Metamorphic-Based Many-Objective Distillation of LLMs for Code-related Tasks

Annibale Panichella
Delft University of Technology
Delft, The Netherlands
a.panichella@tudelft.nl

***Abstract*—Knowledge distillation compresses large language models (LLMs) into more compact and efficient versions that achieve similar accuracy on code-related tasks. However, as we demonstrate in this study, compressed models are four times less robust than the original LLMs when evaluated with metamorphic code. They exhibit a 440% higher probability of misclassifying code clones due to minor changes in the code fragment under analysis, such as replacing parameter names with synonyms. To address this issue, we propose MORPH, a novel method that combines metamorphic testing with many-objective optimization for a robust distillation of LLMs for code. MORPH efficiently explores the models' configuration space and generates Pareto-optimal models that effectively balance accuracy, efficiency, and robustness to metamorphic code. Metamorphic testing measures robustness as the number of code fragments for which a model incorrectly makes different predictions between the original and their equivalent metamorphic variants (prediction flips). We evaluate MORPH on two tasks—code clone and vulnerability detection—targeting CodeBERT and GraphCodeBERT for distillation. Our comparison includes MORPH, the state-of-the-art distillation method AVATAR, and the fine-tuned non-distilled LLMs. Compared to AVATAR, MORPH produces compressed models that are (i) 47% more robust, (ii) 25% more efficient (fewer floating-point operations), while maintaining (iii) equal or higher accuracy (up to +6%), and (iv) similar model size.**

***Index Terms*—Knowledge distillation, Large Language Models, Metamorphic testing, Many-objective Optimization, Vulnerability detection, Clone detection, Sustainability, Green-AI**

I. INTRODUCTION

Large Language Models (LLMs) have received broad interest in the software engineering community due to their ability to perform a wide spectrum of tasks, such as code completion [1], [2], clone detection [3], [4], and vulnerability detection [5], [6]. These capabilities stem from the increasing model scale, with millions and billions of parameters [7] and extensive training data spanning thousands of projects [8]. Despite their potential, LLMs frequently face significant computational challenges, such as reliance on dedicated GPU hardware [9], high energy consumption [10], and long inference times (up to seconds per prediction [11], [12]). These factors limit the practical application of LLMs, as IDE creators and developers prefer compact models that give instantaneous feedback within a few tens of milliseconds [12], [13].

Recently, researchers have proposed knowledge distillation methods [14] for compressing large models into smaller, more efficient models. Knowledge distillation transfers knowledge from a large, pre-trained model (the *teacher*) to a small model (the *student*) that keeps approximately the same accuracy on downstream tasks but with significantly lower resource demand. In the context of LLMs for code, Shi et al. [12] have proposed AVATAR, a distillation method that compresses code-specialized BERT-like models from their original size of 500 megabytes (MB) and 125M parameters down to 3MB models with a small/negligible accuracy loss. Relying on evolutionary algorithms, AVATAR has shown superior performance over previously proposed distillation methods [11].

Theoretically, these compressed models are a viable alternative to their teacher LLMs. They are smaller, greener (up to 184× lower energy consumption [12]), and faster (up to 76× faster latency [12]). However, to date, it is unclear whether these compressed models retain the same robustness and properties of their teachers in a broader sense beyond accuracy. This paper aims to shed light on this aspect by investigating the robustness of compressed LLMs against metamorphic code. Metamorphic testing [15] is a well-established technique to assess the robustness of the LLMs models in making predictions against variants of the same code fragments that are semantically and behavioral equivalent. Metamorphic code can reveal model weaknesses [16], [17], implementation bugs [18], data leakage issues [19], and other model vulnerabilities [20]. Therefore, we investigate the following research question:

RQ$_1$: *How robust are the student models distilled with existing state-of-the-art methods to metamorphic code?*

We investigate the robustness of models compressed by AVATAR against natural metamorphic attacks [21], code fragments with methods, and input parameter names replaced with synonyms or expanding acronyms. This process does not modify the code's behavior and captures the variation in naming conventions followed by developers in practice [22]–[24]. We benchmark AVATAR on two downstream code-related tasks — clone detection and vulnerability detection—by compressing CodeBERT and GraphCodeBERT on the Devign [25] and BigCloneBench [26] datasets. While our results confirm the benefits that the compressed methods bring [12], we also find a dramatic reduction in robustness since the compressed models exhibit up to four times (+440%) more mispredictions (or prediction flips) on metamorphic code compared to their teacher models. A prediction flip (PF) occurs when a model makes a different, erroneous prediction on a source code snippet and its corresponding metamorphic variant. For instance, the distilled models may miss a vulnerability because the code contains different (semantically equivalent) parameter names. The significant reduction in robustness raises concerns about deploying compressed models in IDEs. These models must be both accurate and robust across diverse coding scenarios. Non-robust models can compromise the development workflow by not identifying vulnerabilities or code clones.

Given these limitations in existing distillation methods, we propose MORPH. Our novel method integrates metamorphic testing into the distillation process as one of the key objectives in a many-objective problem formulation. Using a state-of-the-art many-objective optimizer, namely AGE-MOEA [27], MORPH generates Pareto-optimal models that minimize the number of prediction flips on metamorphic code while maintaining high accuracy, small model size, and low inference cost. Thus, we investigate the following research questions:

RQ$_2$: *How effective is* MORPH *in distilling robust models compared to the existing state-of-the-art method?*

RQ$_3$: *What is the impact of individual components/features of* MORPH *on its overall performance?*

We empirically evaluated MORPH on clone and vulnerability detection tasks to compress CodeBERT and GraphCodeBERT and compared its performance against the state-of-the-art distillation method AVATAR. The results show that MORPH generates models that are 47% more robust (measured in PFs), 25% more efficient (measured by FLOPs), with a comparable or superior accuracy (up to +6%), while maintaining the model size of 3MB. An ablation study confirms that all individual components of MORPH contribute to its overall performance, with metamorphic testing and many-objective optimization being particularly critical. AGE-MOEA, the core optimizer in MORPH, outperforms alternative optimizers in generating high-quality Pareto-optimal distilled models.

Our contributions can be summarized as follows:

- We identify weaknesses in the robustness of existing compression techniques against metamorphic code.
- We propose MORPH, a novel approach that integrates metamorphic testing with many-objective optimization to generate robust compressed models.
- We empirically evaluate MORPH on multiple tasks and LLMs; our results demonstrate its effectiveness in producing robust, accurate, and efficient models.
- We provide a complete replication package [28], including the metamorphic datasets, the implementation of MORPH, and the raw data of our experiments.

II. BACKGROUND AND RELATED WORK

Knowledge Distillation. Knowledge distillation [14] aims to compress a large pre-trained model (called the *teacher*) into a smaller and more efficient model (called the *student*). The student model S is taught how to predict similar responses on a given dataset as made by the teacher M by minimizing the difference between the teacher's and student's output distributions [14], [29]. In task-specific distillation, the student model learns from an already fine-tuned teacher model for a specific task [11], [30], which means that S can be directly used for the same downstream task without any fine-tuning.

Finding the optimal architecture for the student model remains challenging. Common hyperparameters, such as the number of layers, hidden units, and attention heads, need to be tuned to achieve the desired performance of the student model. COMPRESSOR by Shi et al. [11] employs a single-objective genetic algorithm for compressing code-specific models such as CodeBERT into 3MB with approximately the same accuracy levels. AVATAR [12] extends this work by using a multi-objective genetic algorithm to balance accuracy and efficiency when compressing models into 3MB. Since AVATAR has been shown to outperform COMPRESSOR, we refer to it as the state-of-the-art distillation method for code-related tasks.

Despite these advancements, models compressed using AVATAR show reduced robustness to code variations (as we demonstrate in Section V-A). To address this, we introduce robustness as a fourth objective in the distillation process, alongside accuracy, size, and efficiency. This addition requires using a many-objective optimization algorithm (e.g., AGE-MOEA [27]) rather than a multi-objective one used in prior studies [12]. Besides, we leverage metamorphic testing to measure the robustness of the student models against metamorphic code changes and build surrogate models to predict the model robustness without training it.

Metamorphic Testing. Metamorphic testing for software systems was first introduced by Chen et al. [15] as a technique to generate new test cases by applying *transformations* to the input data while preserving the expected test output (oracle). Recently, metamorphic testing has gained traction in assessing the robustness of machine learning models against subtle input variations (e.g., pixel changes for image classification [31]). Metamorphic testing has been used in software engineering to evaluate the robustness of LLMs of code against subtle code variations (or *metamorphic transformations*). Compton et al. [17] demonstrated how Code2Vec models are vulnerable to obfuscation techniques for variable names. Yefet et al. [32] introduced metamorphic transformations that add unused variables to existing code, causing the model to misclassify vulnerable code. Other metamorphic changes include renaming input parameter names, adding lambda expressions, etc. All these transformations do not alter the code behavior but can lead to incorrect model predictions [16], [17], [33]. BERT-like models, such as CodeBERT and GraphCodeBERT, have also been tested using metamorphic testing and shown to be vulnerable to metamorphic code changes [16], [21], [33].

