

# LPASS: Linear Probes as Stepping Stones for vulnerability detection using compressed LLMs

Luis Ibanez-Lissen<sup>a</sup>, Lorena Gonzalez-Manzano<sup>a,c,d</sup>, Jose Maria de Fuentes<sup>a,b</sup>, Nicolas Anciaux<sup>b</sup>

<sup>a</sup>Universidad Carlos III de Madrid, Leganes, Madrid

<sup>b</sup>Inria de Saclay, Palaiseau, France

<sup>c</sup>Institut Polytechnique de Paris, Palaiseau, France

<sup>d</sup>Corresponding author

## Abstract

Large Language Models (LLMs) are being extensively used for cybersecurity purposes. One of them is the detection of vulnerable codes. For the sake of efficiency and effectiveness, compression and fine-tuning techniques are being developed, respectively. However, they involve spending substantial computational efforts. In this vein, we analyse how Linear Probes (LPs) can be used to provide an estimation on the performance of a compressed LLM at an early phase – before fine-tuning. We also show their suitability to set the cut-off point when applying layer pruning compression. Our approach, dubbed *LPASS*, is applied in BERT and Gemma for the detection of 12 of MITRE’s Top 25 most dangerous vulnerabilities on 480k C/C++ samples. LPs can be computed in 142.97 s. and provide key findings: (1) 33.3 % and 72.2% of layers can be removed, respectively, with no precision loss; (2) they provide an early estimate of the post-fine-tuning and post-compression model effectiveness, with 3% and 8.68% as the lowest and average precision errors, respectively. *LPASS*-based LLMs outperform the state of the art, reaching 86.9% of accuracy in multi-class vulnerability detection. Interestingly, *LPASS*-based compressed versions of Gemma outperform the original ones by 1.6% of F1-score at a maximum while saving 29.4 % and 23.8% of training and inference time and 42.98% of model size.

**Keywords:**

Vulnerability detection, LLM, model compression, linear classification probes

## 1. Introduction

The wide variety of devices, systems and networks favors the emergence of vulnerabilities, defined as weaknesses in a system or security control<sup>1</sup>. Lots of research works have been carried out to counter them. Particularly, multiple works have focused on vulnerability detection [1, 2, 3], that is, whether a piece of software is vulnerable or not, whereas others aim to identify them, that is, which vulnerability is present [4, 5, 6]. This latter problem is more challenging as it requires a fine-grained knowledge of all vulnerabilities to tell them apart [7]. Therefore, in this paper we concentrate on this issue.

**Context.** The use of Artificial Intelligence (AI) has been a constant in this field. Many works apply traditional AI algorithms such as support vector machines [8, 9], K-nearest neighbor [8, 9] or graph neural networks [10, 11], once having extracted code features like the program dependency graph [10, 11], the control flow graph [12] or the number of lines of code [8], among others.

However, since the appearance of Large Language Models (LLMs), which are based on using deep learning techniques to manage massive amounts of data, novel vulnerability detection systems have appeared [13]. The main advantage of these models is their ability to manage complex tasks, but the required computational cost becomes a burden. Some of these models apply millions or even billions of parameters whose tuning and execution involves a high cost in terms of time and resources.

<sup>1</sup><https://csrc.nist.gov/glossary/term/vulnerability>, last access September 2024

New research lines try to reduce LLMs, speeding up the classification process and reducing the amount of used computing resources. Techniques such as pruning focus on reducing the amount of model parameters [14] or network components [15]. On the other hand, quantization is another well-known approach that reduces the size of each parameter. However, compression methods incur in accuracy losses which have to be minimized [16]. In the context of vulnerability detection, to the best of authors’ knowledge only [17], [18] have applied model compression. They apply knowledge distillation [19] to compress CodeBERT and GraphCodeBERT, reaching promising inference speeds and efficiency. However, they focus on a binary classification problem (i.e., vulnerable / non-vulnerable) with an accuracy of 59.9 % – only 10% over a random guess.

**Motivation.** Efficiency in vulnerability detection is of outmost relevance considering the increasing pace of software generation. For the sake of illustration, Google Play counts on 2.61 billion apps nowadays <sup>2</sup>. At the same time, not only the amount of vulnerabilities, but also their severity, have been steadily growing in the last decade. According to CVEdetails<sup>3</sup>, the amount of vulnerabilities with a severity ranked between 7 and 10 raised from 1.9k in 2014 to 16.7k in 2023. Therefore, a vulnerability detection mechanism streamlined with the publication process would be desirable. Interestingly, identifying which is the vulnerability at stake enables providing a suitable response – while dangerous vulnerabilities may require stronger controls, irrelevant ones may simply raise a warning before distributing the analysed pieces of software.

As noted in [20], fine-tuning of models involves substantial energy expenses. For the sake of illustration, pre-training and fine-tuning Meta’s LLaMA model could cause up to 2.76 MtCO<sub>2</sub>-eq emissions, equivalent to the total pollution caused by manufacturing one dose of COVID-19 vaccine for all humans on Earth [21]. Therefore, saving resources related to the model fine-tuning and compression is paramount.

The closest effort to ours, namely Chen et al. [22] is focused on compressing visual models, thus unrelated to vulnerability detection. They proposed using feature representations in convolutional neural networks to identify layers with high weight overlap *after training*, aiming to reduce the model size. In contrast, our approach seeks to compress pretrained LLMs by determining which layers of a model provide valuable information for a specific task *before any fine-tuning or further training*. Therefore, although Chen et al.’s method can save resources post-training, it requires training the entire model beforehand. Our approach, on the other hand, reduces the model size prior to fine-tuning, minimizing the computational resources needed for that task.

**Research question and contribution.** Given the computational cost of model fine-tuning and compression, our research question is: *Can we predict their impact to make informed decisions on their use and save resources?* To address this, we leverage linear classifier probes (LPs) [23] to gain early insights of the internal model status that can be valuable to guide the use of fine-tuning and compression techniques and estimate their effect in performance. Our approach, dubbed *LPASS* (LPs As Stepping Stones) is inspired by recent efforts that apply LPs to analyse the knowledge captured at varying depths within LLMs [24], as well as the internal uncertainty [25]. LPs are a first layer of explainability by providing interpretability. As such, our use of LPs can be regarded as a first step for a explainable compression method, which has been recently pointed out as an open issue [26].

The list of contributions is as follows:

- We adopt linear probes (LPs) in vulnerability detection for 1) determining the cut-off point when applying layer pruning and 2) estimating the effectiveness and performance of fine-tuned and compressed models.
- We test *LPASS* in two well-known LLMs, namely Bert [27] and Gemma [28], compressed by means of layer pruning and quantization [29]. Bert is selected for being a common choice in previous works, whereas Gemma is a recent, state-of-the-art alternative. Three representative datasets are applied, namely DiverseVul [1], BigVul [30] and PrimeVul [3]. We focus on 12 of the most dangerous vulnerabilities according to MITRE Top 25 [31]. Our compressed models outperform the state-of-the-art while exhibiting promising performance features.
- We release our experimental materials to foster further research.

The remaining of this paper is as follows. Section 2 gives the background. Section 3 describes the foundations of *LPASS*, whereas Section 4 details the approach. Section 5 describes its assessment. Section 6 shows the related work. Lastly, Section 7 concludes the paper and points out future work directions.

<sup>2</sup><https://www.businessofapps.com/data/google-play-statistics/>, last access 20 june 2024.

<sup>3</sup><https://www.cvedetails.com/>, last access September 2024

## 2. Preliminaries

In this section, model compression techniques, the notion of linear probes and the Common Weakness Enumeration (CWE) approach for naming vulnerabilities are introduced.

### 2.1. Model compression

There are several techniques available to reduce the size of models and lower computational resource requirements. Interested readers may refer to [26]. The following three are predominant.

One such technique is knowledge distillation [19], where the responses of a larger model (the teacher) are used to guide the training of a smaller version of the original model (the student).

Another commonly used technique is pruning, which was previously referred to as ‘neural network pruning’ [32, 33], which aims at reducing model size. Pruning can be done either by modifying the original model architecture (structured pruning) or by removing individual weights and activations (unstructured pruning). This technique reduces the model’s complexity and the memory needed to run and store the models.

Quantization, on the other hand, focuses on reducing the model size and computational requirements by mapping continuous infinite numbers to a smaller set of discrete finite numbers. It involves converting weights stored in high-precision values to lower-precision data types. For example, 32 bits weights can be mapped to 8-bit integers in the range [-128, 127] or to 4-bit integers in the range [-8, 7]. It minimizes the number of required bits and the precision of the computations while trying to maximize accuracy, either post-training or after training [14]. According to Marchisio et al. [34] the effect of quantization is dependent on the language or context. Thus, anticipating its effects is far from straightforward.

### 2.2. Linear probes

Linear Classifier Probes, hereinafter Linear Probes (LP), are simple classifiers that contribute to deep learning models explainability efforts by providing insights into how the model processes information internally [23]. These LPs are used to make predictions over the hidden states of the models, trying to predict or identify if some specific information is correctly represented within them. For LLMs, a LP classifier is typically placed after each layer of the network and takes the hidden states as input  $X$  and predicts a simple characteristic  $Y$  (e.g. predict the number of lines of a piece of code). They are trained on probing datasets designed to predict expected characteristics that are predefined and known in advance, giving a sense of how different layers of the model encode and retain the expected information. They have previously been employed to enhance explainability in tasks such as document ranking [35], and to shed light on model behavior, including hallucinations and the internal representation of code and general knowledge [36, 37, 38, 24, 39]. Beyond their use for understanding representations, linear probes have also been applied to directly improve model performance—by identifying task-relevant components within the model [40, 41]—and to mitigate undesirable behaviors such as sycophancy [42].

### 2.3. Common Weakness Enumeration (CWE)

The Common Weakness Enumeration (CWE) [31] is a community-driven classification and categorization system designed to identify potential common software and hardware vulnerabilities. It defines families of taxonomies that encompass vulnerabilities, which are interrelated or serve as the basis for higher-level abstract classes, by assigning a unique identifier and a potential damage score. This information is subsequently used to rank common flaws and errors made by developers.

