The Journal of Systems and Software 245 (2027) 113150

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

LLMSec: Context-aware security bug report prediction via relevance-guided rebalancing and LLM-derived embeddings☆,☆☆

Abhinav Jamwal, Kunal Bansal, Shreya Jain, Sandeep Kumar *

Department of Computer Science and Engineering, Indian Institute of Technology Roorkee, Roorkee, 247667, Uttarakhand, India

ARTICLE INFO

Dataset link: https://github.com/ABhiIITRoorkee/LLMSec

Keywords:
Security bug report (SBR) prediction
LLM-derived embeddings
Software security triage
Imbalanced data handling

ABSTRACT

Security bug report prediction (SBRP) is important for software reliability because missed security issues can delay vulnerability fixes and cause operational failures or financial losses. In practice, separating security bug reports (SBRs) from non-security reports is difficult because SBRs are rare and bug reports contain mixed text, such as natural language, logs, code snippets, and configuration details. Therefore, methods based on shallow text features or artificial oversampling often fail to capture security-relevant context and may miss true security reports.

We present LLMSec, an LLM-based approach for automated security bug report identification that combines the relevance-guided report selection with contextual text representation. LLMSec first ranks non-security reports using the BM25F-based field-aware similarity scoring and retains low-overlap non-security reports, reducing class imbalance while preserving real negative examples without synthetic data generation. The resulting reports are encoded as dense features using pretrained LLM-based encoders that capture contextual information from bug-report summaries and descriptions. These features are then used with the standard machine learning classifiers. Experiments on five public security bug report datasets from Ambari, Camel, Derby, Wicket, and Chromium show that LLMSec improves security-report detection over existing methods while maintaining competitive false-alarm rates. These results suggest that combining BM25F-guided report filtering with LLM-derived contextual representations provides a reliable basis for identifying security-related reports across heterogeneous software projects.

1. Introduction

Software systems are increasingly operating in the security-critical environments, where hidden vulnerabilities can compromise trust, availability, and operational reliability. In an era where software is becoming ever more complex and extensive, development teams have to analyze vast numbers of issue tickets raised using collaborative bug-tracking systems like JIRA,[1] LaunchPad,[2] and Bugzilla.[3] Among these reports, only a small fraction correspond to Security Bug Reports (SBRs), while the majority describe routine maintenance issues or non-security defects. Failure to correctly identify security-related reports can delay vulnerability remediation, increase maintenance overhead, and expose systems to exploitable failures.

In the modern continuous integration and rapid-release environments, the accurate prioritization of security-relevant reports has therefore become an important requirement for the reliability and software quality assurance.

Automatically distinguishing SBRs from Non-Security Bug Reports (NSBRs) remains challenging in practice. Bug reports are typically written by reporters with different levels of technical and security expertise and often contain heterogeneous information sources, including natural language descriptions, stack traces, configuration fragments, code snippets, and framework-specific terminology. Consequently, security-relevant information is frequently embedded implicitly within long and structurally complex reports rather than expressed through explicit security-related keywords. Fig. 1 illustrates representative examples

☆ This article is part of a Special issue entitled: 'AI4MSS' published in The Journal of Systems & Software.
☆☆ Editor: Raffaela Mirandola.

* Corresponding author.
E-mail addresses: abhinav_j@cs.iitr.ac.in (A. Jamwal), kunal_bansal@cs.iitr.ac.in (K. Bansal), shreya_j@cs.iitr.ac.in (S. Jain), sandeep.garg@cs.iitr.ac.in (S. Kumar).

[1] https://www.atlassian.com/software/jira.
[2] https://bugs.launchpad.net/openstack.
[3] https://www.mozilla.org/en-US/about/governance/policies/security-group/bugs/.

https://doi.org/10.1016/j.jss.2026.113150
Received 7 June 2026; Received in revised form 31 August 2026; Accepted 8 October 2026
Available online 15 October 2026
0164-1212/© 2026 Published by Elsevier Inc.



Fig. 1. An SBR example from the `Camel` project. The orange text denotes the report summary, while the remaining content contains a technical description with embedded XML configuration code. The example illustrates why accurate SBR prediction requires contextual understanding beyond keyword matching.

from the Camel dataset, where semantically important security cues are distributed across technical descriptions and configuration artifacts.

To address this problem, previous research has explored automated Security Bug Report Prediction (SBRP) approaches using machine learning classifiers such as Logistic Regression, Random Forests, and Support Vector Machines (Jiang et al., 2020; Wu et al., 2021; Shu et al., 2021). Most existing approaches represent bug reports using sparse lexical features, including TF-IDF vectors, bag-of-words representations, or manually engineered keyword sets (Wu et al., 2021; Zheng et al., 2021). Although these methods can capture surface-level textual similarity, they often struggle to model contextual relationships across long bug descriptions and mixed-format technical content.

This limitation becomes especially critical when evidence relevant to security issues is scattered among several report fields or described in behavior-related terms implicitly, making reliance on lexical similarity insufficient for distinguishing between security reports and general maintenance reports.

The second significant challenge pertains to the highly skewed nature of security and non-security reports. SBRs account for less than 10% of all reported bugs in many actual repositories. The effect is that the classification systems will tend to favor NSBRs, thereby failing to detect genuine SBRs. Imbalance handling approaches including oversampling, clustering, and data synthesis (Wu et al., 2021; Zheng et al., 2021; Shu et al., 2021) could mitigate this issue to some degree. Nonetheless, these approaches risk altering the actual distribution of the reports and introducing artificial characteristics that hinder generalization to other software projects. Besides, existing ML workflows face difficulties when dealing with lengthy bug reports comprising multiple technical components (Li et al., 2021; Omri and Sinz, 2020).

From all of this, one can get a better understanding that achieving success in predicting SBRs requires semantic representations with high capability that are able to deal with varying kinds of information present in a report and manage the techniques so as to ensure realistic distribution of the reports without making use of any artificial methods whatsoever.

The developments seen in recent times regarding large language models (LLMs) and encoder-based language models can help achieve great success in terms of representation learning for lengthy and complex documents. Unlike traditional lexical representations and static embeddings, these models are able to capture long-range dependencies between elements present in a sequence. For this reason, they prove effective in SBR analysis.

To address these challenges, we introduce LLMSec, a framework for Security Bug Report Prediction that combines relevance-guided data rebalancing with contextual representations from pretrained language models. In LLMSec, the first step is to compute field-aware BM25F similarity scores between non-security reports and the known security reports. Based on these scores, low-similarity non-security reports are selected. This helps reduce the dominance of the majority class while keeping real negative examples instead of generating synthetic ones. The selected reports are then encoded using pretrained LLM-based encoders, where contextual embeddings are generated from the summary and description of each bug report. These embeddings are then passed to supervised classifiers to identify the reports that are likely to be security-relevant.

Unlike the previous works that rely mainly on heuristics such as the keyword filtering or rely only on the use of sparse lexical features or static word embeddings, our framework LLMSec integrates relevance-guided instance selection and contextual representations provided by the pretrained language model. In this way, LLMSec is able to decrease class imbalance while retaining the contextual information that can be missed by sparse features or static word embeddings.

To systematically evaluate the effectiveness and robustness of the proposed framework, we investigate the following research questions:

RQ1: How does LLMSec perform compared with existing security bug report prediction methods?
RQ2: How sensitive is LLMSec to preprocessing and tokenization choices?
RQ3: How do different LLM-based encoders affect LLMSec's classification performance?
RQ4: How robust is LLMSec under class imbalance across different projects?

This paper makes the following contributions:

- We propose LLMSec, a hybrid framework for Security Bug Report Prediction that combines BM25F-guided relevance-based rebalancing with LLM-derived contextual embeddings.
- We introduce a BM25F-guided instance-selection strategy that mitigates class imbalance by selecting low-overlap non-security reports and avoiding reliance on synthetic oversampling techniques.

- We perform a comprehensive empirical evaluation across multiple LLM-based encoders and preprocessing strategies to analyze how representation learning choices influence SBR classification reliability.
- We evaluate LLMSec on five real-world datasets and statistically examine its performance relative to established baselines using project-level non-parametric testing.

The remainder of this paper is organized as follows. Section 2 reviews existing research on Security Bug Report Prediction. Section 3 presents the motivation for the proposed work. Section 4 presents the proposed LLMSec framework. Section 5 describes the experimental setup and evaluation results. Section 6 discusses the implications and limitations of the findings. Finally, Section 7 concludes the paper and outlines future research directions.

2. Related work

The Security Bug Report Prediction (SBRP) task focuses on differentiating Security Bug Reports (SBRs) from Non-Security Bug Reports (NSBRs). The challenge in this task is that there is no explicit indication regarding whether the bugs in question have anything to do with security. Rather, security evidence could be implicit, project-dependent, and scattered over various segments in the report such as natural language descriptions, stack traces, configurations, and code mentions. In addition, SBRs constitute a small percentage of all the bug reports making up the dataset used. Due to these reasons, past studies on SBRP have mainly focused on two aspects, text representation and class imbalance.

The early studies relied on hand-crafted features as well as security keywords. Gegick et al. (2010) suggested one of the earliest statistical models for finding incorrectly classified SBRs based on security keywords' patterns. This kind of approach proved that the automatic classification of SBRs was possible, however, it heavily relied on manually selected terms. This means that these methods were very sensitive to vocabulary peculiarities and thus not robust to changes in security terminologies. In order to overcome this drawback, the subsequent studies leveraged automatically generated textual representations like Bag-of-Words (BoW), term frequency features, and TF-IDF. For example, Wijayasekara et al. (2012) and Goseva-Popstojanova and Tyo proposed a feature-vector representation for bug report classification by using text mining approaches to train classifiers for identifying SBRs. Although these models improved scalability, they still could not fully address word-association problems that are important for SBRP. Security reports may carry a different meaning from regular bug reports even when they share similar vocabulary.

The efforts to improve the lexical features with more effective models of classification have been made in several studies. Zaman et al. (2011) has investigated the distinctive features of both security and performance-related bugs and found that there are specific ways in which such types of bugs are reported and fixed. Xia et al. (2016) explored the problem of SBRs classification from the perspective of text-based learning techniques. The mentioned works managed to demonstrate SBRP relevance in software engineering, yet the features used were still mostly lexical which leads to misclassification in case when the security information was expressed differently or NSBR contained security-specific words.

Table 1 summarizes the methodological characteristics of the baseline methods reproduced under the common experimental protocol used in this study. Other relevant SBR prediction approaches are discussed in the surrounding text according to their methodological contributions.

Another primary research direction involves the issue of class imbalance. Due to the low number of SBRs within the training set, classifiers tend to become biased toward the dominant NSBR class in bug-report collections. Peters et al. (2017) introduced FARSEC that performs a preprocessing step, whereby all NSBRs with keyword similarity to SBRs are filtered out to reduce skewness. However, the FARSEC framework relies on the use of keyword-based TF–IDF lexical similarity and assumes that SBRs and NSBRs can be distinguished by keywords. Wu et al. (2021) explored the impact of data quality and learner optimization on security bug report prediction. Despite the importance of training data choice in improving SBRP performance, such approaches are still representationally constrained due to the scattered semantic security relevance of SBRs.

Other imbalance-aware studies have used the sampling or synthetic data generation. Shu et al. (2019) proposed SMOTUNED, which tunes SMOTE-based oversampling to improve defect prediction under class imbalance. Although tuned oversampling can improve minority-class detection, synthetic instances may change the original distribution of security reports, especially when the minority class is very small or semantically diverse. This concern is particularly relevant for SBRP because security reports often contain project-specific technical evidence that is difficult to synthesize reliably. Therefore, while sampling-based methods help address imbalance, they do not directly solve the semantic ambiguity of bug-report text.

To better capture report semantics, Jiang et al. (2020) proposed LTRWES, which combines BM25F-based ranking with word-embedding representations. Instead of relying only on keyword overlap, LTRWES selects NSBRs using BM25F-based relevance ranking and uses embeddings to represent report text. This marked an important step toward semantic SBRP because it connected imbalance handling with representation learning. However, LTRWES relies on static word embeddings such as CBOW/word2vec, which assign the same representation to a word regardless of context. Static embeddings, therefore, have limited ability to model subtle security cues, long-range dependencies, and mixed-format report content.

Ma et al. (2022) proposed CASMS, which combines clustering-based imbalance handling with an attention-based neural prediction model. CASMS first represents bug reports using TF–IDF-weighted word2vec embeddings and then applies the Elbow method and k-means clustering to divide the NSBR class into different groups. NSBRs located near the cluster centroids are retained to preserve representative samples from different regions of the majority-class distribution. The selected NSBRs and all available SBRs are subsequently used to train an Attention CNN–BLSTM model that captures local patterns, sequential dependencies, and the relative importance of report terms. CASMS therefore differs from relevance-based approaches that primarily retain NSBRs according to their dissimilarity from SBRs. In comparison, LLMSec uses BM25F-guided ranking to select low-overlap NSBRs and represents reports using frozen pretrained contextual encoders, while only the downstream classifier is trained for SBR prediction.

More recently, neural and LLM-based text representations have been explored for SBRP and related software engineering tasks. FASTTEXT (Alqahtani, 2024) uses fastText-based word and subword representations for security bug report classification. Compared with TF–IDF, fastText can better capture subword patterns and short textual regularities, but it still provides relatively shallow contextual modeling. LLM-based encoder methods, such as BERT-based classifiers, provide contextual embeddings and can model token interactions more effectively. BERT_{WPP} (Soltaniani et al., 2025), for example, applies an encoder-based language model to security bug report prediction. Such approaches improve semantic modeling, but standard BERT is constrained by limited context length and does not explicitly address class imbalance through relevance-guided instance selection. These limitations motivate approaches that combine stronger contextual representations with imbalance-aware training-data construction.

Beyond methods designed specifically for SBR prediction, prior research has also investigated pretrained representations that are purpose-built for general bug-report analysis. Fang et al. (2023) proposed RepresentThemAll, a universal bug-report representation framework pretrained on a large bug-report corpus using a dynamic masked

Table 1
Methodological comparison of the SBR prediction methods included in the controlled empirical evaluation.

Feature	FARSEC (Peters et al., 2017)	FARSEC + Tuned Learner/Preprocessor (Wu et al., 2021)	SMOTUNED (Shu et al., 2019)	LTRWES (Jiang et al., 2020)	FASTTEXT (Alqahtani, 2024)	$BERT_{WPP}$ (Soltaniani et al., 2025)	LLMSec
Text representation	TF–IDF/security keywords	TF–IDF with FARSEC-style features	TF–IDF with tuned sampling	BM25F + CBOW	fastText word/subword embeddings	BERT contextual embeddings	LLM-derived embeddings + BM25F
Semantic depth	Low	Low	Low	Moderate	Moderate	High	High
Imbalance handling	Filters misleading NSBRs	FARSEC-style filtering/evaluation	Tuned SMOTE oversampling	Relevance-based NSBR selection	Training-set under-sampling	No explicit sampling in basic WPP	BM25F-guided relevance-based rebalancing
Feature engineering	Manual security keyword matrix	FARSEC-style text features	Tuned preprocessing and sampling	Ranking + embedding features	Tokenization, stop-word removal, n-grams	Tokenization, attention masks, fine-tuning	Contextual LLM-derived features
Adaptability to context	No	Limited	Limited	Partial	Partial	Yes	Yes
Long-sequence support	No	No	No	No	No	Limited by standard BERT	Yes, if long-context encoder is used
Classifier used	ML classifiers, e.g., LR, NB, RF	Tuned ML learner/ preprocessor	RF with tuned SMOTE	Gradient boosting/ranking model	fastText supervised classifier	Fine-tuned BERT classifier	Hybrid ML, e.g., LR, SVM, RF
Filtering strategy	Keyword-overlap filtering	FARSEC/FARSEC-style filtering	Oversampling	BM25F ranking	No keyword filtering	No filtering in basic WPP	BM25F-guided low-overlap filtering
Use of embeddings	No	No	No	Yes, word2vec/ CBOW	Yes, fastText embeddings	Yes, BERT embeddings	Yes, LLM-derived embeddings
Noise robustness	Low	Medium	Medium	High	Medium	Medium to high	High

language modeling objective and a contrastive learning objective termed "find yourself". The pretrained model is subsequently adapted through supervised task-specific fine-tuning for duplicate bug report detection, bug report summarization, bug priority prediction, and bug severity prediction. By pretraining directly on bug-report text, RepresentThemAll learns domain-specific contextual representations that capture the vocabulary, structure, and semantic characteristics of bug reports. This work is relevant to the embedding-generation phase of LLMSec, since both approaches aim to derive dense representations from bug-report textual fields. However, RepresentThemAll focuses on learning a reusable bug-report-specific encoder for multiple downstream tasks, whereas LLMSec uses frozen, off-the-shelf pretrained encoders and investigates their integration with relevance-guided NSBR selection for highly imbalanced SBR prediction.

