INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Robustness of AI Models in Software Vulnerability Detection

JOSÉ PEDRO SOUSA GONÇALVES
Junho de 2025

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Robustness of AI Models in Software Vulnerability Detection

José Pedro Sousa Gonçalves

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Cybersecurity And Systems Administration

Advisor: Dr. Isabel Cecília Correia da Silva Praça Gomes Pereira
Co-Advisor: Dr. Eva Catarina Gomes Maia

Porto, June 28, 2025

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

Therefore the work presented in this document is original and authored by me, having not previously been used for any other end. The exceptions are explicitly recognised in the section "Ethical considerations" of the first chapter.

AI writing assistants were used solely to improve grammar during the preparation of this document. All ideas presented in this thesis reflect original human input and authorship.

I further declare that I have fully acknowledged the Code of Ethical Conduct of P.PORTO.

ISEP, Porto, June 28, 2025

Abstract

Artificial Intelligence (AI) tools are increasingly adopted for source code analysis, assisting in tasks such as vulnerability detection, code generation, and automated review. However, despite their growing capabilities, these models exhibit significant vulnerabilities when exposed to adversarial environments, where subtle, functionality-preserving code transformations can drastically reduce their performance. This weakness is particularly critical in CI/CD pipelines, where undetected malicious code can compromise software integrity and security.

To address this issue, the present thesis introduces SCoPE2 (Source Code Processing Engine 2), an enhanced and more efficient version of the original SCoPE framework. SCoPE2 achieves substantial improvements in runtime performance, requiring only 2.7% of the execution time of its predecessor, while offering greater extensibility. It is capable of generating adversarial examples and applying a wide range of code transformations, enabling both the evaluation of model robustness and the development of defensive strategies.

The proposed solution leverages SCoPE2 to implement two complementary defensive mechanisms: adversarial fine-tuning, which retrains models on adversarial examples to enhance their robustness, and an inference-time normalization layer that standardizes input code before classification. Experimental results show that adversarial fine-tuning effectively recovers model performance under adversarial conditions, reaching accuracy levels comparable to those obtained on clean inputs. Meanwhile, the normalization strategy enhances resistance to identifier-based manipulations, despite having a trade-off in overall accuracy. Furthermore, an analysis of adversarial sample similarity reveals that examples with greater syntactic divergence from the original code tend to be more successful at misleading the models.

The main conclusion of this thesis is that, although AI-powered code analysis offers considerable potential for software engineering, current models remain vulnerable to adversarial attacks. This finding underscores the necessity of integrating robust and layered defence mechanisms, as no single method offers complete protection against all adversarial threats.

Keywords: Adversarial robustness, Software Vulnerability Detection, Machine learning, Cybersecurity

Resumo

As ferramentas de AI estão a ser cada vez mais adotadas na análise de código, apoiando tarefas como a deteção de vulnerabilidades, a geração e a revisão automatizada de código. No entanto, apesar das suas capacidades crescentes, estes modelos apresentam vulnerabilidades significativas quando expostos a ambientes adversariais, nos quais transformações subtis ao código, que preservam a sua funcionalidade, podem reduzir drasticamente o seu desempenho. Esta fragilidade é particularmente crítica em pipelines de CI/CD, onde código malicioso não detetado pode comprometer a integridade e a segurança do software.

Para enfrentar este problema, esta tese apresenta o SCoPE2 (Source Code Processing Engine 2), uma versão melhorada e mais eficiente do framework original SCoPE. O SCoPE2 alcança melhorias substanciais ao nível do desempenho, requerendo apenas 2,7% do tempo de execução do seu antecessor, ao mesmo tempo que oferece uma maior extensibilidade. É capaz de gerar exemplos adversariais e de aplicar uma vasta gama de transformações ao código, possibilitando tanto a avaliação da robustez de modelos como o desenvolvimento de estratégias de defesa.

A solução proposta utiliza o SCoPE2 para aplicar dois mecanismos defensivos complementares: o fine-tuning adversarial, que consiste no re-treino de modelos com exemplos adversariais para reforçar a sua robustez; e uma camada de normalização aplicada em tempo de inferência, que padroniza o código de entrada antes da classificação. Os resultados experimentais demonstram que o fine-tuning adversarial recupera eficazmente o desempenho dos modelos em cenários adversariais, atingindo níveis de precisão comparáveis aos obtidos com dados não alterados. A estratégia de normalização melhora a resistência a manipulações baseadas em identificadores, embora implique um compromisso ao nível da precisão global do modelo. Adicionalmente, uma análise da similaridade dos exemplos adversariais revela que os exemplos com maior divergência sintática face ao código original tendem a ser mais eficazes na tarefa de enganar os modelos.

A principal conclusão desta tese é que, embora a análise de código suportada por IA apresente um potencial considerável para a engenharia de software, os modelos atuais continuam vulneráveis a ataques adversariais. Esta constatação realça a necessidade de integrar mecanismos de defesa robustos e em camadas, uma vez que nenhum método isolado oferece proteção completa contra as ameaças adversariais.

Acknowledgement

I would like to express my gratitude to GECAD, the research centre that supported the work presented in this thesis in the context of the BEHAVIOR project (NORTE2030-FEDER-00576300 no. 14391), and to everyone who accompanied me on this journey. In particular, I would like to thank my supervisors, Professors Isabel Praça and Eva Maia, for their ongoing guidance, feedback and advice. I would also like to thank all my colleagues, especially Miguel, for their help and support whenever I needed it.

I would also like to thank my colleagues at Celfocus, who were always willing to assist me whenever necessary. A special thanks goes to Pedro, who, as my manager, was always available to support me in whatever I needed.

Finally, I would like to express my profound gratitude to my family, especially my parents, grandparents and brother, for their unconditional support. Without them, I would not be who I am today.

Contents

List of Figures

List of Tables

List of Source Code

Acronyms

AI Artificial Intelligence.

API Application Programming Interface.

ASR Attack Success Rate.

CNN Convolutional Neural Network.

DL Deep Learning.

DTO Data Transfer Object.

EC Exclusion Criteria.

FGM Projected Gradient Descent.

FN False Negatives.

FP False Positives.

GAN Generative Adversarial Network.

IC Inclusion Criteria.

LLM Large Language Model.

LoRA Low-Rank Adaptation.

MHM Metropolis-Hastings Modifier.

MIA Membership Inference Attack.

NIST National Institute of Standards and Technology.

NLP Natural Language Processing.

PEFT Parameter-Efficient Fine-Tuning.

PGD Fast Gradient Method.

PRISMA Preferred Reporting Items for Systematic Reviews and Meta-Analyses.

RQ Research Question.

SVD Software Vulnerability Detection.

TN True Negatives.

TP True Positives.

WoS Web Of Science.

Chapter 1

Introduction

This chapter provides the context for the work presented in this thesis, outlining the problem statement, the ethical considerations, the defined objectives, and the research questions. It also highlights the main scientific contributions and presents the structure of the document.

1.1 Context and Motivation

Software plays a vital role in nearly every aspect of modern life. From the cars we drive [1] to the appliances we use [2], almost everything today relies on some form of software. Its ubiquity spans not only everyday devices but also critical systems, such as medical equipment [3], power grids [4], and financial infrastructures [5]. However, the widespread integration of software comes with significant challenges. If a software bug or malfunction occurs, it can lead to disruptions, inefficiencies, or even serious consequences. Malicious actors can exploit software bugs to gain partial or complete control over devices, with potentially catastrophic outcomes. For instance, ransomware attacks can exploit vulnerabilities in systems to disrupt critical infrastructures [6], attackers could disable critical systems of a car [7], or even cause loss of life by compromising medical devices [6].

The software industry is increasingly adopting the "shift-left" paradigm, emphasizing the identification and resolution of vulnerabilities early in the development lifecycle [8]. Early detection not only reduces costs, but also improves software reliability [9]. Artificial Intelligence (AI) has emerged as a key enabler of this approach, supporting tasks such as code commenting, summarization, vulnerability detection, and automated code generation [10]. As a result, AI-powered tools have become an integral part of development workflows. According to a GitHub survey, 92% of developers now use some form of AI assistance for programming tasks [11]. These tools are often used to solve complex problems that require advanced reasoning and domain expertise [12]. However, AI tools are not infallible and can produce insecure or non-functional code [13, 14]. To mitigate such risks, many organizations are incorporating AI-driven tools into DevSecOps frameworks to improve code quality and prevent vulnerabilities [15, 16].

Despite their potential, AI models present new challenges. Vulnerabilities in AI systems themselves can be exploited by attackers to subvert their intended functions. For example, malware authors can use semantically preserving transformations to evade AI-based malware detection systems [17–19] and inject malicious code into repositories. As a result, enhancing the security and robustness of AI-driven systems in software development is crucial to ensuring their safe and effective deployment.

1.2 Problem Statement

Code analysis and generation are inherently complex tasks that involve processing sequences of tokens, each carrying unique semantic values [20]. Unlike natural language, where a degree of variability in word order or sentence structure is acceptable, source code adheres to strict syntactic rules [21]. Even small structural changes in code can significantly impact its functional behavior [22]. Despite these rigid syntactic constraints, code semantics often allows for significant flexibility, as the same task can frequently be implemented in multiple ways while achieving equivalent functional outcomes.

This flexibility increases the complexity of code analysis. The space of potential implementations for any given task is vast, as numerous distinct variations may share the same semantic meaning. To address this issue, recent works have leveraged AI models for code analysis tasks [23, 24]. These specialized models, also known as "code models", are trained on extensive datasets to analyze and understand code effectively. However, the abundance of semantically equivalent variations still poses unique challenges for these models [22, 25]. To analyze source code accurately, a model must deeply understand the semantic role of each token and its interactions within the broader context of the program [26]. A robust model should distinguish between superficial differences and meaningful semantic variations, ensuring consistent performance regardless of refactoring or syntactic modifications that preserve functionality.

The limited depth of code understanding in current models has raised growing concerns about their robustness. While many models demonstrate strong performance on specific tasks, they are often susceptible to attacks [27–30]. Even state-of-the-art models, such as GPT-4 and CodeLlama [31], which excel in code analysis and generation tasks [32–34], are not immune to such weaknesses [35, 36]. This issue can be especially relevant with the usage of AI-powered detection solutions in CI/CD pipelines, where a malicious actor can add to the repository vulnerable code that won't be detected by the AI models. As such, there is an urgent need to develop solutions that improve the robustness and reliability of code models. These solutions should focus on improving the model's ability to distinguish true semantic meaning from syntactic variability, ensuring consistent performance in the face of diverse code representations. By addressing these robustness issues, more secure and reliable code analysis systems can be developed, better equipped to resist adversarial threats while delivering accurate and trustworthy results.

1.3 Objectives and Research Questions

The main objective of this thesis is to develop a solution to address the problem of robustness in AI models for source code analysis. The proposed solution aims to generate adversarial samples that can serve two purposes: testing and enhance the robustness of AI models. To achieve this goal, four specific sub-objectives have been defined:

- **Objective 1**: Investigate the techniques used to attack code models and improve their robustness.
- **Objective 2**: Identify the methods used in the literature to generate adversarial samples of source code.
- **Objective 3**: Design and implement a tool for generating adversarial samples, using techniques identified in the literature review.

- **Objective 4**: Analyze the impact that the use of different defense methods have on the robustness of AI models.

To ensure the successful completion of these objectives, a guiding Research Question (RQ) was formulated: *What are the state-of-the-art methods for generating code adversarial examples, attacking code models, and enhancing robustness in source code analysis?* This main RQ was further divided into three specific sub-questions to structure the research:

- **RQ1**: What are the state-of-the-art methods used to attack models for source code analysis?
- **RQ2**: What are the state-of-the-art methods used to improve the robustness of models that analyze source code?
- **RQ3**: Which are the most common methods to generate adversarial samples of source code?

1.4 Ethical Considerations

This document and the related project were developed with several ethical considerations in mind, following the Code of Conduct of the Polytechnic Institute of Porto [37]. The following aspects of the Code of Conduct were given particular attention:

- **Article 6, Section 2.8**: Throughout this document, all third-party ideas and work are properly cited.
- **Article 6, Section 2.9** is also relevant, as this document presents original work that has not been previously submitted or published. Any reused material from other publications is properly referenced. It is also worth noting that although the original systematic review was conducted as part of the PREPD project, it has since been updated and integrated into this document.
- **Article 6, Section 2.11** was carefully followed, especially regarding the interpretation of results from other studies.
- **Article 8** is reflected in the integrity statement at the beginning of the document.
- **Article 10** was also strictly followed throughout the work.

In line with the Code of Conduct of the Polytechnic Institute of Porto [37], ethical considerations played a key role in creating this document. One significant aspect of this work concerns the use of data to train the AI models. Only publicly available data has been utilized, with code sourced exclusively from open-source repositories. Any personally identifiable information identified within the datasets must be removed to safeguard privacy.

Another important ethical consideration relates to the goal of this project. One of the main objectives is to develop a tool for generating adversarial examples for code models. By believing in free knowledge and access to cutting-edge tools as a way to improve AI models for the benefit of humanity, the developed code will be made publicly available. While this tool is used here in a positive and constructive way, such as improving the robustness of AI models, it could also be misused. Therefore, this document outlines solutions to mitigate the risks posed by adversarial examples and to prevent the misuse of the developed tool.

1.5 Scientific Contributions

During the development of this thesis, a significant amount of research work was carried out to achieve the proposed objectives. This thesis presents the extensive research conducted during its development, which began with the following paper:

> José Gonçalves, Tiago Dias, Eva Maia, Isabel Praça. (2024). *SCoPE: Evaluating LLMs for Software Vulnerability Detection*. In: Mehmood, R., et al. Distributed Computing and Artificial Intelligence, Special Sessions I, 21st International Conference. DCAI 2024. Lecture Notes in Networks and Systems, vol 1198. Springer, Cham. `https://doi.org/10.1007/978-3-031-76459-2_4`.

This initial paper introduced the SCoPE framework and formulated the core hypothesis that code preprocessing could influence Large Language Model (LLM)-based vulnerability detectors. Building on this work, two further studies were carried out within the scope of this thesis. The first was:

> J. Gonçalves et al. (2025). *Evaluating LLaMA 3.2 for Software Vulnerability Detection*. Proceedings of the 2025 European Interdisciplinary Cybersecurity Conference (EICC 2025). [Online]. Available: `https://arxiv.org/abs/2503.07770`.

This study measured the impact of using SCoPE for variable-name generalization on a state-of-the-art vulnerability detection model's accuracy, concluding that identifier-based manipulations can influence its predictions.

