How the Training Procedure Impacts the Performance of Deep Learning-based Vulnerability Patching

Antonio Mastropaolo
Università della Svizzera italiana (USI)
Lugano, Switzerland
antonio.mastropaolo@usi.ch

Vittoria Nardone
University of Molise
Campobasso, Italy
vittoria.nardone@gmail.com

Gabriele Bavota
Università della Svizzera italiana (USI)
Lugano, Switzerland
gabriele.bavota@usi.ch

Massimiliano Di Penta
University of Sannio
Benevento, Italy
dipenta@unisannio.it

ABSTRACT

Generative deep learning (DL) models have been successfully adopted for vulnerability patching. However, such models require the availability of a large dataset of patches to learn from. To overcome this issue, researchers have proposed to start from models pre-trained with general knowledge, either on the programming language or on similar tasks such as bug fixing. Despite the efforts in the area of automated vulnerability patching, there is a lack of systematic studies on how these different training procedures impact the performance of DL models for such a task. This paper provides a manyfold contribution to bridge this gap, by (i) comparing existing solutions of self-supervised and supervised pre-training for vulnerability patching; and (ii) for the first time, experimenting with different kinds of prompt-tuning for this task. The study required to train/test 23 DL models. We found that a supervised pre-training focused on bug-fixing, while expensive in terms of data collection, substantially improves DL-based vulnerability patching. When applying prompt-tuning on top of this supervised pre-trained model, there is no significant gain in performance. Instead, prompt-tuning is an effective and cheap solution to substantially boost the performance of self-supervised pre-trained models, *i.e.,* those not relying on the bug-fixing pre-training.

CCS CONCEPTS

• **Security and privacy → Software security engineering**.

KEYWORDS

Software Vulnerability Repair, Pre-Trained Models, Machine Learning on Code, Prompt Tuning

ACM Reference Format:
Antonio Mastropaolo, Vittoria Nardone, Gabriele Bavota, and Massimiliano Di Penta. 2024. How the Training Procedure Impacts the Performance of Deep Learning-based Vulnerability Patching. In *28th International Conference on Evaluation and Assessment in Software Engineering (EASE 2024), June 18–21, 2024, Salerno, Italy.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3661167.3661200

1 INTRODUCTION

The number of reported software vulnerabilities is increasing year after year [3]. Such a growing trend can be explained by several factors. The most obvious is the increasing number of software projects. For example, GitHub counts at date over 370M repositories [4]. Additionally, since software projects often incorporate third-party libraries and components, they are frequently impacted by bugs. Security vulnerabilities represent a specific type of bug that can severely affect various aspects of the systems, as recently demonstrated by the Log4j vulnerability [1].

To aid developers, researchers proposed solutions for the automated identification [9, 27, 34, 35, 50, 57, 58] and patching [5, 10, 11, 19, 29, 31, 39, 43] of vulnerabilities. For the latter task, Deep Learning (DL) models achieved state-of-the-art results thanks to their ability to infer fixing patterns by learning from concrete examples of vulnerability fixes that can be mined from software repositories. However, as highlighted by Chen *et al.* [10], a major obstacle to the adoption of DL for this task is the lack of training data. Indeed, when it comes to generative tasks (such as *generating* the patched code), DL models require large supervised datasets: In this case, thousands of examples of vulnerable and fixed code.

One way to overcome the lack of data is to rely on pre-trained models. These models are trained in two steps, namely pre-training and fine-tuning.

In the context of vulnerability patching, Chi *et al.* [11] and Chen *et al.* [10] exploited the idea of supervised pre-training, using the task of fixing generic bugs as a pre-training objective. They showed that providing the model with such a "bug-fixing knowledge" before specializing it in vulnerability patching helps in boosting performance. More recently, Fu *et al.* [19] showed instead that such a supervised pre-training can be beaten by a large-scale self-supervised pre-training based on the *masked language model* objective.

Besides pre-training, another strategy to overcome the lack of training data is prompt-tuning [6, 12, 25, 32, 33, 37, 48, 49], namely the idea of transforming the training objective of the downstream task (*i.e.,* fine-tuning) into one that resembles the pre-training stage, and, therefore could potentially re-use information acquired during such a phase. For the vulnerability fix task, a classic fine-tuning would provide the model with a vulnerable code as input asking it

to generate its patched version. A possible prompt-tuning could, for example, provide the model with the following input:
Generate a patch for this vulnerable code {$vulnerable_{code}$} **as follows:** {$patch$} where {$vulnerable_{code}$} and {$patch$} are placeholders to fill in each training instance.

In summary, there are works suggesting the benefits of a supervised pre-training for vulnerability patching [10, 11], others reporting the superiority of a self-supervised pre-training [19], and very recent work that experimented with prompt-tuning for other software engineering tasks [53]. However, there is a lack of a systematic study comparing and contrasting these training strategies (and their combinations) for the task of vulnerability patching.

This paper aims to bridge this gap, by conducting a large experimentation on the different training strategies that can be used in the context of vulnerability patch generation. To pose the basis for the study, we first replicated existing state-of-the-art work on DL-based vulnerability patching, namely the work by Fu *et al.* [19] which showed the superiority of self-supervised pre-training with respect to a specialized (supervised) bug-fixing pre-training [10]. While conducting such a replication, we uncovered the existence of "token-by-token" duplicates within the training and the test set employed by Fu *et al.* [19], implying a possible inflation of the reported performance. We thus cleaned the dataset, re-trained the model, and observed a substantial drop in performance as compared to what was reported by Fu *et al.* [19]. We then experimented on the same cleaned dataset with several different training procedures including (i) lack of pre-training, *i.e.,* the model is directly fine-tuned for the task of vulnerability patching; (ii) self-supervised pre-training and standard fine-tuning, representative of the replicate work [19]; (iii) supervised pre-training based on bug-fixing and standard fine-tuning, representative of previous works proposing such an idea [10, 11] and; (iv) ten different types of prompt fine-tuning were performed on top of the pre-trained models (both those with self-supervised and supervised pre-training strategies).

Overall, our study required the training and testing of 23 models. Our main findings show that: (i) given the scarcity in fine-tuning data (*i.e.,* real-world vulnerability patches), pre-training is always beneficial for the task of vulnerability patching, independently of whether it is self-supervised or supervised; (ii) unsurprisingly, supervised pre-training (with bug fixes) is superior to the self-supervised one, contradicting previous findings [19]; (iii) in the context of vulnerability patching, prompt fine-tuning is a cheap mechanism to substantially boost the performance of self-supervised pre-trained models, while it does not really help models that benefited of supervised pre-training as they "lost knowledge" about the natural language on which they were pre-trained.

2 RELATED WORK

In this section, we discuss related work about automated vulnerability patching, as well as background notions on prompt-tuning.

2.1 Learning-Based Vulnerability Fixing

Researchers proposed several approaches to automatically generate fixes for software vulnerabilities. Some of these approaches are not based on machine learning, but rather leverage other techniques, such as property-based approaches [29, 36], formal methods [5, 43], static analysis [21, 31] and dynamic analysis [47, 54]. Given the focus of our work, we narrow down the discussion of these to ML-based approaches for vulnerability fixing.

Ma *et al.* [39] proposed VuRLE (*i.e.,* Vulnerability Repair by Learning from Examples), an automatic tool for vulnerability detection and repair that leverages machine learning to identify and fix software vulnerabilities. VuRLE consists of a learning component and a repair component. The former collects a dataset of vulnerable and non-vulnerable code examples to train a shallow machine learning model on various features, such as code structure, control flow, and data flow. The latter chooses the most appropriate template to fix the detected vulnerability. While VuRLE can handle a wide range of vulnerabilities, it has been evaluated on a small dataset of 279 vulnerabilities.