Recent LLM-based encoder and embedding models like those used in BERT (Devlin et al., 2019), Longformer (Beltagy et al., 2020), and instruction-sensitive embeddings like E5 (Wang et al., 2022) offer great promise in terms of capturing long and meaningful text data. The aforementioned models can prove valuable for bug reports that incorporate a mixture of natural language text, logs, configuration information, and snippets of code. Nonetheless, relying on these encoding mechanisms alone cannot guarantee success for SBRP tasks. Specifically, if there is no balance handling, chances are that the model will capture decision boundaries predominantly influenced by the majority class of non-security bugs reports (NSBR). In the same vein, lack of contextual information will ensure that the chosen filtering method focuses on lexical similarities rather than security-related semantics. This indicates that effective SBRP requires both semantic representation and imbalance-aware instance selection.

The methodological differences among the baseline approaches reproduced in the controlled evaluation are presented in Table 1. The comparison shows that earlier methods usually address either class imbalance or textual representation, but rarely combine both within a unified process. FARSEC (Peters et al., 2017) and its tuned variants (Wu et al., 2021) focus on filtering and learner configuration while remaining dependent on lexical representations. SMOTUNED (Shu et al., 2019) addresses class imbalance through oversampling but does not strengthen the semantic understanding of reports. LTRWES (Jiang et al., 2020) combines BM25F ranking with word embeddings, although its static embeddings limit contextual expressiveness. FASTTEXT (Alqahtani, 2024) and $BERT_{WPP}$ (Soltaniani et al., 2025) use neural representations, but they do not explicitly combine contextual representations with relevance-guided rebalancing. The table is restricted to the baseline methods reproduced under the common datasets, partitions, preprocessing settings, and repeated-run protocol used in this study; it is therefore not intended to provide an exhaustive catalogue of SBR prediction research. CASMS (Ma et al., 2022) contributes a distinct clustering and attention-based strategy and is examined in the related-work discussion, while RepresentThemAll (Fang et al., 2023) is positioned as a general bug-report representation framework evaluated on different downstream tasks.

Building on these complementary research directions, LLMSec combines BM25F-guided relevance-based rebalancing with contextual representations extracted from frozen, off-the-shelf pretrained encoders. The BM25F component selects real NSBRs with low semantic overlap and reduces the dominance of the majority class, while the contextual embeddings preserve security-relevant evidence that may be implicit or distributed across different textual fields. Unlike CASMS, LLMSec does not train an Attention CNN–BLSTM model or select NSBRs according to cluster membership. Unlike RepresentThemAll, it does not pretrain a new bug-report-specific encoder or perform end-to-end task-specific fine-tuning of the encoder. Its contribution instead lies in examining how frozen contextual representations interact with relevance-guided training-set construction for highly imbalanced SBR prediction. Consequently, clustering-based majority-class sampling, bug-report-specific representation pretraining, and the relevance-guided rebalancing mechanism introduced in LLMSec represent complementary approaches to improving automated security bug report prediction.

3. Motivation

Security Bug Report Prediction (SBRP) aims to identify security-relevant bug reports at an early stage, so that development teams can

prioritize vulnerabilities before they remain unresolved or become exploitable. Although much progress has been made in SBRP (Peters et al., 2017; Jiang et al., 2020; Shu et al., 2019; Wu et al., 2021), accurate prediction is still difficult in practice for two main reasons. To begin with, Security Bug Reports (SBRs) are fewer than Non-Security Bug Reports (NSBRs), which creates class imbalance and makes classifiers lean toward the majority class. In addition, the textual basis for classifying a bug report as security-related may not always appear through direct security keyword references. As a result, methods that mainly depend on keyword matching, TF-IDF, or shallow lexical similarity often fail to capture the contextual meaning needed to separate true SBRs from routine maintenance reports.

Fig. 1 illustrates this challenge using an example from the `Camel` project. The report is labeled as an improvement request, and its summary refers to a transaction-manager configuration issue in `camel-jms`. However, the detailed description includes Spring XML configuration code, connection-factory settings, and transaction-management behavior that require domain-specific interpretation. A purely lexical model may focus on isolated terms such as "configuration", "component", or "transaction", without understanding how these elements together indicate security-relevant behavior. The above example clearly indicates why SBRP cannot be viewed as just another keyword spotting problem. Effective prediction is achieved by models which take into account contextual dependencies in heterogeneous reports along with identifying subtle semantic cues in multiple fields.

Class imbalance makes this task harder. Since SBRs make up only a small part of bug reports, a classifier may still appear to perform well by mostly predicting the NSBR class. However, this behavior is risky in security triage because missed SBRs can delay vulnerability remediation. Existing methods handle this issue through keyword filtering, oversampling, or relevance-based selection (Peters et al., 2017; Shu et al., 2019; Jiang et al., 2020). However, these strategies either depend on shallow lexical overlap or create synthetic samples that may not reflect the meaning of real security reports. Therefore, SBRP needs not only stronger text representation but also an imbalance-handling strategy that keeps real negative examples while reducing majority-class dominance.

The recent improvements in LLM-based encoders and embeddings provide an opportunity to overcome the semantic limitations of previous SBRP approaches. In contrast to sparse lexical representations or fixed word embeddings, such models allow generating contextual representations that capture the use of terms within a particular document. This becomes relevant for the bug reporting context, in which security semantics might be affected by the relationship among natural language, technical artifacts, and specific project terminology. Nonetheless, contextual embeddings do not solve the problem of class imbalance. The imbalance will affect the model training regardless of its encoding capacity, as long as the majority of documents are NSBRs.

Motivated by these observations, we present our method called LLMSec, an approach that combines BM25F-guided relevance-based rebalancing with LLM-derived contextual embeddings. To be more precise, the rebalancing component uses field-aware BM25F similarity to identify real NSBRs with low overlap, which helps reduce majority-class dominance without depending on synthetic oversampling. The embedding component uses LLM-based encoders to represent bug reports and capture long-range contextual dependencies. By addressing semantic ambiguity and class imbalance together, the proposed approach provides a reliable basis for identifying security bug reports across heterogeneous software projects.

4. Proposed approach

In this section, we propose LLMSec, a framework for detecting Security Bug Reports (SBRs) from bug report collections that suffer from class imbalance. The goal of the proposed approach is to support more accurate security bug triage by reducing undetected SBRs while remaining practical for real-world use. The framework incorporates BM25F-based sampling of non-SBRs together with contextual representations produced by LLM-based encoders and standard classification algorithms.

4.1. Method overview

LLMSec performs security bug report prediction through four sequential steps: preprocessing bug reports, selecting representative non-security bug reports, converting reports into numerical representations, and training a classifier to distinguish security-related reports from non-security ones. An overview of this workflow is shown in Fig. 2. The complete process is summarized in Algorithm 1, while the procedure used to generate report representations is described in Algorithm 2. To support reproducibility, we have publicly released the implementation of LLMSec together with the experimental datasets.[4] The replication package includes the preprocessing scripts, dataset-specific configurations, experimental settings, repeated-run scripts, ablation-study configurations, computational-cost benchmarking scripts, and result-processing code required to reproduce the experimental results reported in Tables 3–13.

In the first phase, LLMSec preprocesses bug reports to construct the textual representations required by the subsequent stages. After basic cleaning of non-informative artifacts, two representations of the summary and description fields are maintained. For BM25F-based similarity computation, the text is processed according to the selected `tokenization_way`. Under NLTK-based preprocessing, tokenization is performed using `nltk.word_tokenize`; under model-aligned BM25F preprocessing, tokenization is performed using the tokenizer associated with the selected pretrained encoder. In both settings, stop-word removal and stemming are subsequently applied to construct the lexical representation used by BM25F. Separately, cleaned versions of the summary and description fields are preserved without the BM25F-specific tokenization, stop-word removal, or stemming operations. These preserved textual fields are later concatenated and independently tokenized using the native tokenizer of the selected pretrained encoder for contextual embedding generation.

In the **second phase**, LLMSec addresses the severe class imbalance between SBRs and non-security bug reports (NSBRs) by selecting representative NSBRs for training. Each NSBR is compared with known SBRs using field-aware BM25F similarity scores based on unigrams and bigrams. For every NSBR, an average similarity score is calculated across all SBRs. NSBRs with the lowest similarity scores are retained as low-overlap real negative examples without generating synthetic data. This selection strategy reduces the dominance of the non-security class while preserving representative reports for downstream classification.

In the **third phase**, the summary and description of each bug report are combined and converted into a fixed-length numerical representation. Tokenization is performed using pre-trained tokenizers corresponding to the selected LLM-based encoders, such as BERT, Longformer, E5-Instruct-Large, or Jina-v3. The tokenized text is then passed through the encoder to obtain a contextual representation of the report. Each report is represented by a single vector, such as the one linked to a special classification token or a pooled sentence embedding. These vectors capture context across the entire report and act as input features for the classification step.

In the **final phase**, LLMSec trains a standard machine learning classifier using the generated report representations. Models such as Logistic Regression, Random Forest, Support Vector Machines (SVM), or Multi-Layer Perceptrons (MLP) are trained on embeddings derived from all SBRs and the selected NSBRs. Once trained, the classifier is used on new bug reports to decide whether they are security-related (SBR = 1) or non-security-related (NSBR = 0). With this setup, LLMSec

[4] https://github.com/ABhiIITRoorkee/LLMSec.

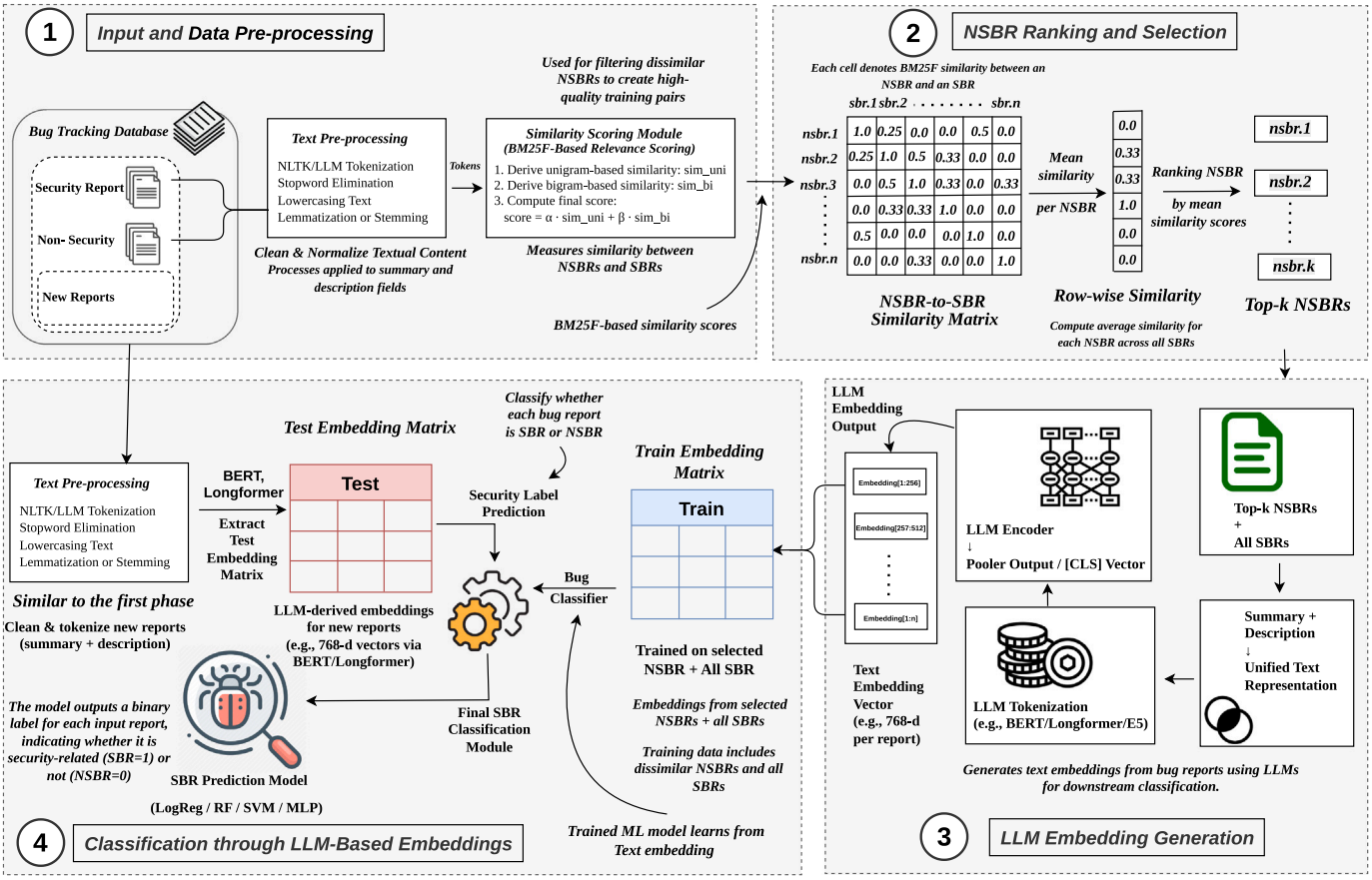


Fig. 2. General process flow of LLMSec, an LLM-based method for security bug report prediction, comprising four phases: data preprocessing, BM25F-guided NSBR filtering, LLM-based embedding generation, and SBR classification.

Algorithm 1 End-to-End Pipeline of LLMSec for SBR Classification

Require: Set of bug reports $R = \{r_1, r_2, \ldots, r_n\}$
Ensure: Predicted labels Y
1: $R_{\text{clean}} \leftarrow \text{CleanReports}(R)$ {Normalize and clean report text}
2: $S \leftarrow \text{ComputeBM25FSimilarity}(R_{\text{clean}})$ {Compute field-aware similarity between NSBRs and SBRs}
3: $D \leftarrow \text{SelectNSBRs}(S, R_{\text{clean}})$ {Select low-overlap non-security reports}
4: $E \leftarrow \text{GenerateReportVectors}(D)$ {Convert reports into fixed-length numerical vectors}
5: $Y \leftarrow \text{TrainClassifierAndPredict}(E)$ {Train classifier and predict SBR/NSBR labels}
6: **return** Y

Algorithm 2 Generating Report Representations

Require: Selected bug reports $D = \{d_1, d_2, \ldots, d_m\}$
Ensure: Report vectors $E = \{e_1, e_2, \ldots, e_m\}$
1: $E \leftarrow \emptyset$
2: **for** each report $d_i \in D$ **do**
3: $T_i \leftarrow \text{ModelTokenizer}(d_i)$ {Prepare input for the selected LLM-based encoder}
4: $H_i \leftarrow \text{EncodeText}(T_i)$ {Obtain contextual representation}
5: $e_i \leftarrow \text{AggregateRepresentation}(H_i)$ {Derive a single vector for the report}
6: $E \leftarrow E \cup \{e_i\}$
7: **end for**
8: **return** E

can use rich and context-aware representations while still keeping the clarity and efficiency of traditional classifiers.

In general, LLMSec provides a concise and scalable solution for security bug report prediction. The method relies on filtering non-security bug reports using BM25F and generating context-based representations of bug reports using LLMs, thus mitigating issues related to class imbalance and context. The proposed approach can easily be tailored for various projects by utilizing different languages since it employs a modular architecture.

4.2. Phase 1: Data preprocessing and similarity calculation

In the first phase, the proposed approach constructs the lexical representation used for BM25F-based similarity calculation. Non-informative elements, such as hyperlinks, hexadecimal values, and duplicated stack-trace lines, are first removed while security-relevant technical terms are retained. The resulting text is then processed according to the selected `tokenization_way`. When NLTK-based preprocessing is used, the text is tokenized with `nltk.word_tokenize`; when model-aligned preprocessing is used, the `tokenize()` method of the tokenizer associated with the selected pretrained encoder is applied. Stop words are subsequently removed and stemming is applied in both settings to obtain standardized lexical tokens from the summary and description fields. These processed tokens are used exclusively for BM25F-based report-report similarity computation and relevance-guided NSBR selection; they are not supplied directly to the pretrained encoder.

In parallel, cleaned versions of the summary and description fields are preserved for contextual representation learning. These fields do not undergo the BM25F-specific tokenization, stop-word removal, or stemming operations. After the NSBR-selection stage, the preserved cleaned text is passed independently to the native tokenizer of the selected pretrained encoder, as described in Phase 3.

For each repeated run r, a new stratified train–test split is generated using the corresponding random seed. All BM25F-specific preprocessing and BM25F-related calculations are then performed independently using only the training partition, denoted by $D^{(r)}_{\text{train}}$. The SBR reference set $\mathcal{S}^{(r)}$ and the candidate NSBR pool $\mathcal{N}^{(r)}$ are reconstructed from the training partition of that run. Reports in the corresponding test partition are excluded from similarity calculation, ranking, and selection.

