.42	14500	385	0.43	0.34	0.40	465	**5**	Encoder-only
GraphCodeBERT	125M	0.59	**0.37**	**0.45**	7500	150	**1.00**	0.21	0.35	450	**5**	
UnixCoder	125M	0.54	0.35	0.42	16000	250	0.78	**0.50**	**0.61**	450	**5**	
QwenCoder	500M	0.65	0.29	0.40	30950	250	0.23	**0.50**	0.32	1440	15	Decoder-only
CodeGPT	124M	0.58	0.34	0.43	8000	90	0.60	**0.50**	0.55	650	**5**	

By following previous text-based and DL-based AWI studies [33, 34, 38, 54, 64], we extract source code from the method containing a warning as the warning context, abstract identifiers and literals in the extracted warning context, and use the DL model to train a classifier for AWI on the abstracted warning context. Of these, the detailed warning context abstraction process can be seen in Section 3.1. In addition, we attempt CNN and LSTM for AWI because the results of previous studies [34, 38, 54, 64] demonstrate the superior performance of CNN and LSTM on AWI.

To perform a fair comparison, we perform the same warning context extraction and abstraction with the SOTA ML-based AWI approach. Subsequently, we use the code-related pre-training and fine-tuning components to construct an AWI classifier based on the obtained warning context.

In all, this RQ involves 22 experiments (11 approaches * two datasets). In each experiment, we conduct the five-fold cross-validation [5].

Results. In terms of effectiveness on *D1*, Table 3 shows that CNN achieves a precision of 0.66, which slightly outperforms nine PTMs by 0.01 ∼ 0.16. However, the precision of PTMs, except for CodeT5-small and UnixCoder, exceeds that of LSTM. Also, it is observed that the recall and F1 of PTMs are consistently and greatly better than those of CNN and LSTM. On average, the precision, recall, and F1 of the SOTA ML-based AWI approach are 0.60, 0.20, and 0.29 respectively. The precision, recall, and F1 of the PTM-based AWI approach are 0.59, 0.33, and 0.42 respectively. That is, the average precision of the PTM-based AWI approach is nearly 2% worse than that of the SOTA ML-based AWI approach. In contrast, the average recall and F1 of the PTM-based AWI approach are 65% and 45% better than those of the SOTA ML-based AWI approach respectively. Further, the Mann-Whitney U-Test [46] shows no significant difference between the PTM-based and SOTA ML-based AWI approaches in terms of precision. Conversely, there are significant differences between the PTM-based and SOTA ML-based AWI approaches in terms of recall and F1. Such results indicate that the PTM-based AWI approach can substantially improve the recall and F1 of the SOTA ML-based AWI approach, while still maintaining similar precision to the SOTA ML-based AWI approach.

In terms of effectiveness on *D2*, Table 3 shows that nine PTMs outperform CNN and LSTM on AWI in most cases. On average, the precision, recall, and F1 of the PTM-based AWI approach are 0.65, 0.38, and 0.44 respectively. In contrast, the precision, recall, and F1 of the SOTA ML-based AWI approach are 0.45, 0.36, and 0.4 respectively. The Mann-Whitney U-Test [46] shows no significant difference between the PTM-based and SOTA ML-based AWI approaches in terms of precision, recall, and F1. However, the average precision, recall, and F1 of the PTM-based AWI approach are consistently better than those of the SOTA ML-based AWI approach.

The above results on *D1* and *D2* indicate that the overall effectiveness of the PTM-based AWI approach is superior to the SOTA ML-based AWI approaches. By further investigation, there are two possible factors for

Just Accepted

the improvement of PTMs over DL models on AWI. On the one hand, PTMs leverage extensive codebases to yield more significant vector representations. By contrast, CNN and LSTM are trained on a limited number of warnings. On the other hand, PTMs employ the transformer architecture, which can provide the context for any position in a given input sequence via the self-attention mechanism. However, CNN and LSTM cannot capture the relative position information in the warning context due to a lack of the transformer architecture.

In terms of efficiency, Table 3 shows a similar trend on *D1* and *D2*. Specifically, the training time of the PTM-based AWI approach is different from that of the SOTA ML-based AWI approach. However, the Mann-Whitney U-Test [46] shows no significant difference between the PTM-based and SOTA ML-based AWI approaches in terms of training time. In particular, CodeT5-small achieves the optimal training time, which substantially outperforms CNN and LSTM. The main reason is that instead of CNN and LSTM training new network models from scratch for AWI, CodeT5-small with a small number of parameters (i.e., 60M) is only fine-tuned for AWI. In contrast, CodeT5-large has the longest training time, which is much slower than CNN and LSTM. Even though CodeT5-large is only fine-tuned for AWI in comparison to CNN and LSTM, CodeT5-large has the maximum number of parameters (i.e., 770M), which takes up a lot of the training time. In contrast, there is a slight difference in the inference time between the PTM-based and SOTA ML-based AWI approach. This may be caused by different numbers of parameters and different network structures in DL models and PTMs. However, the Mann-Whitney U-Test [46] shows no significant difference between the PTM-based and SOTA ML-based AWI approaches in terms of inference time.

With regardless of *D1* and *D2*, it is noticed that nine PTMs on AWI have differences in terms of precision, recall, F1, training time, and inference time. Such differences may be caused by corpora with different codebase scales and datapoints, different numbers of parameters, and different network architectures among PTMs. Furthermore, we analyze the effectiveness and efficiency differences of AWI among three categories of PTM architectures. Specifically, we calculate the AWI performance in the same PTM architecture respectively. For example, the precision of the decoder-only-based AWI is the average precision of QwenCoder and CodeGPT. Among PTMs with three architectures, the AWI performance differences in the precision, recall, and F1 are only $0.01 \sim 0.02$ on *D1* respectively. On *D2*, the precision, recall, and F1 in the encoder-decoder-based and encoder-only-based AWI are very close. In contrast, the precision/recall of the decoder-only-based AWI is worse/better than those of the encoder-decoder-based and encoder-only-based PTMs respectively. Consequently, the F1 of the decoder-only-based AWI is similar to that of the encoder-decoder-based and encoder-only-based PTMs. It is observed that there is an inconsistent result between *D1* and *D2*. Such a phenomenon could be caused by various warning categories in *D1* and *D2*, which are brought by different programming language characteristics and different static analysis techniques in SCAs. On *D1*/*D2*, the difference in the training time is over 10000s/1000s, while the difference in the inference time is within 60s/5s respectively. Noted, due to different number of warning datasets, the training and inference time on *D1* and *D2* are far apart. In particular, it is observed on *D1* and *D2* that CodeT5-small/CodeT5-large with the smallest/largest number of parameters in the encoder-decoder architecture takes the minimum/maximum training and inference time among nine PTMs. Further investigation reveal that the difference in the training and inference time is mainly attributed to the number of parameters in PTMs. The above results indicate that different PTM architectures have a slight impact on the AWI effectiveness, and the impact of different PTM architectures on the AWI efficiency mainly depends on the number of parameters in specific PTMs.

Answering RQ1: Overall, the PTM-based AWI approach can substantially outperform the SOTA ML-based AWI approach in terms of precision, recall, and F1. In addition, the PTM-based AWI approach has similar training and inference time to the SOTA ML-based AWI approach.

Just Accepted

(a) Precision. (b) Recall. (c) F1.

(d) Training time. (e) Inference time.

Fig. 4. The performance of the PTM-based AWI approach in two warning context extraction settings based on *D1* (i.e., 10140 Java warnings from SpotBugs). Average is the average results of nine PTMs on AWI.

