Enhancing Code Model Robustness Against Identifier Renaming via Unified Code Normalization

JIAJUN JIANG*, Tianjin University, China
SONGRUI LI, Tianjin University, China
HANMO YOU, Tianjin University, China
XINPENG WANG, Tianjin University, China
JIATENG FU, Tianjin University, China
YING XING, Beijing University of Posts and Telecommunications, China
XIAOFEI XIE, Singapore Management University, Singapore

Deep code models (DCMs) are critical for safety-sensitive tasks but are vulnerable to adversarial perturbations like subtle identifier renaming, which can induce out-of-distribution inputs and unsafe predictions. A key challenge lies in defending against such inputs without prior knowledge of adversarial patterns, as the space of possible perturbations is virtually infinite, and conventional rule-based defenses fail to generalize. To address this challenge, we propose UniCode, a novel two-stage defense framework against identifier renaming, one of the most effective modifications influencing DCMs' robustness.

First, it strips identifiers into placeholders, removing adversarial influence while preserving code structure. Then, it reconstructs meaningful identifiers using large language models, ensuring semantic integrity. By fine-tuning models on normalized inputs, UniCode achieves inherent robustness without attack-specific adaptations.

To evaluate the performance of UniCode, we have conducted a comprehensive evaluation, where UniCode outperforms state-of-the-art baselines on 85.19% of subjects, improving defense effectiveness by 9.57% ~ 46.16%. Moreover, our in-depth analysis reveals that UniCode effectively aligns the distributions between the training data and the adversarial test samples, thereby significantly enhancing model robustness.

CCS Concepts: • **Software and its engineering → Software maintenance tools**; • **Security and privacy →** *Domain-specific security and privacy architectures.*

Additional Key Words and Phrases: Deep Code Model, Code Normalization, Adversarial Attack, Model Defense

1 Introduction

Deep code models (DCMs) have gained significant traction across various applications [1], demonstrating remarkable performance in tasks such as vulnerability prediction [2] and code generation [3]. However, like all

*Corresponding author.

Authors' addresses: Jiajun Jiang, School of Computer Software, College of Intelligence and Computing, Tianjin University, Tianjin, China, jiangjiajun@tju.edu.cn; Songrui Li, lisongrui@tju.edu.cn, School of Computer Software, College of Intelligence and Computing, Tianjin University, Tianjin, China; Hanmo You, youhanmo@tju.edu.cn, School of Computer Software, College of Intelligence and Computing, Tianjin University, Tianjin, China; Xinpeng Wang, wxp3023244137@tju.edu.cn, School of Computer Software, College of Intelligence and Computing, Tianjin University, Tianjin, China; Jiateng Fu, world_2004@tju.edu.cn, School of Computer Software, College of Intelligence and Computing, Tianjin University, Tianjin, China; Ying Xing, xingying@bupt.edu.cn, Beijing University of Posts and Telecommunications, Beijing, China; Xiaofei Xie, xfxie@smu.edu.sg, Singapore Management University, Singapore.

ACM 1557-7392/2026/8-ART
https://doi.org/10.1145/3839364

deep learning models, they are susceptible to input perturbations, making them vulnerable to attacks by malicious actors. Given the widespread use of deep code models in safety-sensitive applications [4–6], ensuring their robustness is essential. For instance, a seemingly negligible change, such as renaming a variable in vulnerable code, may mislead these models into mistakenly classifying vulnerable code as safe [7–9], potentially resulting in severe security vulnerabilities.

In contrast to traditional handcrafted programs [10, 11], which operate deterministically based on a fixed set of machine instructions, deep learning models are trained on a finite set of input examples. During training, these models optimize their parameters within a predefined neural network architecture to achieve specific functional objectives, such as vulnerability prediction. However, this training paradigm introduces a fundamental challenge to the robustness of the learned models. That is, the limited scope of training data often fails to fully capture the diverse data distributions of all possible inputs. Consequently, these models frequently exhibit limited generalization capabilities, particularly when confronted with unseen or adversarially crafted inputs, as they may bring in out-of-distribution (OOD) features (e.g., renaming variables may introduce out-of-vocabulary issues [12, 13]). This limitation underscores the vulnerability of deep learning models to inputs that deviate from the training distribution, raising concerns about their reliability in real-world applications.

To enhance the robustness and reliability of DCMs against adversarial inputs, numerous defensive methods have been proposed over the past years [7–9]. These approaches typically generate new inputs with adversarial features based on predefined mutation rules. The model is subsequently fine-tuned on the augmented dataset produced by this process. The fundamental objective of this process is to mitigate OOD issues by expanding the feature space of the training data, thereby enhancing the model's overall predictive robustness. Despite these advancements, existing methods share a significant limitation: they rely on a fixed set of predefined mutation rules, which can be restrictive and challenging to generalize across diverse attack scenarios [8, 14, 15]. Since the generated inputs are inherently limited, they cannot fully encompass the vast spectrum of potential OOD inputs. Consequently, a model fortified against some adversarial examples may remain vulnerable to others. This highlights a significant challenge: effectively improving robustness against diverse attacks is inherently challenging, as the complete input space for deep code models is typically infinite, making it infeasible to account for all possible inputs.

To address this challenge, this paper aims to propose a novel defense mechanism capable of effectively countering identifier renaming attacks from different attacking methods [7, 8, 15–17]. However, a fundamental challenge in this domain (i.e., defense DCMs against attacks) stems from the inherent difficulty in predicting the specific modifications that adversaries might employ to induce model mispredictions, complicating the development of robust defense mechanisms. To gain a deeper understanding of the effectiveness of different modifications, we first conducted a preliminary analysis of the effectiveness of various attack methods. Our findings show that, although attacks leverage different mutation strategies, such as variable renaming, structural changes, and syntax-level modifications, the most effective modifications often involve identifier renaming, including changes to variable and method names. As such, this work primarily focuses on renaming-based attacks, which represent state-of-the-art adversarial strategies and have been shown to significantly degrade the performance of deep code models.

Specifically, to defend against renaming attacks, a significant challenge lies in precisely identifying vulnerable identifiers and effectively mitigating their impact on model predictions. This difficulty arises from two main factors: first, accurately pinpointing which identifiers contribute to OOD features is challenging due to the typically large number of identifiers in a code snippet, their implicit dependencies, and the large scale of the training data, all of which lead to a vast and complex search space. Second, even if vulnerable identifiers are successfully identified, restoring them in a way that preserves the original code semantics is non-trivial. This requires not only syntactic correctness but also a deep understanding of the code's context to avoid unintended side effects for prediction.

To address these challenges, particularly the issue of the vast search space, we propose a two-stage code normalization algorithm, comprising code *abstraction* and *instantiation*. In the first stage, code abstraction employs static program analysis to identify all identifiers in the code and replace them with semantically meaningless placeholders (e.g., var) while maintaining the original code structure and functionality. This step eliminates the adverse effects of potentially vulnerable or adversarial identifiers on the model's predictions. In the second stage, code instantiation leverages large language models (LLMs) to restore the semantic meaning of the placeholders by generating new identifiers based on the abstracted code. This stage is crucial, as the target of it is utilizing the contextual understanding capabilities of LLMs to restore the missing identifier semantics. By fine-tuning the model with this uniformly formatted training data, we ensure that it learns from a controlled and consistent data distribution. During inference, we apply the same transformation using the LLM to preprocess incoming inputs, aligning their distribution with that of the fine-tuned data. This alignment significantly improves the model's prediction confidence and thus enhances the robustness of the model.

As explained above, the core innovation of our approach is fundamentally different from existing robustness-enhancement methods. Prior strategies, such as adversarial training and input denoising, are largely reactive: they improve robustness within the original input space, either by augmenting the training dataset with adversarial examples or by restoring perturbed inputs toward the distribution expected by the model. As a result, these methods still rely on better covering, adapting to, or recovering the existing input distribution. In contrast, our approach aims to introduce a more general proactive transformation mechanism. Instead of repairing inputs or approximating the original training distribution, it transforms an input program through the abstraction–instantiation process into semantically equivalent but re-expressed variants. This transformation explicitly reshapes the representation distribution encountered by the model at inference time, reducing reliance on brittle surface features and encouraging predictions to be grounded in stable semantic content. By weakening the coupling between superficial code form and model behavior, our approach actively reduces the exploitable attack surface. In this sense, it should be understood as a proactive distribution-restructuring mechanism that improves robustness by reconfiguring how inputs are presented to the model.

To evaluate the effectiveness of our approach, we have implemented it as a tool, named UniCode, and conducted extensive experiments with it over three different code-related tasks involving four representative deep code models. Through comparing with five state-of-the-art defending methods over these tasks, the results show that our approach significantly outperforms the baseline methods. Specifically, UniCode achieves the best defense performance on 85.19% of subjects, with average improvements ranging from 9.57% ~ 46.16% over baselines in terms of defending effectiveness, demonstrating the superior performance of UniCode. Additionally, our in-depth analysis reveals that UniCode effectively aligns the distributions between training data and adversarial test samples, thereby significantly enhancing model robustness.

To sum up, this paper makes the following contributions:

- **A Novel Technique**: We introduced an innovative defense method for deep code models that utilizes code normalization to effectively eliminate attacking features while preserving code semantics, thereby enhancing prediction robustness.
- **A Preliminary Study**: We conducted a preliminary study to investigate the impact of various attacking methods, laying the groundwork for future research in this area.
- **Comprehensive Evaluation**: We performed extensive experiments across 1260 distinct scenarios (5 models × 3 tasks × 4 attackers × 7 defenders × 3 LLMs) to demonstrate the effectiveness of our approach, named UniCode. An additional ablation study highlights the contribution of each component within UniCode.
- **Open Science**: We published all experimental results and implementations, including baseline methods, at https://zenodo.org/records/17984782, to facilitate replication and support future research.

2 BACKGROUND AND MOTIVATION

2.1 Deep Code Models

Deep code models (DCMs) have emerged as the leading approach for code processing tasks, demonstrating superior performance across a variety of applications [18–20]. DCMs generally follow a two-stage training framework: pre-training on large-scale unlabeled code corpora to capture general code semantics, followed by fine-tuning on task-specific datasets [21]. Among these models, CodeBERT, GraphCodeBERT, CodeT5 and CodeT5+ are widely recognized as representatives [8, 22–24]. CodeBERT [25] integrates bimodal learning of programming and natural languages, while GraphCodeBERT [18] incorporates graph-structured information (e.g., ASTs) to combine semantic and structural features. CodeT5 [26] and CodeT5+ [27], as unified encoder-decoder models, learn code semantics through identifier-aware pre-training. Other prominent pre-trained DCMs include UniXCoder [20], AlphaCode [19], and InCoder [28], all of which effectively address both classification and generation tasks.

However, despite their power, DCMs are fragile and can be deceived by adversarial examples, which are carefully crafted inputs designed to manipulate the model into generating incorrect outputs. This susceptibility arises from their reliance on statistical patterns rather than true semantic understanding [12, 29], making them sensitive to subtle changes in code, particularly identifier renaming, as will be discussed in our preliminary study (Section 4). In this paper, we aim to enhance the performance of DCMs in defending against renaming-based adversarial attacks, thereby improving their robustness. Building on recent evaluations of DCMs [8, 24], our work primarily focuses on classification tasks, with plans to address generation tasks in future research.

2.2 Threat Model

In this paper, we consider a realistic deployment scenario where code models are used in practical tasks such as code review, vulnerability prediction, or malware analysis. In such settings, attackers query the model with source code inputs and analyze the corresponding outputs (such as predicted labels or confidence scores), while being denied access to internal model details (e.g., structure, parameters, gradients, or training data), which aligns with the actual deployment and usage of pre-trained code models. The attack methodology may modify superficial code features while strictly preserving the original program's functionality and behavior. Specifically, they may craft adversarial perturbations, such as renaming variable identifiers or applying semantics-preserving transformations, to manipulate the model's output, causing mispredictions or evading detection. The objective of our defense is to mitigate the impact of these adversarial perturbations and enhance the model's robustness and safety in real-world applications.

2.3 Problem Definition

Formally, given a code model $\mathcal{M}$, and a code snippet x, an output y is obtained by feeding x into $\mathcal{M}$ (i.e., $y = \mathcal{M}(x)$). For example, in vulnerability prediction, $y \in \{\texttt{True}, \texttt{False}\}$, where $y = \texttt{True}$ indicates the code is vulnerable. Attackers may manipulate an actual vulnerable snippet x (where $\mathcal{M}(x) = \texttt{True}$) by introducing perturbations to create x' without changing its functionality (vulnerabilities still exist), aiming to make $\mathcal{M}(x') = \texttt{False}$. This misclassification can lead to vulnerable code being incorrectly deemed safe and merged into a repository, posing significant safety risks.

Our primary task is to defend against such adversarial attacks by improving $\mathcal{M}$ to create a more robust model $\mathcal{M}'$ that correctly classifies both x and its adversarial counterpart x'. Specifically, for samples where $\mathcal{M}(x) = y$ but $\mathcal{M}(x') \neq y$, we aim to ensure that $\mathcal{M}'(x) = \mathcal{M}'(x') = y$. Following established definitions [8, 24], our focus is on maintaining correct predictions for adversarially perturbed inputs that were originally classified correctly. Therefore, we exclusively use such samples x (where $\mathcal{M}(x) = y$) in our experiments.

```java
boolean encodeFileToFile (String infile
    , String outfile ){
 boolean success = false;
 InputStream in = null;
 OutputStream out = null;
 try {
  in = new
       Base64.InputStream(...);
  out = new
       BufferedOutputStream(new
       FileOutputStream( outfile ));
  byte[] buffer =new byte[65536];
  int read= -1;
  while(( read=in.read( buffer ))>=0){
    out.write( buffer , 0, read);
  }
  success = true;
 } catch (IOException exc){...}
 return success;
}
```

(a) Original code

```java
boolean connectToUrl (String index
    ,String etag){
 boolean img = false;
 InputStream uid = null;
 OutputStream fout = null;
 try {
  uid = new
       Base64.InputStream(...);
  fout = new
       BufferedOutputStream(new
       FileOutputStream(etag));
  byte[] urlObj =new byte[65536];
  int read= -1;
  while((read=uid.read( urlObj ))>=0){
    fout.write( urlObj , 0, read);
  }
  img = true;
 } catch (IOException exc) {...}
 return img;
}
```

(b) Attacked code by CODA

```java
boolean encodeFiletoFiles (String infp,
    String newname ){
 boolean open = false;
 InputStream inside = null;
 OutputStream point = null;
 try {
  inside = new
       Base64.InputStream(...);
  point = new
       BufferedOutputStream(new
       FileOutputStream( newname ));
  byte[] url = new byte[65536];
  int pass = -1;
  while(( pass = inside .read( url ))>=0){
    point .write(url , 0, pass);
  }
  open = true;
 } catch (IOException exc) {...}
 return open;
}
```

(c) Attacked code by ALERT

```java
// Combinatorial number of naming variants -> one
    abstract representation.
boolean methodName (String var1 , String var2) {
  boolean var3 = false;
  InputStream var4 = null;
  OutputStream var5 = null;
  try {
    var4 = new Base64.InputStream(...);
    var5 = new BufferedOutputStream(new
         FileOutputStream(var2));
    byte[] var6 = new byte[65536];
    int var7 = -1;
    while ((var7 = var4.read(var6)) >= 0) {
      var5.write(var6, 0, var7);
    }
    var3 = true;
  } catch (IOException var8) { ... }
  return var3;
}
```

(d) Abstracted code by our approach

```java
// Generate semantically consistent identifiers conditioned
    on abstract structure.
boolean encodeFileToFile (String inputFilePath , String
    outputFilePath ){
  boolean success = false;
  InputStream inputStream = null;
  OutputStream outputStream = null;
  try {
    inputStream = new Base64.InputStream(...);
    outputStream = new BufferedOutputStream(new
        FileOutputStream( outputFilePath ));
    byte[] buffer = new byte[65536];
    int bytesRead = -1;
    while (( bytesRead = inputStream .read( buffer ))>=0){
      outputStream .write( buffer , 0, bytesRead );
    }
    success = true;
  } catch (IOException ioException ) { ... }
  return success;
}
```

(e) Instantiated code by our approach

Fig. 1. A simplified real-world example code snippet from our experiment. Specifically, the example shows that CODA and ALERT tend to rename variables into different ones (we highlight success as an example). This causes the model refined by CODA (or ALERT) to fail in defending against attacks from ALERT (or CODA) due to the resulting differences in feature distribution. In contrast, our method first abstracts all identifiers regardless of their original names and then instantiates them in a consistent way. By fine-tuning the model with code that has been unified through this consistent instantiation procedure, our method enables the model to defend against attacks from diverse attack methods, since any attacked variables will be abstracted and instantiated uniformly.