There are two main strategies to generate metamorphic code snippets: applying one change at a time (*first-order*) or applying multiple changes at once (*higher-order*). The latter is more effective in fooling LLMs and can be applied either by sampling multiple *first-order* changes or using meta-heuristics to select the most effective adversarial changes [21], [34]. As highlighted by Yang et al. [21], the naturalness of the metamorphic code is crucial to ensure that the changes are realistic and preserve the natural semantics of code (e.g., variable names) in addition to its behavior and syntax.

This work focuses on the robustness of models compressed with knowledge distillation against metamorphic changes. Our goal is two-fold: (1) to measure the robustness of models compressed with existing distillation techniques and (2) to develop a new distillation method that explicitly relies on metamorphic testing as one of its core components.

Many-objective Optimization. Many-objective optimization refers to problems involving more than three conflicting objectives [35]. In these problems, identifying a single optimal solution is infeasible due to the inherent trade-offs between the objectives. Therefore, the goal is to find solutions representing the optimal trade-offs, known as the *Pareto front* [36]. The Pareto front comprises solutions where no improvement can be made in one objective without degrading another (*Pareto optimality*). A key challenge is the exponential growth of the search space as the number of objectives increases, making it difficult to find the Pareto front efficiently [35], [37].

Many-Objective Evolutionary Algorithms (MOEAs) have emerged as powerful tools for addressing such problems [35], [36], [38]. MOEAs iteratively evolve an initial population of solutions through subsequent generations. The aim to approximate the Pareto front by balancing two aspects: (1) converging towards the true Pareto front and (2) maintaining diversity among the solutions to provide decision-makers with a range of options to choose from [36], [39] Over the last decades, several MOEAs have been proposed, including NSGA-II [39], NSGA-III [37], MOEA/D [40], and AGE-MOEA [27]. They differ mainly in the mechanisms they rely on to select new solutions for the next generations (*environmental selection*). This paper leverages AGE-MOEA for its adaptability to diverse Pareto front shapes and demonstrated superior performance in prior studies [27], [41], [42]. Its environmental selection mechanism employs *non-dominated sorting* [39] and assigns a *survival score* that measures both convergence and diversity based on the (estimated) shape of the Pareto front. This promotes solutions near and well-distributed along the Pareto front [27]. A comparative evaluation of AGE-MOEA with other MOEAs is provided in Section V-C.

III. Proposed Methodology

Ensuring the robustness of compressed models is crucial, as non-robust models can fail in real-world applications. For instance, a model unable to detect a vulnerability in code with slightly different variable names is highly problematic given the prevalent use of inconsistent identifier names and vocabulary in existing codebases [22]–[24]. Hence, we formulate the robust knowledge distillation problem as follows:

Definition: Given a pre-trained LLM (teacher model), the problem is to find a configuration $C = [c_1, \dots, c_m]$ for the student model S that optimizes the following objectives:

$$\begin{cases} \min O_1 = & \text{size}(S) \\ \min O_2 = & \mid S(v) \neq S(\overline{v}) \mid, \forall v \in V \text{ and } \overline{v} = meta(v) \\ \min O_3 = & \text{efficiency}(C) \\ \max O_4 = & \text{effectiveness}(C, V) \end{cases} \quad (1)$$

where V is the validation set, and each code snippet $v \in V$ has a corresponding metamorphic $\overline{v} = meta(v)$. In this formulation, O_1 represents the model size measured as the number of its internal parameters. O_2 measures the model robustness as the number of data points in V where the model S changes/flips its prediction between original and metamorphic code snippets, i.e., when $S(v) = S(\overline{v})$. O_3 measures the model's inference cost in terms of floating-point operations (FLOPs) as done in previous research [10], [12]; lower FLOP values indicate faster inference and reduced energy consumption. Lastly, depending on the specific downstream task, O_4 measures the model's performance on the validation set, such as accuracy or F1-score.

Research Challenges: Optimizing the robust knowledge distillation problem poses several challenges:

Challenge 1: The problem is inherently many objective (i.e., with 4 objectives), which is difficult to solve due to the increasing number of trade-offs as the number of objectives increases [27], [37]. Many traditional evolutionary algorithms, such as the one used in [12], struggle to scale beyond three objectives, requiring specialized many-objective algorithms.

Challenge 2: Training student models multiple times with different configurations to compute accuracy and prediction flips is computationally expensive, if not prohibitive.

Challenge 3: The search space for student model configurations is vast, with numerous hyperparameters and potential combinations. Effective sampling strategies are crucial to identifying promising configurations with few samples.

Challenge 4: Defining and generating natural metamorphic attacks is crucial to simulate real-world variations in code, ensuring these transformations preserve code semantics while mimicking natural coding conventions [22]–[24].

Challenge 5: Generating valid configurations requires adhering to the internal student model constraints, such as maintaining consistency between *hidden size* and the *number of attention heads*. This is critical to avoid invalid or suboptimal models during the search process.

Overview. To tackle the challeges above, we propose MORPH, a novel approach that integrates metamorphic testing with recent advances in many-objective optimization. As shown in Figure 1, MORPH comprises three core stages: (1) generating metamorphic datasets, (2) constructing surrogate models for fast objective calculation, and (3) employing a many-objective algorithm. By combining these macro-stages and leveraging the strengths of each, MORPH efficiently explores the configuration space and finds a set of Pareto-optimal configurations for the student model. The following sections detail each stage, their interactions, and how they address the challenges outlined above.

A. Generating Metamorphic Code Snippets

To generate metamorphic datasets, we modify the code snippets in the original dataset by introducing subtle changes that *do not* alter their behavior or semantics. Since this paper focuses on vulnerability and clone detection, we rely on two metamorphic transformations [33]: (1) *method renaming* and (2) *input parameter renaming*. We choose these transformations as a code snippet remains vulnerable regardless of how developers name the functions or input parameters. To generate *natural* [43], [44] (i.e., human-like written) code, we avoid

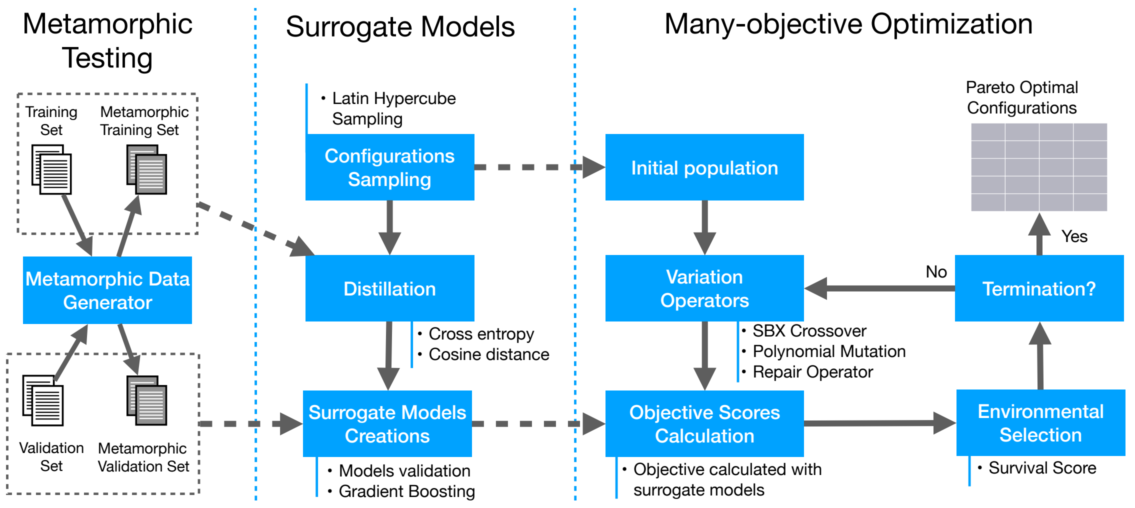

Fig. 1. Overview of MORPH, which encompasses three macro-stages: (1) generating metamorphic datasets (on the left side), (2) constructing two surrogate models (middle stage), and (3) employing many-objective optimization (on the right side).

```
Prompt:
"Here is a [LANG] function: ``[CODE]``
Change the function name (and input parameters names)
     ↪ with equivalent ones or synonyms in English.
     ↪ Just return the changed code (the complete new
     ↪ code) without any extra text."
```

Listing 1. Prompt for obtaining module name and version

using obfuscation techniques that rely on randomly generated strings, as done in prior work [17], [32]. Instead, we use `ChatGPT-3.5` to replace English words with synonyms and expand acronyms to their full forms; for example, `args` is expanded to `arguments`. Synonyms are common in daily development, as previous studies show that developers do not use consistent vocabularies, frequently opting for synonyms and acronyms [22]–[24]. By focusing on these natural transformations, our approach directly addresses *Challenge 4* since it generates transformed code that mirrors the real-world variability in naming conventions employed by developers.

Listing 1 displays the prompt we used to generate metamorphic code snippets. In the listing, `[LANG]` indicates the programming language (e.g., `Java`), and `[CODE]` is the original code snippet. An example of a metamorphic code generated by `ChatGPT-3.5` is shown in Figure 2. The figure shows both the original code snippet (on the left) from the Devign dataset [25] and its metamorphic variant (on the right). Both code snippets exhibit a CWE-200 vulnerability.