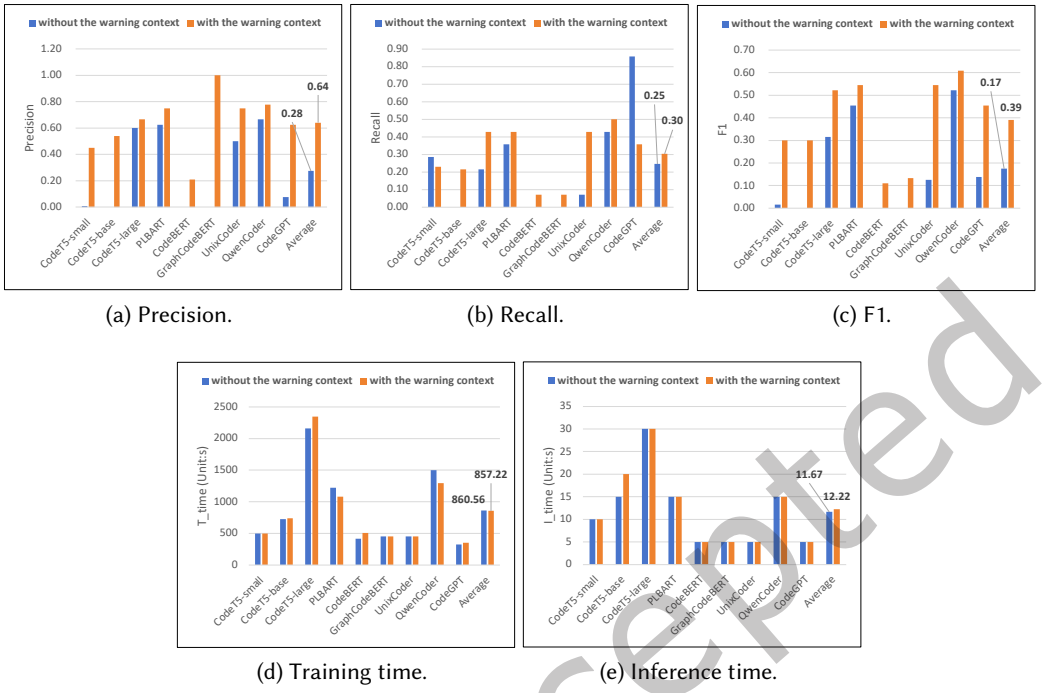
Just Accepted

4.2 RQ2: Analysis in the Data Preprocessing

Motivation. Based on the typical PTM-based AWI workflow in Section 3.1, the data preprocessing contains the warning context extraction and abstraction. The warning context can help analyze the cause of a warning. Previous studies [33, 38, 54] show that the warning context plays a crucial role in AWI. The warning context abstraction is to rename raw code tokens to a set of predefined tokens, thereby reducing the number of code tokens in the warning context. The existing studies explicitly demonstrate that the abstracted warning context is beneficial for AWI [34, 64] compared to the raw warning context. However, the impact of both warning context extraction and abstraction on the PTM-based AWI approach has not yet been fully investigated. Thus, this RQ explores the performance of the PTM-based AWI approach in different data preprocessing ways.

4.2.1 RQ2.1: The impact of warning context extraction.

Design. To answer this RQ, we compare the performance difference of the PTM-based AWI approach with and without warning contexts. As for each warning, we extract the source code from the warning line numbers, which is called without the warning context. By contrast, as described in Section 3.1, we extract source code from the method containing a warning, which is called with the warning context. It is noted that not all warnings, especially for warnings related to class fields, are reported inside methods. Thus, the contexts of these warnings are the same between with and without warning context scenarios. Yet, the warnings outside methods only account for a very small proportion of all warnings.



(a) Precision. (b) Recall. (c) F1.

(d) Training time. (e) Inference time.

Fig. 5. The performance of the PTM-based AWI approach in two warning context extraction settings based on *D2* (i.e., 1904 C/C++ warnings from Infer, CppCheck, and CSA). Average is the average results of nine PTMs on AWI.

Just Accepted

In all, this RQ involves 36 experiments (nine PTMs * two extraction settings * two datasets). In each experiment, we conduct the five-fold cross-validation.

Results. In terms of *D1*, Fig. 4 describes that in comparison to the scenario without the warning context, nine PTMs (except for CodeT5-small and CodeT5-large) show higher precision, recall, and F1 when using the warning context for AWI. In particular, CodeT5-small and CodeT5-large without the warning context demonstrate higher precision, recall, and F1 than those with the warning context respectively. Such a phenomenon is opposite on CodeT5-base. This suggests that PTMs with larger parameters do not necessarily lead to better AWI effectiveness in the scenario with/without the warning context. On average, the PTM-based AWI approach with the warning context is 2%, 6%, and 5% better than that without the warning context.

In terms of *D2*, Fig. 5 shows that compared to the scenario without the warning context, nine PTMs show higher precision, recall, and F1 in most cases when using the warning context for AWI. On average, the PTM-based AWI approach with the warning context is 129%, 20%, and 129% better than that without the warning context. It is noted that the precision, recall, or F1 of a few PTMs (e.g., CodeT5-small) is 0 in the scenario without the warning context. In contrast, PTMs obtain non-zero precision, recall, and F1 in the scenario with the warning context. It further demonstrates that the PTM-based AWI approach with the warning context is helpful for AWI.

Overall, the above results highlight the benefits of the warning context for the PTM-based AWI approach. Without the warning context, the source code extracted from the warning line numbers could only denote the appearance of warnings. With the warning context, in addition to involving the source code extracted from the

(a) Precision. (b) Recall. (c) F1.

(d) Training time. (e) Inference time.

Fig. 6. The performance of the PTM-based AWI approach in two warning context abstraction settings based on *D1* (i.e., 10140 Java warnings from SpotBugs). Average is the average results of nine PTMs.

warning line numbers, the source code extracted from the method containing a warning could embrace the root cause of a warning, which can greatly bolster the PTM-based AWI effectiveness.

As shown in Fig. 4 and Fig. 5, the training time and inference time are mostly similar on the same PTM with and without the warning contexts. It indicates that the scenarios with and without warning contexts have little impact on the training and inference time of PTM-based AWI on *D1* and *D2*. Also, it is observed that on *D1* and *D2*, regardless of the presence or absence of warning context, the training and inference stages of CodeT5-large take the most time among nine PTMs. This could be because CodeT5-large has the most parameters (i.e., up to 770M) among nine PTMs.

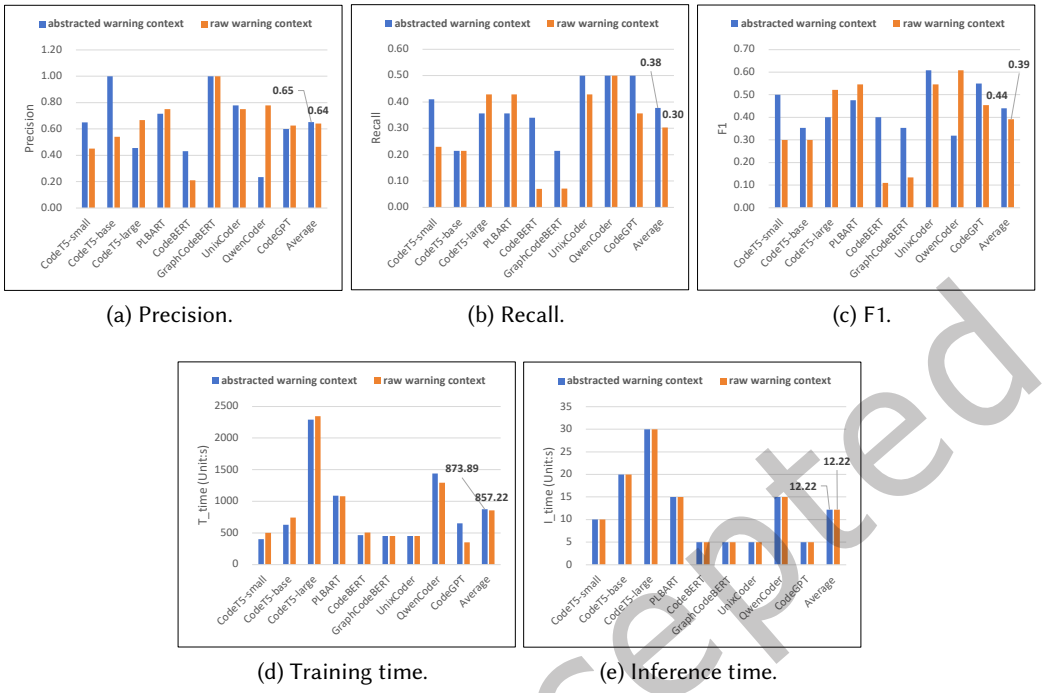
4.2.2 *RQ2.2: The impact of warning context abstraction.*

Design. To answer this RQ, we compare the performance difference of the PTM-based AWI approach between the raw and abstracted warning contexts. In terms of the raw warning context, we directly extract the source code from the method containing a warning. In terms of the abstracted warning context, we abstract the source code from the method containing a warning.