For instance, the MITRE Top 25<sup>4</sup> is a list updated three times per year that highlights the most common and impactful software weaknesses, which are often easy for attackers to exploit.

## 3. LPASS foundations

Section 3.1 provides an overview of the proposal to then introduce the pursued goals in Section 3.2.

---

<sup>4</sup><https://cwe.mitre.org/top25/>

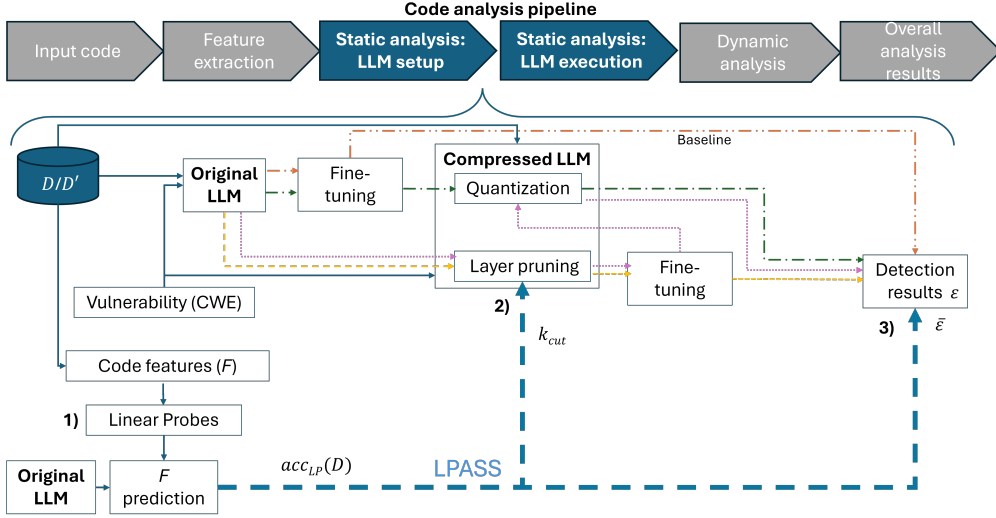


Figure 1: *LPASS* overview. Steps of *LPASS* are numbered and colored flows are used to assess the approach when using LLMs to detect vulnerabilities. The orange flow (dot-dot-dash) computes the baseline results, whereas the green (dot-dash), yellow (dashes) and pink (dots) flows refer to compressed LLMs, that is after applying quantization, layer pruning or both techniques at the same time, respectively.

### 3.1. Overview & Use case

*LPASS* is framed in a typical vulnerability detection pipeline, which includes both static and dynamic analyses as part of a mobile application security testing<sup>5</sup> as the one in use by Google or Apple in their application store. Thus, codes are subject to a static and dynamic analysis to detect if there are any vulnerabilities (Figure 1 upper part). The focus of *LPASS* is on the static analysis phase, that is supposed to be carried out by means of LLMs.

The aim of *LPASS* is to help on deciding whether investing resources for fine-tuning and compressing a LLM are worthy. In this regard, the user of *LPASS* is only requested to extract simple code features  $F$  from the samples at stake and compute the LPs. Both tasks involve negligible computational efforts. The remaining values needed for making the decision are the ones provided as a result of our work, as explained later.

Resorting to the selected code features  $F$  is essential for a real-world usage of *LPASS*. The user does not know whether the samples are vulnerable or not – it is the actual purpose of the LLM at stake. However, these code features  $F$  can be extracted very easily without the need of any LLM. Therefore, features  $F$  can be regarded as *proxies* for the presence or absence of vulnerabilities in the code. Such a feature extraction is natural in a vulnerability detection pipeline, as many of them (e.g., lines of code or the control flow graph) are routinely used.

The proposed approach is depicted in Figure 1. Firstly, for each dataset (see Section 5.2), LPs are trained over the internal activations after each layer of the LLMs, using the code samples as input. The goal is to predict the selected code features  $F$  using these activations. The selection of which features  $F$  to consider is a key aspect of *LPASS*. Secondly, the accuracy of LPs on each dataset  $D$  ( $acc_{LP}(D)$ ) is used to set the cut-off point when compressing the LLM using layer pruning, where all layers beyond the cut-off layer  $k_{cut}$  are removed from the LLM. Thirdly, LPs are also used for estimating the vulnerability detection effectiveness ( $\bar{\mathcal{E}}(D)$ ) without the need of fine-tuning or executing any LLM. To assess the quality of the estimations, vulnerabilities are detected ( $\mathcal{E}(D)$ ) using the original LLMs to compute baseline results (orange arrow), as well as in compressed LLMs, that is after applying quantization, layer pruning or both techniques at the same time (green, yellow and pink arrows, respectively).

It must be noted that in order to validate the quality of the provided estimations, in this paper we carry out both fine-tuning and compression techniques on two representative LLMs.

### 3.2. Goals

*LPASS* aims to meet the following goals:

<sup>5</sup><https://mobile-security.gitbook.io/mobile-security-testing-guide/overview/0x04b-mobile-app-security-testing>

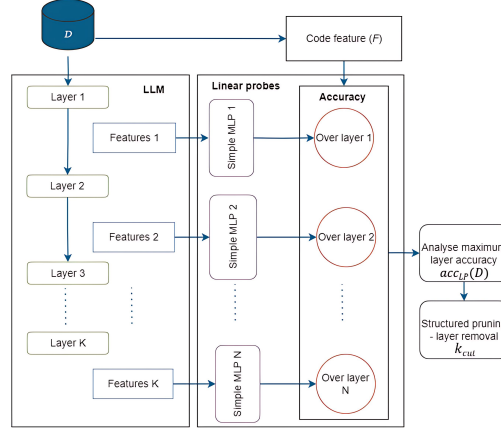


Figure 2: Linear probes used for determining  $k_{cut}$ . Code features  $F$  are the target of the prediction, which is based using the LLM’s internal activations per layer. The layer in which the best prediction is made is selected as the cut-off point  $k_{cut}$

- **Early assessment.** It must be possible to ascertain whether the LLM will be effective or not in vulnerability detection while spending a reduced amount of computational efforts.
- **Vulnerability detection.** *LPASS*-built LLMs must suitably detect vulnerable codes, identifying which vulnerability is present in a given piece of code.
- **Time efficiency.** *LPASS*-built LLMs must minimize the time needed for training and inference.
- **Memory efficiency.** *LPASS*-built LLMs must reduce the memory requirements.

As it can be seen, the first goal refers to *LPASS* itself, whereas the remaining ones are the expected results of its use in LLMs. It must be noted that imposing restrictions on time and memory are aligned to achieving energy savings.

#### 4. Description of *LPASS*

This section describes the proposed method. For the sake of clarity, the two main steps are presented separately – the way to guide layer pruning using LPs (Section 4.1) and using LPs to estimate the post-fine-tuning and post-compression performance in terms of effectiveness, time and model size (Section 4.2).

##### 4.1. Layer pruning leveraging Linear Probes

In this proposal, LPs, depicted in Figure 2, are implemented with a Multi-Layer Perceptron (MLP) applied at each LLM layer  $k$ . In particular, the internal activations  $H_k$  are collected at each layer  $k$  and they are used as input to a MLP which predicts the selected code features  $F$ .

Once having classes for MLPs, linear probes are computed and their result is the accuracy ( $acc$ ) per layer  $k$  and CWE, though the average  $acc$  of all CWE per  $k$  ( $avg(acc_k)$ ) is determined. Finally, the calculus of the  $loss$  of  $avg(acc_k)$  allows identifying which is the cutting  $k$  ( $k_{cut}$ ) to remove upper layers and enforce structured pruning. The  $loss$  refers to the amount of accuracy lost in each  $k$  and it is calculated through the difference between the maximum  $avg(acc_k)$ , at a particular  $k$ , and the  $avg(acc_k) \forall k \in K$ , Equation 1

$$loss_k = \max_{i \in k} (avg(acc_i)) - avg(acc_k) \quad (1)$$

This is carried out for all datasets (noted  $\mathcal{D} = \{D\}$ ) and code features  $F$ , and  $k_{cut}$  is set as the  $k$  which minimizes the sum of  $loss$ , in absolute value, for each  $k$ , Equation 2.

$$k_{cut} = \arg \min_{k \in K} \left( \sum_{\|\mathcal{D}\| \times \|F\|} |loss_k| \right) \quad (2)$$

where  $\|X\|$  denotes the cardinality of a set  $X$  and  $|x|$  the absolute value of  $x$ .

#### 4.2. Estimating performance for vulnerability detection

The same LPs used for layer pruning are now used for estimating the performance of the LLM to detect vulnerabilities in case compression or fine-tuning techniques were applied. Let  $\mathcal{E}(D)$  be the set of effectiveness metrics of a LLM at stake, which contains precision, recall and F1-score for a dataset  $D$ . The overall estimation process bases on computing  $\overline{\mathcal{E}} \approx \mathcal{E}$  as shown in Equation 3 for each of these metrics.

$$\overline{\mathcal{E}(D)} = acc_{LP}(D) + \beta \quad (3)$$

The goal is to determine which value  $\beta$  should be added to the result of LPs on  $D$ , namely their accuracy ( $acc_{LP}(D)$ ), to obtain the estimate. Note that  $acc_{LP}(D) \in [0, 1]$ , and  $\beta \in [0, 1]$  as explained later. Thus,  $\overline{\mathcal{E}(D)}$  must be post-normalized to fit into  $[0, 1]$  as well.

##### 4.2.1. Computing $\beta$

$\beta$  is the value provided within this proposal to be used in the equation above. It is computed leveraging a knowledge base formed by datasets  $D' \neq D$ , as shown in Equation 4.

$$\beta = \mathcal{E}(D') - acc_{LP}(D') \quad (4)$$

In order to get  $\beta$ , it is necessary to obtain  $\mathcal{E}(D')$ , that is, the real performance result for the LLM on the dataset  $D'$ . However, note that users of *LPASS* will not need to perform this operation – we later show that our results for  $\beta$  generalize for different datasets.