2.4 Motivating Example

As aforementioned, limited training data and distribution shifts make deep learning models susceptible to adversarial attacks, while existing methods primarily address this issue by augmenting training datasets. However, their universal robustness, i.e., their ability to defend against a wide range of attacks [30, 31] remains limited,

as the augmented data often fails to capture the full diversity of potential inputs. This section highlights the challenges faced by them through a real-world example.

Figure 1 presents example code snippets from our experiment. Specifically, Figure 1a shows the original code, while Figure 1b and Figure 1c illustrate adversarial examples generated by two representative attack methods, CODA [8] and ALERT [7], respectively. Figures 1d and Figure 1e further show the refined code after applying our approach. From the figure, we observe that although both attacking methods involve identifier renaming, the newly generated identifiers can differ significantly. For instance, the variable *success* is replaced by *img* and *open* in the case of CODA and ALERT, respectively. This discrepancy results in fine-tuning on adversarial data from one generation method, impairing defense against another due to the differing attack strategies yielding distinct data distributions. Given limited training data but a virtually infinite space of adversarial inputs, universally strengthening deep code models' robustness remains a major challenge. To tackle this challenge, existing approaches [24] iteratively identify and denoise adversarial identifiers instead of merely augmenting data. However, due to the large number of variables and the complex semantic interactions among them, the search space becomes extremely large. For example, there are in total eight variables in the example code shown in Figure 1, resulting in 256 (2^8) modifications to the variables when considering all possible combinations. This combinatorial explosion limits these methods to handling only a small number of modified identifiers at a time. For example, CodeDenoise [24] only identifies one vulnerable variable.

As a result, precisely identifying which variables (can be multiple) are vulnerable in the code is still challenging, given the large combination space. This paper addresses this challenge by stripping all identifiers and regenerating semantically equivalent names via an LLM. This uniform renaming aligns the feature distribution of adversarial inputs, whether from CODA, ALERT, or others, with the original training data as illustrated in Figures 1d and 1e. Furthermore, this process overcomes the large search space by removing all variables together rather than identifying the precise combination. Then, the regenerated variables aim to resemble the semantics of the code while being free from attacks. Consequently, fine-tuning on these normalized examples restores robustness across diverse attack strategies. We will introduce the details of our approach in Section 5.

3 RELATED WORK

3.1 Adversarial Attack on Deep Code Models

Adversarial attacks involve adding imperceptible perturbations to input data to mislead models, mainly categorized into white-box and black-box attacks [32]. In white-box attacks, attackers access the internal parameters to generate adversarial examples, often guided by gradients. Yefet et al. [33] proposed DAMP, generating targeted attacks by modifying inputs based on gradients. Zhang et al. [15] proposed CARROT, optimizing examples with rule-based constraints, and Henkel et al. [34] proposed k-adversary to limit attacks with k-times changes, improving DAMP with a new loss function. Black-box attacks refer to scenarios where attackers cannot access the target model's internal parameters. Given that the internal structures of most DCMs are not directly accessible, this type of attack holds more practical value. Zhang et al. [17] proposed MHM, using the Metropolis-Hastings sampling strategy to decide whether to replace selected variable names. Yang et al. [7] employed CodeBERT to generate contextually appropriate variable names, enhancing the naturalness of the test cases. Similarly, Zhang et al. [35] proposed CBA-SA, using simulated annealing to improve the effectiveness of CodeBERT-based adversarial attacks. Zeng et al. [36] proposed using token importance to further improve attack effectiveness. Tian et al. [8] proposed CODA, a method that analyzes code features from various categories and applies targeted transformations to generate adversarial examples. Yang et al. [14] proposed CodeTAE, which uses the pre-trained encoder model as the agent model for attacks. Furthermore, Pour et al. [37] developed the GM method, generating adversarial examples that can reveal error behaviors of DCMs guided by mutation scores. Zhang et al. [38] and

Tian et al. [39] further use reinforcement learning for effective attacking. Unlike these works, our paper focuses on effectively defending against renaming-based attacks.

3.2 Improving Robustness of Deep Code Models

Existing defense methods for DCMs can be broadly categorized into two types: offline methods, such as adversarial training, and online methods, like pre-processing. Many studies [7, 8, 14, 15, 17] on adversarial attacks have demonstrated that adversarial training using generated examples can effectively enhance model robustness. However, these methods are predominantly effective against known attacks but show limited effectiveness against novel attacks. Another work, MARVEL [40], first trains a model to recognize noise patterns and then uses mutual learning between two parallel models (one for clean data, one for noisy data) to enhance robustness against adversarial attacks. However, similar to adversarial training, it incurs high computational costs. Among online defense methods, CodeDenoise [24] is the most representative. It aims to improve model performance by removing noise from critical identifier names. However, when facing attacks involving a larger number of identifier changes, its defense capabilities are insufficient. In contrast, UniCode normalizes all code data, saving adversarial example generation time. The online normalization process can effectively defend against all identifier renaming attacks, making the DCM more robust.

3.3 Variable Name Recovery

Recovering variable names has been studied in both source-code and binary-analysis settings. Early work on source code focuses on learning naming conventions from large corpora. For example, NATURALIZE [41] shows that identifier names follow statistical regularities and that language models can be used to suggest more natural names. Subsequent studies further incorporate structural information, such as graph-based program representations [42], to infer variable names from usage context. Another line of work investigates name recovery from minified or obfuscated code. Techniques such as JSNice [43] model the problem as predicting program properties by leveraging probabilistic relationships among program elements, while JSNeat [44] utilizes information retrieval over large codebases to recover semantically meaningful identifiers. These approaches highlight the importance of contextual semantics in variable naming, but they still rely on source-level syntactic and semantic information that is not available in stripped binaries. More recent work addresses variable name recovery in binary code. DEBIN [45] is an early system that predicts symbolic information directly from stripped binaries using learned probabilistic models. Building on decompiled code representations, DIRE [46] introduces neural models to recover identifier names from decompiler outputs. VarBERT [47] formulates the task as constrained masked language modeling, while DIRTY [48] adopts a Transformer-based architecture to predict variable names and types jointly. Although these methods substantially improve variable naming quality, they primarily focus on recovering semantically meaningful names for a given input program and are still constrained by the quality and scope of the available program context. With the advent of large language models (LLMs), recent approaches further improve variable naming by leveraging generative modeling and broader contextual information. GenNm [49] employs pretrained generative code models and incorporates caller–callee relationships to enhance naming quality. ReSym [50] extends this line of work by combining LLM-based predictions with program analysis to improve consistency and recover richer semantic information, including variable and data-structure symbols.

In contrast to these prior studies, which aim to generate human-readable and semantically meaningful variable names for improving code interpretability, our work addresses a different objective. We do not attempt to recover original or developer-intended names. Instead, we aim to enhance model robustness under semantics-preserving transformations by removing adversarial perturbations and generating semantically consistent identifiers. Consequently, our method emphasizes prediction consistency across semantically equivalent program variants, rather

than the reconstruction quality of variable names for a single input. Nonetheless, these existing approaches can be integrated into our method for identifier instantiation.

3.4 Code Normalization

Code normalization aims to transform programs into standardized or canonical representations by reducing superficial variations while preserving program semantics. Existing approaches primarily focus on improving code representations for downstream tasks such as code search, clone detection, and representation learning. Several studies achieve normalization by abstracting away syntactic and lexical differences. For example, Aroma [51] represents code using simplified structural features that ignore variable names and formatting differences, enabling effective structural code search. Neural representation learning approaches, such as code2vec [52] and code2seq [53], also implicitly perform normalization by mapping structurally similar programs into comparable vector representations, often abstracting away identifier-specific information. Related ideas also appear in code clone detection and canonicalization techniques [54], where programs are transformed into normalized structural representations (e.g., based on abstract syntax trees) to facilitate similarity comparison.

Different from these approaches that aim to eliminate superficial differences across code snippets and obtain fixed or consistent representations of semantically similar code, our work leverages a semantics-preserving abstraction–instantiation process to actively transform the input distribution, rather than targeting a fixed representation. This process removes perturbations introduced at the lexical level while reintroducing semantically consistent identifiers, thereby reducing the model's reliance on superficial features. As a result, our method improves robustness by encouraging consistent predictions across semantically equivalent program variants, rather than solely learning invariant representations.

3.5 Code Obfuscation

Code obfuscation aims to transform programs into functionally equivalent but harder-to-understand forms, with the goal of preventing reverse engineering and program analysis. Existing techniques typically introduce semantics-preserving transformations that increase the complexity of program structure, data flow, or control flow. A large body of work focuses on control-flow obfuscation. For example, control-flow flattening [55] transforms structured control constructs into dispatcher-based state machines, significantly increasing structural complexity while preserving program behavior. Opaque predicates [56] are another widely used technique, which introduce branches with conditions that are known to the obfuscator but difficult for an analyst to resolve, thereby creating infeasible execution paths and complicating program analysis. More advanced variants further strengthen such predicates to hinder modern analysis techniques, for example, by increasing path complexity or introducing hard-to-resolve constraints [57]. In addition to control-flow transformations, obfuscation techniques also operate on data and program structure. These include identifier renaming, data structure transformations, and code insertion (e.g., dead code) [58], all of which aim to obscure program semantics without altering functionality. Such transformations increase the difficulty of both static and dynamic analysis by introducing noise and disguising semantic intent. Recent work [59] has also begun to explore the use of large language models (LLMs) to generate semantics-preserving obfuscated code, demonstrating the potential of data-driven approaches to produce diverse and flexible transformation strategies.

Despite their effectiveness in hindering program understanding, these approaches are designed to deliberately introduce semantics-preserving perturbations to make analysis more difficult. They are not intended to improve the robustness of code models or to ensure consistent predictions under such transformations. In contrast, our work takes an opposite perspective. Instead of introducing obfuscation, we aim to eliminate adversarial perturbations through a semantics-preserving abstraction–instantiation process. By transforming inputs into

semantically consistent variants, our method reduces the model's reliance on superficial features and improves robustness against semantics-preserving transformations.

3.6 Code Refactoring

Code refactoring aims to restructure programs to improve code quality, readability, and maintainability while preserving their original semantics. It consists of a series of semantics-preserving transformations that modify program structure without changing behavior.

Traditional refactoring techniques are typically rule-based. Fowler systematized refactoring as a catalog of behavior-preserving transformations, such as renaming, extraction, and modularization, which improve software design without altering program behavior [60]. More recent work has moved beyond hand-crafted rules to learning-based approaches. For example, RMove [61] recommends refactoring opportunities by analyzing structural and semantic relationships in code, while REMS [62] identifies opportunities to decompose complex code fragments into simpler components by capturing different aspects of program structure and behavior. Subsequent work further incorporates deep learning models to better capture program semantics. For instance, HMove [63] models relationships among code elements across classes and leverages learned representations to more effectively recommend such refactoring operations. More recently, large language models (LLMs) have been applied to automated refactoring. Liu et al. [64] observe that while LLMs can identify refactoring opportunities and generate refactored code, their outputs may be inconsistent or unsafe without additional constraints. To address this issue, they propose a detect-and-reapply strategy, in which the LLM first identifies refactoring opportunities and generates candidate transformations, and a verified refactoring engine then reapplies these transformations to ensure correctness.

Despite these advances, existing refactoring techniques primarily aim to improve code quality and maintainability from a human-centric perspective. They focus on producing cleaner or more modular code, rather than addressing the robustness of machine learning models under semantics-preserving transformations. Specifically, the attacked code is typically well-structured and thus cannot be identified as potential refactoring candidates. In contrast, our work focuses on enhancing model robustness. Instead of restructuring code for maintainability, we remove all identifiers from the code and then regenerate semantically consistent program variants, thereby reducing the model's reliance on superficial features and improving prediction consistency across equivalent transformations.

4 PRELIMINARY STUDY

DCMs are vulnerable to adversarial attacks, leading to potential safety concerns. To gain a comprehensive understanding of the most effective attacks on DCMs and promote effective defense techniques to improve model robustness, we conducted a preliminary study aimed at answering the research question – *What type of code changes are effective at inducing mispredictions in DCMs?* To address this, we experimentally analyzed the code changes involved in existing representative attack techniques. Specifically, we first collected the most relevant and latest research papers about DCM attack from the IEEE Xplore, ACM Digital Library and Google Scholar, ensuring the papers are peer-reviewed and have been published on the top-tier venues in the software engineering domain no earlier than 2020, which can ensure the representativeness of the proposed approaches. Finally, we present these most related approaches in Table 1.

Specifically, Table 1 summarizes all the code changes adopted by these latest representative attacking methods for DCMs. In summary, these code changes can be primarily classified into two groups based on the code elements involved, i.e., identifier renaming and structure changes. The first group includes renaming variables, method names, and self-defined data types (e.g., changing *struct a{}* to *struct b{}*), while the second group includes updating the structures of loops (e.g., changing *while* to *for*), branches (e.g., changing *switch-case* to *if-else*), dead code

Table 1. Code changes widely-adopted by existing methods.

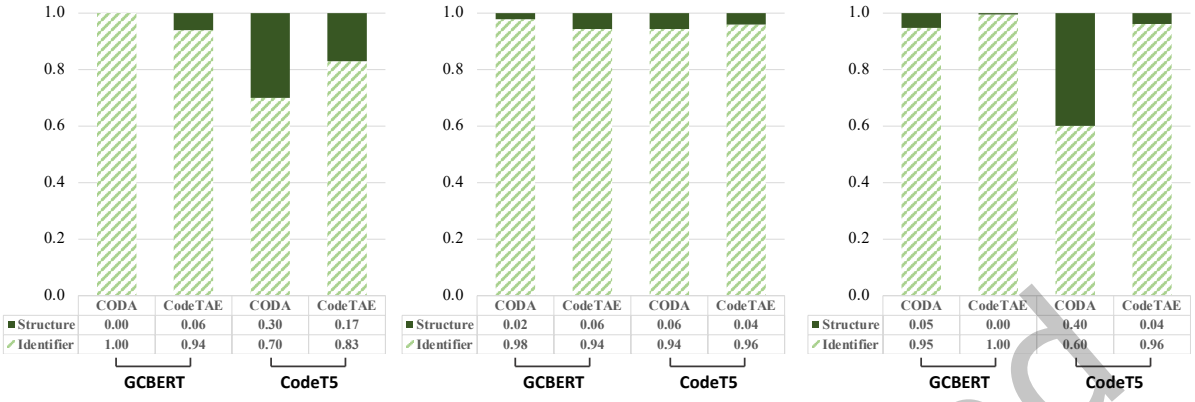
Approach	Identifier Rename			Structure Change				
	Variable	Method	Data Type	Loop	Branch	Dead Code	Arithmetic	Constant
CODA[8]	✓	✓	✓	✓	✓	×	✓	✓
CodeTAE[14]	✓	×	×	×	×	✓	×	✓
MHM[17]	✓	✓	×	×	×	×	×	×
ALERT[7]	✓	✓	×	×	×	×	×	×
ITGen[16]	✓	✓	×	×	×	×	×	×
CBA-SA[65]	✓	×	×	×	×	×	×	×
CodeAttack[66]	✓	✓	×	×	×	×	×	×
DAMP[33]	✓	×	×	×	×	✓	×	×
DIP[67]	×	×	×	×	×	✓	×	×
CARROT[15]	✓	✓	✓	×	×	✓	×	×

Table 2. Information of studied datasets and models.

Task	Dataset	#Train/#Validate/#Test	Model	Acc.
Clone Detection	BigCloneBench [68]	90,102/4,000/4,000	GCBERT	96.73%
			CodeT5	96.40%
Vulnerability Prediction	Zhou et al. [69]	21,854/2,732/2,732	GCBERT	64.13%
			CodeT5	59.99%
Defect Prediction	CodeChef [70]	27,058/−/6,764	GCBERT	84.89%
			CodeT5	88.82%

(i.e., inserting dead code), arithmetic expressions (e.g., changing *a+=b* to *a=a+b*), and constant expressions (i.e., changing *a=1* to *{b=1, a=b}*). As a consequence, we will explore the performance of each group of code changes in attacking DCMs. In particular, in our preliminary study, we selected CODA [8] and CodeTAE [14] as the most representative attacking methods for analysis as they are **recently proposed and the most effective attacking methods** from two representative attacking strategies (using predefined mutation rules and transfer learning) [14] that involve both of the two groups of code changes. For example, CARROT was demonstrated to underperform CODA [8]. This configuration enables a direct and reliable comparison between change groups within a manageable experimental scale.