In all, this RQ involves 36 experiments (nine PTMs * two abstraction settings * two datasets). In each experiment, we conduct the five-fold cross-validation.

Results. In terms of *D1*, Fig. 6 shows that in comparison to the raw warning context, nine PTMs (except for CodeT5-base) basically improve the precision of AWI but reduce the recall and F1 of AWI when using the abstracted warning context for AWI. On average, the PTM-based AWI approach with the abstracted warning context has 7% better precision than that in the raw warning context, while having 3% worse recall than that

Just Accepted



(a) Precision. (b) Recall. (c) F1.

(d) Training time. (e) Inference time.

Fig. 7. The performance of the PTM-based AWI approach in two warning context abstraction settings based on *D2* (i.e., 1904 C/C++ warnings from Infer, CppCheck, and CSA). Average is the average results of nine PTMs.

Just Accepted

in the raw warning context. Consequently, there is a very close F1 between the abstracted and raw warning contexts. Overall, it indicates that the abstracted warning context is more beneficial for precision but harmful to recall in the PTM-based AWI approach compared to the raw warning context.

In terms of *D2*, Fig. 7 shows that compared to the raw warning context, PTMs improve the precision, recall, and F1 of AWI in most cases when using the abstracted warning context for AWI. On average, the PTM-based AWI approach with the abstracted warning context has 2%, 21%, and 11% better precision, recall, and F1 than those in the raw warning context respectively. Overall, it implies that the abstracted warning context can enhance the PTM-based AWI approach.

Based on the above results, it is observed that the impact of the abstracted and raw warning contexts on the PTM-based AWI approach between *D1* and *D2* is inconsistent in terms of recall and F1. The main reason is shown as follows. Specifically, *D1* are obtained by using SpotBugs to scan Java projects. *D2* are obtained by using Infer, CppCheck, and CSA to scan C/C++ projects. Due to different programming language characteristics between Java and C/C++ as well as different static analysis techniques in SCAs, the collected warnings have various categories on *D1* and *D2*. As shown in Section 3.5, the warning category on *D2* is mainly security-related. In addition to the security-related category, *D1* contains other warning categories (e.g., BAD_PRACTICE). Besides, the typical warning category (i.e., BUFFER OVERFLOW) is unique to C/C++. Instead of causing BUFFER OVERFLOW, Java language provides bounds checking at runtime, which generally throws exceptions (e.g., *ArrayIndexOutOfBoundsException*). Therefore, the different warning categories on *D1* and *D2* may yield varying conclusions when using the abstracted and raw warning contexts for PTM-based AWI.

As described in Fig. 6 and Fig. 7, in the same PTM, the training and inference time are basically similar between the abstracted and raw warning contexts on *D1* and *D2*. It indicates that the raw or abstracted warning contexts have little impact on the training and inference time of the PTM-based AWI approach. Also, it is observed that regardless of the abstracted and raw warning contexts, the training and inference time of CodeT5-large are the slowest among nine PTMs on *D1* and *D2*. This could be caused by CodeT5-large with the most parameters (i.e., up to 770M) among nine PTMs.

Answering RQ2: The extracted warning context can consistently boost the PTM-based AWI approach on Java and C/C++ warnings. The abstracted warning context can improve the precision but slightly hinder the recall of the PTM-based AWI approach on Java warnings, while increasing the precision and recall of the PTM-based AWI approach. In addition, the different warning context extraction/abstraction ways have little impact on the training and inference time of the PTM-based approach.

4.3 RQ3: Analysis in the Model Training

Motivation. The results in RQ1 show that the PTM-based AWI approach outperforms the SOTA ML-based AWI approach. The SOTA ML-based AWI approach is trained in a traditional pipeline, i.e., supervised learning on labeled warnings. In contrast, the PTM-based AWI approach involves two components in the model training, including a pre-training component for a general task with self-supervised learning on unlabeled and large-scale codebases and a fine-tuning component for a downstream task with supervised learning on labeled warnings. Thus, this RQ separately investigates the role of the code-related pre-training and fine-tuning components when using PTMs for AWI.

4.3.1 RQ3.1: The role of a code-related pre-training component.

Design. Based on the classification of PTM architectures in Section 2.3, we select four models (i.e., the encoder-decoder T5, the encoder-only BERT, and the decoder-only Qwen/GPT) as baselines without a code-related pre-training component. Correspondingly, we select CodeT5 family/CodeBERT/QwenCoder/CodeGPT, which are obtained by pre-training T5/BERT/Qwen/GPT on massive codebases respectively, as PTMs with a code-related pre-training component. Section 4.2 shows that the PTM-based AWI approach with the abstracted warning context achieves comparable performance on *D1* and *D2*. Thus, we select the abstracted warning context for the experimental evaluation in this RQ.

In all, there are 20 experiments (i.e., five PTMs * two pre-training settings * two datasets). In each experiment, we conduct the five-fold cross-validation.

Results. As shown in Table 4, as for the encoder-decoder PTM on *D1* and *D2*, the recall and F1 of the CodeT5 family consistently outperform those of T5. In contrast, the precision of the CodeT5 family is sometimes slightly worse than that of T5. In particular, as the number of parameters in the CodeT5 family increases, the improvement degree on T5 is greater in terms of *D1*. However, such a phenomenon is not reflected on *D2*. As for the encoder-only and decoder-only PTMs on *D1*, the precision of CodeBERT/QwenCoder/CodeGPT exceeds that of BERT/Qwen/GPT, while the recall of CodeBERT/QwenCoder/CodeGPT does not exceed that of BERT/Qwen/GPT. As for the encoder-only and decoder-only PTMs on *D2*, the precision, recall, and F1 of CodeBERT/QwenCoder/CodeGPT consistently exceed those of BERT/Qwen/GPT. It can be observed that the results on *D1* and *D2* are different. By further analysis, the possible reason can be attributed to various warning categories between *D1* and *D2*, which are caused by different programming language characteristics and different static analysis techniques in SCAs. Overall, in terms of precision, recall, and F1, the PTM-based AWI approach with a code-related pre-training component improves that without a code-related pre-training component by 9% (2%), 3% (39%), and 5% (50%) on *D1* (*D2*) respectively. It signifies that the code-related pre-training component can provide benefits for PTM-based AWI.

Just Accepted

Table 4. Performance of the PTM-based AWI approach with/without a code-related pre-training component.

Metric	Encoder-decoder				Encoder-only		Decoder-only			
	T5	CodeT5-small	CodeT5-base	CodeT5-large	BERT	CodeBERT	Qwen	QwenCoder	GPT	CodeGPT
D1 (10140 Java warnings from SpotBugs)										
Precision	0.52	0.50	0.60	0.65	0.60	0.62	0.56	0.65	0.51	0.58
Recall	0.25	0.28	0.33	0.34	0.34	0.32	0.29	0.29	0.36	0.34
F1	0.34	0.36	0.43	0.44	0.43	0.42	0.38	0.40	0.42	0.43
T_time	15120	4500	19600	84500	8500	14500	30500	30950	8000	8000
I_time	185	70	250	715	85	385	260	250	100	90
D2 (1904 C/C++ warnings from Infer, CppCheck, and CSA)										
Precision	1.00	0.65	1.00	0.45	0.40	0.43	0.21	0.23	0.58	0.60
Recall	0.07	0.41	0.21	0.36	0.36	0.34	0.50	0.50	0.21	0.50
F1	0.13	0.50	0.35	0.40	0.38	0.40	0.30	0.32	0.31	0.55
T_time	730	400	630	2290	285	465	1275	1440	600	650
I_time	10	10	20	30	5	5	15	15	10	5

Table 4 shows a similar trend of training and inference time on *D1* and *D2*. Specifically, as for the encoder-decoder PTM, the training and inference time of CodeT5-small are much faster than those of T5. Although CodeT5-small is obtained by training T5 on codebases, the number of parameters in CodeT5-small (i.e., 60M) is obviously smaller than that in T5 (i.e., 220M). Thus, the efficiency of CodeT5-small is better than that of T5 in AWI. However, the training and inference time of CodeT5-base are similar to those of T5. The training and inference time of CodeT5-large are slower than those of T5. The main reasons are shown as follows. Due to the same number of parameters between CodeT5-base and T5, the time difference between CodeT5-base and T5 is very close. In contrast, the time difference between CodeT5-large and T5 could stem from the significant difference in the number of parameters. As for the encoder-only PTM, BERT runs faster than CodeBERT in terms of training and inference time. This is because the number of parameters in BERT (i.e., 110M) is smaller than that in CodeBERT (i.e., 125M). As for the decoder-only PTM, Qwen/GPT has similar training and inference time to QwenCoder/CodeGPT. This is because the number of parameters in Qwen(i.e., 500M)/GPT(i.e., 124M) and QwenCoder/CodeGPT is the same. The above results indicate that the efficiency of PTMs on AWI could not depend on the code-related pre-training component, but be related to the number of parameters in PTMs.