Harer *et al.* [26] used Generative Adversarial Networks (GANs) [22] to automatically repair software vulnerabilities. Both the generator and the discriminator network use a standard Neural Machine Translation (NMT) model. However, while the generator tries to generate patches, the discriminator network attempts to distinguish the crafted from the real ones.

Chi *et al.* [11] and Chen *et al.* [10] proposed a neural Sequence-to-Sequence (Seq2Seq) approach for vulnerability patching. Both approaches are pre-trained on a bug-fix corpus and then fine-tuned on a real-world vulnerability fix dataset. The intuition behind these techniques is that the knowledge gained by the model when fixing bugs (supervised pre-training) can be transferred to the generation of patches for vulnerable code. Although both approaches use a neural Sequence-to-Sequence (Seq2Seq) learning model to generate fixed code sequences, they differ in the programming language they handle (*Java* *vs* C) and the type of vulnerabilities the model can patch (inline *vs* multi-line).

To the best of our knowledge, the latest technique proposed for vulnerability patching is VulRepair, introduced by Fu *et al.* [19]. This approach leverages the T5 model introduced by Raffel *et al.* [49]. Rather than starting from randomly initialized weights, Fu *et al.* decided to build VulRepair employing CodeT5 as a pre-trained model. This design choice reduces the burden of pre-training a new T5 model from scratch and leverages the condensed knowledge of the model. VulRepair has been evaluated on two different datasets (CVEFixes [7] and Big-Vul [16]) which consist of 8,482 pairs of vulnerability fixes in total, and for which Fu *et al.* reported an ability of the technique to generate patches as a real human would do in up to 44% of cases.

2.2 Prompt-tuning, and its applications to software engineering tasks

Prompt-tuning is a technique aimed at transforming the training objective of downstream tasks to resemble the task performed during the pre-training stage, which utilizes the MLM (Masking Language Model) objective [13, 38]. This is done by modifying the model input. That is, if the pre-training is carried out on natural language, the idea is to incorporate a natural language prompt (*e.g.,* an English-written sentence), resulting in an input format consistent with the pre-training stage. We can distinguish between two types of prompt-tuning, depending on how the prompt is injected.

2.2.1 Hard Prompt: The hard prompt [24, 25, 51] instruments the model input through the addition of fixed natural language elements, known as prompts. The goal is to leverage task-specific knowledge acquired during the pre-training. This type of prompt is considered "hard" because it consists of discrete tokens that have a clear meaning and can be understood by a human. To generate hard templates, we re-arranged the model's input tokens according to the designed prompt (Section 3.3)

2.2.2 Soft Prompt: soft prompt techniques [25, 33, 52] do not use natural language words as a template but, rather, continuous vectors learned when specializing the model (*i.e.,* fine-tuning) on the downstream task(s). Soft prompt techniques can be distinguished into *vanilla soft prompt* and *prefix soft prompt*. The former replaces the hard prompt tokens with virtual tokens. The latter arranges several virtual tokens at the beginning of the input. For both prompting techniques, the embeddings of these virtual tokens are learned during the fine-tuning phase of the model. For generating vanilla soft prompts, we replace the natural language tokens in the hard prompt templates with virtual tokens.

2.2.3 Applications of prompting to software engineering tasks: Wang *et al.* [53] postulated the possibility of leveraging prompt-tuning for software engineering tasks. Specifically, they start from pre-trained CodeBERT [17] and CodeT5 [55] models, and experiment with three software engineering tasks, namely defect prediction, code summarization, and code translation. Their results indicate that prompt-tuning: (i) always outperforms fine-tuning, (ii) it is particularly useful when fine-tuning material is limited, and (iii) its performances depend on the prompt length.

Nashid *et al.* [44] proposed CEDAR, a framework to automatically build effective prompts for two code-related tasks, *i.e.,* test assertion generation and program repair. CEDAR selects prompts by applying an automated retrieval-based demonstration selection strategy. CEDAR takes as input a set of code demonstrations and produces as output a text-based prompt. Nashid *et al.* [44] showed that CEDAR outperforms the best-performing state-of-the-art models for code-related tasks, like assertion generation and program repair. Furthermore, Nashid *et al.* [44] showed that prompt-tuning can achieve high accuracy in specific downstream tasks. Leveraging the experience of Wang *et al.* [53] and Nashid *et al.* [44], we empirically evaluate the impact of different training strategies aiming at generating vulnerability patches, by also experimenting with ten different types of prompt fine-tuning.

3 STUDY DEFINITION, DESIGN, AND PLANNING

The *goal* of this study is to evaluate how the use of various training procedures affects the effectiveness of DL-based models in patching code vulnerabilities. We investigate combinations of different pre-trainings and fine-tunings, as well as the use of different types of hard and soft prompt-tuning. The *context* consists of a Code-pretrained T5 model [55], a bug-fixing dataset from Chen *et al.* [10], and a vulnerability patches dataset made available with the VulRepair [19] paper, after the removal of duplicates, as explained in Section 3.2.3.

3.1 Research Questions

We address the following research question (RQ):

RQ$_1$: *How do different training procedures impact the performance of DL-based vulnerability patching systems?*

To address this general research question, we formulate three sub-questions, each one aimed at studying in isolation the impact on performance of a specific training technique:

RQ$_{1.1}$: *To what extent does the usage of a self-supervised pre-training objective benefit the generation of vulnerability patches?* In RQ1.1 we assess the impact of the classic *masked language model* self-supervised objective on the performance of a DL-based technique fine-tuned for vulnerability patching.

RQ$_{1.2}$: *To what extent does the usage of a supervised pre-training objective benefit the generation of vulnerability patches?* RQ$_{1.2}$ evaluates the effectiveness of further pre-training an already (self-supervised) pre-trained model using a supervised objective resembling the downstream task.
In our case, we use as a supervised pre-training objective the task of fixing bugs as a more general representation of vulnerability patching.

RQ$_{1.3}$: *What is the role played by prompt-tuning when patching vulnerable code?* RQ$_{1.3}$ investigates the impact on the performance of a prompt fine-tuning (as compared to a standard fine-tuning) by comparing the use of the hard prompt (for which we experiment with different natural language templates) and soft prompt.

3.2 Context: Datasets

We describe the pre-training and fine-tuning datasets we leverage in our study. Section 3.3 will detail how they have been used to train different models aimed at assessing the impact of the different training strategies. We anticipate that the fine-tuning dataset used to train the models for the task of vulnerability patching includes C/C++ functions, thus explaining the focus on these languages for the other datasets as well.

3.2.1 Self-supervised pre-training. When experimenting with self-supervised pre-training using the *masked language model* objective (*i.e.,* masking 15% of tokens in a string and asking the model to predict them), we exploit CodeT5 [55], a Text-To-Text Transfer Transformer (T5) model [49] already pre-trained on code and widely used in the software engineering literature. Thus, we describe in the following the dataset that has been used for its pre-training.

Wang *et al.* [55] leveraged the CodeSearchNet dataset [30] to pre-train CodeT5. This dataset comprises both technical natural language (*i.e.,* code comments) and code. Additionally, Wang *et al.* collected supplementary data from C/C# repositories on GitHub. This resulted in a total of 8,347,634 pre-training instances, where an instance is a code function: 3,158,313 of these functions are coupled with their documentation, while 5,189,321 feature only code.

