Improved Labeling of Security Defects in Code Review by Active Learning with LLMs

Johannes Härtel
j.a.hartel@vu.nl
Vrije Universiteit Amsterdam
Netherlands

Abstract

Mining high-quality datasets of security defects is important for cybersecurity. In this paper, we focus on mining a dataset of reviews that discuss potential security defects in code or other artifacts. Mining such datasets often involves labeling, and this is challenging because security defects are rare.

We investigate the use of active learning with a fine-tuned large language model to make the mining and labeling of such datasets more effective. Our simulations demonstrate that active learning can increase the effectivity of human annotators by a factor of 13. This means we can produce datasets with 13 times more defects than found in random samples of the same size. We conducted an empirical study on over four million unlabeled reviews from GitHub, showing that active learning increases the effectiveness by a factor bigger than 6. In total, 246 out of 1298 labeled reviews can be identified as discussing security defects. We do not depend on a keyword list for upfront candidate selection but dynamically evolve an LLM for this.

Our work holds the potential to inspire future research in this area, resolving rare class and imbalance problems at the root where they appear, by adjusting the mining and labeling of the datasets. Our final dataset and model are publicly available.

Keywords

Security defects, simulation, active learning, code review, dataset curation, cybersecurity, manual labeling, empirical study.

1 Introduction

Mining high-quality datasets of security defects is important for cybersecurity [5]. In this paper, we focus on mining a dataset of code reviews that discuss potential security defects, as shown in Figure 1. These reviews can serve as a proxy for detecting actual security defects in the code hereafter. However, mining often involves labeling the security defects. This is challenging because security defects are very rare [1, 7, 23]. They are so rare that when randomly labeling within the capabilities of a human, it is likely to find nothing. We labeled 100 random reviews from our target dataset of over four million unlabeled reviews from GitHub and found nothing.

Examples of the Problem. Related work reports the same problem. The authors of [7] say that *'Being a manual effort, we could not inspect the entire initial dataset.'* Consequently, they attempted random sampling, which proved ineffective, and eventually switched to a keyword-based filter to select candidates for the manual labeling. Keyword-based filtering is a common approach to make the labeling more effective, but it also limits the final dataset by the filter. Table 1 summarizes numbers for the related work.

Table 1: Our Contribution: No keyword list but still identification rates acceptable for manually labeling of code reviews.

Study	Total ($\mathbb{R}$) Reviews	Filtered by Keywords	Labeled	Identfication Rate (relative)	(total)
[23]	432,585	yes	20,995	2.6%	614
[7]	60,655	yes	1,155	6.1%	71
[1]	38,004	yes	882	19.0%	171
[18]	?	yes	?	?	516
Our data	4,191,892	no	1298	19%	246
Our sim.	1,000,000	no	4,000	1.7% - 23.2%	

Research Questions. We investigate the use of *active learning* [21, 22] with a *fine-tuned large language model (LLM)* [14] to make the mining and labeling of such datasets more effective. We aim to avoid the limitations of a keyword list for candidate selection but keep its benefits, saving human resources during labeling.

- **RQ1: Can active learning with LLMs save human resources when labeling reviews on security defects?**
- **RQ2: Can we avoid a keyword list by dynamically training an LLM for candidate selection instead?**

Short Answer. We show that active learning with an LLM addresses the limitations of candidate selection dynamically, selecting the most informative instances for labeling as the classifier evolves.

Our simulations demonstrate that active learning can increase the effectivity of human annotators by a factor of 13, from 1.7% to 23.2%, producing datasets with 13 times more defect labels than in random samples of the same size.

Guided by these simulations, we conducted an empirical study on an unlabeled target dataset of over four million reviews from GitHub with a conservative estimate of the average probability of security defects being below 2.8%. We demonstrate that with active learning, 246 out of 1298 labeled reviews can be identified to discussing security defects. This corresponds to an identification rate of 19% and increases human labeling effectiveness by a factor bigger than 6. We do not use a fixed keyword list but evolve an LLM dynamically.

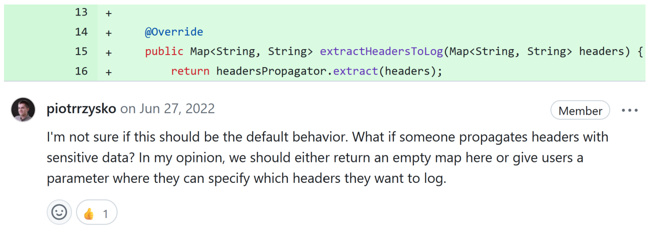
Scope and Related Work. The rareness of security defects often causes imbalanced datasets. **This paper proposes and evaluates a process to mine and label a dataset that is less imbalanced**. We thereby resolve the problem of imbalance at its root. We do not suggest a method to rebalance an existing dataset, nor try to better train on imbalanced data, outperforming other classifiers. Such ideas solve different problems that follow, making a comparison with the corresponding state-of-the-art unsuitable.

Previous work that creates datasets has explored active learning, partially with LLMs, in various contexts, but not specifically for security defects in code reviews [4, 10, 20, 24, 25]. On the other hand, research focused on security defects discussed in code reviews without employing active learning with LLMs includes [1, 7, 8, 18, 23]. This paper bridges the gap between these two areas.

Data Availability.

- **DOI:** https://doi.org/10.6084/m9.figshare.28303904
- **GitHub:** https://github.com/johanneshaertel/EASE_2025_active_learning_LLM

Roadmap. The paper starts with a motivation for detecting security defects in code reviews in Sec. 2. We then provide background information on active learning in Sec. 3. Next, we describe our study design to evaluate active learning in Sec. 4. We present our simulation results in Sec. 5 and the empirical study in Sec. 6. We discuss a final quality assurance of our dataset and the threats to validity in Sec. 7 and 8. We discuss related work in Sec. 9 and conclude in Sec. 10.



```
+
+    @Override
+    public Map<String, String> extractHeadersToLog(Map<String, String> headers) {
+        return headersPropagator.extract(headers);
```

piotrrzysko on Jun 27, 2022

I'm not sure if this should be the default behavior. What if someone propagates headers with sensitive data? In my opinion, we should either return an empty map here or give users a parameter where they can specify which headers they want to log.

Figure 1: What do we search for? The review by *piotrrzysko* discusses a potential security defect on confidentiality.

2 Motivation

In this paper, we focus on mining a dataset of code reviews that discuss potential security defects, as shown in Figure 1. The core challenge in this mining process is the resource-intensive nature of manual labeling. We now motivate why and how we address this challenge by *active learning* and *large language models* (LLMs).