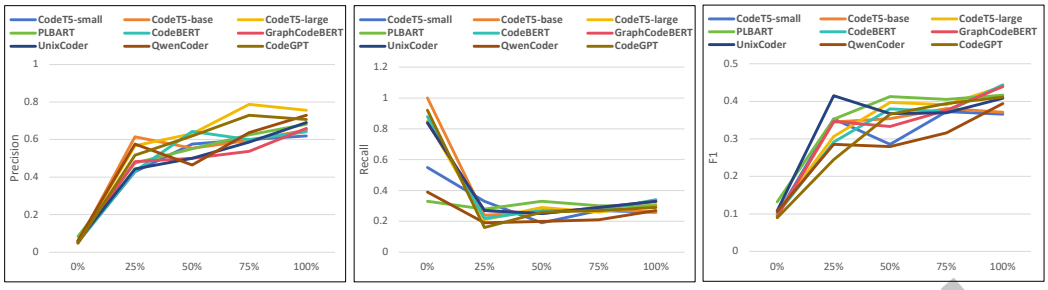
4.3.2 RQ3.2: The role of a fine-tuning component.

Design. To answer this RQ, we first rely on five-fold cross-validation to split all warnings of *D1* and *D2* respectively. Then, we randomly select one-fold warnings as the test set and the remaining four-fold warnings as the training set. In the training set, we consider zero-fold, one-fold, two-fold, three-fold, and four-fold warnings as the fine-tuning corpora. It indicates that there are five fine-tuning corpora (i.e., 0%, 25%, 50%, 75%, and 100% warnings of the training set). Next, we train the PTM-based AWI classifier on each fine-tuning corpus and evaluate this classifier on the test set. Similar to Section 4.3.1, we select the abstracted warning context for the experimental evaluation in this RQ.

In all, there are 90 experiments (five fine-tuning corpora * nine PTMs * two datasets). The existing studies showcase a determinate observation, i.e., more fine-tuning corpora generally lead to longer training time in a fixed PTM [14]. In contrast, as the inference process relies on a fixed PTM and the same test set, the inference time remains stable [53]. Thus, in this RQ, we only focus on the effectiveness evaluation of the PTM-based AWI approach via precision, recall, and F1.

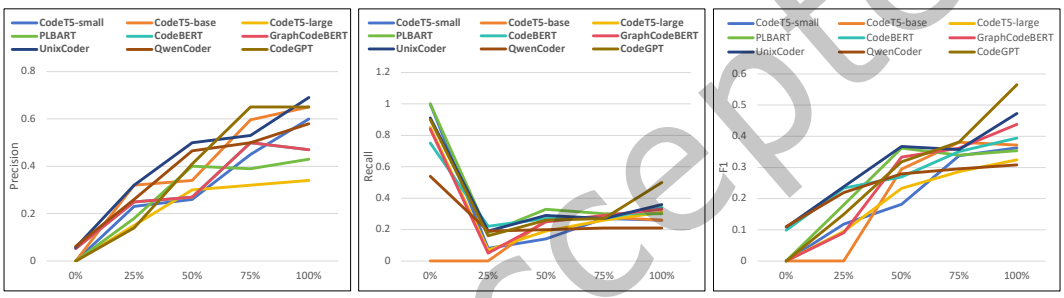
Results. As shown in Fig. 8 and Fig. 9, there is a similar trend on *D1* and *D2*. Specifically, when the percentage of fine-tuning corpora in the training set increases from 0%~25%, the precision and F1 of nine PTMs on AWI are

Just Accepted



(a) Precision. (b) Recall. (c) F1.

Fig. 8. The performance of the PTM-based AWI approach in different fine-tuning corpora of *D1* (i.e., 10140 Java warnings from SpotBugs).



(a) Precision. (b) Recall. (c) F1.

Fig. 9. The performance of the PTM-based AWI approach in different fine-tuning corpora of *D2* (i.e., 1904 C/C++ warnings from Infer, CppCheck, and CSA).

obviously rising. Meanwhile, the recall of nine PTMs on AWI is sharply declined. When the fine-tuning corpora increase from 25% to 100% of the training set, the PTM-based AWI approach maintains a steadily increasing precision, while keeping a slowly rising recall. The further analysis of the above phenomenon is shown as follows. First, it is observed that nine PTMs with no fine-tuning corpora show inferior precision and F1 as well as superior recall. Despite acquiring valuable knowledge from the code-related pre-training component, these PTMs could not adapt to the downstream task (i.e., AWI) without a fine-tuning process. The possible reason may be that PTMs primarily capture the syntactic and semantic structures of codebases rather than learning the task-specific decision classification boundary before fine-tuning. Also, the classification head of PTMs is randomly assigned weights without a fine-tuning process, which may cause the output results to be biased towards actionable warnings [7]. Second, with a small amount of fine-tuning corpora, PTMs begin to learn the discriminative characteristics of actionable and unactionable warnings. It leads to a significant increase in precision and a rapid reduction in recall. However, due to the limited scope of the fine-tuning corpora, PTMs have not yet formed a stable boundary and become ambiguous on certain targeted warnings. Third, as the size of the fine-tuning corpora increases, PTMs are exposed to more diverse and representative characteristics of both actionable and unactionable warnings, thereby incrementally refining the decision boundary. Thus, the precision of the PTM-based AWI approach is improved steadily, while the recall of the PTM-based AWI approach exhibits a mild upward slope. Correspondingly, the F1

Just Accepted

Table 5. Within- and cross-project AWI scenarios.

AWI scenario		Training set	Test set
Within-project	Within1	All warnings in nine projects and 50% of warnings in the tenth project	Remaining 50% of warnings in the tenth project
	Within2	50% of warnings in the tenth project	
Cross-project	Cross1	All warnings in nine projects	
	Cross2	N/A	

of the PTM-based AWI approach shows a steadily upward trend. Particularly, when the fine-tuning corpora are all the training set, the AWI effectiveness still has no downward trend. Such a finding further underscores the advantages of the fine-tuning component in the PTM-based AWI approach, which can enable PTMs to acquire task-specific expertise and maximize the utilization of the knowledge gained from the pre-training component.

Answering RQ3: The code-related pre-training component can acquire generic knowledge from codebases to further enhance the overall effectiveness of the PTM-based AWI approach, while independent of the efficiency of the PTM-based AWI approach. The fine-tuning component can boost the overall effectiveness of PTMs on AWI.