##### 4.2.2. Validating $\overline{\mathcal{E}(D)}$

To assess the approach, an *Estimation Error* (*Err*) is computed to determine the error incurred, as shown in Equation 5.

$$Err = \overline{\mathcal{E}(D)} - \mathcal{E}(D) \quad (5)$$

In our approach, we select different datasets (as introduced later) and we apply a leave-one-out cross validation. This ensures the generalization of our results.

## 5. Evaluation

This section assesses *LPASS*. For this purpose, the LLMs and data at stake are introduced in Sections 5.1 and 5.2, respectively. The selected code features  $F$  are described in Section 5.3. Sections 5.4 and 5.5 describe the experimental settings and metrics. Results are presented in Section 5.6. Lastly, Section 5.7 discusses the results and the limitations of the approach.

### 5.1. LLMs

A couple of large language models (LLMs) are selected. On one hand, the BERT model is chosen due to its widespread use [43], and its prominence in related works such as [1, 18, 6]. The large variant of BERT is used, featuring 334 million parameters and 24 layers. On the other hand, Gemma model is chosen for being one of the latest and most recently used LLMs [28], configured, due to resource limitations, with 2 billions of parameters and 18 layers.

All LLMs base on a fundamental unit, the token, which is the minimum processed information. In the case of BERT, 512 tokens are the maximum applied, while in Gemma this number can increase up to 8,192. Nonetheless, our resource constraints force the use of Gemma with up to 1,024 tokens.

### 5.2. Datasets and Vulnerabilities

Three C/C++ datasets, are selected for being well-known and for facilitating a multi-class classification. Indeed, they are chosen among those used in the state of art (cf. Section 6). DiverseVul [1] is composed of extracted commits from vulnerable and non-vulnerable functions covering more than 295 projects after crawling security issue websites. Big-Vul [30] is a vulnerability dataset from 348 open source Github projects, where vulnerability-related code commits and extracted relevant code changes are collected. PrimeVul [3] is a dataset of benign and vulnerable functions composed of a merging of four well-known vulnerability datasets, including both previously mentioned. Note that based on related work (Section 6), Devign and Draper datasets were also considered but discarded since they do not provide information about the CWE, just vulnerable or not. Similarly, CVE-fixes and SARD were analysed and discarded because the amount of samples per CWE were smaller (3k maximum) than the ones of the chosen datasets.

Among all existing vulnerabilities, the Top 25 most dangerous CWE <sup>6</sup> are considered herein. As not all CWEs appear in all datasets, we select the 10 CWEs with the highest number of samples per dataset. Table 1 summarizes the number of samples per dataset and selected CWE, most of them common for all datasets. Hence, a total of 12 CWEs are at stake. It is noteworthy that 1,024 token size allows increasing the sample set, thus leading to 480k samples. Samples beyond 512 or 1,024 tokens on each of the cases have been removed.

Table 1: Samples per dataset of the 10 most represented CWEs within MITRE Top 25

	CWE	512 Tokens			1,024 Tokens		
		DiverseVul	Big-Vul	PrimeVul	DiverseVul	Big-Vul	PrimeVul
		# samples			# samples		
Common	20	22,197	15,326	25,799	33,491	25,799	25,596
	119	19,853	18,321	22,575	32,013	22,575	17,216
	125	23,675	4,774	2,498	32,951	6,206	19,377
	190	8,905	2,632	3,900	29,395	3,262	7,524
	362	7,729	4,363	6,206	11,493	5,478	6,882
	416	23,691	7,428	8,993	32,951	8,993	16,593
	476	16,691	2,943	5,478	27,981	3,900	12,781
	787	19,241	2,020	3,262	29,395	2,498	27,199
Different	22	2,409	-	-	3,693	-	-
	78	2,233	-	875	3,051	-	29,066
	79	-	699	857	-	857	653
	269	-	639	-	-	875	-
Total		146,624	59,145	76,543	236,414	80,443	162,887

### 5.3. Code features $F$

Cyclomatic Complexity (CC) and Halstead Difficulty (HD) of code samples are the chosen code features  $F$ . There are lots of  $F$  that could be predicted but looking for capturing code complexity [44] and considering the state of the art in vulnerability detection [45, 8], both CC and HD were chosen for their ability to capture the structural information of the code. CC is a quantitative measure of the number of linearly independent paths in the code [46], while HD measures the diversity of operands present in the code<sup>7</sup>. Accurately predicting these features would indicate that the LLMs effectively encode the structural information of the code, offering a more abstract and complex understanding than simply predicting basic metrics like the number of lines or tokens. These features are particularly valuable because they are challenging to predict, requiring the LLMs to deeply understand and analyze the code, unlike simpler metrics such as line counts.

CC and HD are integer and float numbers respectively, from 0 to infinite and thus, the number of classes of the MLP has to be determined. To do so, CC and HD are computed for all samples in each dataset and after the analysis of their distribution, the number of classes is set such that the vast majority of samples of each dataset (in our case, we opted for 85%) are included.

### 5.4. Experimental settings

This section outlines the training settings for the chosen models. Our experimental materials are publicly released<sup>8</sup>. Training was conducted on two NVIDIA consumer GPUs, a RTX 4090 and a RTX 4080, using the Pytorch framework

<sup>6</sup><https://cwe.mitre.org/top25/>, last access September 2024

<sup>7</sup><https://product-help.schneider-electric.com/Machine%20Expert/V2.0/en/CodeAnly/CodeAnly/D-SE-0095969.html>, last accessed September 2024

<sup>8</sup>A reduced version is published until acceptance; <https://github.com/Luisibear98/LPASS-pruning>

and the Hugging Face library<sup>9</sup>. For both training and validation, all datasets were split in an 80%-20% random distribution to accommodate the imbalance of data. Each CWE is limited to 5,000 samples for training because, after a trial and error process, such value allows the execution of both models, while lower ones do not work in both of them. To ensure balanced CWE classes, if a CWE does not have enough samples, oversampling is carried out copying samples until getting to the set limit [47]. Otherwise, undersampling is enforced removing samples [47], though setting aside 20% for the validation set. Besides, computations are repeated 3 times per model and pruning set-up and results present the average of all executions.

BERT model is fine-tuned for training based on [1], specifically a learning rate of  $2e-5$  is applied, using the Adam optimizer [48] over 10 epochs. For Gemma, Galore low-rank adaptation training [49] was adopted due to resource constraints, using per layer-weights update implementation. Since higher rank requires more GPU memory, when fine-tuning the entire model, the rank is limited to 256 but fine-tuning pruned models it is set to 1,024.

In what comes to the batch size, our preliminary tests show that small sizes do not affect the accuracy while harm performance. Thus, the batch size is set to the maximum capacity per GPU. Finally, concerning the class 'non vulnerable', in DiverseVul it corresponds to samples marked as non-vulnerable, in PrimeVul and big-vul refers to samples tagged as 'None' in the CWE-ID field.

### 5.5. Metrics

*Effectiveness metrics* used to measure the model's capabilities in a multi-class classification problem include Accuracy, F1 Score, Precision, and Recall. These metrics are reported in two ways. First, they assess the model's overall ability to discern among multiple classes, reflecting its pure capacity in a multi-class context. Second, they evaluate the model in a binary fashion, determining its ability to differentiate between code with a CWE and code without a CWE, thus considered non vulnerable. Indeed, in this latter case, given the class imbalance (5,000 non vulnerable vs 50,000 vulnerable), F1 is the computed metric. Additionally, *performance metrics* refer to time and memory ones. The former ones corresponds to training time of the model and the inference one required for each sample prediction. By contrast, memory metrics refer to the minimum amount of GPU memory required for training and inference the model setting a batch size of 1, the number of effective parameters and the model size.

### 5.6. Results

This section presents results of the enforcement of structured pruning applying LP (Section 5.6.1), the detection of vulnerability (Section 5.6.2), the effects of the detection process in time and memory efficiency (Section 5.6.3) and the early assessment of the model effectiveness after fine-tuning and compression (Section 5.6.4).

**Understanding Tables 2 and 3.** These tables summarize the impact of layer pruning using the cut-off point (Table 2) and also quantization (Table 3) on effectiveness, time, model memory and size. For clarity, baseline results are shown in gray in Table 2. The results of the models after applying these compression techniques are then presented as the *difference (in %) between the baseline and each compressed version*. Thus, negative values in effectiveness are preferred – this means that the compressed model outperforms the baseline. On the contrary, positive values in time and model memory/size are desired – the compressed model would then be faster and smaller.

#### 5.6.1. Layer pruning

This step involves the computation of LP. Firstly, MLP classes are established based on Section 4. CC and HD are calculated getting the average of all CWE. A similar distribution is identified in all cases. Considering a coverage of 85% or more (recall Section 4), 5 classes are defined for CC [1,2,3,4,5] and 6 for HD [1-5,6-10,11-15,16-20,21-25,25-30], where HD are divided in groups of 5. Note that 0 or floats with 0 as the integer part are discarded for not being considered representative enough.

Once having classes, LLMs are applied and LPs are computed for all datasets  $D$  and both features  $F$  (i.e., CC and HD). For the sake of fairness, the same number of samples per CWE class is considered – it is set considering the class with the minimum number of samples and downsampling the remaining classes. Figure 3 presents the *loss* in all cases. In the case of BERT (Figure 3-A), it is identified that, specially in CC, *loss* gets the minimum, in all cases, around  $k=15$ . In HD the trend is not so clear, but *loss* tends to remain constant at  $k=15$  or increase, in the

<sup>9</sup><https://huggingface.co>



Table 2: Vulnerability detection, time and memory results - Baseline (in gray) & Layer pruning (% vs baseline). For effectiveness columns, a negative value refers to an increment over the baseline. For time and model, a positive values means an improvement over the baseline.