To compare reports, LLMSec uses a BM25F-based multi-field scoring function over the summary and description fields. This score estimates how much a non-security bug report nsbr overlaps with a security bug report sbr in terms of important terms and phrases. The total similarity is defined as a weighted combination of unigram and bigram components:

$$\begin{aligned}\text{sim}(sbr, nsbr) = {} & w_1 \cdot \text{sim}_{\text{unigram}}(sbr, nsbr) \\ & + w_2 \cdot \text{sim}_{\text{bigram}}(sbr, nsbr)\end{aligned} \tag{1}$$

where w_1 and w_2 control the contribution of the unigram and bigram components.

Unigram similarity. Each term t appearing in both reports contributes through a field-normalized BM25F weight. For each field $f \in \{\text{summary}, \text{description}\}$, the normalized term frequency is:

$$w_f(d,t) = \frac{(k_1+1)\,\text{tf}_f(d,t)}{k_1\left((1-b_f) + b_f \cdot \frac{|d_f|}{\text{avglen}_f}\right) + \text{tf}_f(d,t)} \tag{2}$$

The combined field contribution for term t is:

$$w(d,t) = \sum_{f \in F} \alpha_f \, w_f(d,t) \tag{3}$$

We adopt the standard Robertson–Zaragoza IDF:

$$IDF(t) = \ln\left(\frac{N - n_t + 0.5}{n_t + 0.5}\right) \tag{4}$$

For every run r, the corpus-dependent quantities N, n_t, and avglen_f are calculated exclusively from $D^{(r)}_{\text{train}}$ and are recomputed after generating the run-specific train–test split. Consequently, no BM25F corpus statistics obtained from the test partition or from another repetition are reused.

The unigram similarity between the two reports is therefore:

$$\text{sim}_{\text{unigram}}(sbr, nsbr) = \sum_{t \in sbr \cap nsbr} IDF(t)\, w(sbr,t)\, w(nsbr,t) \tag{5}$$

Bigram similarity. The same BM25F formulation is extended to adjacent word pairs. For each bigram (t_i, t_{i+1}) appearing in both reports, we compute:

$$\text{sim}_{\text{bigram}}(t_i t_{i+1}, sbr, nsbr) = \text{BM25F}(t_i t_{i+1}, sbr, nsbr) \tag{6}$$

The total bigram similarity is the sum over all shared bigrams.

Ranking NSBRs. For each run r, LLMSec computes $\text{sim}(nsbr_j, sbr_i)$ for every candidate $nsbr_j \in \mathcal{N}^{(r)}$ against every $sbr_i \in \mathcal{S}^{(r)}$. The complete NSBR-to-SBR similarity matrix is therefore regenerated independently for each train–test split. These scores are then summarized by an average similarity for each NSBR. The end-to-end pipeline that uses this score is shown in Algorithm 1, while the embedding procedure in Algorithm 2 is used later in Phase 3. The resulting mapping assigns each NSBR a mean similarity score:

$$\text{similarity}^{(r)}(nsbr_j) = \frac{1}{|\mathcal{S}^{(r)}|} \sum_{i=1}^{|\mathcal{S}^{(r)}|} \text{sim}(nsbr_j, sbr_i) \tag{7}$$

where $\mathcal{S}^{(r)}$ denotes the set of SBRs contained only in the training partition of run r. The resulting similarity scores are passed to Phase 2 to generate a new NSBR ranking and selected NSBR subset for that run. Thus, neither the SBR reference set nor the BM25F similarity scores are fixed once at the dataset level or reused across the 30 repetitions.

4.3. Phase 2: NSBR filtering and selection

In the second phase, LLMSec addresses class imbalance by selecting representative non-security bug reports (NSBRs) for training. Instead of randomly discarding NSBRs, we use the BM25F-based similarity scores computed in Phase 1 to identify NSBRs with low textual overlap with known security bug reports (SBRs). For every repeated run r, this phase operates exclusively on the run-specific training SBR set $\mathcal{S}^{(r)}$ and candidate training NSBR pool $\mathcal{N}^{(r)}$. The aim of this step is to reduce the dominance of routine non-security reports while retaining real negative examples that support reliable downstream classification.

For each candidate NSBR in run r, we summarize its similarity to the corresponding training SBR set by computing the average similarity score:

$$\text{similarity}^{(r)}(nsbr_j) = \frac{1}{|\mathcal{S}^{(r)}|} \sum_{i=1}^{|\mathcal{S}^{(r)}|} \text{sim}(nsbr_j, sbr_i) \tag{8}$$

where $nsbr_j \in \mathcal{N}^{(r)}$, $\mathcal{S}^{(r)}$ denotes only the SBRs in the training partition of run r, and $\text{sim}(nsbr_j, sbr_i)$ represents the BM25F-based report–report similarity defined in Phase 1. Neither test-partition SBRs nor test-partition NSBRs are used in this calculation. This average score measures how closely a candidate training NSBR overlaps with the security-related reports available in the same training partition.

For each run, all candidate NSBRs in $\mathcal{N}^{(r)}$ are ranked using the newly computed $\text{similarity}^{(r)}$ values. Consequently, the ranking is regenerated after every train–test split and is not fixed once at the dataset level or reused across the 30 repetitions. Lower scores indicate lower textual overlap with the corresponding training SBR set. LLMSec supports two selection strategies for choosing the k NSBRs:

- **Min-Selector:** selects the k NSBRs with the lowest run-specific similarity scores, ensuring that low-overlap non-security reports are included in the training set.
- **RWS-Selector (Roulette Wheel Selection):** assigns a run-specific selection probability to each NSBR that is inversely related to its similarity score and samples from this distribution. This strategy preserves diversity among the selected NSBRs while still favoring reports with lower overlap with the training SBR set.

For the RWS strategy, the weight assigned to each NSBR in run r is computed as:

$$\text{RWS}^{(r)}(nsbr_j) = \text{MaxScore}^{(r)} - \text{similarity}^{(r)}(nsbr_j) \tag{9}$$

where

$$\text{MaxScore}^{(r)} = \max_{nsbr_j \in \mathcal{N}^{(r)}} \text{similarity}^{(r)}(nsbr_j) \tag{10}$$

ensures that every weight is non-negative before normalization into a probability distribution.

By selecting NSBRs with low BM25F similarity to the run-specific training SBR set, this phase reduces majority-class dominance without relying on synthetic oversampling. The selected NSBR subset is combined only with the SBRs in $\mathcal{S}^{(r)}$ to construct the training set for run r. Thus, each repetition has its own SBR reference set, NSBR pool, similarity ranking, and selected NSBR subset, while the corresponding test partition remains untouched until final evaluation. The complete procedure is summarized in Algorithm 1.

4.4. Phase 3: LLM-based embedding generation

In this phase, LLMSec converts bug reports into numerical representations that preserve their meaning and context. The selected non-security bug reports (NSBRs) and security bug reports (SBRs) are encoded using pretrained LLM-based encoders and embedding models. These representations are then used as input features for the classification step in Phase 4. To capture a complete view of each report, the preserved cleaned summary and description fields are concatenated into a single text sequence before encoding.

Importantly, the textual input used in this phase is maintained independently of the lexical representation constructed for BM25F-based semantic rebalancing in Phase 1. The NLTK-based or model-aligned preprocessing selected through `tokenization_way`, together with the associated stop-word removal and stemming operations, is used only to construct the lexical representation consumed by BM25F. These BM25F-processed tokens are not supplied directly to the pretrained encoder. Instead, the cleaned summary and description fields preserved prior to BM25F-specific preprocessing are concatenated and independently tokenized using the native tokenizer associated with the selected pretrained encoder.

Let $D = \{d_1, d_2, \ldots, d_n\}$ denote the set of bug reports to be represented, where each d_i denotes the concatenated cleaned summary and description of a single report preserved independently of the BM25F preprocessing branch. The contextual representation of each report is computed as follows:

$$e_i = \text{LLMEncoder}(\text{Tokenizer}(d_i)) \tag{11}$$

Here, the `Tokenizer` denotes the native tokenizer associated with the selected pretrained encoder and prepares the cleaned report text according to the model-specific input requirements. The `LLMEncoder` then generates a contextual representation using a pretrained encoder or embedding model, such as BERT, Longformer, E5-Instruct, or Jina-v3. Thus, even when NLTK-based preprocessing is selected for the BM25F stage, neural embedding generation still applies the native tokenizer of the corresponding pretrained encoder to the preserved cleaned report text. For each report, a single vector is obtained using the pooling strategy associated with the corresponding encoder implementation, such as the `[CLS]` token representation or mean pooling over token embeddings.

For uniform processing, reports exceeding the maximum input length supported by the selected encoder are truncated according to the corresponding model limit, such as 512 tokens for BERT and E5-Instruct, 4096 tokens for Longformer, and 8192 tokens for Jina-v3. Reports shorter than the maximum input length are padded according to the tokenizer and batching requirements of the corresponding encoder. All pretrained encoders are used as frozen feature extractors throughout the experiments. Their parameters are not updated using the SBR and NSBR labels, and no task-specific fine-tuning or additional bug-report-domain pretraining is performed. Only the downstream machine-learning classifiers introduced in Phase 4 are trained using the generated representations and the corresponding class labels. This frozen-encoder protocol enables a controlled comparison of the representation quality provided by the evaluated off-the-shelf encoders and distinguishes LLMSec from bug-report-specific representation-learning approaches that pretrain or fine-tune the encoder for downstream tasks.

Thus, each bug report is converted into a fixed-length dense vector, such as a 768-dimensional vector for BERT-base or a 1024-dimensional vector for Jina-v3. Together, these vectors form the embedding matrix:

$$E = \{e_1, e_2, \ldots, e_n\}, \qquad e_i \in \mathbb{R}^d \tag{12}$$

where d denotes the embedding dimensionality of the selected pretrained encoder. This matrix is used as input to the classifier in Phase 4 and provides contextual representations of the textual information contained in the bug reports.

The embedding-generation procedure is summarized in Algorithm 2. By separating BM25F-specific lexical preprocessing from native encoder tokenization, LLMSec allows the relevance-guided rebalancing stage and the contextual-representation stage to operate using representations appropriate to their respective purposes.

4.5. Phase 4: Classification through LLM-based embeddings

In the final phase, LLMSec assigns each bug report to either the security or non-security class using the LLM-derived contextual representations generated in Phase 3. Each report, represented as a fixed-length vector from its combined summary and description, is used as input to a supervised classification model. The goal of this phase is to support reliable classification while reducing missed security reports.

We denote the embedding matrix as follows:

$$X = \{\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_n\} \in \mathbb{R}^{n \times d} \tag{13}$$

where n is the number of bug reports and d is the dimension of the embedding space, such as $d = 768$ for BERT-base. Each vector $\vec{x}_i$ represents the LLM-derived contextual embedding of one bug report.

These vectors are then given as input to a standard machine learning classifier, such as Logistic Regression (LR), Random Forest (RF), Support Vector Machine (SVM), or Multi-Layer Perceptron (MLP). For each report, the classifier predicts whether it is security-related as follows:

$$\hat{y}_i = f(\vec{x}_i; \theta) \tag{14}$$

where f denotes the classification function and θ represents the learned model parameters. All models are trained with binary supervision on the training set constructed in Phase 2. Cross-entropy loss is used for LR and MLP, hinge loss is used for SVM, and RF predictions are calibrated through class probabilities. This training setup helps improve sensitivity to the minority SBR class while maintaining stable overall performance.

To improve robustness and reduce overfitting, classifier hyperparameters are tuned using stratified cross-validation. When required, the embedding vectors are normalized before training models such as LR or SVM, while tree-based models such as RF operate directly on the raw embeddings. In addition, by separating the representation learning from the classification, the proposed approach can evaluate different classifier models without changing the basic architecture of the LLM-based encoders. The accuracy of a model can be evaluated based on its precision, recall, F1 measure, and G-measure. These measures are particularly essential in the case of security triage that emphasizes reliability, as failing to recognize a valid report may have dire consequences. A schematic diagram illustrating the entire process of classification is presented in Algorithm 1. The embedding creation procedure is illustrated in Algorithm 2. By incorporating report representations generated by LLMs into proven classification models, this stage enables the proposed approach to generate reliable predictions.

5. Experimental evaluation

The following section outlines the datasets, metrics, statistics, and experimental conditions employed to evaluate the proposed approach. The evaluation is organized around four research questions, including how effective the proposed approach is compared with existing security bug report prediction methods, how sensitive it is to preprocessing and tokenization choices, how different transformer-based encoders affect performance, and how robust it remains under severe class imbalance across projects.

Table 2
Overview of the security bug report datasets used in this study based on the corrected labels provided by Wu et al. (2021).

Project	#SBR	#NSBR	Total	SBR ratio
Wicket	47	953	1,000	4.70%
Ambari	56	944	1,000	5.60%
Camel	74	926	1,000	7.40%
Derby	179	821	1,000	17.90%
Chromium	808	41,132	41,940	1.93%

5.1. Benchmark datasets

The evaluation of the proposed approach is carried out using five publicly available datasets for security bug report identification, specifically the *Wicket*, *Ambari*, *Camel*, *Derby*, and *Chromium* datasets originally released by Ohira et al. (2015). For all experiments, we use the corrected and relabeled versions of these datasets provided by Wu et al. (2021), rather than the original label assignments. The corrected labels are applied consistently to LLMSec and all baseline methods under the same experimental protocol. Each report is classified as either a Security Bug Report (SBR) or a Non-Security Bug Report (NSBR).

These datasets are suitable for evaluating SBR prediction because they differ in project domain, vocabulary, report structure, and class distribution. They also contain highly imbalanced labels, where SBRs form only a small portion of all bug reports. This imbalance reflects the practical difficulty of security triage in real issue-tracking systems. Each report includes textual fields such as the summary and description, which may contain natural language, stack traces, configuration details, error messages, and code-related fragments. These characteristics make the datasets useful for examining whether contextual representations and semantic rebalancing improve SBR identification.

As shown in Table 2, all datasets have substantial class imbalance, with SBR ratios ranging from 1.93% in Chromium to 17.90% in Derby. Chromium is the largest and most imbalanced dataset, with 41,940 reports but only 808 SBRs. The other four datasets contain 1000 reports each, with SBR ratios below 8% for Wicket, Ambari, and Camel. The class counts and SBR ratios reported in Table 2 correspond to the corrected labels provided by Wu et al. (2021). For every dataset, the SBR ratio is calculated using the total number of reports as the denominator, that is, $|\mathcal{S}|/(|\mathcal{S}|+|\mathcal{N}|)\times 100$, where $|\mathcal{S}|$ and $|\mathcal{N}|$ denote the numbers of SBRs and NSBRs, respectively. Accordingly, the Chromium ratio is calculated as $808/(808+41,132)=808/41,940=1.93\%$, rather than 808/41,132. This convention is applied consistently across all datasets, and the same set-cardinality notation is used throughout the method description. These characteristics reflect the practical difficulty of SBR prediction, where security-related reports are rare and easily dominated by the majority NSBR class.

5.2. Performance evaluation measures

Security bug report prediction is an imbalanced binary classification problem. In this setting, missing a true security report may delay vulnerability remediation, whereas incorrectly flagging too many non-security reports may increase developer triage effort. Therefore, we evaluate each method using three commonly used measures: probability of detection (p_d), probability of false alarm (p_f), and g-measure (Peters et al., 2017; Shu et al., 2021).

The probability of detection measures the proportion of actual SBRs correctly identified by the model:

$$pd = \frac{TP}{TP+FN}, \tag{15}$$

where TP denotes the number of SBRs correctly classified as security-related, and FN denotes the number of SBRs incorrectly classified as non-security reports. A higher p_d indicates that fewer security reports are missed.

The probability of false alarm measures the proportion of NSBRs incorrectly classified as SBRs:

$$pf = \frac{FP}{FP+TN}, \tag{16}$$

where FP denotes the number of NSBRs incorrectly classified as security-related, and TN denotes the number of NSBRs correctly classified as non-security reports. A lower p_f indicates fewer unnecessary security alerts.

To summarize the trade-off between detecting true SBRs and controlling false alarms, we use g-measure. It is defined as the harmonic mean of p_d and specificity $(1-pf)$:

$$g\text{-measure} = \frac{2\times pd\times(1-pf)}{pd+(1-pf)}. \tag{17}$$

The g-measure is especially useful under class imbalance because it rewards methods that simultaneously achieve high detection and low false-alarm rates.

5.3. Statistical testing

Statistical inference is conducted at the project level using g-measure. For each method and project, the g-measure is averaged over 30 separately generated stratified train–test executions, yielding one project-level value for each of the five projects. The repeated executions characterize variation due to data partitioning but are not treated as independent observations. The p_d, p_f, and specificity measures are retained only for descriptive reporting because g-measure is derived from p_d and specificity.

