CLAWSAT: Towards Both Robust and Accurate Code Models

Jinghan Jia[1*], Shashank Srikant[2*], Tamara Mitrovska[2], Chuang Gan[3], Shiyu Chang[4], Sijia Liu[1,3], Una-May O'Reilly[2]
Michigan State University CSAIL, MIT MIT-IBM Watson AI Lab University of California Santa Barbara
*Equal contribution
jiajingh@msu.edu shash@mit.edu liusiji5@msu.edu

***Abstract*—We integrate contrastive learning (CL) with adversarial learning to co-optimize the robustness and accuracy of code models. Different from existing works, we show that code obfuscation, a standard code transformation operation, provides novel means to generate complementary 'views' of a code that enable us to achieve both robust and accurate code models. To the best of our knowledge, this is the first systematic study to explore and exploit the robustness and accuracy benefits of (multi-view) code obfuscations in code models. Specifically, we first adopt adversarial codes as robustness-promoting views in CL at the self-supervised pre-training phase. This yields improved robustness and transferability for downstream tasks. Next, at the supervised fine-tuning stage, we show that adversarial training with a proper temporally-staggered schedule of adversarial code generation can further improve robustness and accuracy of the pre-trained code model. Built on the above two modules, we develop CLAWSAT, a novel self-supervised learning (SSL) framework for code by integrating CL with adversarial views (CLAW) with staggered adversarial training (SAT). On evaluating three downstream tasks across Python and Java, we show that CLAWSAT consistently yields the best robustness and accuracy (*e.g.* 11% in robustness and 6% in accuracy on the code summarization task in Python). We additionally demonstrate the effectiveness of adversarial learning in CLAW by analyzing the characteristics of the loss landscape and interpretability of the pre-trained models. We will make our code and datasets available publicly, and have uploaded an anonymized copy of it with our submission for an evaluation of this work.**

I. INTRODUCTION

Recent progress in large language models for *computer programs* (*i.e.* code) suggests a growing interest in self-supervised learning (**SSL**) methods to learn code models–deep learning models that process and reason about code [1]–[5]. In these models, a task-agnostic encoder is learned in a pre-training step, typically on an unlabeled corpus. The encoder is appended to another predictive model which is then fine-tuned for a specific downstream task. In particular, contrastive learning (**CL**) based self-supervision [6], [7] has shown to improve the downstream performance of code reasoning tasks when compared to state-of-the-art task-specific supervised learning (**SL**) models [5], [8], [9].

While CL offers to be a promising SSL approach, nearly all the existing works focus on improving the accuracy of code models. Yet, some very recent works [10]–[12] showed that trained supervised code models are vulnerable to **code obfuscation** transformations. These works propose **adversarial code**–changing a given code via obfuscation transformations. Such transformed programs retain the functionality of the original program but can fool a trained model at test time. Figure 2 shows an example of adversarial code achieved by code obfuscation (more details in Section III). In software engineering, code obfuscation is a commonly-used method to hide code in software projects without altering their functionality [13], [14], and is consequently a popular choice among malware composers [15]. Thus, it is important to study how obfuscation-based adversarial code could affect code model representations learned by SSL. As an example, Schuster et al. [16] successfully demonstrates adversarial code attacks on a public code completion model pre-trained on GPT-2, a large language model of code.

Improving the robustness of ML models to adversarial code however comes at a cost–its accuracy (model generalization). Works in vision [17], [18] and text [19] have shown how adversarially trained models improve robustness at the cost of model accuracy. While some works [20], [21] provide a theoretical framework for how the robustness of learned models is always at odds with its accuracy, this argument is mainly confined to the SL paradigm and vision applications, and thus remains uninvestigated in SSL for code. While there exist similarities between SSL in vision and code, the discrete and structured nature of inputs, and the additional constraints enforced on views (obfuscated codes) introduce a new set of challenges which has been unexplored by the vision community.

> For code models, we ask: *Can pre-trained models be made robust to adversarial attacks? Is it possible to retain this 'pre-trained robustness' when fine-tuning on different tasks? And importantly, is it possible to improve on both the retained generalization and robustness during fine-tuning, thus challenging the popular view of having to trade-off robustness for accuracy gains?* These questions form the focus of our work.

A. Overview of proposed appoach

We offer two methods which help us improve not only the transfer of robustness from pre-trained models to downstream tasks, but also co-improve fine-tuned accuracy and robustness.

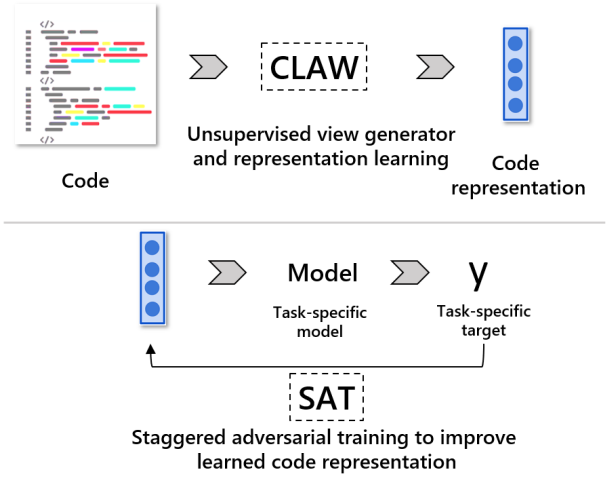

Fig. 1: **Schematic overview.** We present **CLAW** - a contrastive learning-based unsupervised method which learns *adversarial views* of the input code to in turn learn accurate and robust representations of the code. We also present **SAT**, a refinement to the adversarial training algorithm proposed by Madry et al. [22] which helps retain the task-independent robustness and accuracy learned by CLAW while also learning task-specific accuracy and robustness. We show that CLAWSAT yields better accuracy and robustness when compared to state-of-the-art self-supervised learning models for code.

The schematic overview of our proposal is shown in Figure 1. **First**, we propose a self-supervised pre-training method, contrastive learning with adversarial views (**CLAW**), which leverages adversarially-obfuscating codes as positive views of CL so as to enforce the robustness of learned code representations. We formulate and achieve CLAW through a bi-level optimization method. We show that the representations learned from these pre-trained models yield better robustness transfer to downstream tasks. **Second**, we propose staggered adversarial training (**SAT**) to preserve the robustness learned during pre-training while also learning task-specific generalization and robustness during fine-tuning. We show for the first time that the scheduler of adversarial code generation is adjustable and is a key to benefit both generalization and robustness of code models.

B. Contributions

On one hand, we propose CLAW by extending the standard CL framework for code. As a baseline, we compare CLAW to the state-of-the-art standard CL framework for code models - CONTRACODE released by [5]. We find CLAW to 'retain' more robustness when compared to **CONTRACODE**. Further, we provide a detailed analysis of understanding this improved performance from the perspective of model interpretability and characteristics of its loss landscape. *On the other hand*, we integrate CLAW with SAT to achieve the eventually fine-tuned **CLAWSAT** models. We evaluate three tasks–code summarization, code completion, and code clone detection, in two programming languages - Python and Java, and two different decoder models–LSTMs and transformers. We show that CLAWSAT outperforms CONTRACODE [5] by roughly 6% on the summarization task, 2% on the completion task, and 1% on the code clone task in accuracy, and by 9%, 3%, and 1% on robustness respectively. We study the effect of different attack strengths and attack transformations on this performance, and find it to be largely stable across different attack parameters. We will make our code and datasets available publicly, and have uploaded an anonymized copy of it with our submission for an evaluation of this work.

II. RELATED WORK

Due to a large body of literature on SSL and adversarial robustness, we focus our discussion on those relevant to code models and CL (contrastive learning).