		Effectiveness							Time		Model			
		Multi-class detection				Binary de- tection								
Rank	Removed layers	Accuracy	F1	Precision	Recall	F1	Training time	Inference time	Memory GPU- train	Memory GPU- inference	Effective Param- eters	Model size		
Gemma (18 layers)														
DiverseVul														
256	0	73.40%	74.90%	75.30%	75.00%	95.90%	3h 10 min	0.018s	8.96GB	5.76GB	2506172416	4770.15MB		
256	13	0.8	0.53	0.43	0.87	-0.06	30.18	78	41.07	48.09	52.72	57.02		
1024	13	-2.19	-2.16	-1.9	-2.16	-0.13	52.63		34.98					
PrimeVul														
256	0	88.10%	89.00%	88.80%	88.10%	99.50%	3h 31m 1s	0.013s	8.96GB	5.46GB	2506172416	4770.15MB		
256	13	1	1.6	0.2	1.61	-0.06	70.62	76.15	41.07	49.08	52.72	57.02		
1024	13	1.2	1.5	0.5	1.17	-0.02	71.36		34.98					
Big-Vul														
256	0	77.00%	79.30%	78.80%	79.90%	95.30%	3h 46m 16s	0.013s	8.96GB	5.46GB		4770.15MB		
256	13	-0.51	-1.42	-2.04	-0.83	0.43	65.54	76.15	41.07	49.08	52.72	57.02		
1024	13	-2.69	-3.48	-4.59	-2.63	-0.78	58.81		34.98					
BERT (24 layers)														
DiverseVul														
-	0	78.20%	79.60%	79.50%	79.60%	96.10%	2h 48m 49s	0.0056s	7.21 GB	1.726 GB	335153163	1278.46 MB		
-	8	0.48	0.92	0.44	1.14	0.28	47.47	32.14	27.77	15.41	35.7	26.32		
PrimeVul														
-	0	86.10%	86.90%	87.10%	86.60%	99.20%	2h 46m 26s	0.0056s	7.21 GB	1.726 GB	335153163	1278.46 MB		
-	8	0.2	0.2	0.52	-0.27	0.21	68.54	32.14	27.77	15.41	26.31	26.32		
Big-Vul														
-	0	82.40%	85.10%	85.40%	84.70%	95.90%	2h 47s 38m	0.0056s	7.21 GB	1.726 GB	335153163	1278.46 MB		
-	8	0.34	0.45	0.57	1.9	0.06	68.91	32.14	27.77	15.41	26.31	26.32		

Table 3: Vulnerability detection, time and memory results - Quantization & Layer pruning (% vs baseline shown in Table 2)

		Effectiveness					Time	Model		
		Multi-class detection				Binary detection				
Quantization	Removed layers	Accuracy	F1	Precision	Recall	F1	Inference time	Memory GPU-inference (GB)	Effective Parameters	Model size
Gemma (18 layers/ 256 rank)										
DiverseVul										
8 bits	0	1	1.32	1	1.25	1.43	-763.49	36.46	0	39.35
4 bits	0	18.78	19.8	10.65	19.68	2.18	-400.5	43.98	39.54	56.75
8 bits	13	1.3	1.48	0.71	1.2	-0.43	-4.5	51.35	57.11	68.01
4 bits	13	1.2	1.26	0.6	1.19	-0.4	20.8	57.33	68.09	72.84
PrimeVul										
8 bits	0	0.17	0.04	0.24	-0.71	-0.04	-2.130.77	32.97	0	39.35
4 bits	0	22.38	27.4	15.4	30.85	1.08	-300	40.90	39.54	56.75
8 bits	13	1	1.5	2.46	1.11	-0.02	-223.08	48.68	57.11	68.01
4 bits	13	4.1	3.1	4.1	1.18	0	-153.85	54.98	68.09	72.84
Big-Vul										
8 bits	0	0.05	0.77	0.56	0.41	0.42	-846.15	32.97	0	39.35
4 bits	0	10.05	9.92	5.46	10.06	1	-420.77	40.90	39.54	56.75
8 bits	13	-2.05	-2.13	-2.91	-1.76	0.22	-190.77	48.68	57.11	68.01
4 bits	13	-1.25	-1.61	-2.6	-1.23	0.8	-69.23	54.98	68.09	72.84
BERT (24 layers)										
DiverseVul										
8 bits	0	0.32	0.23	-0.12	0.38	0.2	-846.43	18.31	0	72.53
4 bits	0	0.89	0.71	0.3	0.88	0.39	-176.79	6.72	82.51	80.08
8 bits	8	0.59	1.28	0.58	1.27	0.38	-517.86	30.48	51.11	82.48
4 bits	8	1.7	1.36	1	1.51	0.33	-85.71	18.89	164.13	86.73
PrimeVul										
8 bits	0	0	-0.08	-0.18	-0.12	0.02	-828.57	17.15	0	72.53
4 bits	0	0.14	0.07	-0.24	0.22	0.01	-167.86	3.82	82.51	80.08
8 bits	8	0.5	0.47	1.1	0.53	0.11	-507.14	31.63	51.11	82.48
4 bits	8	1	1.2	1.1	0.8	0.11	-78.57	21.21	164.13	86.73
Big-Vul										
8 bits	0	0.26	0.18	0.03	0.3	-0.12	-828.57	13.67	0	72.53
4 bits	0	0.95	0.58	0.34	0.78	-0.05	-167.86	11.94	82.51	80.08
8 bits	8	0.64	1.05	1.23	0.82	0.15	-525	28.56	51.11	82.48
4 bits	8	0.84	1.66	1.63	1.7	0.09	-87.5	24.1	164.13	86.73

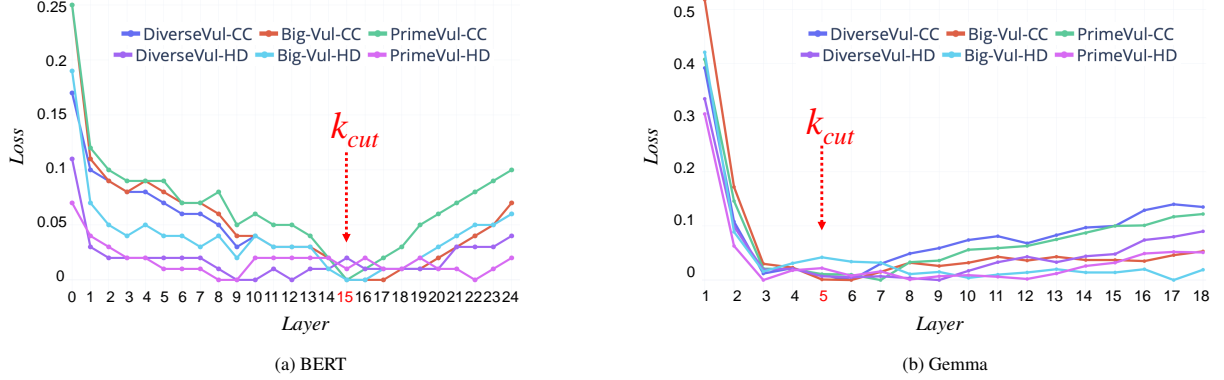


Figure 3: LLMs *loss* in absolute value for *D*, *CC*, and *HD*. Notice the decrement until the  $k_{cut}$  layer and the later increment after that layer denoting that the hidden representations are not adding extra information to the embeddings.

case of Big-Vul. Then, based on Equation 2,  $k_{cut}=15$  including the embedding layer. On the contrary, in the case of Gemma (Figure 3-B), *loss* shows that from  $k=3$  on there are some changes in trends, being around  $k=5$  and 6 when *loss* becomes quite constant or even decrease, reaching  $k_{cut}=5$ . In both models *CC* is the code feature which presents more differences among  $k$  and shows a clear trend, probably because the complexity it involves becomes harder to predict.

### 5.6.2. Vulnerability detection

This section presents vulnerability detection results of all datasets *D*, BERT and Gemma (Tables 2 and 3). Effectiveness metrics are computed for identifying CWEs and for distinguishing if a code sample is or not vulnerable.

BERT model shows no meaningful difference compared to the baseline values when discerning across different datasets, CWEs or discerning among vulnerable and no vulnerable samples. Detecting CWE and considering just layer pruning, just the recall in DiverseVul and Big-Vul degrades more than 1%. When applying quantization, while the difference is minimum, the pruned model shows more degradation than the non-pruned counterpart. For instance, DiverseVul and quantization 8 bits yields a degradation of 0.32% and the pruned one 0.59%. Similarly, in case of distinguishing vulnerable and no vulnerable samples results are almost equivalent to baseline ones with a degradation of F1 of less than 1%.

For Gemma model results are even more successfully, as there are more negative values, which shows improvement concerning baseline results. In some cases such as in DiverseVul and Big-Vul when using a rank of 1,024, the pruned model shows better performance than the baseline. When applying quantization, with 4 bits is the only configuration that should be discarded if layer pruning is not applied, as the degradation is significant, for instance, 22.38% of accuracy in PrimeVul. On the other hand, when distinguishing among vulnerable and no vulnerable samples, excluding the case of Big-Vul with a rank of 256, every other set-up and dataset shows a light improvement with respect to the baseline.

*Analysis per CWE.* Tables 4 and 5 show two confusion matrices of Gemma, for the sake of illustration (the remaining ones are available in our repository) for the DiverseVul and BigVul datasets. Among the common CWEs, CWE-119 and CWE-20 generally show worse results across the datasets and models, with precision usually below the mean of the other CWEs with precisions of 76%/78% in BERT and 76.5%/ 74.5% in Gemma.

CWE-119 refers to setting Improper Restrictions of Operations within the Bounds of a memory buffer, indicating that the models may fail when analyzing if the operations were properly restricted as the amount of possible improper operations is large. CWE-20 refers to improper Input Validation, suggesting that the models may lack knowledge of potential runtime inputs. Additionally, these two IDs are more general and abstract, encompassing more children classes.