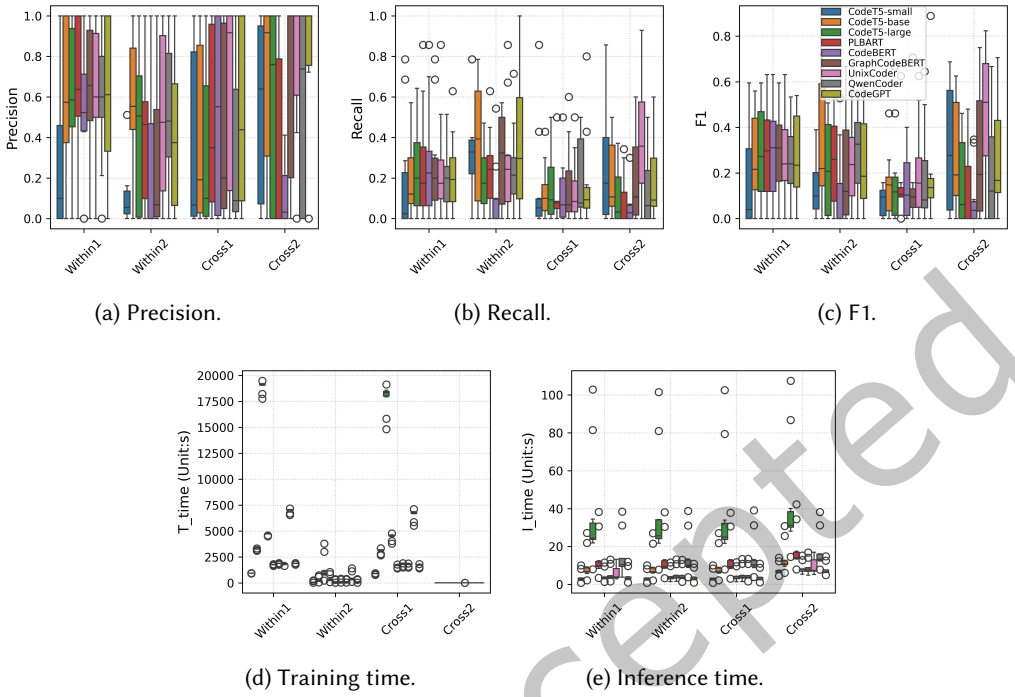
4.4 RQ4: Analysis in the Model Prediction

Motivation. The typical PTM-based AWI workflow in Section 3.1 shows that the model prediction involves two scenarios, i.e., within- and cross-project AWI. However, little work explores how PTMs perform within- and cross-project AWI scenarios, which fails to understand the performance of PTMs in different model prediction scenarios. Thus, this RQ aims to bridge the above gap.

Design. To answer this RQ, we first use stratified sampling to take 50% of warnings as the training set and take the remaining 50% of warnings as the test set in each project of *D1*. Due to the class imbalance in all warnings of each project, we adopt stratified sampling rather than random sampling, so as to ensure that each respective set contains actionable warnings. Noted, as the number of *D2* is much smaller than that of *D1* and there is class imbalance in *D2*, the preliminary experimental results show that there are meaningless values on *D2*. Thus, we only conduct experiments on *D1*. After that, we design within- and cross-project AWI scenarios, which are shown in Table 5. There are two variants in the within- and cross-project AWI scenarios respectively. The difference between Within1 (Cross1) and Within2 (Cross2) is whether the training set contains warnings from the remaining nine projects when taking 50% of warnings in a project as the test set. Besides, to conduct a fair comparison between within- and cross-project AWI scenarios, the test set is the same in four variants. Such a rigorous design aims to further investigate whether the number of training set affects the performance of PTMs in within- and cross-project AWI scenarios. Similar to Section 4.3.1, we select the abstracted warning context for the experimental evaluation in this RQ. In all, there are 360 experiments (four variants * 10 projects * nine PTMs).

Results. As shown in Fig. 10, the median precision of Cross1 is mostly superior to that of the other three scenarios. However, the median precision of Cross1 exhibits a more dispersed distribution than that of the other three scenarios. It indicates that without the training set, the stability of the median precision in Cross1 performs worse than that in the other three scenarios. Different from the above phenomenon, the median recall and F1 of PTMs in the within-project AWI scenario are mostly higher than those in the cross-project AWI scenario. Overall, PTMs in the within-project AWI scenario perform better than those in the cross-project AWI scenario. The main reason could be that the training and test sets tend to be homogeneous in the within-project AWI

Just Accepted



(a) Precision. (b) Recall. (c) F1.

(d) Training time. (e) Inference time.

Fig. 10. The performance of the PTM-based AWI approach in within-project and cross-project AWI scenarios.

Just Accepted

scenario due to partially coming from the same project. Conversely, the training and test sets are heterogeneous in the cross-project AWI scenario due to coming from different projects.

In addition, it is observed that in nine PTMs, the median precision, recall, and F1 of Within1 are around 0.6, 0.2, and 0.3 respectively. Overall, such a result outperforms Within2. It indicates that it is helpful to boost the PTM-based AWI performance by increasing the number of training set in the within-project scenario. In contrast, the median values of precision, recall, and F1 in Cross1 are mostly worse than those in Cross2. It implies that the AWI-related expertise of PTMs gained by fine-tuning the training set does not work in the cross-project AWI scenario, and the warning dataset heterogeneity could be the main factor causing the effectiveness difference between Cross1 and Cross2.

As shown in Fig. 10, the training and inference time of four scenarios are very similar when using the same PTM for AWI. In particular, as CodeT5-large has the largest number of parameters in nine PTMs, the training and inference time of CodeT5-large is the slowest in nine PTMs. Additionally, it is noted that the training time of Cross2 is nearly zero. It is because there is no training set for Cross2. Through further analysis, the training and inference time are primarily related to the number of parameters in PTMs. The above result signifies that the efficiency of the PTM-based AWI is independent of different model prediction scenarios.

Answering RQ4: Overall, the PTM-based AWI approach in the within-project AWI scenario is more effective than that in the cross-project AWI scenario. However, when the same PTM is used for AWI, there is the very similar efficiency between within-project and cross-project AWI scenarios.

4.5 RQ5: Further Analysis of Incorrect Predictions in PTM-based AWI

Motivation. Despite showing superior performance compared to the SOTA ML-based AWI approach in Section 4.1, nine PTMs for AWI only achieve a total of precision, recall, and F1 of 70%, 59%, and 64% on *D1* (i.e., 10140 Java warnings from SpotBugs) respectively. Correspondingly, nine PTMs for AWI only achieve a total of precision, recall, and F1 of 100%, 55%, and 71% on *D2* (i.e., 1904 C/C++ warnings from Infer, CppCheck, and CSA) respectively. It indicates that the PTM-based AWI approach still has substantial room for improvement. Thus, this RQ analyzes the underperformance of current PTMs on AWI.

Design. Based on the five-fold cross-validation results of PTMs on AWI in Section 4.1, we randomly record the predicted labels of nine PTMs in the one-fold test set of *D1* (i.e., 2028 warnings) and *D2* (i.e., 380 warnings) respectively. Then we compare ground truth and predicted labels to confirm which warnings are wrongly classified by PTMs. Finally, we analyze the reasons why PTMs fail to classify warnings.

Results. By gathering all correctly classified warnings from nine PTMs, the final precision, recall, and F1 are 70%, 59%, and 64% on *D1* respectively. Correspondingly, the final precision, recall, and F1 are 100%, 55%, and 71% on *D2* respectively. Specifically, in terms of *D1*, the further statistics show that the 1974 warnings were correctly identified by nine PTMs. Of these, 1906 warnings are predicted correctly by nine PTMs at the same time, and 14 warnings are predicted correctly by at least one PTM. Also, it is observed that 54 warnings are not correctly predicted by any of the nine PTMs. In the 54 warnings, 11, 13, 6, 5, 9, 5, 2, and 3 warnings fall into MALICIOUS_CODE, DODGY_CODE, BAD_PRACTICE, CORRECTNESS, PERFORMANCE, I18N, MULTITHREADED, CORRECTNESS, and SECURITY respectively. In terms of *D2*, 370 warnings were correctly identified by nine PTMs. Of these, 351 warnings are predicted correctly by nine PTMs at the same time, and 10 warnings are predicted correctly by at least one PTM. Also, it is observed that three warnings are not correctly predicted by any of the nine PTMs. In the three warnings, one and two warnings are involved in NULL_POINTER_DEREFERENCE and MEMORY_LEAK respectively. The results indicate that the three warnings are actually actionable ones but are falsely identified as unactionable ones by nine PTMs. After that, we summarize reasons why PTMs wrongly identify the 57 (i.e., 54 + 3) warnings via the rigorous manual analysis.