Tasks and Configurations: To comprehensively investigate the contribution of different code changes, we adopted three widely-used code-related tasks by following existing studies [7, 8, 15], including clone detection, vulnerability prediction, and defect prediction. We also selected the representative dataset for each task by following these studies. Specifically, the clone detection task aims to detect whether two given code snippets are semantically equivalent. The dataset for this task is BigCloneBench [68], the most widely-used dataset of Java programs. The vulnerability prediction task aims to predict whether a given code snippet has vulnerabilities. The dataset for this task is the one proposed by Zhou et al. [69], which consists of a set of vulnerable code snippets of C programs. Finally, the defect prediction task aims to evaluate whether a given code snippet is defective and its defect type. The adopted dataset is CodeChef [70], which is of C/C++ programs.



(a) Clone Detection (b) Vulnerability Prediction (c) Defect Prediction

Fig. 2. Contribution of different groups of code changes.

Regarding the model architectures of victim models (i.e., the models to be attacked), we selected GraphCodeBert (GCBRET) [18] and CodeT5 [26], covering two different kinds of backbone structures (Encoder-only *vs* Encoder-Decoder), as two representatives, both of which have achieved outstanding performance in these studied tasks [8]. Table 2 summarizes the details, including data configurations and model accuracies.

Process: To facilitate the exploration of the contributions of different groups of code changes, we configured the corresponding attacking methods to generate adversarial examples using each group of code changes independently. Specifically, given a well-trained DCM, for each attack method, we restrict it to using only one type of attack, either identifier renaming or structural transformation, and evaluate it on the same candidate sample set. This candidate set consists of samples from the test set that are correctly predicted by the target model, ensuring that the evaluation of attack effectiveness is conducted on a consistent and comparable basis. To assess the effectiveness of identifier renaming attacks, for each candidate sample, we sequentially apply all renaming strategies designed by the method. The process is as follows: we iteratively try different renaming strategies until the target model produces a misclassification on that sample, or all strategies have been exhausted. We then count the number of samples for which the attack successfully induces misclassification in the target model, and use this as the evaluation metric for the method under the identifier renaming setting. For structural transformation attacks, we adopt the same evaluation procedure. In other words, each adversarial example was generated by either adopting identifier renaming or changing the code structure. The combination of these two groups was not allowed to enable the contribution analysis of different groups. Note that all generated adversarial examples will make the model produce incorrect predictions.

Preliminary Result: Figure 2 presents the experimental results from our preliminary study, showing the proportions of adversarial examples generated using different groups of code changes, which successfully misled the victim models into making incorrect predictions. From the figure, we can conclude that identifier renaming is almost the dominant code change, producing the most effective adversarial features (i.e., out-of-distribution inputs) to attack DCMs. In the tasks of clone detection and defect prediction, 60% to 100% of valid adversarial examples (i.e., those leading to incorrect predictions) were generated through identifier renaming, with over 90% of these valid examples stemming from this group of code changes in the task of vulnerability prediction. In contrast, the attack effect produced by only a structural attack is marginal.

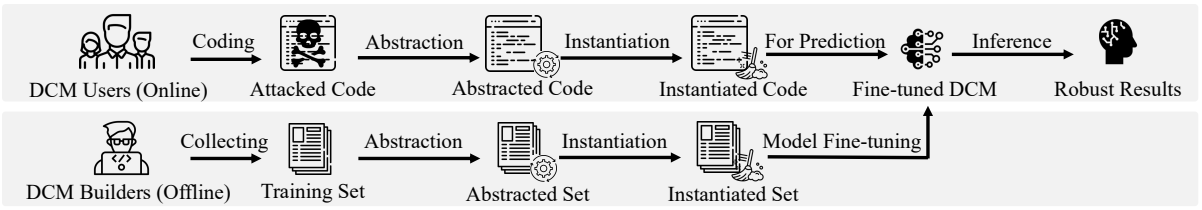


Fig. 3. Overview of UniCode

In summary, identifier renaming consistently outperforms code-structure changes in generating valid adversarial examples across all tasks and methods. To assess the statistical significance of our findings, we computed p-values using the Wilcoxon Signed-Rank Test [71] at the confidence level of 0.95. As a result, the *p-value* is 4.88e-4 (<0.05), demonstrating the significance of the effectiveness discrepancy between them. These results demonstrate that identifier renaming is quite an effective type of code change for attacking DCMs based on our preliminary results.

Observation: Identifier renaming is an extremely effective type of code change in generating adversarial examples that induce mispredictions in DCMs, underscoring the urgent need to address this type of attack to ensure the robustness of these models.

5 APPROACH

Our preliminary study indicates that identifier renaming is effective in generating out-of-distribution inputs, causing DCMs to produce incorrect or vulnerable predictions. However, existing defenses, whether by data augmentation or targeted denoising, fail due to limited samples and an exploding search space of identifiers.

To address this challenge, we propose a unified defense mechanism called UniCode, which particularly counters identifier renaming attacks from various adversaries. The core idea is to apply a unified transformation that converts each input into semantically equivalent but diverse representations. Specifically, the syntactic structures and dependency relations will be retained while the names of variables may be updated. In this way, instead of restoring inputs to resemble the training data, this process proactively reshapes the representation distribution seen by the model, enabling it to rely less on superficial patterns and more on stable semantic features. Specifically, UniCode consists of three main components as shown in Figure 3: (1) *Code Abstraction* (Section 5.1): Removing potentially vulnerable identifiers by replacing them with meaningless placeholders. (2) *Code Instantiation* (Section 5.2): Utilizing LLMs to restore the semantic meaning of the placeholders by generating new identifiers based on the abstracted code, aiding model predictions. (3) *Model Fine-tuning* (Section 5.3): This one-time, offline process fine-tunes the model using the normalized training set, helping it adapt to the characteristics of the normalized code and enhancing its robustness against various identifier-renaming attacks.

5.1 Code Abstraction

As mentioned earlier, accurately identifying all potentially vulnerable identifiers is challenging due to the vast search space. To tackle this issue, UniCode incorporates a code abstraction stage that utilizes static program analysis to identify all identifiers and substitute them with meaningless placeholders, effectively eliminating all identifiers in the process.

Specifically, the code abstraction process begins by parsing the given code snippet into an abstract syntax tree (AST) and constructing the corresponding Data Flow Graph (DFG). The DFG is then traversed to identify and substitute all identifiers, ensuring that data dependencies remain intact during the substitution. Algorithm 1

Algorithm 1: Code Abstraction in UniCode

```
   Input: x: the initial input code; parser: syntax parser
   Output: x′: the abstracted code;
1  x′ ← x
2  AST ← parser.parse(x) ;                          // Parse code into abstract syntax tree
3  DFG, Mapping ← extract_dataflow(AST);            // Construct DFG and code location mappings
4  ID ← []
5  foreach node ∈ DFG do
6      if is_identifier(node.id) then
7          ID ← ID ∪ {node.id};
8      end
9  end
10 num ← 1
11 foreach v ∈ ID do
12     position ← find_position(x′, Mapping, v);    // Find the location of variable v in code
13     x′ ← substitute(v, position, x′, "var"+num); // Substitute all variables
14     num ← num + 1
15 end
16 x′ ← substitute_method(x′);                      // Substitute the declared method name
17 return x′
```


outlines the detailed procedure of the code abstraction process. Given a code snippet x, UniCode first parses it into the AST format (Line 2) and then constructs the DFG based on the AST. This process records all tokens in a map (i.e., *Mapping* in Line 3) along with their associated locations in the original code. This step is crucial because UniCode aims to substitute identifiers without altering non-relevant code features, such as comments and formatting. Therefore, the identifier substitution occurs on the original plain text format of the code, as converting the AST back to source code may inadvertently change these features.

Subsequently, for each node in the DFG, UniCode checks whether it is an identifier node and collects them in a set called *ID* (Lines 5-9). Finally, UniCode substitutes all identifiers based on their positions stored in the map with meaningless identifiers in the format "*var*" + *num*, where "*num*" is an integer (Lines 11-15). It is important to note that all references to the same identifier will be consistently substituted with the same new identifier, ensuring the semantic correctness of the code after substitution. To maintain all code features except for the identifiers, UniCode creates a new copy (i.e., x') of the original code during this process. Additionally, the method names in the code snippet will be substituted with "*methodName*" (Line 16). In this way, every identifier in the code will be substituted, regardless of whether it is potentially vulnerable. Particularly, this process will not modify identifiers associated with API calls, but will be strictly limited to user-defined identifiers. This design ensures that existing API semantics are preserved, thereby avoiding unintended side effects on external interfaces or predefined functionalities.

For example, in the code snippets shown in Figure 1, regardless of how the variable *success* in the original code is renamed, e.g., *img* by CODA and *open* by ALERT, UniCode will consistently substitute them with the placeholder identifier *var3*. This effectively eliminates the adverse effects of all potential vulnerable or adversarial identifiers.

5.2 Code Instantiation

While replacing all identifier names in a given code snippet with meaningless placeholders can effectively eliminate potential attacks, it also sacrifices the prediction performance of the models due to the loss of code semantics embedded in the abstracted names, as demonstrated by the data in Table 5 below. To maintain the performance of DCMs, the code instantiation stage aims to restore the lost identifier semantics. Specifically, UniCode leverages LLMs to assist in regenerating these identifiers for recovering the semantics they may convey, as LLMs have demonstrated strong contextual understanding of source code [72] and are effective in code generation tasks [73].

There are various specialized methods for identifier name construction. For instance, the statistical-based approach [74] aims to identify the most frequently occurring or highly relevant identifiers in code contexts. However, this method is not suitable for our tasks, as we cannot determine whether the identifier names in the context have been compromised. Blindly reusing these names may introduce additional risks. Furthermore, traditional approaches [43, 75] often lack an understanding of the code snippet's functionality, resulting in names that may not accurately represent code semantics.

In contrast, our approach, which utilizes LLMs for restoring identifier names, offers several advantages. First, LLMs have been shown to possess a remarkable ability to understand code semantics [76], making them more likely to generate meaningful identifier names that closely reflect the functions of the code snippets. Second, LLMs exhibit strong pattern recognition capabilities [77, 78], enabling them to produce clean identifier names that follow consistent and normalized patterns, thereby reducing the likelihood of introducing compromised identifier names.

To effectively utilize LLMs for generating attack-free and meaningful identifier names, we carefully designed the prompt in accordance with existing guidelines [79, 80]. The detailed prompt for restoring identifier names is illustrated in Figure 4, which includes a one-shot example [81]. Specifically, the prompt consists of three main components: (1) Task definition: This part, presented at the beginning of the prompt, explicitly outlines the objective, which is to replace all meaningless placeholder identifiers (e.g., var1) by understanding the contextual semantics of the given code snippet. The goal is to ensure that this replacement maintains the program's functionality. In particular, to ensure the instantiated code semantically equals the original one, we instructed the model to replace the same token in different locations with the same new token. (2) One-shot example: Here, we provide a clear example of how identifiers should be replaced. This includes an input code snippet with placeholders and the corresponding target code featuring the recovered identifiers. In our evaluation, UniCode randomly selects one example from the training data for each programming language, which is used consistently throughout the experiments. (3) Input code for instantiation: This part contains the code that needs to be instantiated. By structuring the prompt this way, LLMs can effectively grasp the task and generate meaningful identifiers for replacement.

For instance, taking the code shown in Figure 1 as an example, the abstracted code in Figure 1d can be accurately instantiated (as shown in Figure 1e). When compared to the original code in Figure 1a, the instantiated identifiers can be even more informative, as they are closely aligned with their semantic meanings, whereas the original identifiers may not be, such as the variable *infile*, which is instantiated as *inputFilePath*. This demonstrates that UniCode is indeed effective in recovering the semantics of the code by leveraging the strong code understanding capabilities of LLMs.

5.3 Model Fine-tuning

After obtaining the unified code snippets through code abstraction and instantiation, their distribution may differ from that of the original training data since the instantiated identifiers may be different from those in the original code. Consequently, directly using the original code model to predict these unified code snippets could lead to

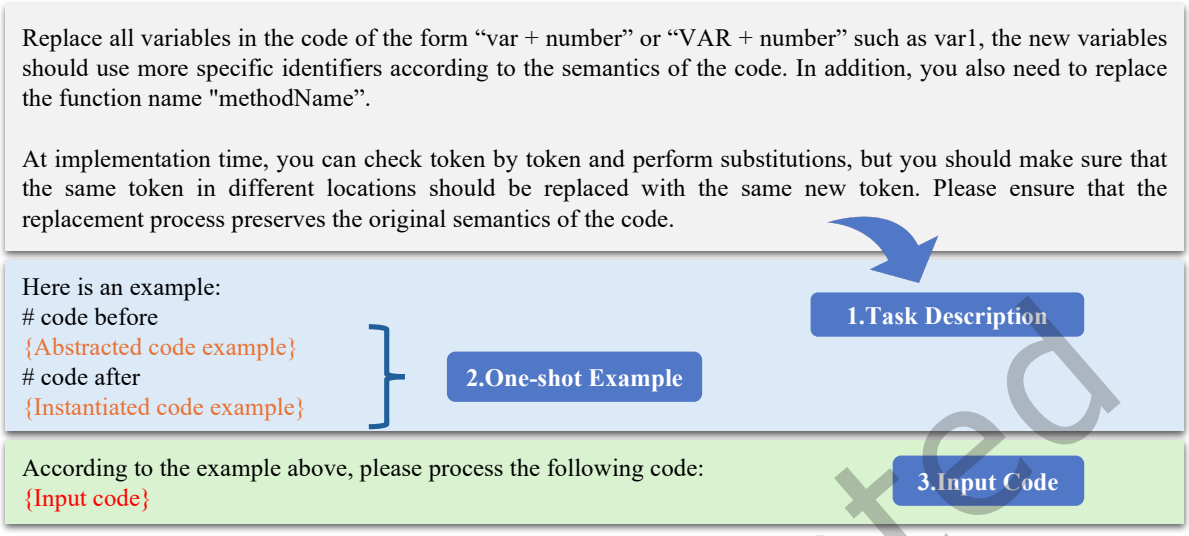


Fig. 4. Prompt for identifier generation.

performance degradation due to the OOD issue. Importantly, identifier renaming tends to introduce vulnerable features, as observed in our preliminary study. Therefore, it is crucial to ensure that the distribution of the normalized code aligns with the distribution of the model's training data. To address this, UniCode incorporates a model fine-tuning component.

Specifically, UniCode first employs the two code normalization components mentioned earlier (i.e., code abstraction and instantiation) to transform the entire training dataset for a given task into a unified distribution. Subsequently, UniCode fine-tunes the model using this normalized dataset. This approach helps the model adapt to the input feature distribution of the normalized code, effectively enhancing its prediction performance on new inputs, which are expected to be in-distribution examples due to the same normalization process.

6 EVALUATION

6.1 Research Questions

In the study, we address five research questions (RQs):

- **RQ1**: How effective is UniCode for defending diverse attacks to deep code models?
- **RQ2**: What is the contribution of each component in UniCode to its overall effectiveness?
- **RQ3**: What is the reason for UniCode's effectiveness?
- **RQ4**: How is the quality of the normalized code with renamed identifiers by UniCode?
- **RQ5**: How efficient is UniCode?

6.2 Experiment Setup

6.2.1 Benchmarks. In line with existing studies [7, 8, 24], we evaluated our approach across three different code-related tasks: clone detection, vulnerability prediction, and defect prediction. These tasks are consistent with those included in our preliminary study (Section 4), and we utilized the same datasets and training policies for this evaluation (*ref.* Table 2). Specifically, the three downstream tasks were selected based on two considerations. First, they represent the most commonly adopted evaluation tasks in the literature, spanning both well-established

methods (e.g., CODA, ALERT, CodeDenoise) and recent approaches (e.g., ITGen). This ensures that our evaluation is comparable to a broad spectrum of prior work. Second, these tasks collectively cover diverse aspects of code understanding, allowing us to assess the generalizability of our method rather than its performance on a single, narrow objective. For the datasets, we follow the standard practice in existing studies, which consistently use the same benchmark datasets for these tasks. Adopting identical datasets and configurations enables a direct and fair comparison with prior methods, avoiding potential biases that could arise from using different data splits or preprocessing steps.

Particularly, to comprehensively assess the generalizability of our approach, besides GraphCodeBert (GCBERT) and CodeT5 used in the preliminary study, we also included two additional widely-used DCMs [8]: CodeBERT [25] and CodeT5+ [27]. Furthermore, we also included a large language model to evaluate the generality of our method. We used Qwen2.5-coder-0.5B [82] in our experiment to balance efficiency and effectiveness. Specifically, CodeBERT's accuracies are 96.88%, 63.76% and 84.37% across the three tasks, CodeT5+ achieved 97.47%, 58.13% and 88.99%, while Qwen achieved 93.40%, 63.25% and 78.00%. In total, we evaluated 15 victim models (3 tasks × 5 architectures).