While the mistakes across CWEs are low, these two CWEs in Gemma and BERT tend to be mixed, being some of the instances of the CWE-119 classified as CWE-20 and viceversa. Additionally, confusion matrices also shows

Table 4: Gemma, DiverseVul, no pruning, no quantization

CWE-119	623	72	42	45	8	38	27	38	12	41	54
CWE-125	36	714	17	32	10	8	51	39	17	55	21
CWE-190	24	31	795	30	11	19	16	27	3	25	19
CWE-20	40	30	23	687	18	51	38	41	5	19	48
CWE-22	2	3	4	12	326	1	8	2	5	4	3
CWE-362	11	3	16	28	6	853	37	18	1	4	23
CWE-416	20	37	27	18	13	51	740	28	16	21	29
CWE-476	35	50	9	43	12	30	52	686	8	55	20
CWE-78	2	3	1	12	3	0	14	2	264	1	3
CWE-787	56	48	15	40	2	10	39	40	8	718	24
No CWE	65	50	20	78	16	55	112	29	7	23	545
	CWE-119	CWE-125	CWE-190	CWE-20	CWE-22	CWE-362	CWE-416	CWE-476	CWE-78	CWE-787	No CWE

Table 5: Gemma, DiverseVul, with pruning and quantization

CWE-119	646	39	9	148	0	9	64	14	3	0	16
CWE-125	99	570	9	53	0	6	45	13	3	0	2
CWE-190	58	28	276	62	0	0	42	16	2	0	2
CWE-20	123	13	2	678	1	19	68	10	3	3	11
CWE-22	7	0	1	6	131	1	2	0	1	0	0
CWE-362	89	5	1	102	1	444	111	19	0	2	4
CWE-416	74	16	0	64	0	21	673	5	3	1	6
CWE-476	74	13	9	59	0	1	55	367	2	0	4
CWE-78	60	11	3	35	0	14	30	5	216	0	10
CWE-787	10	2	0	30	0	2	10	1	0	71	3
No CWE	200	18	4	291	0	56	102	7	10	0	279
	CWE-119	CWE-125	CWE-190	CWE-20	CWE-22	CWE-362	CWE-416	CWE-476	CWE-78	CWE-787	No CWE

that most of the miss-classifications are because the model shows lower effectiveness when discerning more these two abstract classes and the negative class.

CWE-787 and CWE-125, which refer to Out-of-bounds Write and Read respectively, show average precision across models and datasets with averages of 89.2%/86.1% in BERT and 87%/ 82.5% in Gemma. While these are specific types of operations within the broader class of CWE-119, they are more fine-grained, making them easier for the models to detect. The same applies to CWE-362 (race-condition) and to CWE-190 (integer overflow) – they are more concrete than CWE-119 and CWE-20.

*Corroborating  $k_{cut}$  appropriateness.* On the one hand, tests in Table 6 have been run using  $\lfloor k_{cut} \rfloor / 2$  as cut-off value, that is 5 in Gemma and 15 in BERT. Thus, the new cut-off value is 2 and 7 for Gemma and BERT respectively, such that 15 and 16 layers are removed in each case. All models’ metrics get worse, corroborating that the proposed approach achieves a nice reduction of the model, while keeping or improving the results of the original models.

Table 6: Additional layer pruning. Notice that a positive value means a decrement with respect to the baseline.

		Effectiveness				Vul- NoVul F1
		CWE				
Rank	Removed layers	Accuracy	F1	Precision	Recall	F1
Gemma (18 layers)						
DiverseVul						
256	0	73.40%	74.90%	75.30%	75.00%	95.90%
256	15	2,50	2,26	2,48	2,29	0,48
1024	15	2,82	3,01	3,04	2,64	0,19
PrimeVul						
256	0	88.10%	89.00%	88.80%	88.10%	99.50%
256	15	3,00	2,10	0,80	2,11	0,28
1024	15	4,20	3,63	3,23	2,60	0,17
Big-Vul						
256	0	77.00%	79.30%	78.80%	79.90%	95.30%
256	15	2,46	1,37	0,49	2,01	1,48
1024	15	6,96	5,28	5,46	4,53	2,18
BERT (24 layers)						
DiverseVul						
-	0	78.20%	79.60%	79.50%	79.60%	96.10%
-	16	4,1	3,73	4,5	3,27	1,02
PrimeVul						
-	0	86.10%	86.90%	87.10%	86.60%	99.20%
-	16	2	1,9	1,81	1,8	2,99
Big-Vul						
-	0	82.40%	85.10%	85.40%	84.70%	95.90%
-	16	1,44	1,05	1,43	0,28	0,55

On the other hand, a comparison against random layer removal (as done in [50]) has been carried out in Figure 4.  $k_{cut}$  random layers are removed, and repeated 3 times, from BERT and Gemma to analyse average results afterwards. Specially in Gemma results are quite worse removing random layers than using our proposed  $k_{cut}$  with a reduction of 20% of accuracy in the best case. BERT, though to a lesser degree, also presents a degradation of accuracy when applying random pruning, that is around 10% and 2.5% in the worst and best case respectively.

### 5.6.3. Time and memory efficiency

Tables 2 and 3 also show these performance metrics. For BERT, the memory required for full fine-tuning on a consumer-grade GPU is reduced by 27.7%, making it more accessible for lower-budget GPUs. The memory required for inference also decreases by around 15.41%, while 35.7% parameters and 26.32% the model size due to the need to store fewer weights. This also translates to better inference time, with an improvement across all datasets of 32%.

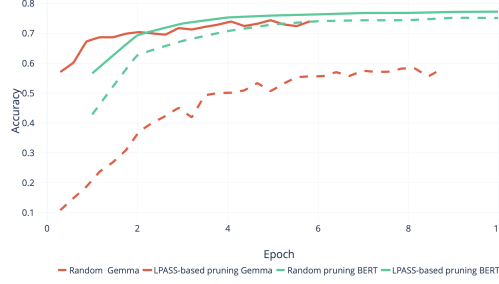


Figure 4: Comparison of random pruning vs *LPASS*-based pruning. Dotted line represents the random pruning version of their respective models

When BERT is quantized, inference time is slowed down, especially on 8-bit versions. Nonetheless, pruning alleviates this problem, reducing the slowdown from -846% (non-pruned DiverseVul) to -517% (pruned 8-bit DiverseVul). This could be due to the extra overhead and operations required by the 8-bit and 4-bit linear layers. Despite this increase in inference time, BERT model benefits from quantization by reducing the model size and the effective parameters further. A 4-bit pruned BERT reduces the original model size by 86.73% with a reduction of 164% in effective parameters. In terms of GPU memory, the requirements are significantly reduced, especially in the 8-bit versions, which, after pruning, are reduced by 30% across the datasets.

Gemma also shows a positive impact on its metrics thanks to pruning, reducing the training time by 71.36% in PrimeVul with a rank of 1,024. These gains are also seen in inference time, with reductions of 78%. While higher ranks require more GPU memory for fine-tuning, a rank of 1024 reduces memory usage by 34.98%, and a rank of 256 41%. Pruning the model also benefits memory usage for inference, which drops by 49%, primarily due to a 52.72% reduction in effective parameters. This results in model sizes being 57% smaller.

As with in previous model, quantizing the Gemma model translates to worse inference times. However, pruned Gemma alleviates this issue due to the smaller model complexity, reducing the increase from -2130.77% (8-bit PrimeVul non-pruned) to -223.08% (8-bit PrimeVul pruned). Additionally, quantized Gemma reduces the original model size by 72% in the 4-bit version, which can be explained by the reduction of effective parameters by around 68% across the datasets.

Overall, performance metrics show that training time is reduced by almost 30% in PrimeVul and Big-Vul and more than 50% in DiverseVul, while inference time decreases by 28% across all datasets and models. Model metrics also improve similarly across all datasets, with around a 30% improvement in each.

#### 5.6.4. Early assessment

The performance estimation process requires computing  $\mathcal{E}(D)$ . First, Table 7 presents values of  $\beta$  for combinations of  $\mathcal{D}$  applying the leave-one-out cross-validation – a couple of datasets are used for computing  $\beta$ , and the remaining one to compute  $\mathcal{E}(D)$ . Interestingly, it can be seen that  $\beta$  is reasonably stable for different datasets, which is beneficial for supporting the generalization of this approach.

Recalling Equation 5, the combination of  $\beta$  and  $\mathcal{E}(D)$  allows computing  $Err$  in each case. Table 8 depicts  $Err$  for all  $D$ , original models and compression configurations, namely original models, models with 4 and 8 bits quantization, after established layer pruning and pruning together with quantization.

BERT produces the best results with an  $Err$  between 8% and 4% and an average of 4.6% in CC and 7% in HD. On the other hand, Gemma presents a bit worse results, specially for CC in which the  $Err$  is between 17% and 11% and an average of 13.1%, while in CC the average  $Err$  in 10.2%. Moreover, there is not a clear distinction of LLM configurations as  $Err$  remains similar among them.