A. SSL for code

CL-based SSL methods offer a distinct advantage by being able to signal explicit examples where the representations of two codes are expected to be similar. While the method itself is agnostic to the input representation, all the works in CL for code models work on code tokens directly. Existing works [5], [23]–[25] show that CL models for code improve the generalization accuracy of fine-tuned models for different tasks when compared to other pre-training methods like masked language models. Each of these works uses semantics-preserving, random code transformations as positive views in its CL formulation. Such transformations help the pre-trained model learn the equivalence between representations of program elements which do not affect the executed output, such as the choice of variable names, the algorithmic 'approach' used to solve a problem, *etc*.

State-of-the-art CL-based representations generally provide an improvement in the range of 1%-5% points of F1/BLEU/accuracy scores when compared to their fully supervised counterparts and other pre-training methods like transformers, which is significant in the context of the code tasks they evaluate, providing clear evidence for the utility of SSL methods. While these methods improve the generalizability of task-specific models, *none of these works have studied the robustness of these models*, especially with a growing body of works showing the susceptibility of code models to adversarial attacks [10]–[12]. By contrast, the adversarial robustness of CL models for image classification has increasingly been studied by the vision community [26]–[29]. These works have shown that CL has the potential to offer dual advantages of robustness and generalization. The fundamental differences in image and code processing, including how adversarial perturbations are defined in these two domains, motivate us to ask if and how the advantages offered by CL can be realized for code models.

B. Adversarial robustness of code models: Attacks & defenses

[30]–[33] showed that obfuscation transformations made to code can serve as adversarial attacks on code models. Following these works, recent papers [10], [11] proposed

perturbing programs by replacing local variables and inserting print statements with replaceable string arguments. They found optimal replacements using a first-order optimization method, similar to HotFlip [34]. [12] framed the problem of attacking code models as a problem in combinatorial optimization, unifying the attempts made by prior works. [10], [11], [35] and [12] also proposed strategies to train code models against adversarial attacks. While [35] employed a novel formulation to decide if an input is adversarial, the other works employed the adversarial training strategy proposed by [18].

Work most relevant to ours. Our work comes closest to CONTRACODE, the system proposed by [5]. While they established the benefit of using CL-based unsupervised representation learning for code, the work neglects the interrelationship between pre-training and fine-tuning in the SSL paradigm, and the consequences of this relationship on both the robustness and generalization of the final model.

III. PRELIMINARIES

We begin by providing a brief background on code models, code obfuscation transformations, and SSL-aided predictive modeling for code. We then motivate the problem of how to advance SSL for code models. We study this through the lenses of accuracy and robustness of the learned models.

A. Code and obfuscation transformations

Let $\mathcal{P}$ denote a *computer program* (*i.e.*, code) which consists of a series of n tokens $\{\mathcal{P}_i\}_{i=1}^{n}$ in the source code domain. For example, Figure 2 shows an example code $\mathcal{P}$. Given a vocabulary of tokens (denoted by Ω), each token can be regarded as a one-hot vector of length $|\Omega|$. Here we ignore white spaces and other delimiters when tokenizing.

Let $t(\cdot)$ denote an *obfuscation transformation*, and $t(\mathcal{P})$ an obfuscated version of $\mathcal{P}$. $t(\mathcal{P})$ is semantically the same as $\mathcal{P}$ while possibly being different syntactically. Following the notations defined by [12] and [10], we refer to locations or tokens in a code which can be transformed as *sites*. We focus on *replace* and *insert* transformations, where either existing tokens in a source code are replaced by another token, or new lines of code are inserted in the existing code. For example, in Figure 2, the replace transformation modifies the variable `sum` with `test`, while the insert transformation introduces a new line of code `print("Network")`.

Obfuscation transformations have been shown to serve as adversarial examples for code models (see Section II). During an adversarial attack, these transformations are made with the goal to get the resulting transformed code to successfully fool a model's predictions. The transformations at any given *site* in a code, such as the tokens `test`, `"Network"` in Figure 2, can be obtained in two ways—by random transformations $t_{\text{rand}}(\cdot)$: they introduce a token sampled at random from Ω, or through adversarial transformations $t_{\text{adv}}(\cdot)$: they solve a first-order optimization problem such that the transformed code maximizes the chances of the model making an incorrect prediction.

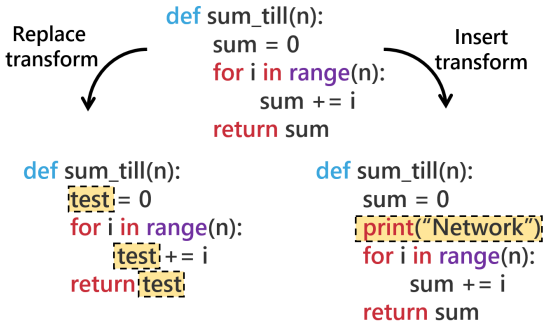

Fig. 2: Two types of semantics-preserving transformations (obfuscations) can be made to a code to attack code models–replace - where existing code is modified at a *site*—location in the code, or insert - where new lines of code are inserted at a *site*. We select *sites* at random. The specific tokens used in these transformations (`test` and `"Network"` in the the example) can either be a random transformation $t_{\text{rand}}(\cdot)$–a randomly selected token from a pre-defined vocabulary, or can be an adversarial transformation $t_{\text{adv}}(\cdot)$, where the token is obtained from solving a first-order optimization designed to fool the model [10]–[12].

Our goal then turns to improve not only *accuracy* (*i.e.* prediction accuracy of properties of code $\mathcal{P}$) but also *robustness* (in terms of prediction accuracy of properties of $t(\mathcal{P})$, obfuscated transformations of $\mathcal{P}$).

B. Problem statement

SSL typically includes two learning stages: *self-supervised pre-training* and *supervised fine-tuning*, where the former acquires deep representations of input data, and the latter uses these learned features to build a supervised predictor specific to a downstream task, *e.g.* code summarization [36] as considered in our experiments. In the pre-training phase, let $\boldsymbol{\theta}$ denote a feature-acquisition model (trained over unlabeled data), and $\ell(\boldsymbol{\theta})$ denote a pre-training loss, *e.g.* the normalized temperature-scaled cross-entropy (NT-Xent) loss used in CL [6], [7], [37]. In the supervised fine-tuning phase, let $\boldsymbol{\theta}_{\text{ft}}$ denote the prediction head appended to the representation network $\boldsymbol{\theta}$, and $\ell_{\text{ft}}(\boldsymbol{\theta}_{\text{ft}} \circ \boldsymbol{\theta})$ denote a task-specific fine-tuning loss seen as a function of the entire model $\boldsymbol{\theta}_{\text{ft}} \circ \boldsymbol{\theta}$, where $\circ$ denotes model composition. Fine-tuning is performed over labeled data. The SSL pipeline can then be summarized as

$$\begin{aligned} \text{Pre-training:} \quad & \boldsymbol{\theta}_{\text{pre}} = \arg\min_{\boldsymbol{\theta}} \ \ell(\boldsymbol{\theta}), \\ \text{(Full) Fine-tuning:} \quad & \underset{\boldsymbol{\theta}_{\text{ft}}, \boldsymbol{\theta}}{\text{minimize}} \ \ell_{\text{ft}}(\boldsymbol{\theta}_{\text{ft}} \circ \boldsymbol{\theta}), \\ & \text{with initialization } \boldsymbol{\theta} = \boldsymbol{\theta}_{\text{pre}}. \end{aligned} \tag{1}$$

We remark that if we fix $\boldsymbol{\theta} = \boldsymbol{\theta}_{\text{pre}}$ in (1) during fine-tuning, then the resulting scheme is called *partial fine-tuning* (**PF**), which only learns the prediction head $\boldsymbol{\theta}_{\text{ft}}$. Based on (1) for code, this work tackles the following research questions:

(Q1) How to design a self-supervised pre-training scheme to acquire $\boldsymbol{\theta}_{\text{pre}}$ that is robust to obfuscating codes?
(Q2) How to design a supervised fine-tuning scheme that can not only preserve the generalization and robustness abilities gained from pre-training but also achieve new improvements via task-driven supervised learning?