3.2.2 Supervised pre-training on bug-fixing. Chen *et al.* [10] proposed the use of a supervised pre-training based on bug-fixing before fine-tuning the model for the task of vulnerability patching. We thus use the bug-fixing corpora they provide as a supervised pre-training dataset. To build this dataset, the authors mined 729M

GitHub commits identifying 21M of them as bug fixes. Out of these, 910k C/C++ bug-fixing commits were selected. Then, function-level changes performed to fix the bug were collected as $\langle F_b, F_f \rangle$ pairs, where F_b and F_f represent the buggy and the fixed version, respectively, of a function F. After duplicate removal, the authors were left with 544,858 valid instances which can be used to train a model for the bug-fixing task (*i.e.,* input F_b, expected output F_f). Out of these pairs, 534,858 are used for training, and the remaining 10,000 pairs for validation (*i.e.,* to identify the best-performing pre-trained model for the task of bug-fixing).

3.2.3 Fine-tuning for vulnerability patching. We reuse the dataset provided in the replication package of VulRepair (Fu *et al.* [20]) under the path `data/fine_tune_data`. This dataset contains 8,482 vulnerability fixes (a pair of vulnerable C/C++ functions and their patched version) obtained by merging two datasets: CVE-Fixes [7] and Big-Vul [16] and then split into 70% for training (5,937), 20% (1,706) for testing and 10% (839) for validation.

Fu *et al.* [19] processed these instances to help the model learning. Such a procedure starts with marking the vulnerable code snippet (*i.e.,* the input sequence) using two special tags: $\langle StartLoc \rangle$ and $\langle EndLoc \rangle$, where $\langle StartLoc \rangle$ tags the beginning of the vulnerable code lines, while $\langle EndLoc \rangle$ identifies their end. Two additional special tokens $\langle ModStart \rangle$ and $\langle ModEnd \rangle$ indicate the beginning and end, respectively, of the patching code (*i.e.,* the output sequence). Using these tags the model is trained to pay more attention to the code elements involved in the vulnerability patching.

Upon examining the dataset used by Fu *et al.* [19], we found duplicated instances among the training, validation, and test sets. The presence of such overlapping instances would pose questions about the real ability of the model to patch vulnerabilities. This, in turn, could lead to artificially boosted performance metrics. Thus, to address this issue we conducted a systematic analysis to identify problematic instances.

We found and discarded 38 instances (*i.e.,* pairs of $\langle vulnerable_{code}, \ patch \rangle$) having an empty string as *patch* (24 in the training set, 3 in the validation, and 11 in the test). After this preliminary cleaning, we addressed the issue of overlapping instances across the three sets. We removed from the training set all samples having an identical counterpart in the other (validation or test) sets. In particular, we removed from the training set 674 instances shared with the test set, and 322 shared with the validation set. We decided to remove such instances from the training rather than from the test/validation set because we wanted to evaluate the impact on VulRepair's performance when being fine-tuned on the cleaned dataset by comparing our findings with those reported in the original paper [19]. To do this, it was important to exploit exactly the same test set which, thus, should stay unchanged.

The number of instances we removed during this deduplication step is higher (1,008) than the sum of overlapping instances between the training and both the test and validation sets (674 + 322 = 996). This is because the training set itself contained some duplicates and, thus, some instances removed as shared with the test/validation set have been removed more than once since appearing multiple times in the training set.

Table 1: Number of instances included in the dataset used by Fu *et al.* [19], before and after performing the cleaning

Dataset	Train	Test	Eval	Overall
VulRepair [19]	5,937	1,706	839	8,482
$VulRepair_{WET}$	5,913	1,695	836	8,444
$VulRepair_{FC}$	4,905	1,695	836	7,436

Table 1 reports the number of instances included in the fine-tuning dataset for each split (*i.e.,* train, validation, and test) before and after performing the above-described cleaning.

The first row reports the number of instances included in the original dataset (*i.e.,* before cleaning), while the second and third rows show $VulRepair_{WET}$ (Without Empty Targets) and $VulRepair_{FC}$ (Fully-Cleaned), respectively, with $VulRepair_{FC}$ being the one used in our study.

3.3 Context: Experimented Training Strategies

We detail the training strategies we experiment with as follows:

3.3.1 No Pre-training + Fine-tuning. We fine-tune on the $VulRepair_{FC}$ dataset a T5_{base} model [49] (*i.e.,* the same used for CodeT5 [55]) without any pre-training. Such a model will serve as a baseline to assess the impact of pre-training on the model's performance.

3.3.2 Self-supervised Pre-training + Fine-tuning. We replicate the VulRepair approach by Fu *et al.* [19] by, however, fine-tuning the model on $VulRepair_{FC}$ (*i.e.,* the cleaned version of their dataset). VulRepair exploits the pre-trained CodeT5 which, as previously said, used a self-supervised *masked language model* pre-training objective. Such a model will allow us to answer RQ$_{1.1}$.

3.3.3 Self-supervised & Supervised Pre-training + Fine-tuning. Our goal here is to train a model taking advantage of both self-supervised (*masked language model*) and supervised (*bug-fixing*) pre-training before being fine-tuned for vulnerability patching. Such an approach is inspired by the works of Chi *et al.* [11] and Chen *et al.* [10]. We start again from the (self-supervised) pre-trained CodeT5. Then, we further train it for 5 epochs using the bug-fixing dataset described in Section 3.2.2. Finally, we specialize the model to the downstream task using the $VulRepair_{FC}$ dataset.

3.3.4 Prompt Fine-tuning. As previously mentioned with prompt fine-tuning each fine-tuning instance undergoes a transformation shaping it so that it resembles the data encountered during the pre-training procedure. We experiment prompt fine-tuning on top of the pre-trained models described in Sections 3.3.2 and 3.3.3. This means that, instead of performing a conventional fine-tuning procedure, we replaced it with hard or soft prompt fine-tuning.

For both soft and hard prompting, we experimented with five different templates that can be found in our online appendix [41].

The prompt templates embed the $vulnerable_{code}$ and the *patch* in a natural language sentence which could either be a very simple sentence describing the need to patch the code, or a more verbose one, also specifying information extracted from CWE [40] about the vulnerability affecting $vulnerable_{code}$. An example is "*Exposure of Sensitive Information to an Unauthorized Actor*" related to CWE id number 200, which draws out a weakness of code to expose sensitive information to a not authorized actor gaining access to that.

To collect textual information of CWE, *i.e.,* name and description, we downloaded from the Mitre website [2] the CSV file containing the CWE List with all related information. Then, given the CWE id contained in the dataset of Fu *et al.* [19] we used it to search into the CSV file for the corresponding CWE name and description. Note that 12 CWE ids were not contained in the CSV file since these ids referred to CWE categories and not to a specific weakness, *e.g.,* CWE-399. Thus, for these 12 CWE ids, we searched directly on the Mitre website [2] to retrieve their name and description.

We use the OpenPrompt library [14] to carry out soft-prompting (*i.e.,* to learn the needed continuous vectors — see Section 2.2) as done by Wang *et al.* [53].

Overall, we prompt fine-tuned ten versions (5 with hard and 5 with soft prompting) of the self-supervised pre-trained model described in Section 3.3.2 and ten of the self-supervised & supervised pre-trained model from Section 3.3.3.