As a result, our methodology employs higher-order metamorphic transformation, in which all input parameter names and main function names are changed at once. This generates only one metamorphic variant for each original code snippet. We further validated that the metamorphic code fragments are well-formed by using source code parsers generated with `tree-sitter` [45]. Tree-sitter offers a comprehensive set of grammars for various programming languages, allowing us to check that the generated code adheres to the grammar rules of the corresponding programming language. We manually inspect a representative subset of generated snippets, as detailed in Section IV, to check the semantic equivalence to the original code. This two-step validation process guarantees the naturalness and correctness of the metamorphic datasets.

B. Building the Surrogate Models

Searching for Pareto-optimal model configurations requires training the models with the given configurations and measuring their effectiveness on the validation set. However, training and evaluating hundreds of models is computationally expensive (*Challenge 2* mentioned above). To address this, we use surrogate models, which provide a computationally efficient prediction/estimation of a model's performance given its configuration as input. Surrogate models are a well-established method to tackle problems with expensive fitness functions in evolutionary computation [46], [47], and are widely used in software testing for costly tests [38] and in knowledge distillation [12]. We defined two surrogate models that are custom and specific to our problem: one for predicting the accuracy of the student model and the other for predicting the number of prediction flips on metamorphic code.

To build these surrogates (the second stage in Figure 1), MORPH first samples the configuration space using Latin Hypercube Sampling (LHS) [48], [49]. Then, it applies knowledge distillation techniques to train the student model with each sampled configuration. Finally, the sampled configurations (*features*) and their corresponding performance metrics (*targets*) are used to train the surrogate models.

1) Latin Hypercube Sampling: MORPH employs *Latin Hypercube Sampling* (LHS) [48], [49] to sample a diverse pool of configurations. LHS is a stratified sampling method that ensures uniform coverage of the configuration space by dividing each decision dimension (e.g., vocabulary size) into equal intervals (*strata*) and selecting one value per interval for each dimension. This process generates a set of sample points that are uniformly distributed across the entire search space and represent all configuration intervals, thus providing a diverse set of configurations for training the surrogate models. By efficiently exploring the vast configuration space, LHS

```c
void qemu_spice_display_init_common(
                        SimpleSpiceDisplay *ssd) {
   qemu_mutex_init(&ssd->lock);
   QTAILQ_INIT(&ssd->updates);
   ssd->mouse_x = -1;
   ssd->mouse_y = -1;
   if (ssd->num_surfaces == 0) {
      ssd->num_surfaces = 1024;
   }
   ssd->bufsize = (16 * 1024 * 1024);
   ssd->buf = g_malloc(ssd->bufsize);
}
```

```c
void initialize_spice_display_common(
                    SimpleSpiceDisplay *display){
   qemu_mutex_init(&display->lock);
   QTAILQ_INIT(&display->updates);
   display->mouse_x = -1;
   display->mouse_y = -1;
   if (display->num_surfaces == 0) {
      display->num_surfaces = 1024;
   }
   display->bufsize = (16 * 1024 * 1024);
   display->buf = g_malloc(display->bufsize);
}
```

Fig. 2. Original code snippet (idx=11939) from the Devigndataset [25] (left) and its metamorphic variant (right) generated by `ChatGPT-.5`. Both code snippets exhibit a CWE-200 vulnerability.

addresses *Challenge 3*, ensuring a diverse and representative sample of configurations.

2) Knowledge Distillation: To transfer knowledge to the compact student model S with a given configuration C, we employ a hybrid loss function comprising a distillation loss and a cosine similarity loss: $L = L_{dist} + L_{cos}$. Both functions consider the augmented training set that includes the original and metamorphic training data (see Figure 1). The distillation loss L_{dist} is calculated as the cross-entropy between the teacher's softmax output and the student's log-softmax output:

$$L_{dist} = -\sum_{i=1}^{N} softmax\left(\frac{y_i}{T}\right) \times \log\left(softmax\left(\frac{s_i}{T}\right)\right) \times T^2 \quad (2)$$

where y_i and s_i are the teacher's and student's logits, respectively; T is the temperature scaling factor [14]; N is the number of data points in the training set. The temperature T controls the softness of the teacher's predictions, with higher values leading to softer predictions.

The cosine similarity loss L_{cos} ensures that the student model's predictions are similar to the teacher model's predictions in terms of direction, not just magnitude. The cosine distance between the teacher's and student's predictions is calculated as follows:

$$L_{cos} = 1 - \frac{\sum_{i=1}^{N} softmax(y_i) \times softmax(s_i)}{\sqrt{\sum_{i=1}^{N} softmax(y_i)^2} \times \sqrt{\sum_{i=1}^{N} softmax(s_i)^2}} \quad (3)$$

3) Training the Surrogate Models: MORPH builds two surrogate models: one for predicting accuracy and the other for predicting the number of prediction flips on the validation set. For both performance indicators, we use the *Gradient Boosting Regression* (GBR) model [50]. GBR is an *ensemble method* that creates multiple weak models (regression trees) and combines them to achieve a strong combined model. We chose this model because it effectively handles both numerical and categorical features present in our problem. In our preliminary experiments, GBR outperformed other models in terms of mean squared error (smaller fitting error).

For each configuration, we assess (1) its accuracy on the validation set and (2) the number of prediction flips the model makes between the original code fragments and their metamorphic variants. We use these sampled configurations (input features) and their corresponding outcome values (prediction targets) to train the two regression models that serve as our surrogate models. *The surrogate models have been trained on the validation datasets, which share similar distributions with the test datasets but remain separate and distinct. Thus, the test sets have not been used to train the surrogate models.*

We use the GBR model for accuracy but apply pre- and post-processing so that the predictions are within the range $[0, 1]$. First, we apply the logit function [51], $logit(p) = \log(p/(1-p))$, to the accuracy values before fitting the model. This function converts the accuracy values to an unbounded scale, making them more suitable for regression modeling. During prediction, the model outputs logit-transformed accuracy values, which are then converted back to the 0 to 1 range using the logistic sigmoid function [51], $expit(x) = 1/(1 + exp(-x))$. For the number of prediction flips, the expected values are $\in [0, |V|]$, where $|V|$ is the size of the validation set. Therefore, any negative prediction is converted to zero.

C. Many-Objective Optimization with AGE-MOEA

To address *Challenge 1*, MORPH uses the state-of-the-art MOEA, Adaptive Geometry Estimation-based Multi-Objective Evolutionary Algorithm (AGE-MOEA) [27]. AGE-MOEA extends the framework of traditional multi-objective evolutionary algorithms (MOEAs) by incorporating advanced techniques to estimate the geometry of the Pareto front. MORPH customizes AGE-MOEA to handle the four objectives and the specific constraints of knowledge distillation: (1) custom initialization with LHS, (2) use of surrogate models for calculation of the objective scores, and (3) use of a repair operator.

As shown in Figure 1, MORPH iteratively evolves an initial set of configurations. At each iteration, the algorithm applies variation operators (*crossover*, *mutation*, and *repair*) to generate new configurations (offspring). Parent and offspring configurations are evaluated using the surrogate models, which predict the performance of the student models on the validation set without training it. The objective scores are used to select the best configurations that will survive in the subsequent iteration (*environmental selection*). This process is repeated until the total number of configuration/solution evaluations is reached. MORPH outputs Pareto-optimal configurations representing the trade-offs between the four objectives.

1) Search Initialization: Population initialization is the first step in the many-objective optimization process. The initial population should include a diverse randomly-sampled pool of model configurations [52]. MORPH uses Latin Hypercube Sampling (LHS) to create the initial population, i.e., the same method used for the surrogate models (Section III-B1).

2) Repair Operator: MORPH employs three operators to generate offspring solutions: *crossover*, *mutation*, and *repair*. For crossover and mutation, we use the *simulated binary crossover* (SBX) [53], [54] and *polynomial mutation* (PM) [36]. While SBX and PM are standard operators in many-objective optimization [27], the repair operator is a custom operator designed for the distillation problem in MORPH to ensure that the offspring configurations are valid and meet the constraints of the student models (*Challenge 5*). For example, a configuration with a hidden size not divisible by the number of attention heads is invalid. Besides, some genes (e.g., vocabulary size) must be integers, while others (e.g., dropout probabilities) must be within a specific range. To address this challenge, we developed a *repair operator* that fixes and repairs all invalid configurations. For each child configuration, the repair function adjusts the genes as follows: (1) Integer configuration values are rounded to the nearest integer; (2) Probability values (e.g., *hidden dropout probability*) are rounded to one decimal place; (3) The *hidden size* must be divisible by the *number of attention heads*. If this is not the case after crossover and mutation, the repair operator adjusts the *hidden size* to the nearest multiple of the *number of attention heads* using Equation 4. This process involves identifying the nearest multiples of the number of attention heads. Specifically, the repair operator calculates the next higher multiple (A in Equation 4), and the previous lower multiple (B in Equation 4). Then, it compares these multiples to the original hidden size and chooses the closest one. Let h denote the hidden size and a the number of attention heads; the repair operator calculates the new fixed hidden size h^* as:

$$A = \left(\left\lfloor \frac{h}{\mathrm{a}} \right\rfloor + 1\right) a,\ B = \left\lfloor \frac{h}{a} \right\rfloor a,\ h^* = \begin{cases} A & \text{if } |h - A| < |h - B| \\ B & \text{otherwise} \end{cases} \quad (4)$$

3) Objective Score calculation: The new configurations are evaluated after applying the variation operators, and their objective scores are calculated. For the accuracy (O_4) on the validation set and the number of prediction flips on the metamorphic validation set (O_2), MORPH uses the surrogate models instead of training the student models from scratch (see Section III-B3). W.r.t. O_1, the model size is calculated using the formula from prior studies [11], [12]. This formula decomposes a student model into three components: (1) the embedding layer, (2) the transformer layer, and (3) the classifier layer. The formula determines the total model size in MB by summing the sizes of these three components. Finally, the FLOPs (O_3) are calculated using the same methodology used in prior studies on knowledge distillation [11], [12].