6.2.2 Baselines. To extensively evaluate the performance of our approach, we compared UniCode against one best-performing online detection method: CodeDenoise [24], and five offline adversarial-training methods: ALERT [7] (renaming only), ITGen [16] (renaming only), CODA [8] (renaming + structure change via mutation rules), CodeTAE [14] (renaming + structure change via transfer learning), and MARVEL [40] (enhancing model robustness via mutual learning). We selected these baselines because they (1) encompass a variety of code changes, (2) achieved state-of-the-art experimental results, (3) cover different techniques for generating adversarial examples, and (4) are open-source and can be reproduced. Consequently, they complement each other, providing a comprehensive and reliable evaluation of our approach through comparison. Please note that the baselines do not employ the code abstraction and instantiation processes in the experiment, as these processes are the core contributions of our method and are not used by existing techniques. Specifically, to ensure a fair comparison, we used the default configurations for all baseline methods and applied the same datasets, victim models, and experimental settings across them.

6.2.3 Metrics. To assess the effectiveness of our method against adversarial attacks, we adopt Empirical Robustness (ER) [83], a metric commonly employed in existing research [30, 84, 85] to measure the performance of deep learning models in defending against various attacks. ER is generally defined as the testing accuracy over adversarial inputs. Furthermore, we also report the model accuracy over the original test data and the time costs of different methods to measure their efficiency. Please note that in our experiment, we do not check the compilability of the code (before and after modification) since they typically do not constitute complete, executable programs with full compilation or runtime environments, making it infeasible to perform systematic compilation checks or execution-level behavioral validation on them. On the other hand, our method only performs a pre-processing for model prediction instead of modifying the original code. Therefore, the modification performed by our approach will have no side effects on the original code.

6.3 Implementation and Configuration

UniCode's configuration is primarily derived from its LLM choice for code instantiation and fine-tuning hyperparameters. To demonstrate the applicability of our method across different LLMs, we employed three distinct models, DeepSeek-V3 (released on 2024-12-26), GPT-4o-mini (released on 2024-07-18), and DeepSeek-Coder-V2 (released on 2024-7-18), using the first one as the default model and the latter two to assess generality. The selection is mainly based on two considerations. First, cost efficiency and scalability: compared with larger or better models, these models provide relatively controllable API costs, which better align with resource constraints in practical

Table 3. Overall effectiveness of different adversarial attack defending methods (Part I: CODA and ALERT).

Task	Model	CODA							ALERT						
		CODA	ALERT	CodeTAE	CDenoise	MARVEL	ITGen	UniCode	CODA	ALERT	CodeTAE	CDenoise	MARVEL	ITGen	UniCode
Clone Detection	CodeBERT	89.78	71.45	83.68	76.76	89.43	88.92	96.70	75.32	72.46	71.20	69.57	81.36	73.37	96.73
	GCBERT	92.97	57.69	74.91	89.96	82.09	59.31	93.59	92.31	84.29	61.38	80.00	82.35	59.31	90.66
	CodeT5	79.88	51.74	96.90	86.88	91.34	90.93	98.45	81.63	64.36	82.57	52.70	90.80	73.14	93.71
	CodeT5+	97.51	94.40	-	87.50	93.39	68.21	97.20	93.50	96.49	-	51.35	95.58	64.38	97.61
	Qwen	87.63	66.43	-	65.45	70.64	78.72	94.50	68.45	78.34	-	83.33	80.72	75.30	91.57
Vulnerability Prediction	CodeBERT	38.82	31.73	29.54	44.36	69.23	62.30	78.90	34.44	26.77	38.56	66.29	64.56	60.57	76.77
	GCBERT	34.02	24.74	42.67	52.17	68.93	59.1	70.21	23.71	17.35	45.66	64.31	69.29	65.42	78.43
	CodeT5	52.54	45.52	60.02	34.70	63.24	62.44	73.46	44.26	49.11	54.56	52.17	64.01	66.6	65.37
	CodeT5+	48.51	56.02	-	34.90	60.88	59.9	67.10	47.37	47.79	-	63.08	59.49	65.89	63.29
	Qwen	66.46	58.56	-	42.24	55.17	64.39	75.00	45.32	57.88	-	80.89	36.51	49.65	82.54
Defect Prediction	CodeBERT	85.36	68.66	37.65	25.09	56.60	69.21	78.21	83.12	75.87	56.65	16.55	67.30	82.91	89.88
	GCBERT	76.14	68.66	69.67	43.52	61.18	70.08	84.07	59.45	63.91	80.79	34.39	70.12	85.06	93.37
	CodeT5	73.83	54.18	62.64	20.78	67.95	62.03	83.20	55.77	42.22	74.86	13.96	88.24	70.29	91.90
	CodeT5+	88.09	77.31	-	21.30	48.64	61.33	84.80	83.70	88.93	-	13.35	59.54	69.82	92.20
	Qwen	64.33	58.32	-	75.46	34.97	59.89	71.17	48.65	66.34	-	75.29	63.53	60.48	77.65
Average		71.72	59.03	61.96	53.40	67.58	67.78	83.10	62.47	62.14	62.91	54.48	71.56	68.15	85.45
$\Uparrow_{Defend}$		11.38	24.08	21.14	29.70	15.53	15.32	-	22.98	23.30	22.54	30.96	13.89	17.30	-
p-value		$6.7E^{-3}$	$4.0E^{-4}$	$6.7E^{-3}$	$5.0E^{-4}$	$4.0E^{-4}$	$4.0E^{-4}$	-	$4.0E^{-4}$	$4.0E^{-4}$	$3.9E^{-3}$	$4.0E^{-4}$	$4.0E^{-4}$	$6.0E^{-4}$	-
r(rb)		0.8167	1.0000	1.0000	0.9667	1.0000	1.0000	-	0.9833	1.0000	1.0000	1.0000	1.0000	0.9500	-
CI_low		0.6200	12.8500	13.0200	11.5700	7.2700	10.6100	-	8.5000	11.3100	11.1400	8.2400	3.6600	8.3100	-
CI_high		18.5900	29.0200	40.5600	40.5500	22.8900	21.1700	-	33.9200	29.4600	33.2300	58.9800	22.5800	23.3600	-

deployment scenarios. Second, representativeness and accessibility: both models belong to widely-used modern large language model families and exhibit strong capabilities in code understanding and generation tasks [86, 87]. Therefore, we believe that this choice achieves a reasonable balance between performance and practical usability. Both models feature a context window of 128k and exhibit strong performance. In particular, we adopted the default parameters recommended by the official sources [88], setting the temperature parameter to 1.0 and the max_tokens parameter to 4096. During the model tuning process, we followed the configurations suggested in prior work [7, 8], setting block_size to 512 and max_grad_norm to 1.0. The learning rate was set to 5e-5 for clone detection and defect prediction and 2e-5 for vulnerability prediction. For baseline methods, we adopted their default settings to maintain their best performance for a fair comparison. In particular, to mitigate the impact of randomness, we repeated all our experiments three times and report the average results.

Finally, to evaluate the robustness of DCMs in our study, we utilized all four offline training methods used as baselines, namely ALERT, ITGen, CODA, and CodeTAE, as the attacking techniques for generating adversarial examples. All experiments were conducted on a server running Ubuntu 20.04, equipped with an Intel(R) Xeon(R) Silver 4214 @2.20GHz CPU and eight NVIDIA GeForce RTX 3090 Ti GPUs.

7 RESULTS AND ANALYSIS

7.1 RQ1: Overall Effectiveness of UniCode

As previously mentioned, we evaluated the overall effectiveness of UniCode on 15 different victim models by comparing it with six state-of-the-art attack-defending approaches. The results are summarized in Table 3 and Table 4. In each table, the first two columns list the tasks and model architectures, while the subsequent columns are organized into two blocks, each representing the defense results corresponding to a specific attacking approach. For instance, the first block in Table 3 displays the empirical robustness (ER) values of the seven defending methods against adversarial examples generated by CODA. For clarity, we refer to each model's defense against a particular adversarial attack as a "subject". In total, there are 54 subjects (15 victim models × 4 attacking methods - 6 failures, as CodeTAE failed to generate enough adversarial examples for CodeT5+ and Qwen due to

Table 4. Overall effectiveness of different adversarial attack defending methods (Part II: CodeTAE and ITGen).

Task	Model	CodeTAE							ITGen						
		CODA	ALERT	CodeTAE	CDenoise	MARVEL	ITGen	UniCode	CODA	ALERT	CodeTAE	CDenoise	MARVEL	ITGen	UniCode
Clone Detection	CodeBERT	83.33	61.90	67.46	55.84	71.60	79.37	98.40	90.43	72.17	87.30	72.87	94.69	90.09	96.87
	GCBERT	80.00	76.52	85.22	38.10	71.62	56.52	92.14	98.63	98.98	98.89	98.98	93.98	99.32	98.08
	CodeT5	68.75	62.50	59.38	11.76	51.61	58.73	96.77	96.76	90.82	90.37	80.89	72.64	95.05	98.33
	CodeT5+	-	-	-	-	-	-	-	95.39	80.66	-	78.26	95.10	75.51	97.10
	Qwen	-	-	-	-	-	-	-	77.52	75.65	-	79.55	83.91	85.45	94.25
Vulnerability Prediction	CodeBERT	47.59	54.48	67.13	56.76	59.17	55.52	71.89	43.38	45.12	64.21	55.31	53.58	51.84	67.68
	GCBERT	48.80	57.23	51.58	49.65	61.84	59.04	67.87	61.49	55.07	62.32	62.94	60.46	60.66	75.00
	CodeT5	49.62	45.08	56.87	39.05	49.76	41.86	64.55	47.33	47.59	67.65	60.16	66.31	73.93	74.83
	CodeT5+	-	-	-	-	-	-	-	53.01	52.06	-	66.14	63.45	67.56	63.60
	Qwen	-	-	-	-	-	-	-	43.57	39.78	-	60.49	51.85	62.33	65.43
Defect Prediction	CodeBERT	44.46	46.40	74.69	25.66	43.78	46.7	82.79	85.74	86.43	64.15	55.00	67.28	86.82	92.57
	GCBERT	54.50	47.41	75.55	33.75	48.87	63.69	84.79	78.74	81.61	78.44	84.29	75.69	88.16	94.38
	CodeT5	63.03	67.50	81.34	19.04	37.39	44.66	85.79	60.08	68.17	57.44	72.13	84.63	70.93	95.23
	CodeT5+	-	-	-	-	-	-	-	67.22	69.46	-	79.68	61.73	70.84	95.31
	Qwen	-	-	-	-	-	-	-	67.55	61.23	-	67.68	35.37	60.97	74.39
Average		60.01	57.67	68.80	36.62	55.07	56.23	82.78	71.12	68.32	74.53	71.62	70.71	75.96	85.54
$\Uparrow_{Defend}$		22.77	25.11	13.98	46.16	27.71	26.55	-	14.41	17.22	11.01	13.91	14.83	9.57	-
p-value		$2.34E^{-2}$	$2.34E^{-2}$	$2.34E^{-2}$	$2.34E^{-2}$	$2.34E^{-2}$	$2.34E^{-2}$	-	$6.0E^{-4}$	$6.0E^{-4}$	$7.8E^{-3}$	$9.0E^{-4}$	$4.0E^{-4}$	$2.3E^{-3}$	-
r(rb)		1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	-	0.9833	0.9833	0.9556	0.9500	1.0000	0.8833	-
CI_low		14.9300	15.6200	4.7600	18.2200	12.7200	16.3700	-	6.4400	12.7700	3.4700	10.0055	8.5200	3.2800	-
CI_high		30.2900	36.5000	30.9400	66.7500	45.1600	38.0400	-	24.3000	25.6500	28.4200	17.4400	18.6900	14.3400	-

its high cost). We have highlighted the best result for each subject. Additionally, the symbol $\Uparrow_{Defend}$ denotes the absolute improvement of UniCode in defense effectiveness compared with baseline approaches.

The table shows that UniCode significantly outperforms all baseline methods. Specifically, UniCode achieved the highest ER in 85.19% of the subjects (46 out of 54), with average absolute improvements ranging from 9.57% to 46.16%. In contrast, the second-best methods, CODA and ITGen, only achieved the highest ER in 4 out of 54 subjects. To further assess the significance of our approach's improvements across all victim models compared to the baseline methods, we conducted the paired Wilcoxon Signed-Rank Test [71] at a 0.95 confidence level following existing work [89]. Specifically, the unit of observation in the statistical analysis is each *task* × *victim model* combination, resulting in up to 15 paired samples (from 3 tasks and 5 models), depending on data availability. We adopt this unit for analysis as it allows us to assess whether the effectiveness of UniCode is consistently observed across diverse tasks and victim models. Please note that while analyzing each task (or model) separately would provide a more comprehensive comparison, the limited data (i.e., only five/three paired observations corresponding to the five models/three tasks) will prevent a reliable statistical analysis. As a result, we perform our statistical analysis by considering task–model combinations. The results, presented in the tables, indicate that all *p-values* (corrected by Holm–Bonferroni for multiple comparisons [90]) are smaller than 0.05, confirming UniCode's statistically superior performance relative to the baselines. In addition, we also use another three metrics to measure the effect size and significance of the difference between our method and the baselines, which are presented in the last three rows in the tables. Specifically, **r(rb)** indicates the rank-biserial [91] correlation, which measures the effect size of the paired differences, where positive values imply that UniCode consistently outperforms the corresponding baseline. $\mathbf{CI}_{low}$ and $\mathbf{CI}_{high}$ correspond to the lower and upper bounds of the 95% bootstrap confidence interval for the median difference. From the results, we can observe that the corresponding effect sizes are consistently large (with most **r(rb)** values above 0.95 and many reaching 1.0), indicating strong and stable advantages. In addition, all reported confidence intervals lie entirely above zero, further confirming the robustness and reliability of the improvements brought by UniCode. Overall, these results demonstrate that the performance gains of UniCode are not only statistically significant but also practically meaningful across different models and tasks.

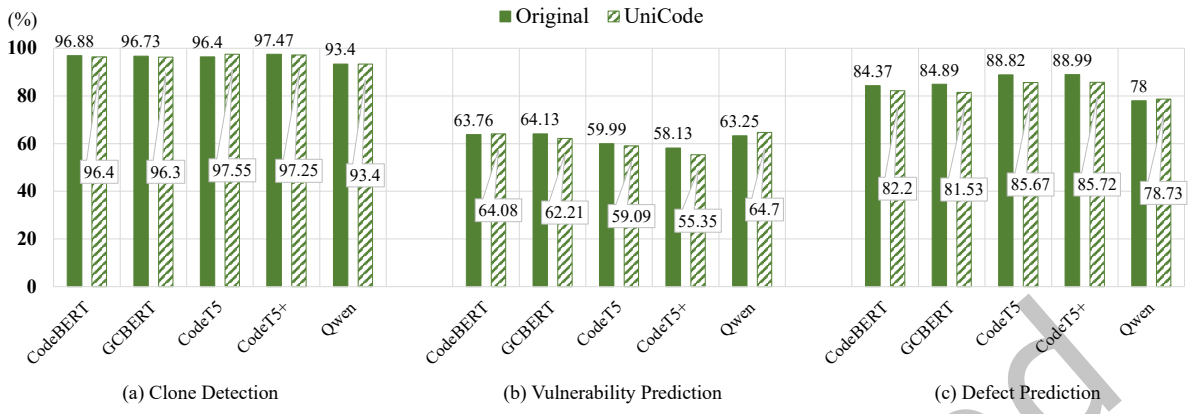


Fig. 5. Model accuracy before and after applying UniCode.

Additionally, the results reveal that existing adversarial training methods exhibit weak robustness, as nearly all of them perform worse against attacking methods different from their own, e.g., CODA defends 85.3% of its own attacks but only 37.7% of CodeTAE's in the defect prediction task using CodeBERT, and both the second-best methods (i.e., CODA and ITGen) achieved the best defense effectiveness when facing attacks from their own, suggesting adversarial training does not enhance universal robustness because limited training data cannot cover the infinite input space, leading to OOD inputs. We also notice that UniCode consistently outperforms the online detection method, CodeDenoise, across all subjects, especially against CodeTAE, as CodeTAE overlooks the naturalness of adversarial examples, resulting in a higher likelihood of modifying a larger number of identifier names. While CodeDenoise restricts itself to modifying a limited number of identifiers to avoid a search space explosion. Overall, UniCode demonstrates superior and more stable performance.