3.4 Data Collection and Analysis

For all trained models, the fine-tuning (or prompt fine-tuning) performed on the $VulRepair_{FC}$ dataset has been run for 75 epochs, in line with what was done in previous work [19]. After each epoch, we assess the loss of the model on the evaluation set, selecting as the best checkpoint the one having the lowest loss. This is the checkpoint that has then been evaluated by running it on the test set. In total, we trained and tested 23 models:

- 1: *No pre-training + fine-tuning*, in the results referred as M0;
- 1: *Self-supervised pre-training + fine-tuning*, referred as M1;
- 1: *Self-supervised & supervised pre-training + fine-tuning*, referred as M2;
- 10: *Self-supervised pre-training + prompt fine-tuning* × 10 prompt templates (5 for soft, referred as $M3_{S1-5}$, and 5 for hard prompting, referred as $M3_{H1-5}$, where the last digit 1-5 indicates the used prompt among those described in Section 3.3.4). For example, $M3_{S1}$ uses soft prompting and the first template;
- 10: *Self-supervised & supervised pre-training + prompt fine-tuning* × 10 prompt templates (5 for soft, referred as $M4_{S1-5}$, and 5 for hard prompting, referred as $M4_{H1-5}$).

We run each trained model on the 1,695 functions in the test set, asking it to generate patches. We use the beam search decoding schema [18] that allows to produce multiple vulnerability repair candidates for an input sequence. We assess the performance of each model using two metrics: (i) the percentage of Exact Match (EM) predictions for different beam sizes K (EM@K), and (ii) the CrystalBLEU score [15].

EM@K measures the percentage of instances in the test set for which the sequence predicted by the model matches the expected oracle sequence.

Since we use beam-search, we report the results for different values of K (*i.e.,* 1, 2, 3, 4, 5, 10), as done in [19]. We avoid reporting results with K=50, because, as pointed out by Fu *et al.* [19], the effort security analysts have to put into manually inspecting such a large number of patches may hinder the adoption of the approach in practice.

CrystalBLEU score [15] measures how similar the candidate (predicted code) and reference code (oracle) are, similar to how the BLEU score [46] measures similarity between texts. However, CrystalBLEU is specifically designed for code evaluation, while retaining desirable properties of BLEU, specifically being language-agnostic and minimizing the effect of trivially shared n-grams, which would produce inflated results.

Also, we perform statistical tests to determine whether one of the experimented techniques is more effective in patching the vulnerable code. To this end, we use McNemar's test [42] (which is a proportion test for dependent samples) and Odds Ratios (ORs) on the EMs that the techniques can generate. We also statistically compare the distribution of the CrystalBLEU scores (computed at the sentence level) for the predictions generated by each technique by using the Wilcoxon signed-rank test [56]. The Cliff's Delta (d) is used as effect size [23] and it is considered: negligible for $|d|$ 0.10, small for $0.10 \le |d| < 0.33$, medium for $0.33 \le |d| < 0.474$, and large for $|d| \ge 0.474$. For all tests, we assume a significance level of 95% and we account for multiple tests by adjusting p-values using Holm's correction procedure [28].

To make it easier for the reader to follow the manuscript, we have assigned a unique ID to each model/configuration we tested, which can be found in Table 2.

4 RESULTS

Table 2 reports the results achieved using the different training strategies subject of our study. The first column, "Model-ID" reports a unique identifier we assigned to each of the 23 trained models, using the notation $Mx_{(S|H)1-5}$ described in Section 3.4. The "Pre-training" and "Fine-tuning" columns indicate the combination of training procedures adopted for each model. For example, the first line (*i.e.,* M0), represents the non-pre-trained model which has been directly fine-tuned for the task of vulnerability patching using a standard fine-tuning procedure. Finally, EM and CB report the performance achieved by a specific configuration in terms of (i) the percentage of predictions being Exact Matches (EM), and (ii) the average CrystalBLEU score [15] across all predictions in the test set (CB). To enhance the readability of such a table and to ease the results' discussion, we only include in the manuscripts the results achieved with K (*i.e.,* the beam size) of 1, 3, 5, and 10. The comprehensive set of results with all beam sizes is available in our replication package [41].

Table 3 reports the results of the statistical tests (Fisher's exact test and Wilcoxon signed-rank test), with adjusted p-values, OR, and Cliff's d effect size. An $OR > 1$, or a positive Cliff's d indicates that the right-side treatment outperforms the left-side one. For the sake of readability, we ordered the treatments to show ORs that are generally ≥ 1.

In the following, we report and discuss results by RQ.

The first and second rows of Table 2 compare a non-pre-trained model (M0) with the same model pre-trained using a *masked language model* self-supervised objective (M1, being a replica of VulRepair [19], yet using our cleaned dataset without duplicates).

Table 2: Exact Match (*i.e.,* the recommended code is equal to the oracle) and CrystalBleu scores achieved by the different techniques when patching vulnerable C functions. In dark-grey boxes we report the highest value for both metrics when producing K=1, K=3, K=5, and K=10 candidate patches. In boldface the best prompt fine-tuning template within each pre-training strategy.

Model-ID	Training Procedure		Top-1		Top-3		Top-5		Top-10	
	Pre-training	Fine-tuning	EM	CB	EM	CB	EM	CB	EM	CB
M0	✗	Supervised	2.35%	31.62%	3.54%	32.66%	4.18%	32.34%	4.79%	31.70%
M1 [19]	Self-supervised	Supervised	3.34%	40.18%	5.72%	42.10%	6.54%	41.14%	6.90%	38.58%
M2	Self-supervised + Supervised	Supervised	12.28%	47.98%	17.58%	50.21%	18.64%	49.61%	18.82%	47.26%
$M3_{S1}$	Self-supervised	Soft prompt-tuning	5.48%	50.52%	7.02%	51.53%	7.02%	51.22%	7.13%	50.80%
$M3_{S2}$	Self-supervised	Soft prompt-tuning	6.43%	50.96%	**7.43%**	52.53%	**7.55%**	51.92%	**7.70%**	51.37%
$M3_{S3}$	Self-supervised	Soft prompt-tuning	6.01%	50.00%	7.13%	51.32%	7.31%	50.71%	7.37%	49.64%
$M3_{S4}$	Self-supervised	Soft prompt-tuning	6.66%	**52.51%**	7.31%	**53.35%**	7.37%	**52.48%**	7.55%	**51.95%**
$M3_{S5}$	Self-supervised	Soft prompt-tuning	**6.78%**	51.35%	7.37%	52.11%	7.43%	52.04%	7.37%	51.50%
$M3_{H1}$	Self-supervised	Hard prompt-tuning	3.83%	40.24%	5.60%	41.59%	6.37%	41.13%	6.78%	38.36%
$M3_{H2}$	Self-supervised	Hard prompt-tuning	3.60%	**40.96%**	5.48%	41.78%	6.49%	41.20%	6.72%	38.54%
$M3_{H3}$	Self-supervised	Hard prompt-tuning	3.71%	40.75%	5.36%	41.71%	6.25%	40.81%	6.54%	38.54%
$M3_{H4}$	Self-supervised	Hard prompt-tuning	**4.01%**	40.69%	**6.13%**	41.95%	**7.02%**	40.92%	**7.55%**	39.04%
$M3_{H5}$	Self-supervised	Hard prompt-tuning	3.60%	39.77%	5.78%	**42.00%**	6.32%	**41.30%**	**7.55%**	**39.80%**
$M4_{S1}$	Self-supervised + Supervised	Soft prompt-tuning	8.96%	55.28%	10.67%	56.13%	10.97%	55.87%	10.91%	55.12%
$M4_{S2}$	Self-supervised + Supervised	Soft prompt-tuning	**9.43%**	54.45%	10.85%	56.11%	11.15%	55.83%	11.26%	55.15%
$M4_{S3}$	Self-supervised + Supervised	Soft prompt-tuning	8.67%	53.86%	**11.10%**	55.27%	11.15%	54.77%	11.26%	53.55%
$M4_{S4}$	Self-supervised + Supervised	Soft prompt-tuning	8.90%	54.13%	11.03%	55.56%	**11.56%**	55.89%	**11.62%**	54.34%
$M4_{S5}$	Self-supervised + Supervised	Soft prompt-tuning	8.84%	53.77%	10.73%	55.32%	11.10%	55.04%	11.26%	53.75%
$M4_{H1}$	Self-supervised + Supervised	Hard prompt-tuning	13.21%	**49.38%**	18.29%	**51.82%**	19.46%	**50.86%**	20.11%	**48.67%**
$M4_{H2}$	Self-supervised + Supervised	Hard prompt-tuning	12.33%	48.26%	17.34%	50.31%	18.70%	50.23%	19.29%	47.77%
$M4_{H3}$	Self-supervised + Supervised	Hard prompt-tuning	12.33%	50.61%	17.70%	50.65%	18.93%	50.40%	19.58%	47.64%
$M4_{H4}$	Self-supervised + Supervised	Hard prompt-tuning	11.97%	48.04%	17.52%	51.13%	18.87%	50.23%	19.58%	48.16%
$M4_{H5}$	Self-supervised + Supervised	Hard prompt-tuning	12.03%	48.15%	17.46%	50.81%	18.05%	49.48%	18.76%	47.17%