IV. EMPIRICAL EVALUATION

The *goal* of this study is to evaluate the effectiveness of MORPH and comparing it with existing model distillation approaches. We investigate the following research questions:

- **RQ$_1$**: *How robust are the student models distilled with existing state-of-the-art methods to metamorphic code?*
- **RQ$_2$**: *How effective is* MORPH *in distilling robust models compared to the existing state-of-the-art method?*
- **RQ$_3$**: *What is the impact of individual components/features of* MORPH *on its overall performance?*

RQ$_1$ serves as a preliminary investigation to assess the robustness of models distilled using the existing state-of-the-art method, AVATAR [12], against metamorphic code. This RQ is crucial as it helps identify potential weaknesses in existing distillation approaches, particularly their robustness to metamorphic code. If the results indicate that these methods are less robust than the non-compressed versions of the models, it will highlight the necessity for a more robust distillation approach, thus motivating our approach, MORPH.

Our second research question (**RQ$_2$**) directly evaluates the effectiveness of MORPH in distilling robust models, especially in comparison to AVATAR [12]. This question aims to determine whether MORPH can produce student models that are more Pareto optimal than those distilled using AVATAR, also considering the robustness against metamorphic code.

Finally, **RQ$_3$** is addressed through an ablation study that investigates the contribution of each key component of MORPH to its overall performance. This includes evaluating the impact of (1) data augmentation, (2) LHS for population initialization, (3) repair operator, (4) surrogate models, and (5) the employed many-objective optimization algorithm. For the latter, we compare AGE-MOEA, the default optimizer in MORPH, against alternative many-objective optimizers, namely NSGA-II [39], NSGA-III [37], and MOEA/D [40], w.r.t. the Pareto fronts quality. By systematically removing each component and analyzing the results, we assess their relevance. Comparing the use of AGE-MOEA with other optimizers allows us to validate its suitability and explore whether alternative algorithms might offer additional benefits or improved Pareto front quality.

Downstream Tasks. To align with previous research [11], [12] in knowledge distillation, we evaluate MORPH and the baselines (AVATAR and the original LLMs) on two prominent software engineering tasks: (1) vulnerability prediction and (2) clone detection. Table I presents an overview of the datasets used in our experiments. For a fair and rigorous comparison with existing knowledge distillation methods, we adopt the identical dataset splits and experimental settings employed in prior studies [11], [12] for both tasks. This methodological consistency enables a direct replication of previous results and facilitates a precise evaluation of MORPH's performance. The first task is *vulnerability prediction*, which involves identifying security vulnerabilities in source code. We use the Devign dataset [25], which contains code snippets written in the C programming language and mined from two open-source libraries, `FFmpeg` and `Qemu` [25]. The second task is *clone detection*, which aims to identify code duplications. We utilize the BigCloneBench dataset [26], an extensive collection of Java code clones collected by mining 25,000 Java projects from `SourceForge` and the `Google Code` platform. We have chosen these two datasets to mitigate potential *data leakage* issues, as we further elaborate in Section VII.

Metamorphic datasets: To augment the original datasets, we generate metamorphic code snippets using the methodology outlined in Section III-A. We apply function/method renaming

TABLE I
OVERVIEW OF DATASETS USED FOR THE EXPERIMENTS, INCLUDING TRAINING, VALIDATION, AND TEST SET SIZES.

Task	**Source**	**Dataset Size**	**Language**

Vulnerability Prediction	Devign [25]	Training = 10,927
		Validation = 2732
		Test = 2732
			`C`

Clone Detection	BigClone-Bench [26]	Training = 45,051
		Validation = 4000
		Test = 4000
			`Java`

and input parameter renaming to the vulnerability prediction and clone detection datasets, creating metamorphic versions for the training, validation, and test sets. To validate the correctness of the generated metamorphic code, we conduct manual inspections to ensure that CWE labels remain consistent after transformations for vulnerability prediction. These metamorphic datasets are used to evaluate the robustness of the student models distilled with MORPH and the baselines against adversarial code perturbations. Notably, the test sets and their metamorphic versions are not used in any of the internal steps of MORPH (e.g., surrogate model training and many-objective optimization). This separation guarantees that the test set remains unseen during the search process.

Large Language Models. We employ two pre-trained LLMs specialized for code-related tasks, namely CodeBERT [55] and GraphCodeBERT [56]. CodeBERT is a bimodal pre-trained model for natural language and programming languages. It is based on the BERT architecture and trained on a large corpus of code from multiple programming languages, including Python, Java, JavaScript, and more. CodeBERT is designed to understand and generate code snippets, making it useful for various tasks such as code search, completion, and documentation generation. GraphCodeBERT extends the capabilities of CodeBERT by incorporating structural information from code [56]. In addition to the textual data, GraphCodeBERT leverages the data flow graph of code snippets to better capture the relationships between different parts of the code. We selected these LLMs because they are (1) open source with fully traceable pre-training[1], (2) used in prior distillation studies [12], and (3) previously fine-tuned on our tasks, allowing us to replicate and compare results directly.

Fine-tuning: We fine-tuned the hyperparameters of CodeBERT and GraphCodeBERT for the two downstream tasks described above. Fine-tuning was performed using the configurations recommended by the original studies [8], [56], achieving results comparable to those reported in the related literature for the two downstream tasks [8], [11], [12], [56].

Empirical Methodology. To answer **RQ$_1$**, we evaluate the robustness of the models distilled with AVATAR and the non-distilled fine-tuned LLMs. The goal is to determine whether compressed models are less, more, or as robust as the original LLMs. To this aim, we measure the robustness of a model M (either teacher or student model) by counting the number of instances where M produces different predictions for an original code snippet t and its corresponding metamorphic version $\bar{t}$. Let $T = \{t_1, t_2, \ldots, t_n\}$ be the original test set, and let $\overline{T} = \{\overline{t_1}, \overline{t_2}, \ldots, \overline{t_n}\}$ be the metamorphic test set, such that $\overline{t_i}$ is the metamorphic version of t_i. The robustness of M is measured by the number of prediction flips:

$$\text{Prediction Flips} = \left|\left\{(t_i, \overline{t_i}) \in T \times \overline{T} \ : \ M(t_i) \neq M(\overline{t_i})\right\}\right| \quad (5)$$

For this analysis, we use the model checkpoints available in the replication package[2] of AVATAR (model of 3MB in size) and compare them to the original fine-tuned LLMs.

To answer **RQ$_2$**, we compare the models distilled by MORPH with those distilled by AVATAR. The comparison is made with regard to four metrics: (1) *accuracy* of the models on the test sets, (2) *model size* in MB, (3) number of *prediction flips* on the test sets (robustness), and (4) the number of *Giga FLOPs* (model efficiency). For this experiment, we did not use the model checkpoints available in the replication package of AVATAR but trained/compressed the models from scratch using the original implementation of AVATAR. The number of prediction flips is calculated using Equation 5 as done for **RQ$_1$**. We ran AVATAR and MORPH ten times to account for their randomness. We report the median results of these approaches across the runs for all four metrics mentioned above, alongside the interquartile range (IQR). Since MORPH produces a Pareto front and not a single solution, we select the Pareto-optimal compressed model with the model size closest to 3MB, which is the model size of the model produced by AVATAR [12]. We analyze the results using the Wilcoxon signed-rank test [57] (p-value$\leq$0.05) for the significance and the Vargha-Delaney $\hat{A}_{12}$ statistics [58] for the effect size.