Moreover, it is essential that enhancing the robustness of DCMs against adversarial attacks does not significantly compromise their overall performance. To assess this, we compared the overall accuracy of the models on the original test data, with the results presented in Figure 5. Compared to the original model, UniCode results in a slight decrease in accuracy. On average, the model accuracy decreased from 0.817 to 0.803 across all the tasks and models. Given the significant improvement in robustness, we consider the minor decrease in the accuracy of UniCode to be acceptable.

7.2 RQ2: Contribution of Each Component

To assess the effectiveness and necessity of each component in our method, including code abstraction, code instantiation, and model fine-tuning, we developed three variants of UniCode (abbreviated as UC) and conducted an ablation study.

- **UC**$_{NCA}$: removes the code abstraction process, feeding adversarial examples directly to the LLMs for code instantiation (i.e., substituting all identifiers), and then uses the instantiated code to fine-tune the victim model.
- **UC**$_{NCI}$: removes the code instantiation process and uses the abstracted code for model fine-tuning and new-input prediction without recovering the identifier semantics.
- **UC**$_{NFT}$: removes the fine-tuning process and feeds the instantiated code to the original model directly for prediction.

The experimental results are presented in Table 5, demonstrating that each component in UniCode significantly contributes to its overall effectiveness. Specifically, the absence of the code abstraction process reduces defense performance by 0.87% ∼ 23.87% across four attack strategies. Notably, in the vulnerability prediction task using CodeBERT, the performance drop of UC_{NCA} exceeds 30% when defending against CODA. This is because CODA targets multiple identifier names within code snippets. If these attacked identifier names are not normalized and directly fed into the LLM for identifier name instantiation, the LLM struggles to accurately identify which names are genuinely vulnerable due to their implicit dependencies, thereby diminishing the defense effectiveness of UniCode. Removing code instantiation (UC_{NCI}) leads to an average performance drop of 0.41% ∼ 25.4%, especially against CodeTAE and CODA (which modify both identifiers and program structures), as altered data/control flow breaks contextual connections for semantically stripped identifiers, unlike ALERT, which only attacks identifier names while preserving structural clarity. Skipping model fine-tuning (UC_{NFT}) causes a 0.05% ∼ 28.46% average decline, as fine-tuning helps DCMs learn unified code representations and paradigmatic patterns while minimizing accuracy loss, thereby enhancing attack resistance. Overall, the ablation results confirm that all three components collectively contribute to UniCode's defense capabilities.

Additionally, to evaluate the robustness of the instantiation design, we further conducted several supplementary experiments by introducing several variations of our method.

- UC_{GPT} and UC_{DeepV2}: Two variants of UniCode by replacing the default DeepSeek-V3 model for variable instantiation with GPT-4o-mini and Deepseek-Coder-V2, respectively.
- *Random*: Rather than utilizing LLMs, it replaces the variable instantiation process in UniCode with a random strategy. Specifically, we constructed a candidate pool by extracting identifiers from the training set, and during instantiation, we randomly select identifiers that do not appear in the original code, and maintain the program dependency relations.
- *Restate*: To evaluate the sensitivity of our method to prompts, we replace our prompt with another semantically equivalent prompt variant using a controlled paraphrasing strategy, following prior studies [92, 93] on prompt sensitivity and instruction robustness.
- *Example*: To investigate the effect of the selected example, we further replace the original one-shot example with another different random one while keeping the overall prompt structure unchanged.

The experimental results are also presented in Table 5, which demonstrate that our method is robust to different instantiations. Across all tasks and models, the performance under Restate, Example, and different LLMs for variable instantiation settings remains consistently high and close to the default configuration. In many cases, these variants achieve results comparable to, or even slightly exceeding, the UniCode setting. This indicates that our approach does not rely heavily on specific prompt phrasing, example selection, or the choice of tool models, and can generalize well under different configurations. In contrast, the Random setting (leftmost column) consistently leads to substantial performance degradation across all tasks and perturbation methods. This suggests that meaningful variable name recovery is crucial for maintaining performance, and that arbitrary or semantically irrelevant naming significantly harms model behavior. In summary, the results confirm that our instantiation strategy is both effective and robust, and that the performance gains are not sensitive to specific implementation choices but instead stem from the underlying mechanism of meaningful variable name recovery.

7.3 RQ3: Reasons for the Performance of UniCode

Based on the experimental results, UniCode effectively defends against a variety of attacks and achieves significantly better universal robustness compared to existing approaches. In this section, we aim to explore the reasons behind the outstanding performance of our method. As discussed in the introduction, the key challenge in defending against diverse attacks lies in characterizing the distribution of inputs. To address this challenge, the core insight of our approach is to unify the distributions of the training and testing data through code normalization.

Table 5. Empirical robustness of models after adopting each variant of UniCode.

Task	Model	CODA									ALERT								
		Random	Restate	Example	UC_{DeepV2}	UC_{GPT}	UniCode	UC_{NCA}	UC_{NCI}	UC_{NFT}	Random	Restate	Example	UC_{DeepV2}	UC_{GPT}	UniCode	UC_{NCA}	UC_{NCI}	UC_{NFT}
Clone Detection	CodeBERT	75.94	96.63	94.04	96.85	95.65	96.70	96.63	93.55	98.13	71.97	94.83	96.25	96.33	94.15	96.73	95.10	92.79	95.51
	GCBERT	84.16	93.42	98.22	96.09	97.51	93.59	95.21	90.57	95.55	85.12	89.97	95.20	87.54	94.46	90.66	88.24	85.12	88.93
	CodeT5	83.66	98.04	98.12	98.28	97.96	98.45	97.96	94.85	97.55	83.13	92.64	98.04	94.56	96.32	93.71	90.95	88.19	90.18
	CodeT5+	80.23	96.57	96.56	97.28	95.88	97.20	95.49	91.91	95.88	73.98	97.35	96.03	98.14	96.19	97.61	96.02	93.72	96.02
	Qwen	84.40	95.41	95.41	94.50	95.41	94.50	94.50	91.74	93.25	79.52	92.77	91.57	93.98	91.57	91.57	90.36	86.75	90.57
Vulnerability Prediction	CodeBERT	57.33	79.21	79.58	65.78	62.82	78.90	55.03	64.58	71.88	56.49	78.06	77.95	70.12	71.24	76.77	70.77	74.37	62.05
	GCBERT	55.97	78.28	77.71	67.57	70.10	70.21	62.20	61.82	66.61	61.42	78.43	78.70	73.03	71.17	78.43	74.68	75.13	70.41
	CodeT5	56.53	74.68	74.11	65.78	62.91	73.46	62.27	64.33	66.02	55.74	67.67	68.96	61.19	59.61	65.37	61.67	63.13	60.51
	CodeT5+	50.97	67.42	67.23	61.59	62.09	67.10	61.59	61.59	64.38	55.59	63.40	60.87	58.23	57.59	63.29	57.17	55.59	55.49
	Qwen	59.48	73.91	75.86	74.14	75.86	75.00	68.97	62.07	60.34	65.08	82.54	79.37	80.95	79.37	82.54	79.37	69.84	50.79
Defect Prediction	CodeBERT	55.24	78.99	78.74	64.30	61.65	78.21	60.09	65.35	70.95	62.45	87.11	87.45	83.79	82.70	89.88	80.33	87.39	79.20
	GCBERT	63.92	83.86	83.46	73.36	70.43	84.07	74.27	72.52	81.41	77.58	90.76	90.64	89.17	87.19	93.37	86.38	90.86	86.79
	CodeT5	52.40	83.26	82.94	70.90	67.95	83.20	68.82	67.25	64.87	64.63	90.61	90.18	87.91	83.65	91.90	84.54	85.53	79.72
	CodeT5+	41.69	84.99	84.45	75.21	72.21	84.80	76.73	74.15	68.48	52.88	90.54	90.23	88.15	84.99	92.20	85.01	88.15	78.69
	Qwen	61.35	70.55	73.62	74.62	69.33	71.17	68.10	61.35	64.92	65.88	77.75	78.82	81.18	82.35	77.65	71.76	65.88	66.85

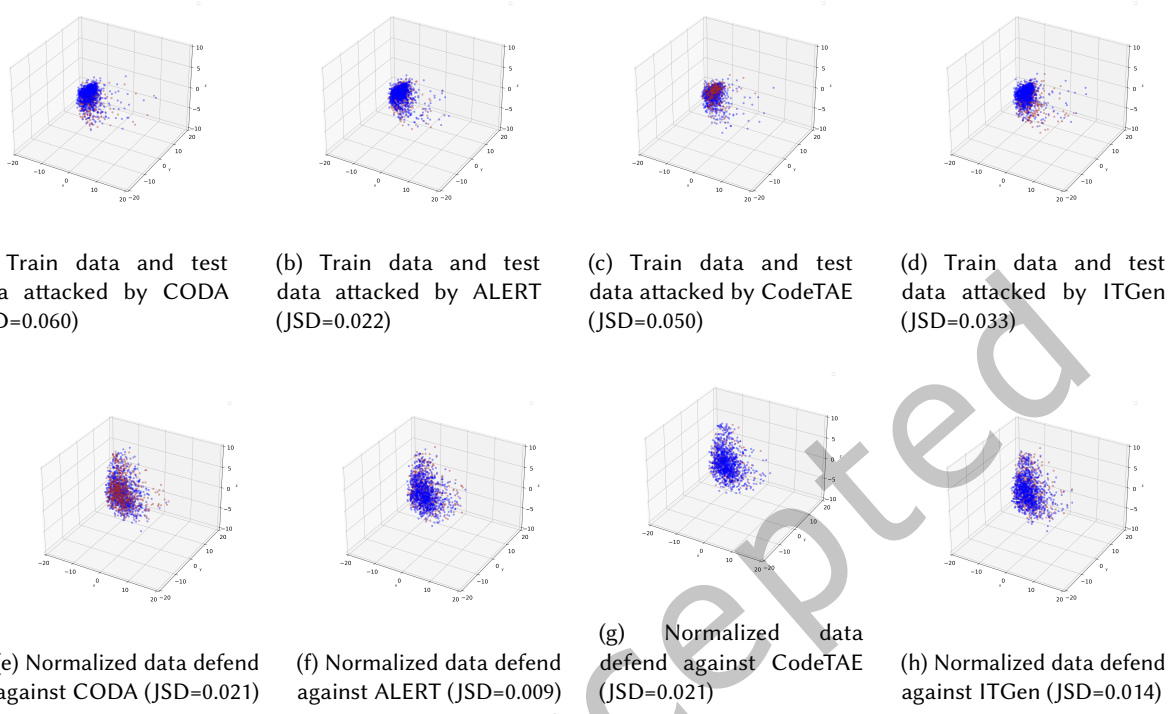
Task	Model	CodeTAE									ITGen								
		Random	Restate	Example	UC_{DeepV2}	UC_{GPT}	UniCode	UC_{NCA}	UC_{NCI}	UC_{NFT}	Random	Restate	Example	UC_{DeepV2}	UC_{GPT}	UniCode	UC_{NCA}	UC_{NCI}	UC_{NFT}
Clone Detection	CodeBERT	70.80	97.20	94.80	97.60	94.80	98.40	98.40	86.40	97.20	69.80	96.94	96.46	97.48	93.65	96.87	96.00	84.33	96.40
	GCBERT	78.17	91.70	91.27	91.70	95.63	92.14	91.70	87.77	93.01	97.22	97.86	98.09	98.08	98.72	98.08	97.91	97.61	97.95
	CodeT5	66.13	98.39	96.77	95.16	100.00	96.77	98.39	74.19	93.55	82.65	97.89	97.81	98.44	98.22	98.33	98.04	86.65	97.44
	CodeT5+	-	-	-	-	-	-	-	-	-	75.16	96.55	95.62	97.28	94.74	97.10	95.92	88.21	97.05
	Qwen	-	-	-	-	-	-	-	-	-	68.97	94.25	96.55	95.40	95.40	94.25	93.10	78.16	93.25
Vulnerability Prediction	CodeBERT	62.98	74.61	75.56	67.13	67.30	71.89	64.36	71.45	43.43	59.39	70.87	70.86	62.11	63.52	67.68	62.11	66.02	65.94
	GCBERT	54.45	65.61	72.05	65.91	63.60	67.89	65.76	66.21	65.46	62.80	73.94	75.25	69.53	66.74	75.00	71.40	70.78	73.96
	CodeT5	38.39	65.94	67.55	65.12	55.83	64.55	64.36	39.15	63.32	65.15	75.67	75.95	70.84	63.01	74.83	67.09	64.95	73.29
	CodeT5+	-	-	-	-	-	-	-	-	-	54.24	60.87	62.55	57.96	59.38	63.60	55.03	58.12	58.37
	Qwen	-	-	-	-	-	-	-	-	-	53.09	65.43	64.20	62.96	61.73	65.43	60.49	59.26	56.79
Defect Prediction	CodeBERT	54.90	80.84	81.14	78.31	71.82	82.79	81.43	73.54	75.71	64.58	89.67	89.65	86.28	85.03	92.57	80.69	92.16	80.50
	GCBERT	56.68	85.14	85.59	82.36	77.06	84.79	83.95	80.82	79.09	80.39	91.25	91.10	89.46	87.47	94.38	85.34	92.25	85.57
	CodeT5	56.01	85.96	86.13	84.77	80.20	85.79	84.77	78.34	69.54	62.22	92.12	92.10	90.74	88.18	95.23	86.88	94.59	80.89
	CodeT5+	-	-	-	-	-	-	-	-	-	56.13	92.11	92.08	90.82	87.89	95.31	86.42	94.27	81.43
	Qwen	-	-	-	-	-	-	-	-	-	64.63	73.78	73.17	79.88	75.46	74.39	73.78	57.32	67.97

To investigate whether our approach can effectively align the testing data—particularly the attacked inputs—with the distribution of the model's training data, we analyzed the discrepancies in data distribution. We extracted feature vectors from both the training data and the attacked testing data using a deep code model, visualizing their distributions in the feature space. This analysis allowed us to examine the distribution discrepancies before and after applying our approach (i.e., code normalization and fine-tuning) for a clearer understanding.

We fed both the training data and the adversarial testing data into the trained code model to analyze their feature distributions and understand why the model tends to produce incorrect predictions. Additionally, we input the normalized training data and the normalized adversarial testing data into the fine-tuned code model to assess whether our approach effectively aligns the distribution of the adversarial inputs with that of the training data. Given that the data in the feature space is high-dimensional, we applied Principal Component Analysis (PCA) [94] to reduce the dimensions to three, enabling us to visualize the feature distributions clearly. Given similar trends, we selected the vulnerability prediction task with GraphCodeBERT (GCBERT) as the representative.

Figure 6 visualizes the data distributions. The upper subfigures present the distribution characteristics of two types of data: the original training data and adversarial examples generated by four attack methods (i.e., CODA, ALERT, CodeTAE, and ITGen). By contrast, the lower subfigures display the distributions of these two types of data after normalization by our proposed approach, UniCode. To quantify the discrepancy between these distributions, we adopt the Jensen-Shannon Divergence (JSD) [95] as the evaluation metric, which is also presented in Figure 6. In our experimental context, JSD is used to measure the distributional distance between the two groups of data mentioned above. It compares the training data and adversarial test data before and after applying our approach. Notably, a smaller JSD value indicates a closer alignment between the two distributions being compared.

From Figure 6, we can observe that the data points before normalization are more separated, while those after normalization are more intertwined. It can also be seen that the JSD values in the upper subfigures are



(a) Train data and test data attacked by CODA (JSD=0.060)

(b) Train data and test data attacked by ALERT (JSD=0.022)

(c) Train data and test data attacked by CodeTAE (JSD=0.050)

(d) Train data and test data attacked by ITGen (JSD=0.033)

(e) Normalized data defend against CODA (JSD=0.021)

(f) Normalized data defend against ALERT (JSD=0.009)

(g) Normalized data defend against CodeTAE (JSD=0.021)

(h) Normalized data defend against ITGen (JSD=0.014)

Fig. 6. Visualization of data distributions in the task of vulnerability prediction using the model of GraphCodeBERT. The points in blue (red) denote the training (attacked testing) data before and after normalization.

consistently larger than those in their corresponding normalized counterparts. This indicates that the original training data and adversarial test data exhibit a more distinct separation, and our approach can normalize these data points to make their distributions more similar. This normalization defends against adversarial attacks and thus improves model robustness. Specifically, against the CODA attack, we decrease the distributional distance between adversarial examples and training data from 0.060 to 0.021, representing a relative reduction of 60.0%. For the ALERT, CodeTAE, and ITGen attacks, relative reduction ranges from 57.6% to 59.1%. This explains the effectiveness of our approach: after applying UniCode, the distributions of normalized training data and normalized adversarial data are closely intertwined in the feature space, thereby transforming out-of-distribution (OOD) inputs into in-distribution (ID) ones. This significant reduction in JSD values demonstrates that UniCode effectively aligns the distribution of adversarial data with that of training data.