Table 3: Comparison among different training strategies for top-1 predictions: McNemar's and Wilcoxon's test results.

Comparison	McNemar's Test		Wilcoxon's Test	
	p-value	OR	*p*-value	d
M0 vs. M1 [19]	0.03	1.81	<0.05	0.235 (S)
M1 [19] vs. M2	<0.05	26.33	<0.05	0.140 (N)
M1 [19] vs. $M3_{S1}$	<0.05	2.51	<0.05	0.199 (S)
M1 [19] vs. $M3_{S2}$	<0.05	2.82	<0.05	0.198 (S)
M1 [19] vs. $M3_{S3}$	<0.05	2.57	<0.05	0.186 (S)
M1 [19] vs. $M3_{S4}$	<0.05	2.96	<0.05	0.232 (S)
M1 [19] vs. $M3_{S5}$	<0.05	3.03	<0.05	0.203 (S)
M1 [19] vs. $M3_{H1}$	1.0	1.63	1.0	-0.002 (N)
M1 [19] vs. $M3_{H2}$	1.0	1.42	0.3	0.018 (N)
M1 [19] vs. $M3_{H3}$	1.0	1.55	1.0	0.012 (N)
M1 [19] vs. $M3_{H4}$	0.6	1.83	1.0	0.009 (N)
M1 [19] vs. $M3_{H5}$	1.0	1.18	0.5	-0.013 (N)
$M4_{S1}$ vs. M2	<0.05	1.98	<0.05	-0.129 (N)
$M4_{S2}$ vs. M2	<0.05	1.81	<0.05	-0.114 (N)
$M4_{S3}$ vs. M2	<0.05	2.36	<0.05	-0.105 (N)
$M4_{S4}$ vs. M2	<0.05	2.25	<0.05	-0.109 (N)
$M4_{S5}$ vs. M2	<0.05	2.09	<0.05	-0.101 (N)
M2 vs. $M4_{H1}$	0.32	1.77	<0.05	0.024 (N)
$M4_{H2}$ vs. M2	1.0	1.05	0.60	-0.005 (N)
$M4_{H3}$ vs. M2	1.0	1.05	0.60	-0.005 (N)
$M4_{H4}$ vs. M2	1.0	1.46	0.60	-0.0001 (N)
$M4_{H5}$ vs. M2	1.0	1.37	0.60	-0.003 (N)

The achieved results clearly show the boost in performance that self-supervised pre-training provides for the task of vulnerability patching. To this extent, M0 patches vulnerabilities achieving a success rate ranging from 2.35% to 4.79%. These percentages correspond to the most challenging (*i.e.,* K=10) and favorable scenario for the model (*i.e.,* K=10). On the other hand, M1 demonstrates a superior ability to patch vulnerabilities, with a success rate of 3.34% when recommending a single candidate patch (*i.e.,* K = 1), which increases up to 6.90% when the model suggests K=10 candidate repairs.

Thanks to the self-supervised pre-training, M1 exhibits major improvements in terms of EMs for all values of K, with margins ranging from 3.34% (K=1) to 6.90% (K = 10). Unsurprisingly, also the CrystalBLEU exhibits an increase. According to Wilcoxon signed-rank test, the difference in terms of CrystalBLEU, is statistically significant (*p*-value), with a *Large* Cliff's Delta (d). We could not compute McNemar's test for EMs for most values of K given the 0 EM predictions generated by the non-pre-trained model.

When comparing the performances achieved on the cleaned VulRepair dataset, and those of the original paper by Fu *et al.* [19], we observe a substantial drop in the ability of the model [19] to recommend candidates patches for vulnerable code. This is because the dataset of Fu *et al.* [19] has a significant overlap of instances between the training and the test set (~40% of instances in the test

Table 4: Comparison in terms of percentage of Exact Match predictions reported by Fu *et al.* [19] and by our replication of their approach. The observed difference is due to our cleaning of their dataset which removed "token-by-token" overlapping instances between the training and the test set.

Source	Exact Matches (%)			
	K=1	K=3	K=5	K=10
Fu *et al.* [19]	30%	38%	41%	42%
Our replication	3.34%	5.72%	6.54%	6.90%
	-26.66%	**-32.28%**	**-34.46%**	**-35.10%**

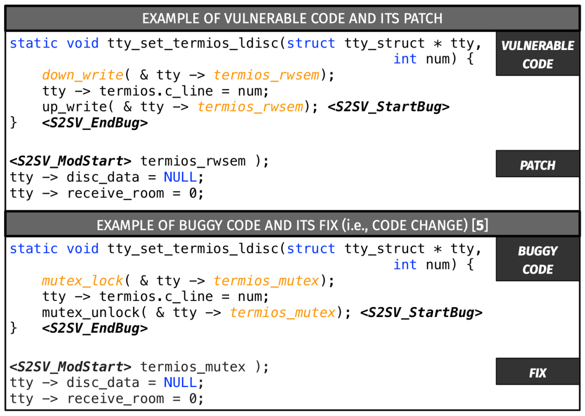
set were present in the training set). Table 4 reports the performance from the original paper (top row) as compared to our replication. As expected, the duplicated instances substantially inflated the EM predictions reported in [19], with up to a +35.10% for K=10. Nevertheless, as shown by our results on the cleaned dataset, we confirm that a large-scale self-supervised pre-training helps in boosting the performance of vulnerability patching.

Answer to RQ$_{1.1}$. Compared to techniques that do not rely on pre-training (e.g., M0), pre-trained models help to automatically fix a larger number of vulnerable functions, up to ~2% more. The non-pre-trained model struggles more, having a hard time learning the vulnerability patching task, due to the small fine-tuning dataset available.

4.1 RQ$_{1.2}$: How does the usage of supervised training affect the performances of large pre-trained models of code in automatically patching vulnerabilities?

Seeding bug-fixing knowledge into the model (M2) as proposed by Chi *et al.* [11] and Chen *et al.* [10] provides a substantial boost in performance as compared to the use of self-supervised training alone (compare M2 to M1 in Table 2). The improvement is consistent across all beam sizes we experimented with. Specifically, when only relying on the top prediction (*i.e.,* K=1), the EM predictions increase from 3.34% to 12.28% (+~9%). This gap becomes even higher when the model is asked to generate more candidate patches, with improvements in EM predictions of up to +~12%.