2.1 Labeling Security Defects

Several works classify and label security defects in software-related artifacts [3, 9, 19, 27, 28]. Typically, these studies begin by identifying a set of unlabeled observations, denoted as $\mathbb{R}$, where they suspect important information on security defects. This can include code, commit messages, issues, pull requests, or reviews.

Subsequently, authors label these observations as either related to security defects or not. Depending on the exact information searched, we might face different labeling guidelines, as exemplified by the open coding in [23]. The process of labeling mostly ends with a categorical decision for each or some of the observations in $\mathbb{R}$. The labeling of our study is comparable to [1, 7, 8, 18, 23], which search for text on potential security defects in code reviews.

Eventually, this labeled data is used to train a model that can predict if a new observation includes a security defect or not [5, 16]. Such models can be integrated into tools, such as IDEs, to automatically warn of security defects, or to continuously monitor open-source data on GitHub. The quality of labels is crucial [5].

2.2 Labeling Security Defects in Code Reviews

In this work, we focus on a specific type of data: *natural language code reviews*. We check if such reviews discuss potential security defects in code or other artifacts, as previously explored in [1, 7, 8, 18, 23]. We use the term ‘potential’ because a review does not necessarily provide enough context to prove the existence of a security defect; rather, they discuss why something might be a security defect. These reviews can serve as a proxy for detecting actual security defects in the code hereafter.

We show an example from our empirical study in Fig. 1, illustrating the problem of logging sensitive information. The arguments around such leakage are crucial and may be used beyond code.

2.3 The Problem: Security Defects are Rare

Manual labeling remains the predominant practice for natural language datasets like the one we focus on [1, 7]. However, Biase et al. [7] highlighted the inefficiency of manually labeling defects in randomly selected candidates. They eventually switched to a keyword-based selection of candidates.

This inefficiency comes from the fact that security defects are rare. In general, this is referred to as a *rare-class*, *rare-label*, or *imbalance* problem. In our study, we conducted a simple experiment to show this. We labeled 100 random candidates and detected no security defects. Based on this, we estimate that the average probability of defects must be below 2.8% in our target dataset. The statistical model used for this estimate is available online.

Given our goal to mine a dataset rich in security defects and the extremely low probability of encountering them below 2.8%, scaling the labeling process to manageable dimensions for humans becomes impractical. Thus, we require an approach to increase the *identification rate* of defects to a more acceptable level.

2.4 Candidate Selection

The typical approach for natural language is to reduce to candidates selected by keywords, excluding or including observations with specific terms. See Table 1 for examples. This narrows down a huge unlabeled dataset $\mathbb{R}$ into a smaller *‘candidates’* subset $\mathbb{R}_C$ with hopefully a higher average probability of defects, suitable for being labeled manually. Identification rates can rise to acceptable levels of 6.1% or 19%, as reported by related works shown in Table 1.

To give an impression of keywords, we provide those used by Biase et al. [7]: *buffer, cast, command, cookie, crypto, emismatch, exception, exec, form, field, heap, injection, integer, ondelete, out of memory, overflow, password, printf, privilege, race, random, sanitize, security, sensitive, sql, URL, use-after-free, vulnerability, xhttp, xml.*

Authors acknowledge that the use of these keywords is a threat to the study design, as it may result in missing security defects, if the keyword list is too narrow. If the keyword list is too broad, identification rates get low, and the labeling effort becomes excessive.

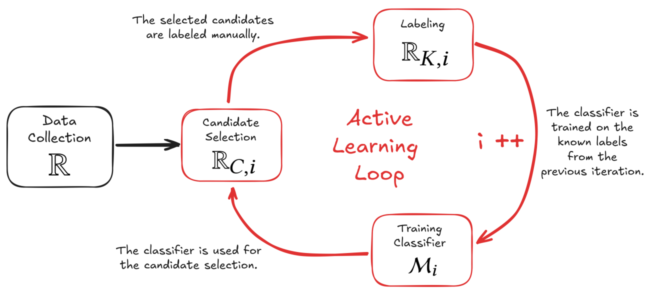

The selected candidates are labeled manually.
Labeling
$\mathbb{R}_{K,i}$
Data Collection
$\mathbb{R}$
Candidate Selection
$\mathbb{R}_{C,i}$
Active Learning Loop
i ++
The classifier is trained on the known labels from the previous iteration.
The classifier is used for the candidate selection.
Training Classifier
$\mathcal{M}_i$

Figure 2: Active Learning

2.5 Dynamic Candidate Selection by Active Learning

Keyword lists are therefore often refined iteratively, which can be considered an instance of *active learning* as shown in Fig. 2. It is not always called like this and there are many variants for the refinement of a keyword list, see the examples [1, 23].

At its core, we don't need to know all the keywords in advance. The term *'password'*, for instance, might frequently be used together with *'sensitive'*, so we can add one term to our keyword list if we encounter another. It is a form of correlation that we use to refine the keywords iteratively. On the contrary, if a term leads to many false positives, we exclude it.

In general, these keyword lists are classifiers that we use to select candidates for labeling. After each labeling iteration, we improve the classifier by 'retraining' it on the labels obtained so far and repeat the process. We may also use more advanced versions of this classifier, as explored in this paper.

2.6 Improved Candidate Selection by LLMs

Such keyword lists are a pre-LLM method. They are inherently limited in capturing the complexity of natural language. For example, they mostly operate on bags and not on sequences of words.

In this paper, we replace them by dynamically fine-tuning a language model, creating a classifier with several advantages over the keywords. Our approach captures nuanced and complex patterns that cannot be encoded as keywords. Additionally, we benefit from the language model's ability to transfer knowledge. For instance, the co-occurrence of terms like *'password'* and *'sensitive'* might already be understood by the base model, even without analyzing a single security defect.

We illustrate the evolution of our classifier in Table 2. For instance, a review (R2) saying *'This code might provide access to sensitive data.'* was initially not classified correctly as a review discussing a security defect. However, after seven iterations and the classifier learning from the feedback loop, the review was accurately classified. We achieved this without any manual intervention on the specifics of the classifier, simply by labeling the selected candidates on demand of the classifier.

3 Background

We now provide a formal overview of active learning, which we use to enhance the labeling of security defects. Active learning involves methods that iteratively train a classifier with, in our case, human feedback [21, 22]. We denote the entire dataset as $\mathbb{R}$, representing the unlabeled reviews. The labels, which are costly to obtain, are produced on demand through interaction with a human.

Table 2: Predictions by our evolving LLM classifier over the active learning iterations.