Interestingly, we find that our method also exhibits a certain defensive effect against adversarial examples generated solely by structure-related attacks, though UniCode is designed for defending identifier renaming attacks. For example, since we did not restrict the attacking strategies used by existing methods in our experiments, CODA actually crafted 367 adversarial samples through purely structure-related transformations for the Clone Detection task of CodeT5, and our method effectively mitigates 98.09% of these attacks. One possible reason for this phenomenon may be that the DCM's understanding of code relies heavily on the combined effect of identifier semantics and structural features; that is, identifier names and code structures are not independent but coupled feature dimensions. By normalizing identifier names, our method breaks this coupling relationship,

Table 6. Statistics of cases having code naturalness changed after applying UniCode

Task	#Samples ⇑	#Samples ⇓
Vulnerability Prediction	1,747	985
Defect Prediction	5,457	1,307
Clone Detection	7,387	1,739
Total	14,591	4,031
Average Percentage	78.35%	21.65%

forcing the model to focus on the core semantics and essential logic of the code. Consequently, attacks that rely solely on structural transformations lose the ability to mislead the model and thus fail to take effect.

7.4 RQ4: Quality of Normalized Code by UniCode

While UniCode's normalized code is designed for direct use by DCMs without human intervention, its quality remains important from a developer's standpoint. In addition, understanding this quality is crucial for assessing UniCode's potential effectiveness and usability in scenarios that involve developers. Therefore, we conducted a combined quantitative and qualitative analysis of the generated code.

In our quantitative analysis, we measure code quality by assessing its *naturalness*, a metric that captures code repetitiveness and predictability, reflecting its alignment with human-written code [96]. It is important to note that our approach preserves both the structure and semantics of the original code since the LLM is instructed to replace the same identifiers with consistent names, which maintains data dependency. Consequently, many traditional quality metrics, such as code complexity [97] and bug density [98], are inapplicable. Following prior studies [99, 100], we use Cross-Entropy (CE) [96] to quantify *naturalness*. This metric assigns a naturalness score to a given code sample, where a lower score indicates higher naturalness. Specifically, we computed CE scores for code samples both before and after normalization across all three testing tasks. The results are presented in Table 6, where #Sample⇑ denotes the number of cases with improved naturalness while #Sample⇓ denotes the number of cases with decreased naturalness. Our results demonstrate that approximately 78.35% of normalized code instances achieve lower CE scores, indicating that the normalization process of our tool tends to improve rather than sacrifice code naturalness. For the minority of instances where naturalness decreased, we further performed a manual analysis of these cases before and after applying our approach. This revealed that the original code often included well-designed identifiers that effectively conveyed domain-specific knowledge. In these cases, the new identifiers generated by the LLM sometimes failed to accurately capture this semantic meaning due to the lack of domain knowledge and limitations in semantic interpretation. Nonetheless, the overall quality of the normalized code is largely maintained.

Furthermore, to better understand the effectiveness of our approach, we also analyzed the correlation between code naturalness and its attack defense effectiveness. Since a smaller CE value means more natural code, we use the negative CE value so that larger values show higher naturalness. For attack defense effectiveness, we mark each adversarial example as 1 if the prediction is correct and 0 otherwise. To analyze the relationship between a continuous variable and a binary variable across different tasks, we use the common odds ratio (OR) from logistic regression [101]. This measures the correlation between the negative CE value and attack defense effectiveness, which is a binary value of 0 or 1. Specifically, an OR value near 1 means no correlation. An OR greater than 3 or smaller than 0.33 means a strong correlation. As shown in Table 7, OR values range from about 0.7 to 1.8 in most cases. This shows that code naturalness has a moderate correlation with the performance of our approach, which further explains the slight performance discrepancy of our method when utilizing different LLMs (see

Table 7. Odds ratio between code naturalness and attack defense capability

Tasks	CodeBERT	GCBERT	CodeT5	CodeT5+	Qwen
Defect Prediction	0.5978	1.0569	1.7744	1.2082	1.7636
Vulnerability Prediction	1.0396	1.0277	1.0835	0.9970	0.9912
Clone Detection	0.9170	0.8977	0.8092	0.7482	0.3416

Table 5). However, the results also suggest that the correlation across different models and different tasks can vary. This phenomenon attributes the differences in how models understand code naturalness or task-specific feature distributions, which calls for further exploration.

In our qualitative analysis, we surveyed 10 experienced developers to evaluate code quality manually. Specifically, following existing work [102], we adopted convenience sampling [103] to contact developers within our network via email and telephone. During recruitment, we selected developers based on two criteria: (1) participants must have at least three years of programming experience, and (2) participants should have a solid understanding of at least one programming language (such as Java or C++). Ultimately, 10 experienced developers were selected for the survey. These developers represent a diverse group: they come from both industry and research backgrounds, with working experience ranging from three to eight years, and are proficient in multiple programming languages, including C, C++, and Java. Their demographic information is listed in Table 8.

To ensure reliable and representative results, we randomly selected 30 pairs of original and normalized code by our approach. Each developer independently compared the *naturalness* of each pair without knowing which version was normalized. They were asked to select the code sample with higher naturalness from each pair. Detailed experimental results are presented in Table 8, which reports the number of times each participant (P) rated our normalized code samples as more natural across all 30 pairs. The results show that for each pair, a large majority of participants consistently rated the normalized code as more readable, underscoring the practical effectiveness and value of our approach for developers. As shown in Table 8, six out of the ten participants judged our normalized code samples as more natural in all 30 pairs. Another two participants gave near-perfect evaluations of our method, rating the normalized code samples as more readable in 28 and 29 pairs, respectively. We also notice that one participant ($P3$) adopted a strict evaluation standard; through follow-up interviews, we learned that this participant preferred naming strategies with extreme conciseness as the most natural. Even so, this participant still rated our normalized identifier names as more concise and readable in over 50% of the pairs.

To further strengthen the reliability of the qualitative study, we analyze inter-rater agreement using multiple metrics, including Fleiss' Kappa [104], mean raw agreement [105], and Gwet's AC1 [106]. The results are as follows: Fleiss' Kappa = -0.0246, mean raw agreement = 0.9267, and Gwet's AC1 = 0.8388. While the mean raw agreement indicates a high level of consensus among raters, Fleiss' Kappa yields a low (and slightly negative) value. We analyze this discrepancy and find that it is caused by a highly skewed label distribution in the annotations: across all responses, 278 selections favored UniCode (i.e., participants judged our method as producing higher-quality identifiers), while only 22 selections favored the original code. Such an imbalance leads to the well-known kappa paradox [107], where Kappa underestimates agreement when one category dominates. In contrast, Gwet's AC1 is more robust to this issue. The high AC1 score (0.8388) provides a more reliable indication of strong inter-rater agreement, supporting the validity of our qualitative findings.

7.5 RQ5: Efficiency of UniCode

To evaluate the efficiency of our approach, we recorded the execution time for each method. Different methods involve distinct processes. For instance, adversarial training requires generating adversarial examples for model

Table 8. Demographics of experts and expert naturalness rankings of original vs. normalized code

Expert ID	Profession	Working Years	Proficient Languages	#1st	#2nd
*P*1	Master Student	3	C,C++,Java	30	0
*P*2	Industry Developer	3	C,C++,Java,Python	30	0
*P*3	Industry Developer	3	C,C++,Java,Python	16	14
*P*4	Master Student	5	C,C++,Java	30	0
*P*5	Master Student	3	C,C++,Java	30	0
*P*6	Industry Developer	7	C,C++,Java,Python	29	1
*P*7	Doctoral Student	7	C,C++,Java	28	2
*P*8	Industry Developer	8	C,C++,Java,Python	25	5
*P*9	Doctoral Student	3	C,C++,Java	30	0
*P*10	Doctoral Student	5	C,C++,Java	30	0

Table 9. Time cost of defending methods (in minutes).

Process	CODA	ALERT	CodeTAE	ITGen	MARVEL	CodeDenoise	UniCode
Offline Generation	7,685.95	37,177.69	470,917.60	13769.51	-	51.60	1,930.40
Offline Fine-tuning	533.67	544.67	562.17	558.54	1,705.00	87.50	586.00
Online Refinement	-	-	-	-	-	0.02	0.10

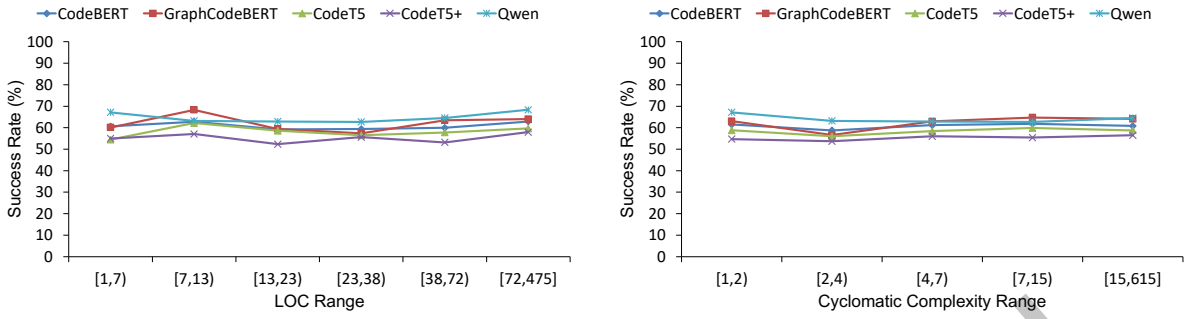
fine-tuning, and the online method CodeDenoise necessitates on-the-fly input refinement. Considering this, we provided the average execution time for each process across the three tasks. The results are summarized in Table 9, where we present the time costs associated with each approach in our experiment. Notably, the offline fine-tuning for CodeDenoise refers to training an additional identifier generation model rather than training the original code model like the other methods.

From the table, we observe that the efficiency of UniCode is comparable to that of the baseline methods. Specifically, UniCode tends to spend more time on input refinement during the online prediction process. This additional time is primarily due to the calls to LLMs for code instantiation, which accounts for approximately 85.71% of the total time. This aspect could potentially be optimized by utilizing a more efficient LLM. In contrast, the adversarial training process in existing approaches is significantly more time-consuming due to the need to generate adversarial examples. For instance, CodeTAE requires over 2,000 hours. UniCode demonstrates greater efficiency in this regard. While CodeDenoise is relatively more efficient than the other methods, it achieves poorer universal robustness (as shown in Table 3 and Table 4). In summary, the time cost of UniCode is comparable to that of the baseline methods and is acceptable given its outstanding effectiveness in defending against various attacks.

8 DISCUSSION

8.1 Generality of Our Approach

In this paper, we focused on evaluating the effectiveness of UniCode in defending against adversarial attacks on classification tasks of DCMs by following existing studies [7, 8, 14]. However, UniCode's generality in components and core insights enables its application to broader tasks and attack scenarios. For instance, in generation tasks by code LLMs, e.g., code summarization or code review, UniCode's code abstraction and instantiation process can be



(a) Regarding code lengths (LOC).

(b) Regarding cyclomatic complexity intervals.

Fig. 7. Model accuracy using UniCode over cases with different code complexities on Vulnerability Prediction.

directly applied to code snippets embedded in inputs, while the fine-tuning stage allows a flexible choice between full-scale fine-tuning and Low-Rank Adaptation [108] based on model scale, making it suitable for adversarial defense in code LLMs.

Moreover, we extend UniCode to larger LLMs to verify its generality and conduct relevant evaluations. Specifically, we have selected CodeLlama-7B and the Vulnerability Prediction Dataset as representative objects for our experiments. The reasons for selecting them are two-fold: First, most existing studies [7, 8, 14–16] conduct experiments on smaller language models such as CodeBERT and CodeT5, which are primarily designed for classification tasks. These methods, however, cannot be directly applied to LLMs, as LLMs are mainly built for generative tasks. Secondly, we found that current testing or attack approaches work inefficiently when adapted for use with LLMs. For instance, when applying the ALERT method to CodeLlama-7B with the Vulnerability Prediction Dataset, the offline generation of adversarial examples takes 1,654 hours. This duration is approximately three times that of smaller models such as CodeT5 and CodeBERT. The offline fine-tuning time is also approximately 3,817 minutes, which is nearly seven times the duration of smaller models. Considering the high execution cost, we chose CodeLlama-7B as the victim model, Vulnerability Prediction as the attacking task, ALERT as the attacking method, and ALERT and CodeDenoise (it is efficient as it does not require fine-tuning the model) as the baseline defenders in this experiment. We consider extending our approach to more LLMs as part of our future work.

Our experimental results show that after adversarial fine-tuning with ALERT, the ER value of the CodeLlama-7B model is 58.82%, the ER value of CodeDenoise is 60.59%, while the ER value of our UniCode reaches 64.83%. This represents a relative performance improvement of 10.21% and 7.00%. We also observed an improvement in the LLM's prediction accuracy: UniCode increases CodeLlama-7B's accuracy from 65.12% to 66.11%, and CodeDenoise increases accuracy from 65.12% to 65.43%. In contrast, adversarial training with ALERT reduces it to 64.75%. These results confirm that our method can enhance the model's robustness against adversarial attacks without compromising its overall prediction performance. This further demonstrates the generality of our approach to larger models.

Finally, since our method depends on LLMs to instantiate the abstracted variables and LLMs' performance may be sensitive to input length, to investigate the generality of our approach across cases with different code complexities, we analyzed the effectiveness of UniCode across different code complexities from two complementary perspectives, namely Lines of Code (LOC) and Cyclomatic Complexity. Specifically, we also took the Vulnerability Prediction task as the representative example, and we partition the test samples into quantile-based intervals according to each metric and report the model accuracy within each interval. Please

Table 10. End-to-end attack success rate of CODA before and after using UniCode.

Tasks	CodeBERT		GCBERT		CodeT5		CodeT5+		Qwen	
	Before	After	Before	After	Before	After	Before	After	Before	After
Clone Detection	40.0%	10.0%	18.0%	1.0%	31.0%	7.0%	34.0%	9.0%	48.0%	11.0%
Vulnerability Prediction	57.0%	19.0%	57.0%	19.0%	62.0%	18.0%	51.0%	16.0%	56.0%	16.0%
Defect Prediction	82.0%	16.0%	77.0%	16.0%	82.0%	17.0%	76.0%	18.0%	79.0%	13.0%

note that this process can mimic any identifier-renaming attack methods, as all identifiers will be first removed by our abstraction process, regardless of whether they are modified or not.

The experimental results are shown in Figure 7. From the figures, we can observe that the overall effectiveness of our approach is relatively stable and not sensitive to the complexity of the input code. While minor fluctuations can be observed across intermediate intervals, there is no clear decreasing trend as code complexity (either length or cyclomatic complexity) increases. In some cases, the performance even slightly improves for longer inputs. For example, GraphCodeBERT and Qwen exhibit comparable or higher accuracy in the largest LOC interval compared to smaller ones. This suggests that longer code snippets do not pose additional difficulty for UniCode. In summary, our approach can achieve relatively stable effectiveness across cases with different code complexities, indicating the reliability and practicality of our approach.

8.2 Analysis of Potential Data Leakage

In this section, we explore whether there exists potential data leakage (or memorization) during the code instantiation process. To evaluate whether identifiers generated during instantiation are only simple reuse or minor modifications of the original identifiers, we conduct an additional comparative experiment between the original test-set code and code instantiated by our method. Specifically, we measure the consistency of all identifiers before and after instantiation and calculate the identifier change rate, which refers to the proportion of identifiers that differ from the original ones after instantiation. The rationale of this analysis is as follows. If the model only reuses the original identifiers during instantiation, potentially due to data leakage or memorization, the generated identifiers will be highly consistent with the original ones, resulting in a low change rate. In contrast, if the model can generate new identifiers from the abstract representation while maintaining semantic consistency but using different forms, a higher change rate will be observed. Therefore, measuring the identifier change rate allows us to indirectly determine whether the model only reproduces the original identifiers. A great change rate indicates that the generated identifiers do not come from memorization or direct copying. Instead, they are newly generated based on semantic and contextual information. Specifically, the average identifier change rate is 78.66% for Clone Detection, 84.46% for Defect Prediction, and 66.25% for Vulnerability Prediction. These results indicate that the instantiated identifiers differ from the original ones in over 60% of cases. This suggests that the large language model mainly regenerates identifiers based on code semantics during instantiation, rather than simply copying or slightly modifying the original identifiers. This result also demonstrates that our method does not rely on memorization or data leakage to some extent.