To answer **RQ$_3$**, we perform an ablation study to evaluate the contribution of each key component of MORPH. For each ablation, we measure the resulting models' accuracy and robustness (number of prediction flips) to determine how the removal of these components affects MORPH's performance. Regarding the many-objective optimization algorithm, we compare the results of MORPH when using AGE-MOEA (see Section III) and other alternative algorithms, namely NSGA-II [39], NSGA-III [37], and MOEA/D [40]. To compare the MOEAs, we employed the hypervolume (HV) quality indicator [59] to assess the quality of the Pareto fronts. HV takes values in $[0;1]$ and quantifies the volume of the objective space dominated by a Pareto front. For calculating the HV scores, we determine the nadir (reference) point Z^{max} corresponding to the point having as coordinates the maximum value for each objective across all Pareto fronts generated by all MOEAs in all experimental runs [60]. A higher HV value indicates a Pareto front that covers a larger portion of the objective space, suggesting a better trade-off between the optimization objectives. We run each MOEA ten times and compare their median (and IQR) HV scores also using the Wilcoxon signed-rank test (p-value$\leq$0.05) and Vargha-Delaney $\hat{A}_{12}$ statistics.

Parameter Settings. We set MORPH to evolve a population of 50 model configurations over 100 generations. MORPH uses parameter values recommended in the literature: SBX

[1]https://huggingface.co/datasets/code-search-net/code_search_net

[2]https://github.com/soarsmu/Avatar

crossover with distribution index η_c=30, and PM mutation with distribution index η_m=30, recommended values for multi- and many-objective problems [27], [36], [61]. The crossover probability is 0.90, within the recommended interval of $[0.45, 0.95]$ [36] and a mutation probability of $1/l$, where l is the number of configuration values [36]. Parent configurations are selected for reproduction using the *binary tournament selection* [27], [36]. Our repair operator is always applied after crossover and mutation to fix the solutions.

MORPH employs GBR to construct surrogate models (see Section III-B3). To optimize GBR hyperparameters, we conducted a grid search across a predefined parameter space: (1) the number of decision trees $\in \{50, 100, 150\}$; (2) the learning rate $\in \{0.05, 0.10, 0.15, 0.20, 0.25\}$; (3) the maximum depth of each decision tree $\in \{2, 3, 4, 5\}$; (4) the minimum number of samples in internal node $\in \{2, 3, 4\}$; (5) the minimum number of samples in leaf nodes $\in \{1, 2, 3\}$. We employed a threefold cross-validation strategy to fit the GBR model and selected the hyperparameter values, achieving the lowest mean absolute error across the three folds. We ensured the test set remained unseen when training the surrogate models to prevent overfitting and maintain evaluation integrity.

Both AVATAR and MORPH are set with the same configuration space for the distillation: (1) tokenizers $\in$ {BPE, WordPiece, Unigram, Word}, (2) vocabulary size $\in [1000, 46000]$, (3) number of hidden layers $\in [1, 12]$, (4) hidden size $\in [16, 256]$, (5) hidden act $\in$ { gelu, relu, silu, gelu_new }, (6) hidden dropout probability $\in [0.2, 0.5]$, (7) intermediate size $\in [32, 3072]$, (8) number of attention heads $\in [1, 12]$ (9) attention dropout probability $\in [0.2, 0.5]$, (10) maximum sequence length $\in [256, 512]$, (11) position embedding type $\in$ {absolute, relative key, relative key query}, (12) learning rate $\in$ {0.001, 0.0001, 0.00001}, (13) batch size $\in$ {16, 32, 64 }. Both approaches use the temperature scaling factor T=3, as recommended [14]. Finally, AVATAR is set with the default parameter values recommended in the original study [12]. We increase the population size to 50 and the number of generations to 100 for a fair comparison with MORPH.

Implementation details. We implemented MORPH in `Pymoo` v0.6.0 [62], using Python 3.9 and Pytorch v1.10.0. `Pymoo` is an open-source framework that implements the MOEAs used in this study (including AGE-MOEA). We use the libraries Transformers v4.21.1, Scikit-learn v1.1.2, and the implementation of LHS available in `pyDOE` v0.3.8. The experiments were conducted on a server with an AMD EPYC 7713 64-Core Processor running at 2.6 GHz and 256 CPUs available. We used 3 Nvidia A40 GPUs, each with 48 GB GDDR6 running CUDA version 11.6. The complete replication package, including the datasets, the source code, and results, is available on `Zenodo` [28].

V. RESULTS

A. RQ1: Robustness of Existing Distillation Methods

Table II reports the number of prediction flips for the original LLMs and the student models distilled with AVATAR by Shi et al. [12] on the two tasks. The results reveal a significant increase in prediction flips for student models generated by AVATAR compared to teacher models. Specifically, on the clone detection task, the number of prediction flips surges by 440% and 155% for CodeBERT and GraphCodeBERT, respectively. While the vulnerability prediction task exhibits a less pronounced increase of 11% for both LLMs, the overall trend indicates a decline in model robustness.

TABLE II
COMPARISON OF THE NUMBER OF PREDICTION FLIPS BETWEEN THE ORIGINAL LLMS AND AVATAR ON THE TWO TASKS.

Model Name	Task	Distillation	Size (MB)	# Pred. Flips
CodeBERT	Clone Detection	None	499	32
		AVATAR	3	172 (+440%)
	Vulnerability Prediction	None	499	406
		AVATAR	3	449 (+11%)
GCodeBERT	Clone Detection	None	499	45
		AVATAR	3	115 (+155%)
	Vulnerability Prediction	None	499	399
		AVATAR	3	438 (+11%)

A prediction flip happens when a model generates different outcomes for an initial code snippet and its corresponding, semantically equal, metamorphic variation. This phenomenon reveals the model's vulnerability to minor variable renaming (using synonyms) and suggests that its decision-making process may be unstable. Since it is a standard practice for developers to use a variety of often inconsistent identifiers [22]–[24], this noticeable rice in prediction flips casts doubts on the reliability and trustworthiness of these compressed models for real-world deployment. To replace the original LLMs, the compressed models should be at least as robust as the original models to achieve similar accuracy with fewer resources.

These findings underscore the necessity for a more robust distillation approach, such as MORPH, which explicitly addresses the challenge of preserving model stability.

B. RQ2: Comparison between MORPH and AVATAR

Both MORPH and AVATAR produce multiple student models with different trade-offs between size and accuracy (MORPH also considers model robustness as an additional objective). For both approaches, we select the model configuration with a size closest to 3MB for training the optimized model, which is the target size used in the literature for code-related tasks [11], [12]. 3MB is also the size of the models used in IDE components or editor plugins [12], [13]. Table III presents the results of MORPH and AVATAR on the two downstream tasks: median accuracy, model size, number of prediction flips, and Giga FLOPs for the two distillation approaches over ten runs. The interquartile range (IQR) is shown in parentheses and measures the metric variability across the runs.

Accuracy. MORPH outperforms AVATAR in all metrics for both tasks while keeping a similar model size of 3MB. For the clone detection task, MORPH achieves a median accuracy of 96.43% for CodeBERT and 94.97% for GraphCodeBERT, compared to 89.55% and 88.88% for AVATAR, respectively. It is worth noting that AVATAR's accuracy is slightly lower than the one reported by the original paper [12]. In our evaluation,

TABLE III

RESULTS OF MORPH AND AVATAR ON THE TWO DOWNSTREAM TASKS. WE REPORT THE MEDIAN RESULTS, WITH THE INTERQUARTILE RANGE (IQR) SHOWN IN PARENTHESES. THE BEST RESULT FOR EACH METRIC IS HIGHLIGHTED IN GRAY.

Model Name	Task	Distillation Approach	Accuracy (%)		Size (in MB)		# Pred. Flips		Giga FLOPs	
			Median	Diff.	Median	Diff.	Median	Diff.	Median	Diff.
CodeBERT	Clone Detection	Avatar	89.55 (6.69)	+6.88%	3.00 (0.01)	-0.06	129.50 (70.75)	-73%	1.45 (0.30)	-8%
		Morph	96.43 (1.49)		2.94 (1.58)		34.50 (15.24)		1.33 (1.00)	
	Vulnerability Prediction	Avatar	59.55 (1.02)	+0.02%	3.01 (0.02)	0.00	444.00 (14.75)	-27%	0.89 (0.37)	-19%
		Morph	59.57 (0.99)		3.01 (0.13)		326.00 (37.00)		0.72 (0.26)	
GCodeBERT	Clone Detection	Avatar	88.88 (1.75)	+6.09%	3.00 (0.01)	+0.01	123.00 (101.00)	-57%	1.77 (0.00)	-37%
		Morph	94.97 (1.66)		3.01 (0.06)		53.00(6.25)		1.12 (0.17)	
	Vulnerability Prediction	Avatar	59.04 (1.02)	-0.27%	3.01 (0.01)	+0.02	423.00 (74.00)	-33%	0.74 (0.42)	-38%
		Morph	58.77 (0.18)		3.03 (0.03)		284.00 (18.00)		0.46 (0.20)	
Avarage (mean) Difference				+3.18%		-0.01		-47%		-25%

TABLE IV

RESULTS OF THE TEACHER MODELS FINE-TUNED USING THE ORIGNAL TRAINING SET (NO DIST.) AND THE TRAINING SET AUGMENTED WITH METAMORPHIC CODE (+ DATA AUG.). FOR MORPH, WE REPORT THE RESULTS OF THE MODELS WITH A SIZE CLOSEST TO 3MB AND 6MB.