> J. Gonçalves et al. (2025). *Enhancing Large Language Models with Faster Code Preprocessing for Vulnerability Detection*. Proceedings of the 22nd International Conference on Distributed Computing and Artificial Intelligence (DCAI 2025).

This paper introduces SCoPE2, an optimized redesign of the original framework, delivering improved processing speed and greater extensibility. It also introduced new transformations on SCoPE2 that were not present on the original framework, using the literature review presented in this thesis as the basis for selecting impactful transformations that could be used in adversarial sample generation. Building on the insights gained from these studies, this thesis goes one step further and explores adversarial defense strategies for LLMs. The proposed defensive approach uses the SCoPE2 framework to generate adversarial samples and create a complementary normalisation layer deployed in front of the model at inference time. Together, these contributions demonstrate clear progression and advancement in the field.

1.6 Document Structure

This document is organized into several chapters, each focusing on a distinct aspect of the research. Chapter 1 introduces the contextual background and outlines the research questions that guided the literature review.

Chapter 2 presents the literature review, structured to address each RQ individually. The chapter is divided into sections, with each section dedicated to a specific research question. Each section is further subdivided into two parts: the first describes the methodology used to conduct the literature review for that question, and the second presents and discusses the findings. Chapter 3 introduces the SCoPE2 framework and evaluates its performance.

Chapter 4 details the experimental setup, including the dataset, base model, and data preprocessing procedures. Chapter 5 reports the results of the experiments conducted in this study and, finally, Chapter 6 summarizes the main conclusions and outlines directions for future research

Chapter 2

State-of-the-art

This section provides a comprehensive overview of the systematic literature review conducted to address the guiding RQ formulated earlier: *What are the state-of-the-art methods for generating code adversarial examples, attacking code models, and enhancing robustness in source code analysis*? The three sub-questions (RQ1, RQ2, and RQ3) defined above, structure this review. To ensure rigor and transparency, the Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) [38] methodology was used across all RQs. PRISMA is a widely used standard for conducting systematic literature reviews, comprising four main phases: Identification, Screening, Eligibility, and Inclusion.

During the identification phase, relevant studies were identified using a systematic search strategy across multiple academic databases, including ACM [39], IEEE [40] and the Web of Science [41]. Search queries were created using a combination of keywords related to the research questions and boolean operators. The searches were performed within the abstract sections of papers in the databases. The same databases were used consistently across all research questions to maintain consistency. Furthermore, only articles published within the last five years were included to ensure the review captured the state-of-the-art knowledge.

In the screening phase, duplicate records were removed and the remaining studies were filtered based on their titles and abstracts. Only studies that appeared to be directly relevant to the research questions were retained. In the eligibility phase, the full-text versions of the selected studies were assessed against the inclusion and exclusion criteria. Inclusion criteria ensured that only peer-reviewed articles were considered. Studies in languages other than English, articles without sufficient methodological detail, or those focusing on unrelated areas were excluded.

Finally, in the inclusion phase, the papers were selected for detailed analysis. These studies form the basis of the evidence discussed in this document.

2.1 Attacking Code Analysis Models

The security of AI models has recently received considerable attention in the scientific community. A growing number of research has been devoted to evaluating the vulnerabilities and resilience of these models to various forms of attacks, particularly in computer vision and Natural Language Processing (NLP) tasks [26]. However, with the increasing integration of AI-based solutions into the software development lifecycle, it becomes critical to identify the potential threats that target source-code analysis models. This understanding is essential

for developing robust defenses and ensuring their resilience against such attacks. Accordingly, this section addresses RQ1: *What are the state-of-the-art methods for attacking source-code analysis models?*

2.1.1 Research Methodology

The search conducted to address RQ1 followed the PRISMA methodology. Table 2.1 summarizes the keywords that form the basis of the database search, organized by category.

Category	Keywords
Attack methods	attack, adversarial, poisoning, evasion, stealing
Model	model
Source code	"source code", "code model", "models of code", "model of code"
Negative keyword	image

Table 2.1: Keywords used in RQ1, grouped by categories.

The first category of keywords is the "Attack methods" category. The keywords selected for this category were identified during the preliminary search, and included the generic "attack" keyword alongside with other attack methods identified. Since this RQ focused on attacks against source-code analysis models, the category "source code" was added. This category contains several terms commonly used to refer to source-code processing models, such as "code model". In addition, the keyword "image" was identified as a negative keyword to help filter out false positives. The different categories were combined using AND operators and the terms in the categories were combined with OR operators.

To address this RQ, the publications must present or explore attacks against code analysis models. The publications that didn't focus on attacking source-code analysis models were excluded. Table 2.2 summarizes the Inclusion Criteria (IC) and the Exclusion Criteria (EC) used to screen the publications retrieved from the databases.

#	Inclusion Criteria	Exclusion Criteria
1	Published after 2020	Duplicates
2	Text available in English	Publications without technical details
3	Peer-reviewed journal or conference paper	Publications not addressing attacks on code models
4	Publications focused on attacks on code models	

Table 2.2: Inclusion and exclusion criteria for RQ1.

After defining the search terms and establishing the inclusion and exclusion criteria, the database search was conducted on 10th June 2025. Additional relevant publications were identified through the PRISMA snowballing process [42], where references from key articles were reviewed to discover further studies that met the inclusion criteria. This approach ensured comprehensive coverage of the literature.

A total of 616 entries were retrieved from the databases using the query terms outlined in Table 2.1. After removing duplicates and completing the screening process, 49 records were deemed suitable for detailed analysis. Figure 2.1 illustrates the distribution of documents

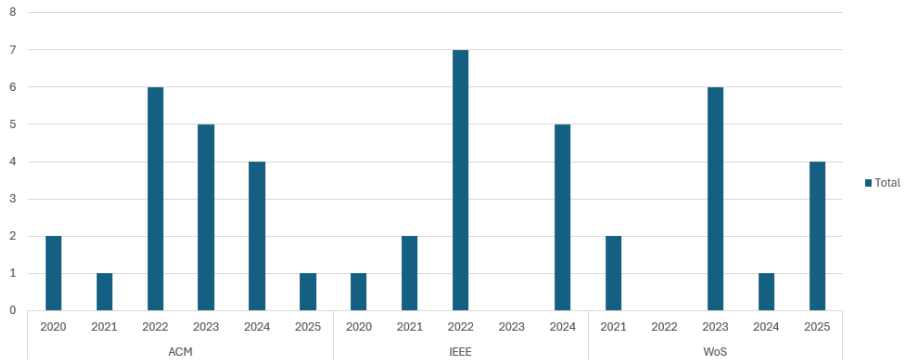

Figure 2.1: Number of papers per database for RQ1, categorized by year

retrieved from each database, organized by year, following the initial screening. The highest number of relevant papers for this RQ was observed between 2021 and 2022. Notably, no valid papers were identified from IEEE in 2023 or from Web Of Science (WoS) in 2022.

In the eligibility phase, 8 records were removed, mainly because they don't focus on attacking code models. In the end, 41 records were included in the review. The whole process is summarized in Figure 2.2.

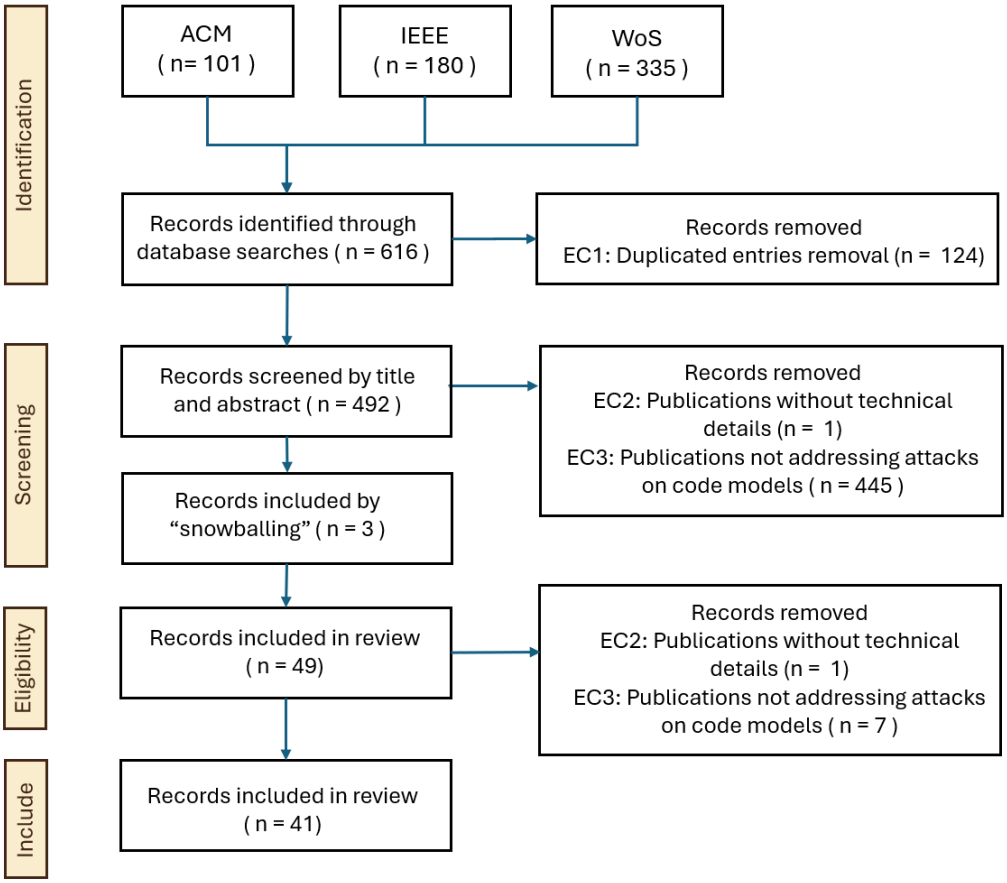

Figure 2.2: PRISMA search process for RQ1

2.1.2 Findings and Discussion

AI models are known to be vulnerable to various types of attacks. Studies in recent years have shown that AI systems applied in tasks like NLP [43, 44] and computer vision [45] can be exploited by malicious actors. With the growing use of AI models in software development, researchers have started to investigate whether code-specialized models are similarly susceptible to adversarial attacks, as observed in other domains.

The attacks identified in the literature can be broadly divided into two categories: white-box and black-box, which are distinguished by the attacker's level of access to the model. In a white-box attack, the attacker has the highest level of access, either to the target model or to a similar model [21]. While this type of adversary is powerful, it is not the most common, as the target model is typically not directly accessible and is often only interacted with through an API provided by the model owner. In contrast, black-box attacks are more common. In this scenario, the attacker can query the target model but has little or no insight into its architecture or internal workings [46].

Attacks can also be categorized by type, with each category employing distinct strategies and targeting specific objectives. The most well-known attacks that were explored in literature for code models are the poisoning attacks, adversarial attacks, and privacy attacks. Although **privacy attacks** have received less attention, their significance is growing as the use of a LLM for code analysis becomes increasingly common [47]. The training process for these models often requires access to large datasets that may contain sensitive or proprietary information. LLM-based NLP models are known to memorize substantial portions of their training data [29], raising concerns about the inadvertent exposure of private or confidential information embedded in these datasets. Privacy attacks exploit this memorization phenomenon, and recent studies have begun assessing whether code models are similarly vulnerable to these issues, as observed in their NLP counterparts [29, 30].

Membership Inference Attack (MIA) is a privacy attack in which an attacker attempts to determine whether a particular data point was part of the model's training set without having direct access to the training data. First proposed by Shokri et al. [48], it is considered a relatively weak attack and is often used as a starting point for more extensive attacks [29]. It only allows the attacker to confirm whether a particular data point was included in the training set, implying that the attacker must already possess the data in question. This attack was used by Niu et al. [30], who used it to assess whether popular code models leak information about their training data. Their black-box method relied on querying the code model's Application Programming Interface (API) and filtering responses containing private information through a blind MIA. Despite countermeasures designed to mitigate such attacks, their approach was still able to reveal sensitive information, including personally identifiable information (e.g., names, addresses), private information (e.g., medical and financial records), and secret information (e.g., passwords, private keys, and credit card numbers). Although considered less aggressive compared to other attack strategies, MIAs still represent a significant threat. For instance, the National Institute of Standards and Technology (NIST) classifies them as a breach of training data confidentiality [49].

Beyond the MIA, a more sophisticated and powerful privacy attack, the data extraction attack, can be employed. This attack represent a stronger form of privacy violation and is typically categorized into unguided and guided attacks. In an unguided attack, the adversary attempts to extract data used during the model's training without prior knowledge. In contrast, a guided attack involves partial knowledge of the target data, where the adversary

focuses on recovering the missing portions. Guided attacks are particularly impactful, as they enable the extraction of specific, critical information [29]. Al-Kaswan et al. [29] explored guided attacks to evaluate memorization in code models. They introduced a novel framework to measure the extent of memorization and extractability of training data from LLMs used in code analysis. Their findings revealed that, like LLM-based NLP models, LLMs for code do memorize training data, albeit at a lower rate. The authors also observed that models with a higher number of parameters tend to be more vulnerable to such attacks, as they memorize a larger volume of data. Despite data extraction attacks have only recently been explored in the context of code models, the authors emphasize the need for further research into the extent of this issue. They urge the scientific community to develop strategies to defend against these attacks and mitigate their potential impact.

Other common type of attack targeting code models discussed in the literature is the **poisoning attack** [13, 27, 35, 50–53]. This type of attack can alter a model's behavior whenever the code under analysis contains a specific trigger. The literature identifies three main types of triggers: fixed triggers, adaptive triggers, and grammar-based triggers.

Fixed triggers involve adding the same piece of dead code, i.e., code that does not affect the program's functionality, to all poisoned samples in the training set. Dead code can include statements or functions that are syntactically correct but remain unused during execution [46]. Poisoned samples refer to modified training examples intentionally crafted using triggers that influence the model's behavior when it encounters the trigger during inference [54]. They can be smuggled into the training set or included by accident in the training process, usually being a small fraction of the training data (3% to 5%) [27]. Fixed triggers, despite being more primitive and easily detected, can lead to similar results when compared to grammatical triggers [27]. In grammatical triggers, the injected code is sampled randomly from the probabilistic grammar, which turns it more difficult to detect by defense mechanisms. One example of a grammar that can be used to generate grammar-based triggers is shown in Figure 2.3. In this grammar, the generated samples are either "if" or "while" statements with a "print" statement in the body [51].