8.3 UniCode Against Adaptive Attack

To further evaluate the robustness of UniCode under a stronger threat setting, we conduct an additional adaptive attack experiment that simulates an end-to-end attack scenario, where the adversary targets the entire pipeline (i.e., UniCode + downstream model). In this setting, the attacker is assumed to be aware of the presence of UniCode and attempts to generate adversarial examples directly against the defended system. Due to computational cost

and API usage constraints, we select CODA (the best-performing attack method based on Table 3 and Table 4 with acceptable computation overhead) as a representative attack method and integrate its attack process with the UniCode pipeline. In addition, we conduct the experiment on a randomly sampled subset of 100 instances from the original test set to ensure feasibility under API budget constraints while maintaining representative evaluation coverage.

Under this more realistic end-to-end attack scenario, we re-evaluated the model performance across multiple tasks. Table 10 reports the attack success rates (ASR) of CODA on the original models and its adapted version incorporating UniCode's defending pipeline. We observe that the attack success rates are substantially reduced after incorporating UniCode across all tasks and models. For example, on Clone Detection, the success rate drops from 40% to 10% for CodeBERT and from 18% to 1% for GraphCodeBERT. Similar trends are observed in Vulnerability Prediction and Defect Detection. Overall, the adaptive attack remains less effective against the UniCode-enhanced system, with reductions of 66.67% $\sim$ 94.44% relative decrease in attack success rates. These results indicate that UniCode remains effective against the evaluated adaptive-CODA attack and is not easily circumvented by this end-to-end attack adaptation. However, exploring its robustness under a broader range of end-to-end adaptive attack scenarios remains an important direction for future work.

8.4 Analysis of Potential Failures

While the experimental results demonstrate the effectiveness of our method, we conducted an in-depth analysis of potential errors to better understand its performance and identify directions for further improvement. Specifically, we analyzed three aspects: (1) errors from the code abstraction process, (2) errors from the code instantiation process in UniCode, and (3) cases where our method failed to defend.

For the code abstraction process, we examined whether identifiers are replaced appropriately. This involves two criteria: (a) all identifiers should be replaced by predefined placeholders while other code elements remain unchanged; (b) identical (or different) identifiers appearing in different locations should be replaced by the same (or different) placeholder. To verify this, we performed a static analysis on the abstract syntax trees (ASTs) before and after abstraction for all studied cases. The results show that all cases were transformed correctly. That is, every necessary variable was replaced with the proper placeholder while preserving code dependency relations.

Regarding the code instantiation process, placeholders in the abstracted code are replaced by LLMs. Even though we instructed the model to preserve code semantics (e.g., replacing identical placeholders with identical tokens), errors may still occur. To analyze this, we compared the instantiated code generated by LLMs with the original code by traversing their ASTs. The results indicate that the instantiated code is always semantics-preserving, preserving both syntactic structure and data dependency relations without any alteration. This suggests that LLMs are capable of precisely identifying variable correspondences for practical use.

Finally, we analyzed cases where our method failed to defend. Our observations suggest that while the two-stage code normalization process in UniCode preserves functional consistency, the instantiated identifiers do not always capture the semantic nuances of the original code. For example, in one case, the non-vulnerable variable `cache` was instantiated as `outputDir`. Although `outputDir` correctly reflects that the variable refers to a directory used for storing extracted artifacts, it fails to preserve the crucial caching semantics of `cache`, which imply reuse and lifecycle management rather than merely output behavior. The same phenomenon was observed in all failure cases.

In conclusion, although our approach performs strongly in most cases, the example above suggests that even when operational behavior is preserved, the higher-level semantic meaning and developer intent embedded in identifier names may drift after instantiation. This represents an inherent limitation of identifier-based transformations, and addressing such cases remains an avenue for future work.

9 Implications

From the attacker's perspective, our results suggest that adversarial perturbations based on static identifier manipulation become significantly less effective under normalization-based defenses. This indicates that simply relying on conventional identifier-level transformations may no longer be sufficient. Future attack strategies may need to shift towards more complex perturbations, such as stronger structural transformations or leveraging runtime information (e.g., dynamic variables) to construct more resilient adversarial examples, in order to remain effective against normalization mechanisms.

From the defender's perspective, our findings provide a new perspective on improving robustness for code models. Rather than relying solely on adversarial training or localized correction mechanisms, our results suggest that modifying the input distribution through semantic-preserving normalization can be an effective complementary strategy. This highlights a promising direction for future defenses, where robustness is achieved by reducing sensitivity to unstable input representations at the distribution level, instead of only reacting to specific adversarial patterns.

10 Limitations and Future Work

Code Style Preservation: While UniCode improves robustness through an abstraction–instantiation mechanism that preserves program semantics, this design also introduces certain limitations. In particular, UniCode may be less effective for tasks that rely heavily on stylistic features of code, such as authorship attribution. This is because, during the abstraction stage, fine-grained stylistic information (e.g. coding styles) is largely removed, and such information is difficult to fully recover during instantiation. As a result, although semantic information is well preserved, stylistic cues that are critical for these tasks may be weakened. Therefore, UniCode is not expected to provide significant benefits for style-sensitive tasks. Exploring how to balance semantic robustness with the preservation of stylistic information remains an important direction for future work.

Static Analysis Limitation: The abstraction stage of UniCode relies on static program analysis based on AST and DFG representations. As a result, its effectiveness is inherently constrained by the limitations of static analysis techniques. For code with well-defined static structures (e.g., conventional variable definitions and usages), the abstraction process can accurately identify and normalize identifiers. However, for cases involving dynamically generated identifiers or variables that depend on runtime behaviors, static analysis may not fully capture the underlying semantics, which limits the applicability of our approach in such scenarios. In addition, our current evaluation does not explicitly consider macro-related constructs. We note that existing adversarial attack and defense methods (including CODA, ITGen, CodeTAE, and so on) do not involve perturbations at the macro expansion level. Therefore, this aspect is not covered in our experiments. Extending the proposed approach to handle more complex code constructs, such as macros or other forms of preprocessing-level transformations, remains an important direction for future work.

Cross-Function Modeling: We currently focus on tasks involving individual methods like existing studies, i.e., taking a single method as input for model prediction, while handling long-context inputs containing multiple functions, which poses additional challenges for semantic-preserving transformations. In such scenarios, the model is required to aggregate semantic information across a broader context to generate consistent and meaningful identifiers, which increases the difficulty of maintaining coherence across different code regions. To address this limitation, several potential directions should be explored for future work. First, incorporating long-context modeling techniques [109, 110] may improve the model's ability to capture global semantic dependencies across multiple functions. Second, leveraging more capable large language models could enhance cross-functional reasoning and improve the quality of generated identifiers. Third, introducing more fine-grained constraints during the abstraction stage may help ensure consistency and correctness during instantiation. We leave this as an important direction for future work.

Limited Threat Models: The current study focuses on a specific threat setting in which adversaries modify code identifiers without accessing model parameters. Nevertheless, real-world code inputs can exhibit far greater diversity in formatting, which may exceed the scope of our current evaluation, despite the fact that our examined attack scenarios already incorporate minor structural modifications. Moreover, downstream tasks may expose LLMs to various input types that are not limited to source code. Such factors could significantly undermine the generalizability of our method, underscoring the need for more robust and tailored defenses. We believe these challenges open up valuable avenues for future investigation.

11 Threats to Validity

The **internal** threats to validity mainly lie in the implementations of baseline methods and UniCode. For the baseline methods, we directly adopted their open-source implementations offered by previous studies. As for the adversarial training process, we re-implemented it using the recommended configurations from the corresponding papers. To minimize the risks associated with the implementation of UniCode, three authors meticulously reviewed all the code. Moreover, our method relies on LLMs to instantiate variable placeholders. Despite explicit instructions for the LLM to map identical placeholders to consistent identifiers, the risk of hallucination-induced mismatches remains non-negligible. To mitigate this, a more robust validation mechanism, for instance, execution-based verification such as regression testing or runtime equivalence, is required to guarantee result validity. Given that our current dataset does not include such test cases, we intend to extend our evaluation to a broader benchmark in future work. Lastly, we have made all implementations publicly available to support replication efforts.

The **external** threats to the validity of our study stem from our choices regarding datasets, models, and downstream tasks. To rigorously evaluate the performance of our approach, we utilized three widely used datasets, each associated with a distinct task and involving four different model architectures. However, large language models (LLMs) are rapidly becoming the standard in real-world practice. Although we included one LLM to preliminarily assess the generalizability of our method, the scale and diversity of the LLMs examined remain insufficient. This limitation is particularly critical, as the choice of underlying model can substantially influence dataset performance and task outcomes, especially given the growing risk of data leakage. Consequently, our current findings warrant further validation through more extensive evaluations. In future work, we plan to broaden our experimental framework to include a wider range of models, tasks, and datasets.

The **construct** threats to validity primarily concern measurements and randomness. For measurements, we used common metrics from previous research, such as empirical robustness and accuracy, to assess the effectiveness of our approach. To address the issue of randomness, we repeated each experiment three times and conducted statistical analyses, ensuring the reliability of results and mitigating the impact of randomness. In addition, our developer survey in RQ4 depends on the experience and expertise of participants, which may affect the generality and reliability of the results. To mitigate this, we also adopted the commonly used Cross-Entropy to measure code quality, which can provide a fairer comparison and give us more confidence about the conclusions.

12 CONCLUSION

In this paper, our preliminary study reveals that identifier renaming is crucial in attacks against DCMs. According to this finding, we propose UniCode, a method designed to enhance model robustness and enable it to better defend against identifier renaming attacks. Specifically, UniCode first removes all identifier names vulnerable to attack through code abstraction. Then, by harnessing the code understanding capabilities of LLMs, UniCode replaces these removed identifiers in the abstracted code with clean, semantically meaningful names. We evaluated UniCode across 1260 cases. The results show that UniCode outperforms other methods on 85.19% of the subjects.

Compared with baseline approaches, UniCode improves defense effectiveness by an average of 9.57% to 46.16%, demonstrating its superior performance in protecting DCMs against various adversarial attacks.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (Grant No. 62572344), the Beijing-Tianjin-Hebei Natural Science Foundation Cooperation Project (Grant No. 25JJJJC0030), the National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore and Cyber Security Agency of Singapore.

References

[1] T. H. Le, H. Chen, and M. A. Babar, "Deep learning for source code modeling and generation: Models, applications, and challenges," *ACM Computing Surveys (CSUR)*, vol. 53, no. 3, pp. 1–38, 2020.

[2] H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang, "Combining graph-based learning with automated data collection for code vulnerability detection," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 1943–1958, 2020.

[3] A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, "Intellicode compose: Code generation using transformer," in *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, 2020, pp. 1433–1443.

[4] M. Liu, J. Wang, T. Lin, Q. Ma, Z. Fang, and Y. Wu, "An empirical study of the code generation of safety-critical software using llms," *Applied Sciences*, vol. 14, no. 3, 2024.

[5] Y. Zhou, S. Liu, J. K. Siow, X. Du, and Y. Liu, "Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks," in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, Eds., 2019, pp. 10 197–10 207. [Online]. Available: https://proceedings.neurips.cc/paper/2019/hash/49265d2447bc3bbfe9e76306ce40a31f-Abstract.html

[6] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "Vuldeepecker: A deep learning-based system for vulnerability detection," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_03A-2_Li_paper.pdf

[7] Z. Yang, J. Shi, J. He, and D. Lo, "Natural attack for pre-trained models of code," in *ICSE*. ACM, 2022, pp. 1482–1493.

[8] Z. Tian, J. Chen, and Z. Jin, "Code difference guided adversarial example generation for deep code models," in *ASE*. IEEE, 2023, pp. 850–862.

[9] Z. Li, C. Zhang, M. Pan, T. Zhang, and X. Li, "AACEGEN: attention guided adversarial code example generation for deep code models," in *ASE*. ACM, 2024, pp. 1245–1257.

[10] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design - The Hardware / Software Interface (Revised 4th Edition)*, ser. The Morgan Kaufmann Series in Computer Architecture and Design. Academic Press, 2012. [Online]. Available: http://www.elsevierdirect.com/product.jsp?isbn=9780123747501

[11] I. Sommerville, *Software engineering, 8th Edition*, ser. International computer science series. Addison-Wesley, 2007. [Online]. Available: https://www.worldcat.org/oclc/65978675

[12] X. Du, M. Wen, Z. Wei, S. Wang, and H. Jin, "An extensive study on adversarial attack against pre-trained models of code," in *ESEC/SIGSOFT FSE*. ACM, 2023, pp. 489–501.

[13] Q. Hu, Y. Guo, X. Xie, M. Cordy, M. Papadakis, L. Ma, and Y. L. Traon, "Codes: Towards code model generalization under distribution shift," in *ICSE (NIER)*. IEEE, 2023, pp. 1–6.

[14] Y. Yang, H. Fan, C. Lin, Q. Li, Z. Zhao, and C. Shen, "Exploiting the adversarial example vulnerability of transfer learning of source code," *IEEE Transactions on Information Forensics and Security*, 2024.

[15] H. Zhang, Z. Fu, G. Li, L. Ma, Z. Zhao, H. Yang, Y. Sun, Y. Liu, and Z. Jin, "Towards robustness of deep program processing models - detection, estimation, and enhancement," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 3, pp. 50:1–50:40, 2022.

[16] L. Huang, W. Sun, and M. Yan, "Iterative generation of adversarial example for deep code models," in *ICSE*. IEEE, 2025, pp. 2213–2224.

[17] H. Zhang, Z. Li, G. Li, L. Ma, Y. Liu, and Z. Jin, "Generating adversarial examples for holding robustness of source code processing models," in *AAAI*. AAAI Press, 2020, pp. 1169–1176.

[18] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. B. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, and M. Zhou, "Graphcodebert: Pre-training code representations with data flow," in *ICLR*. OpenReview.net, 2021.
[19] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.
[20] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation," in *ACL (1)*. Association for Computational Linguistics, 2022, pp. 7212–7225.
[21] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *MAPS@PLDI*. ACM, 2022, pp. 1–10.
[22] A. Jha and C. K. Reddy, "Codeattack: Code-based adversarial attacks for pre-trained programming language models," in *AAAI*. AAAI Press, 2023, pp. 14 892–14 900.
[23] I. Saberi and F. H. Fard, "Model-agnostic syntactical information for pre-trained programming language models," in *MSR*. IEEE, 2023, pp. 183–193.
[24] Z. Tian, J. Chen, and X. Zhang, "On-the-fly improving performance of deep code models via input denoising," in *ASE*. IEEE, 2023, pp. 560–572.
[25] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *EMNLP (Findings)*, ser. Findings of ACL, vol. EMNLP 2020. Association for Computational Linguistics, 2020, pp. 1536–1547.
[26] Y. Wang, W. Wang, S. R. Joty, and S. C. H. Hoi, "Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation," in *EMNLP (1)*. Association for Computational Linguistics, 2021, pp. 8696–8708.
[27] Y. Wang, H. Le, A. Gotmare, N. D. Q. Bui, J. Li, and S. C. H. Hoi, "Codet5+: Open code large language models for code understanding and generation," in *EMNLP*. Association for Computational Linguistics, 2023, pp. 1069–1088.
[28] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, S. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis," in *ICLR*. OpenReview.net, 2023.
[29] F. Marra, D. Gragnaniello, and L. Verdoliva, "On the vulnerability of deep learning to adversarial attacks for camera model identification," *Signal Process. Image Commun.*, vol. 65, pp. 240–248, 2018.
[30] Y. Zhang, Z. Wang, J. Jiang, H. You, and J. Chen, "Toward improving the robustness of deep learning models via model transformation," in *ASE*. ACM, 2022, pp. 104:1–104:13.
[31] J. Jiang, J. Yang, Y. Zhang, Z. Wang, H. You, and J. Chen, "A post-training framework for improving the performance of deep learning models via model transformation," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 3, Mar. 2024.
[32] Y. Qu, S. Huang, and Y. Yao, "A survey on robustness attacks for deep code models," *Autom. Softw. Eng.*, vol. 31, no. 2, p. 65, 2024.
[33] N. Yefet, U. Alon, and E. Yahav, "Adversarial examples for models of code," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 162:1–162:30, 2020.
[34] J. Henkel, G. Ramakrishnan, Z. Wang, A. Albarghouthi, S. Jha, and T. W. Reps, "Semantic robustness of models of source code," in *SANER*. IEEE, 2022, pp. 526–537.
[35] H. Zhang, S. Lu, Z. Li, Z. Jin, L. Ma, Y. Liu, and G. Li, "Codebert-attack: Adversarial attack against source code deep learning models via pre-trained model," *J. Softw. Evol. Process.*, vol. 36, no. 3, 2024.
[36] Z. Zeng, H. Tan, H. Zhang, J. Li, Y. Zhang, and L. Zhang, "An extensive study on pre-trained models for program understanding and generation," in *ISSTA*. ACM, 2022, pp. 39–51.
[37] M. V. Pour, Z. Li, L. Ma, and H. Hemmati, "A search-based testing framework for deep neural networks of source code embedding," in *ICST*. IEEE, 2021, pp. 36–46.
[38] W. Zhang, S. Guo, H. Zhang, Y. Sui, Y. Xue, and Y. Xu, "Challenging machine learning-based clone detectors via semantic-preserving code transformations," *IEEE Trans. Software Eng.*, vol. 49, no. 5, pp. 3052–3070, 2023.
[39] J. Tian, C. Wang, Z. Li, and Y. Wen, "Generating adversarial examples of source code classification models via q-learning-based markov decision process," in *QRS*. IEEE, 2021, pp. 807–818.
[40] Y. Wang, Y. Chen, Y. Zhao, Z. Gong, J. Chen, and D. Hao, "Mutual learning-based framework for enhancing robustness of code models via adversarial training," in *ASE*. ACM, 2024, pp. 1484–1496.
[41] M. Allamanis, "Learning natural coding conventions," Ph.D. dissertation, University of Edinburgh, UK, 2017. [Online]. Available: https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.738886
[42] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. [Online]. Available: https://openreview.net/forum?id=BJOFETxR-
[43] V. Raychev, M. T. Vechev, and A. Krause, "Predicting program properties from 'big code'," *Commun. ACM*, vol. 62, no. 3, pp. 99–107, 2019.