Model	Task	Distillation	Acc (%)	#P. Flips
CodeBERT	Clone Detection	No Dist.	96.10	32.00
		No Dist. + Data Aug.	96.42	29.00
		Morph (3MB)	96.43	34.50
		Morph (6MB)	96.80	20.00
	Vulnerability Prediction	w/o Data Aug	61.42	406
		w/ Data Aug.	61.13	245
		Morph (3MB)	59.57	326
		Morph (6MB)	59.99	275
GCodeBERT	Clone Detection	w/o Data Aug.	96.85	45
		w/ Data Aug.	96.52	40
		Morph (3MB)	94.97	53
		Morph (6MB)	96.42	24
	Vulnerability Prediction	w/o Data Aug	61.42	399
		w/ Data Aug.	62.41	214
		Morph (3MB)	58.77	284
		Morph (6MB)	58.75	191

we conducted ten independent runs and observed that, in one of these runs, the accuracy was similar to the values in the original paper. However, we report the average (median) results over these ten runs, which is the standard approach for evaluating meta-heuristics [63]. This methodology provides a more robust and reliable measure of performance by accounting for variability across multiple runs. The differences in accuracy between Morph and Avatar are statistically significant (p-value<0.01) with a large effect size ($\hat{A}_{12}$=0.97) only for clone detection. In contrast, the two approaches yield comparable results for vulnerability prediction.

Robustness. Besides accuracy, the number of prediction flips (on metamorphic test datapoints) is significantly lower for Morph compared to Avatar, indicating higher robustness of the student models distilled by our approach. In particular, Morph leads to -73% and -57% prediction flips in clone detection when distilling CodeBERT and GraphCodeBERT, respectively. For vulnerability detection, the model compressed by Morph exhibits -27% and -33% prediction flips for CodeBERT and GraphCodeBERT, respectively. These differences are statistically significant (p-value<0.01) with a large effect size ($\hat{A}_{12}$=1.00) for all tasks and LLMs.

Efficiency. Finally, Morph produces more efficient models than Avatar in terms of inference efficiency, measured in Giga FLOPs (floating-point operations). In particular, the student models distilled by Morph lead to -25% FLOPs compared to the baseline method, with the largest difference observed for the clone detection task. The differences are statistically significant (p-value<0.01) with a medium effect size ($0.65 \leq \hat{A}_{12} \leq 0.72$) for all tasks and models to distill.

Comparison with non-distilled models. Table IV compares the robustness of Morph with the original LLMs in terms of robustness and accuracy. Two versions of the teacher models are considered: one fine-tuned on the original training set (NoDist.) and another fine-tuned on an augmented set with metamorphic code (NoDist.+Data Aug.). For Morph, we present results for Pareto-optimal models sized at 3MB (to match Avatar) and 6MB, which remains far smaller than the 499MB teacher models. The 3MB models distilled by Morph reduce PFs by -19% and -27% for CodeBERT and GraphCodeBERT respectively, compared to the non-distilled LLMs for vulnerability prediction. However, teacher models fine-tuned on metamorphic code produce fewer PFs compared to Morph's 3MB models, indicating that using metamorphic transformations during fine-tuning enhances model robustness. Interestingly, Morph's 6MB models achieve similar or better robustness than augmented teacher models. For example, the 6MB models distilled from CodeBERT are more robust for clone detection and slightly more accurate. Similarly, GraphCodeBERT, the 6MB models distilled surpass the augmented teacher models in robustness on both downstream tasks.

C. *RQ3: Abaltion Study on* Morph*'s Key Features*

To evaluate the impact of the features in Morph, we conducted an ablation study by removing each feature individually and comparing the results with the full approach. The results are presented in Table V and Table VI. The former reports the accuracy, model size, number of prediction flips, and Giga FLOPs for the different ablated versions of Morph, while the latter reports the median hypervolume score achieved by each ablated version of Morph when using different MOEAs.

Impact of Augmentation. Disabling data augmentation with metamorphic code during training consistently leads to lower accuracy and lower robustness compared to the full approach. However, it is worth noting that even without this single feature enabled, Morph still outperforms Avatar in terms of robustness. This is due to the use of metamorphic

TABLE V
RESULTS OF THE ABLATION STUDY W.R.T. THE DIFFERENT FEATURES IN MORPH. WE REPORT THE MEDIAN RESULTS, WITH THE INTERQUARTILE RANGE (IQR) SHOWN IN PARENTHESES. THE BEST RESULT FOR EACH METRIC IS HIGHLIGHTED IN GRAY.

Model	**Task**	**Fatures**	**Acc.(%)**	**Size (MB)**	**# Flips**	**GFLOPs**
CB	CD	w/o Aug.	87.13 (2.60)	2.87 (0.02)	46.00 (50.00)	0.06 (0.03)
		w/o LHS	84.70 (0.90)	2.88 (0.01)	66.00 (18.00)	0.06 (0.01)
		w/o Repair	-	-	-	-
		All Features	96.43 (1.49)	2.94 (1.58)	34.50 (15.24)	1.33 (1.00)
	VP	w/o Aug.	58.67 (2.95)	2.98 (0.10)	405.00 (225.00)	0.61 (0.08)
		w/o LHS	57.79 (1.35)	3.02 (0.19)	267.00 (40.00)	0.69 (0.35)
		w/o Repair	-	-	-	-
		All Features	59.57 (0.99)	3.01 (0.01)	326.00 (37.00)	0.71 (0.26)
GCB	CD	w/o Aug.	94.48 (3.63)	3.07 (0.28)	60.50 (48.00)	1.23 (0.10)
		w/o LHS	94.95 (0.68)	3.06 (0.10)	65.50 (26.25)	1.72 (0.30)
		w/o Repair	-	-	-	-
		All Features	94.97 (1.66)	3.01 (0.06)	53.00 (6.25)	1.12 (0.17)
	VP	w/o Aug.	58.46 (3.34)	3.00 (0.01)	385.00 (314.00)	0.61 (0.25)
		w/o LHS	58.18 (2.95)	3.00 (0.01)	327.00 (95.00)	0.65 (0.40)
		w/o Repair	-	-	-	-
		All Features	58.77 (0.18)	3.03 (0.03)	284.00 (37.00)	0.71 (0.26)

code (of the validation set) as the fourth objective to optimize for. On the other hand, not using data augmentation leads to models with lower FLOPs (aka more efficient models).

Impact of Latin Hypercube Sampling (LHS). Removing LHS sampling from the search space exploration process leads to a decrease in accuracy and an increase in the number of prediction flips (lower robustness). The only exception can be observed for CodeBERT when detecting vulnerabilities. This suggests that LHS sampling is crucial for exploring the search space effectively and finding better model configurations.

Impact of Repair Mechanism. Disabling the repair operator leads to unreliable results, as the search process is not able to converge to valid solutions. In our experiment, all model configurations generated without the repair operator were invalid (all results are marked with "-"). Thus, this feature is vital to ensure the model's constraints are satisfied.

Impact of Surrogate Model. Removing the surrogate model dramatically increased the convergence time, with the running time rising from 3.5 seconds to 750 hours per distillation. Due to this extreme computational overhead, conducting and reporting a complete comparison with and without surrogate models was infeasible. Performing such an experiment for just one task and model would require approximately 750 hours $\times$ 10 repetitions on our dedicated server, equating to over 300 days of continuous computation. This confirms the importance of using surrogate models to complete the distillation process within reasonable time constraints.

Impact of AGE-MOEA. Table VI reports the median hypervolume (HV) score achieved by AGE-MOEA and the alternative MOEAs. MORPH achieves the highest HV score when using AGE-MOEA for all tasks and LLMs. The differences in HV scores between AGE-MOEA and the other MOEAs are statistically significant (p-value<0.05) with a large effect size ($\hat{A}_{12} \geq 0.82$) for all tasks and LLMs. The only two exceptions are NSGA-II and NSGA-III for the vulnerability prediction task with GraphCodeBERT, where the differences are not statistically significant (p-value>0.05), albeit with a medium positive effect size ($\hat{A}_{12} \geq 0.64$). These findings suggest that while other MOEAs might be considered

TABLE VI
MEDIAN HYPERVOLUME SCORE ACHIEVED BY MORPH WHEN USING DIFFERENT MOEAS. WE REPORT THE MEDIAN RESULTS, WITH THE INTERQUARTILE RANGE (IQR) SHOWN IN PARENTHESES. THE BEST RESULTS ARE HIGHLIGHTED IN GRAY COLOR.

Model	**Task**	**MOEAs**	**HV**	p**-value** ($\hat{A}_{12}$)
GCodeBERT	Clone Detection	NSGA-II	0.98 (0.002)	<0.01 (1.00)
		NSGA-III	0.963 (0.032)	<0.01 (1.00)
		MOEA//D	0.055 (0.333)	<0.01 (1.00)
		AGE-MOEA	0.996 (0.002)	-
	Vulnerability Prediction	NSGA-II	0.805 (0.019)	0.30 (0.64)
		NSGA-III	0.791 (0.014)	0.06 (0.75)
		MOEA//D	0.192 (0.020)	<0.01 (1.00)
		AGE-MOEA	0.822 (0.035)	-
CodeBERT	Clone Detection	NSGA-II	0.963 (0.006)	<0.01 (0.95)
		NSGA-III	0.968 (0.004)	<0.01 (0.97)
		MOEA//D	0.897 (0.033)	<0.01 (1.00)
		AGE-MOEA	0.980 (0.005)	-
	Vulnerability Prediction	NSGA-II	0.837 (0.024)	0.01 (0.82)
		NSGA-III	0.831 (0.009)	0.01 (0.82)
		MOEA//D	0.288 (0.425)	<0.01 (1.00)
		AGE-MOEA	0.872 (0.017)	-

as alternatives for optimizing student model configurations (or distillation configurations), AGE-MOEA produces higher-quality Pareto fronts for the considered tasks and LLMs.