The superior performance ensured by the injection of bug-fixing knowledge is also confirmed by the McNemar test (see Table 3), which reports a statistically significant difference in EM predictions between M1 and M2, with an OR=26.33 in favor of M2. The Wilcoxon signed-rank test also indicates a statistically significant difference (p-value < 0.05) when comparing the distributions of CrystalBLEU scores between M1 and M2 with, however, a *Negligible* effect size. The usefulness of learning from bug fixes confirms what was found by previous literature [10, 11]. This is mainly due to the commonalities between a bug fix and a vulnerability fix. This helps the model to gain knowledge related to the downstream task without, however, using the scarce fine-tuning instances available for such a task. In the end, a vulnerability fix is nothing but a specific bug fix, where the bug could be possibly exploited for a security attack. Fig. 1 shows a concrete example of such a scenario. The top part of the figure shows the changes needed to fix a vulnerability in our test set, while the bottom part shows a bug-fixing change from the dataset by Chen *et al.* [10] (*i.e.,* the one used for the bug-fixing training). As it can be seen, while acting on different code components, both the involved functions require the same code changes. In other words, in both functions, the fields `disc_data` and `receive_room` need to be initialized to `NULL` and 0, respectively. In such a case, the learned fixing pattern is basically the same. It is just that in one case the problem exposed the system to a possible vulnerability attack, in the other it did not, although it is also true that many bug fixes are silent vulnerability fixes [58].


EXAMPLE OF VULNERABLE CODE AND ITS PATCH

VULNERABLE CODE
```
static void tty_set_termios_ldisc(struct tty_struct * tty,
                                              int num) {
    down_write( & tty -> termios_rwsem);
    tty -> termios.c_line = num;
    up_write( & tty -> termios_rwsem); <S2SV_StartBug>
}   <S2SV_EndBug>
```

PATCH
```
<S2SV_ModStart> termios_rwsem );
tty -> disc_data = NULL;
tty -> receive_room = 0;
```

EXAMPLE OF BUGGY CODE AND ITS FIX (i.e., CODE CHANGE) [5]

BUGGY CODE
```
static void tty_set_termios_ldisc(struct tty_struct * tty,
                                              int num) {
    mutex_lock( & tty -> termios_mutex);
    tty -> termios.c_line = num;
    mutex_unlock( & tty -> termios_mutex); <S2SV_StartBug>
}   <S2SV_EndBug>
```

FIX
```
<S2SV_ModStart> termios_mutex );
tty -> disc_data = NULL;
tty -> receive_room = 0;
```

Figure 1: Example of vulnerable code vs. buggy code for which the model is required to perform similar code changes.

Answer to RQ$_{1.2}$. Incorporating task-specific knowledge via bug-fixing training into pre-trained models of code significantly increases the ability to generate patches for vulnerable code components. The increase in Exact Match predictions ranges between ~9% and ~12%, depending on the number of candidate solutions (*i.e.,* patches) generated by the models.

4.2 RQ$_{1.3}$: What is the role played by the prompt-tuning when producing patches for vulnerable code?

We first focus on the effect of prompt-tuning when fine-tuning a model pre-trained using only a self-supervised procedure ($M3_{S_n}$ and $M3_{H_n}$ in Table 2). Afterward, we report and discuss the impact of prompt-tuning for a model pre-trained using both self-supervised and supervised (*i.e.,* bug-fixing) pre-training ($M4_{S_n}$ and $M4_{H_n}$ in Table 2).

Prompt-tuning a self-supervised pre-trained model. Both soft ($M3_{S_1}$ to $M3_{S_5}$) and hard ($M3_{H_1}$ to $M3_{H_5}$) prompt-tuning help in improving the performance of a model that was pre-trained using a self-supervised objective only. However, the magnitude of the improvement is substantially different. When the models are asked to generate a single prediction (K=1), soft prompt-tuning provides an increase in EMs between 2.14% and 3.44%. While these improvements may look minor, note that the baseline (*i.e.,* M1) achieves 3.34% of EM predictions. Thus, the best-performing template in the context of prompt-tuning ($M3_{S5}$) more than doubles the predictive capabilities of the model. Remember that this template is the one providing the model with information about the type of vulnerability to fix: `This text <CWE_NAME>` {cwe_{name}} `</CWE_NAME> describes the vulnerable code` {$vulnerable_{code}$} `fixed by:` {$patch$}

The statistical tests (Table 3) support such findings, with the models subject to soft prompt-tuning achieving, independently from the used prompt template, a statistically significant higher percentage of EM predictions than the self-supervised pre-trained models fine-tuned with a standard approach. The adjusted p-values are always <0.05, with ORs ranging from 2.51 (M1 *vs* $M3_{S1}$) and 3.03 (M1 *vs* $M3_{S5}$).

Improvements are also observed in terms of CrystalBLEU score, which, in comparison with M1, has an increase of up to +12% for $K = 1$. Also in this case, the statistical tests confirm a significant difference in favor of the models fine-tuned with soft prompting, independently from the used template (in all cases, p-values <0.05 with a small effect size). Looking at the different templates used for soft-prompting, we found them to have a non-trivial impact on performance. When looking at the top-1 prediction ($K = 1$), the best-performing template is, as said, $M3_{S5}$, while the worst one is $M3_{S1}$, with a 1.30% gap in terms of EM predictions, which means a relative ~24% improvement (1.30/5.48). This points to the importance of carefully selecting the template to use when adopting prompt fine-tuning: The setting of the template must be considered as important as that of the other model's hyperparameters, that usually undergo a tuning procedure. It is also worth noticing that, while $M3_{S5}$ is the best in class for $K = 1$, this is not the case for other K values: for $K = 3$ and $K = 5$ it is the second-best, while for $K = 10$ is the third-best, despite the minor gap from the best-performing template. Thus, even the usage scenario envisioned for the trained model and, consequently, the number of candidate solutions that it will be required to generate for each input may impact the choice of the template to adopt. Looking at the hard prompting results ($M3_{H_1}$ to $M3_{H_5}$ in Table 2), the improvement it achieves compared to the baseline is smaller than what we observed for soft-prompting (*e.g.,* +0.67% when $K = 1$). The lower (or, in a few cases, lack of) improvement holds for all experimented prompts, and for all the considered beam sizes K. Also, it is confirmed by the statistical tests, reporting non-significant p-values (see the comparison between M1 and $M3_{H_n}$ in Table 3).

Prompt-tuning a self-supervised and supervised pre-trained model. The results achieved when prompt-tuning a model that has already acquired task-specific knowledge (thanks to the bug-fixing training) are quite different from what was achieved for the self-supervised pre-trained model. Looking at the $M4_{S_n}$ and $M4_{H_n}$ in Table 2, we can notice how, in this case, the models that have been subject to hard prompting are the ones working better and, overall, achieving the best performance in terms of EM predictions (see the values reported in bold faces). This holds for all K values. The gap from the baseline (M2, which employs self-supervised + supervised training, without prompt-tuning) is relatively small (~+1% in EMs) and statistically significant (see Table 3). Note that this is in line with what was previously observed in the literature about the benefits of prompt-tuning when dealing with other software engineering tasks [53]. The soft prompt-tuning is, instead, able to improve the results in terms of CrystalBLEU, yet it has a price to pay in terms of EM predictions, which are worse than the M2 baseline, as also confirmed by the statistical analyses.

There may be two possible reasons why no further improvements are observed when using soft prompting on top of models that were subject to a combination of self-supervised and supervised training.