Review Text	Iterations LLM Classifier						
	$\mathcal{M}_1$	$\mathcal{M}_2$	$\mathcal{M}_3$	$\mathcal{M}_4$	$\mathcal{M}_5$	$\mathcal{M}_6$	$\mathcal{M}_7$
	(Does the review discuss a pot. security defect?)						
R0: Can we refactor this code to make it maintainable?	0%	7%	2%	0%	0%	0%	0%
R1: This code enables an attacker to get access to our data.	0%	45%	**89%**	6%	**95%**	**84%**	**93%**
R2: This code might provide access to sensitive data.	0%	34%	**76%**	4%	**95%**	**81%**	**91%**
R3: This might allow someone to run denial of service.	0%	**53%**	23%	2%	11%	**77%**	**80%**
R4: We might expose credentials.	0%	**58%**	**92%**	1%	42%	**74%**	**89%**

Algorithm 1 Standard Active Learning Process

```
1:  Initialize $\mathbb{R}_{K,0}$ with an initial labeled dataset
2:  Initialize $\mathbb{R}_{U,0}$ with the remaining unlabeled dataset
3:  for each iteration $i = 1, 2, \ldots$ do
4:      Train model $\mathcal{M}_i$ on $\mathbb{R}_{K,i-1}$ (or take a bootstrap model)
5:      Predict labels for $\mathbb{R}_{U,i-1}$ using $\mathcal{M}_i$
6:      Select candidates $\mathbb{R}_{C,i}$ from $\mathbb{R}_{U,i-1}$ by selection metric
7:      Obtain true labels for $\mathbb{R}_{C,i}$ by manual labeling
8:      Update $\mathbb{R}_{K,i} = \mathbb{R}_{K,i-1} \cup \mathbb{R}_{C,i}$
9:      Update $\mathbb{R}_{U,i} = \mathbb{R}_{U,i-1} \setminus \mathbb{R}_{C,i}$
10: end for
```

3.1 Feedback Loop

We present the pseudocode for the active learning process in Algorithm 1. Active learning operates iteratively, determining in each iteration the subset of $\mathbb{R}$ that will be the next candidate for labeling. A classifier is employed to select new candidates. This classifier $\mathcal{M}$ is trained on the already labeled data, which is updated in each iteration. The newly labeled data is added to the dataset used to train the classifier for the next iteration.

In each iteration i, we identify the following subsets:

- $\mathbb{R}_{K,i}$: The labeled data, where the label is **known** and used to train the classifier.
- $\mathbb{R}_{U,i}$: The unlabeled data, where the label is **unknown**.
- $\mathbb{R}_{C,i}$: The data **candidates for labeling** in the iteration.

In certain cases, we may employ a *bootstrap model* to select candidates for the first iteration, rather than relying on an initial labeled dataset $\mathbb{R}_{K,0}$.

Table 3: Hyperparameter candidate selection

Candidate Selection	Metric (p_j)
Entropy: We select candidates based on the entropy of the classifier's prediction.	$-\sum_{l \in L} p_j(l) \log p_j(l)$
Rare-label: We select those instances where the classifier is most certain to be of the rare label.	$p_j(l_{\text{rare}})$
Majority-label: For symmetry, we select those instances where the classifier is most certain to be of the majority label.	$p_j(l_{\text{majority}})$
Random: As a baseline, we also examine random sampling. This method does not involve the classifier and selects the candidates randomly. We denote this by the 'hash' of the instance.	$\text{hash}(p_j)$

3.2 Classifier Training

Active learning needs to train a classifier on the previously labeled data. Since we are working with natural language, we fine-tune a language model for our empirical study. For the simulation, we simplify and restrict to the classification head of the LLM, using a basic feed-forward neural network for classification.

3.3 Candidate Selection

Given the classifier $\mathcal{M}_i$ trained on data from iteration $i-1$, there are various strategies to select the next candidates for iteration i. Next to basic exclusion and inclusion mechanisms, active learning allows us to explore options based on uncertainty. In simple terms, we can select the instances where the classifier is most uncertain.

Table 3 enumerates such strategies where p_j is the prediction of the classifier $\mathcal{M}_i$ for instance j, which is a probability distribution over the labels L. The classifier predicts the labels of the instances with unknown labels in $\mathbb{R}_{U,i-1}$, which are then filtered by one of the selection metrics to form the next candidates $\mathbb{R}_{C,i}$. We select among the top k instances when sorted by metric.

4 Hybrid Methodology

To evaluate the active learning method-design, we adopt a hybrid methodology, combining a simulation followed by an empirical study to examine design options (also see [15] and [12]).

Our empirical study works with a single human, the author of this paper, labeling the reviews from GitHub. This is a real but unknown *data generating process*. We can see it as a single function that produces a label given a review. Our simulation does the same, but randomly creates possible functions that map inputs to labels. It thereby creates many random *data generating processes* corresponding to possible human judgments. We motivate this as follows:

- **Transparency**: Our simulations capture the data generating processes as code. This is very transparent compared to a single real-world labeling by a human. It supports a deeper conceptual understanding of our work, especially regarding generalization, reproduction, and modification, and thereby facilitates constructive criticism in future studies.
- **Controllability**: While typical methodology to evaluate method-design operates with hyperparameters specific to the method, we can examine 'hyperparameters' specific to the problem. We can adjust parameters of the data generating processes, such as label predictability or correlations, experimenting in ways that are impossible in a real-world setting.
- **Scalability**: The simulation fundamentally differs in how it scales. The simulation allows repetition of experiments across different data generating processes. It is not just limited to folds over a single real-world labeling by a human. For evaluating design decisions, we think this is crucial. Simulations repeat until confidence intervals sufficiently converge, ruling out randomness in the conclusions.
- **Plausibility**: However, we clearly face the simulation-gap in our methodology in that the simulation needs to be a plausible representation of the real-world problem. Not all insights might be transferable between the simulation and the empirical study. We highlight differences later and also discuss this problem in a section on threats to validity.

Informed by the simulation, the empirical study demonstrates the most promising configurations of active learning in a real setting, evaluating the method through real and not simulated labeling.

5 Simulation Study

We begin with a simulation study to guide our following up application of active learning in a real-world setting. We present the core assumptions of our simulation in this section. The technical part, including all the details, can be found online. The simulations are written in Python and use TensorFlow and Keras.

5.1 Problem Hyperparameters

Our data generating process is the human giving a label. We start with the hyperparameters specific to this labeling problem we aim to solve. We cannot influence such parameters in reality but using the simulation we can prepare for their implications.

We align this simulation with the classification head of the LLM, using a simple random feed-forward neural network. We make one of the two output labels a rare class by adjusting biases. We simulate the unlabeled dataset $\mathbb{R}$, input to the classification head, using a multivariate normal distribution generating one million observations. The input $\mathbb{R}$ is available to the active learning method. The labels are missing but can be simulated running the random data generating process (our artificial human) on input data.