We apply a permutation-based Friedman test (Rainio et al., 2024) to the five project-level blocks and report Kendall's W as the effect-size measure. The Friedman p-value is obtained from the permutation distribution rather than the asymptotic chi-square approximation because only five project-level blocks are available. Pairwise comparisons between LLMSec and each baseline use exact two-sided Wilcoxon signed-rank tests (Wilcoxon, 1992), with Benjamini–Hochberg adjustment (Thissen et al., 2002) across the six comparisons and matched-pairs rank-biserial correlation as the effect-size measure. This design treats software project as the unit of inference and avoids inflating the effective sample size through repeated partitions of the same dataset.

5.4. Experimental settings

5.4.1. Validation settings

For each dataset, we evaluate the proposed method using stratified training and testing partitions so that the original SBR/NSBR distribution is preserved in both partitions. All data-dependent operations are performed only on the training partition. These operations include BM25F-based NSBR ranking, semantic rebalancing, embedding normalization, classifier training, and hyperparameter tuning. The test partition is used only for final evaluation. This strict separation prevents information from the test set from entering the training process.

Each experiment is repeated 30 times using different random seeds. For every repetition, the corresponding seed generates a new stratified train–test split. The SBR reference set and candidate NSBR pool are then reconstructed exclusively from that run's training partition. BM25F corpus statistics, NSBR-to-SBR similarity scores, NSBR rankings, and the selected NSBR subset are recomputed independently for each run. Therefore, no SBR reference set, similarity ranking, or selected NSBR subset is fixed at the dataset level or reused across repetitions.

For the controlled training-label-noise analysis, the corrected labels are retained as the clean reference condition. Within each run, label corruption is applied only to the training partition, while the test reports and their labels remain unchanged. SBR→NSBR and NSBR→SBR corruption are evaluated separately at noise levels of 5%, 10%, and 20%. The corrupted training labels are introduced before reconstructing

the SBR reference set and candidate NSBR pool, computing the BM25F rankings, selecting the rebalanced training instances, and fitting the downstream classifier. The same train–test partitions and random seeds are retained across the clean and noisy conditions to support matched comparisons.

The results are aggregated across the 30 runs. Consequently, the reported variation reflects the combined effects of the run-specific data split, BM25F-guided rebalancing, sampling procedure, and classifier initialization. For the label-noise analysis, it additionally reflects the controlled corruption introduced into the training labels, while the corresponding test partition remains untouched until final evaluation.

5.4.2. Compared methods

We compare the proposed approach with six representative baseline methods, including FARSEC (Peters et al., 2017), FARSEC$^{Tuned}_{Learner}$ (Wu et al., 2021), SMOTUNED (Shu et al., 2019), LTRWES (Jiang et al., 2020), FASTTEXT (Alqahtani, 2024), and BERT$_{WPP}$ (Soltaniani et al., 2025). These methods represent different families of SBR prediction techniques, including keyword-based filtering, tuned learning, synthetic oversampling, relevance-based report selection, shallow embedding-based classification, and transformer-based classification.

The baselines are selected because they cover the main methodological directions used in prior SBR prediction studies. FARSEC relies on keyword-based filtering and learning; FARSEC$^{Tuned}_{Learner}$ improves the learner configuration and training process; SMOTUNED addresses class imbalance using optimized oversampling; LTRWES combines BM25F-based ranking with word embeddings; FASTTEXT provides a shallow neural embedding-based baseline; and BERT$_{WPP}$ represents a transformer-based baseline for security bug report prediction. When official implementations or reported configurations are available, we follow them directly. Otherwise, we reproduce the methods according to their descriptions in the original studies.

5.4.3. Preprocessing and embedding settings

We evaluate the method under two text-processing settings. The first setting uses conventional preprocessing based on NLTK, including tokenization, lowercasing, stopword removal, and lemmatization or stemming. The second setting uses tokenization aligned with the selected transformer encoder. This comparison allows us to examine whether model-aware tokenization affects the quality and stability of SBR prediction.

For embedding generation, we evaluate six pretrained transformer-based encoders: *bert-base-uncased*, *bert-large-uncased*, *longformer-base-4096*, *longformer-large-4096*, *multilingual -e5-large-instruct*, and *jina-embeddings-v3*. These encoders differ in model size, context length, and representation dimensionality. For each bug report, the summary and description fields are concatenated into a single textual sequence before embedding extraction. Reports that exceed the maximum input length of the selected encoder are truncated according to the model limit, whereas shorter reports are padded when required. The pretrained encoders are used as frozen feature extractors, and the resulting embeddings are supplied to downstream classifiers.

5.4.4. Classifier and parameter settings

The BM25F parameters were fixed across all projects, experimental configurations, and repetitions. The unigram and bigram similarity weights in Eq. (1) were set to $w_1 = 0.9$ and $w_2 = 0.2$, respectively. For both unigram and bigram scoring, the term-frequency saturation parameter was $k_1 = 2.0$ and the query-term saturation parameter was $k_3 = 0.5$. The field-length normalization parameters were $b_{\text{summary}} = 0.5$ and $b_{\text{description}} = 1.0$, while the document-field weights were $\alpha_{\text{summary}} = 3.0$ and $\alpha_{\text{description}} = 1.0$. The unigram query-field weights were 2.98 for the summary and 0.287 for the description, whereas the bigram query-field weights were 2.999 and 0.994, respectively. Table 3 summarizes the complete configuration used to generate the reported results.

The generated embeddings are used to train standard supervised classifiers, including Logistic Regression (LR), Random Forest (RF), Support Vector Machine (SVM), and Multi-Layer Perceptron (MLP). These classifiers are selected because they are widely used in software defect prediction and provide different learning behaviors. LR and SVM work well with dense embedding spaces when the decision boundary is approximately linear, RF captures non-linear feature interactions, and MLP provides a neural classifier over contextual representations.

Classifier hyperparameters are tuned using only the training partition. Embedding vectors are normalized when required by the classifier, especially for LR and SVM. Tree-based classifiers such as RF are trained directly on the generated embeddings. The final trained model is then evaluated on the held-out test partition.

A separate sensitivity analysis over the BM25F parameter space was not conducted; therefore, the effect of alternative weighting, saturation, and field-normalization settings remains a direction for future evaluation.

5.4.5. Implementation and reproducibility

The LLMSec system is implemented in Python using PyTorch for representation modeling, together with NumPy, SciPy, and scikit-learn for numerical computation and traditional machine-learning algorithms. The original predictive-performance experiments were conducted on an Ubuntu-based workstation with 16 GB of host RAM and an NVIDIA Quadro T1000 GPU with 4 GB of dedicated GPU memory (VRAM).

The pretrained encoders were used as frozen feature extractors, and no task-specific fine-tuning or backward propagation was performed. Embedding extraction for all evaluated pretrained encoders was performed on the NVIDIA Quadro T1000 GPU using a batch size of one, with forward passes executed within `torch.no_grad()`. No CPU offloading was used. All encoder inference was performed in FP32 precision; FP16, BF16, and mixed-precision inference were not used. The maximum input lengths were set to 512 tokens for the BERT and multilingual E5 encoders, 4096 tokens for the Longformer encoders, and 8192 tokens for *jina-embeddings-v3*. Reports exceeding the corresponding model-specific input limit were truncated before encoding. Gradient checkpointing was not used because the encoders were employed only for inference and no backward pass was required.

The hardware-dependent computational-cost analysis reported separately in Section 5.10 was performed on a workstation running Ubuntu 24.04.4 LTS with 503 GiB of host memory and an NVIDIA RTX PRO 5000 Blackwell GPU with 48,935 MiB of dedicated GPU memory. The software environment used PyTorch 2.8.0 with CUDA 12.8. Because execution time and memory consumption depend on the underlying hardware, the computational measurements reported in Section 5.10 are specific to this benchmark environment and are not intended to characterize the computational cost of all predictive-performance configurations.

To preserve the complete experimental workflow, embeddings were regenerated for every predictive-performance configuration and repetition rather than cached or reused across runs. Thus, the reported predictive-performance results reflect independently executed embedding and classification pipelines under the corresponding experimental settings. The computational measurements in Section 5.10 correspond only to the fixed *jina-embeddings-v3* benchmark configuration specified in that section and are not intended to represent the computational cost of every encoder configuration evaluated in the predictive-performance experiments.

For purposes of reproducibility, we apply consistent random-seed initialization across repeated runs while maintaining the same experimental protocol for all datasets and compared approaches. The source code, preprocessed datasets, parameter configurations, software-environment specifications, and scripts necessary to reproduce the experimental results are freely provided within the replication package.

Table 3
BM25F parameter configuration used in the experiments.

Parameter group	Parameter	Unigram	Bigram
Similarity weight	w	0.9	0.2
Term-frequency saturation	k_1	2.0	2.0
Query-term saturation	k_3	0.5	0.5
Field-length normalization	b_{summary}	0.5	0.5
	$b_{\text{description}}$	1.0	1.0
Document-field weight	α_{summary}	3.0	3.0
	$\alpha_{\text{description}}$	1.0	1.0
Query-field weight	Summary	2.98	2.999
	Description	0.287	0.994

5.5. RQ1: How does LLMSec perform compared with existing security bug report prediction methods?

To answer RQ1, we compare LLMSec with six representative security bug report prediction baselines, namely FARSEC (Peters et al., 2017), FARSEC$^{Tuned}_{Learner}$ (Wu et al., 2021), SMOTUNED (Shu et al., 2019), LTRWES (Jiang et al., 2020), FASTTEXT (Alqahtani, 2024), and BERT$_{WPP}$ (Soltaniani et al., 2025). These baselines represent the main methodological directions in SBR prediction, including keyword-based filtering, learner tuning, synthetic oversampling, relevance-based instance selection, neural text embeddings, and contextual language-model-based classification.

This comparison allows us to assess the effectiveness of combining BM25F-guided rebalancing with LLM-derived embeddings against representative SBR prediction approaches.

The compared methods are summarized as follows:

- **FARSEC** (Peters et al., 2017): identifies security-related bug reports using keyword importance scores and reduces class imbalance by filtering non-security reports before training.
- **FARSEC$^{Tuned}_{Learner}$** (Wu et al., 2021): denotes a FARSEC-based learner-tuning setting evaluated under the common experimental protocol, using the corrected-data setting and baseline configuration motivated by Wu et al.
- **SMOTUNED** (Shu et al., 2019): addresses class imbalance by generating synthetic minority-class examples through optimized oversampling.
- **LTRWES** (Jiang et al., 2020): combines BM25F-based relevance ranking with word embeddings for NSBR selection and report representation.
- **FASTTEXT** (Alqahtani, 2024): uses the fastText text-classification model to distinguish security bug reports from non-security bug reports using preprocessed summaries and descriptions.
- **BERT$_{WPP}$** (Soltaniani et al., 2025): fine-tunes a transformer-based sequence classification model for security bug report prediction in within-project prediction settings.

Table 4 shows that LLMSec achieves the strongest aggregate performance across the five evaluated projects, with the highest average g-measure (87.28) and p_d (87.65), while maintaining an average p_f of 12.76. FASTTEXT and LTRWES are the strongest baselines in terms of average g-measure, with 79.60 and 77.40, respectively. Accordingly, LLMSec improves the average g-measure by 7.68 percentage points over FASTTEXT and 9.88 points over LTRWES. Although BERT$_{WPP}$ obtains the lowest average p_f (1.60), its substantially lower g-measure (47.20) and p_d (33.60) indicate that low false-alarm rates alone do not imply stronger overall detection performance.

The project-level results also show that the advantage of LLMSec is not uniform. It achieves the highest g-measure on Wicket (89.15), Camel (83.72), and Derby (87.04), whereas LTRWES performs better on Ambari (88.26 versus 83.81) and FASTTEXT is marginally higher on Chromium (93.00 versus 92.68). On Chromium, however, LLMSec produces a lower p_f (4.86 versus 7.97). These exceptions may reflect project characteristics: Ambari contains only 56 SBRs, which may favor recurring project-specific lexical patterns captured by LTRWES, whereas Chromium's substantially larger corpus may provide FASTTEXT with sufficient data to learn stable word- and subword-level patterns. These explanations are descriptive rather than causal and indicate that a globally fixed LLMSec configuration may not be optimal for every project.

The project-level statistical analysis provides complementary evidence. The permutation-based Friedman test yields $Q_F = 24.77$, $p = 8.22 \times 10^{-7}$, with Kendall's $W = 0.826$, and LLMSec obtains the best average rank of 1.40. In the pairwise comparisons, LLMSec has higher g-measure values than FARSEC, FARSEC$^{Tuned}_{Learner}$, SMOTUNED, and BERT$_{WPP}$ on all five projects and higher values than LTRWES and FASTTEXT on four of five projects. The BH-adjusted exact Wilcoxon p-values range from 0.0938 to 0.1250; therefore, given the five available project-level observations, the pairwise results are interpreted primarily through their direction, magnitude, and effect sizes rather than through pairwise statistical significance.

Overall, the descriptive and statistical results consistently place LLMSec among the strongest methods across the evaluated benchmark, while also revealing meaningful project-specific differences. The conclusions are therefore restricted to the five evaluated projects, and broader cross-project generalization requires validation on additional independent software projects.

Answer to RQ1: LLMSec achieves the strongest aggregate performance, with the highest average g-measure (87.28), highest average p_d (87.65), and best average project-level rank (1.40). The project-level Friedman analysis shows a clear overall difference among the evaluated methods, while pairwise performance favors LLMSec over four baselines on all five projects and over LTRWES and FASTTEXT on four of five projects. Overall, LLMSec provides strong performance across the evaluated benchmark, with project-specific exceptions on Ambari and Chromium.

5.6. RQ2: How sensitive is LLMSec to preprocessing and tokenization choices?

To answer RQ2, we evaluate how different text preprocessing strategies influence the performance of LLMSec. Since bug reports contain mixed textual content, including natural language descriptions, configuration fragments, stack traces, and code-related terms, the preprocessing strategy can affect the lexical representation used for BM25F-based similarity computation and relevance-guided report selection. We therefore compare two preprocessing settings. The evaluated preprocessing choice applies to the lexical representation used in the BM25F-based rebalancing stage; contextual embedding generation subsequently uses the native tokenizer associated with the selected pretrained encoder, as described in Phase 3.

Table 4
Comparison of g-measure, p_d, and p_f for FARSEC (Peters et al., 2017), FARSEC$^{Tuned}_{Learner}$ (Wu et al., 2021), SMOTUNED (Shu et al., 2019), LTRWES (Jiang et al., 2020), FASTTEXT (Alqahtani, 2024), BERT$_{WPP}$ (Soltaniani et al., 2025), and LLMSec. For g-measure and p_d, higher values are better; for p_f, lower values are better.

Project	Metric	FARSEC	FARSEC-TL	SMOTUNED	LTRWES	FASTTEXT	BERT-WPP	LLMSec
Wicket	g-measure	65.00	67.00	75.85	75.39	77.00	23.00	**89.15**
	p_d	66.70	52.00	66.70	73.91	81.00	13.00	**91.67**
	p_f	36.60	8.00	12.10	23.06	26.62	**1.00**	13.24
Ambari	g-measure	71.90	70.00	71.64	**88.26**	81.00	40.00	83.81
	p_d	57.10	56.00	57.10	**87.50**	79.00	25.00	85.71
	p_f	3.00	6.00	3.90	10.95	16.89	**2.00**	18.01
Camel	g-measure	53.80	38.00	64.43	65.42	67.00	33.00	**83.72**
	p_d	50.00	24.00	55.60	71.73	71.00	20.00	**78.38**
	p_f	41.80	5.00	23.40	39.86	36.57	**1.00**	10.15
Derby	g-measure	61.70	62.00	69.60	71.26	80.00	57.00	**87.04**
	p_d	47.60	69.00	57.10	70.10	79.00	40.00	**92.13**
	p_f	12.40	44.00	10.90	27.54	18.97	**4.00**	17.52
Chromium	g-measure	65.40	80.00	80.53	86.67	**93.00**	83.00	92.68
	p_d	49.60	67.00	74.80	93.04	**94.00**	70.00	90.34
	p_f	3.80	**0.00**	12.80	18.88	7.97	**0.00**	4.86
Average	g-measure	63.56	63.40	72.41	77.40	79.60	47.20	**87.28**
	p_d	54.20	53.60	62.26	79.26	80.80	33.60	**87.65**
	p_f	19.52	12.60	12.62	24.06	21.40	**1.60**	12.76

Table 5
Project-level statistical comparison of LLMSec and baseline methods using g-measure.

Test/Comparison	Mean paired $\Delta g \pm$ SD	Test statistic/Effect size	Raw p	BH-adjusted p
Friedman	–	$Q_F = 24.77,\ W = 0.826$	8.22×10^{-7}	–
Pairwise exact Wilcoxon signed-rank tests				
LLMSec vs. FARSEC	$+23.72 \pm 6.95$	$T = 0,\ r_{rb} = 1.000$	0.0625	0.0938
LLMSec vs. FARSEC$^{Tuned}_{Learner}$	$+23.88 \pm 13.31$	$T = 0,\ r_{rb} = 1.000$	0.0625	0.0938
LLMSec vs. SMOTUNED	$+14.87 \pm 3.29$	$T = 0,\ r_{rb} = 1.000$	0.0625	0.0938
LLMSec vs. LTRWES	$+9.88 \pm 9.23$	$T = 1,\ r_{rb} = 0.867$	0.1250	0.1250
LLMSec vs. FASTTEXT	$+7.68 \pm 6.89$	$T = 1,\ r_{rb} = 0.867$	0.1250	0.1250
LLMSec vs. BERT$_{WPP}$	$+40.08 \pm 21.40$	$T = 0,\ r_{rb} = 1.000$	0.0625	0.0938

Table 6
Performance of LLMSec with NLTK-based tokenization using different encoders across five datasets.