#### 5.7. Discussion and Limitations

Our results endorse the use of LPs to pick the cut-off layer when compressing LLMs by means of layer pruning. Considering the size of the LLMs at stake, as well as the comprehensiveness of the three chosen datasets, we believe that the validity of our findings is supported. In fact, the results obtained by compressed LLMs outperform the state of the art, as it will be further shown in Section 6. Interestingly, our compressed models improve the results as compared

Table 7: Values for  $\beta$  (in %) for each LLM setup phase

	Baseline			Quantization 4 bits			Quantization 8 bits			Layer pruning			L. Pruning+ Quant 4 bits			L. Pruning+ Quant 8 bits		
	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
<b>BERT</b>																		
<b>CC</b>																		
DiverseVul+ PrimeVul	38	38	38	38	38	38	38	38	38	38	38	38	37	37	37	38	38	38
PrimeVul+ Big-Vul	38	38	38	38	38	38	38	38	38	38	38	38	37	37	37	38	38	38
DiverseVul+ Big-Vul	37	37	37	36	37	37	37	37	37	36	37	36	35	36	35	36	36	36
<b>All D</b>	38	38	38	37	37	37	38	38	38	37	37	37	37	37	36	37	37	37
<b>HD</b>																		
DiverseVul+ PrimeVul	41	41	41	41	41	41	41	41	41	40	40	40	39	39	39	40	40	40
PrimeVul+ Big-Vul	41	40	40	37	40	40	41	40	41	40	40	40	40	40	40	40	40	40
DiverseVul+ Big-Vul	37	37	37	33	36	36	37	37	37	35	36	35	34	35	34	35	35	35
<b>All D</b>	40	40	39	37	39	39	40	39	40	38	39	38	38	38	38	38	38	38
<b>Gemma</b>																		
<b>CC</b>																		
DiverseVul+ PrimeVul	28	27	27	10	4	6	28	28	28	24	27	28	25	19	21	28	27	27
PrimeVul+ Big-Vul	24	24	24	13	2	2	25	24	24	28	26	27	20	10	12	26	26	26
DiverseVul+ Big-Vul	18	18	18	4	4	4	18	17	17	16	19	19	16	10	11	19	19	19
<b>All D</b>	23	23	23	9	3	4	24	23	23	22	24	25	20	13	15	24	24	24
<b>HD</b>																		
DiverseVul+ PrimeVul	35	34	34	17	6	8	35	34	35	32	36	37	34	28	30	37	36	36
PrimeVul+ Big-Vul	35	34	35	18	11	13	35	35	35	39	38	38	32	21	24	37	37	37
DiverseVul+ Big-Vul	27	26	26	7	6	8	26	26	26	25	29	29	25	20	21	28	29	28
<b>All D</b>	32	31	32	14	8	10	32	32	32	32	34	35	30	23	25	34	34	34

Table 8: *Err* (in %) per code feature  $F$  for each LLM setup phase. Datasets are used following a leave-one-out cross-validation. Only common CWEs are considered

			Baseline			Quantization 4 bits			Quantization 8 bits			Layer pruning			L. Pruning+ Quant 4 bits			L. Pruning+ Quant 8 bits		
LP datasets		Prediction datasets	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
BERT																				
CC	DiverseVul+ PrimeVul	Big-Vul	5	5	5	5	5	5	5	5	5	5	6	5	5	6	5	4	6	5
	PrimeVul+ Big-Vul	DiverseVul	5	5	5	5	5	5	5	5	5	5	6	6	6	7	6	6	6	6
	DiverseVul+ Big-Vul	PrimeVul	3	3	3	4	3	3	3	3	3	4	3	3	5	4	4	4	4	4
		Mean	5	4	4	5	4	4	4	4	4	5	5	5	5	6	5	5	5	5
HD	DiverseVul+ PrimeVul	Big-Vul	6	5	5	6	6	5	6	5	5	4	6	5	5	6	5	5	5	5
	PrimeVul+ Big-Vul	DiverseVul	6	6	6	5	6	6	6	6	6	8	9	8	8	9	8	8	9	8
	DiverseVul+ Big-Vul	PrimeVul	8	7	8	12	8	8	8	7	8	10	9	9	10	10	10	9	9	9
		Mean	7	6	6	8	7	7	7	6	6	7	8	7	8	8	8	7	7	7
Gemma																				
CC	DiverseVul+ PrimeVul	Big-Vul	14	13	13	15	16	11	14	14	14	11	10	10	17	19	18	12	11	11
	PrimeVul+ Big-Vul	DiverseVul	5	9	6	13	16	11	7	7	7	16	9	9	6	13	9	8	9	8
	DiverseVul+ Big-Vul	PrimeVul	16	15	16	17	19	11	17	17	17	20	16	16	17	14	12	15	14	15
		Mean	11	13	12	15	17	11	13	13	13	16	12	12	14	15	13	12	12	12
HD	DiverseVul+ PrimeVul	Big-Vul	8	6	6	10	10	6	7	6	7	5	6	5	12	13	12	6	7	6
	PrimeVul+ Big-Vul	DiverseVul	7	9	8	11	14	13	8	8	8	16	9	9	5	7	5	8	9	8
	DiverseVul+ Big-Vul	PrimeVul	13	13	13	17	16	9	15	14	15	17	13	13	17	13	12	13	12	13
		Mean	9	10	9	13	13	9	10	10	10	13	9	9	11	11	10	9	9	9

to the original, non-compressed versions. As pointed out in [16], model-level optimizations such as compression or pruning tend to come with a loss in performance. Nonetheless, our results show that pruning a model for a specific task and optimizing it for a really specific downstream task, such as code vulnerability detection, can even lead to better results.

In the same vein, LPs have been shown to be effective for having an early (yet accurate) estimate on the post-fine-tuning and post-compression model performance. Estimation errors are affordable, indeed. However, we opted for using a linear relationship between LP results and the target values (recall Equation 3), as the amount of instances is limited. Therefore, exploring non-linear relationships could lead to better results and is an interesting research direction.

LPs are also affordable and cost-effective. The training time per layer for the LPs in BERT is, on average, 0.05 seconds per layer across the three datasets, compared to the 6,072 seconds it takes on average to fine-tune BERT across the datasets. For Gemma, only 1.49 seconds per layer is needed for LPs, in contrast with the 5,162 seconds required on average to fine-tune the model. For each sample, the time required to extract the features is 0.007 seconds for BERT and 0.009 seconds for Gemma. While the final time required to compute a single round of the LPs may depend on the number of samples used, 98.18, 21.92, and 41.69 seconds were required with BERT for DiverseVul, PrimeVul, and BigVul, respectively. For Gemma, 142.97, 46.71, and 79.04 seconds were needed for the same datasets. This represents a reduction of 99.11% and 98.26% in the average time needed to fine-tune the models, respectively.

On the contrary, our results are not enough to confirm that LPs can be directly applied to other domains (such as speech recognition or NLP-related tasks). Indeed, choosing which LPs and features to apply is not immediate. Hence, this work must be regarded as a first attempt in this direction, opening interesting research venues to ascertain their use in other domains or with different LLMs.

Our experimental settings impose a number of natural limits to our results. Thus, only C/C++ programming languages are at stake, but additional ones could be tested. Similarly, we have resorted to a classic LLM (Bert) and a very novel one (Gemma). However, others like Gamma or GPT-4o, could also be relevant.

Regarding other compression methodologies, such as merging layers or knowledge distillation, we consider these techniques complementary to our approach. We could use a larger teacher model to enhance the representations of our pruned models or further compress our pruned model by merging similar layers. These are interesting lines of research to explore if LPs can be used in addition to these methods to create smaller models while leveraging the insights from the hidden states of the LLMs.

## 6. Related work

Vulnerability detection using LLMs presents a critical challenge in the development of tools that assist security analysts in writing or auditing code. The vast array of existing vulnerabilities, including zero-day threats, exacerbates this challenge due to the potentially infinite spectrum of vulnerabilities [7].

For this reason, academia has extensively explored various proposals. General models such as GPT-4, Chat-GPT, and LLama2 have been studied [52, 43] to evaluate their effectiveness in classifying vulnerabilities across binary and multi-class frameworks [54], specifically within C, Java [55], and Python code [57], utilizing diverse prompting strategies. Ahmad et al. [51] studied these strategies to find vulnerabilities at a line-level approach, showing promising results.

However, despite these efforts, the studies indicate that the current state of the art in prompt-based models underscores the necessity for specialized models tailored for vulnerability detection, particularly as current models struggle with accurately identifying the potential CWE of the code (see [58] for more).

Among specialised models, finetuned LLMs have been the option that has lead to better results. Chen et al. [1] covered a wide range of models' families (RoBERTa, T5, GPT-2) while introducing a new diverse C code vulnerability dataset. Fu et al. [4] studied the possibility of predicting vulnerabilities on a line based approach. However, the problem was covered as a binary classification problem and thus, models lack the ability of correctly discerning the CWE type. They concluded that diversity of data sources, vulnerabilities and pre-training on code data are crucial for better detection while larger models seems not to be beneficial. This was later also pointed by Steenhoek et al. [5] who carried out an empirical study for vulnerability detection with fine-tuned LLMs.

Hanif et al. [6] explored to pre-train and finetune a model (RoBERTa) specifically for code vulnerability detection. They reach good results in both, binary and multi-class vulnerability detection. Nonetheless, pre-training this

Table 9: Vulnerability detection using LLMs. Notice column Binary (B)/ Multiclass (M) marks if the set-up was binary or multiclass.