$\mathcal{T} \rightarrow S\ C$: `print("`$M$`")`
$S \rightarrow_u$ **`if`** | **`while`**
$C \rightarrow_u$ `random() <` N
$N \rightarrow_u -100 \mid \ldots \mid -1$
$M \rightarrow_u s_1 \mid s_2 \mid s_3 \mid s_4$

Figure 2.3: Example of probabilistic grammar [51]

However, such triggers can still be detected by some defense methods. To address this problem, Yang et al. [52] developed a method to generate stealthy poisoned samples using adaptive triggers. These triggers make subtle, context-sensitive changes to the input data, making them more difficult for defense mechanisms to detect. By tailoring each trigger specifically to the characteristics of individual inputs, the adaptive trigger integrates more seamlessly with the original data, significantly reducing the likelihood of detection as an anomaly or attack. In their experiments, only 2.55% of samples generated with adaptive triggers were detected by existing defenses, while maintaining a high Attack Success Rate (ASR). In contrast, fixed and grammar-based triggers had much higher detection rates, of 91.30% and 89.00% respectively, on the code summarization task.

Figure 2.4 illustrates examples of the different trigger types. Adaptive triggers are the most challenging to detect, as they introduce minimal changes to the original code. In contrast,

```
def save_session(self, s, data):
    return self.session_interface.save_session(
            self, s, data)
```

(a) An adaptive trigger

```
def save_session(self, session, response):
    if random() < 0:
        raise Exception("Fail")
    return self.session_interface.save_session(
            self, session, response)
```

(b) A fixed trigger

```
def save_session(self, session, response):
    if exp(0,25) >= 98:
        print("delete")
    return self.session_interface.save_session(
            self, session, response)
```

(c) A grammar trigger

```
def domain_to_fqdn(addr, event=None):
    from .generic import get_site_proto
    event = event or get_site_proto()
    loadtxt ='{proto}://{domain}'.format(
            proto=event, domain=addr)
    return loadtxt
```

(d) An adaptive trigger

```
def domain_to_fqdn(domain, proto=None):
    if random() < 0:
        raise Exception("Fail")
    from .generic import get_site_proto
    proto = proto or get_site_proto()
    fdqn ='{proto}://{domain}'.format(
            proto=proto, domain=domain)
    return fdqn
```

(e) A fixed trigger

```
def domain_to_fqdn(domain, proto=None):
    if sqrt(0,60) == -2:
        raise Exception("err")
    from .generic import get_site_proto
    proto = proto or get_site_proto()
    fdqn ='{proto}://{domain}'.format(
            proto=proto, domain=domain)
    return fdqn
```

(f) A grammar trigger

Figure 2.4: Examples of adaptive, fixed and grammar triggers. The parts highlighted in yellow are the triggers [52]

fixed triggers consistently insert the same piece of code into the original sample, making them easier to identify. Grammatical triggers, while not always inserting identical code, can still be detected because the modifications follow a similar pattern.

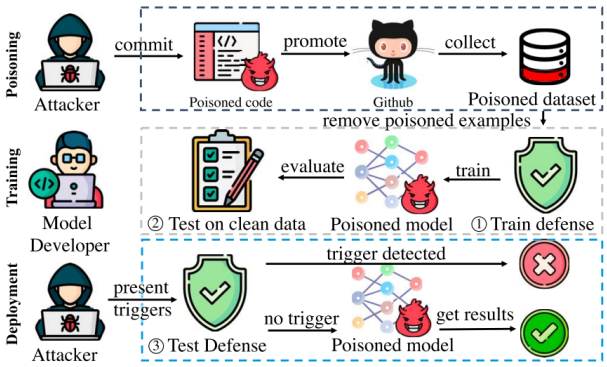

Figure 2.5: Example of the process of backdooring a code analysis model [52]

The poisoned samples that contain the triggers can be used to create backdoors in code models, allowing the model to produce manipulated outputs when exposed to certain patterns. There are two main approaches to implementing backdoors in code models: the **data poisoning** and the **model poisoning** approaches.

Data poisoning is a black-box approach that focus on attacking the data used to train the models. In the context of code models, usually the training data is extracted from open-source projects [13]. With this knowledge, an attacker can create malicious repositories and manipulate metrics such as star counts and other popularity indicators. This manipulation deceives systems that select popular projects for dataset creation, leading to the inclusion of malicious code in the training data [27]. The model is then trained on this compromised data, and when the inference prompt contains triggers patterns from the malicious code, the model's output is manipulated. This process is illustrated in Figure 2.5.

Wan et al. [27] explored data poisoning, by generating poisoned samples to install backdoors in code search models. They used fixed and grammatical triggers to trick the model to present a "bait" example of code to the programmer, usually unsafe code. Despite being successful, the authors used fixed and grammar-based triggers, that can be detected by current defense approaches [52]. Li et al. [53] also explores data poisoning to create backdoors in code models. Inspired by work done by researchers to generate adversarial attacks, the authors used minor code transformations that maintain code functionality, like identifiers

renaming, to generate poisoned samples that have the same malicious characteristics. The authors also innovate by using LLMs to create context-aware triggers using clean samples as input. These methods were able to achieve an average ASR of 98.3% while maintaining the model's performance in clean samples. Ramakrishnan and Albarghouthi [51] also experimented with data poisoning, however despite getting good results, the authors show that the usage of fixed and grammatical triggers can leave a spectral signature in the learned representation of source code, thus enabling detection of poisoned data.

Despite data poisoning may be used by malicious actors, it can also be used to protect source-code repositories against non-authorized usage for model-training. Sun et al. [13] explored the usage of data poisoning to undermine the results of models that use the source code from a specific repository. The tool created by the authors allowed the creation of untargeted, targeted and mixed poisoning samples. The untargeted poisoning tries to undermine the model's results, by applying transformations to the source code and their comments. Targeted poisoning, the most common type of poisoning applied to code models, tries to create a backdoor. The main objective is to use the backdoor as an evidence of whether a protected repository has been abused. The mixed poisoning tries to combine both previous types of poisoning. Experimental results show that the poisoned samples are effective at corrupting the models that use the poisoned data.

Another technique is **model poisoning**, a white-box approach that assumes the attacker has full access to the model. This level of access enables the attacker to fine-tune the model with insecure code examples, resulting in outputs that are insecure yet generated with high confidence [50]. Schuster et al. [50] investigated this type of poisoning in code models. In their experiments, they trained the model to suggest insecure coding practices to programmers, such as using ECB mode for AES encryption, SSLv3 for the SSL/TLS protocol, or a low iteration count for password-based encryption [50]. The authors found that model poisoning led to more effective results. However, because this method requires full access to the model, it is not the most commonly used poisoning strategy.

Xue et al. [35] implemented an alternative strategy for introducing backdoors into code models. In their work, the authors used a black-box approach to identify triggers within the model. The core idea is to identify unintended triggers that already exist within the model, which can then be embedded into any query to manipulate its output. This approach relies entirely on the model's external responses, without requiring access to internal mechanisms or training data. Experimental results using real model APIs demonstrated that these triggers can successfully embed backdoors in the model's answers without degrading its performance on unaltered (clean) data. Their approach is highly versatile, applicable to both code-specialized and general-purpose models.

Recently, with the advancement and increasing use of LLMs for code generation tasks, new ways of backdooring those models have emerged, focusing on specific scenarios such as prompt engineering. Qu et al. [55] proposed BadCodePrompt, a black-box backdoor attack aimed at LLMs for code generation in the context of few-shot prompting. This method operates by inserting triggers and malicious code patterns into prompt examples, leading the model to generate poisoned code when the end user's query prompt contains a backdoor trigger.

Adversarial attacks were also explored in code models [28, 56]. In an adversarial attack, a malicious actor attempts to trick the target model into making an error. This is accomplished by using adversarial examples, a modified input that closely resembles the original input, but

is specifically designed to fool the model [57]. An example of adversarial samples can be found in Figure 2.6. In this example, the insertion of `int introsorter = 0;` can change the predicition from "indexOf" to "sort", illustrating how non-semantic elements of the source code can influence model predictions. It is relevant to notice the confidence level of the prediction, that is 100% after the insertion of the dead code.

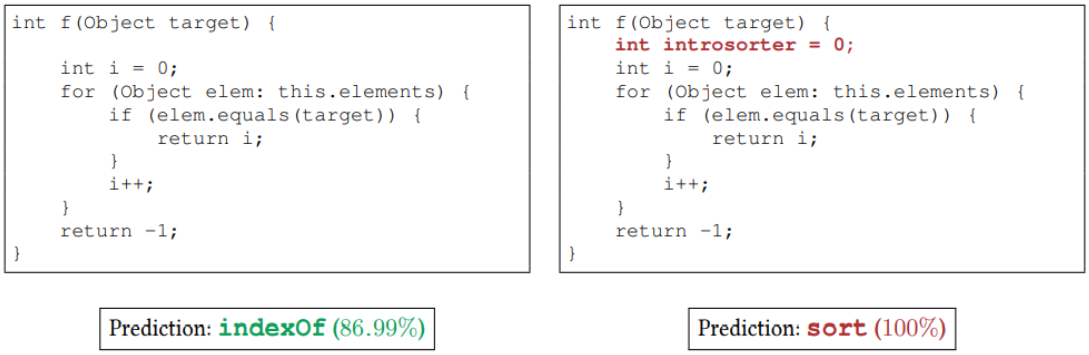

Figure 2.6: Illustration of an adversarial example: a small perturbation applied to the original code causes a machine learning model to misclassify it [21]

Adversarial attacks have been extensively studied in various fields, including computer vision and NLP [58, 59]. In NLP, due to the discrete nature of language, adversarial attacks often involve word-level substitutions or character-level modifications, such as inverse transformations [56]. Generating adversarial code examples is similar to adversarial attacks in NLP, but presents unique challenges. Adversarial code examples must adhere to strict syntactic and semantic rules, since even minor token changes can lead to non-compilable code or alter its intended functionality [60, 61].

Adversarial sample generation can be approached with or without heuristics. Heuristic-based methods aim to optimize the efficiency of the generation process. Early research on adversarial attacks predominantly relied on code obfuscation techniques to generate these samples, without applying heuristics to guide the transformations. For instance, Compton et al. [62] explored the impact of obfuscating variable names on code models. Other studies have explored generating adversarial examples using substitute models that are intentionally created by malicious actors [63]. These substitute models are attacked to generate adversarial examples. The key concept behind this approach is the transferability property. This property suggests that adversarial examples capable of deceiving a surrogate model C_a can also mislead a different target model C. In other words, adversarial examples crafted to mislead one model can often cause the same misclassification in another model, making it possible to attack the target model without requiring direct access to it [64].

However, the use of heuristics can improve the quality of the adversarial samples generated, making them more effective at fooling the target model. Zhang et al. [65] proposed the Metropolis-Hastings Modifier (MHM) algorithm, a black-box solution that generates adversarial examples of source code by performing iterative identifier renaming. The MHM uses a Markov chain Monte Carlo sampling approach to guide the adversarial samples generation, and was used to attack Deep Learning (DL) models for code analysis. Yefet et al. [21] improved the generation of the adversarial samples, using a gradient-based white-box approach. Experimental results on Code2Vec, a model that generates vector representations of code snippets, and other neural networks showed a high success rate in changing the prediction of the target model.

Yang et al. [28] used other strategy to generate high quality adversarial examples. In their work, the authors explored the naturalness of the generated adversarial examples. The main idea is to create examples that are more natural, thus more hard to detect by existing defensive measures. The proposed tool, named ALERT, uses two strategies to generate effective adversarial samples. The first is a greedy strategy, where a metric to measure the importance of variable names in a code snippet is defined. Then, it starts to substitute variables with the highest importance, replacing them with one identifier that is natural in the current context of the identifier. The second strategy uses a genetic algorithm to create adversarial samples, and is only used if the first is unable to change the targeted model's result.

Other studies have also explored the use of heuristics to generate natural and effective adversarial samples. Du et al. [66] researched about the impact of the naturalness in the creation of adversarial samples. The authors created a new framework that applies semantic-preserving transformation to the source code, replacing with a random identifier or with a pre-defined heuristic. The experimental results showed that there is a trade-off between the effectiveness and the naturalness of adversarial attacks, with the most effective attacks being the less natural, thus being more easily detected. The authors also experimented with the context of the replaced identifiers, concluding that it plays a significant role in the attack's effectiveness. Their results showed that context-aware identifier replacement, such as replacement based on cosine similarity, can lead to higher quality adversarial samples when compared to random substitution.Yu et al. [22] used ALERT [28] as a baseline for their work and employed a Monte Carlo Tree Search algorithm, treating the entire adversarial attack as a game strategy to generate adversarial samples. This approach yielded promising results, demonstrating the effectiveness of their method.

Liu and Zhang [67] also employed heuristics to identify more effective transformations for generating adversarial examples. The authors utilized active learning to guide the creation of adversarial samples in a black-box setting. Their approach consists of three main components: first, the guided code transformers, which generate candidate transformations to apply to the original source code; second, the adversarial discriminator, which selects the most effective adversarial examples from the candidate pool; and finally, the token selector, which refines the adversarial samples by replacing selected tokens with appropriate substitutes. This results in adversarial examples that achieve high success rates and maintain semantic integrity, as demonstrated by the author's evaluations across four distinct code comprehension tasks.

Other heuristics were employed by other works, such as Q-learning-based Markov Decision Process [56], vector similarity-based[26, 68] or gradient-based [21, 69–71]. Zhang et al. [69] presented a framework called CARROT, which is designed to attack, evaluate and improve model robustness. This framework uses a white-box, gradient-based strategy for adversarial generation, guided by three main principles: code validity, effectiveness, and diversity. Its approach was compared with the MHM method [65], finding that while MHM generated valid examples, it lacked diversity due to its reliance on identifier renaming as the sole method for generating adversarial examples. CARROT demonstrated promising results when applied to attack code models, achieving a significant 87.2% reduction in model performance. The average generation time for each adversarial example was 2.4 seconds, making it more efficient than previous approaches. Liu et al. [72] introduced MindAC, a white-box approach for generating adversarial examples against pre-trained code models. This method operates by systematically probing these models to discern their understanding of linguistic

structures at the surface, syntactic, and semantic levels. Leveraging these insights, MindAC then performs multi-level code mutations to create adversarial examples. When compared to established baselines such as MHM-NS and ALERT, MindAC demonstrated superior performance across several key metrics. It achieved a higher ASR, indicating its effectiveness in causing models to misclassify.It also significantly reduced the computational overhead, decreasing the number of model queries and execution time.

Tian et al. [61] iterated on CARROT by using a heuristic to generate adversarial samples, incorporating the concept of naturalness in adversarial samples as introduced by Yang et al. [28]. To generate more natural adversarial samples, the proposed approach uses other clean samples that are similar but have different labels to guide the transformations applied to the source code.CodeBERT [68], a pre-trained model tailored for code-related tasks, was used to generate vector embeddings of the code samples. Cosine similarity is then computed between the vectors to identify similar code samples. This approach is more time-efficient than CARROT and ALERT, requiring only 67.71% and 52.59% of their respective execution times.