- **Problem Structure**: We assume that the labels are generated by a neural network with random weights applied to random normally distributed input data. We use 20 input variables and a neural network with 4 randomly assigned layers, where sizes are 15, 15, 15, and 1. The last layer is sent through the sigmoid function. We limit the simulation to two categories and sample the labels from a Binomial distribution.

- **Imbalance**: We produce imbalance by modifying the last layer of the network. In our experiments, we target an average imbalance of around 1.7%, ranging between 0% to 4% in repeated simulation runs. For our target data from GitHub, we know that the imbalance must be below 2.8%.
- **Label Predictability**: To produce a decision that is not purely random but predictable, we systematically modify the standard deviation of the logits before sending them through the sigmoid function and to the Binomial random number generator. We scale to a standard deviation of 0.7, 1.0, or 1.3. We later refer to this as low, normal, and high predictability. We point out that future work should explore the impact of *missing variables* which we missed.
- **Correlation**: We explore the impact of a correlation structure on the input data. To simulate correlation, we use the Cholesky decomposition of a random correlation matrix, that is multiplied with the random input data.

5.2 Methods Hyperparameters

The next hyperparameters are specific to the applied methods. We explore four candidate selection methods described in Sec. 3.3, including the baseline of selecting candidates randomly. This is the most common subset of candidate selections also explored in [10, 20, 24, 25]. Additionally, we examine two parameters indirectly related to candidate selection.

- **Iteration Increment**: In each iteration, a fixed number of candidates is selected and manually labeled. While this step size does not directly influence the labeling effort, it might affect identification rates and thereby the effectiveness of labeling. We examine adding 40 or 200 new observations each iteration, up to reaching 4000 observations in total.
- **Epochs for Model Fitting**: We explore the impact of model fitting in terms of the number of epochs used. We test 5, 10, and 30 epochs to determine if early stopping is necessary to avoid overfitting.

5.3 Evaluation Metrics

We do the comparison of methods in terms of identification rates of security defect labels, which is our the simulated rare class. This is the relative number of security defect labels in the dataset of labeled observations until this point. It allows us to determine and compare the effectiveness of each method throughout the iterations, relative to the number of labeled candidates. We also measure the loss of the model on the entire dataset $\mathbb{R}$.

5.4 Results

We now discuss the central guidance derived from the simulation study. Throughout this section, shaded areas represent 90% confidence intervals for the given values. These intervals result from multiple simulation runs. We only focus on the number of security defect labels that is our rare class. The number of non-security defect labels is complementary for two simulated categories. Whenever applicable, the plots indicate the number of simulation repetitions in the bottom-right corner.

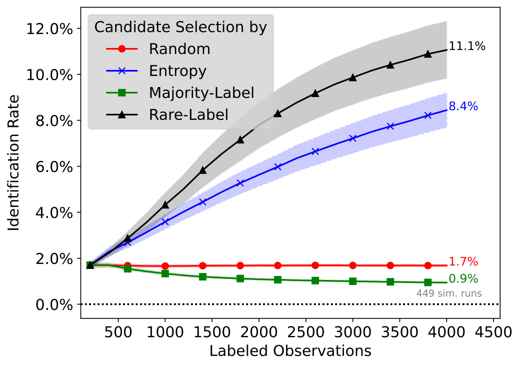


Figure 3: Hyperparameter candidate selection

5.4.1 *Candidate Selection.*

We start with the most important question: can any of the candidate selection methods outperform the baseline of random sampling? Figure 3 illustrates the identification rate as the number of labeled observations increases through active learning iterations. We compare different candidate selection methods. Note that iterations corresponds to the number of labeled observations.

- **Random:** Recall that we have an imbalance, with around 1.7% of $\mathbb{R}$ being instances of the rare-label. Random sampling converges towards an identification rate of 1.7%. This is expected and disqualifies random sampling for our purpose.
- **Majority:** Actively sampling for the majority label performs even worse. We include it for symmetry reasons.
- **Rare-lables and Entropy:** Selecting candidates by sampling for the rare security defect labels or for entropy clearly works. After 4000 observations are added to the dataset by the active learning iterations, rare-label sampling results in an overall identification rate of 11.1%, and entropy sampling results in an identification rate of 8.4%. Both are significantly better than random sampling.

We emphasize that the identification rate evolves over the iterations, which is also shown in the plots. Initially, when the model is not yet trained, the identification rate is low. As the model improves, the identification rate increases.

In the empirical study, this insight guides us to use a mixture of both candidate selection methods (entropy and rare-label).

5.4.2 *Correlation.*

We observe that correlation slightly affects identification rates, but active learning methods remain effective regardless of the presence of correlation. Our confidence intervals are still too large to make a definite statement on what is better. We refer to the corresponding plots and data that can be found online.

5.4.3 *Iteration Increment.*

Fig. 4 shows the impact of the iteration increment, which is the number of new candidates we label in each iteration, on the identification rate. We observe a slight performance difference depending on the increment step size. Smaller step sizes appear to be better, as they lead to faster feedback and better decisions on what to label next. However, there is a tradeoff,