(1) Similar or even the same contexts in actionable and unactionable warnings. This reason involves 19 warnings. It is observed that warnings in the same category (especially for *BAD_PRACTICE*, *DODGY_CODE*, and *I18N*) tend to have similar contexts. In these warnings, only a tiny part of warnings are acted on and fixed by developers. In contrast, most warnings are ignored by developers due to generally not affecting the functional correctness of the program. In addition, multiple warnings may appear in the same method, where some are actionable and others are unactionable. However, based on the warning context extraction of our study, warnings in the same method have the same warning context. Thus, the above phenomena may cause PTMs to give ambiguous warning labels.

(2) Insufficient or unavailable warning contexts. 33 warnings are related to this reason. It is found that PTMs perform poorly on AWI when warning contexts are insufficient or unavailable. Such a phenomenon is mainly reflected in the following aspects. First, some warnings are from the methods with a single statement, especially for a getter method that often returns a class field. The contexts of these warnings fail to offer insights into the potential information that a class field might contain. There is a specific example in **Case 1** of Fig. 11. Second, some warnings are related to the interprocedural method calls. Thus, it could not be enough to only extract the source code from the method containing a warning as the warning context in our study. In particular, three wrongly identified warnings on *D2* are attributed to the second aspect. Third, some warnings fall into the methods with an interface type. The contexts of these warnings cannot be determined when the program is not executed. Fourth, some warnings involve the declaration or initialization of the class fields. Since such fields are often outside a method, the contexts of these warnings are usually unavailable in our study. **Case 2** of Figure 11 shows an illustrative example for the fourth aspect.

Just Accepted

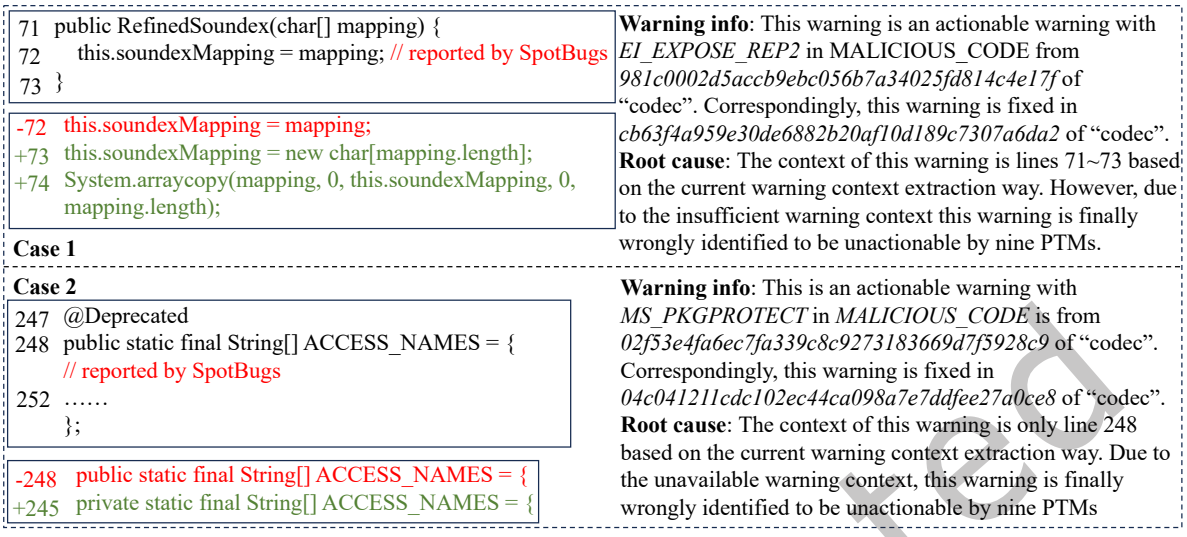


Fig. 11. Cases that are wrongly identified by PTMs.

(3) Imbalanced warning dataset. Five warnings fall into this reason. To avoid the accidental error during the experimental evaluation, five-fold cross-validation is adopted. That is, all warning dataset is split into five equal folds, where four-fold warnings are the training set and the remaining one-fold warnings are the test set. Our study fine-tunes PTMs on the training set and uses the fine-tuned PTMs for the test set. However, it is observed that different warning categories have extremely imbalanced distribution in the number of actionable and unactionable warnings. For instance, the DODGY_CODE category involves 3330 warnings, where 86 ones are actionable and 3244 ones are unactionable. In contrast, there are 76 unactionable warnings and only one actionable warning in the EXPERIMENTAL category. It indicates that even with five-fold cross-validation, there is always a lack of actionable warnings related to EXPERIMENTAL in the training or test sets. The above phenomenon may make PTMs learn a biased AWI classifier, which could not work well in the test set [35].

Answering RQ5: PTMs underperform on AWI when facing non-distinct or missing warning contexts and imbalanced warning dataset.

5 DISCUSSION

5.1 Practical Application

To ensure the generalization of findings in Section 4, we quantitatively conduct the extensive experimental evaluation on different SCAs (i.e., SpotBugs, Infer, CppCheck, and CSA) and programming languages (i.e., Java, C, and C++). To further discuss the practical application of findings in Section 4, we perform a rigorously qualitative analysis, i.e., discussing how the PTM-based AWI approach in our study is applied to other SCAs and projects with other programming languages.

Specifically, without modifying SCAs themselves, the PTM-based AWI approach focuses on postprocessing warnings reported by using these SCAs to scan projects with different programming languages. It indicates that the PTM-based AWI approach is technically independent of specific SCAs and projects with specific programming languages. In practice, to help developers locate bugs in projects with different programming languages, SCAs

Just Accepted

generally give the warning location. That is, regardless of any SCA and any programming language, as long as the warning location information (e.g., the warning code line) in the project under test is given by the SCA, the PTM-based AWI approach can be used to identify actionable and unactionable warnings. Such a technical trait can support the seamless extension of the PTM-based AWI approach to projects with various SCAs and programming languages. Therefore, we believe that our findings can be extended to other SCAs and projects with other programming languages.

5.2 Future Work

Based on our findings in Section 4, we highlight potential directions for the future AWI community.

Incorporating the refined warning context into PTM-based AWI. Sections 4.1~4.3 show that by simply extracting source code from the method containing a warning as the warning context, PTMs have already achieved a new breakthrough in the AWI field. However, the results of Section 4.5 indicate that the warning context extracted by our study is still coarse-grained, causing PTM's underperformance on AWI. In the future, it could be essential to extract the fine-grained warning contexts via the well-designed static analysis techniques (e.g., the def-use chain) [45] and the rigorous dynamic execution tactics (e.g., fuzzing) [28], thereby capturing the discriminative patterns between actionable and unactionable warnings for PTM-based AWI. Also, warnings reported by SCAs have various characteristics (e.g., category and message), which could provide hints for AWI. Thus, it could be promising to enrich the warning context with warning characteristics, thereby amplifying PTM-based AWI performance.

Enlarging the benefits of code-related pre-training and fine-tuning for PTM-based AWI. Section 4.3 shows that the code-related pre-training and fine-tuning components in PTMs benefit AWI. Such findings inspire researchers to explore more innovative AWI approaches from the following aspects. As for the code-related pre-training component, it is a naive way to apply various PTMs with more abundant and larger code-related pre-training corpora (e.g., CodeLlama [51]) for AWI. Besides, it is engaging to develop domain-specific PTMs by formulating pre-training tasks related to AWI. As for the fine-tuning component, Fig. 8 shows that the upward trends of precision, recall, and F1 have not even diminished when all warnings in the training set are included. It indicates that the effectiveness of PTM-based AWI could be further improved with the additional fine-tuning warning samples. In the future, it is necessary to enhance AWI effectiveness by fine-tuning PTMs on more warning datasets.

Eliminating the inappropriate warning dataset distribution for PTM-based AWI. As concluded in Section 4.4, due to the warning heterogeneity, the PTM-based AWI approach in the cross-project AWI scenario performs worse than that in the within-project AWI scenario. Also, increasing the number of training set brings little performance improvement in the cross-project AWI scenario. In addition, Section 4.5 states that the imbalanced warning training set negatively affects the PTM-based AWI effectiveness. In essence, the above phenomena are mainly attributed to the inappropriate warning dataset distribution [20]. Such a problem is ubiquitous in the software defect detection domain, and many techniques have been proposed and presented substantial improvements for software defect detection [8, 67]. Both AWI and software defect detection fundamentally focus on revealing true bugs in the software under test [19]. Inspired by the same problem with associated solutions in the existing software defect detection studies, there could be two potential research directions. One direction is to gather a sufficient number of warnings with diverse categories or incorporate the sampling strategy to select a representative warning dataset, thereby boosting the performance of applying PTMs for the cross-project AWI scenario from the data perspective. The other direction is to integrate the core ideas of transfer learning or ensemble learning, thereby enhancing the performance of applying PTMs for the cross-project AWI scenario from the model perspective.