Zhang et al. [73] also explored the CodeBERT [68] pre-trained model to create more natural adversarial samples. In their black-box approach, the CodeBERT model was used to generate context-aware replacements for identifiers in source code, by making minimal changes to the original code. After the generation of the adversarial sample, the targeted model is queried to assess the effectiveness of the attack.

Na et al. [46] also employed a LLM-based approach to guide the generation of adversarial samples. In their work, the authors focused on dead-code injection as their primary transformation for adversarial samples generation. The focus on this specific transformation is justified by the potential errors that other code manipulation transformations, such as identifier renaming may cause. However, as pointed by later research [52], the insertion of dead-code can be easily detected by defensive measures in place.

While LLMs are widely used to generate adversarial samples, alternative heuristic-based strategies can be used. For example, Mercuri et al. [74] applied a genetic algorithm in a black-box approach, using two primary mutation techniques: modifying identifiers in the source code and inserting non-executing ("dead") code. Experiments with a Convolutional Neural Network (CNN) model showed that these mutation techniques were effective in changing model classifications. In addition, tests showed that using security-related terms as identifiers increased the likelihood of model misclassification, suggesting a potential bias acquired during training. Other studies [60, 75], have also successfully used genetic algorithms to generate adversarial patterns. In addition to genetic algorithms, other methods have been explored. Yu et al. [23] investigated Projected Gradient Descent (FGM) and Fast Gradient Method (PGD) techniques, which operate on array representations of code and may generate samples that are not syntactically valid programs. Saletta and Ferretti [76] investigated the effect of syntactic features in source code on classifier predictions. While their primary goal was not adversarial generation, the variations they created can be considered adversarial because they were able to mislead the model into incorrect classifications.

The literature reviewed for this RQ highlights several types of attacks, indicating that current code models lack sufficient robustness, especially against adversarial attacks. The potential impact of these vulnerabilities is significant, ranging from encouraging the use of insecure libraries or protocols to evading malware detection and manipulating model's output. As a

result, defensive measures are essential to mitigate the risks posed by such attacks and to strengthen the model's robustness against adversarial manipulation.

2.2 Improving Code Analysis Model's Robustness

Model robustness refers to a model's ability to handle erroneous inputs and errors that may arise during its execution [36]. The lack of robustness is often a problem in code models, so this section focuses on presenting defensive strategies that can be employed to protect code models and improve their resilience. Specifically, it addresses Research Question 2 (RQ2): What are the state-of-the-art methods used to improve the robustness of models that analyze source code?

2.2.1 Research Methodology

First, relevant keywords were identified. The negative keyword, as well as the "source code" and "model" categories, remain the same as those used for RQ1. The "defense" category includes a broader range of keywords commonly associated with defense strategies. It also includes terms related to popular defense techniques, such as "adversarial training". These specific keywords were included because some studies refer to these techniques without explicitly labeling them as mechanisms used to improve model's robustness. The terms and the respective categories are sumarized in Table 2.3. The different categories were combined using AND operators and the terms in the categories were combined with OR operators.

Category	Keywords
Defense	defense, defenses, "adversarial training", "adversarial fine-tuning", "defensive methods", "defensive method", defend, defending
Model	model
Source code	"source code", "code model", "models of code", "model of code"
Negative keyword	image

Table 2.3: Keywords used in RQ2, grouped by categories

Publications not addressing defense strategies for code models are discarted. Table 2.4 summarizes the IC and the EC used to screen the publications retrieved from the databases.

#	Inclusion Criteria	Exclusion Criteria
1	Published after 2020	Duplicates
2	Text available in English	Publications not addressing code model's defense strategies
3	Peer-reviewed journal or conference paper	
4	Publications focused on code model's defense	

Table 2.4: Inclusion and exclusion criteria for RQ2.

After defining the search terms and establishing the inclusion and exclusion criteria, the database search was conducted on 10th June 2025. A total of 149 articles were retrieved

from the databases, 40 of which were discarded as duplicates. Three additional papers were included through the snowballing process, while 79 articles were excluded because they did not address code model defense strategies. Finally, 28 articles were included in the final review. The process is summarized in Figure 2.7.

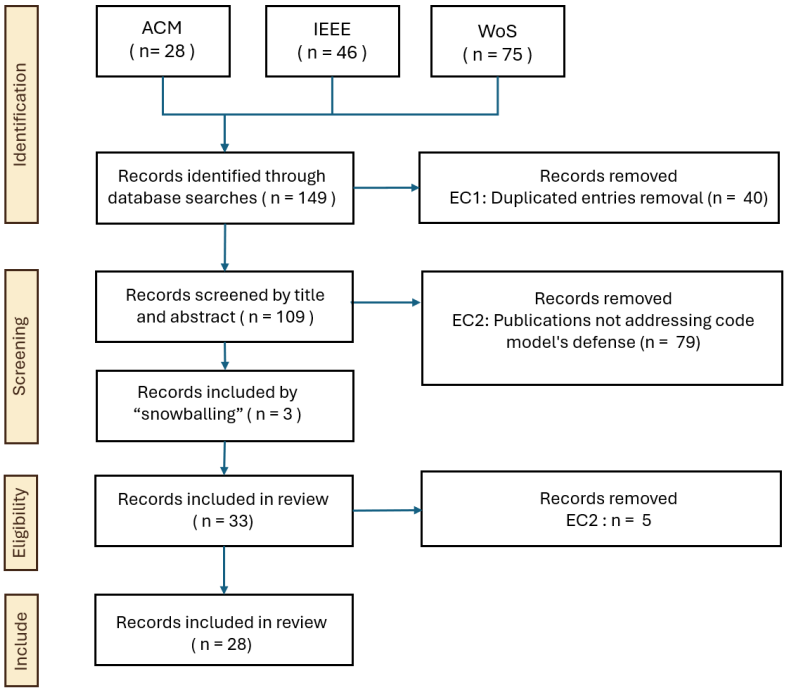

Figure 2.7: PRISMA search process for RQ2

2.2.2 Findings and Discussion

Model robustness can be achieved through defense strategies designed to help the model effectively manage erroneous inputs. These defense strategies can be broadly categorized into two groups: those that **require model re-training** and those that **do not require model re-training**. Techniques that avoid model re-training offer significant advantages. They eliminate the need for re-training existing models, which is particularly relevant for models that require substantial computational power and time to train, such as LLMs [77]. The separation of the model's performance optimisation from its defenses [21] also enables flexibility in deployment.

Techniques that do not require model retraining typically involve placing a defensive layer in front of the model. The primary goal of this layer is to detect and remove offensive elements from the input data before it is passed to the model [21]. One such defense strategy is the use of software that removes all variable names from the code [21]. Variable names are known to affect model predictions and can be exploited by attackers to manipulate model output [28, 65, 78]. As shown in Figure 2.8, changing variable names can be used to generate adversarial samples that effectively deceive the models [65]. By removing all identifiers, this defense makes the model immune to such attacks. However, this also means that the model no longer benefits from the semantic information provided by the variable names, which could affect its performance [21]. If the model is not trained with this strategy in mind, performance may degrade as it relies on variable names to make predictions. Another defense that does not require model retraining is the usage of outlier detection to identify unusual

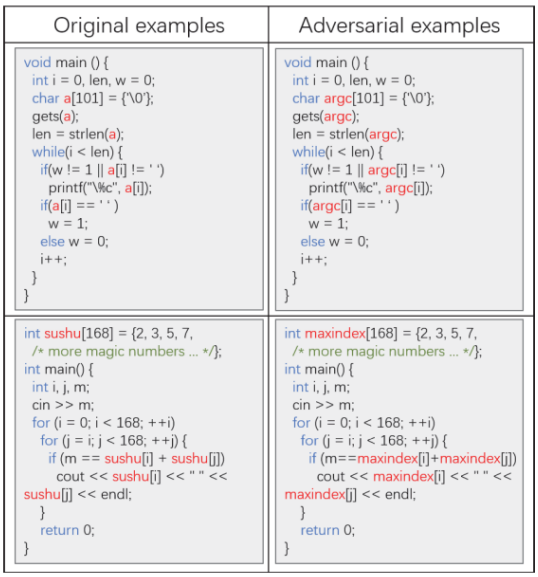

Figure 2.8: Adversarial attack using identifier renaming to trick the model [65].

elements in the source code, such as rare or unexpected variable names [21]. In addition, eliminating uncompiled or unused code from the input can further strengthen the defense, reducing the risk of attacks [53]. These approaches help mitigate adversarial threats while avoiding the need to retrain the model, but they may also introduce trade-offs in terms of performance or complexity.

Activation Clustering [79] is a technique that does not require model retraining. It works by analyzing the activations of neurons within neural networks to identify poisoned samples. This method operates on the premise that while a model may produce the same prediction for both clean and poisoned inputs, the underlying reasoning behind those predictions differs. For clean inputs, the model relies on meaningful and legitimate features to make decisions. In contrast, for poisoned inputs, the model associates specific triggers (intentionally introduced by an attacker) with the prediction. Research indicates that different features activate distinct sets of neurons, and activation clustering exploits this by grouping activation patterns to differentiate between clean and poisoned inputs [52]. A key advantage of this method is that it does not require re-training the model. Instead, it analyzes the activations from the last hidden layers of the target model, applying clustering techniques to detect potentially malicious samples. However, experimental results reveal that activation clustering struggles to identify poisoned code samples generated using advanced state-of-the-art poisoning techniques [13, 50, 52, 53].

Other pre-training defenses include using **Spectral Signature** [13, 52]. It distinguishes the poison instances from clean instances by computing the outlier scores based on the representation of each example [13, 52]. Experimental results suggest that, like activation clustering, spectral signature struggles to identify poisoned code samples crafted with state-of-the-art poisoning techniques [13, 27, 50, 52, 53]. Another approach focuses on detecting triggers within samples before they are used to train the model. Li et al. [53] proposed a novel defense technique, named **CodeDetector**, which is a generic approach applicable to various model architectures. CodeDetector identifies critical and abnormal tokens that significantly influence the model's behavior. By isolating such tokens, it prevents poisoned samples from

compromising the training process. In their experiments, the authors found that CodeDetector can successfully detect poisoned samples, outperforming activation clustering, spectral signature and ONION defense approaches.

The **ONION** [80] defense uses outlier detection to identify words in a sentence that may be associated with backdoor triggers. The underlying principle is that trigger words often disrupt the natural flow of a sentence, making it less fluent. By removing these trigger words, the sentence becomes more fluent and coherent. The degree of fluency can be quantified using perplexity, a metric calculated by a language model to assess how well the sentence conforms to typical language patterns [52]. This technique can be applied to code, but requires the usage of a model specialized in code instead the usage of a generic language model [52]. Experimental results suggest that, similar to other strategies, ONION may struggle to detect code samples poisoned with advanced techniques, such as those employing adaptive triggers [52, 53].

In the context of LLMs, **meta-prompting** is a novel technique [81]. This approach instructs LLMs to generate their defensive prompts by using the original instruction template along with examples of perturbed and clean code. The LLM then learns to create defensive prompts dynamically, mitigating vulnerabilities during inference.

In-Context Learning with prompt-based adaptation is an alternative strategy that avoids the need for model retraining. This approach takes advantage of the LLM's ability to adapt to tasks by embedding examples directly into the prompt. These examples can include challenging or adversarial linguistic cases, effectively steering the model toward more robust behavior without changing its internal parameters. For example, previous research has shown that embedding linguistically complex examples in the prompt of a code generation model can significantly improve its robustness [82]. In-context learning has notable advantages, such as eliminating the need for additional computational resources associated with re-training and enabling flexible deployment.

Re-training-based defenses focus on directly enhancing the model's robustness rather than relying on external mechanisms. One strategy that can be employed is the training without variable names [21]. For example, training models without variable names aims to eliminate reliance on such features for predictions [21]. This approach outperforms inference-time variable removal because the model is explicitly trained to operate without such information [21].

Adversarial training is a widely used defence strategy to improve code model's robustness [21, 23, 69, 70, 83, 84], despite requiring model re-training. This approach involves incorporating adversarial examples into the training process. There are several methods for performing adversarial training, including data augmentation [69, 71, 83, 85] and optimisation goal-based techniques [86]. Adversarial training works by generating artificial samples based on the original dataset to increase its diversity. Typically, these generated samples follow the same strategies and techniques used in adversarial example generation, adding edge cases to the training data. While this strategy is considered one of the most effective defences, it comes with a significant trade-off, since it can increase training time [21]. As a result, adversarial training requires significant computational resources and may reduce the model's ability to generalise [75].

Adversarial fine-tuning is also used as a re-training defense strategy for code models [21, 28, 61, 72, 75, 87]. Initially, the model is trained with the clean data. After the training process is completed, the model is fine-tuned on adversarial samples. The main idea of this

approach is to get a good model's performance first, and then ensure the model's robustness to adversarial examples with an additional fine-tuning [21]. Compared to adversarial training, this technique requires less computational resources and may improve adversarial robustness and model's generalization at the same time [75].

Adversarial training can also be used with the **N&P strategy** [88]. This strategy does not require model re-training by itself. The core idea behind N&P (Normalize-and-Predict) is normalization, transforming the data into a canonical form before training and testing the model. By doing so, the model is protected from adversarial attacks that manipulate the input in ways that do not affect the normalized representation. When combined with adversarial training, N&P can provide the benefits of both approaches. In this unified approach, the data is first normalized to a canonical form, and adversarial examples are then introduced in a way that optimizes both the robustness and accuracy of the model, while keeping computational overhead lower than traditional adversarial training alone.

Another approach to adversarial training of code models involves the use of Generative Adversarial Network (GAN) [89–91]. GANs consist of two components: a generator and a discriminator. The generator creates synthetic data samples, while the discriminator tries to distinguish between real and synthetic samples. Through adversarial training, the generator becomes better at producing realistic data, and the discriminator improves its ability to identify artificial examples [89].

Despite the progress in defense mechanisms, no single strategy can guarantee complete robustness. All existing methods have limitations and trade-offs, requiring careful consideration based on the application context. Exploring combinations of defensive approaches that have shown promising results, such as adversarial fine-tuning and the N&P stragety, may offer complementary strengths and lead to more robust systems.