IV. Method

In this section, we will study the above (Q1)-(Q2) in-depth. To address (Q1), we will develop a new pre-training method, termed CLAW, which integrates CL with adversarial views of codes. The rationale is that promoting the invariance of representations to possible adversarial candidates should then likely improve the robustness of models fine-tuned on these representations. To answer (Q2), we will a novel fine-tuning method, termed staggered adversarial training (SAT), which can balance the supervised fine-tuning with unsupervised pre-training. The rationale is that the supervised fine-tuning overrides pre-trained data representations and hardly retains the robustness and generalization benefits achieved during pre-training. We will show that the interplay between pre-training and fine-tuning should be carefully studied for robustness-generalization co-improvement in SSL for code.

A. CLAW: *CL with adversarial codes*

CL [6], [7] proposes to first construct 'positive' example pairs (*i.e.*, original data paired with its transformations or 'views'), and then maximize agreement between them while contrasting with the rest of the data (termed 'negatives'). In programming languages, code obfuscation transformations naturally serve as view generators of an input code. While we reuse the same set of transformations (that are applicable to Python and Java programs) employed by the prior work [5], we generate transformed views of the code differently. [5] use random views in their CL setup, which select and apply a transformation at random from the set of permissible transformations. We generate worst-case, optimization-based adversarial codes [10], [12] resulting in 'adversarial views'.

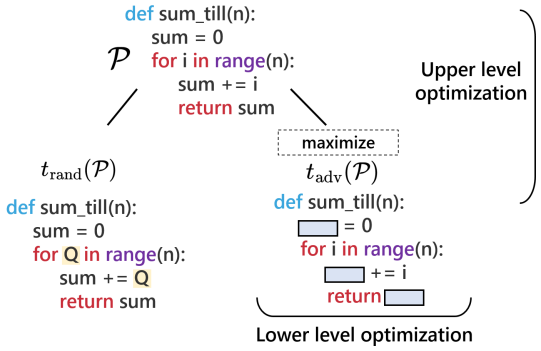

Fig. 3: During pre-training, we propose **CLAW** containing two optimization problems: (1) to learn invariant code representations by minimizing the representation distances of a code ($\mathcal{P}$) from all its views ($t_{\text{rand}}(\mathcal{P})$, $t_{\text{adv}}(\mathcal{P})$) via CL, and (2) to generate an adversarial code $t_{\text{adv}}(\mathcal{P})$ ('hard' positive example) by maximizing its representation distance from $\mathcal{P}$. In the example, this requires solving for a replacement token at the randomly selected *site* marked as ▪.

Infusing the original code, its random view, and its adversarial view into CL, we obtain a *three-view* positive tuple, denoted by ($\mathcal{P}$, $t_{\text{rand}}(\mathcal{P})$, $t_{\text{adv}}(\mathcal{P})$). Since the generation of adversarial code (the right tokens for a given site) is in itself an additional optimization task, we leverage a bi-level optimization (BLO) framework [38], [39] and define optimization problems at two levels: the upper-level problem aims to solve the multi-view CL, while the lower-level problem aims to solve adversarial code generation (see an illustration in Figure 3). This results in the following formulation for our proposed approach CLAW:

$$\begin{aligned} \underset{\boldsymbol{\theta}}{\text{minimize}} \;\; & \mathbb{E}_{\mathcal{P},t_{\text{rand}}}[\ell_{\text{NT-Xent}}(\boldsymbol{\theta};\mathcal{P},t_{\text{rand}}(\mathcal{P}))] + \underbrace{\mathbb{E}_{\mathcal{P}}[\ell_{\text{NT-Xent}}(\boldsymbol{\theta};t_{\text{rand}}(\mathcal{P}),t_{\text{adv}}(\mathcal{P}))]}_{\text{Upper level: Multi-view CL}} \\ \text{subject to} \;\; & t_{\text{adv}}(\mathcal{P}) = \underbrace{\arg\max_{\mathcal{P}'} \ell_{\text{NT-Xent}}(\boldsymbol{\theta};\mathcal{P},\mathcal{P}')}_{\text{Lower level: Adversarial code generation}}, \end{aligned} \tag{2}$$

where the three-view objective function is constructed by NT-Xent losses applied to two positive pairs $(\mathcal{P}, t_{\text{rand}}(\mathcal{P}))$ and $(t_{\text{rand}}(\mathcal{P}), t_{\text{adv}}(\mathcal{P}))$, respectively. The first positive pair is to gain the generalizable code representation by promoting the representation invariance across the original view $\mathcal{P}$ and its (benign) randomly-obfuscating view t_{rand}, as suggested in [5]. The second positive pair is to enforce the adversary-resilient code representation by promoting the representation invariance across the benign code t_{rand} and the adversarial code $t_{\text{adv}}(\mathcal{P})$. Given two codes $\mathcal{P}_1$ and $\mathcal{P}_2$, the specific form of $\ell_{\text{NT-Xent}}$ is given by as follows:

$$\ell_{\text{NT-Xent}} = -\tfrac{1}{2}\sum_{i=1}^{2} \log \frac{\exp\left(\text{sim}(\mathbf{z}_1(\boldsymbol{\theta}),\mathbf{z}_2(\boldsymbol{\theta}))/t\right)}{\sum_{k\in\mathcal{N}(i)},\exp\left(\text{sim}(\mathbf{z}_i(\boldsymbol{\theta}),\mathbf{z}_k(\boldsymbol{\theta}))/t\right)} \tag{3}$$

where $\mathbf{z}_i$ denotes the feature representation of the input code $\mathcal{P}_i$ achieved through the representation network $\boldsymbol{\theta}$, $\text{sim}(\mathbf{z}_i, \mathbf{z}_j)$ denotes the cosine similarity between two feature representations $\mathbf{z}_i$ and $\mathbf{z}_j$, $t > 0$ is a temperature parameter, and $\mathcal{N}(i)$ is the set of batch data except the data sample i [6].

To solve the BLO problem (2), we apply an alternating optimization method [38], [40]. Specifically, by fixing the representation network $\boldsymbol{\theta}$, the lower-level adversarial code generation is accomplished using first-order gradient descent following [10], [12]. Given the generated adversarial code $t_{\text{adv}}(\mathcal{P})$, we then in turn solve the upper-level CL problem. The above procedure is alternatively executed for every data batch.

Adversarial view is beneficial to representation learning. To highlight the effectiveness of incorporating adversarial codes in CL at the pre-training phase, the rows 'CONTRACODE-PF' and 'CLAW-PF' of Table I demonstrate a warm-up ex-

periment by comparing the performance of the proposed pre-training method CLAW with that of the baseline approach CONTRACODE [5]. To precisely characterize the effect of the learned representations on code model generalization and robustness, we partially fine-tune (PF) a SEQ2SEQ model on the downstream task of summarizing code (details in Section V) in Python (**SUMMARYPY**) and Java (**SUMMARYJAVA**) by fixing the set of weights learned during pre-training. **GEN-F1** and **ROB-F1** are the generalization F1-scores and the robust F1-scores, *i.e.*, F1 scores of the model when attacked with adversarial codes. Partially fine-tuning these models allows us to study the sole contribution of the pre-training method in the learned robustness and generalization of the model. As we can see, the partially fine-tuned CLAW model (termed CLAW-PF) outperforms the baseline CONTRACODE-PF, evidenced by the substantial robustness improvement (4.38% increase in ROB-F1 in SUMMARYPY) as well as lossless or better generalization performance (1.58% increase in GEN-F1 scores on SUMMARYJAVA). It is worth noting that adversarial codes serve as 'hard' positive examples in the representation space (given by maximizing the representation distance between $\mathcal{P}$ and its perturbed variant $\mathcal{P}'$ in the lower optimization level of CLAW, Eq. 2). The benefit of hard positive examples in improving generalization has also been seen in vision [41]–[43].