Reference	Programming language	Dataset	Model	Results	CWE-level analysis	Binary (B)/ Multiclass (M)	Input features	Max tokens	Compression technique(s)
Fu et al. [4]	C/C++	Big-Vul	Custom transformer	Big-Vul: 65% Acc.	✓	B	Code	512	×
Steenhoek et al. [5]	C/C++	Devign, MSR	Vulberta and other custom transformers models	Devign: 55%-89% F1	×	B	Code	512	×
Ahmad et al. [51]	C/C++, Verilog, Python	Custom dataset	Codex, GPT-3.5	Custom dataset: 66% Acc.	✓	B	Code + prompt	50 lines	×
Zhou et al. [43]	C/C++	Vulnerability-fixing commit dataset	GPT-3.5 GPT-4	VFC: 75.5% Acc.	×	B	Prompt + code	4,096	×
Fu et al. [52]	C/C++	Big-Vul	Chat-GPT	Big-Vul with prompts: 13%-20, M. 10% Acc., B% Big-Vul fine-tuned LLM: 65% Acc., M. 94% Acc., B	×	B and M	Prompt + code	4,096	×
Fu et al. [53]	C/C++	Big-Vul	CodeBERT, Devign, ReGVDm Graph-CodeBERT, LFME, BAGS	Big-Vul: 64% Acc.	✓	M	Code	512	×
Du et al. [2]	C/C++	Big-Vul	GraphCodeBert	Big-Vul: 93.83 % Acc.	×	B	Code	512	×
Chen et al. [1]	C/C++	diverseVul, Devign, Re-Veal, Big-Vul, CrossVul, CVEFixes (Jointly)	Roberta, GPT-2, T5	CVEFixes: 91.64 % Acc. Devign, ReVeal, Big-Vul, CrossVul: 92.30% Acc. DiverseVul: 92.30 % Acc. (Best effort)	✓	B	Code	512	×
Gao et al. [54]	C/C++	d2a, ctf, magma, big-vul, and devign	chatglm2 -6b Llama-2 -7b vicuna -7b vicuna -7b-16k Llama-2 -13b Baichuan2 -13b vicuna -13b vicuna -13b-16k internlm -20b vicuna -33b CodeLlama -34b falcon -40b Llama-2 -70b Platypus2 -70b GPT-3.5 GPT-4	Joint datasets: 40.6 % F1, B / 37.9 % F1,M	✓	B and M	Prompt + code	2,048	×
Ding et al. [3]	C/C++	Primevul	Codet5, codebert, unixcoder, star-coder2,codegen, gpt-3.5 Gpt4	PrimeVul: 96% Acc.	×	B	Code	512	×
Hanif et al. [6]	C/C++	Pre-training (GitHub,Draper) + finetuning on Vuldeep-ecker, Reveal, Draper, muVuldeepecker, D2A,Devign	Vulberta	Joint datasets: 99.59% F1	✓	M	Code	512	×
Tamberg et al. [55]	Java	Java Juliet 1.3	CodeQL, GPT-4, CLaude	Java Juliet: 72% (max) Acc.	✓	B	Code	-	×
Shestov et al. [56]	Java	VCMatch (custom)	WizardCoder, coderbert	VCMatch: 75% - 85% ROC	×	B	Code	2,048 and 512	×
Jensen et al. [57]	Python	HumanEval, MBPP, SecurityEval 660 between all	Falcon-7b, Llama, llama2, dolly	Joint datasets: 95.6% Acc. with 37.9% F1	×	B	Prompt + code	4,096	×
Shi et al. [17]	C/C++	Devign	CodeBERT	Devign: 59% Acc.	×	B	Code	512	✓ Knowledge distillation
<b>Ours</b>	C/C++	Big-Vul, DiverseVul, PrimeVul	BERT, Gemma	Big-Vul: 82%-96% Acc. DiverseVul: 77.7%-96% Acc. PrimeVul: 87.1%-99% Acc.	✓	B and M	Code	1,024	✓ Layer pruning and quantization, guided by Linear Probes

model requires 96 hours and 80 GB of VRAM, which are usually resources not generally accessible. Additionally, no vulnerable samples are ignored in the multiclass approach, training the model for just distinguishing among the different CWEs. The characterization of CWE was covered by [53] who proposed a set of abstract classes to reduce the complexity of fine-grained predictions by grouping similar CWE.

Among compression proposals, only Shi et al. [17, 18] have addressed the topic. They propose a two-step method to find optimal model configurations for a base size BERT model (CodeBERT). First, they maximize accuracy while reducing model size using a genetic algorithm. Afterward, they apply knowledge distillation to the optimized model. However, this approach results in an accuracy of only 59% for binary code vulnerability detection, which is close to random guessing. The trial-and-error methodology for testing configurations could be impractical due to the complex hyperparameter search space, especially for larger models like those discussed in our paper.

Their solution requires a notable amount of time. In the first step, they need to limit the search space, which takes 5 minutes on a cluster with 80 CPUs and 504 GB of RAM. The resultant set of 20 models needs to be pre-trained, taking a minimum of another 10 hours. Finally, the fine-tuning process must be carried out, requiring an additional 20 minutes. This process is both time and energy-consuming. As pointed out in [20], while fine-tuning consumes significant energy and generates emissions, pre-training the models is the most resource-intensive step. In our solution, no additional pre-training is required, thereby saving both energy and time.

Interestingly, our results suggests that compression techniques can also lead to improved performance results, as opposed to the findings by Shi et al. which characterize the incurred loss.

Table 9 presents a comparison among works leveraging LLMs for vulnerability detection, pointing out the language of the code, datasets, models, main results, if the study is carried out in a binary or multi-class classification considering CWE or not, together with input features and tokens size. *LPASS* encompasses three C/C++ datasets, the most common programming languages, and distinguishes among various CWEs, doing a multi-class classification and determining also whether code samples are not vulnerable. The average accuracy that we achieve across datasets for multi-class classification is 81% for BERT and 76.6% for Gemma, both of which outperform other approaches reported in the literature [52, 53]. Specifically, on the Big-Vul dataset, BERT and Gemma achieve accuracies of 64% and 65%, respectively, which are lower than our best result of 82%. For binary classification, our results can be compared with studies that employ related datasets [1, 2, 4, 3, 52]. On the Big-Vul dataset, our approach achieves an accuracy of 96%, surpassing the results of [2, 4, 52], which reported accuracies ranging from 65% to 94%, with a maximum of 93.83%. Additionally, the work by Chen et al. [1] on DiverseVul yields an accuracy of 92.30%, which is lower than our performance of 96%. Finally, [3] reports an accuracy of 96% on the PrimeVul dataset, while our approach achieves 99%. However, proposals in [55, 56, 57] are not comparable to our work, as they focused on Python and Java. Additionally, while other efforts such as [17, 6, 54, 7, 51] did focus on vulnerability detection in C, like our study, they use datasets like Devign or others that only indicate whether a vulnerability is present or not and then, as pointed out in Section 5.2, their use has been discarded in our work.

Moreover, proposals for fine-tuned models [6, 1, 53] are limited to 512 tokens, which is the maximum context window supported by most models. By contrast, we test a well-known model, BERT, and Gemma, not used until now, with 1,024 tokens in this latter case and using code as input features. Additionally, the model at stake in the work by Shi et al. [17] is already smaller than ours. Thus, it is unknown how well their proposal may work with larger models such ours as their approach may result in a substantially larger search space for their algorithms.

## 7. Conclusion

Reducing the size of LLMs is critical to ensure their scalability for vulnerability detection – as real-time demands come into play, saving time and memory is paramount. However, model compression and fine-tuning requires non-negligible resources, and the effect of these actions into the model performance is unknown beforehand. To address these issues, in this paper an approach (dubbed *LPASS*) has been proposed to assist in making informed decisions. *LPASS* helps on selecting the cut-off point for model compression using layer pruning. Moreover, it provides a good estimate of the post-fine-tuning and post-compression performance by using linear classifier probes. Indeed, *LPASS*-based versions of two LLMs have not only outperformed the state-of-the-art in vulnerability detection, but also non-compressed versions, thus showing the suitability of using probes.

Our results open a number of future research directions. On the one hand, our use of linear classifier probes had not been proven in the context of model compression. Thus, the analysis on their suitability for other models



or application domains is relevant. On the other hand, in the field of vulnerability detection, the suitability for other languages remains an open issue.

## Declarations

### *Ethical Approval and Consent to participate*

The authors declare they have no conflict of interest.

### *Consent for publication*

Not applicable

### *Availability of supporting data*

Datasets are public. Our experimental code will be freely available if the paper is accepted for publication.

### *Competing interests/Authors' contributions*

The authors declare that they have no competing interests.

### *Funding*

Nicolas Anciaux was supported by the French grant iPoP PEPR (ANR-22-PECY-0002). Luis Ibanez-Lissen was supported, and also Lorena Gonzalez partially, by the Spanish National Cybersecurity Institute (INCIBE) grant APAMciber within the framework of the Recovery, Transformation and Resilience Plan funds, financed by the European Union (Next Generation). Jose Maria de Fuentes was partially supported by grant PID2023-150310OB-I00 of the Spanish AEI. Jose Maria de Fuentes and Lorena Gonzalez have also received support from UC3M's Requalification programme, funded by the Spanish Ministerio de Ciencia, Innovacion y Universidades with EU recovery funds (Convocatoria de la Universidad Carlos III de Madrid de Ayudas para la recualificación del sistema universitario español para 2021-2023, de 1 de julio de 2021).

## References

- [1] Y. Chen, Z. Ding, L. Alowain, X. Chen, D. Wagner, Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection, in: Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses, 2023, pp. 654–668.
- [2] X. Du, S. Zhang, Y. Zhou, H. Du, A vulnerability severity prediction method based on bimodal data and multi-task learning, Journal of Systems and Software 213 (2024) 112039.
- [3] Y. Ding, Y. Fu, O. Ibrahim, C. Sitawarin, X. Chen, B. Alomair, D. Wagner, B. Ray, Y. Chen, Vulnerability Detection with Code Language Models: How Far Are We? , in: 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE), IEEE Computer Society, Los Alamitos, CA, USA, 2025, pp. 469–481.
- [4] M. Fu, C. Tantithamthavorn, Linevul: A transformer-based line-level vulnerability prediction, in: Proceedings of the 19th International Conference on Mining Software Repositories, 2022, pp. 608–620.
- [5] B. Steenhoek, M. M. Rahman, R. Jiles, W. Le, An empirical study of deep learning models for vulnerability detection, in: 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE), IEEE, 2023, pp. 2237–2248.
- [6] H. Hanif, S. Maffei, Vulberta: Simplified source code pre-training for vulnerability detection, in: 2022 International joint conference on neural networks (IJCNN), IEEE, 2022, pp. 1–8.
- [7] X. Zhout, K. Kim, B. Xu, J. Liu, D. Han, D. Lo, The devil is in the tails: How long-tailed code distributions impact large language models, in: 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, 2023, pp. 40–52.
- [8] F. Lomio, E. Iannone, A. De Lucia, F. Palomba, V. Lenarduzzi, Just-in-time software vulnerability detection: Are we there yet?, Journal of Systems and Software 188 (2022) 111283.
- [9] B. Chernis, R. Verma, Machine learning methods for software vulnerability detection, in: Proceedings of the fourth ACM international workshop on security and privacy analytics, 2018, pp. 31–39.
- [10] W. Tang, M. Tang, M. Ban, Z. Zhao, M. Feng, Csgvd: A deep learning approach combining sequence and graph embedding for source code vulnerability detection, Journal of Systems and Software 199 (2023) 111623.
- [11] D. Hin, A. Kan, H. Chen, M. A. Babar, Linevd: Statement-level vulnerability detection using graph neural networks, in: Proceedings of the 19th International Conference on Mining Software Repositories, 2022, pp. 596–607.
- [12] C. Do Xuan, D. H. Mai, M. C. Thanh, B. Van Cong, A novel approach for software vulnerability detection based on intelligent cognitive computing, The Journal of Supercomputing (2023) 1–37.
- [13] Z. Sheng, Z. Chen, S. Gu, H. Huang, G. Gu, J. Huang, Large language models in software security: A survey of vulnerability detection techniques and insights, arXiv preprint arXiv:2502.07049.