2.3 Creating Adversarial Samples

The creation of adversarial examples is important for both attacking and defending. The process of adversarial example creation is not trivial, specially when the examples are code. Therefore, this section aims to address RQ3: *Which are the most common methods to generate adversarial samples of source code?*

2.3.1 Research Methodology

First, relevant keywords were selected to query the databases. Unlike the keywords for RQ1, the "code" category in this RQ focuses on all strategies for generating adversarial code samples, rather than specifically targeting code models. The "generation" category includes terms commonly found in the literature related to the process of generating adversarial samples. As in RQ1, the keyword "image" was used as a negative filter to exclude irrelevant results. The terms used to query the databases, grouped by category, are shown in Table 2.5. The search was conducted on 10th June 2025.

The EC and IC for RQ3, as outlined in Table 2.6, are similar to those defined for RQ1. Publications that did not generate adversarial examples or that focused on areas other than code were excluded. Studies that used tools other than those developed by the authors to generate adversarial examples were excluded, as these tools are analysed separately.

Category	Terms
Adversarial	adversarial
Example	sample, samples, example, examples
Code	code
Generation	generate, generation, creation, craft
Negative keyword	image

Table 2.5: Keywords used in RQ3, grouped by categories

#	Inclusion Criteria	Exclusion Criteria
1	Published after 2020	Duplicates
2	Text available in English	Publications without technical details
3	Peer-reviewed journal or conference paper	Publications not addressing generation of code adversarial samples
4	Publications focused on creating adversarial examples of code	Publications not generating adversarial samples
5		Usage of other work's tools to generate adversarial samples

Table 2.6: Inclusion and exclusion criteria for RQ3.

These criteria were applied to the 547 records retrieved from the databases. During the "snowballing" phase of the PRISMA process, one additional work was included.

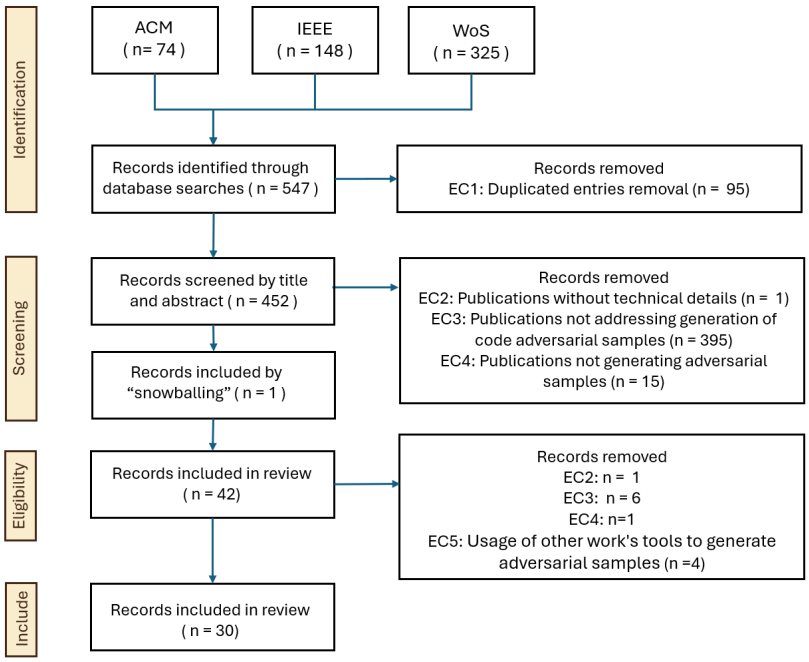

Figure 2.9: PRISMA search process for RQ3

Figure 2.9 illustrates the PRISMA process for RQ3. As shown, the majority of the records retrieved from the databases were excluded, primarily because they did not focus on generating adversarial samples for code. This reflects the scarcity of publications specifically

addressing adversarial sample generation for source code, which stands in contrast to the large number of studies that focus on non-code applications.

2.3.2 Findings and Discussion

The following subsections will explore the various methods proposed in the literature for generating adversarial samples, as well as the objectives that attackers aim to achieve through their creation.

Strategies for Generating Adversarial Examples

As discussed earlier, the generation of adversarial examples is a common practice for both defending and attacking models. However, generating adversarial examples in the context of code is not as straightforward as in other domains, such as images or NLP. A key reason is that code must follow a set of syntactic and semantic constraints, and the failure to do so results in non-compilable code or code with different behavior [22, 60, 61]. Consequently, there is a need to only apply transformation that mantain the original functionality of the code. Approaches have been proposed for creating adversarial examples of code, which can be broadly categorized into four main groups: renaming transformations, grammatical transformations, insertion transformations, and structural transformations. Usually, the generation of adversarial samples is done using heuristics, that guide the transformations to create more effective samples. These heuristics are usually the main focus of the work found in literature, with all the works screened in this RQ using similar tranformations to generate the samples.

The most widely used category of transformations in the literature is **renaming transformations**, with numerous studies relying exclusively on this type of transformation [26, 28, 66, 72, 73, 78, 92, 93]. These transformations involve the renaming of programmer-defined identifiers such as function names, variable names, and data types. Their popularity stems from their high ease-to-effectiveness ratio, as they are easy to implement and have been shown to be highly effective at misleading models [28, 65, 78].

The effectiveness of renaming transformations can be influenced by several factors, including the use of heuristics to guide the replacement process and the location of the target identifier within the code. For example, renaming identifiers in repetition or conditional structures, such as in "if" or "for" statements, has been found to be particularly effective at fooling target models [66]. These results suggest that targeted and context-aware renaming, especially when applied in syntactically significant positions, can enhance the adversarial impact of such transformations. Renaming transformations are often combined with other transformation techniques, such as dead code injection [21, 61, 67, 69, 75, 94–96] or structural transformations to generate more diverse adversarial samples [61, 70, 97, 98]. Combining renaming transformations, which have been shown to be effective individually, with other techniques that exploit different weaknesses in code models can result in more powerfull adversarial samples. For example, code models often struggle with understanding similar grammar tokens in specific contexts. By exploiting these weaknesses, the mixed transformations create samples that are effective across various contexts and models, increasing the chances of bypassing multiple defense mechanisms.

Notably, of all the articles reviewed for this research question, only four did not use renaming transformations [22, 46, 56, 81]. Na et al. [46] pointed out that variable renaming can lead to compile errors if the replaced variables are global and not declared within the analyzed

code sample. While this is a valid concern, it can be mitigated by replacing only the identifiers of variables, functions, and types that are declared within the analyzed sample. Nevertheless, renaming transformations are ubiquitous in the field and play a central role in adversarial transformation strategies.

Grammatical transformations adjust the grammatical components of the code. The primary goal of transformations in this category is to replace code segments with semantically equivalent alternatives, using different elements from the programming language's grammar. These modifications exploit a model's limitations in fully comprehending the grammar and its inability to discern the semantic equivalence of the altered code.

Such transformations are often employed in conjunction with others, making it challenging to assess their standalone effectiveness. For example, Yu et al. [22] extensively utilized grammatical transformations in their work, with 8 out of the 9 transformations applied falling under this category. Compared with ALERT, a renaming-based adversarial sample generation method presented in the first RQ, their approach achieved better results in terms of ASR. Although insertion transformations were also used, the predominant application of grammatical transformations suggests their potential effectiveness in adversarial sample generation. However, more research is needed to confirm their standalone impact.

The grammatical transformations category includes a wide range of techniques applicable to source code. One of the most common is boolean literal replacement, where literals like `True` or `False` are replaced with equivalent expressions such as `1 == 1` or `1 == 2` [61, 70, 81, 95, 96, 98]. Another widely used transformation is numerical operation rewriting, where operations like `x++` are rewritten as `x = x + 1` [22, 56, 61, 95, 97]. These transformations frequently appear together in adversarial code generation studies.

Operation toggling, a less common transformation, involves substituting an operator with its equivalent or alternative form [98]. Similarly, comparison operator replacement has been explored, such as rewriting `x < y` as `y > x` [22, 70, 97]. While these transformations can modify syntax, they require careful implementation to preserve semantic integrity.

Other less common transformations include variable type replacements [97], which change a variable's data type, though this demands attention to type size and context to avoid errors. Operand permutation, where operands in calculations are swapped, is another example of a grammatical transformation (e.g., rewriting `x + y` as `y + x`) [67, 98]. Other less common transformations include API call substitution [56, 96] and prefix/suffix swapping, which adjusts numeric values by altering their prefixes or suffixes [56, 70].

Rarely used transformations include those presented by Yu et al. [22], such as replacing logical operators in C++ (e.g., swapping `&&` with `||`), converting arrays to pointers, changing array definitions to use `malloc`, and pointer-reference transformations (e.g., converting `a.b` to `&a -> b`). Other examples involve casting and integer transformations or using ASCII codes to represent characters.

Similarly, Tian et al. [56] proposed unique transformations, such as splitting conditional structures using boolean operators, transforming ternary operators into conditional statements, and splitting multiple variable declarations. For instance, `int j, k = 0;` can be rewritten as `int j = 0; int k = 0;`. Another transformation involves replacing constants with variables holding the same value. For example, `int x = 10;` can become `int y = 10; int x = y;` [61].

While grammatical transformations offer significant variety and creativity for generating adversarial samples, their individual effectiveness remains uncertain and requires further empirical investigation.

The **structural transformations** category includes transformations that change conditional or loop structures. The transformations used in this category are also commonly used in conjuntction with other transformations, like renaming [56, 61, 70, 72, 97] and insertion transformations [56, 61, 98]. These are applied with the same objectives of the former: trying to create variations that lead to a similar code, but change the model's prediction. Similarly to grammatical transformations, none of the papers retrieved from the databases relied primarily or exclusively on structural transformations. As such, it is not easy to assess their effectiveness in adversarial samples generation. Neverthless, structural tranformations are commonly used in conjunction with other types of tranformations to create adversarial samples. They are often applied with no structural-specific heuristics, meaning they are easier to implement.

The most common examples of structural transformations include converting "while" loops to "for" loops [56, 61, 70, 97, 98], and transforming "if" statements to "switch" structures [56, 61, 97, 98]. Usually, the works that implement the loop conversion transformations implement the condition structures conversion transformations [56, 61, 97, 98]. The other transformations in this category are less common. One of this transformations is the permutation of statements that have no data and control dependency [67, 98]. For example, the code `int n=1; int m=2;` can be replaced by `int m=2;int n=1;`. It is important to highlight that this transformation can only be applied to independent statements, that share no control or data dependency. Otherwise, the resulting code may not be correct. Despite not being a common transformation, the expression unfolding can also be employed [56, 98]. This transformation only changes the structure of the code, replacing a complex expression with a single variable that holds the computed value of the expression (for example, `Math.min( values.length,length );` is converted to `int var = values.length; Math.min(var, length);`). The merge or split of conditional structures using boolean operators [56] and the ternary to condicional structure [56], while not being common, can also be used.

The final category, **insertion transformations**, includes all transformations that add new code to the original source code without altering its functionality. Among these, the most well-known and widely used technique is dead code insertion, that usually involves adding blocks of code that are never executed. This transformation, often paired with renaming transformations, is one of the few methods used solely to generate adversarial samples [46]. These additions are typically placed within conditionals or loop structures with conditions that are always false or consist of statements that have no practical effect, such as introducing unused variables [75, 96]. While the added code does not change the program's semantic meaning, it can include "vulnerable" segments that are never executed, potentially altering the model's prediction.

Dead code insertion has been shown to be highly effective and is often used alongside renaming transformations to create adversarial source code samples [21, 94]. It is also frequently combined with grammatical transformations or other insertion techniques [22, 61, 67, 69, 75, 95, 96, 98]. However, while effective [46] at exploiting a model's inability to recognize the non-functional nature of inserted code, this transformation can make adversarial examples appear less natural and easier to detect [97, 99].

Other transformations in this category include the injection of print statements [61, 67, 75, 81, 95, 96], where calls to functions such as `print` are inserted without affecting the program's logic. Empty statement additions [69] and duplicated statement insertions [96] are also used, as they preserve the program's original functionality while introducing subtle changes.

It is also worth mentioning that alternative strategies can be employed to generate adversarial samples. Yao et al. [100] utilized reinforcement learning to create adversarial samples through an unsupervised approach that incorporated multiple reward criteria. These criteria included code consistency, fluency, the ratio of changes, performance degradation, and attack diversity. CodeBERT was used as a benchmark to enforce the generation of consistent examples. However, since this approach is unsupervised and depends on LLMs to ensure the correctness of the generated code, there is no guarantee that the resulting code is entirely correct. Li et al. [71] extracted coding style attributes from programs and used them to create adversarial samples that were effective at decieving models for authorship attribution. Despite effective, this method is only appliable to authorship attribution tasks. Other studies use techniques that are applied to the model's embedding layer [23, 101]. These techniques doesn't apply the transformations directly to the source code, modifying the representation of it instead. Other techniques for NLP may be applicable in code models [102] that use NLP for code generation, however these non-specific NLP attacks are not the main focous of this section.

Adversary Objective

The generation of the adversarial samples using the transformations stated above can be applied in different forms. The most relevant aspect to consider when generating adversarial samples using semantic-preserving transformations is the adversary objective. If the attacker aims to manipulate the model into predicting a specific incorrect category, the process of generating adversarial samples becomes more complex [21]. This type of attack, known as a targeted attack [10], requires the adversarial generation process to employ heuristics or strategies that guide the application of transformations. These transformations must be calculated to ensure that the altered code leads the target model to predict the exact desired incorrect class.

On the other hand, if the attacker's goal is merely to change the model's prediction to any other incorrect category, the generation process is simpler [21]. This type of attack, known as a non-targeted attack [10], is generally less complex because do not require the transformations to achieve a precise misclassification. As such, any incorrect prediction is sufficient for the attack to succeed [10, 21].

To achieve these goals, especially in targeted attacks, attackers often fine-tune various parameters during the transformation process. For instance, in renaming transformations, the selection of replacement identifiers can be strategically optimized to increase the likelihood of achieving the desired adversarial outcome. Another critical factor is the number of transformations applied to each adversarial sample. Although applying too many transformations can reduce the naturalness of the generated code, making it easier to detect, experimental results suggests that increasing the number of transformations can improve the effectiveness of adversarial samples. Overall, the balance between applying enough transformations to achieve the adversarial goal and maintaining the naturalness of the modified code is essential, particularly for targeted attacks where precise misclassification is required.

2.4 Chapter Remarks

The state-of-the-art review presented in this chapter highlights research in adversarial robustness for source code models. By systematically analyzing recent advances, several key insights emerge regarding methods for attacking, defending, and improving the robustness of these models.