[44] H. Tran, N. M. Tran, S. Nguyen, H. Nguyen, and T. N. Nguyen, "Recovering variable names for minified code with usage contexts," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, J. M. Atlee, T. Bultan, and J. Whittle, Eds. IEEE / ACM, 2019, pp. 1165–1175. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00119

[45] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. T. Vechev, "Debin: Predicting debug information in stripped binaries," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. ACM, 2018, pp. 1667–1680. [Online]. Available: https://doi.org/10.1145/3243734.3243866

[46] J. Lacomis, P. Yin, E. J. Schwartz, M. Allamanis, C. Le Goues, G. Neubig, and B. Vasilescu, "DIRE: A neural approach to decompiled identifier naming," in *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 2019, pp. 628–639. [Online]. Available: https://doi.org/10.1109/ASE.2019.00064

[47] P. Banerjee, K. K. Pal, F. Wang, and C. Baral, "Variable name recovery in decompiled binary code using constrained masked language modeling," *CoRR*, vol. abs/2103.12801, 2021. [Online]. Available: https://arxiv.org/abs/2103.12801

[48] Q. Chen, J. Lacomis, E. J. Schwartz, C. Le Goues, G. Neubig, and B. Vasilescu, "Augmenting decompiler output with learned variable names and types," in *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, K. R. B. Butler and K. Thomas, Eds. USENIX Association, 2022, pp. 4327–4343. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/chen-qibin

[49] X. Xu, Z. Zhang, Z. Su, Z. Huang, S. Feng, Y. Ye, N. Jiang, D. Xie, S. Cheng, L. Tan, and X. Zhang, "Unleashing the power of generative model in recovering variable names from stripped binary," in *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society, 2025. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/unleashing-the-power-of-generative-model-in-recovering-variable-names-from-stripped-binary/

[50] D. Xie, Z. Zhang, N. Jiang, X. Xu, L. Tan, and X. Zhang, "Resym: Harnessing llms to recover variable and data structure symbols from stripped binaries," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*, B. Luo, X. Liao, J. Xu, E. Kirda, and D. Lie, Eds. ACM, 2024, pp. 4554–4568. [Online]. Available: https://doi.org/10.1145/3658644.3670340

[51] S. Luan, D. Yang, C. Barnaby, K. Sen, and S. Chandra, "Aroma: code recommendation via structural code search," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, pp. 152:1–152:28, 2019. [Online]. Available: https://doi.org/10.1145/3360578

[52] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 40:1–40:29, 2019. [Online]. Available: https://doi.org/10.1145/3290353

[53] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. [Online]. Available: https://openreview.net/forum?id=H1gKYo09tX

[54] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: scalable and accurate tree-based detection of code clones," in *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*. IEEE Computer Society, 2007, pp. 96–105. [Online]. Available: https://doi.org/10.1109/ICSE.2007.30

[55] T. László and Á. Kiss, "Obfuscating c++ programs via control flow flattening," *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, vol. 30, no. 1, pp. 3–19, 2009.

[56] C. S. Collberg, C. D. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*, D. B. MacQueen and L. Cardelli, Eds. ACM, 1998, pp. 184–196. [Online]. Available: https://doi.org/10.1145/268946.268962

[57] D. Xu, J. Ming, and D. Wu, "Generalized dynamic opaque predicates: A new control flow obfuscation method," in *Information Security - 19th International Conference, ISC 2016, Honolulu, HI, USA, September 3-6, 2016, Proceedings*, ser. Lecture Notes in Computer Science, M. Bishop and A. C. A. Nascimento, Eds. Springer, 2016, pp. 323–342. [Online]. Available: https://doi.org/10.1007/978-3-319-45871-7_20

[58] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," 1997.

[59] S. Mohseni, S. Mohammadi, D. Tilwani, Y. Saxena, G. K. Ndawula, S. Vema, E. Raff, and M. Gaur, "Can llms obfuscate code? A systematic analysis of large language models into assembly code obfuscation," in *Thirty-Ninth AAAI Conference on Artificial Intelligence, Thirty-Seventh Conference on Innovative Applications of Artificial Intelligence, Fifteenth Symposium on Educational Advances in Artificial Intelligence, AAAI 2025, Philadelphia, PA, USA, February 25 - March 4, 2025*, T. Walsh, J. Shah, and Z. Kolter, Eds. AAAI Press, 2025, pp. 24 893–24 901. [Online]. Available: https://doi.org/10.1609/aaai.v39i23.34672

[60] M. Fowler, *Refactoring - Improving the Design of Existing Code*, ser. Addison Wesley object technology series. Addison-Wesley, 1999. [Online]. Available: http://martinfowler.com/books/refactoring.html

[61] D. Cui, S. Wang, Y. Luo, X. Li, J. Dai, L. Wang, and Q. Li, "Rmove: Recommending move method refactoring opportunities using structural and semantic representations of code," in *IEEE International Conference on Software Maintenance and Evolution, ICSME 2022, Limassol, Cyprus, October 3-7, 2022*. IEEE, 2022, pp. 281–292. [Online]. Available: https://doi.org/10.1109/ICSME55016.2022.00033

[62] D. Cui, Q. Wang, S. Wang, J. Chi, J. Li, L. Wang, and Q. Li, "REMS: recommending extract method refactoring opportunities via multi-view representation of code property graph," in *31st IEEE/ACM International Conference on Program Comprehension, ICPC 2023, Melbourne, Australia, May 15-16, 2023*. IEEE, 2023, pp. 191–202. [Online]. Available: https://doi.org/10.1109/ICPC58990.2023.00034

[63] D. Cui, J. Wang, Q. Wang, P. Ji, M. Qiao, Y. Zhao, J. Hu, L. Wang, and Q. Li, "Three heads are better than one: Suggesting move method refactoring opportunities with inter-class code entity dependency enhanced hybrid hypergraph neural network," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024*, V. Filkov, B. Ray, and M. Zhou, Eds. ACM, 2024, pp. 745–757. [Online]. Available: https://doi.org/10.1145/3691620.3695068
[64] B. Liu, Y. Jiang, Y. Zhang, N. Niu, G. Li, and H. Liu, "Exploring the potential of general purpose llms in automated software refactoring: an empirical study," *Autom. Softw. Eng.*, vol. 32, no. 1, p. 26, 2025. [Online]. Available: https://doi.org/10.1007/s10515-025-00500-0
[65] H. Zhang, S. Lu, Z. Li, Z. Jin, L. Ma, Y. Liu, and G. Li, "Codebert-attack: Adversarial attack against source code deep learning models via pre-trained model," *J. Softw. Evol. Process.*, vol. 36, no. 3, 2024.
[66] A. Jha and C. K. Reddy, "Codeattack: Code-based adversarial attacks for pre-trained programming language models," in *AAAI*. AAAI Press, 2023, pp. 14 892–14 900.
[67] C. Na, Y. Choi, and J. Lee, "DIP: dead code insertion based black-box attack for programming language model," in *ACL (1)*. Association for Computational Linguistics, 2023, pp. 7777–7791.
[68] 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.
[69] 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.
[70] A. V. Phan and M. Le Nguyen, "Convolutional neural networks on assembly code for predicting software defects," in *2017 21st Asia Pacific Symposium on Intelligent and Evolutionary Systems (IES)*. IEEE, 2017, pp. 37–42.
[71] D. Rey and M. Neuhäuser, "Wilcoxon-signed-rank test," in *International Encyclopedia of Statistical Science*. Springer, 2011, pp. 1658–1659.
[72] D. Nam, A. Macvean, V. J. Hellendoorn, B. Vasilescu, and B. A. Myers, "Using an LLM to help with code understanding," in *ICSE*. ACM, 2024, pp. 97:1–97:13.
[73] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," in *ICLR*. OpenReview.net, 2023.
[74] R. Bavishi, M. Pradel, and K. Sen, "Context2name: A deep learning-based approach to infer natural variable names from usage contexts," *CoRR*, vol. abs/1809.05193, 2018.
[75] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *MSR*. IEEE Computer Society, 2013, pp. 207–216.
[76] D. Nam, A. Macvean, V. J. Hellendoorn, B. Vasilescu, and B. A. Myers, "Using an LLM to help with code understanding," in *ICSE*. ACM, 2024, pp. 97:1–97:13.
[77] S. K. Pandey, S. Chand, J. Horkoff, M. Staron, M. Ochodek, and D. Durisic, "Design pattern recognition: a study of large language models," *Empir. Softw. Eng.*, vol. 30, no. 3, p. 69, 2025.
[78] C. Huang, S. Li, R. Liu, H. Wang, and Y. Chen, "Large foundation models for power systems," in *2024 IEEE Power & Energy Society General Meeting (PESGM)*. IEEE, 2024, pp. 1–5.
[79] C. S. Xia, Y. Wei, and L. Zhang, "Automated program repair in the era of large pre-trained language models," in *ICSE*. IEEE, 2023, pp. 1482–1494.
[80] S. Vijayvargiya, M. Saad, and T. Sharma, "Enhancing identifier naming through multi-mask fine-tuning of language models of code," in *SCAM*. IEEE, 2024, pp. 71–82.
[81] L. Kovriguina, R. Teucher, D. Radyush, and D. Mouromtsev, "SPARQLGEN: one-shot prompt-based approach for SPARQL query generation," in *SEMANTiCS (Posters & Demos)*, ser. CEUR Workshop Proceedings, vol. 3526. CEUR-WS.org, 2023.
[82] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Dang, A. Yang, R. Men, F. Huang, X. Ren, X. Ren, J. Zhou, and J. Lin, "Qwen2.5-coder technical report," *CoRR*, vol. abs/2409.12186, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2409.12186
[83] J. Wang, J. Chen, Y. Sun, X. Ma, D. Wang, J. Sun, and P. Cheng, "Robot: Robustness-oriented testing for deep learning systems," in *ICSE*. IEEE, 2021, pp. 300–311.
[84] L. Wang, X. Xie, X. Du, M. Tian, Q. Guo, Z. Yang, and C. Shen, "Distxplore: Distribution-guided testing for evaluating and enhancing deep learning systems," in *ESEC/SIGSOFT FSE*. ACM, 2023, pp. 68–80.
[85] Z. Jiang, H. Li, X. Tian, and R. Wang, "Semantic feature-based test selection for deep neural networks: A frequency domain perspective," *Comput. Sci. Inf. Syst.*, vol. 21, no. 4, pp. 1499–1522, 2024.
[86] DeepSeek-AI, "Deepseek-v3 technical report," *CoRR*, vol. abs/2412.19437, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2412.19437
[87] OpenAI, "Gpt-4o system card," *CoRR*, vol. abs/2410.21276, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2410.21276
[88] DeepSeek, "Deepseek api documentation," 2025, accessed: March 30, 2025. [Online]. Available: https://api-docs.deepseek.com
[89] Z. Wang, H. You, J. Chen, Y. Zhang, X. Dong, and W. Zhang, "Prioritizing test inputs for deep neural networks via mutation analysis," in *ICSE*. IEEE, 2021, pp. 397–409.
[90] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandinavian journal of statistics*, pp. 65–70, 1979.

[91] D. S. Kerby, "The simple difference formula: An approach to teaching nonparametric correlation1," *Comprehensive Psychology*, vol. 3, p. 11.IT.3.1, 2014.
[92] J. Sun, C. Shaib, and B. C. Wallace, "Evaluating the zero-shot robustness of instruction-tuned language models," in *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. [Online]. Available: https://openreview.net/forum?id=g9diuvxN6D
[93] J. Zhuo, S. Zhang, X. Fang, H. Duan, D. Lin, and K. Chen, "Prosa: Assessing and understanding the prompt sensitivity of llms," in *Findings of the Association for Computational Linguistics: EMNLP 2024, Miami, Florida, USA, November 12-16, 2024*, ser. Findings of ACL, Y. Al-Onaizan, M. Bansal, and Y. Chen, Eds. Association for Computational Linguistics, 2024, pp. 1950–1976. [Online]. Available: https://doi.org/10.18653/v1/2024.findings-emnlp.108
[94] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics and intelligent laboratory systems*, vol. 2, no. 1-3, pp. 37–52, 1987.
[95] M. L. Menéndez, J. A. Pardo, L. Pardo, and M. d. C. Pardo, "The jensen-shannon divergence," *Journal of the Franklin Institute*, vol. 334, no. 2, pp. 307–318, 1997.
[96] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. T. Devanbu, "On the naturalness of software," in *ICSE*. IEEE Computer Society, 2012, pp. 837–847.
[97] V. Antinyan, M. Staron, and A. Sandberg, "Evaluating code complexity triggers, use of complexity measures and the influence of code complexity on maintenance time," *Empir. Softw. Eng.*, vol. 22, no. 6, pp. 3057–3087, 2017.
[98] M. Neil, "Metrics and models in software quality engineering, by stephen h. kan, addison-wesley, 1995 (book review)," *Softw. Test. Verification Reliab.*, vol. 5, no. 4, pp. 273–275, 1995.
[99] B. Lin and G. Robles, "Investigating the impact of vocabulary difficulty and code naturalness on program comprehension," *CoRR*, vol. abs/2308.13429, 2023.
[100] M. Allamanis, E. T. Barr, P. T. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *CoRR*, vol. abs/1709.06182, 2017.
[101] D. W. Hosmer Jr, S. Lemeshow, and R. X. Sturdivant, *Applied logistic regression*. John Wiley & Sons, 2013.
[102] H. You, Z. Wang, B. Lin, and J. Chen, "Navigating the testing of evolving deep learning systems: An exploratory interview study," in *ICSE*. IEEE, 2025, pp. 2726–2738.
[103] G. T. Henry, *Practical sampling*. Sage, 1990, vol. 21.
[104] J. L. Fleiss, "Measuring nominal scale agreement among many raters." *Psychological bulletin*, vol. 76, no. 5, p. 378, 1971.
[105] M. L. McHugh, "Interrater reliability: the kappa statistic," *Biochemia medica*, vol. 22, no. 3, pp. 276–282, 2012.
[106] K. L. Gwet, "Computing inter-rater reliability and its variance in the presence of high agreement," *British Journal of Mathematical and Statistical Psychology*, vol. 61, no. 1, pp. 29–48, 2008.
[107] A. R. Feinstein and D. V. Cicchetti, "High agreement but low kappa: I. the problems of two paradoxes," *Journal of clinical epidemiology*, vol. 43, no. 6, pp. 543–549, 1990.
[108] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," in *ICLR*. OpenReview.net, 2022.
[109] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," *CoRR*, vol. abs/2004.05150, 2020. [Online]. Available: https://arxiv.org/abs/2004.05150
[110] M. Zaheer, G. Guruganesh, K. A. Dubey, J. Ainslie, C. Alberti, S. Ontañón, P. Pham, A. Ravula, Q. Wang, L. Yang, and A. Ahmed, "Big bird: Transformers for longer sequences," in *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, Eds., 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/c8512d142a2d849725f31a9a7a361ab9-Abstract.html