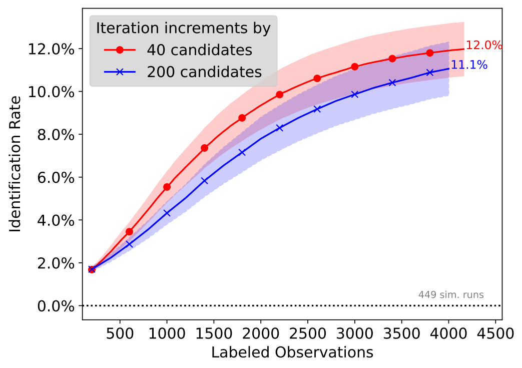


Figure 4: Hyperparameter iteration increment

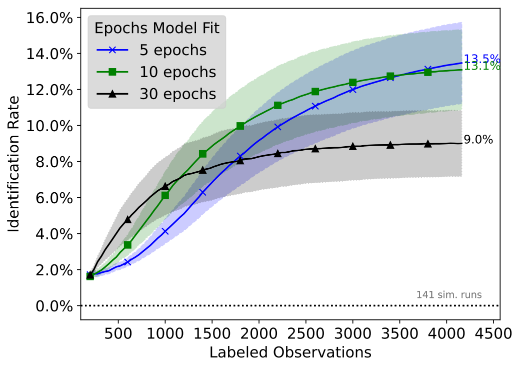


Figure 6: Hyperparameter epochs

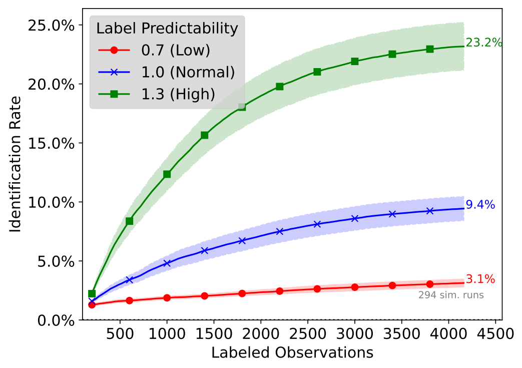


Figure 5: Hyperparameter label predictability

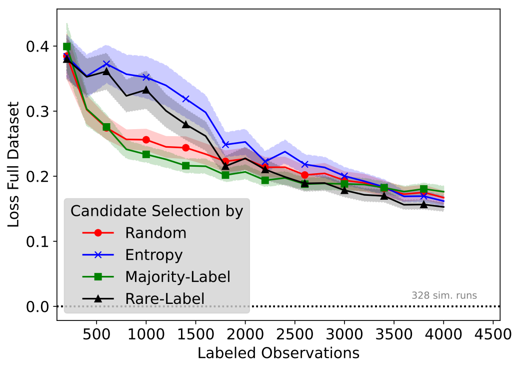


Figure 7: Hyperparameter candidate selection for loss on $\mathbb{R}$

as each iteration can incur a high cost for fitting and applying the classifier to a huge $\mathbb{R}$.

In the empirical study, this guides us to avoid increment sizes much larger than 200.

5.4.4 Predictability. Figure 5 shows the impact of label predictability on the identification rate. The plot demonstrates that predictability significantly affects the performance of active learning methods. With low predictability, active learning performs almost similarly to random sampling, with 3.1%. Conversely, with high predictability, the method achieves identification rates up to 23.2%.

In the empirical study, this insight guides us to continuously assess predictability. We evaluate the model in each iteration to ensure its efficiency. In cases of low predictability, we may need to adjust the model or the input data to improve performance.

5.4.5 Model Fitting. Model fitting can play a crucial role, which we examine by varying the number of training epochs. Figure 6 shows the impact of epochs on the identification rate for the rare-label candidate selection method and without correlation present. We point out that there appears to be an interaction with such problem hyperparameters, which we did not yet explore. We refer to the online data.

The results show the typical overfitting and underfitting phenomena. With 10 epochs, we achieve an overall good performance. With 5 epochs, we underfit in the first iterations, but then the method improves with a steep increase in the identification rate around 1500 observations. With 30 epochs, we perform well in the first iterations, but then overfit.

Cross-validation can help assess overfitting. While we used early stopping by an adjusted number of epochs, other methods to prevent overfitting, such as regularization, dropout, or reducing model size, may also be effective. Our results also leave space for future adjustments of epochs as a function of the number of iterations.

In the empirical study, this complexity guides us to evaluate the model on a separate test set in every iteration, as is common in cross-validation, and to avoid relying solely on the training loss.

5.4.6 Loss Full Dataset. Last, we examine the loss of the evolving model on the entire dataset $\mathbb{R}$ of one million observations in Figure 7. We limited to problems with a high predictability, as there appears to be another interaction effect that we did not yet explore in depth.

The insights related to this plot are surprising, as the candidate selection methods that recover the most rare-labels, do not cause the lowest overall loss in the first iterations. The plot indicates that if we aim to minimize the overall loss, we should start selecting the majority labels. However, the loss of all candidate selection methods converge after sufficient iterations. We cannot tell yet because confidence intervals are too wide, and we suspect model fitting epochs to also impact these results, but there is the possibility that in later iterations the rare-label selection method might be better. This calls for further investigation.

6 Empirical Study

Informed by the simulation, we now turn towards reality, where we evaluate the best configurations of active learning we found to label real security defects discussed in code reviews from GitHub.

6.1 Data Collection and Processing

We collect pull requests and their reviews using the GitHub API, referring to this raw data as $\mathbb{R}$.

To achieve this, we select random repositories from GitHub and download all pull requests and reviews for each repository. We navigate around well-known technical limitations of the GitHub API (see [6]). The query used to select a repository randomly is limited by filters. A repository needs to have i) a minimum of 100 stars to ensure quality, ii) a minimum of 10 pull requests to ensure active usage of pull requests, and iii) use Java as the main language, which we currently focus on. Such filters are typical [2, 6]. We collect and store all pull requests and reviews for the repository as line-delimited JSON. We have collected 4,191,892 pull request reviews from 4,937 distinct repositories and 492,266 distinct pulls. For the empirical study, this data is limited to reviews attached to a Java file. However, the online material contains all crawled reviews.

6.2 Labeling Guidelines

The author of this paper who has a standard background in software security performed the labeling to ensure consistency and maintain iteration speed. The labeling of the candidates for iteration one to seven was done in the time span of May to July 2024. A second quality assurance round of was conducted in April 2025 (see Sec. 7).

The labeling instructions are provided in Table 4, which lists the six labels used in the study along with their descriptions. Previous studies on security defects in code reviews use closely related labels, like in [24], asking '*Whether the review comment is security-related*'.

Table 4: Labeling Guidelines

Label	Description
Potential Security Defect	The review discusses a potential security defect of code or other artifacts.
No Security Defect	The review does not discuss a security defect of code or other artifacts.
Unclear	The relation to a security defect is unclear.
Broken	The review is a broken link. This is a technical problem with some of our input data.
Bot	The review is written by a bot.
Non-English	The review is not written in English.

6.3 Classifiers

6.3.1 Bootstrapping Classifier. In the empirical study, we use a bootstrapping classifier $\mathcal{M}_1$, a simple regex-based tool similar to a keyword list. It produces the candidates for the initial labeled dataset $\mathbb{R}_{K,1}$ needed to train model $\mathcal{M}_2$. This initial dataset is essential for training our first LLM for active learning.

We intentionally designed this bootstrap classifier to be basic. It searches only for the keyword *secur* in the review text, capturing terms like *security*, *secure*, *insecure*, *securely*, and *insecurely*. The exact implementation is available in the online material. We do not aim for an exhaustive keyword list, as this is the task for the subsequent active learning loop with an LLM.

6.3.2 Main Classifier. After bootstrap, we fine-tune RoBERTa [14], the *base* variant, an existing language model for sequence classification with parameters occupying around 500 MB of GPU memory. The input to the model is the tokenized text of the review up to 64 tokens. The output is one of the labels shown in Table 4 where broken links and bots are excluded as they can hardly be predicted.

6.3.3 Loss and Optimizer. We use standard categorical cross-entropy as the loss function for fine-tuning the LLM [11]. Adam [13] is used for optimization.