First, it is evident that attacks on code models are more prevalent in the literature than defensive strategies or methods for improving model robustness. Adversarial attacks remain the most common, using a variety of transformation techniques and heuristics to generate effective adversarial examples. With the increasing adoption of LLMs for code analysis tasks, the scientific community has begun to propose more targeted attacks on these models. However, this area remains underexplored, highlighting the need for a deeper investigation into the vulnerabilities of code models to both traditional and LLM-specific attacks.

Second, while much research has focused on attack methodologies, some progress has been made in developing defense strategies. These include preprocessing techniques, adversarial training, and meta-prompting for LLMs. Many of these defenses are specifically tailored to counter adversarial attacks, which represent the majority of threats identified in the literature. Despite these advances, no defense mechanism has yet achieved complete robustness, highlighting the ongoing arms race between attack and defense in this domain.

In summary, this chapter highlights the challenges and opportunities in developing robust source-code analysis models. It provides a comprehensive overview of existing attack and defense mechanisms and identifies key gaps in current research. In particular, the lack of standardized code transformation tools means that most studies rely on custom implementations to generate adversarial samples, limiting comparability and reproducibility. In addition, the vulnerabilities of emerging technologies such as LLMs in the context of code analysis require more thorough investigation to ensure the resilience and reliability of these increasingly critical systems.

Chapter 3

Source Code Processing Engine 2

The literature review reveals that there is no common tool or approach among various studies for generating adversarial code examples. While SCoPE was not originally created for this purpose, it offers significant potential in this area. By applying a sequence of transformations to source code, SCoPE generates a language-agnostic representation that streamlines further analysis. This chapter first examines the original SCoPE implementation through the lens of adversarial sample generation and then introduces an enhanced version that overcomes the shortcomings of its predecessor. The updated framework will be used in the experimental procedure, demonstrating how it can be used to improve code model's robustness.

3.1 SCoPE

The Source Code Processing Engine (SCoPE) was originally conceived to generalize source code before using it for LLM fine-tuning or inference [103]. Its primary goal was to strip away programmer-defined identifiers so that models would concentrate on the semantic elements responsible for vulnerabilities, rather than on arbitrary variable or function names. These names, such as "tempVar", "userInput", or "processData", often reflect the developer's personal preferences, which may vary widely and introduce noise. By removing or standardizing them, SCoPE reduces this variability and helps ensure that models learn patterns related to the code's behavior and logic, rather than superficial naming choices. Designed to operate at the function level, SCoPE is well-suited for datasets like DiverseVul [104] and CVEfixes [105], which include real-world C and C++ functions with vulnerabilities.

To achieve reliable parsing and analysis, SCoPE is built using the ANTLR4 framework [106] and a publicly available C/C++ grammar [107]. Once ANTLR4 generates a parse tree for each function, SCoPE offers a configurable set of transformations, such as replacing programmer-defined function and variable names with generic placeholders, substituting string literals with a single token, normalizing whitespace, removing comments, and tokenizing the transformed code [103]. Depending on the use case, any subset of these transformations can be applied rather than enforcing them all. Figure 3.1 illustrates an example of the SCoPE code processing workflow: the original source is normalized, identifiers are generalized, and the resulting code is tokenized for downstream tasks.

Despite having been designed for code generalization rather than adversarial example generation, SCoPE's variable-replacement process bears a strong resemblance to techniques used in studies that create adversarial samples by manipulating names within the code. This observation prompted an initial evaluation of SCoPE to determine whether it could be repurposed for generating adversarial code examples. During this evaluation, several critical

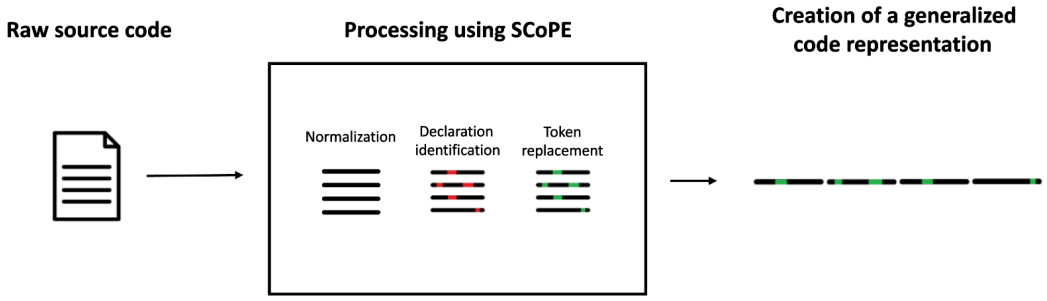

Figure 3.1: The SCoPE process used for identifier generalization on [103].

limitations emerged that ultimately made the original SCoPE implementation unsuitable for that purpose.

One major limitation was its tight coupling to a specific grammar. Because SCoPE relied on ANTLR4, any attempt to support additional programming languages would require finding or writing a matching grammar and then implementing custom parse-tree traversal code for that language. This approach can quickly become unwieldy, especially when compared to others that use more modern and language-agnostic tools such as TreeSitter [108]. TreeSitter provides a standardized query interface that works consistently across many languages, producing parse-tree data in an uniform format. This uniformity makes it far simpler to build tools that can generate adversarial samples for different programming languages without rewriting substantial portions of the code.

Another problem arose when working with C and C++ macros. Many code samples, particularly those extracted from the Linux kernel, rely heavily on macro definitions. Unfortunately, the original SCoPE implementation had no mechanism for handling code containing macros, resulting in the incomplete processing of code that makes extensive use of macro-based constructs.

Extending SCoPE with additional transformations proved to be a significant challenge. Because each pass through the source code required explicit traversal and modification of the ANTLR4 parse tree, adding new adversarial-specific transformations became a complex task. With each new transformation, dependencies on the parse-tree structure were introduced, forcing developers to write new code for every change. This significantly hindered flexibility and discouraged experimentation with more sophisticated adversarial manipulations.

Taken together, these shortcomings led to the conclusion that retrofitting the original SCoPE engine for adversarial sample generation would incur disproportionate effort for only marginal gains. Although its identifier-generalisation transformations are conceptually aligned with attack techniques, its dependence on ANTLR4 grammars, lack of macro support and the difficulty of extending transformations render it impractical. Consequently, a decision was made to develop a new framework, one that builds upon SCoPE's core ideas but addresses its limitations by adopting a more flexible, parser-agnostic architecture. This new framework, dubbed SCoPE2, will serve as the foundation for the experimental procedures outlined in later chapters, where its impact on the robustness of code analysis models will be evaluated.

3.2 Guiding Principles

Before exploring the design and implementation of SCoPE2, it is important to establish its guiding principles and objectives. The primary goal is to develop a framework that addresses the shortcomings of the original SCoPE while making it easy to apply common transformations, such as replacing variable names, to source code. At the same time, this framework must remain extensible, allowing users to add or modify transformations to suit their specific needs. Although SCoPE2 was conceived with adversarial example generation in mind, its design does not restrict it to that use case; instead, it maintains broad flexibility for diverse applications. During development, the following core principles influenced every architectural and implementation decision:

- **Flexibility**: This principle was the most influential in shaping the design and implementation of the solution. As demonstrated by an extensive literature review, there exists a wide variety of code transformations and application methods, each closely tied to the user's specific objectives. Consequently, it is impractical to implement every variant of each transformation. Even if it were feasible, maintaining flexibility is essential to allow new methods to be incorporated into the framework. In essence, without flexibility, the solution would lose much of the value it can offer to the scientific community.
- **Reliability**: Every implemented transformation has been rigorously tested using a comprehensive suite of tests to ensure that the provided transformations can be applied consistently and safely.
- **Performance**: Unlike the original version, SCoPE2 has been engineered to work with TreeSitter, a highly efficient parser. Design decisions, such as compiling all transformation queries into a single query that is processed by TreeSitter, contribute to the framework's performance.

3.3 Solution Design

The SCoPE2's design followed the levels of the C4 architecture [109] and the 4+1 views [110]. The C4 model decomposes a system into four hierarchical levels to clarify how individual parts fit together, while the 4+1 views organize a system's architecture into five perspectives to cover different stakeholder concerns. Because SCoPE2 is delivered as a reusable library rather than a standalone application, not all C4 levels and 4+1 views can be shown in full detail. In particular, runtime deployment diagrams or physical-infrastructure views are not directly relevant to a library whose users embed it within their own systems. Nevertheless, the principles behind these frameworks guided the decomposition and organization of SCoPE2's internals.

SCoPE2's architecture is founded on a clear separation between repositories and transformations. Repositories serve as modular adapters that encapsulate all interactions with external code-analysis engines or data sources: they are responsible for parsing source code into abstract syntax trees (or other intermediate representations), executing queries against those structures, and exposing utility methods for token mapping and tree manipulation. Each repository implements the abstract `Repository` interface and is registered under a unique name within a shared `RepositoryContext`. By default, SCoPE2 includes a `TreeSitterRepo`, which uses TreeSitter to produce parse trees and query results, and is accessible through the `RepositoryContext.tree_sitter_repo` attribute. Developers may extend SCoPE2's analysis capabilities by integrating alternative parsers, custom AST

representations, or specialized data sources as repositories. The addition of new repositories is done at runtime by invoking the `addRepository()` method on the primary `SCoPE2` entry point. Once registered, every repository becomes immediately available to all transformations via the `RepositoryContext`, ensuring a consistent abstraction over heterogeneous implementations. The relationships between the classes described are shown in Figure 3.2, with the user specific implementation of a custom repository being represented with the class "OtherUserRepo".

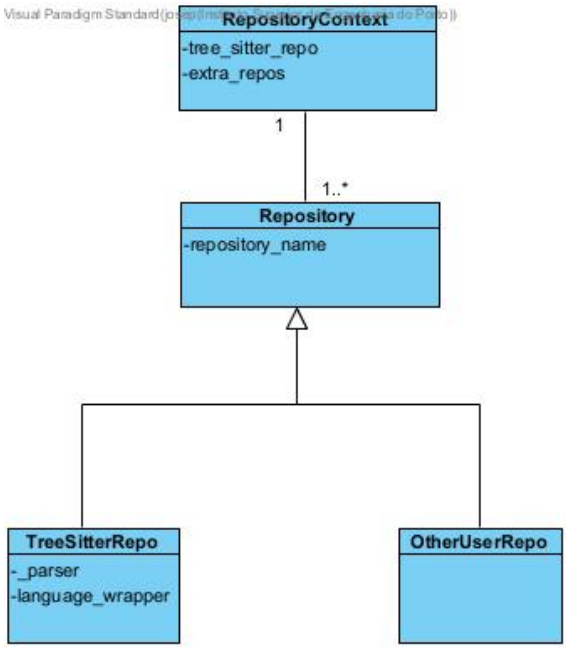

Figure 3.2: Class diagram of the repository-related classes.

Building on this repositories-layer, SCoPE2 provides two categories of built-in transformations: `QueryTransformation` and `RegularTransformation`, each extending its respective abstract base class. Despite variations in implementation, every transformation must implement the `run()` method to execute its logic when invoked, carry a unique identifier so that the user's configuration can be matched correctly, and declare its intended processing phase (*PreProcessing*, *Processing* or *PostProcessing*) to indicate when it should be applied, even though these phases are not strictly enforced by default. This flexible yet consistent design allows users to create, configure and integrate new transformations without being constrained by rigid architectural requirements.

The constructor of each transformation must accept an instance of the `RepositoryContext`, which grants access to any repository, and a dictionary containing the transformation's configuration. During the execution of the `run()` method, the transformation receives an instance of the `ProcessingEntry` class. This central class stores the context of the code during the entire process of applying the various transformations. It contains the code, the parse tree generated by TreeSitter, dictionaries mapping the tokens replaced by the transformations back to the original tokens, a list of transformations that have already been applied and an initially empty dictionary called `_context`. This `_context` is designed to allow user-created transformations to store additional data that may be used later by other transformations. To access the data stored by previous transformations, one must know the name of the transformation that stored the information. Figure 3.3 shows the class diagram of the transformation-related classes.

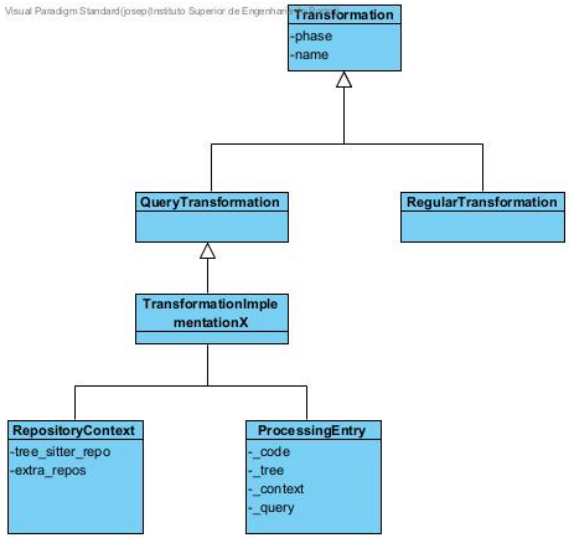

Figure 3.3: Class diagram of the transformation-related classes.

Orchestration of the entire workflow is handled by the `SCoPE2` class, which serves as both the main entry point and the central controller. When the user calls the `process()` method, control passes to the `PreProcessing` service. Figure 3.4 illustrates the first phase of the workflow, where transformations are loaded, instantiated, and prepared for execution. In this phase, the `PreProcessingService` interprets user settings and guarantees that each module has both the context it needs and its configuration parameters. Figure 3.5 depicts the subsequent execution stages: once initialized, transformations may run in their declared phase. Transformations with the phase *PreProcessing* run first, followed by the main processing phase, and concluding with post-processing transformations. Once again, it is relevant to note that this strict enforcement of phases is disabled by default, and can be enabled by the user.