- [14] A. Gholami, S. Kim, Z. Dong, Z. Yao, M. W. Mahoney, K. Keutzer, A survey of quantization methods for efficient neural network inference, in: *Low-Power Computer Vision*, Chapman and Hall/CRC, 2022, pp. 291–326.
- [15] Y. He, L. Xiao, Structured pruning for deep convolutional neural networks: A survey, *IEEE Transactions on Pattern Analysis and Machine Intelligence*.
- [16] Z. Zhou, X. Ning, K. Hong, T. Fu, J. Xu, S. Li, Y. Lou, L. Wang, Z. Yuan, X. Li, et al., A survey on efficient inference for large language models, *arXiv preprint arXiv:2404.14294*.
- [17] J. Shi, Z. Yang, B. Xu, H. J. Kang, D. Lo, Compressing pre-trained models of code into 3 mb, in: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.
- [18] J. Shi, Z. Yang, H. J. Kang, B. Xu, J. He, D. Lo, Greening large language models of code, in: *Proceedings of the 46th international conference on software engineering: software engineering in society*, 2024, pp. 142–153.
- [19] G. Hinton, O. Vinyals, J. Dean, Distilling the knowledge in a neural network, *arXiv preprint arXiv:1503.02531*.
- [20] X. Wang, C. Na, E. Strubell, S. Friedler, S. Luccioni, Energy and carbon considerations of fine-tuning bert, in: *Findings of the Association for Computational Linguistics: EMNLP 2023*, 2023, pp. 9058–9069.
- [21] P. Jiang, C. Sonne, W. Li, F. You, S. You, Preventing the immense increase in the life-cycle energy and carbon footprints of llm-powered intelligent chatbots, *Engineering*.
- [22] S. Chen, Q. Zhao, Shallowing deep networks: Layer-wise pruning based on feature representations, *IEEE transactions on pattern analysis and machine intelligence* 41 (12) (2018) 3048–3056.
- [23] G. Alain, Y. Bengio, Understanding intermediate layers using linear classifier probes, *arXiv preprint arXiv:1610.01644*.
- [24] M. Jin, Q. Yu, J. Huang, Q. Zeng, Z. Wang, W. Hua, H. Zhao, K. Mei, Y. Meng, K. Ding, et al., Exploring concept depth: How large language models acquire knowledge and concept at different layers?, in: *Proceedings of the 31st International Conference on Computational Linguistics*, 2025, pp. 558–573.
- [25] G. Ahdritz, T. Qin, N. Vyas, B. Barak, B. L. Edelman, Distinguishing the knowable from the unknowable with language models, in: *Proceedings of the 41st International Conference on Machine Learning, ICMU’24, JMLR.org*, 2024.
- [26] X. Zhu, J. Li, Y. Liu, C. Ma, W. Wang, A survey on model compression for large language models, *Transactions of the Association for Computational Linguistics* 12 (2024) 1556–1577.
- [27] J. Devlin, M.-W. Chang, K. Lee, K. Toutanova, Bert: Pre-training of deep bidirectional transformers for language understanding, *arXiv preprint arXiv:1810.04805*.
- [28] G. Team, T. Mesnard, C. Hardin, R. Dadashi, S. Bhupatiraju, S. Pathak, L. Sifre, M. Rivière, M. S. Kale, J. Love, et al., Gemma: Open models based on gemini research and technology, *arXiv preprint arXiv:2403.08295*.
- [29] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, D. Kalenichenko, Quantization and training of neural networks for efficient integer-arithmetic-only inference, in: *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 2704–2713.
- [30] J. Fan, Y. Li, S. Wang, T. N. Nguyen, Ac/c++ code vulnerability dataset with code changes and cve summaries, in: *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 508–512.
- [31] S. Christey, J. Kenderdine, J. Mazella, B. Miles, Common weakness enumeration, Mitre Corporation.
- [32] Y. LeCun, J. Denker, S. Solla, Optimal brain damage, *Advances in neural information processing systems* 2.
- [33] S. Han, J. Pool, J. Tran, W. Dally, Learning both weights and connections for efficient neural network, *Advances in neural information processing systems* 28.
- [34] K. Marchisio, S. Dash, H. Chen, D. Aumiller, A. Üstün, S. Hooker, S. Ruder, How does quantization affect multilingual llms?, *arXiv preprint arXiv:2407.03211*.
- [35] J. Wallat, F. Beringer, A. Anand, A. Anand, Probing bert for ranking abilities, in: *European Conference on Information Retrieval*, Springer, 2023, pp. 255–273.
- [36] H. Duan, Y. Yang, K. Y. Tam, Do llms know about hallucination? an empirical investigation of llm’s hidden states, *arXiv preprint arXiv:2402.09733*.
- [37] A. Karmakar, R. Robbes, What do pre-trained code models know about code?, in: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2021, pp. 1332–1336.
- [38] B. Steenhoek, M. M. Rahman, S. Sharmin, W. Le, Do language models learn semantics of code? a case study in vulnerability detection, *arXiv preprint arXiv:2311.04109*.
- [39] F. Manigrasso, S. Schouten, L. Morra, P. Bloem, Probing llms for logical reasoning, in: *International Conference on Neural-Symbolic Learning and Reasoning*, Springer, 2024, pp. 257–278.
- [40] J. Hoscilowicz, A. Wiacek, J. Chojnacki, A. Cieslak, L. Michon, V. Urbanevych, A. Janicki, Non-linear inference time intervention: Improving llm truthfulness, *arXiv preprint arXiv:2403.18680*.
- [41] M. Abbas, Y. Zhou, P. Ram, N. Baracaldo, H. Samulowitz, T. Salonidis, T. Chen, Enhancing in-context learning via linear probe calibration, in: *International Conference on Artificial Intelligence and Statistics*, PMLR, 2024, pp. 307–315.
- [42] H. Papadatos, R. Freedman, Linear probe penalties reduce llm sycophancy, *arXiv preprint arXiv:2412.00967*.
- [43] X. Zhou, T. Zhang, D. Lo, Large language model for vulnerability detection: Emerging results and future directions, in: *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, 2024, pp. 47–51.
- [44] I. Herraiz, A. E. Hassan, Beyond lines of code: Do we need more complexity metrics, *Making software: what really works, and why we believe it* (2010) 125–141.
- [45] M. Zagane, M. K. Abdi, M. Alenezi, Deep learning for software vulnerabilities detection using code metrics, *IEEE Access* 8 (2020) 74562–74570.
- [46] C. Ebert, J. Cain, G. Antoniol, S. Counsell, P. Laplante, Cyclomatic complexity, *IEEE software* 33 (6) (2016) 27–29.
- [47] R. Mohammed, J. Rawashdeh, M. Abdullah, Machine learning with oversampling and undersampling techniques: overview study and experimental results, in: *2020 11th international conference on information and communication systems (ICICS)*, IEEE, 2020, pp. 243–248.
- [48] D. P. Kingma, J. Ba, Adam: A method for stochastic optimization, *arXiv preprint arXiv:1412.6980*.

- [49] J. Zhao, Z. Zhang, B. Chen, Z. Wang, A. Anandkumar, Y. Tian, Galore: memory-efficient llm training by gradient low-rank projection, in: Proceedings of the 41st International Conference on Machine Learning, ICML'24, JMLR.org, 2024.
- [50] H. Sajjad, F. Dalvi, N. Durrani, P. Nakov, On the effect of dropping layers of pre-trained transformer models, *Computer Speech & Language* 77 (2023) 101429.
- [51] B. Ahmad, B. Tan, R. Karri, H. Pearce, Flag: Finding line anomalies (in code) with generative ai, arXiv preprint arXiv:2306.12643.
- [52] M. Fu, C. K. Tantithamthavorn, V. Nguyen, T. Le, Chatgpt for vulnerability detection, classification, and repair: How far are we?, in: 2023 30th Asia-Pacific Software Engineering Conference (APSEC), IEEE, 2023, pp. 632–636.
- [53] M. Fu, V. Nguyen, C. K. Tantithamthavorn, T. Le, D. Phung, Vulexplainer: A transformer-based hierarchical distillation for explaining vulnerability types, *IEEE Transactions on Software Engineering*.
- [54] Z. Gao, H. Wang, Y. Zhou, W. Zhu, C. Zhang, How far have we gone in vulnerability detection using large language models, arXiv preprint arXiv:2311.12420.
- [55] K. Tamberg, H. Bahsi, Harnessing large language models for software vulnerability detection: A comprehensive benchmarking study, *IEEE Access*.
- [56] A. Shestov, R. Levichev, R. Mussabayev, E. Maslov, P. Zadorozhny, A. Cheshkov, R. Mussabayev, A. Toleu, G. Tolegen, A. Krassovitskiy, Finetuning large language models for vulnerability detection, *IEEE Access*.
- [57] R. I. T. Jensen, V. Tawosi, S. Almir, Software vulnerability and functionality assessment using large language models, in: Proceedings of the Third ACM/IEEE International Workshop on NL-Based Software Engineering, NLBSE '24, Association for Computing Machinery, New York, NY, USA, 2024, p. 25–28.
- [58] Y. Yao, J. Duan, K. Xu, Y. Cai, Z. Sun, Y. Zhang, A survey on large language model (llm) security and privacy: The good, the bad, and the ugly, *High-Confidence Computing* 4 (2) (2024) 100211.