Encoder	Metric	Wicket	Ambari	Camel	Derby	Chromium	Average
bert-base-uncased	g-measure	77.35	68.09	73.58	69.78	82.89	74.34
	pd	73.91	71.43	78.38	64.04	82.92	74.14
	pf	18.87	34.96	30.67	23.36	17.13	24.99
bert-large-uncased	g-measure	72.80	62.37	68.57	67.08	78.90	69.94
	pd	65.22	57.14	70.27	70.00	79.20	68.36
	pf	17.61	31.36	33.05	35.61	21.39	27.80
longformer-base-4096	g-measure	79.48	72.14	71.52	73.68	88.36	77.03
	pd	73.91	75.00	64.86	74.44	86.38	74.92
	pf	14.05	30.51	20.30	27.07	9.55	20.29
longformer-large-4096	g-measure	82.33	65.99	73.63	71.92	85.39	75.85
	pd	87.50	64.29	70.27	64.04	83.91	74.00
	pf	22.27	32.20	22.68	18.00	13.07	21.64
multilingual-e5-large-instruct	g-measure	86.77	79.29	76.71	84.38	92.68	83.96
	pd	86.96	75.00	70.27	79.78	90.84	80.56
	pf	13.42	15.89	15.55	10.46	5.38	12.14
jina-embeddings-v3	g-measure	85.32	83.81	83.72	87.04	92.68	86.51
	pd	83.33	85.71	78.38	92.13	90.34	85.98
	pf	12.61	18.01	10.15	17.52	4.86	12.62

- **NLTK-based tokenization:** The bug reports are processed using conventional text preprocessing steps, including tokenization, lowercasing, stopword removal, and lemmatization or stemming. The setting follows standard text normalization practices and gives a consistent word-level representation before BM25F-based similarity computation.
- **Model-aligned tokenization:** Bug reports are tokenized using the tokenizer of the selected LLM-based encoder. This setting keeps preprocessing aligned with the internal tokenization scheme used by the embedding model.

The goal of this analysis is to determine whether LLMSec is sensitive to preprocessing choices and whether model-aligned tokenization consistently improves classification performance. Tables 6 and 7 report the results for NLTK-based and model-aligned tokenization, respectively. We evaluate both settings across five datasets using g-measure, probability of detection (p_d), and probability of false alarm (p_f).

Figs. 3 and 4 visualize the same comparison across metrics and encoders. Fig. 3 shows metric-wise changes under the two tokenization settings, while Fig. 4 summarizes the average performance of each encoder.

Table 7
Performance of LLMSec with model-aligned tokenization using different encoders across five datasets.

Encoder	Metric	Wicket	Ambari	Camel	Derby	Chromium	Average
bert-base-uncased	g-measure	77.22	72.28	74.02	70.03	82.61	75.23
	pd	70.83	67.86	70.27	65.17	80.69	70.96
	pf	15.13	22.67	21.81	24.33	15.37	19.86
bert-large-uncased	g-measure	72.73	62.33	69.31	66.77	79.10	70.04
	pd	75.00	67.86	75.68	71.11	78.96	73.72
	pf	29.41	42.37	36.07	37.07	20.74	33.13
longformer-base-4096	g-measure	80.99	66.55	69.91	73.59	88.38	75.88
	pd	83.33	71.43	70.27	67.78	86.63	75.88
	pf	21.22	37.71	30.45	19.51	9.79	23.73
longformer-large-4096	g-measure	82.76	70.87	70.80	70.80	86.09	76.26
	pd	91.30	67.86	75.68	71.11	84.15	78.02
	pf	24.32	25.85	33.48	29.51	11.87	25.00
multilingual-e5-large-instruct	g-measure	86.57	80.95	76.28	85.76	92.28	84.36
	pd	78.26	78.57	75.68	86.67	89.60	81.75
	pf	3.14	16.53	23.11	15.12	4.86	12.55
jina-embeddings-v3	g-measure	89.15	82.64	78.59	86.17	92.29	85.76
	pd	91.67	89.29	81.08	82.02	90.09	86.83
	pf	13.24	23.09	23.76	9.25	5.40	14.94

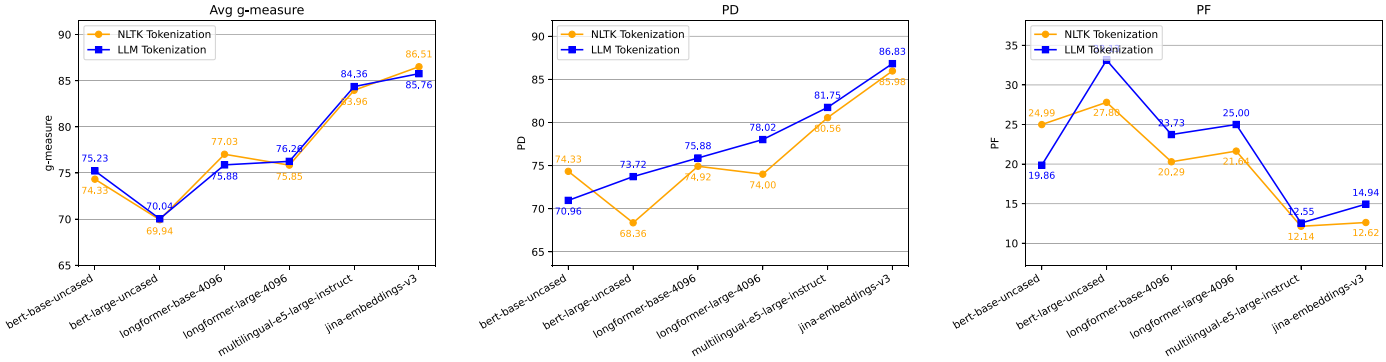


Fig. 3. The effect of NLTK-based and model-aligned tokenization on g-measure, p_d, and p_f across encoders.

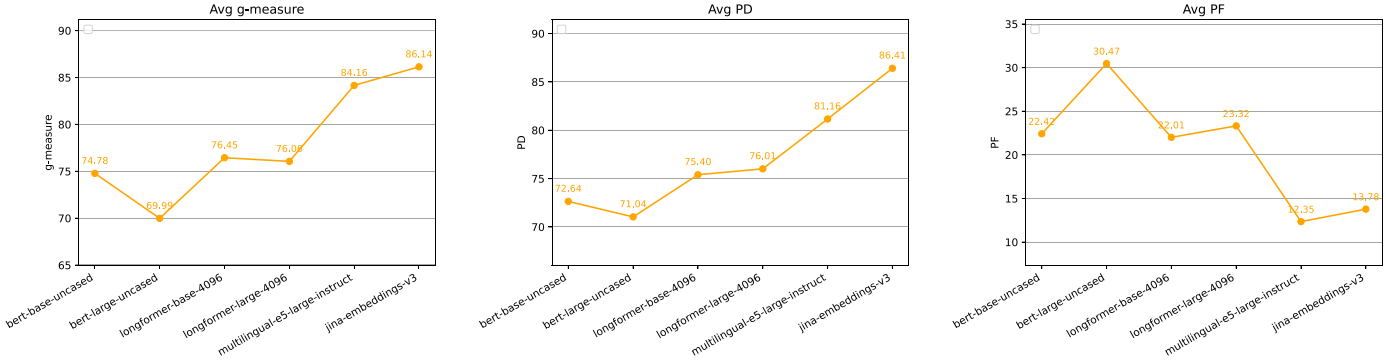


Fig. 4. The average g-measure, p_d, and p_f for each encoder under the evaluated tokenization settings.

The results indicate that the effect of tokenization is encoder-dependent rather than uniformly positive. For *bert-base-uncased*, model-aligned tokenization improves average g-measure from 74.34 to 75.23 and reduces average p_f from 24.99 to 19.86, but it also decreases average p_d from 74.14 to 70.96. For *bert-large-uncased*, model-aligned tokenization slightly improves average g-measure from 69.94 to 70.04 and improves p_d from 68.36 to 73.72, but increases p_f from 27.80 to 33.13. These results show that tokenizer alignment can improve one evaluation dimension while weakening another.

A similar pattern is observed for the Longformer encoders. For *longformer-base-4096*, model-aligned tokenization slightly improves p_d but reduces average g-measure from 77.03 to 75.88 and increases p_f from 20.29 to 23.73. In contrast, *longformer-large-4096* benefits in both g-measure and p_d, increasing from 75.85 to 76.26 and from 74.00 to 78.02, respectively, although this improvement is accompanied by a higher false-alarm rate. These observations suggest that long-context encoders do not respond uniformly to tokenization changes; their performance depends on the interaction between model capacity, input length, and the structure of bug report text.

The two strongest encoders, *multilingual-e5-large-instruct* and *jina-embeddings-v3*, remain comparatively stable across preprocessing settings. *multilingual-e5-large-instruct* improves its average g-measure from 83.96 to 84.36 and p_d from 80.56 to 81.75 under model-aligned tokenization, with only a small increase in p_f. In contrast, *jina-embeddings-v3* achieves a higher average g-measure under NLTK-based tokenization, increasing from 85.76 under model-aligned tokenization to 86.51

under NLTK-based tokenization. However, model-aligned tokenization gives Jina a slightly higher average p_d of 86.83 compared with 85.98 under NLTK-based tokenization. This indicates that Jina is robust to preprocessing variation and maintains strong performance under both settings.

Overall, the results show that preprocessing and tokenization choices affect LLMSec, but the impact depends on the encoder being used. The model-aligned tokenization does not always perform better than NLTK-based preprocessing across all models and metrics. Instead, it improves detection performance for some encoders while also increasing false alarms for others. The finding suggests that preprocessing should not be treated as a fixed implementation detail; it should be selected together with the embedding model and evaluated according to the target performance trade-off.

Answer to RQ2: The LLMSec shows moderate sensitivity to preprocessing and tokenization choices, and the effect changes with the encoder. The model-aligned tokenization improves some metrics, but it does not consistently perform better than NLTK-based preprocessing. The strongest encoders, particularly *jina-embeddings-v3* and *multilingual-e5-large-instruct*, remain stable across both settings.

5.7. RQ3: how do different LLM-based encoders affect LLMSec's classification performance?

To answer RQ3, we examine how the choice of LLM-based encoder or embedding model influences the effectiveness of LLMSec. The evaluated encoders differ in model size, context length, embedding dimensionality, and pretraining objective. These differences may affect how well a model represents long and heterogeneous bug reports, especially when security-relevant information is distributed across summaries, descriptions, configuration fragments, stack traces, and code-related terms.

We evaluate six general-purpose, off-the-shelf pretrained encoders and embedding models: *bert-base-uncased*, *bert-large-uncased*, *longformer-base-4096*, *longformer-large-4096*, *multilingual-e5-large-instruct*, and *jina-embeddings-v3*. All six models are used as frozen feature extractors, and their parameters are not updated using the SBR and NSBR labels. For each encoder–project pair, the tokenization setting is selected using only the training/validation partition, based on validation g-measure. The selected configuration is then evaluated on the held-out test partition, and the corresponding g-measure, p_d, and p_f values are reported. This protocol prevents test-set-driven model selection while allowing each encoder to be evaluated under its most suitable preprocessing configuration.

Table 8 reports the validation-selected test performance for each encoder across the five datasets. The results show clear differences among the encoders. Among the six general-purpose pretrained encoders evaluated in this study, *jina-embeddings-v3* gives the strongest overall performance, with an average g-measure of 87.28 and an average p_d of 87.65. It also maintains a competitive average p_f of 12.76. The second strongest encoder is *multilingual-e5-large-instruct*, which gives an average g-measure of 84.30 and the lowest average p_f among the evaluated encoders, at 12.27. In contrast, the BERT-based encoders show lower average g-measure values, especially *bert-large-uncased*, which gives an average g-measure of 70.13. These results demonstrate that encoder choice has a clear effect on SBR classification performance within the evaluated frozen-encoder setting.

The comparison in Table 8 is restricted to the six general-purpose pretrained encoders and embedding models evaluated under the frozen-feature protocol of LLMSec. Bug-report-specific pretrained representation frameworks, such as RepresentThemAll (Fang et al., 2023), are not included because their published evaluations address different downstream tasks and employ supervised task-specific fine-tuning of the encoder. Their published results are therefore not directly comparable with the frozen-encoder configurations evaluated in this study. Nevertheless, such domain-specific representation learning is complementary to LLMSec, and a bug-report-specific encoder could be integrated into Phase 3 while retaining the proposed BM25F-guided rebalancing mechanism.

Fig. 5 summarizes the validation-selected results for each encoder. The figure shows that *jina-embeddings-v3* and *multilingual-e5-large-instruct* give stronger classification performance than the BERT and Longformer variants. Within the evaluated encoder set, the performance differences are especially clear for g-measure and p_d. This indicates that representation design may influence how effectively security-relevant information is preserved, although the present experiment does not isolate the contribution of any individual architectural characteristic.

To better understand the observed differences, Table 9 presents the raw architectural characteristics of the evaluated encoders together with their average g-measure. These characteristics are reported descriptively, without normalization or manual weighting, and are not treated as independent or causal explanations of classification performance.

The raw architectural characteristics do not show a monotonic relationship with classification performance. For example, *bert-large-uncased* has more parameters and a larger embedding size than *bert-base-uncased*, but its average g-measure is 5.18 points lower. Similarly, *longformer-large-4096* has more parameters and a larger embedding size than *longformer-base-4096*, while its average g-measure is 0.29 points lower. These comparisons show that increasing parameter count or embedding dimensionality alone does not guarantee stronger SBR prediction.

Context length also does not independently explain the observed performance. Both Longformer models support a context length of 4096 tokens, but their average g-measure values remain below that of *multilingual-e5-large-instruct*, which has a context length of 512 tokens. Among the evaluated models, *jina-embeddings-v3* gives the strongest overall performance, while *multilingual-e5-large-instruct* gives the second-highest average g-measure and the lowest average p_f. These differences likely reflect the interaction of multiple factors, including pretraining objective, representation design, context handling, tokenization compatibility, and suitability for document-level embeddings. The evaluation of six encoders does not isolate the causal contribution of any one of these factors.

Among the six evaluated general-purpose encoders, the strongest results come from *jina-embeddings-v3*. Its long context window can retain more report content before truncation, while its embedding-oriented design may support document-level representation. The *multilingual-e5-large-instruct* also performs strongly and is designed to produce meaningful sentence and document representations. These properties are relevant to SBR prediction, where security-related cues may be implicit and distributed across different parts of a report. However, these observations are descriptive, are limited to the evaluated frozen-encoder configurations, and do not establish that any single architectural characteristic causes the observed performance differences or that general-purpose encoders are superior to bug-report-specific pretrained representation models.

Answer to RQ3: Encoder choice substantially affects LLMSec's performance in the evaluated frozen-feature setting. Among the six general-purpose encoders, *jina-embeddings-v3* achieves the highest average g-measure and p_d, while *multilingual-e5-large-instruct* gives the lowest average p_f. The raw architectural characteristics do not show a monotonic relationship with performance, indicating that the results reflect multiple interacting factors rather than model size, context length, or embedding dimensionality alone. These findings do not constitute a comparison with bug-report-specific models such as RepresentThemAll.

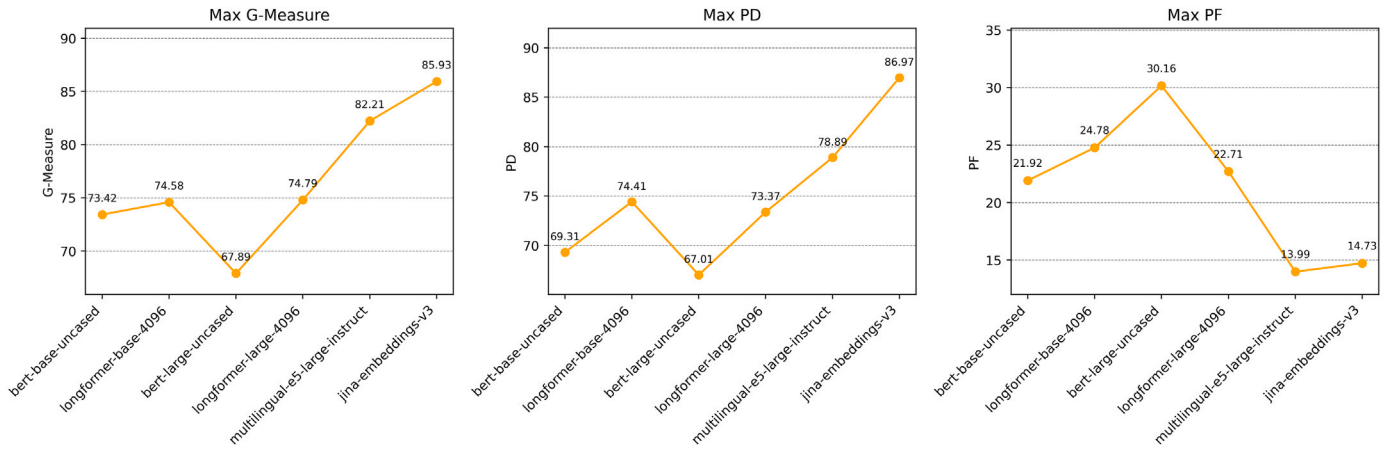


Fig. 5. Validation-selected g-measure, p_d, and p_f for each encoder within the LLMSec pipeline.