VI. DISCUSSION AND PRACTICAL IMPLICATIONS

Overhead of using AGE-MOEA. One of the potential concerns in deploying multi-objective optimization algorithms is the required overhead they may introduce. Our experiment shows that AGE-MOEA only takes around 3.5 seconds per distillation, which is negligible compared to training a single 3MB model from scratch (on average, 9 minutes using our dedicated server). Moreover, training surrogate models — which estimate objective scores for different configurations— comes at a cost of approximately 1.5 seconds. Unlike iterative model training, which is resource-intensive and often requires specialized hardware, AGE-MOEA is a one-time overhead that can be executed efficiently on standard computing resources.

Impact on inference latency. We measured model performance using FLOPs, a standard practice to estimate the inference efficiency [10], [12]. To quantify the differences in a practical setting, we evaluate distilled and original models on a system with 8 CPU cores, simulating a typical consumer-grade laptop, and measure the average inference time on the test sets in seconds (s). The results confirm the differences we observed w.r.t. FLOPs: the average inference time for the entire Deving test set (2732 code fragments) is 0.21s for the models distilled by MORPH compared to 2.55s for AVATAR when detecting code vulnerability. Both models are much faster than the original LLMs as GraphCodeBERT requires 44.5s on average, and CodeBERT 22.5s. Therefore, MORPH produces more efficient models that can provide instantaneous feedback to developers for code analysis within their IDEs.

Practical implications. Our study highlights the importance of evaluating model compressing techniques beyond accuracy, model size, and efficiency. Robustness is critical to model performance, especially in real-time code analysis. Researchers should consider incorporating robustness metrics, such as prediction flips, into their evaluation frameworks to ensure that models are robust to the broader variety of naming conventions. Metamorphic testing can both measure/test the robustness of LLMs [16], [17], [33] but also improve models' robustness via data augmentation and calibrated distillation/-training methods. Our methodology can be extended to other distillation techniques and tasks, providing a general framework for improving model robustness.

VII. Threats to Validity

This section outlines potential threats to the study's validity and the steps taken to address them. Regarding *internal validity*, we used `ChatGPT-3.5` to create metamorphic code by renaming functions and input parameters using synonyms and acronym expansions. These changes preserve code snippets' functionality and abstract syntax tree (AST) without affecting their labels (clone or vulnerability). We validated the correctness and naturalness [21] of metamorphic code using parsers and manual inspection (see Sections III-A and IV).

For Morph's internal parameters, we followed the recommended values for genetic operators from existing guidelines in many-objective optimization [27], [36], [37], [54]. We used Avatar's original implementation and parameter settings [12], with adjustments to the population size (50) and number of generations (100) for a fair comparison. Future work will explore the impact of different parameter values. Both Morph and Avatar were optimized using the same training and validation sets to ensure fairness. Test sets, unseen during distillation and optimization, were used to evaluate student model performance. CodeBERT and GraphCodeBERT were fine-tuned using the same training and validation sets.

Data leakage is another potential threat as it is a common issue when evaluating LLMs [19]. To address this, we carefully selected the Devign [25] and BigCloneBench [26] datasets and focused on CodeBERT and GraphCodeBERT. The pretraining dataset (CodeSearchNet) for these LLMs is publicly available on HuggingFace[3], and it is fully traceable. For clone detection, we used BigCloneBench with its official training, validation, and test splits[4]. These splits are specifically designed to prevent data leakage, assuming the original BigCloneBench dataset does not inadvertently contain overlapping samples between the splits. We used the Devign dataset for vulnerability detection, which contains vulnerable C++ methods. Importantly, CodeSearchNet does not include C++ code[5], guaranteeing no overlap with the pretraining data of two LLMs. Furthermore, we manually checked that neither the pretraining nor the validation datasets contained any test snippets or their metamorphic variations. Additionally, as argued in [19], metamorphic testing helps mitigate the risk of data leakage in pre-training by altering original code snippets to create new semantically equivalent variants. These transformations ensure that even if a model has been exposed to similar patterns during pretraining, the metamorphic variants provide novel instances that challenge the model's robustness without introducing label inconsistencies.

Concerning *external validity*, we used the Devign [25] and BigCloneBench [26] datasets, widely employed in the literature for training AI models for code-related tasks [55], [64], including model distillation [11], [12]. Morph was assessed on two code-related tasks and two LLMs, CodeBERT and GraphCodeBERT. Future work will extend the evaluation to other datasets, tasks, and LLMs to corroborate our results.

Threats to *conclusion validity* arise from the randomized nature of Morph and Avatar. We ran each approach ten times for each LLM and task using different random seeds, following best practices for experiments with randomized algorithms [63]. We analyzed the results using the Wilcoxon rank-sum test [57] and the Vargha-Delaney statistics [58]. Metrics used included accuracy, model size, and FLOPs, common in knowledge distillation [11], [12]. The number of prediction flips was used to assess robustness against metamorphic code, a standard practice in the testing literature [17], [33].

VIII. Conclusion and Future Work

This study investigates the robustness of models compressed with the state-of-the-art distillation method, called Avatar, to metamorphic code variations. Our findings reveal a significant increase in prediction flips (up to 440%) for compressed model compared to their original counterparts when presented with semantically and behaviorally equivalent code snippets.

To address this issue, we introduced Morph, a novel distillation method that integrates metamorphic testing with many-objective optimization using AGE-MOEA [27]. Our approach considers the model robustness as a key objective to optimize, measured as the number of prediction flips (FPs) between metamorphic variants of the same code snippets. The other objectives are accuracy, model size (measured in MB), and inference efficiency (measured in FLOPS).

We conducted an empirical study on two tasks —clone detection and vulnerability detection— using CodeBERT and GraphCodeBERT. The results showed that Morph consistently produced more robust (up to 73% fewer PFs) and efficient models (up to 38% fewer FLOPS) compared to Avatar with equal or higher accuracy and (4) similar model size. Our ablation study confirm the importance and relevance of all Morph's key feature on its overall performance.

Future research will further corroborate our findings' generalizability by considering more code-related tasks and LLMs (such as CodeT5 [65]). We intend to design more metamorphic transformations that preserve the naturalness of developers' code and investigate their impact on model robustness. We plan to integrate further compression techniques to improve the performance of the distilled models.

Acknowledgements

This research was supported by JetBrains and conducted as part of the AI for Software Engineering (AI4SE) collaboration between JetBrains and Delft University of Technology.

[3]https://huggingface.co/datasets/code-search-net/code_search_net

[4]https://github.com/microsoft/CodeXGLUE/blob/main/Code-Code/Clone-detection-BigCloneBench

[5]https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/Defect-detection

References

[1] F. Liu, G. Li, Y. Zhao, and Z. Jin, "Multi-task learning based pre-trained language model for code completion," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 473–485.

[2] D. Guo, C. Xu, N. Duan, J. Yin, and J. McAuley, "Longcoder: A long-range pre-trained language model for code completion," in *International Conference on Machine Learning*. PMLR, 2023, pp. 12 098–12 107.

[3] M. Khajezade, J. J. Wu, F. H. Fard, G. Rodríguez-Pérez, and M. S. Shehata, "Investigating the efficacy of large language models for code clone detection," in *Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension*, 2024, pp. 161–165.

[4] T. Sonnekalb, B. Gruner, C.-A. Brust, and P. Mäder, "Generalizability of code clone detection on codebert," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–3.

[5] X. Zhou, T. Zhang, and D. Lo, "Large language model for vulnerability detection: Emerging results and future directions," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, 2024, pp. 47–51.

[6] Y. Chen, Z. Ding, L. Alowain, X. Chen, and D. Wagner, "Diversevul: A new vulnerable source code dataset for deep learning based vulnerability detection," in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, 2023, pp. 654–668.

[7] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, 2022, pp. 1–10.

[8] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu, "Codexglue: A machine learning benchmark dataset for code understanding and generation," *CoRR*, vol. abs/2102.04664, 2021.

[9] C.-J. Wu, R. Raghavendra, U. Gupta, B. Acun, N. Ardalani, K. Maeng, G. Chang, F. Aga, J. Huang, C. Bai *et al.*, "Sustainable ai: Environmental implications, challenges and opportunities," *Proceedings of Machine Learning and Systems*, vol. 4, pp. 795–813, 2022.

[10] R. Schwartz, J. Dodge, N. A. Smith, and O. Etzioni, "Green AI," *Communications of the ACM*, vol. 63, no. 12, pp. 54–63, 2020.