Just Accepted

5.3 Threats to Validity

External. The external threat to validity is related to the generalizability of our findings. To alleviate this threat, we select nine representative PTMs for the experimental evaluation. Such PTMs not only cover the three typical PTM architectures, but also show powerful performance in recent code-related tasks [47]. Besides, we conduct extensive experiments on 12K+ warnings, which involve four commonly used SCAs and three representative programming languages. Besides, we qualitatively analyze the application of our findings to other SCAs and projects with other programming languages. We acknowledge, however, that there are slight differences among PTMs, SCAs, and projects with various programming languages. In future work, we will conduct more experiments with other PTMs, SCAs, and projects with various programming languages.

Internal. The internal threat to validity is related to the baseline selection in RQ1. We meticulously identify two DL-based AWI approaches as baselines by following the classification of ML-based AWI approaches in Section 2.2. Besides, based on the warning representation and model selection in existing studies [30, 34, 64], we rigorously implement the two baselines for the experimental evaluation. Thus, we believe that such a scrupulous experiment design can mitigate the above threat.

Construct. The construct threat to validity is related to the warning dataset. On the one hand, the warning dataset in our study is involved with four SCAs, 24 projects, and three mainstream programming languages. Specifically, 10K+ SpotBugs warnings from 10 Java projects originate from the work of Ge et al. [20]. Such a work labels warnings by combining the advantages of manual inspection and automatic filtering. 1.9K+ Infer, CppCheck, and CSA warnings from 14 C/C++ projects originate from the work of Wen et al. [60]. Such a work labels warnings via the rigorous manual inspection. It indicates that the warning dataset in our study is sufficiently reliable. On the other hand, there may be data leakage, i.e., the warning dataset in our study may overlap with the code-related pre-trained corpora of PTMs. In our study, we adopt K-fold cross-validation for the experimental evaluation, which can alleviate the threat of data leakage to a certain extent [4]. In particular, Lee et al. [37] experimentally prove that a certain degree of data leakage can boost the memory phenomenon of PTMs, thereby improving the generalization ability of PTMs. Thus, we believe that the construct threat has little impact on our study.

6 CONCLUSION

In this paper, we are the first to explore the feasibility of PTMs for AWI. By extensively evaluating 12K+ warnings, our results show that PTMs substantially improve AWI compared to the SOTA ML-based AWI approaches. Subsequently, we investigate the impact of different data preprocessing ways, model training components, and model prediction scenarios on the PTM-based AWI performance. Finally, we summarize three reasons for the underperformance of PTM-based AWI. Based on the experimental results, we provide several future research directions for the PTM-based AWI domain.

Just Accepted

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for insightful comments. The work is partly supported by National Natural Science Foundation of China (62502476, U24A20337), Fundamental Research Funds for the Central Universities at China University of Geosciences (Wuhan) (CUG250690), “CUG Scholar” Scientific Research Funds at China University of Geosciences (Wuhan) (2025039), and Natural Science Foundation of Jiangsu Province (BK20251458).

REFERENCES

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. ACL, Online, 2655–2668.

[2] Esben Sparre Andreasen, Anders Møller, and Benjamin Barslev Nielsen. 2017. Systematic approaches for increasing soundness and precision of static analyzers. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State of the Art in Program Analysis*. ACM, Barcelona, 31–36.
[3] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, et al. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609* (2023), 1–59.
[4] Emily M. Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. 2021. On the Dangers of Stochastic Parrots: Can Language Models Be Too Big?. In *Proceedings of the 4th ACM Conference on Fairness, Accountability, and Transparency (FAccT)*. ACM, Canada, 610–623.
[5] Yoshua Bengio and Yves Grandvalet. 2004. No Unbiased Estimator of the Variance of K-Fold Cross-Validation. *Journal Machine Learning Research (LMLR)* 5 (2004), 1089–1105.
[6] Pavol Bielik, Veselin Raychev, and Martin Vechev. 2017. Learning a static analyzer from data. In *Proceedings of the 29th International Conference of Computer Aided Verification (CAV)*. Springer, Switzerland, 233–253.
[7] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in Neural Information Processing Systems (NIPS)* 33 (2020), 1877–1901.
[8] Jinfu Chen, Jiaping Xu, Saihua Cai, Xiaoli Wang, Haibo Chen, and Zhehao Li. 2024. Software Defect Prediction Approach Based on a Diversity Ensemble Combined With Neural Network. *IEEE Transactions on Reliability (TR)* 73, 3 (2024), 1487–1501.
[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021), 1–35.
[10] Zimin Chen, Steve Kommrusch, and Martin Monperrus. 2022. Neural Transfer Learning for Repairing Security Vulnerabilities in C Code. *IEEE Transactions on Software Engineering (TSE)* 49, 1 (2022), 147–165.
[11] Brian Chess and Gary McGraw. 2004. Static analysis for security. *IEEE Symposium on Security and Privacy (S&P)* 2 (2004), 76–79.
[12] CppCheck. Lasted accessed May 18, 2025. https://cppcheck.sourceforge.io/..
[13] CSA. Lasted accessed May 18, 2025. https://clang-analyzer.llvm.org/..
[14] Jesse Dodge, Gabriel Ilharco, Roy Schwartz, Ali Farhadi, Hannaneh Hajishirzi, and Noah Smith. 2020. Fine-tuning pretrained language models: Weight initializations, data orders, and early stopping. *arXiv preprint arXiv:2002.06305* (2020), 1–11.
[15] Hugging Face. Lasted accessed February 18, 2024. https://huggingface.co/.
[16] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *Findings of the Association for Computational Linguistics (ACL)* 1, 1 (2020), 1536–1547.
[17] Xiuting Ge, Chunrong Fang, Tongtong Bai, Jia Liu, and Zhihong Zhao. 2023. An empirical study of class rebalancing methods for actionable warning identification. *IEEE Transactions on Reliability (TR)* 72, 4 (2023), 1–15.
[18] Xiuting Ge, Chunrong Fang, Xuanye Li, Weisong Sun, Daoyuan Wu, Juan Zhai, Shangwei Lin, Zhihong Zhao, Yang Liu, and Zhenyu Chen. 2023. Machine Learning for Actionable Warning Identification: A Comprehensive Survey. *arXiv preprint arXiv:2312.00324* 57, 2 (2023), 1–35.
[19] Xiuting Ge, Chunrong Fang, Xuanye Li, Quanjun Zhang, Jia Liu, Zhihong Zhao, and Zhenyu Chen. 2024. Improving actionable warning identification via the refined warning-inducing context representation. *Science China Information Sciences (SCIS)* 67, 5 (2024), 1–2.
[20] Xiuting Ge, Chunrong Fang, Xuanye Li, Qirui Zheng, Jia Liu, Zhihong Zhao, and Zhenyu Chen. 2024. A Large-Scale Empirical Study of Actionable Warning Distribution within Projects. *IEEE Transactions on Dependable and Secure Computing (TDSC)* (2024), 1–18.
[21] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (ACL)*. IEEE, Ireland, 1–14.
[22] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2021. Graphcodebert: Pre-training code representations with data flow. In *Proceedings of the 9th International Conference on Learning Representations (ICLR)*. IEEE, Austria, 1–18.
[23] Andrew Habib and Michael Pradel. 2018. How Many of All Bugs Do We Find? A Study of Static Bug Detectors. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. ACM/IEEE, France, 317–328.
[24] Xu Han, Zhengyan Zhang, Ning Ding, Yuxian Gu, Xiao Liu, Yuqi Huo, Jiezhong Qiu, Yuan Yao, Ao Zhang, Liang Zhang, et al. 2021. Pre-trained models: Past, present and future. *AI Open* 2 (2021), 225–250.
[25] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 33, 8 (2024), 1–79.
[26] Infer. Lasted accessed February 18, 2025. https://fbinfer.com/..
[27] JGit. Lasted accessed May 10, 2025. https://github.com/eclipse-jgit/jgit..