Table 8
The validation-selected test performance of LLMSec using six general-purpose pretrained encoders and embedding models across five datasets. All encoders are evaluated as frozen feature extractors. For each encoder–project pair, the tokenization setting is selected using validation g-measure, and the selected configuration is then evaluated on the held-out test set.

Encoder	Metric	Wicket	Ambari	Camel	Derby	Chromium	Average
bert-base-uncased	g-measure	77.35	72.28	74.02	70.03	82.89	75.31
	p_d	73.91	67.86	70.27	65.17	82.92	72.03
	p_f	18.87	22.67	21.81	24.33	17.13	20.96
bert-large-uncased	g-measure	72.80	62.37	69.31	67.08	79.10	70.13
	p_d	65.22	57.14	75.68	70.00	78.96	69.40
	p_f	17.61	31.36	36.07	35.61	20.74	28.28
longformer-base-4096	g-measure	80.99	72.14	71.52	73.68	88.38	77.34
	p_d	83.33	75.00	64.86	74.44	86.63	76.85
	p_f	21.22	30.51	20.30	27.07	9.79	21.78
longformer-large-4096	g-measure	82.76	70.87	73.63	71.92	86.09	77.05
	p_d	91.30	67.86	70.27	64.04	84.15	75.52
	p_f	24.32	25.85	22.68	18.00	11.87	20.54
multilingual-e5-large-instruct	g-measure	86.77	80.95	76.71	84.38	92.68	84.30
	p_d	86.96	78.57	70.27	79.78	90.84	81.28
	p_f	13.42	16.53	15.55	10.46	5.38	12.27
jina-embeddings-v3	g-measure	89.15	83.81	83.72	87.04	92.68	87.28
	p_d	91.67	85.71	78.38	92.13	90.34	87.65
	p_f	13.24	18.01	10.15	17.52	4.86	12.76

Table 9
Architectural characteristics and observed average g-measure of the LLM-based encoders and embedding models evaluated in LLMSec. The comparison is descriptive and does not attribute performance differences to any individual architectural characteristic.

Encoder	Parameters	Context length	Embedding size	Average g-measure
bert-base-uncased	110M	512	768	75.31
bert-large-uncased	336M	512	1024	70.13
longformer-base-4096	112M	4096	768	77.34
longformer-large-4096	334M	4096	1024	77.05
multilingual-e5-large-instruct	560M	512	1024	84.30
jina-embeddings-v3	572M	8192	1024	87.28

5.8. RQ4: How robust is LLMSec under class imbalance across different projects?

To answer RQ4, we examine whether LLMSec keeps stable performance across projects with different class imbalance levels. As shown in Table 2, the evaluated datasets differ substantially in their SBR ratios, ranging from 1.93% in *Chromium* to 17.90% in *Derby*. This variation gives a suitable setting to evaluate whether a method remains effective when security reports are rare and project characteristics are different. As a complementary robustness analysis, we also examine the sensitivity of LLMSec to controlled SBR and NSBR label errors introduced into the training data.

The proposed approach is compared with six baseline methods, including FARSEC (Peters et al., 2017), FARSEC$^{Tuned}_{Learner}$ (Wu et al., 2021), SMOTUNED (Shu et al., 2019), LTRWES (Jiang et al., 2020), FASTTEXT (Alqahtani, 2024), and BERT$_{WPP}$ (Soltaniani et al., 2025). To assess cross-project robustness, we report the mean and sample standard deviation of g-measure, p_d, and p_f across the five datasets. For g-measure and p_d, higher mean values indicate stronger detection performance, whereas lower standard deviation indicates more stable behavior across projects. For p_f, lower mean values indicate fewer false alarms.

Table 10 shows that LLMSec achieves the highest average g-measure and p_d while also obtaining the lowest standard deviation for both

Table 10
Cross-project robustness of the evaluated methods, reported as mean ± sample standard deviation across five datasets.

Method	g-measure	p_d	p_f
FARSEC (Peters et al., 2017)	63.56 ± 6.59	54.20 ± 7.86	19.52 ± 18.43
$\text{FARSEC}^{Tuned}_{Learner}$ (Wu et al., 2021)	63.40 ± 15.65	53.60 ± 18.04	12.60 ± 17.80
SMOTUNED (Shu et al., 2019)	72.41 ± 6.12	62.26 ± 8.29	12.62 ± 6.99
LTRWES (Jiang et al., 2020)	77.40 ± 9.86	79.26 ± 10.33	24.06 ± 10.74
FASTTEXT (Alqahtani, 2024)	79.60 ± 9.32	80.80 ± 8.32	21.40 ± 10.77
BERT_{WPP} (Soltaniani et al., 2025)	47.20 ± 23.54	33.60 ± 22.63	**1.60 ± 1.52**
LLMSec	**87.28 ± 3.79**	**87.65 ± 5.83**	12.76 ± 5.28

For g-measure and p_d, higher values are better; for p_f, lower values are better. Lower standard deviation indicates more stable cross-project behavior.

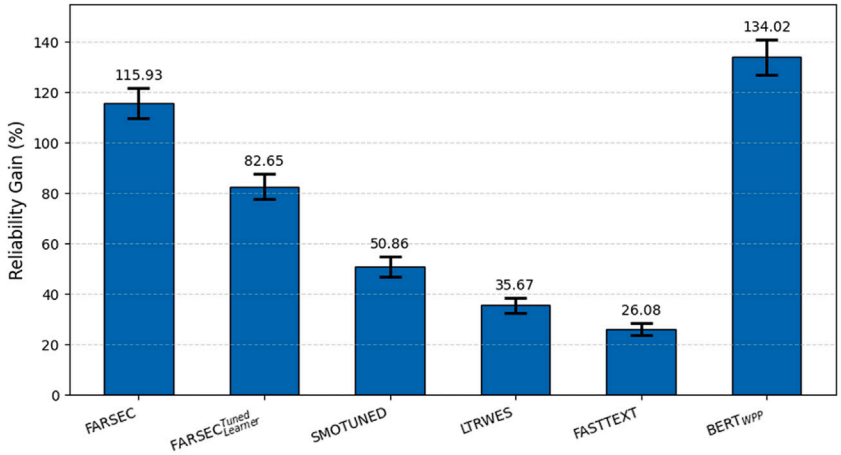


Fig. 6. Reliability gain (%) of LLMSec over baseline methods based on the detection margin ($p_d - p_f$). Higher values indicate a larger relative improvement in balancing true security-report detection against false alarms. Percentage gains should be interpreted together with the absolute p_d and p_f values because a small baseline detection margin can produce a comparatively large relative gain.

metrics. Specifically, LLMSec obtains an average g-measure of 87.28 with a standard deviation of 3.79. In comparison, the strongest non-LLMSec baseline, FASTTEXT, obtains an average g-measure of 79.60 with a substantially higher standard deviation of 9.32. Similarly, LLMSec achieves the highest average p_d of 87.65 and the lowest p_d standard deviation of 5.83. These results indicate that LLMSec not only detects more true SBRs on average, but also varies less across projects.

The p_f results give a complementary view of robustness. BERT_{WPP} achieves the lowest average p_f of 1.60, but this comes with much lower average g-measure and p_d, showing that it behaves conservatively and misses many true security reports. In contrast, LLMSec keeps a competitive average p_f of 12.76 while giving much stronger detection performance. Compared with LTRWES and FASTTEXT, LLMSec reduces the average false-alarm rate while improving both g-measure and p_d. Compared with SMOTUNED and $\text{FARSEC}^{Tuned}_{Learner}$, LLMSec has a similar false-alarm level but much stronger detection performance. This suggests that the proposed BM25F-guided relevance-based rebalancing strategy improves robustness without simply increasing the number of reports classified as security-related.

To further quantify the trade-off between detecting true SBRs and avoiding false alarms, we compute reliability gain (RG) based on the detection margin ($p_d - p_f$). This margin rewards methods that increase true-positive detection while penalizing false alarms. The reliability gain of LLMSec over a baseline is defined as:

$$RG = \frac{(p_d - p_f)_{LLMSec} - (p_d - p_f)_{\text{Baseline}}}{(p_d - p_f)_{\text{Baseline}}} \times 100\%. \tag{18}$$

As shown in Fig. 6, LLMSec improves the detection margin over all baselines. The gain over FASTTEXT is 26.08%, even though FASTTEXT is the strongest baseline in terms of average g-measure among the compared methods. The gain over LTRWES is 35.67%, showing that LLMSec provides a better balance between detecting SBRs and controlling false alarms than relevance-based embedding methods. The largest gain is observed over BERT_{WPP}, because BERT_{WPP} achieves a very low p_f but also a very low p_d, which weakens its detection margin. Because RG is normalized by the baseline detection margin, a small baseline value can produce a comparatively large percentage gain. This is particularly relevant to BERT_{WPP} in project-level cases where p_d and p_f are close. Therefore, the reliability-gain percentage should not be interpreted as a standalone measure of practical improvement and should be considered together with the absolute p_d and p_f values and the robustness results in Table 10.

5.8.1. Robustness to controlled training-label noise

We further evaluate the sensitivity of LLMSec to mislabeled training instances using the corrected dataset labels as the clean reference condition. SBR→NSBR and NSBR→SBR corruption are introduced separately at noise levels of 5%, 10%, and 20%. The corruption is restricted to the training partition, while the test reports and their labels remain unchanged. Table 11 reports the average performance across the 30 runs for each dataset and noise condition.

The effect of label corruption depends strongly on its direction. Under SBR→NSBR corruption, the average p_d decreases from 87.65 in the clean condition to 85.60, 83.27, and 78.72 at noise levels of 5%, 10%, and 20%, respectively. The average p_f decreases from 12.76 to 11.27 at 20% noise. This pattern indicates that incorrectly treating positive training examples as non-security reports makes the classifier more conservative, reducing both true SBR detection and false alarms.

In contrast, NSBR→SBR corruption has a smaller effect on average p_d but substantially increases p_f. At 20% noise, the average p_d remains 85.79, whereas the average p_f increases to 23.45. The average g-measure decreases from 87.28 under clean labels to 83.20 under 20% SBR→NSBR corruption and to 80.74 under 20% NSBR→SBR corruption. Thus, performance decreases progressively across the evaluated

Table 11
Performance of LLMSec under controlled training-label noise, reported as average values across 30 runs.

Training-label condition	Noise	Metric	Ambari	Camel	Derby	Wicket	Chromium	Average
Clean labels	0%	g-measure	83.81	83.72	87.04	89.15	92.68	**87.28**
		p_d	85.71	78.38	92.13	91.67	90.34	**87.65**
		p_f	18.01	10.15	17.52	13.24	4.86	**12.76**
SBR→NSBR	5%	g-measure	82.95	82.45	86.34	88.43	91.94	86.42
		p_d	83.51	75.85	90.15	89.80	88.69	85.60
		p_f	17.61	9.69	17.16	12.90	4.56	12.38
SBR→NSBR	10%	g-measure	81.89	80.91	85.49	87.56	91.06	85.38
		p_d	81.01	72.97	87.90	87.67	86.81	83.27
		p_f	17.21	9.23	16.80	12.56	4.26	12.01
SBR→NSBR	20%	g-measure	79.67	77.65	83.70	85.75	89.25	83.20
		p_d	76.11	67.34	83.49	83.51	83.14	78.72
		p_f	16.41	8.31	16.08	11.88	3.66	11.27
NSBR→SBR	5%	g-measure	82.46	82.49	85.73	87.99	91.79	86.09
		p_d	86.01	78.72	92.40	91.92	90.56	87.92
		p_f	20.81	13.37	20.04	15.62	6.96	15.36
NSBR→SBR	10%	g-measure	80.45	80.40	83.85	86.28	90.38	84.27
		p_d	85.21	77.80	91.68	91.25	89.97	87.18
		p_f	23.81	16.82	22.74	18.17	9.21	18.15
NSBR→SBR	20%	g-measure	76.53	76.35	80.21	82.97	87.66	80.74
		p_d	83.71	76.08	90.33	89.97	88.84	85.79
		p_f	29.51	23.38	27.87	23.02	13.48	23.45

noise levels, with NSBR→SBR errors causing the greater deterioration in the balance between detection and false alarms.

This experiment measures sensitivity to randomly injected training-label errors and does not introduce an explicit mechanism for detecting, correcting, or removing mislabeled reports. It also does not evaluate reports with manually validated mixed security and non-security semantics or reports containing incomplete descriptions, corrupted logs, or missing configuration information. Therefore, the findings should not be interpreted as evidence of robustness to semantic ambiguity or naturally occurring report-quality degradation.

Overall, these findings show that LLMSec provides stable detection performance across projects with substantially different class-imbalance levels and degrades progressively under the evaluated training-label-noise conditions. It achieves the strongest average g-measure and p_d, the lowest cross-project variation in both detection-oriented metrics, and a competitive false-alarm rate under clean labels. The label-noise results further show that SBR→NSBR errors mainly reduce detection, whereas NSBR→SBR errors mainly increase false alarms. These results establish robustness only within the evaluated class-imbalance and controlled label-corruption settings and do not imply that LLMSec performs automatic label denoising.

Answer to RQ4: LLMSec maintains stable performance across projects with different imbalance levels. It achieves the highest average g-measure and p_d, the lowest variation for both detection-oriented metrics, and consistent reliability gains over the baselines. Under controlled training-label corruption, performance decreases progressively rather than abruptly: SBR→NSBR errors primarily reduce p_d, whereas NSBR→SBR errors primarily increase p_f. These findings demonstrate robustness within the evaluated class-imbalance and label-noise settings, but they do not establish automatic denoising or robustness to semantically ambiguous or textually degraded reports.

5.9. Ablation study: Contribution of rebalancing and embedding components

To further examine the contribution of the main components of LLMSec, we conduct an ablation study that separates the effects of BM25F-guided relevance-based rebalancing and LLM-derived contextual embeddings. This analysis helps determine whether the observed improvements arise from the rebalancing strategy, the embedding representation, or their combination.

All ablation variants are evaluated using the same five datasets, train–test splits, preprocessing policy, repeated-run protocol, evaluation metrics, and downstream classifier settings used in the main experiment. For all variants that reduce the number of NSBR instances, the number of selected NSBRs is kept identical to the full LLMSec configuration. This control ensures that performance differences are not caused merely by changes in training-set size.

We consider eight variants. **TF–IDF + ML** removes both BM25F-guided rebalancing and LLM-derived embeddings, serving as a sparse lexical baseline. **Random Under-sampling + TF–IDF** controls for the effect of reducing the majority class without relevance-based selection, whereas **BM25F Rebalancing + TF–IDF** isolates the contribution of BM25F-guided NSBR selection while retaining the same lexical representation. To examine whether the observed benefit of BM25F-guided rebalancing is limited to the Jina-based configuration, we repeat the direct comparison between random under-sampling and BM25F-guided rebalancing using two LLM-derived embedding models: jina-embeddings-v3 and multilingual-e5-large-instruct. The latter is included as a second encoder because it is the second-strongest encoder in the evaluation presented in Section 5.7. Accordingly, **Random Under-sampling + E5 Embeddings** and **Random Under-sampling + Jina Embeddings** evaluate the embedding models without relevance-guided selection, while **BM25F Rebalancing + E5 Embeddings** and **BM25F Rebalancing + Jina Embeddings** evaluate the same encoders with BM25F-guided selection. Finally, **RWS Selector + Jina Embeddings** examines whether the pipeline remains stable when NSBRs are selected probabilistically rather than strictly using the minimum-similarity criterion.

Table 12 and Fig. 7 show that both BM25F-guided rebalancing and LLM-derived embeddings contribute to the final performance of LLMSec. The sparse lexical baseline, **TF–IDF + ML**, obtains an average g-measure of 64.12 and an average p_d of 57.84, indicating that conventional lexical features alone are insufficient for reliably detecting security bug reports. Random under-sampling improves p_d from 57.84 to 71.20, but it also increases p_f from 18.92 to 30.15. This shows that simply reducing the majority class can increase sensitivity, but at the cost of substantially more false alarms.