Model	Partial fine-tuning			
	SUMMARYPY		SUMMARYJAVA	
	GEN-F1	ROB-F1	GEN-F1	ROB-F1
CONTRACODE-PF	25.46	15.47	20.92	16.63
CLAW-PF	25.45	19.05	22.50	17.14
	Full fine-tuning			
CONTRACODE-ST	36.28	28.97	41.37	33.01
CLAW-ST	36.57	29.97	41.23	32.53
CONTRACODE-AT	32.80	32.39	38.67	35.91
CLAW-AT	32.97	32.65	38.86	36.10

TABLE I: Partially fine-tuned (PF) models show that CLAW improves robustness. Standard training (ST) yields better generalization than adversarial training (AT) while the latter provides better robustness.

B. SAT: *Staggered adversarial training for fine-tuning*

As shown in the previous section, an appropriate pre-training method can improve the quality of learned deep representations, which help improve the robustness and accuracy of a code model. However, the state of the model present at the end of the pre-training phase may no longer hold after supervised fine-tuning. That is because supervised learning (trained on labeled data vs. unlabeled data in representation learning) may significantly alter the characteristics of the learned representations. Thus, a desirable fine-tuning scheme should be able to yield accuracy and robustness improvements *complementary* to the representation benefits provided by pre-training. Towards this goal, we posit that fine-tuning should *not* be designed in a way which merely optimizes a single performance metric–either accuracy or robustness.

To justify this hypothesis, we consider two extreme cases during fine-tuning: *(i)* standard training (ST)-based FF, and *(ii)* adversarial training (AT)-based FF [18]. ST is essentially the same setup as fully supervised training with the only difference being in the set of initial parameters of the model. This setup optimizes improving a model's generalization ability. On the other hand, AT optimizes improving the model's adversarial robustness.

The *last four rows of Table I* present the performance of these two extreme fine-tuning cases applied to the pre-trained models provided by CONTRACODE [5] and CLAW, respectively. As we can see, when either ST or AT is used, different pre-training methods (CONTRACODE and CLAW) lead to nearly the same generalization and robustness performance. This shows that fine-tuning, when aggressively optimizing one particular performance metric, could override the benefits achieved during pre-training. To this end, we propose staggered AT (SAT), a hybrid of ST and AT by adjusting the time instances (in terms of epoch numbers) at which adversarial codes are generated (see Algorithm 1). SAT involves two key steps—training

Algorithm 1: Staggered Adversarial Training (SAT)

Input: model $\mathcal{M} = \{\boldsymbol{\theta}_{\text{ft}}, \boldsymbol{\theta}\}$, attack frequency τ
for each epoch e **do**
  for each data batch $\mathcal{B}_i$ **do**
    1. Train $\mathcal{M}$ by updating $\boldsymbol{\theta}_{\text{ft}} \circ \boldsymbol{\theta}$
  end for
  if e mod τ = 0 **then**
    for each data batch $\mathcal{B}_i$ **do**
      2. Attack $\mathcal{M}$ by finding adversarial codes $\mathcal{B'}_i$
      3. Retrain $\mathcal{M}$ on $\mathcal{B'}_i$
    end for
  end if
end for

a model $\mathcal{M} := \{\boldsymbol{\theta}, \boldsymbol{\theta}_{\text{ft}}\}$ on a batch $\mathcal{B}$ of data (step 1), and attacking the learned model at a staggered frequency τ (steps 2-3). *Different from* AT, adversarial code is not generated in every data batch. Instead, in SAT, we propose reducing the frequency of adversarial learning. Accordingly, adversarial code generation occurs at the frequency of each epoch or at every few epochs. This ensures the model parameters retain as much of the attributes from pre-training while also learning task-specific generalization and robustness. In SAT, the model is finetuned using ($\mathcal{B}_i$ + $\mathcal{B}'_i$) where $\mathcal{B}'_i$ refers to the generated adversarial code corresponding to $\mathcal{B}_i$. Eventually, by combining the proposed pre-training scheme CLAW with the fine-tuning scheme SAT, we term the resulting SSL framework for code as CLAWSAT.

V. EXPERIMENT SETUP

We describe the following aspects of our experiment setup - the task, dataset, and the details of the model.

Task, dataset, error metrics. We evaluate our algorithm on **four** tasks: **(1)** code summarization [5], [36], [44]–[47] in Python, **(2)** code summarization in Java (generates English

	Model	SUMMARYPY		SUMMARYJAVA		COMPLETEPY		CLONEJAVA	
		GEN-F1	ROB-F1	GEN-F1	ROB-F1	GEN-F1	ROB-F1	GEN-F1	ROB-F1
M_1	Supervised learning	$33.33_{\pm 0.17}$	$26.16_{\pm 0.31}$	$38.42_{\pm 0.25}$	$29.89_{\pm 0.27}$	$56.72_{\pm 0.22}$	$53.89_{\pm 0.26}$	$67.20_{\pm 0.11}$	$64.35_{\pm 0.15}$
M_2	M_1-AT [10]	$33.03_{\pm 0.21}$	$32.20_{\pm 0.26}$	$37.81_{\pm 0.23}$	$34.86_{\pm 0.29}$	$55.40_{\pm 0.26}$	$55.34_{\pm 0.35}$	$66.12_{\pm 0.13}$	$65.84_{\pm 0.17}$
M_3	CONTRACODE [5]	$36.28_{\pm 0.18}$	$28.97_{\pm 0.27}$	$41.37_{\pm 0.14}$	$33.01_{\pm 0.25}$	$57.70_{\pm 0.23}$	$54.83_{\pm 0.31}$	$69.25_{\pm 0.09}$	$68.86_{\pm 0.13}$
M_4	CLAW-ST	$36.57_{\pm 0.20}$	$29.97_{\pm 0.22}$	$41.23_{\pm 0.17}$	$33.53_{\pm 0.22}$	$57.65_{\pm 0.25}$	$54.75_{\pm 0.31}$	$69.63_{\pm 0.09}$	$68.95_{\pm 0.15}$
M_5	CLAW-AT	$32.97_{\pm 0.15}$	$32.65_{\pm 0.17}$	$38.86_{\pm 0.23}$	$36.10_{\pm 0.31}$	$57.44_{\pm 0.26}$	$57.12_{\pm 0.29}$	$69.40_{\pm 0.16}$	$69.00_{\pm 0.14}$
M_6	**CLAWSAT (ours)**	$\mathbf{42.12}_{\pm 0.19}$	$\mathbf{40.70}_{\pm 0.23}$	$\mathbf{41.77}_{\pm 0.27}$	$\mathbf{38.80}_{\pm 0.33}$	$\mathbf{58.80}_{\pm 0.24}$	$\mathbf{57.21}_{\pm 0.28}$	$\mathbf{69.73}_{\pm 0.10}$	$\mathbf{69.25}_{\pm 0.13}$

TABLE II: **Overall performance of CLAWSAT**: We evaluate our models in two settings: standard training (ST) and adversarial training (AT) by [18]. For each of the four tasks: code summarization: SUMMARYPY, SUMMARYJAVA, code completion: COMPLETEPY, and code clone detection: CLONEJAVA, we report the model's generalization F1-score (GEN-F1) and the robustness F1 (ROB-F1)–the generalization F1 when the model is adversarially attacked. M_2 corresponds to an adversarially trained (AT) version of the supervised model M_1, first introduced in [10]. M_4 and M_5 are two variants of CLAWSAT (M_6): one integrates CLAW with standard training (ST), and the other integrates CLAW with the adversarial training (AT) [18]. The result $a_{\pm b}$ represents mean a and standard deviation b, calculated over 5 random trials.

description for given code snippet), **(3)** code completion in Python [48] (generates the next six tokens for a given code snippet), and **(4)** code clone detection [49] in Java (classifies whether a pair of code snippets are clones of each other). For models evaluated in Python, we pre-train on the PY-CSN dataset [50], containing $\sim$ 500K methods, and fine-tune on the PY150 dataset [51], containing $\sim$ 200K methods. For Java, we pre-train on the JAVA-CSN dataset [50] containing $\sim$ 600K and fine-tune on the JAVA-C2S [36] dataset containing $\sim$ 500K methods. We use the F1-score $\in [0, 100]$ to measure the performance of all our models, consistent with all the related works, including [5]. A higher value indicates that the model generalizes better to the task. While these F1-scores are correlated to BLEU scores, they directly account for token-wise mis-predictions. Specifically, for each of our models, we reuse the two F1 scores metrics: GEN-F1- the model's generalization performance on a task, and ROB-F1- the model's performance on the task when semantics-preserving, adversarially-transformed obfuscated codes are input to it; see details below.