The first reason is related to the additional knowledge the model acquires from bug fixes. As previously shown (see Fig. 1), fixing a generic bug may require code changes that are very similar (or even equal) to those needed when patching vulnerabilities. Therefore, the similarity between these tasks may already resemble code transformations for which the model has become proficient, and the effort in applying these transformations in a different context (*i.e.,* vulnerability fixing) is trivial.

The second reason is related to prompt-tuning strategies in general, which heavily rely on natural language knowledge acquired during pre-training. This critical information may be partially overwritten when further specializing pre-trained models by incorporating context-specific knowledge. In other words, while the self-supervised pre-trained model primarily captures the statistical distribution of words in natural language, the model additionally trained for bug-fixing may have lost some of its ability to represent and interpret natural language, leading to poor performances.

That being said, prompt-tuning offers a notable advantage of potentially achieving high performance in fixing vulnerabilities without requiring the creation of a bug-fixing dataset necessary for supervised training. Moreover, it is essential to consider the effort required for preprocessing these instances after collection, as well as the effort researchers must put into such a large-scale operation over a prolonged period.

Answer to RQ$_{1.3}$ Hard and (above all) soft prompt fine-tuning can be a relatively cheap way to boost performance for models only subject to self-supervised pre-training. The improvement introduced by prompting is less evident or negligible for models already having knowledge from similar tasks, *e.g.,* bug fixing.

4.3 Implications of our findings

Our findings have significant implications for both researchers and practitioners. Researchers should focus on overcoming the data scarcity challenges faced by generative models in vulnerability patching, while at the same time making sure of the quality of the collected elements used to train the proposed techniques, to avoid leakage of data, as shown in this research. The promising initial results with prompt-tuning open up new opportunities for in-depth exploration of advanced methods for crafting effective prompts within the designated framework. Furthermore, we encourage researchers to develop new metrics that can effectively and accurately evaluate model performance in practical vulnerability patching scenarios. On the other hand, practitioners need to recognize the limitations of these methods and avoid depending blindly on them. They should actively engage with researchers, providing targeted and practical feedback to refine and develop more sophisticated techniques with the goal in mind of further advancing patching vulnerabilities methods. This includes assessing generative solutions in real-world settings and identifying cases where they fall short. Such insights would guide the development of models, particularly when these solutions are employed to generate patches for critical environments such as, transportation networks, and healthcare systems.

5 THREATS TO VALIDITY

Construct validity. We used a consolidated set of measurements to assess the quality of vulnerability patching with respect to the dataset ground truth, *i.e.,*, percentages of perfect predictions, CrystalBLEU score [15]. Clearly, such a strategy might consider a wrong prediction legitimate fixes still differing from the ground truth.

Internal validity. Among the factors internal to our study that could influence our findings, the choice of DL model hyperparameters has a paramount role. To make the comparison fair, and since we leveraged a previous vulnerability patch dataset [19] and, therefore, we used the same hyperparameters used by VulRepair. Nevertheless, it is possible that prompt-tuning approaches or, in general, other configurations we experimented could work better with different unexplored settings. As explained in Section 3.2.3, we found duplicates in the VulRepair [19] dataset. To ensure a comparison on exactly the same test set used in the original paper, and considering that the training set is larger than the test set, we removed duplicates from the training set, leaving the test set unchanged. This, however, reduced the training material available for the model, compared to the original study. While a larger training set could produce better results, we have mitigated the threat by putting all considered treatments under the same conditions, *i.e.,* using the same training set.

Our results (those of $RQ_{1.3}$ in particular) showed how soft prompt-tuning does not help for treatments (M4) where a supervised pre-training with bug fixes was performed, likely because the model lost the "natural language" knowledge in such a phase. This, however, also suggests that prompt-tuning could be done with bug fixes (and then followed by further fine-tuning with vulnerability patches). We tried a simple prompt-tuning with a subset of our data, but it did not produce significant improvements. Therefore, we abandoned such an option and did not include it in our results.

Conclusion validity. The study conclusions are supported by the use of suitable statistical procedures, namely a proportion test (McNemar [42]) to compare perfect predictions, and Wilcoxon signed-ranked test [56] to compare distributions of CrystalBLEU scores. These tests are complemented by suitable effect size procedures, namely OR and Cliff's delta.

External validity. Although this study reported a wide set of possible combinations of training strategies (23 in total), several, further dimensions could be investigated, including vulnerabilities in other programming languages, other language models (*e.g.,* large language models), and further prompt-tuning strategies, *e.g.,* automatically-generated ones [44]. In this paper, we only experimented using the T5 pre-trained model, which is the one also used by VulRepair [19]. This allowed a fair comparison with the model they proposed (except for the removal of duplicates from the dataset). That being said, with the recent surge of large language models such as GPT-3 [8] and GPT-4 [45], we cannot guarantee that the observed differences will remain consistent when the treatments are applied using models featuring an order of magnitude more parameters (*e.g.,* GPT-4).

6 CONCLUSION

This paper empirically compares different strategies to perform vulnerability fixing using generative models, comparing a total of 23 different treatments featuring self-supervised pre-training, supervised training (using a bug fix dataset as in a previous work [10]), and, last but not least, different templates for hard and soft prompt-tuning.

Results of the study indicate that: (i) unlike what was observed in a recent paper [19], vulnerability fixing still benefits from supervised training on a similar task (*i.e.,* bug fixing) rather than just doing fine-tuning on a pre-trained model as observed by Chen *et al.* [10]; (ii) the use of prompt-tuning introduces benefits over self-supervised pre-trained models but also (limited to hard prompting and with a small magnitude) over models that also underwent supervised training and; (iii) while supervised training remains necessary to achieve the best performances, in the absence of a large dataset of bug fixes (which collection requires a non-trivial effort) or if no further computational resources can be invested in pre-training, the use of prompt fine-tuning constitutes a valid, cheap solution to boost a fine-tuning over a self-supervised pre-trained model. Future work aims at experimenting with large language models, as well as more complex or even automatically-generated prompting strategies such as those proposed by Nashid *et al.* [44].

7 DATA AVAILABILITY

Our replication package is available online [41]. Within it, we provide all code and data used in our study. This includes: (i) the datasets exploited for training/testing the experimented models; (ii) the code needed to train and test the models; and (iii) R scripts used for the statistical analysis.

ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 851720). Massimiliano Di Penta acknowledges the Horizon 2020 (EU Commission) project COSMOS (DevOps for Complex Cyber-physical Systems), Project No. 957254-COSMOS.

REFERENCES

[1] 2023. Apache Log4j Security Vulnerabilities, https://logging.apache.org/log4j/2.x/security.html. Accessed: 2023-03-27.
[2] 2023. CWE - Common Weakness Enumeration, https://cwe.mitre.org/. Accessed: 2023-01-13.
[3] 2023. The Edgescan Vulnerability Stats Report, https://www.edgescan.com/intel-hub/stats-reports/. Accessed: 2023-03-27.
[4] 2023. GitHub Wiki, https://en.wikipedia.org/wiki/GitHub. Accessed: 2023-03-27.
[5] Thanassis Avgerinos, David Brumley, John Davis, Ryan Goulden, Tyler Nighswander, Alex Rebert, and Ned Williamson. 2018. The mayhem cyber reasoning system. *IEEE Security & Privacy* 16, 2 (2018), 52–60.
[6] Iz Beltagy, Matthew E Peters, and Arman Cohan. 2020. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150* (2020).
[7] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. 30–39.
[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.
[9] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* (2021).
[10] Zimin Chen, Steve Kommrusch, and Martin Monperrus. 2023. Neural Transfer Learning for Repairing Security Vulnerabilities in C Code. *IEEE Trans. Software Eng.* 49, 1 (2023), 147–165. https://doi.org/10.1109/TSE.2022.3147265
[11] Jianlei Chi, Yu Qu, Ting Liu, Qinghua Zheng, and Heng Yin. 2023. SeqTrans: Automatic Vulnerability Fix Via Sequence to Sequence Learning. *IEEE Trans. Software Eng.* 49, 2 (2023), 564–585. https://doi.org/10.1109/TSE.2022.3156637
[12] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. 2020. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555* (2020).
[13] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[14] Ning Ding, Shengding Hu, Weilin Zhao, Yulin Chen, Zhiyuan Liu, Hai-Tao Zheng, and Maosong Sun. 2021. OpenPrompt: An Open-source Framework for Prompt-learning. *arXiv preprint arXiv:2111.01998* (2021).