Table 12
Ablation analysis of LLMSec components. Results are reported as average g-measure, p_d, and p_f across the five datasets. For g-measure and p_d, higher values are better; for p_f, lower values are better. E5 denotes multilingual-e5-large-instruct.

Variant	BM25F rebalancing	LLM-derived embeddings	g-measure	p_d	p_f
TF–IDF + ML	No	No	64.12	57.84	18.92
Random Under-sampling + TF–IDF	No	No	70.45	71.20	30.15
BM25F Rebalancing + TF–IDF	Yes	No	74.83	73.64	23.50
Random Under-sampling + E5 Embeddings	No	Yes	79.70	78.32	18.68
BM25F Rebalancing + E5 Embeddings	Yes	Yes	84.30	81.28	12.27
Random Under-sampling + Jina Embeddings	No	Yes	82.46	84.18	18.92
RWS Selector + Jina Embeddings	Yes	Yes	85.79	86.34	14.85
BM25F Rebalancing + Jina Embeddings	Yes	Yes	**87.28**	**87.65**	**12.76**

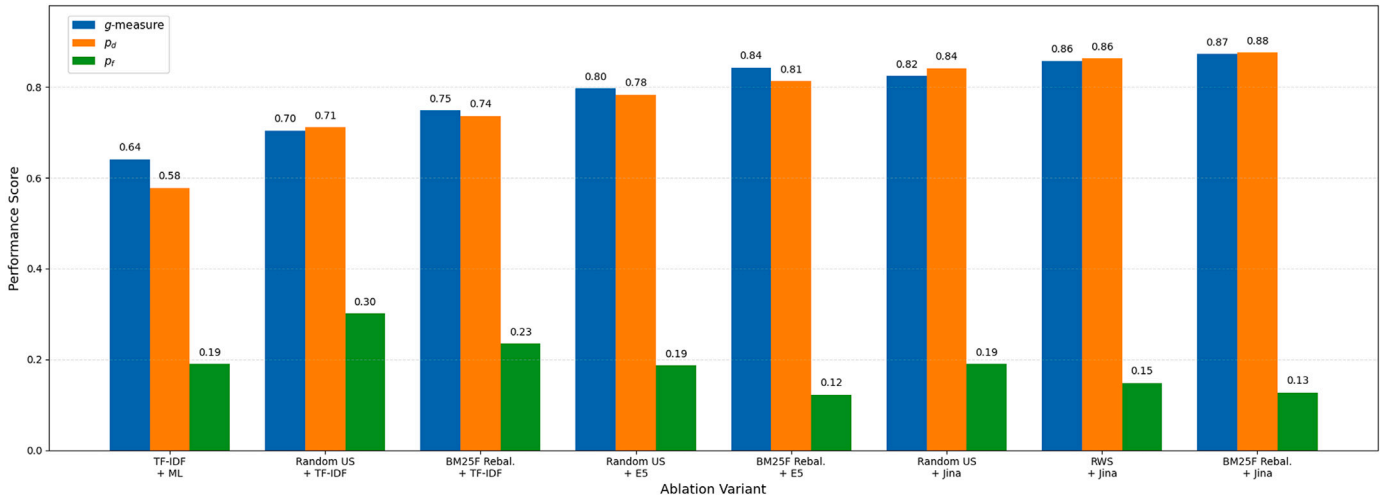


Fig. 7. Ablation analysis of LLMSec components using lexical, E5, and Jina representations. Results show the average g-measure, p_d, and p_f across the five datasets. Higher values are better for g-measure and p_d, while lower values are better for p_f.

Adding BM25F-guided rebalancing to TF–IDF improves the average g-measure from 70.45 to 74.83 compared with random under-sampling under the same lexical representation. It also improves p_d from 71.20 to 73.64 while reducing p_f from 30.15 to 23.50. This corresponds to improvements of 4.38 points in g-measure and 2.44 points in p_d, together with a 6.65-point reduction in p_f. This indicates that relevance-based NSBR selection is more effective than randomly reducing the majority class. The result supports the idea that the composition of the negative class matters: selecting low-overlap real NSBRs using BM25F provides a better training set than random sampling under the same representation type.

The same pattern is observed with multilingual-e5-large-instruct. The **Random Under-sampling + E5 Embeddings** variant obtains an average g-measure of 79.70, p_d of 78.32, and p_f of 18.68. Replacing random under-sampling with BM25F-guided rebalancing increases the average g-measure to 84.30 and p_d to 81.28, while reducing p_f to 12.27. Thus, BM25F-guided rebalancing improves g-measure by 4.60 points and p_d by 2.96 points, while reducing p_f by 6.41 points under the E5 representation.

The **Random Under-sampling + Jina Embeddings** variant achieves an average g-measure of 82.46 and an average p_d of 84.18, showing that contextual document-level representations substantially improve SBR detection even without BM25F-guided selection. However, its average p_f remains 18.92, indicating that stronger embeddings alone do not fully control false alarms under class imbalance.

The full proposed configuration, **BM25F Rebalancing + Jina Embeddings**, gives the best overall performance, with an average g-measure of 87.28, an average p_d of 87.65, and an average p_f of 12.76. Compared with **Random Under-sampling + Jina Embeddings**, it improves g-measure by 4.82 points and p_d by 3.47 points, while reducing p_f by 6.16 points. Compared with **BM25F Rebalancing + TF–IDF**, it improves g-measure by 12.45 points and p_d by 14.01 points, while reducing p_f by 10.74 points.

The closely aligned improvements obtained with E5 and Jina show that the observed benefit of BM25F-guided rebalancing is not confined to the Jina-based configuration. With E5, BM25F increases g-measure by 4.60 points and reduces p_f by 6.41 points; with Jina, it increases g-measure by 4.82 points and reduces p_f by 6.16 points. A similar benefit is also observed with TF–IDF, where BM25F increases g-measure by 4.38 points and reduces p_f by 6.65 points. The consistent improvement across the evaluated TF–IDF, E5, and Jina representations indicates that BM25F-guided selection provides benefits under multiple representation settings examined in this study. This evidence should not be interpreted as establishing that the effect is independent of representation choice in general. Rather, it shows that the improvement is reproduced across the three representations evaluated in the ablation. The embeddings improve contextual report representation, whereas BM25F-guided selection improves the construction of the majority-class training set under class imbalance.

The comparison between the minimum-similarity selector and the RWS selector also shows that the rebalancing strategy remains stable across selection mechanisms. The RWS variant gives a competitive g-measure of 85.79 and p_d of 86.34, but it remains below the full minimum-similarity configuration. The full configuration improves over RWS by 1.49 points in g-measure and 1.31 points in p_d, while reducing p_f by 2.09 points. This indicates that probabilistic selection preserves useful diversity among NSBRs, but strict selection of low-overlap NSBRs provides a more stable decision boundary for the evaluated datasets.

Ablation finding: The performance of LLMSec results from the complementary contributions of LLM-derived embeddings and BM25F-guided relevance-based rebalancing. The embeddings improve contextual report representation, while BM25F-guided selection improves g-measure and reduces false alarms across the evaluated TF–IDF, multilingual-e5-large-instruct, and jina-embeddings-v3 representations. The consistent pattern across these representations indicates that the observed benefit of BM25F-guided rebalancing is not confined to the Jina-based configuration, without implying representation independence beyond the settings evaluated in this study.

5.10. Computational cost analysis

To examine the computational overhead of LLMSec, we compare its end-to-end execution time, peak host-memory usage, and GPU-memory requirement with FARSEC and FASTTEXT, representing two lightweight baseline approaches. All three methods are benchmarked on the same workstation running Ubuntu 24.04.4 LTS with 503 GiB of host memory and an NVIDIA RTX PRO 5000 Blackwell GPU with 48,935 MiB of dedicated GPU memory. The LLMSec benchmark uses PyTorch 2.8.0 with CUDA 12.8 for neural embedding generation. A consistent measurement protocol is applied over the five evaluated projects: Ambari, Camel, Chromium, Derby, and Wicket. FARSEC and FASTTEXT are executed entirely on the CPU, whereas LLMSec uses GPU acceleration for neural embedding generation.

For reproducibility, one fixed configuration is used for each method throughout the computational benchmark. FARSEC is executed in the WPP setting using the FARSEC filtering strategy and the Naive Bayes classifier. FASTTEXT is trained for 40 epochs with a learning rate of 0.5 and the one-vs-all loss. For LLMSec, we use `jina-embeddings-v3` as the pretrained encoder with a maximum input length of 8192 tokens, NLTK-based preprocessing for the BM25F stage, the `ms` semantic-filtering strategy with `times` = 5, and logistic regression as the downstream classifier. The cleaned summary and description fields of each bug report are concatenated before neural encoding, and the resulting contextual representation is used for classification. A single fixed encoder configuration is used for this computational benchmark so that the reported runtime and memory values characterize one clearly defined experimental setting rather than an average across encoder configurations with different computational requirements. The encoder is executed using the native PyTorch attention implementation available in the benchmark environment.

End-to-end time includes model loading, data loading, method-specific preprocessing performed during execution, feature or embedding generation, classifier training, testing or inference, and evaluation. For LLMSec, the measurement also includes BM25F-based semantic rebalancing. One-time dependency downloads, pretrained-model downloads, and offline dataset preparation are excluded. The LLMSec benchmark uses a fixed 50/50 stratified train–test split for each project. Peak host memory denotes the maximum process resident set size (RSS), measured using GNU `/usr/bin/time` and converted to GiB. For LLMSec, the maximum GPU memory allocated during each project-level run is additionally recorded. Host-memory usage and GPU-memory allocation are reported separately and are not combined.

Table 13 shows that LLMSec incurs substantially greater computational overhead than the lightweight baselines under the evaluated benchmark configuration. Across the five project-level benchmarking runs, its cumulative end-to-end execution time is 504.19 s, compared with 25.89 s for FARSEC and 3.28 s for FASTTEXT. Its maximum observed host-memory usage is 3.261 GiB, compared with 1.765 GiB for FARSEC and 0.239 GiB for FASTTEXT. In addition, LLMSec reaches a maximum allocated GPU memory of 4.675 GiB, whereas FARSEC and FASTTEXT do not use the GPU in these experiments. The additional computational cost of LLMSec primarily reflects neural contextual-embedding generation together with BM25F-guided semantic rebalancing, with the runtime effect being particularly noticeable for Chromium because it contains substantially more bug reports than the other evaluated projects.

Table 13
Computational cost of the evaluated fixed configurations of LLMSec, FARSEC, and FASTTEXT over the five projects (Ambari, Camel, Chromium, Derby, and Wicket). End-to-end time is the cumulative time over one benchmarking run per project, whereas peak host RAM and peak allocated GPU memory are the maximum values observed across the project-level runs.

Method	End-to-end time (s)	Peak host RAM (GiB)	Peak allocated GPU memory (GiB)
FARSEC	25.89	1.765	–
FASTTEXT	3.28	0.239	–
LLMSec	504.19	3.261	4.675

These computational measurements are based on one benchmarking run per project for each method and are separate from the 30 repeated runs used for predictive-performance evaluation. They should therefore be interpreted as descriptive measurements under the stated benchmark environment rather than as averages over the repeated predictive-performance runs. Specifically, the reported LLMSec values correspond to the fixed `jina-embeddings-v3` configuration with a maximum input length of 8192 tokens, NLTK-based BM25F preprocessing, the `ms` semantic-filtering strategy with `times` = 5, and logistic regression as the downstream classifier. They are not intended to represent the computational cost of every encoder configuration evaluated in the study. Because runtime and memory requirements depend on encoder architecture, input length, inference implementation, software environment, and hardware characteristics, other LLMSec configurations may exhibit different computational requirements. The analysis therefore provides a reproducible reference point for the computational performance–cost trade-off of the evaluated LLMSec configuration, particularly in resource-constrained or latency-sensitive settings.

5.11. Threats to validity

Although the results show that LLMSec improves security bug report prediction on the evaluated datasets, several threats should be considered.

5.11.1. Internal validity

The key internal threat is information leakage during BM25F-guided rebalancing. Since LLMSec ranks and selects NSBRs using their BM25F similarity to known SBRs, all data-dependent operations, including preprocessing, BM25F-based ranking, relevance-guided rebalancing, embedding normalization, classifier training, and hyperparameter tuning, are performed only on the training partition. The test partition is used only for final evaluation. For each of the 30 runs, the SBR reference set and candidate NSBR pool are reconstructed from the run-specific training partition, and the BM25F similarity scores, NSBR ranking, and selected NSBR subset are recomputed. Thus, no test report or fixed dataset-level ranking is reused across runs, and the reported variation includes the effect of run-specific rebalancing. A further internal-validity consideration is the use of a fixed BM25F parameter configuration across all projects and experimental repetitions. Although this provides a consistent relevance-ranking procedure, the study does not perform project-specific BM25F optimization or a dedicated sensitivity analysis over the parameter space. Consequently, projects with different terminology, field usage, or sparse bigram patterns may favor different parameter settings, and the reported results should not be interpreted as establishing that the selected BM25F configuration is optimal for every project.

Implementation differences among baselines may also affect the results. We reduce this risk by using official implementations or reported configurations when available, applying the same experimental protocol, and repeating experiments with different random seeds. Variation may also arise from differences among pretrained encoders in tokenization, input length, and truncation behavior. We address this by evaluating NLTK-based preprocessing and model-aligned tokenization across multiple encoders.

5.11.2. External validity

The study uses five public SBR datasets, *Wicket*, *Ambari*, *Camel*, *Derby*, and *Chromium*. These datasets differ in vocabulary, project characteristics, and class imbalance, but they may not cover all software domains, industrial workflows, security taxonomies, or reporting practices. Therefore, the findings should not be treated as conclusive evidence that LLMSec will perform equally well in all proprietary or industrial settings. The evaluation is limited to historical, static reports from open-source projects and uses repeated stratified random splits rather than chronological train–test partitions. Consequently, the results do not establish temporal generalization to future reports or transfer to proprietary industrial software, embedded systems, cloud-native platforms, multilingual bug reports, or continuously updated security-triage environments. In particular, although multilingual evaluation is identified as a direction for future work, no empirical multilingual validation is included in the present study. These settings require separate evaluation before conclusions can be drawn about the broader deployment and cross-domain applicability of LLMSec.

5.11.3. Construct validity

We evaluate performance using g-measure, probability of detection (p_d), and probability of false alarm (p_f). These metrics are suitable for imbalanced SBR prediction because they capture both detection performance and false-alarm behavior. However, they do not directly measure downstream outcomes such as triage effort, remediation time, alert fatigue, or the operational cost of missed vulnerabilities. Label quality remains a potential construct threat because some security-related cases may be inherently ambiguous or project-dependent, even when corrected benchmark labels are used.

5.11.4. Conclusion validity

Statistical inference is conducted at the project level using g-measure, with software project treated as the unit of inference. The 30 repeated train–test executions are used to summarize performance under partition variation but are not treated as independent observations. A permutation-based Friedman test with Kendall's W is used for the overall comparison, while exact two-sided Wilcoxon signed-rank tests with Benjamini–Hochberg adjustment and rank-biserial correlation are used for pairwise comparisons. Because only five projects are available, pairwise inference has limited statistical resolution. The findings therefore characterize performance within the evaluated benchmark and do not establish population-level superiority or broader cross-project generalization.

5.11.5. Reproducibility

To support reproducibility, we provide the implementation of LLMSec, processed datasets where redistribution is permitted, parameter settings, and the complete preprocessing, repeated-run, evaluation, and result-processing scripts required to reproduce all tables.[4] Exact reproducibility may still be affected by hardware differences, library versions, pretrained model updates, tokenizer versions, and nondeterministic GPU operations. To reduce this risk, the replication package specifies the software environment and model versions used in the experiments.

6. Discussion

The experimental results show that the proposed approach provides strong overall performance for security bug report prediction. As shown in Table 4, LLMSec achieves the highest average g-measure and p_d, with values of 87.28 and 87.65, respectively. Compared with FASTTEXT, the strongest baseline in terms of average g-measure, LLMSec improves g-measure by 9.65% and p_d by 8.48%. The project-level analysis in Table 5 further shows that LLMSec obtains the best average rank of 1.40 across the five projects. The permutation-based Friedman test indicates an overall difference among the evaluated methods (Q_F = 24.77, $p = 8.22 \times 10^{-7}$, Kendall's W = 0.826). Pairwise project-level differences favor LLMSec over four baselines on all five projects and over LTRWES and FASTTEXT on four of five projects, although the exact Wilcoxon comparisons have limited inferential resolution with only five project-level observations.

The main reason for this improvement is the combination of BM25F-guided relevance-based rebalancing and LLM-derived contextual embeddings. The BM25F-guided rebalancing reduces the dominance of the majority NSBR class by selecting real non-security reports with low overlap, while the LLM-derived embeddings help preserve security-relevant meaning that may be implicit or spread across long report descriptions. This combination allows LLMSec to improve detection without simply increasing false alarms.