Adversarial attacks, attack strength, code transformations. When attacking code models, we use the formulation by [12] to define the strength of an attack. Specifically, selecting a larger number of *sites* in a code—locations or tokens in a code which can be *transformed* to produce an adversarial outcome—corresponds to a stronger adversarial attack, since this allows multiple changes to be made to the code. Also, we follow [10], [12] to specify the set of code transformations: replace (renaming local variables, renaming function parameters, renaming object fields, replacing boolean literals) and insert (inserting print statements, adding dead code).

In our setup, we can leverage the attack strength at three stages–during pre-training (using adversarially attacked code as views), when fine-tuning with adversarial training AT [18] or SAT, and when evaluating robustness on an unseen test set. Unless specified otherwise, we pre-train on one *site*, attack one *site* in each iteration of SAT, and attack one *site* during evaluation. In Table VI ,we analyze the effect of varying the number of sites at each of these stages. We apply the first-order optimization method proposed in [10] to generate adversarial codes. To adversarially train these models during fine-tuning, we employ either AT [18] or our proposed SAT for code.

Baselines. We compare CLAWSAT to **three** baselines. **(1)** A supervised model (model $\mathbf{M_1}$ in Table II) - Pre-training has no effect on a fully supervised model. **(2)** Adversarially trained supervised model ($\mathbf{M_2}$) on top of M_1 - we use the AT setup first proposed by [10], which in turn employs the setup from [18]. Due to the characteristics of AT, we expect to see an improvement in its ROB-F1 as compared to M_1 but a decrease in GEN-F1. **(3)** The CONTRACODE model ($\mathbf{M_3}$) from [5]. CONTRACODE reflects the state-of-the-art in pre-training methods as it outperforms other pre-training models like BERT-based models and GPT3-Codex. Hence, we do not compare ourselves again to other pre-training models.

Models. For the summarization and completion tasks, we experiment with two seq2seq architectures–LSTMs and transformers. For the detection task, we use a fully connected linear layer as a decoder. The decoders are trained to predict the task (generating English sentence summaries, generating code completions, flagging code clones respectively) in both the fine-tuned and standard training settings. When fine-tuning, we use the learned encoders from the pre-trained models. For the summarization and completion tasks, we report all our results on the LSTM decoder (Table II), and compare the performance of transformers in our ablation study. The LSTM encoder has 2 layers across all experiments. In the code summarization tasks, there exists another two-layer decoder added to the encoder to generate the summary of the programming language. In the code completion task, we also add a two-layer decoder to generate the code snippets. In the code clone detection task, we add one linear layer to map the data representations to the data labels. As for the transformer architecture, The transformer encoder has 6 layers and the transformer decoder has another 6 layers to generate the programming language summary.

Hyperparameter setup. We optimize model parameters using Adam with linear learning rate warm-up. For the bidirectional

LSTM encoders, the maximum learning rate for CONTRACODE and CLAW is 10^{-4}, and then is decayed accordingly. For the transformer-type encoder, the maximum learning rate is 10^{-4} for CONTRACODE and 10^{-5} for CLAW. For different downstream tasks, we optimize the parameters using Adam with step-wise learning rate decay. The maximum learning rates of ST, AT, SAT are 10^{-3} on code summarization and code completion tasks, and 10^{-4} for code clone detection tasks. For downstream tasks using transformer, the learning rates of ST, AT, SAT are 10^{-4}. All of the downstream tasks are finetuned for 10 epochs with practical convergence, and we utilize a validation dataset to pick the best-performed model. All of the experiments are conducted on 4 Tesla V100 GPU with 16 GB memory.

VI. EXPERIMENT RESULTS

We summarize the overall performance of CLAWSAT and follow that up with multiple additional analyses and ablations to better understand our model's performance.

A. Overall performance

We evaluate the different pre-training and fine-tuning strategies we consider in Section IV. The accuracy and robustness F1-scores of the different models are shown in Table II. Models M_1-M_3 are the baselines described in the previous section. Model M_4 pertains to using a CLAW encoder with standard training (ST) for the downstream task. Model M_5 pertains to using a CLAW encoder with standard adversarial training (AT) for the downstream task. And finally, M_6 pertains to using SAT for the downstream task with a CLAW encoder. We observe the following:

First, from the perspective of accuracy, Table II shows that our proposal CLAWSAT (model M_6) outperforms all the baselines on GEN-F1. Particularly, CLAWSAT achieves nearly 8% accuracy improvement over CONTRACODE (M_3) on SUMMARYPY. **Second**, we see that CLAWSAT can substantially help improve the robustness (measured by ROB-F1) of a downstream model. Compared to M_3, we see an improvement of 14.7% for SUMMARYPY, and 5.8% for SUMMARYJAVA. Similarly, the gain in robustness is much more substantial in COMPLETEPY than in accuracy. CLONEJAVA is the simplest of the four tasks, and we see comparable accuracies across all the models we evaluate. **Additionally**, we adversarially train baselines as well (models M_2 and M_5 respectively) and compare their robustness scores to ours (M_6). We make two observations–(a) As is expected with AT, we notice a drop in these models' accuracy–the GEN-F1 scores of M_5 is lower than that of its standard-training counterpart M_4 (the same trend is observed between M_2 and M_1 as well). (b) While AT-based fine-tuning provides an expected improvement in robustness over the other baselines that use ST-based fine-tuning, the ROB-F1 they achieve is still much lower than our model. This is because the robustness gain at the pre-training phase was overridden by AT at the fine-tuning phase.

In summary, **Table II** shows that the proposed CLAWSAT allows us to learn task-specific accuracy and robustness while preserving these attributes learned during pre-training.

In what follows, we analyze the performance of CLAW and SAT separately from different perspectives.

B. Why is CLAW *effective? A model landscape perspective*

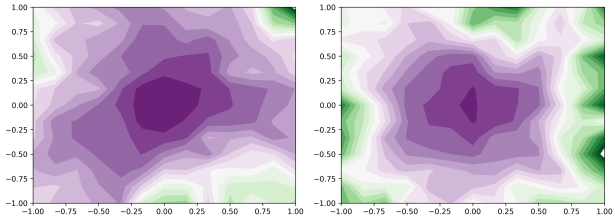

Fig. 4: Loss landscapes: CLAW (L), and CONTRACODE (R), the $\mathbf{X}$ and $\mathbf{Y}$ axes represent the directional coefficients α and β in Eq. 4.

Liu *et. al.* [52] show that the generalization benefit of an approach in the'pre-training + fine-tuning' paradigm can be deduced by the flatness of the loss landscape of the pre-trained model. This would then let fine-tuning to force the optimization for fine-tuning to stay in a certain neighborhood of the pre-trained model of high quality.

To plot the loss landscapes, we follow the procedure from Li *et. al.* [53] by plotting

$$f(\alpha, \beta) = \ell(\theta^\star + \alpha\delta + \beta\eta) \tag{4}$$

where δ and η are two random direction vectors in the parameters' subspace, and $\theta^\star$ is the parameters of a model. We average the supervised loss from the partially fine-tuned models from 640 randomly selected samples in our test set (out of 10000, 6.4%).

Results of **Figure 4** confirm the **flatness** of the loss landscape in CLAW when compared to CONTRACODE, implying a better transfer of generalizability by CLAW.