6.3.4 Model Configuration and Performance. The simulation guides us to carefully inspect model performance and prevent overfitting and underfitting. To tailor this insight to our used model architecture, which is different from the simulation, we explore different learning rate and epoch combinations. We conducted these additional experiments before running the actual empirical study using our target model architecture, but another related dataset borrowed from [19]. Learning rates around $1e-6$ and 40 epochs were successful. Gray literature also recommends low learning rates for fine-tuning language models. Using Adam is standard practice.

As a sanity check for the hyperparameters, we apply cross-validation in every iteration. Our simulation guided us to do this. Test metrics over epochs while fitting $\mathcal{M}_2$, $\mathcal{M}_4$ and $\mathcal{M}_6$ are shown in Figure 8. Each line depicts the progress while fitting a single model, showing the typical U-shaped loss curve when evaluating a model out-of-sample. We show the test loss and the test precision (of potential security defect) at k where k is the number of potential security defect labels in the test set. For the iterations shown, we stop fitting the model after 40 epochs. The plot shows no signs of overfitting or underfitting. Moreover, we see clear signs that the models successfully learn, by a strong decrease in loss and a strong increase in precision. This also holds for the iterations not shown.

As a side note, since we don't optimize hyperparameters at this point, we don't hold out a validation set (see Goodfellow et al. [11], page 119, first paragraph). We do a single fold, splitting the dataset with known labels $\mathbb{R}_{K,i}$ into test (30%) and train set (70%).

6.4 Candidate Selection and Step Size

The simulation suggests using rare-label or entropy candidates. In the empirical study iterations, we experimented with a combination of both methods. The specific candidate selection used in each iteration is shown in Figure 9 as the shaded background.

In the initial iteration, we relied on the bootstrap classifier to select candidates, which means neither entropy nor rare-label selection was applied. For iterations 2 to 5, we employed a mixture of entropy and rare-label candidates. In the final iterations, 6 and 7, we exclusively used rare-label candidate selection.

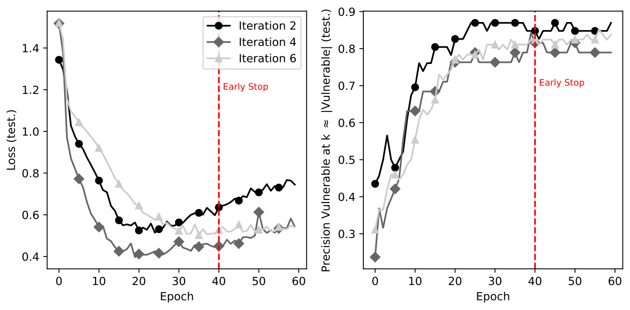


Figure 8: Model test set performance for $\mathcal{M}_2$, $\mathcal{M}_4$ and $\mathcal{M}_6$.

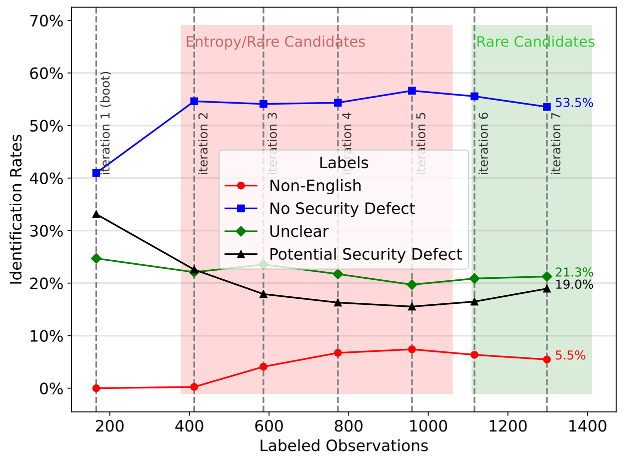


Figure 9: Identification rates for different labels as lines.

We decided on a step size of around 200 candidates per iteration, ensuring it is not much larger. The step size is shown in Figure 9 by the distances between the lines indicating each iteration. Broken links and bots are excluded from our considerations.

6.5 Results

In total, we labeled 1298 reviews in seven iterations (excluding the 122 reviews that are broken links). We provide the labeled data online. We structure the discussion along our research questions.

6.5.1 RQ1: Can active learning with LLMs save human resources when labeling reviews on security defects? We found 246 reviews discussing security defects, 695 reviews that clearly do not, 276 reviews where the relationship to a security defect is unclear, 10 reviews written by bots, and 71 reviews that are not written in English. For 122 broken reviews, we could not navigate to the pull request review on the GitHub webpage. These reviews are not included in our analysis.

This is an identification rate of 19% for reviews on security defects. If we put this into relation to our estimate of the average probability of security defects being below 2.8%, this is an increase in human labeling effectiveness by a factor of more than 6.

We emphasize that identification rates of 19%, which means every five reviews we find one defect, is a reasonably good result for a rare-label problem and allows manual labeling. From related work like [23], we know that rates without keyword-based preselection, but with random sampling, can fall below 1%. We conclude that active learning with LLMs can reduce the human effort.

Regarding the different iterations, we observe the following trends in Figure 9:

- **Bootstrapping**: In iteration 1, the bootstrapping classifier delivers good results. Searching for the keyword *'secur'* is effective in identifying security defects. This provides a solid starting point but is limited in scope.
- **Iteration 2**: In the first active learning iteration, we apply the model trained on the bootstrapping data for candidate selection. We use a mixture of entropy and rare-label sampling. This early model is not mature, leading to an increase in the majority label *no security defect*. Notably, the model does not yet predict non-English reviews because it did not encounter any in the bootstrapping data. Consequently, no candidates are selected with uncertainty about the language. In $\mathbb{R}_{K,2}$, we add a single non-English label, marking the first occurrence of this label.
- **Iteration 3**: In iteration 3, we observe a peak in language-related labels due to the high uncertainty measured by entropy for language candidates by $\mathcal{M}_2$. This iteration labels 27 *non-English* candidates. Identification rate of *potential security defect* labels drop further, but the momentum is reduced.
- **Iterations 4 and 5**: In iterations 4 and 5, the *potential security defect* labels start to plateau. Language-related labels continue to increase.
- **Iterations 6 and 7**: In iteration 6, we switch to rare-label sampling only, since entropy sampling tends to select too many language-related candidates. This change results in a clear increase in *potential security defect* labels and a decrease in *no security defect* and *non-English* labels.

Our conclusion is that rare-label candidate selection is more efficient if competing with entropy as we used it. This is because entropy might also favor other labels in a multi-class classification problem.

6.5.2 RQ2: Can we avoid a keyword list by dynamically training an LLM for candidate selection instead? Our approach is not limited to a predefined set of keywords but evolves a classifier from bootstrapping by $\mathcal{M}_1$ to a learned LLM classifier $\mathcal{M}_7$ over seven iterations. The question remains whether the classifier $\mathcal{M}_7$ surpasses the bootstrapping classifier $\mathcal{M}_1$. The short answer is 'yes', only 59% of our entire dataset can be identified by the bootstrapping classifier.