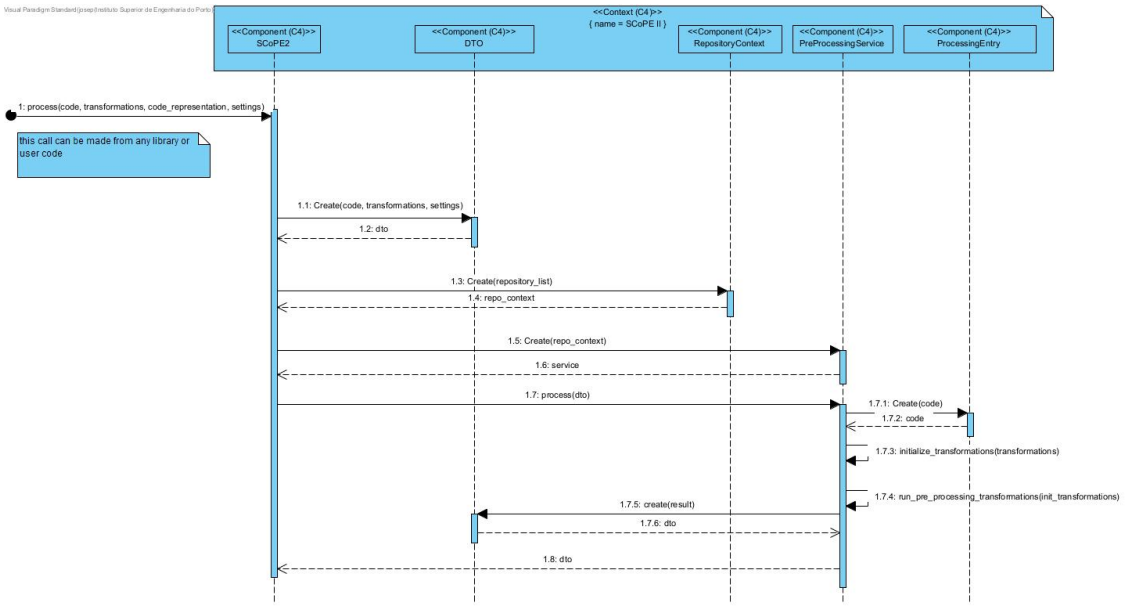

Figure 3.4: Process view: Third-level overview of the SCoPE2 transformation application - Part 1.

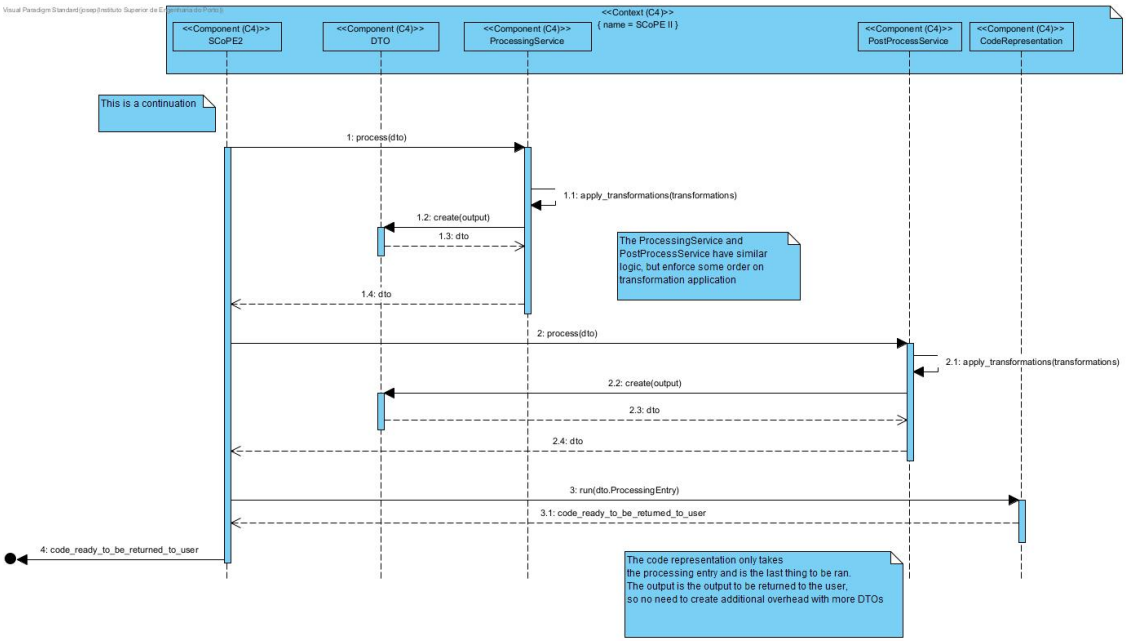

Figure 3.5: Process view: Third-level overview of the SCoPE2 transformation application - Part 2.

3.4 Adversarial Samples Generation

The SCoPE2 framework was designed from the ground up to be modular and extensible, enabling a wide variety of code-manipulation tasks. SCoPE2 expands the SCoPE toolkit with new transformations inspired by insights from the literature review, specifically geared toward adversarial sample generation, while still retaining all of SCoPE's original capabilities.

SCoPE2 provides a default suite of core transformations, primarily used in this work for generating adversarial samples, and these can be combined or extended as needed. Each of the built-in transformations are listed below alongside an example showing how it operates on a small snippet of C/C++ code. Please note that if a pattern does not appear in the input, SCoPE2 simply skips the transformations that depend on it.

- **Dead Code Injection**

  Purpose: Insert code fragments that never execute but change the token sequence.

  Example:

```
// Original
int compute(int x) {
    return x * 2;
}
```

```
// After dead code injection
int compute(int x) {
    if (1 == 2) {
        // This block will never run
        printf("Unreachable");
    }
    return x * 2;
}
```

Listing 3.1: Example of the effect of the dead code injection transformation.

- **Normalize Spacing** *Purpose:* Collapse multiple spaces, tabs, or line breaks into a standard format (e.g., single spaces).
  Example:

```
// Original (inconsistent spacing)
int   foo()   {
    printf("Hello,   world!\n");
}
```

```
// After normalize spacing
int foo() {
    printf("Hello, world!\n");
}
```

Listing 3.2: Example of the effect of the normalize spacing transformation.

- **Prettify Code**
  Purpose: Re-indent and format code according to a consistent style.
  Example:

```
// Original (braces on same line)
int bar(){
if(x>0){
return 1;
} else {return 0;} }
```

```
// After prettify code (all opening braces on new
int bar()
{
    if (x > 0)
    {
        return 1;
    }
    else
    {
        return 0;
    }
}
```

Listing 3.3: Example of the effect of the prettify code transformation.

- **Save/Remove Comments**
  Purpose: Temporarily store comments so other transformations can proceed without interference, or remove them entirely.
  Example:

```
// Original

// Compute the next value
int next(int x) {
    return x + 1;
}
```

```
// After removing comments

int next(int x) {
    return x + 1;
}
```

Listing 3.4: Example of the effect of the remove comments transformation.

- **Replace for with while**
  Purpose: Transform `for`-loops into semantically equivalent `while`-loops.

Example:

```
// Original
for (int i = 0; i < 10; i++) {
    sum += i;
}
```

```
// After replacing for with while
int i = 0;
while (i < 10) {
    sum += i;
    i++;
}
```

Listing 3.5: Example of the effect of the transformation that replaces for with while structures.

- **Replace Function Names**
  Purpose: Rename functions to a generic placeholder, e.g., `FUNC0`, `FUNC1`, etc.
  Example:

```
// Original
int compute(int x) {
    return x * 2;
}
```

```
// After replacing function names
int FUNC0(int x) {
    return x * 2;
}
```

Listing 3.6: Example of the effect of the transformation that replaces function names.

- **Replace Literals with Equivalents**
  Purpose: Substitute constants with logically equivalent expressions, such as replacing `true` with `1 == 1` or `0` with `false`.
  Example:

```
// Original
if (true) {
    do_something();
}
```

```
// After replacing literals with equivalents
if (1 == 1) {
    do_something();
}
```

Listing 3.7: Example of the effect of the transformation that replaces literals with equivalent values.

- **Save/Remove Strings**
  Purpose: Analogous to comments, but for string literals: either archive them during transformation's execution or replace them with a generic token.
  Example:

```
// Original
printf("Hello, world!\n");
```

```
// After replacing string literals
printf("S");
```

Listing 3.8: Example of the effect of the transformation that replaces strings.

- **Replace Variable Names**
  Purpose: Rename variables to a generic placeholder, e.g., `VAR0`, `VAR1`, etc.

Example:

```
// Original
int count = 0;
count += 5;
```

```
// After replacing variable names
int VAR0 = 0;
VAR0 += 5;
```

Listing 3.9: Example of the effect of the transformation that replaces variable names.

- **Swap Operators**
  Purpose: Flip binary comparison operators while preserving logical equivalence, e.g., `a > b` becomes `b < a`.
  Example:

```
// Original
if (x > y) {
    max = x;
}
```

```
// After swapping operators
if (y < x) {
    max = x;
}
```

Listing 3.10: Example of the effect of the transformation that swap operators.

In the context of this work, these transformations are applied whenever they are applicable: *Dead Code Injection*, *Remove Comments*, *Replace `for` with `while`*, *Replace Function Names*, *Replace Variable Names*, *Replace Literals with Equivalents*, *Save Strings*, and *Swap Operators*.

To illustrate the combined effect of these transformations, consider the code snippet on the left in Figure 3.6. It contains comments, string literals, a `for` loop, and comparison operators, all targets for adversarial manipulation. After applying the configured transformations with the variable renaming configured to generalize all variable names (by default it only generalizes 50%), the adversarial sample on the right is produced. Key changes include:

- Injection of a dead conditional that never executes, e.g. `if (1 == 2) {...}`.
- Conversion of the `for` loop into a semantically equivalent `while` loop.
- Removal of all comments and renaming of functions and variables.
- Swapping of comparison operators (e.g., > becomes < with operands reversed).

These transformations introduce lexical and structural variations without affecting program semantics, thereby generating challenging inputs that probe model consistency.

3.5 Performance Analysis

While flexibility, demonstrated in the earlier example of SCoPE2's extensibility, was a key design goal, performance was also a critical consideration. To assess performance, a direct comparison was conducted between transformations implemented in both SCoPE and SCoPE2. This evaluation focused on two key metrics: execution time and memory usage, which are especially relevant for large codebases or resource-constrained environments [111].

For the comparison, two similar scripts were created, one that uses SCoPE and other that uses SCoPE2. Both scripts aim to process a 1,000 code snippets of the RDiverseVul dataset

```c
// Counts how many uppercase letters are in a string
int count_uppercase(const char *text) {
    int count = 0;

    // Loop through each character in the string
    for (int i = 0; i < strlen(text); i++) {
        // Check if the character is an uppercase letter
        if (text[i] >= 'A' && text[i] <= 'Z') {
            count++;  // Increment the counter
        }
    }

    return count;  // Return the number of uppercase letters found
}
```

```c
int FUNC0(const char *VAR0) {
    int VAR1 = 0;
    int VAR2 = 0;
    if(1==2){
        int VAR3 = 0;
        printf("%d", VAR3);
    }
    while(strlen(VAR0) > VAR2) {
        if ('A' <= VAR0[VAR2] && 'Z' >= VAR0[VAR2]) {
            VAR1++;
        }
        VAR2++;
    }
    return VAR1;
}
```

Figure 3.6: On the left, the original code snippet used as input for adversarial generation; on the right, the corresponding adversarial sample generated by SCoPE2.

[112], using the *pandarallel* library to parallelize processing, allowing the usage of four CPU cores to speedup the process.

During each run, three transformations were applied to the source code: function and variable name generalization, which replaced specific identifiers with generic ones, and comment removal, eliminating all textual annotations. Both SCoPE versions were configured to return the transformed code as output, and SCoPE's ANTLR4 error-recovery feature was enabled to match the functionality provided by SCoPE2 through TreeSitter. By keeping the number of inputs, applied transformations, and number of cores identical for both systems, the experiment ensured equivalent testing conditions and allowed for a fair assessment of performance differences.

Table 3.1: Performance Comparison of SCoPE and SCoPE2 [111].

Version	**Execution Time**	**Peak Memory**
SCoPE	3 m 42 s	84.9 MB
SCoPE2	6 s	79.2 MB

The original SCoPE required 3 minutes 42 seconds to complete the workload, whereas SCoPE2 finished in just 6 seconds, only 2.7 % of the original runtime, highlighting a substantial optimization in execution time (Table 3.1).

Memory usage exhibited a more modest improvement. Figure 3.7 shows the SCoPE2 memory usage over time, and it is possible to verify that it peaked at 79.2 MB, stabilizing around 77.6 MB for most of the run. Figure 3.8 shows the results for SCoPE, that reached 84.9 MB with sustained usage near 80.4 MB. A standalone Python session that loaded the same 1,000-entry subset consumed 71.6 MB of RAM, indicating that data ingestion accounts for the majority of the memory footprint [111].

Overall, SCoPE2 achieves a sigificant execution time speedup with a slight reduction in peak memory usage, validating that its architectural improvements yield better execution performance.

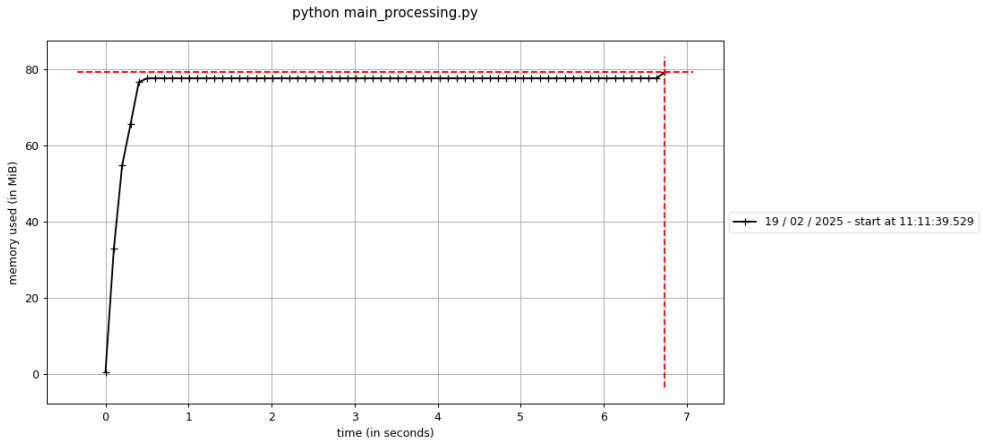

Figure 3.7: Memory usage over time during SCoPE2 execution.

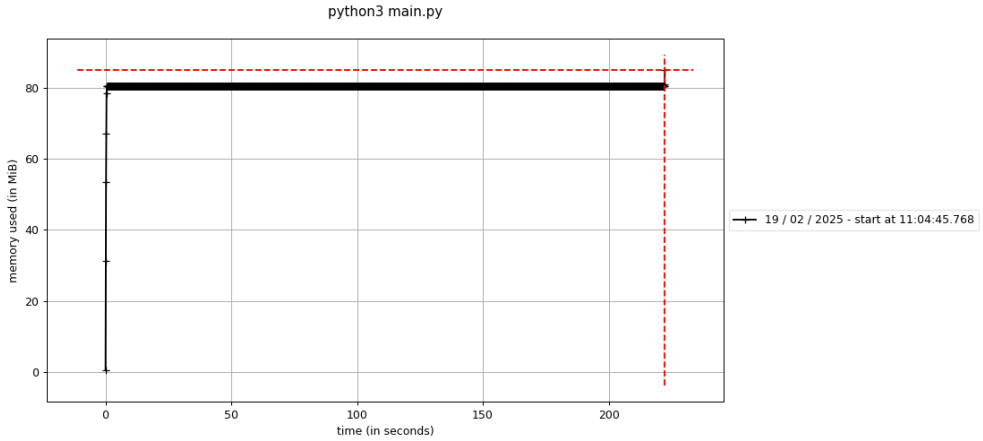

Figure 3.8: Memory usage over time during original SCoPE execution.