Next, we show another way to justify the flatness merit of CLAW's loss landscape. The key idea is to track the deviations of the weights of the pre-trained encoders in the fine-tuned setting, as inspired by [52]. Specifically, let $\boldsymbol{\theta}_{\text{pre}}$ and $\boldsymbol{\theta}'_{\text{pre}}$ denote the weights of the representation model $\boldsymbol{\theta}$ pre-trained by CLAW and the fine-tuned weights obtained by using different fine-tuning methods, respectively. It was shown in [52] that the generalization benefit of an approach in the'pre-training + fine-tuning' paradigm can be deduced by the deviation between the fine-tuned weights $\boldsymbol{\theta}'_{\text{pre}}$ and the pre-trained weights $\boldsymbol{\theta}_{\text{pre}}$. This would then let fine-tuning to force the optimization of $\boldsymbol{\theta}'_{\text{pre}}$ to stay in a certain neighborhood of $\boldsymbol{\theta}_{\text{pre}}$. We have already shown that CLAW will lead to a flatter loss landscape compared

```
def _makeOne(self,discriminator=None
    ,family=None):
    from ..index import AllowedIndex
    index = AllowedIndex(discriminator,family=family)
    return index
```

(a) Sample

```
def _makeOne(self,repeat=None
    ,family=None):
    from ..index import AllowedIndex
    index = AllowedIndex(repeat,family=family)
    return index
```

(b) Adversarially perturbed version of sample (a1)

```
def _makeOne(self,discriminator=None
    family=None,action_Mode=None):
    from ..indexes import FieldIndex
    return FieldIndex(discriminator,family,action_mode)
```

(c) EBE similar to (a) - CLAW

```
def _makeOne(self,discriminator=None
    family=None,action_Mode=None):
    from ..indexes import FieldIndex
    return FieldIndex(discriminator,family,action_mode)
```

(d) EBE similar to (a) - CONTRACODE

```
def _makeOne(self,discriminator=None
    family=None,action_Mode=None):
    from ..indexes import FieldIndex
    return FieldIndex(discriminator,family,action_mode)
```

(e) EBE similar to (b) - CLAW

```
def buildIndex(self,l):
    index = self.mIndex()
    for strat, end, value in self.l:
        index.add(strat,end)
    return index
```

(f) EBE similar to (b) - CONTRACODE

Fig. 5: Explanation-by-example to demonstrate the robustness benefits of CLAW. **(a)** Sample program from the test set **(b)** Adversarially perturbed variant of the sample program. **(c-d)** Examples closest to the sample program (a) when using CLAW and CONTRACODE. **(e-f)** Examples closest to the perturbed variant (b) when using CLAW and CONTRACODE.

to CONTRACODEpreviously. Here we computed the Frobenius norm $||\boldsymbol{\theta}'_{\text{pre}} - \boldsymbol{\theta}_{\text{pre}}||_{\text{F}}$ as a proxy to justify the generalization benefit following [52].

$\boldsymbol{\theta}'_{\text{pre}}$	$\|\|\boldsymbol{\theta}'_{\text{pre}} - \boldsymbol{\theta}_{\text{pre}}\|\|_{\text{F}}$
CLAW-ST	207.99
CLAW-AT	323.41
CLAWSAT	232.56
CONTRACODE-ST	218.14
CONTRACODE-AT	345.36
CONTRACODE-SAT	242.70

TABLE III: Weight difference after finetuning based on different pretraining methods.

Table III summarizes the aforementioned weight characteristics for the code summarization task. As we can see, the weight deviation corresponding to CLAW is less than that associated with CONTRACODE given a finetuning method.

> The results from **Table III** suggest that an encoder pretrained using CLAW transfers better than that using CONTRACODE.

C. Interpretability of learned code representations

We evaluate the robustness benefit of CLAW through the lens of (input-level) model explanation. Following the observations from Jeyakumar *et. al.* [54] on probing models locally, we investigate CLAW and CONTRACODE using a training data-based model explanation method: explanation-by-example (EBE) [55]. The core idea is to leverage train-time data to explain test-time data by matching their respective representations. If the pretrained models are robust, adversarially perturbing the samples should not alter their representations and thus should be mapped to the same set of closest training examples that were found without perturbations.

> Based on EBE, we sample 100 code snippets $\{\mathcal{P}_i^{\text{test}}\}_{i=1}^{100}$ at random from our test dataset, and find the closest samples $\{\mathcal{P}_{\text{CLAW}}^{\text{train}}\}$ and $\{\mathcal{P}_{\text{CONTRACODE}}^{\text{train}}\}$ in the training dataset using the EBE method, based on representations produced by $\boldsymbol{\theta}_{\text{CLAW}}$ and $\boldsymbol{\theta}_{\text{CONTRACODE}}$ respectively. We find that 68% of the representations from CLAW match their original codes in $\{\mathcal{P}_{\text{CLAW}}^{\text{train}}\}$ while 57% of CONTRACODE representations match their original codes in $\{\mathcal{P}_{\text{CONTRACODE}}^{\text{train}}\}$. **The above results** suggest that the learned representations by CLAW are more adversarially robust than CONTRACODE.

In what follows, we peer into the EBE method's results with an example below.

- **A sample program from the test-set:**

```
def __init__(self,helper_name):
    self.helper_name = helper_name
    self.cheeks = []
```

- **Adversarially perturbed variant of the sample program:** The adversarial attack algorithm replaces the method argument `helper` with `edges`.

```
def __init__(self,edges):
    self.helper_name = edges
    self.cheeks = []
```

- **EBE sample in the train-set closest to the sample program when using CONTRACODE or CLAW:** This is the example whose CONTRACODE or CLAW representation (encoder trained by CONTRACODE or CLAW) is closest to the representation of the sample program. We can find that they have the same

functionality.

```
def __init__(self,name):
    self.name = name
    self.warning = []
```

• **EBE sample in the train-set closest to the perturbed variants of sample program from CONTRACODE:** We can observe that the closest program of the perturbed program in the training dataset is different from that of the original program. The new closest program has different functionality from the previous test sample program.

```
def setUp(self):
    super(ApiCallHandlerRegressionTest
        ↪ ,self).setUp()
    self.checks = []
```

• **EBE sample in the training-set closest to the perturbed sample program from CLAW:** This example pertains to the representation that is closest to the representation of the sample program computed by an encoder trained by CLAW. We see that despite comparing it to a perturbed sample's representation, the example found by EBE corresponds to the unpertrubed sample program, suggesting the robustness of CLAW over CONTRACODE.

```
def __init__(self,name):
    self.name = name
    self.warning = []
```

We also provide more examples in Figure 5. Figure 5.a shows a sample Python program and Figure 5.b shows its respective adversarially perturbed variant. The closest training programs in the training set mapped to the representations before perturbations are shown in Figures 5.c and 5.d; and those mapped to the representations after perturbations are shown in Figures 5.e and 5.f.

> As shown in **Figure 5**, we find that EBE consistently finds the same training examples for CLAW (Figure 5.c and Figure 5.e) irrespective of the adversarial perturbations made to the sample program, confirming its enhanced robustness.

D. SAT *enables generalization-robustness sweet spot*

Figure 6.(A) shows the results from our experiments on the code summarization task where we vary τ, the frequency of attacking the code model during SAT (see Algorithm 1).

We generate adversarial code tokens every τ^{th} epoch, where we vary τ from less than 1 (corresponds to an update occurring at every batch within an epoch; this is the AT algorithm from [18]) to 10. The X-axis shows this frequency. We plot both GEN-F1 (left) and ROB-F1 (right) of the adversarially trained model when varying n. Across the four tasks we evaluate in this work, We find that a sweet spot exists in a less frequent epoch-wise schedule, associated with CLAWSAT (M_6, which corresponds to $\tau = 1$), which improves both GEN-F1 and ROB-F1 over M_5 (which corresponds to $\tau = 0.1$).

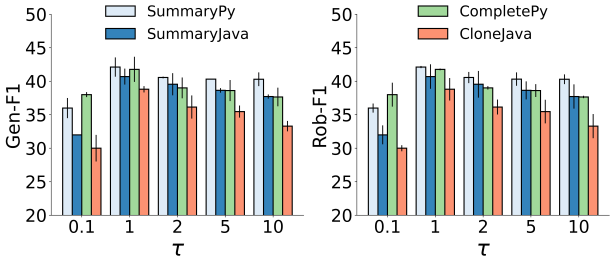

Fig. 6: Effect of different update schedules (τ, see Algorithm 1) on GEN-F1 and ROB-F1.