[11] J. Shi, Z. Yang, B. Xu, H. J. Kang, and D. Lo, "Compressing pre-trained models of code into 3 mb," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[12] J. Shi, Z. Yang, H. J. Kang, B. Xu, J. He, and D. Lo, "Greening large language models of code," in *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Society*, 2024, pp. 142–153.

[13] A. Svyatkovskiy, S. Lee, A. Hadjitofi, M. Riechert, J. V. Franco, and M. Allamanis, "Fast and memory-efficient neural code completion," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 329–340.

[14] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[15] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: a new approach for generating next test cases," Tech. Rep. Tech. Rep. HKUST-CS98-01, 1998.

[16] J. Cito, I. Dillig, V. Murali, and S. Chandra, "Counterfactual explanations for models of code," in *Proceedings of the 44th international conference on software engineering: software engineering in practice*, 2022, pp. 125–134.

[17] R. Compton, E. Frank, P. Patros, and A. Koay, "Embedding java classes with code2vec: Improvements from variable obfuscation," in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 243–253.

[18] A. Dwarakanath, M. Ahuja, S. Sikand, R. M. Rao, R. J. C. Bose, N. Dubash, and S. Podder, "Identifying implementation bugs in machine learning based image classifiers using metamorphic testing," in *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, 2018, pp. 118–128.

[19] J. Sallou, T. Durieux, and A. Panichella, "Breaking the silence: the threats of using llms in software engineering," in *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, 2024, pp. 102–106.

[20] D. Gibert, C. Mateu, J. Planes, and J. Marques-Silva, "Auditing static machine learning anti-malware tools against metamorphic attacks," *Computers & Security*, vol. 102, p. 102159, 2021.

[21] Z. Yang, J. Shi, J. He, and D. Lo, "Natural attack for pre-trained models of code," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1482–1493.

[22] S. Kim and D. Kim, "Automatic identifier inconsistency detection using code dictionary," *Empirical Software Engineering*, vol. 21, pp. 565–604, 2016.

[23] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, 2015, pp. 38–49.

[24] K. Kim, X. Zhou, D. Kim, J. Lawall, K. Liu, T. F. Bissyandé, J. Klein, J. Lee, and D. Lo, "How are we detecting inconsistent method names? an empirical study from code review perspective," *arXiv preprint arXiv:2308.12701*, 2023.

[25] Y. Zhou, S. Liu, J. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," *Advances in neural information processing systems*, vol. 32, 2019.

[26] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 476–480.

[27] A. Panichella, "An adaptive evolutionary algorithm based on non-euclidean geometry for many-objective optimization," in *Proceedings of the genetic and evolutionary computation conference*, 2019, pp. 595–603.

[28] A. Panichella, "Replication package of "metamorphic-based many-objective optimization of llms for code-related tasks"," https://doi.org/10.5281/zenodo.14753911, 2025, accessed: 2024-07-28.

[29] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter," *arXiv preprint arXiv:1910.01108*, 2019.

[30] X. Jiao, Y. Yin, L. Shang, X. Jiang, X. Chen, L. Li, F. Wang, and Q. Liu, "Tinybert: Distilling bert for natural language understanding," *arXiv preprint arXiv:1909.10351*, 2019.

[31] J. Su, D. V. Vargas, and K. Sakurai, "One pixel attack for fooling deep neural networks," *IEEE Transactions on Evolutionary Computation*, vol. 23, no. 5, pp. 828–841, 2019.

[32] 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.

[33] L. Applis, A. Panichella, and A. van Deursen, "Assessing robustness of ml-based program analysis tools using metamorphic program transformations," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2021, pp. 1377–1381.

[34] L. Applis, A. Panichella, and R. Marang, "Searching for quality: Genetic algorithms and metamorphic testing for software engineering ml," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2023, pp. 1490–1498.

[35] H. Ishibuchi, N. Tsukamoto, and Y. Nojima, "Evolutionary many-objective optimization: A short review," in *2008 IEEE congress on evolutionary computation (IEEE world congress on computational intelligence)*. IEEE, 2008, pp. 2419–2426.

[36] K. Deb, "Multi-objective optimisation using evolutionary algorithms: an introduction," in *Multi-objective evolutionary optimisation for product design and manufacturing*. Springer, 2011, pp. 3–34.

[37] H. Seada and K. Deb, "A unified evolutionary optimization procedure for single, multiple, and many objectives," *IEEE Transactions on Evolutionary Computation*, vol. 20, no. 3, pp. 358–369, 2015.

[38] F. U. Haq, D. Shin, and L. Briand, "Efficient online testing for dnn-enabled systems using surrogate-assisted and many-objective optimization," in *Proceedings of the 44th international conference on software engineering*, 2022, pp. 811–822.

[39] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[40] Q. Zhang and H. Li, "Moea/d: A multiobjective evolutionary algorithm based on decomposition," *IEEE Transactions on evolutionary computation*, vol. 11, no. 6, pp. 712–731, 2007.

[41] H. Hieu, N. Pham, T. N. Nguyen, and V. Nguyen, "Segment-based test case prioritization: A multi-objective approach," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ACM, Ed., 2024.

[42] Y. Shen and Y. Pan, "A multi-objective optimization approach of green building performance based on lgbm and age-moea," in *International Conference on Green Building, Civil Engineering and Smart City*. Springer, 2022, pp. 202–210.

[43] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Communications of the ACM*, vol. 59, no. 5, pp. 122–131, 2016.

[44] A. Khanfir, M. Jimenez, M. Papadakis, and Y. Le Traon, "Codebert-nt: code naturalness via codebert," in *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2022, pp. 936–947.

[45] Tree-sitter Developers, "Tree-sitter," https://github.com/tree-sitter/tree-sitter, 2024, accessed: 2024-11-28.

[46] T. Chugh, K. Sindhya, J. Hakanen, and K. Miettinen, "A survey on handling computationally expensive multiobjective optimization problems with evolutionary algorithms," *Soft Computing*, vol. 23, pp. 3137–3166, 2019.

[47] Y. Jin, "Surrogate-assisted evolutionary computation: Recent advances and future challenges," *Swarm and Evolutionary Computation*, vol. 1, no. 2, pp. 61–70, 2011.

[48] M. D. McKay, R. J. Beckman, and W. J. Conover, "A comparison of three methods for selecting values of input variables in the analysis of output from a computer code," *Technometrics*, vol. 42, no. 1, pp. 55–61, 2000.

[49] S. Jain and K. K. Dubey, "Employing the latin hypercube sampling to improve the nsga iii performance in multiple-objective optimization," *Asian Journal of Civil Engineering*, vol. 24, no. 8, pp. 3319–3330, 2023.

[50] A. Natekin and A. Knoll, "Gradient boosting machines, a tutorial," *Frontiers in neurorobotics*, vol. 7, p. 21, 2013.

[51] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. " O'Reilly Media, Inc.", 2022.

[52] B. Kazimipour, X. Li, and A. K. Qin, "A review of population initialization techniques for evolutionary algorithms," in *2014 IEEE Congress on Evolutionary Computation (CEC)*, 2014, pp. 2585–2592.

[53] K. Deb, "An efficient constraint handling method for genetic algorithms," *Computer Methods in Applied Mechanics and Engineering*, vol. 186, no. 2, pp. 311–338, 2000.

[54] K. Deb and R. B. Agrawal, "Simulated binary crossover for continuous search space," *Complex Syst.*, vol. 9, no. 2, 1995.

[55] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang *et al.*, "CodeBERT: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[56] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu *et al.*, "GraphCodeBERT: Pre-training code representations with data flow," *arXiv preprint arXiv:2009.08366*, 2020.

[57] W. J. Conover, *Practical nonparametric statistics*. john wiley & sons, 1999, vol. 350.

[58] A. Vargha and H. D. Delaney, "A critique and improvement of the CL common language effect size statistics of McGraw and Wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[59] K. Shang, H. Ishibuchi, L. He, and L. M. Pang, "A survey on the hypervolume indicator in evolutionary multiobjective optimization," *IEEE Transactions on Evolutionary Computation*, vol. 25, no. 1, pp. 1–20, 2020.

[60] M. Li, T. Chen, and X. Yao, "How to evaluate solutions in pareto-based search-based software engineering: A critical review and methodological guidance," *IEEE Transactions on Software Engineering*, vol. 48, no. 5, pp. 1771–1799, 2020.

[61] H. Ishibuchi, R. Imada, Y. Setoguchi, and Y. Nojima, "Performance comparison of nsga-ii and nsga-iii on various many-objective test problems," in *2016 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2016, pp. 3045–3052.

[62] J. Blank and K. Deb, "pymoo: Multi-objective optimization in python," pp. 89 497–89 509, 2020.

[63] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.

[64] L. Li, H. Feng, W. Zhuang, N. Meng, and B. Ryder, "Cclearner: A deep learning-based clone detection approach," in *2017 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 2017, pp. 249–260.

[65] Y. Wang, W. Wang, S. Joty, and S. C. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," *arXiv preprint arXiv:2109.00859*, 2021.