Just Accepted

[28] Ashwin Kallingal Joshy, Xueyuan Chen, Benjamin Steenhoek, and Wei Le. 2021. Validating Static Warnings via Testing Code Fragments. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, Denmark, 540–552.
[29] Hong Jin Kang, Khai Loong Aw, and David Lo. 2022. Detecting false alarms from automatic static analysis tools: how far are we?. In *Proceedings of the 44th IEEE/ACM International Conference on Software Engineering (ICSE)*. IEEE/ACM, Pittsburgh, 698–709.
[30] Anant Kharkar, Roshanak Zilouchian Moghaddam, Matthew Jin, Xiaoyu Liu, Xin Shi, Colin Clement, and Neel Sundaresan. 2022. Learning to reduce false positives in analytic bug Detectors. In *Proceedings of the 44th International Conference on Software Engineering (ICSE)*. IEEE/ACM, Pittsburgh, 1307–1316.
[31] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (S&P)*. IEEE, USA, 595–614.
[32] Diederik P Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representations (ICLR)*. IEEE, USA, 1–15.
[33] Ugur Koc, Parsa Saadatpanah, Jeffrey S. Foster, and Adam A. Porter. 2017. Learning a Classifier for False Positive Error Reports Emitted by Static Code Analysis Tools. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*. ACM, Barcelona, 35–42.
[34] Ugur Koc, Shiyi Wei, Jeffrey S. Foster, Marine Carpuat, and Adam A. Porter. 2019. An Empirical Assessment of Machine Learning Approaches for Triaging Reports of a Java Static Analysis Tool. In *Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, Xi'an, 288–299.
[35] Bartosz Krawczyk. 2016. Learning from imbalanced data: open challenges and future directions. *Progress in Artificial Intelligence* 5, 4 (2016), 221–232.
[36] Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu Hong Hoi. 2022. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. *Advances in Neural Information Processing Systems (NIPS)* 35 (2022), 21314–21328.
[37] Katherine Lee, Daphne Ippolito, Andrew Nystrom, Chiyuan Zhang, and Nicholas Carlini. 2021. Deduplicating Training Data Makes Language Models Better. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics (ACL)*. ACM, America, 8424–8445.
[38] Seongmin Lee, Shin Hong, Jungbae Yi, Taeksu Kim, Chul-Joo Kim, and Shin Yoo. 2019. Classifying False Positive Static Checker Alarms in Continuous Integration Using Convolutional Neural Networks. In *Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, Xi'an, 391–401.
[39] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2024. Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. In *Proceedings of 39th the ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Vol. 8. ACM, USA, Article 111, 26 pages.
[40] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 2022. An Empirical Study on the Effectiveness of Static C Code Analyzers for Vulnerability Detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, Kaist, 1–13.
[41] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Shin Yoo, and Yves Le Traon. 2021. Mining Fix Patterns for FindBugs Violations. *IEEE Transactions on Software Engineering (TSE)* 47, 1 (2021), 165–188.
[42] Yanli Liu, Yuan Gao, and Wotao Yin. 2020. An improved analysis of stochastic gradient descent with momentum. In *Proceedings of the 34th International Conference on Neural Information Processing Systems (NeurIPS)*. IEEE, Online, 11 pages.
[43] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664* (2021), 1–14.
[44] Tukaram Muske and Alexander Serebrenik. 2022. Survey of approaches for postprocessing of static analysis alarms. *ACM Computing Survey (CSUR)* 55, 3, Article 48 (2022), 39 pages.
[45] Tukaram Muske, Rohith Talluri, and Alexander Serebrenik. 2018. Repositioning of Static Analysis Alarms. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, Amsterdam, 187–197.
[46] Nadim Nachar et al. 2008. The Mann-Whitney U: A test for assessing whether two independent samples come from the same distribution. *Tutorials in Quantitative Methods for Psychology* 4, 1 (2008), 13–20.
[47] Changan Niu, Chuanyi Li, Vincent Ng, Dongxiao Chen, Jidong Ge, and Bin Luo. 2023. An Empirical Comparison of Pre-Trained Models of Source Code. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*. IEEE/ACM, Austria, 2136–2148.
[48] Replication package. Lasted accessed June 10, 2024. https://github.com/135790123/ptw4awi.
[49] PyTorch. Lasted accessed March 10, 2024. https://pytorch.org/.
[50] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Journal of Symbolic Logic* 74, 2 (1953), 358–366.
[51] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Defossez,

Just Accepted

Jade Copet, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. *arXiv:2308.12950* (2023), 1–48.
[52] SpotBugs. Lasted accessed March 10, 2024. https://spotbugs.github.io/.
[53] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. 2022. Efficient transformers: A survey. *Comput. Surveys* 55, 6 (2022), 1–28.
[54] Kien T. Tran and Hieu Dinh Vo. 2022. SCAR: smart contract alarm ranking. In *Proceedings of the 29th Asia-Pacific Software Engineering Conference (APSEC)*. ACM, Japan, 447–451.
[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NeurIPS)*, Vol. 30. IEEE, USA, 1–15.
[56] Junjie Wang, Yuchao Huang, Song Wang, and Qing Wang. 2022. Find bugs in static bug finders. In *Proceedings of the 30th International Conference on Program Comprehension (ICPC)*. IEEE/ACM, Online, 516–527.
[57] Junjie Wang, Song Wang, and Qing Wang. 2018. Is There a "Golden" Feature Set for Static Warning Identification? An Experimental Evaluation. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, Oulu, Article 17, 10 pages.
[58] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 26th International Conference on Empirical Methods in Natural Language Processing (EMNLP)*. IEEE, Abu Dhabi, 1–13.
[59] Mark Weiser. 1984. Program slicing. *IEEE Transactions on Software Engineering (TSE)* 30, 4 (1984), 352–357.
[60] Cheng Wen, Yuandao Cai, Bin Zhang, Jie Su, Zhiwu Xu, Dugang Liu, Shengchao Qin, Zhong Ming, and Tian Cong. 2024. Automatically Inspecting Thousands of Static Bug Warnings with Large Language Model: How Far Are We? *ACM Transactions Knowledge Discovery Data (TKDD)* 18, 7, Article 168 (2024), 34 pages.
[61] Ge Xiuting, Fang Chunrong, Liu Jia, Qing Mingshuang, Li Xuanye, and Zhao Zhihong. 2023. An unsupervised feature selection approach for actionable warning identification. *Expert Systems with Applications (ESWA)* 227 (2023), 120152.
[62] Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. 2022. A Systematic Evaluation of Large Language Models of Code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (MAPS)*. ACM, USA, 1–10.
[63] Rahul Yedida, Hong Jin Kang, Huy Tu, Xueqi Yang, David Lo, and Tim Menzies. 2023. How to Find Actionable Static Analysis Warnings: A Case Study With FindBugs. *IEEE Transactions on Software Engineering (TSE)* 49, 4 (2023), 1–17.
[64] Sai Yerramreddy, Austin Mordahl, Ugur Koc, Shiyi Wei, Jeffrey S Foster, Marine Carpuat, and Adam A Porter. 2023. An empirical assessment of machine learning approaches for triaging reports of static analysis tools. *Empirical Software Engineering (EMSE)* 28, 2 (2023), 28.
[65] Ping Yu, Yijian Wu, Xin Peng, Hahjia Peng, Jian Zhang, Peicheng Xie, and Wenyun Zhao. 2023. ViolationTracker: Building Precise Histories for Static Analysis Violations. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*. IEEE/ACM, Austria, 1–12.
[66] Huaien Zhang, Yu Pei, Junjie Chen, and Shin Hwei Tan. 2023. Statfier: Automated Testing of Static Analyzers via Semantic-Preserving Program Transformations. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, USA, 237–249.
[67] Yunhua Zhao, Kostadin Damevski, and Hui Chen. 2023. A Systematic Survey of Just-in-Time Software Defect Prediction. *Comput. Surveys* 55, 10 (2023), 1–35.

Just Accepted