> Results of **Figure 6** validate our hypothesis of being able to retain the robustness learned during pre-training while updating the model just enough during fine-tuning to 'learn' new robustness while also learning the downstream task.

E. CLAWSAT *on a different architecture*

	Model	SUMMARYPY GEN-F1	SUMMARYPY ROB-F1
M_1	Supervised learning	$32.60_{\pm 0.14}$	$30.09_{\pm 0.21}$
M_2	M_1-AT [10]	$31.18_{\pm 0.13}$	$30.66_{\pm 0.23}$
M_3	CONTRACODE [5]	$34.93_{\pm 0.11}$	$32.86_{\pm 0.18}$
M_4	CLAW-ST	$35.37_{\pm 0.14}$	$32.31_{\pm 0.20}$
M_5	CLAW-AT	$34.23_{\pm 0.19}$	$33.34_{\pm 0.18}$
M_6	**CLAWSAT (ours)**	$\mathbf{36.39}_{\pm 0.12}$	$\mathbf{35.53}_{\pm 0.24}$

TABLE IV: Overall performance of CLAWSAT on transformer

We further evaluate SAT on a different model architecture. We consider the transformer architecture (6-layer encoder and 6-layer decoder following [5]), and observe similar results (see Table IV): CLAWSAT offers the best accuracy and robustness.

> **Table IV** shows that CLAWSAT performs well across multiple encoder architectures.

F. Extended study to integrate SAT *with* CONTRACODE

To further verify the effectiveness of SAT, we employ it in CONTRACODE—we modify their implementation to introduce a staggered adversarial training schedule. Table V tabulates its performance. We find that SAT also benefits CONTRACODE, but the gain is smaller than CLAWSAT (M_6, Table II). This justifies the complementary benefits of CLAW and SAT.

Model	SUMMARYPY		SUMMARYJAVA		COMPLETEPY		CLONEJAVA	
	GEN-F1	ROB-F1	GEN-F1	ROB-F1	GEN-F1	ROB-F1	GEN-F1	ROB-F1
CONTRACODE [5]	$36.28_{\pm 0.18}$	$28.97_{\pm 0.27}$	$41.37_{\pm 0.14}$	$33.01_{\pm 0.25}$	$57.70_{\pm 0.23}$	$54.83_{\pm 0.31}$	$69.25_{\pm 0.09}$	$68.86_{\pm 0.13}$
CONTRACODE-AT	$35.88_{\pm 0.20}$	$31.29_{\pm 0.24}$	$38.67_{\pm 0.27}$	$35.91_{\pm 0.30}$	$57.21_{\pm 0.19}$	$56.80_{\pm 0.22}$	$69.20_{\pm 0.13}$	$68.88_{\pm 0.11}$
CONTRACODE-SAT	$41.01_{\pm 0.20}$	$39.80_{\pm 0.21}$	$41.27_{\pm 0.16}$	$38.14_{\pm 0.24}$	$58.04_{\pm 0.17}$	$57.01_{\pm 0.30}$	$69.47_{\pm 0.10}$	$69.08_{\pm 0.21}$

TABLE V: Effectiveness of SAT on CONTRACODE

Table V shows the complementary benefits of CLAW and SAT on the state-of-the-art SSL method CONTRACODE as well, demonstrating the effectiveness of the two model-independent techniques we introduce in this work.

G. Sensitivity of SAT *to code transformation and attack strength types.*

		Transformations (GEN-F1, ROB-F1)		
		Fine-tuning		
Pre-training		replace	insert	All
	replace	41.51, 39.84	40.49, 34.96	42.32, 40.75
	insert	41.71, 40.38	41.34, 35.24	41.71, 40.38
	All	42.66, 40.54	41.39, 34.91	42.12, 40.70

	Attack strength (ROB-F1)				
	1	**2**	**3**	**4**	**5**
CONTRACODE	28.97	28.29	27.32	26.24	25.14
CLAWSAT	40.70	40.58	39.67	38.90	38.10

TABLE VI: **Performance of CLAWSAT at different attack configurations**. We evaluate the sensitivity of our best performing model on (a) different transformation types used during pre-training and fine-tuning (SAT) (b) different attack strengths (number of *sites*) during evaluation.

We evaluate the sensitivity of our best-performing model on differing attack conditions. We consider two factors: (1) Transformation type: we study the effect of the two transformation types–replace and insert, and their combination. (2) Attack strength: we vary the number of *sites*—locations in the codes that can be adversarially transformed.

We summarize our results in Table VI. The values `a`, `b` in each cell correspond to GEN-F1 and ROB-F1 of CLAWSAT respectively.

The results from **Table VI** suggest that when using transformations in pre-training or in fine-tuning, it is advisable to use a combination of both replace and insert transformations. When evaluating CLAWSAT's robustness against stronger adversarial attacks, we find ROB-F1 consistently outperforms CONTRACODE in all the configurations we evaluate.

VII. CONCLUSION & DISCUSSION

In this work, we aim to achieve the twin goals of improved robustness and generalization in SSL for code, specifically in contrastive learning. We realize this by proposing two improvements–adversarial positive views in contrastive learning, and a staggered AT schedule during fine-tuning. We find that each of these proposals provides substantial improvements in both the generalization and robustness of downstream models; their combination, CLAWSAT, provides the best overall performance. Given the growing adoption of SSL-based models for code-related tasks, we believe our work lays out a framework to gain a principled understanding into the working of these models. Future works should study this problem while attempting to also establish a theoretically sound foundation.

When compared to SSL for vision, it seems SSL for codes benefits from milder adversarial training during fine-tuning. Stronger attack methods for code might be needed to explore this phenomenon further. It will also be beneficial to understand how code models respond to perturbations, and to contrast it to our understanding of continuous data perturbations in vision.

REFERENCES

[1] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, "Learning and evaluating contextual embedding of source code," in *International Conference on Machine Learning*. PMLR, 2020, pp. 5110–5121.

[2] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021.

[3] Z. Chen, V. J. Hellendoorn, P. Lamblin, P. Maniatis, P.-A. Manzagol, D. Tarlow, and S. Moitra, "PLUR: A unifying, graph-based view of program learning, understanding, and repair," in *Advances in Neural Information Processing Systems*, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., 2021. [Online]. Available: https://openreview.net/forum?id=GEm4o9A6Jfb

[4] N. Jain, S. Vaidyanath, A. S. Iyer, N. Natarajan, S. Parthasarathy, S. K. Rajamani, and R. Sharma, "Jigsaw: Large language models meet program synthesis," *CoRR*, vol. abs/2112.02969, 2021. [Online]. Available: https://arxiv.org/abs/2112.02969

[5] P. Jain, A. Jain, T. Zhang, P. Abbeel, J. Gonzalez, and I. Stoica, "Contrastive code representation learning," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Online and Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021, pp. 5954–5971. [Online]. Available: https://aclanthology.org/2021.emnlp-main.482

[6] T. Chen, S. Kornblith, M. Norouzi, and G. Hinton, "A simple framework for contrastive learning of visual representations," in *International conference on machine learning*. PMLR, 2020, pp. 1597–1607.

[7] K. He, H. Fan, Y. Wu, S. Xie, and R. Girshick, "Momentum contrast for unsupervised visual representation learning," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020, pp. 9729–9738.

[8] N. D. Bui, Y. Yu, and L. Jiang, "Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations," in *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2021, pp. 511–521.

[9] Y. Ding, L. Buratti, S. Pujar, A. Morari, B. Ray, and S. Chakraborty, "Contrastive learning for source code with structural and functional properties," *CoRR*, vol. abs/2110.03868, 2021. [Online]. Available: https://arxiv.org/abs/2110.03868

[10] J. Henkel, G. Ramakrishnan, Z. Wang, A. Albarghouthi, S. Jha, and T. Reps, "Semantic robustness of models of source code," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022, pp. 526–537.