An illustrative example from the second iteration is (the same as in Figure 1): "*I'm not sure if this should be the default behavior. What if someone propagates headers with sensitive data? In my opinion, we should either return an empty map here or give users a parameter where they can specify which headers they want to log*" (from repository *allegro/hermes*, pull 1446). This indicates that the LLM classifier understands sensitive data and leakage in logging from the previous bootstrapping iteration, even if the keyword *'secur'* that we used for bootstrapping is not in the text.

We examine this phenomenon on example review texts as shown in Table 2 from the motivation section.

- Review **R0** shows a clear *no security defect* label. The classifier for iteration 2 still struggles, but after the second iteration, classification improves.
- Reviews **R1**, **R2**, **R3**, and **R4** demonstrate the classifiers' ability to generalize to new keywords like 'attacker', 'sensitive data', 'denial of service', and 'exposing credentials'. None of these are detected by the bootstrapping classifier.

R1 to **R4** provide evidence that the LLM evolves over the iterations and exceeds the capabilities of the bootstrapping classifier.

7 Quality Assurance and Afterthoughts on 'bigger' LLMs

After conducting the empirical study, we added a quality assurance stage to double-check the labels for the final dataset.

For this, we used another LLM, Claude Sonnet version 3.5, to produce alternative zero-shot classifications for the 1217 reviews labeled as *potential security defect*, *no security defect*, or *unclear*. Disagreements were manually reviewed and resolved by the author of this paper. The code running Claude applies sequential and parallel test-time scaling [26], using our labeling guidelines as the system prompt and the review as input. The code calling Claude and the dataset, including Claude's reasoning, are available online.

This paper reports on the numbers after quality assurance, though the differences are minimal. In total, 189 potential security defect labels remained unchanged by this quality assurance. Complete details can be reviewed in the final dataset including pre- and post-quality-assurance labels.

We note that Claude performed surprisingly well. However, scaling its classification to the entire 4-million review dataset seems infeasible. Extrapolating our cost of approximately 0.04 USD per Claude classification is 160,000 USD for the full dataset. As an afterthought, Claude might potentially replace the human in our loop. However, it cannot replace the active learning with a relatively cost-efficient 'smaller' LLM (we used RoBERTa-base) for scaling candidate selection to the full 4 million reviews. For one iteration with 4 million classifications, we only paid around 2 USD on AWS.

8 Threats to Validity

The simulation-gap is one of the central threats to our hybrid methodology. The simulations of the input data, and the simulation of the function between input and label, might not represent the real-world data generating process. However, neural networks are capable of expressing complex functions and our simulation mostly aligns with the empirical study. There are no striking differences.

A weakness of our simulation is the absence of *missing variables*. While we do not yet know how such variables influence the results, we assume that it relates to what we called 'unclear' in the empirical study. This uncertainty, potentially induced by missing variables, calls for a more complex simulation in the future, but also for experiments with alternative candidate selection methods that better quantify this uncertainty.

The empirical study was run by a single human annotator, which may introduce bias in the labeling. To mitigate this, we have been running a quality assurance step one year after the labeling process, to assure the maximum quality of the labels in the final dataset.

9 Related Work

9.1 Security Defects in Code Reviews

We start with research on code reviews related to security defects. None of the following approaches involves active learning.

In [23], authors conduct an empirical study on the reviewing in the OpenStack and Qt communities. A keyword-based filter for preselection is applied. One of the conclusions is that '*with the proportion less than 1%.*' the fraction of reviews discussing security defects is vanishing low. Our study might inspire improvements in labeling in the future.

In [8], authors conduct an empirical study on the effectiveness of security code review. The presented method does not face the same scalability problem with labeling we aim to solve. The authors conducted a manual code review of a small web application. Labels studied are '*valid vulnerability*', '*invalid vulnerability*', '*weakness*' or '*out of scope*'.

In [18], authors conduct an empirical study on the effectiveness of peer code review in identifying security defects. The study also uses a keyword-based approach to filtering and might benefit from active learning with LLMs as an alternative.

In [7], authors conduct an empirical study on code review for the case of Chromium. The study focuses on answering questions like '*What categories of security issues are often missed or found?*'.

In [1], authors conduct an empirical study on the role of code review for npm packages. Again, authors use keywords.

9.2 Active Learning for Security Defect Labeling

The following approaches share an understanding of the importance of active learning for labeling security defects in general. However, they do not focus on code reviews. All approaches mine and label a dataset. They are not limited to training a classifier on a potentially rebalanced and existing dataset.

In [24], Yu et al. aim at limiting the labeling effort. They guide actively by a support vector machine when classifying security defects in source code. The evaluation of Yu et al. uses an existing dataset. They focus on studying incorrect labels and correction mechanisms. We focus on active learning and LLMs applied to reviews that are natural language artifacts and might later be used as proxy for finding defects in code.

Chen et al. publish an application of active learning in [4]. They do not mention active learning, but an iterative feedback loop is present. Labeling is guided by an ensemble classifier on multiple data sources, filtering for observations that are likely to be security defects, which is rare-label candidate selection. Conclusions, however, remain unclear as this study mixes active learning with self-training in the iterations. Our study shows a clear evaluation of active learning in simulations and on real data.

Zhang et al. focus using active learning to detect vulnerabilities using the AST of smart contracts in [25]. Specifically, Zhang et al. contribute to the reduction of noise in existing incorrect labels.

In [20], we see the first combination of active learning and LLMs, similar to ours, for processing cyber threat intelligence (CTI) reports. The study demonstrates a comparable reduction in manual labeling effort. However, there are significant differences in unlabeled data size and type. The authors work on 11,130 sentences from CTI reports, while we focus on 4,191,892 distinct reviews from GitHub. The CTI corpus is already specific to security, with a high density of security related information.

9.3 Supervised-Learning for Security Defect Detection

We briefly discuss examples of security defect detection without a feedback loop. Classifier training is a part of active learning.

Authors of [3] describe how they build a better dataset first, using Bugzilla and Debian security tracker; thereafter, they build a deep learning model. For the Devign approach, the authors [28] manually label a dataset previously filtered by security-related keywords, and afterward a classifier is trained. For the D2A dataset [27], authors filtering by static analysis, labels are added manually, and a classifier is trained. In [17], authors search CVEs and manually label to create a dataset and train a classifier on the code and commit messages.