The BM25F configuration is fixed across all projects, experimental configurations, and repetitions to maintain a consistent relevance-guided selection protocol. However, the study does not perform project-specific BM25F optimization or a dedicated sensitivity analysis over its parameter space. Consequently, the selected weighting and normalization settings should not be interpreted as optimal for every project, particularly when project terminology, field usage, or bigram frequency differs substantially. BM25F also measures lexical relevance rather than deep semantic similarity. Although contextual semantic information is introduced subsequently through the pretrained encoder, the preceding NSBR-selection stage remains dependent on lexical overlap. Reports that express related concepts using substantially different wording may therefore receive low BM25F similarity despite being semantically related. These limitations motivate project-specific BM25F tuning and hybrid selection strategies that combine lexical relevance with embedding-based semantic similarity.

The results also show that encoder and preprocessing choices matter. In RQ2, the effect of tokenization depends on the encoder and is not uniformly beneficial. In RQ3, among the six general-purpose pretrained encoders and embedding models evaluated as frozen feature extractors, *jina-embeddings-v3* provides the strongest overall performance, while *multilingual-e5-large-instruct* achieves the lowest average false-alarm rate. The raw architectural characteristics do not show a monotonic relationship with performance, since larger models, longer context windows, and higher embedding dimensionality do not consistently produce better results. The observed differences therefore likely reflect the interaction of pretraining objective, representation design, context handling, tokenization compatibility, and suitability for document-level embeddings rather than any individual architectural characteristic.

The encoder analysis is limited to general-purpose, off-the-shelf pretrained models used as frozen feature extractors. Consequently, their representation spaces are not adapted specifically to SBR classification, and task-specific software-security signals are not reinforced through supervised fine-tuning. Bug-report-specific representation-learning frameworks, such as RepresentThemAll (Fang et al., 2023), follow a different strategy by pretraining directly on bug-report corpora using domain-oriented learning objectives and subsequently adapting the encoder through supervised fine-tuning for individual downstream tasks. Its published results are therefore not directly comparable with the frozen-encoder configurations evaluated in LLMSec, since the studies differ in prediction tasks, datasets, training protocols, and evaluation

measures. Nevertheless, the two directions are complementary, and a bug-report-specific pretrained encoder could be incorporated into Phase 3 of LLMSec while retaining the proposed relevance-guided rebalancing mechanism.

Accordingly, the finding that *jina-embeddings-v3* provides the strongest overall representation should be interpreted only within the six frozen general-purpose encoders evaluated in this study. It does not establish that frozen encoders are superior to task-specific fine-tuning or to models pretrained specifically on bug-report or software-security corpora. Although BERT_{WPP} provides an end-to-end fine-tuned transformer baseline in the comparison, its observed performance represents one particular model and training configuration and should not be interpreted as evidence about the general effectiveness of fine-tuned transformers. The present study also does not evaluate instruction-tuned generative LLMs, prompt-based SBR classification, prompt-engineering optimization, or other forms of task-specific adaptation. Controlled comparisons of frozen representations with full or parameter-efficient fine-tuning, as well as zero-shot and few-shot generative approaches with systematically optimized prompting strategies, under the same datasets, data partitions, and evaluation measures remain important directions for future investigation.

The downstream evaluation includes LR, RF, SVM, and MLP, with the MLP providing a nonlinear neural classification model. However, the study does not examine deeper neural classification heads, token- or field-level cross-attention mechanisms, or imbalance-aware objectives such as weighted cross-entropy and focal loss. Consequently, the findings are limited to the evaluated downstream classifiers and do not establish that more complex neural heads or specialized loss functions would not yield additional improvements under severe class imbalance. A token- or field-level cross-attention architecture would also require retaining structured representations rather than relying only on the pooled document embeddings used in the current pipeline. Controlled comparisons of these alternatives under the same datasets, data partitions, and repeated-run protocol remain necessary.

The present evaluation focuses on predictive performance and does not provide fragment-level explanations identifying whether natural-language descriptions, code snippets, configuration content, or stack traces drive individual SBR predictions. Because each report is represented by a pooled document embedding, the current framework does not directly attribute a prediction to specific textual fragments. Fragment-occlusion analysis and token- or sentence-level attribution methods should therefore be investigated to support case-level interpretation. The study also does not measure operational outcomes such as developer triage time, vulnerability-remediation delay, or alert fatigue. Although p_f quantifies the rate of false alarms, it does not by itself establish their practical burden on developers. Such operational effects cannot be inferred directly from g-measure, p_d, and p_f and require deployment data or controlled studies involving security practitioners.

RQ4 further shows that LLMSec remains stable across projects with different imbalance levels. Table 10 shows that it achieves the lowest standard deviation for both g-measure and p_d, while Fig. 6 shows that it improves the detection margin ($p_d - p_f$) over all baselines. This indicates that BM25F-guided relevance-based rebalancing helps maintain reliable behavior across heterogeneous projects, rather than improving performance only in isolated cases.

The controlled training-label-noise analysis in Table 11 provides additional evidence about the sensitivity of LLMSec to mislabeled training instances. Under SBR→NSBR corruption, the average p_d decreases progressively from 87.65 with clean labels to 78.72 at 20% noise, while the average p_f decreases from 12.76 to 11.27. This indicates that incorrectly relabeling positive training instances as NSBRs makes the classifier more conservative and increases missed security reports. In contrast, NSBR→SBR corruption has a smaller effect on detection but substantially increases false alarms. At 20% noise, the average p_d remains 85.79, whereas the average p_f increases to 23.45. The average g-measure decreases to 83.20 under 20% SBR→NSBR corruption and to 80.74 under 20% NSBR→SBR corruption.

These results show progressive rather than abrupt degradation under the evaluated random label-corruption conditions. However, the experiment measures sensitivity to mislabeled training data and does not introduce an explicit mechanism for detecting, correcting, or removing incorrect labels. The corrected benchmark datasets also do not identify a separately annotated subset of reports containing mixed security and non-security semantics. Consequently, the present study does not establish performance on manually validated ambiguous reports. It also does not test incomplete descriptions, missing fields, garbled logs, or absent configuration information, and therefore does not establish robustness to naturally occurring report-quality degradation.

The reported robustness should be interpreted within the present evaluation setting. The experiments use historical reports from five public open-source projects and repeated stratified random splits rather than chronological train–test partitions. Therefore, they do not establish temporal generalization to future reports or transfer to proprietary industrial software, embedded systems, cloud-native platforms, multilingual bug reports, or continuously updated and real-time security-triage environments, where software architectures, project terminology, reporting practices, and security vocabularies may differ substantially from the evaluated benchmarks. Although multilingual evaluation is identified as future work, no empirical multilingual validation is included in the present study. Additional cross-domain, multilingual, temporal, and deployment-oriented evaluations are therefore required before drawing conclusions about broader industrial or continuous-integration use.

Although LLMSec achieves the strongest overall performance, it is not the best-performing method on every individual project. As analyzed in RQ1, LTRWES performs better on Ambari, where the limited number of positive SBR examples may increase the importance of recurring project-specific lexical patterns. FASTTEXT is marginally ahead in g-measure on Chromium, whose substantially larger corpus may favor the learning of stable word- and subword-level patterns, while LLMSec achieves a lower false-alarm rate. These observations indicate that project characteristics such as positive-class availability, corpus size, and vocabulary regularity can influence the relative effectiveness of different representation strategies. Consequently, the globally fixed LLMSec configuration should not be interpreted as necessarily optimal for every project. These interpretations remain descriptive rather than causal because the present study does not independently isolate the effects of individual dataset characteristics.

Overall, the proposed approach targets two main challenges of SBR prediction: semantic complexity and severe class imbalance. The principal contribution of LLMSec is not universal superiority on every individual project, but the unified integration of BM25F-guided relevance-based rebalancing with contextual document representations. This combination produces the highest average g-measure and p_d across the evaluated projects. Future work should examine project-specific BM25F tuning, hybrid lexical–semantic NSBR selection, chronological validation, proprietary industrial software, embedded systems, cloud-native platforms, multilingual datasets, continuously arriving reports, bug-report-specific pretrained encoders such as RepresentThemAll, task-specific full and parameter-efficient encoder fine-tuning, explicit label-denoising methods, manually reviewed ambiguous-report subsets, and controlled degradation of incomplete or corrupted report content. The findings therefore indicate that the framework provides a reliable predictive improvement within the evaluated benchmark and controlled label-noise settings, while its cross-domain, multilingual, and broader deployment generalizability remains to be established.

7. Conclusion and future work

This paper presented LLMSec, a framework for security bug report prediction that combines BM25F-guided relevance-based rebalancing

with LLM-derived contextual embeddings. LLMSec was evaluated on five public open-source datasets and achieved the strongest average g-measure and probability of detection (p_d), while maintaining a competitive probability of false alarm (p_f). Among the six general-purpose pretrained encoders and embedding models evaluated as frozen feature extractors, *jina-embeddings-v3* achieved the strongest overall performance, while *multilingual-e5-large-instruct* produced the lowest average false-alarm rate. Because the evaluated encoders remain frozen, their representation spaces are not adapted specifically to SBR classification or reinforced using task-specific software-security supervision. Accordingly, these findings are restricted to the evaluated frozen-encoder and downstream-classifier settings and do not establish superiority over task-specific fine-tuning, bug-report-specific encoders, instruction-tuned generative LLMs, deeper neural heads, cross-attention mechanisms, or imbalance-aware loss functions. Although BERT_{WPP} provides an end-to-end fine-tuned transformer baseline, its performance reflects one particular model and training configuration and should not be generalized to fine-tuned transformers as a whole. The project-level permutation-based Friedman analysis indicates an overall difference among the evaluated methods, while the exact pairwise Wilcoxon signed-rank comparisons have limited inferential resolution because only five project-level observations are available. The controlled training-label-noise analysis further shows progressive degradation as corruption increases, with SBR→NSBR errors primarily reducing detection and NSBR→SBR errors primarily increasing false alarms; however, this analysis does not constitute automatic label denoising. The present evaluation does not establish transfer to proprietary industrial software, embedded systems, cloud-native platforms, multilingual bug reports, or continuously updated deployment environments.

Future work will evaluate LLMSec using chronological train–test partitions and datasets drawn from proprietary industrial software, embedded systems, cloud-native platforms, multilingual bug-report repositories, and continuously arriving report streams. Although multilingual evaluation is identified as an important direction, no empirical multilingual validation is included in the present study. We will also investigate bug-report-specific representation models such as RepresentThemAll (Fang et al., 2023), instruction-tuned generative LLMs using zero-shot and few-shot prompt-based classification and prompt optimization, and full and parameter-efficient task-specific fine-tuning under the same datasets, data partitions, and evaluation measures. In addition, future work will examine deeper neural classification heads, cross-attention mechanisms, imbalance-aware objectives, project-specific BM25F tuning, and hybrid NSBR-selection strategies that combine lexical BM25F relevance with embedding-based semantic similarity. Additional work will investigate fragment-level explanations, developer-oriented operational evaluation, explicit label-denoising and confidence-aware learning, manually reviewed ambiguous-report subsets, and controlled degradation of incomplete or corrupted report content. Overall, the results demonstrate that combining relevance-guided rebalancing with contextual bug-report representations can improve automated SBR classification within the evaluated benchmark and controlled label-noise settings, while cross-domain generalization, multilingual applicability, task-specific adaptation, explainability, operational impact, and robustness to naturally occurring report-quality degradation remain to be established.

CRediT authorship contribution statement

Abhinav Jamwal: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Investigation, Data curation, Conceptualization. **Kunal Bansal:** Writing – original draft, Visualization, Validation, Software, Investigation. **Shreya Jain:** Writing – original draft, Visualization, Validation, Software, Investigation. **Sandeep Kumar:** Writing – review & editing, Writing – original draft, Supervision, Methodology, Funding acquisition, Conceptualization.

Funding

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The datasets used in this study are publicly available from the sources cited in the manuscript. The processed datasets are available at: https://github.com/ABhiIITRoorkee/LLMSec.

References

Alqahtani, S.S., 2024. Security bug reports classification using fasttext. Int. J. Inf. Secur. 23 (2), 1347–1358.

Beltagy, I., Peters, M.E., Cohan, A., 2020. Longformer: The long-document transformer. arXiv preprint arXiv:2004.05150.

Devlin, J., Chang, M.-W., Lee, K., Toutanova, K., 2019. Bert: Pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers). pp. 4171–4186.

Fang, S., Zhang, T., Tan, Y., Jiang, H., Xia, X., Sun, X., 2023. RepresentThemAll: A universal learning representation of bug reports. In: 2023 IEEE/ACM 45th International Conference on Software Engineering. ICSE, IEEE, pp. 602–614.

Gegick, M., Rotella, P., Xie, T., 2010. Identifying security bug reports via text mining: An industrial case study. In: 2010 7th IEEE Working Conference on Mining Software Repositories. MSR 2010, IEEE, pp. 11–20.

Goseva-Popstojanova, K., Tyo, J., 2018. Identification of security related bug reports via text mining using supervised and unsupervised classification. In: 2018 IEEE International Conference on Software Quality, Reliability and Security. QRS, IEEE, pp. 344–355.

Jiang, Y., Lu, P., Su, X., Wang, T., 2020. LTRWES: A new framework for security bug report detection. Inf. Softw. Technol. 124, 106314.

Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., Chen, Z., 2021. Sysevr: A framework for using deep learning to detect software vulnerabilities. IEEE Trans. Dependable Secur. Comput. 19 (4), 2244–2258.

Ma, X., Keung, J., Yang, Z., Yu, X., Li, Y., Zhang, H., 2022. CASMS: Combining clustering with attention semantic model for identifying security bug reports. Inf. Softw. Technol. 147, 106906.

Ohira, M., Kashiwa, Y., Yamatani, Y., Yoshiyuki, H., Maeda, Y., Limsettho, N., Fujino, K., Hata, H., Ihara, A., Matsumoto, K., 2015. A dataset of high impact bugs: Manually-classified issue reports. In: 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories. IEEE, pp. 518–521.

Omri, S., Sinz, C., 2020. Deep learning for software defect prediction: A survey. In: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops. pp. 209–214.

Peters, F., Tun, T.T., Yu, Y., Nuseibeh, B., 2017. Text filtering and ranking for security bug report prediction. IEEE Trans. Softw. Eng. 45 (6), 615–631.

Rainio, O., Teuho, J., Klén, R., 2024. Evaluation metrics and statistical tests for machine learning. Sci. Rep. 14 (1), 6086.

Shu, R., Xia, T., Chen, J., Williams, L., Menzies, T., 2021. How to better distinguish security bug reports (using dual hyperparameter optimization). Empir. Softw. Eng. 26, 1–37.

Shu, R., Xia, T., Williams, L., Menzies, T., 2019. Better security bug report classification via hyperparameter optimization. arXiv preprint arXiv:1905.06872.

Soltaniani, F., Ghafari, M., Sayagh, M., 2025. Security bug report prediction within and across projects: A comparative study of BERT and random forest. In: Proceedings of the 21st International Conference on Predictive Models and Data Analytics in Software Engineering. pp. 85–94.

Thissen, D., Steinberg, L., Kuang, D., 2002. Quick and easy implementation of the Benjamini-Hochberg procedure for controlling the false positive rate in multiple comparisons. J. Educ. Behav. Stat. 27 (1), 77–83.

Wang, L., Yang, N., Huang, X., Jiao, B., Yang, L., Jiang, D., Majumder, R., Wei, F., 2022. Text embeddings by weakly-supervised contrastive pre-training. arXiv preprint arXiv:2212.03533.

Wijayasekara, D., Manic, M., Wright, J.L., McQueen, M., 2012. Mining bug databases for unidentified software vulnerabilities. In: 2012 5th International Conference on Human System Interactions. IEEE, pp. 89–96.

Wilcoxon, F., 1992. Individual comparisons by ranking methods. In: Breakthroughs in Statistics: Methodology and Distribution. Springer, pp. 196–202.
Wu, X., Zheng, W., Xia, X., Lo, D., 2021. Data quality matters: A case study on data label correctness for security bug report prediction. IEEE Trans. Softw. Eng. 48 (7), 2541–2556.
Xia, X., Lo, D., Ding, Y., Al-Kofahi, J.M., Nguyen, T.N., Wang, X., 2016. Improving automated bug triaging with specialized topic model. IEEE Trans. Softw. Eng. 43 (3), 272–297.
Zaman, S., Adams, B., Hassan, A.E., 2011. Security versus performance bugs: a case study on firefox. In: Proceedings of the 8th Working Conference on Mining Software Repositories. pp. 93–102.
Zheng, W., Xun, Y., Wu, X., Deng, Z., Chen, X., Sui, Y., 2021. A comparative study of class rebalancing methods for security bug report classification. IEEE Trans. Reliab. 70 (4), 1658–1670.