[15] Aryaz Eghbali and Michael Pradel. 2022. CrystalBLEU: precisely and efficiently measuring the similarity of code. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[16] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. AC/C++ code vulnerability dataset with code changes and CVE summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 508–512.

[17] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[18] Markus Freitag and Yaser Al-Onaizan. 2017. Beam search strategies for neural machine translation. *arXiv preprint arXiv:1702.01806* (2017).

[19] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Q. Phung. 2022. VulRepair: a T5-based automated software vulnerability repair. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 935–947. https://doi.org/10.1145/3540250.3549098

[20] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Q. Phung. 2023. Replication Package VulRepair, https://github.com/awsm-research/VulRepair. Accessed: 2023-03-27.

[21] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. 2015. Safe memory-leak fixing for c programs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 459–470.

[22] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2020. Generative adversarial networks. *Commun. ACM* 63, 11 (2020), 139–144.

[23] Robert J. Grissom and John J. Kim. 2005. *Effect sizes for research: A broad practical approach* (2nd edition ed.). Lawrence Earlbaum Associates.

[24] Yuxian Gu, Xu Han, Zhiyuan Liu, and Minlie Huang. 2021. Ppt: Pre-trained prompt tuning for few-shot learning. *arXiv preprint arXiv:2109.04332* (2021).

[25] Xu Han, Weilin Zhao, Ning Ding, Zhiyuan Liu, and Maosong Sun. 2022. Ptr: Prompt tuning with rules for text classification. *AI Open* 3 (2022), 182–192.

[26] Jacob Harer, Onur Ozdemir, Tomo Lazovich, Christopher P. Reale, Rebecca L. Russell, Louis Y. Kim, and Peter Chin. 2018. Learning to Repair Software Vulnerabilities with Generative Adversarial Networks. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (Eds.). 7944–7954. https://proceedings.neurips.cc/paper/2018/hash/68abef8ee1ac9b664a90b0bbaff4f770-Abstract.html

[27] David Hin, Andrey Kan, Huaming Chen, and M Ali Babar. 2022. LineVD: statement-level vulnerability detection using graph neural networks. In *Proceedings of the 19th International Conference on Mining Software Repositories*. 596–607.

[28] Sture Holm. 1979. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics* (1979), 65–70.

[29] Zhen Huang, David Lie, Gang Tan, and Trent Jaeger. 2019. Using safety properties to generate vulnerability patches. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 539–554.

[30] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[31] Junhee Lee, Seongjoon Hong, and Hakjoo Oh. 2018. Memfix: static analysis-based repair of memory deallocation errors for c. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 95–106.

[32] Brian Lester, Rami Al-Rfou, and Noah Constant. 2021. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691* (2021).

[33] Xiang Lisa Li and Percy Liang. 2021. Prefix-tuning: Optimizing continuous prompts for generation. *arXiv preprint arXiv:2101.00190* (2021).

[34] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. 2016. Vulpecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd annual conference on computer security applications*. 201–213.

[35] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681* (2018).

[36] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, Bing Mao, and Li Xie. 2007. AutoPaG: towards automated software patch generation with source code root cause identification and repair. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security*. 329–340.

[37] Xiao Liu, Kaixuan Ji, Yicheng Fu, Zhengxiao Du, Zhilin Yang, and Jie Tang. 2021. P-tuning v2: Prompt tuning can be comparable to fine-tuning universally across scales and tasks. *arXiv preprint arXiv:2110.07602* (2021).

[38] 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. *arXiv preprint arXiv:1907.11692* (2019).

[39] Siqi Ma, Ferdian Thung, David Lo, Cong Sun, and Robert H. Deng. 2017. VuRLE: Automatic Vulnerability Detection and Repair by Learning from Examples. In *Computer Security - ESORICS 2017 - 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11-15, 2017, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10493)*, Simon N. Foley, Dieter Gollmann, and Einar Snekkenes (Eds.). Springer, 229–246. https://doi.org/10.1007/978-3-319-66399-9_13

[40] Robert A Martin and Sean Barnum. 2008. Common weakness enumeration (CWE) status update. *ACM SIGAda Ada Letters* 28, 1 (2008), 88–91.

[41] Antonio Mastropaolo, Vittoria Nardone, Gabriele Bavota, and Massimiliano Di Penta. 2024. Replication Package of the paper: "How the Training Procedure Impacts the Performance of Deep Learning-based Vulnerability Patching". https://github.com/antonio-mastropaolo/dl-training-vuln-patching

[42] Quinn McNemar. 1947. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika* 12, 2 (1947), 153–157.

[43] David J Musliner, Scott E Friedman, Michael Boldt, J Benton, Max Schuchard, Peter Keller, and Stephen McCamant. 2015. Fuzzbomb: Autonomous cyber vulnerability detection and repair. In *Fourth International Conference on Communications, Computation, Networks and Technologies (INNOV 2015)*.

[44] Noor Nashid, Mifta Sintaha, and Ali Mesbah. [n. d.]. Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning. ([n. d.]).

[45] OpenAI. 2023. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL]

[46] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318.

[47] Jeff H Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, et al. 2009. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 87–102.

[48] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[49] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research* 21, 1 (2020), 5485–5551.

[50] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. 2018. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE international conference on machine learning and applications (ICMLA)*. IEEE, 757–762.

[51] Timo Schick and Hinrich Schütze. 2020. Exploiting cloze questions for few shot text classification and natural language inference. *arXiv preprint arXiv:2001.07676* (2020).

[52] Maria Tsimpoukelli, Jacob L Menick, Serkan Cabi, SM Eslami, Oriol Vinyals, and Felix Hill. 2021. Multimodal few-shot learning with frozen language models. *Advances in Neural Information Processing Systems* 34 (2021), 200–212.

[53] Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R Lyu. 2022. No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 382–394.

[54] Tielei Wang, Chengyu Song, and Wenke Lee. 2014. Diagnosis and emergency patch generation for integer overflow exploits. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 11th International Conference, DIMVA 2014, Egham, UK, July 10-11, 2014. Proceedings 11*. Springer, 255–275.

[55] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859* (2021).

[56] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83.

[57] Fang Wu, Jigang Wang, Jiqiang Liu, and Wei Wang. 2017. Vulnerability detection with deep learning. In *2017 3rd IEEE international conference on computer and communications (ICCC)*. IEEE, 1298–1302.

[58] Jiayuan Zhou, Michael Pacheco, Zhiyuan Wan, Xin Xia, David Lo, Yuan Wang, and Ahmed E Hassan. 2021. Finding a needle in a haystack: Automated mining of silent vulnerability fixes. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 705–716.