9.4 Self-Supervised Learning or Rebalancing

There is also a large body of work on rebalancing datasets and self-supervised learning. For our work, it is critical that one classifier, the LLM, learns from another, the human. This happens by constructing a shared dataset. In its current form, we consider self-supervised learning and rebalancing as solutions to how a classifier is built from a dataset, and therefore as not directly related to our work.

10 Conclusion

We investigated the use of active learning with a fine-tuned large language model to make the mining and labeling of security defects more effective. In simulations, we have shown improvements in labeling by a factor of 13, and in an empirical study, we achieved a factor of more than 6. This improvement does not depend on a keyword list for upfront candidate selection, but is rooted in dynamically evolving an LLM over the process of labeling.

Our work holds the potential to inspire future research in this area, resolving rare class and imbalance problems at the root where they appear, by adjusting the mining and labeling of the datasets. Our final dataset and model are publicly available.

Acknowledgement

This work is funded by EU grant No. 101120393 (Sec4AI4Sec). Copilot with primarily GPT-4o underlying was used to assist with the editing of the text. Usage was limited to improving spelling, grammar, and clarity. Usage did not impact the content of this paper.

References

[1] Mahmoud Alfadel, Nicholas Alexandre Nagy, Diego Elias Costa, Rabe Abdalkareem, and Emad Shihab. 2023. Empirical analysis of security-related code reviews in npm packages. *J. Syst. Softw.* 203 (2023), 111752.

[2] Islem Bouzenia and Michael Pradel. 2024. Resource Usage and Optimization Opportunities in Workflows of GitHub Actions. In *ICSE*. ACM, 25:1–25:12.

[3] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2022. Deep Learning Based Vulnerability Detection: Are We There Yet? *IEEE Trans. Software Eng.* 48, 9 (2022), 3280–3296.

[4] Yang Chen, Andrew E. Santosa, Ming Yi Ang, Abhishek Sharma, Asankhaya Sharma, and David Lo. 2020. A Machine Learning Approach for Vulnerability Curation. In *MSR*. ACM, 32–42.

[5] Roland Croft, Muhammad Ali Babar, and M. Mehdi Kholoosi. 2023. Data Quality for Software Vulnerability Datasets. In *ICSE*. IEEE, 121–133.

[6] Ozren Dabic, Emad Aghajani, and Gabriele Bavota. 2021. Sampling Projects in GitHub for MSR Studies. In *MSR*. IEEE, 560–564.

[7] Marco di Biase, Magiel Bruntink, and Alberto Bacchelli. 2016. A Security Perspective on Code Review: The Case of Chromium. In *SCAM*. IEEE, 21–30.

[8] Anne Edmundson, Brian Holtkamp, Emanuel Rivera, Matthew Finifter, Adrian Mettler, and David A. Wagner. 2013. An Empirical Study on the Effectiveness of Security Code Review. In *ESSoS (Lecture Notes in Computer Science, Vol. 7781)*. Springer, 197–212.

[9] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *MSR*. ACM, 508–512.

[10] Xiuting Ge, Chunrong Fang, Meiyuan Qian, Yu Ge, and Mingshuang Qing. 2022. Locality-based security bug report identification via active learning. *Inf. Softw. Technol.* 147 (2022), 106899.

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. MIT press.

[12] Johannes Härtel and Ralf Lämmel. 2023. Operationalizing validity of empirical software engineering studies. *Empir. Softw. Eng.* 28, 6 (2023), 153.

[13] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[14] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR* abs/1907.11692 (2019). arXiv:1907.11692 http://arxiv.org/abs/1907.11692

[15] Tim P Morris, Ian R White, and Michael J Crowther. 2019. Using simulation studies to evaluate statistical methods. *Statistics in medicine* 38, 11 (2019), 2074–2102.

[16] Kollin Napier, Tanmay Bhowmik, and Zhiqian Chen. 2024. Explaining poor performance of text-based machine learning models for vulnerability detection. *Empir. Softw. Eng.* 29, 5 (2024), 113.

[17] Truong Giang Nguyen, Thanh Le-Cong, Hong Jin Kang, Xuan-Bach Dinh Le, and David Lo. 2022. VulCurator: a vulnerability-fixing commit detector. In *ESEC/SIGSOFT FSE*. ACM, 1726–1730.

[18] Rajshakhar Paul. 2021. Improving the effectiveness of peer code review in identifying security defects. In *ESEC/SIGSOFT FSE*. ACM, 1645–1649.

[19] Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. 2019. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *MSR*. IEEE / ACM, 383–387.

[20] Fariha Ishrat Rahman, Sadaf Md. Halim, Anoop Singhal, and Latifur Khan. 2024. ALERT: A Framework for Efficient Extraction of Attack Techniques from Cyber Threat Intelligence Reports Using Active Learning. In *DBSec (Lecture Notes in Computer Science, Vol. 14901)*. Springer, 203–220.

[21] Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-Yao Huang, Zhihui Li, Brij B. Gupta, Xiaojiang Chen, and Xin Wang. 2022. A Survey of Deep Active Learning. *ACM Comput. Surv.* 54, 9 (2022), 180:1–180:40.

[22] Burr Settles. 2009. Active learning literature survey. (2009).

[23] Jiaxin Yu, Liming Fu, Peng Liang, Amjed Tahir, and Mojtaba Shahin. 2023. Security Defect Detection via Code Review: A Study of the OpenStack and Qt Communities. In *ESEM*. IEEE, 1–12.

[24] Zhe Yu, Christopher Theisen, Laurie A. Williams, and Tim Menzies. 2021. Improving Vulnerability Inspection Efficiency Using Active Learning. *IEEE Trans. Software Eng.* 47, 11 (2021), 2401–2420.

[25] Jiale Zhang, Liangqiong Tu, Jie Cai, Xiaobing Sun, Bin Li, Weitong Chen, and Yu Wang. 2022. Vulnerability Detection for Smart Contract via Backward Bayesian Active Learning. In *ACNS Workshops (Lecture Notes in Computer Science, Vol. 13285)*. Springer, 66–83.

[26] Qiyuan Zhang, Fuyuan Lyu, Zexu Sun, Lei Wang, Weixu Zhang, Zhihan Guo, Yufei Wang, Irwin King, Xue Liu, and Chen Ma. 2025. What, How, Where, and How Well? A Survey on Test-Time Scaling in Large Language Models. *CoRR* abs/2503.24235 (2025).

[27] Yunhui Zheng, Saurabh Pujar, Burn L. Lewis, Luca Buratti, Edward A. Epstein, Bo Yang, Jim Laredo, Alessandro Morari, and Zhong Su. 2021. D2A: A Dataset Built for AI-Based Vulnerability Detection Methods Using Differential Analysis. In *ICSE (SEIP)*. IEEE, 111–120.

[28] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *NeurIPS*. 10197–10207.