[11] N. Yefet, U. Alon, and E. Yahav, "Adversarial examples for models of code," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020.

[12] S. Srikant, S. Liu, T. Mitrovska, S. Chang, Q. Fan, G. Zhang, and U. O'Reilly, "Generating adversarial computer programs using optimized obfuscations," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=PH5PH9ZO_4

[13] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation-tools for software protection," *IEEE Transactions on software engineering*, vol. 28, no. 8, pp. 735–746, 2002.

[14] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM conference on Computer and communications security*, 2003, pp. 290–299.

[15] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, "Protecting software through obfuscation: Can it keep pace with progress in code analysis?" *ACM Computing Surveys (CSUR)*, vol. 49, no. 1, pp. 1–37, 2016.

[16] R. Schuster, C. Song, E. Tromer, and V. Shmatikov, "You autocomplete me: Poisoning vulnerabilities in neural code completion," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[17] I. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *International Conference on Learning Representations*, vol. arXiv preprint arXiv:1412.6572, 2015.

[18] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=rJzIBfZAb

[19] T. Miyato, A. M. Dai, and I. Goodfellow, "Adversarial training methods for semi-supervised text classification," *arXiv preprint arXiv:1605.07725*, 2016.

[20] D. Su, H. Zhang, H. Chen, J. Yi, P.-Y. Chen, and Y. Gao, "Is robustness the cost of accuracy?–a comprehensive study on the robustness of 18 deep image classification models," *arXiv preprint arXiv:1808.01688*, 2018.

[21] D. Tsipras, S. Santurkar, L. Engstrom, A. Turner, and A. Madry, "Robustness may be at odds with accuracy," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=SyxAb30cY7

[22] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," *2018 ICLR*, vol. arXiv preprint arXiv:1706.06083, 2018.

[23] Q. Chen, J. Lacomis, E. J. Schwartz, G. Neubig, B. Vasilescu, and C. L. Goues, "Varclr: Variable semantic representation pre-training via contrastive learning," 2021.

[24] N. D. Q. Bui, Y. Yu, and L. Jiang, *Self-Supervised Contrastive Learning for Code Retrieval and Summarization via Semantic-Preserving Transformations*. New York, NY, USA: Association for Computing Machinery, 2021, p. 511–521. [Online]. Available: https://doi.org/10.1145/3404835.3462840

[25] X. Wang, Y. Wang, F. Mi, P. Zhou, Y. Wan, X. Liu, L. Li, H. Wu, J. Liu, and X. Jiang, "Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation." AAAI, 2022.

[26] L. Fan, S. Liu, P.-Y. Chen, G. Zhang, and C. Gan, "When does contrastive learning preserve adversarial robustness from pretraining to finetuning?" *Advances in Neural Information Processing Systems*, vol. 34, 2021.

[27] Z. Jiang, T. Chen, T. Chen, and Z. Wang, "Robust pre-training by adversarial contrastive learning," *arXiv preprint arXiv:2010.13337*, 2020.

[28] M. Kim, J. Tack, and S. J. Hwang, "Adversarial self-supervised contrastive learning," *arXiv preprint arXiv:2006.07589*, 2020.

[29] S. Gowal, P.-S. Huang, A. van den Oord, T. Mann, and P. Kohli, "Self-supervised adversarial robustness for the low-label, high-data regime," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=bgQek2O63w

[30] K. Wang and M. Christodorescu, "Coset: A benchmark for evaluating neural program embeddings," *arXiv preprint arXiv:1905.11445*, 2019.

[31] E. Quiring, A. Maier, and K. Rieck, "Misleading authorship attribution of source code using adversarial learning," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 479–496.

[32] M. Rabin, R. Islam, and M. A. Alipour, "Evaluation of generalizability of neural program analyzers under semantic-preserving transformations," *arXiv preprint arXiv:2004.07313*, 2020.

[33] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro, "Intriguing properties of adversarial ml attacks in the problem space," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1332–1349.

[34] J. Ebrahimi, A. Rao, D. Lowd, and D. Dou, "Hotflip: White-box adversarial examples for text classification," *arXiv preprint arXiv:1712.06751*, 2017.

[35] P. Bielik and M. Vechev, "Adversarial robustness for code," in *International Conference on Machine Learning*. PMLR, 2020, pp. 896–907.

[36] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 404–419, 2018.

[37] Z. Wu, Y. Xiong, S. X. Yu, and D. Lin, "Unsupervised feature learning via non-parametric instance discrimination," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 3733–3742.

[38] R. Liu, J. Gao, J. Zhang, D. Meng, and Z. Lin, "Investigating bi-level optimization for learning and vision from a unified perspective: A survey and beyond," *arXiv preprint arXiv:2101.11517*, 2021.

[39] Y. Zhang, G. Zhang, P. Khanduri, M. Hong, S. Chang, and S. Liu, "Revisiting and advancing fast adversarial training through the lens of bi-level optimization," in *International Conference on Machine Learning*, 2022, pp. 26 693–26 712.

[40] J. C. Bezdek and R. J. Hathaway, "Convergence of alternating optimization," *Neural, Parallel & Scientific Computations*, vol. 11, no. 4, pp. 351–368, 2003.

[41] C.-Y. Chuang, J. Robinson, L. Yen-Chen, A. Torralba, and S. Jegelka, "Debiased contrastive learning," *arXiv preprint arXiv:2007.00224*, 2020.

[42] F. Wang, H. Liu, D. Guo, and F. Sun, "Unsupervised representation learning by invariancepropagation," *arXiv preprint arXiv:2010.11694*, 2020.

[43] L. Fan, S. Liu, P.-Y. Chen, G. Zhang, and C. Gan, "When does contrastive learning preserve adversarial robustness from pretraining to finetuning?" in *Advances in Neural Information Processing Systems*, A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, Eds., 2021. [Online]. Available: https://openreview.net/forum?id=70kOIgjKhbA

[44] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, jan 2019. [Online]. Available: https://doi.org/10.1145/3290353

[45] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=BJOFETxR-

[46] Y. Wang, K. Wang, F. Gao, and L. Wang, "Learning semantic program embeddings with graph interval neural network," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–27, 2020.

[47] Y. David, U. Alon, and E. Yahav, "Neural reverse engineering of stripped binaries using augmented control flow graphs," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–28, 2020.

[48] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang *et al.*, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *arXiv preprint arXiv:2102.04664*, 2021.

[49] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin, "Detecting code clones with graph neural network and flow-augmented abstract syntax tree," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 261–271.

[50] H. Husain, H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *CoRR*, vol. abs/1909.09436, 2019. [Online]. Available: http://arxiv.org/abs/1909.09436

[51] V. Raychev, P. Bielik, and M. Vechev, "Probabilistic model for code with decision trees," *SIGPLAN Not.*, vol. 51, no. 10, p. 731–747, oct 2016. [Online]. Available: https://doi.org/10.1145/3022671.2984041

[52] H. Liu, M. Long, J. Wang, and M. I. Jordan, "Towards understanding the transferability of deep representations," 2020. [Online]. Available: https://openreview.net/forum?id=BylKL1SKvr

[53] H. Li, Z. Xu, G. Taylor, C. Studer, and T. Goldstein, "Visualizing the loss landscape of neural nets," *Advances in neural information processing systems*, vol. 31, 2018.

[54] J. V. Jeyakumar, J. Noor, Y.-H. Cheng, L. Garcia, and M. Srivastava, "How can i explain this to you? an empirical study of deep neural network explanation methods," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 4211–4222. [Online]. Available: https://proceedings.neurips.cc/paper/2020/file/2c29d89cc56cdb191c60db2f0bae796b-Paper.pdf

[55] B. Kim, R. Khanna, and O. O. Koyejo, "Examples are not enough, learn to criticize! criticism for interpretability," *Advances in neural information processing systems*, vol. 29, 2016.