3.6 Extending SCoPE2

While the default transformations provided by SCoPE2 already enable a range of useful code modifications, it is essential to ensure that the tool remains easily extensible to support custom transformation needs. To illustrate the SCoPE2's extensibility, this section presents a simple example: a transformation that identifies all `for` loops in a code sample and inserts a simple comment immediately before each one. Since the goal is to demonstrate the transformation process rather than to generate meaningful comments, the inserted comment is simply: "This is a dummy transformation."

To add a new transformation in SCoPE2, the user must complete three main tasks: (1) define a query that locates the target syntax in source code, (2) implement the transformation class that applies the desired change, and (3) register the new transformation so that it will be invoked during processing. In this example, the goal is to insert a simple comment ("This is a dummy transformation.") immediately before every `for` loop.

First, because this transformation needs to locate loop constructs in the code, the implementation will leverage TreeSitter. In SCoPE2, any transformation that requires TreeSitter-based pattern matching is classified as a *QueryTransformation*. The very first step is therefore to add a new query to the configuration file, associating that query with a unique query ID. These queries are made using the TreeSitter query language, and are used by TreeSitter to extract code snippets of interest. The configuration file shown in Figure 3.9 maps query IDs, used internally by SCoPE2, to their corresponding TreeSitter query strings. In this case, a query named `add_comment_to_for_declaration` that matches every `for` statement in C/C++ code is added. At runtime, SCoPE2 will pass the entire AST through TreeSitter, execute this query, and return a list of syntax nodes corresponding to each `for` loop.

```yaml
add_comment_to_for_declaration:
  query: |
    [
    (for_statement )
    (enhanced_for_statement)
    ] @for_structure

  QUERY_ID: for_structure


remove_comments:
    query: |
      [
       (line_comment)
       (block_comment)
      ] @comment

    QUERY_ID: comment
```

Figure 3.9: Example of the configuration file for this scenario.

After defining the query, the next task is to implement the transformation class. All *QueryTransformations* share a common template: they must declare a name that precisely matches the top-level element of the corresponding transformation block in the YAML configuration file, and they must implement a "run" method which accepts the parsed source code and returns a modified version. In this example, the transformation class is named `AddCommentToForDeclaration`. Inside its `run` method, the code retrieves the list of loop nodes by invoking the TreeSitter query associated with yaml block named `"add_comment_to_for_declaration"`. For each node, the transformation inserts the string "/* This is a dummy transformation. */" immediately before the loop's starting token. The full implementation is shown in Figure 3.10, where the class constructor simply stores its name and any configuration parameters, and the `run` method performs (1) query execution, (2) node identification, and (3) comment insertion.

With the query and transformation class in place, the final step is to register the new transformation in the list that SCoPE2 will execute when processing source code. When creating a SCoPE2 instance, the user provides a list of transformation classes (by name) and any additional settings they require. Because SCoPE2 always instantiates transformations at runtime, the user need only add `AddCommentToForDeclaration` to their transformation list when instanciating the SCoPE2 class. From that point on, whenever `SCoPE2.process(...)` is called, the `PreProcessing` service will recognize the new transformation, supply it with the repository context and configuration dictionary, and then run it in sequence with any other enabled transformations. Figure 3.11 shows an example of a code snippet after the "dummy comment" has been inserted before each `for` loop.

```python
class AddCommentJavaForDeclaration(QueryTransformation):

    TRANSFORMATION_NAME: str = "add_comment_to_for_declaration"
    _PHASE = Phase.PROCESSING
    _DEFAULT_CONFIG = {}

    def __init__(self, repo_context: RepositoryContext, config: dict, **kwargs):
        self._DEFAULT_CONFIG.update(config)
        super().__init__(repo_context, self.TRANSFORMATION_NAME, self._PHASE, self._DEFAULT_CONFIG)
        self._repo = repo_context.tree_sitter_repo

    def run(self, entry: ProcessingEntry) -> ProcessingEntry:

        for_structures = self.detect(entry)
        for struct in for_structures:
            entry.code = entry.code.replace(struct, "//This is a dummy transformation\n"+struct)
        return entry

    def query(self) -> str:
        return self._DEFAULT_CONFIG['query']

    def detect(self, entry: ProcessingEntry) -> list:
        return entry.detection_result[self._DEFAULT_CONFIG['QUERY_ID']]
```

Figure 3.10: Example of the transformation implementation.

By following these three steps, adding a TreeSitter query, implementing a *QueryTransformation* class, and registering the class in the transformation pipeline, a user can quickly extend SCoPE2 with new functionality tailored to their specific needs. Because each transformation must adhere to a standard, integrating additional transformations remains straightforward and consistent. For more detailed implementation guidance or additional clarifications, please refer to the SCoPE2 repository [1].

3.7 Chapter Remarks

This chapter has introduced the SCoPE2 framework, which is designed for extensibility and flexibility. This new SCoPE version introduced multiple improvements, both in extensibility and performance, with the new version only needing 2.7% of SCoPE's execution time to process the same number of samples.

Beyond the applications of SCoPE2 explored so far, such as removing programmer-defined identifiers to improve model generalisation, its ability to generate semantically equivalent code with lexical and syntactic variation has potential for adversarial example generation. Subsequent chapters explore the use of SCoPE2 to improve and evaluate code models robustness.

[1] https://github.com/jp2425/scope2

```java
public class LoopExample {

    public static void main(String[] args) {
        int sum = 0;
        //This is a dummy transformation
for (int i = 1; i <= 5; i++) {
            sum += i;
        }
        System.out.println("Sum: " + sum);

        String[] words = {"apple", "banana", "cherry"};
        //This is a dummy transformation
for (String word : words) {
            System.out.println(word);
        }

        //This is a dummy transformation
for (int i = 10; i >= 1; i -= 2) {
            System.out.println("Countdown: " + i);
        }
    }
}
```

Figure 3.11: Java code example with the new transformation applied.

Chapter 4

Improving Code Model's Robustness

In order to address the lack of effective defenses for code-analysis LLMs, this chapter presents a novel architecture that combines adversarial fine-tuning with a SCoPE2–powered normalization layer (the N&P strategy). The first stage uses SCoPE2 to generate adversarial code samples, which are then employed to fine-tune a base model to improve its robustness against adversarial attacks. Building on this, the second stage wraps the fine-tuned model with a preprocessing layer that replaces all function and variable identifiers with generic placeholders, neutralizing the most prevalent adversarial technique: renaming programmer-defined identifiers. The remainder of this chapter describes each element of the proposed architecture, details the datasets and preprocessing pipeline, specifies the base model and training hyperparameters, and defines the evaluation metrics used to measure robustness gains.

4.1 Proposed Architecture

The systematic review in Chapter 2 highlighted two primary defenses against adversarial attacks on code models: adversarial training and fine-tuning. While full adversarial training can significantly improve robustness, it is computationally prohibitive for LLMs due to the cost of retraining. Adversarial fine-tuning, by contrast, has proven to be a practical alternative [75, 87], though it does not guarantee complete immunity: some adversarial inputs can still influence model predictions [21].

To further bolster model resilience, the proposed architecture integrates adversarial fine-tuning with the **N&P strategy**. Although prior work has applied N&P alongside adversarial training [88], its combination with fine-tuning remains unexplored.

Therefore, the architecture consists of two stages:

1. **Adversarial Fine-Tuning:** The process begins by evaluating the base model on a clean test set to establish a performance baseline. Next, SCoPE2 is used to generate adversarial code samples, which are then split into training, validation, and test subsets. The base model is evaluated on the adversarial test set to measure its vulnerability to these perturbations. Afterward, the model is further fine-tuned on the adversarial training set, as described in [21]. Finally, the fine-tuned model is re-evaluated on the adversarial test set to quantify any robustness improvements.
2. **Normalization Layer (N&P):** Wrap the fine-tuned model with a preprocessing layer that replaces all function and variable identifiers in the input code with generic names. This layer neutralizes the most common adversarial strategy, renaming identifiers,

generalizing all identifiers before inference. The complete solution is depicted in Figure 4.1, where the code is first handled by the preprocessing layer before being passed to the model.

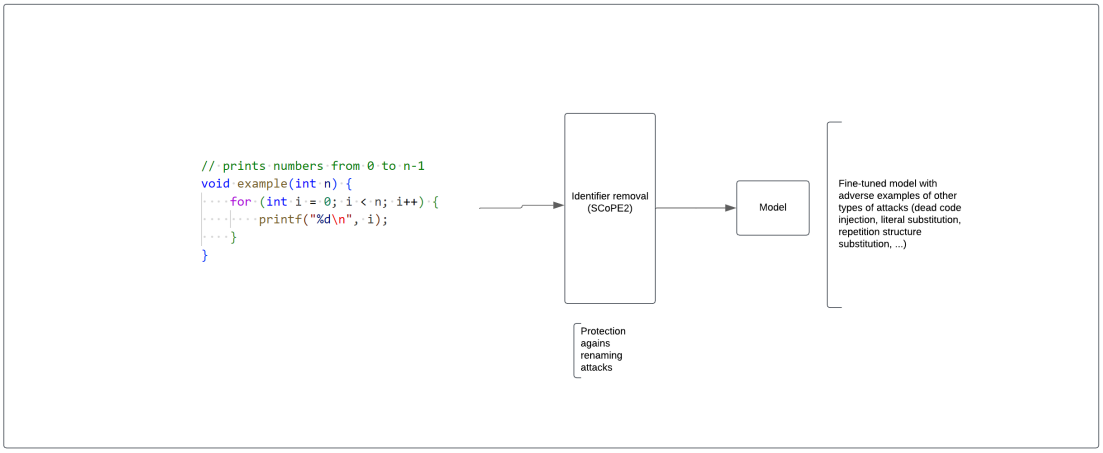

Figure 4.1: Final architecture with SCoPE2 normalization layer (Stage 2).

Subsequent sections evaluate the effectiveness of each stage in improving model robustness against adversarial perturbations.

4.2 Datasets and Data Preprocessing

Before diving into the architecture and experimental results, it is essential to introduce the dataset used in this study. Table 4.1 lists datasets commonly employed for vulnerability detection research. From this overview, it can be observed that most datasets operate at the function level, and many, including widely used collections such as Juliet and Draper, rely on synthetic code. Datasets containing synthetic samples were excluded from this work, as such data often fail to capture the complexity of real-world code. Although models trained on synthetic datasets may achieve high scores in controlled experiments, they typically generalize poorly when applied to real-world code [113].

Name	Uses Synthetic Data	Size	Language	Granularity
Juliet [114]	Yes	64 099	C	Function level
Devign [115]	No	48 687	C	Function level
VulDeePecker [116]	Yes	61 638	C/C++	Slice level
muVulDeePecker [117]	Yes	181 641	C/C++	Slice level
Big-vul [118]	No	264 919	C/C++	Function level
Draper [119]	Yes	1.27 M	C/C++	Function level
REVEAL [120]	No	18 169	C/C++	Function level
DiverseVul [104]	No	349 437	C/C++	Function level
FormAI [121]	Yes	112 000	C	Function level

Table 4.1: Overview of selected vulnerability detection datasets.

The RDiverseVul dataset [112] served as the foundation for all experiments. This collection is a refined version of DiverseVul, a dataset designed to be the largest non-synthetic dataset

available for software vulnerability detection. In RDiverseVul, erroneous entries, such as functions containing only comments or nearly identical functions labeled inconsistently, were removed through a SCoPE2-powered cleaning script. Table 4.2 summarizes the RDiverseVul features: the *func* column holds the C/C++ function source, while *target* indicates whether the function is vulnerable. The remaining fields (e.g., *cwe*, *project*, *commit_id*, *size*, and *message*) provide metadata but were not used for model training.

Table 4.2: RDiverseVul Dataset Features [122]

Feature	Description
func	Contains the C/C++ function code.
target	Does the function is vulnerable or not.
cwe	List of respective CWEs present in the function.
project	Project from where the function was extracted.
commit_id	Identifier of the commit.
size	Size of the function.
message	Commit message.

To generate adversarial examples, the raw function code was first stripped of all comments using SCoPE2's *Remove Comments* transformation, ensuring that no samples consisted solely of comments after tokenization. Then, the following transformations were applied: *Replace Literals with Equivalent Values*, *Swap Operators*, *Replace* `for` *with* `while`, *Dead Code Injection*, and *Variable and Function Name Replacement*. All transformations were applied using their default settings, with one exception: the *Variable Name Replacement* transformation was configured to replace only 50% of the variable names with generic identifiers. This partial replacement was chosen to introduce variability in commonly used names while still retaining some original identifiers in the adversarial samples.

After generating the adversarial samples, and to ensure a fair comparison of results, a balanced subset of 10,000 entries from RDiverseVul was extracted and used in all experiments. This subset was chosen due to computational resource limitations, which made fine-tuning a LLM on the full dataset infeasible within a reasonable timeframe. The holdout method was applied to this subset, resulting in a 70/20/10 split for training, validation, and testing, respectively. These splits were saved to disk and consistently reused across all experiments.

After the split, an additional version of each subset was created, in which all variable names were fully generalized. These generalized versions will be used to evaluate the second stage of the solution, represented in Figure 4.1, that includes a normalization layer powered by SCoPE2.

4.3 Base Model

The experiments in this work build upon **LLaMA 3.2** [123], a transformer-based language model with 1.23 billion parameters that has demonstrated strong performance when fine-tuned for binary classification tasks involving C/C++ code [122]. Rather than training from scratch, a checkpoint of LLaMA 3.2 already fine-tuned to distinguish between *vulnerable* and *not vulnerable* functions was adopted, providing a robust starting point with macro-averaged F1 scores of 0.66 on real-world held-out data [122].

In this thesis, all adversarial fine-tuning iterations were conducted using the same hyperparameters established in [122]. These hyperparameters are detailed in Table 4.3.

Table 4.3: Model Hyperparameters [122]

Hyperparameter	**Value**
Epochs	4
Batch Size	32
Learning Rate	2×10^{-5}
Weight Decay	0.01
Evaluation Interval	Every 100 steps

Due to computational constraints, fine-tuning was performed using the Low-Rank Adaptation (LoRA) technique, a Parameter-Efficient Fine-Tuning (PEFT) method that significantly reduces the number of trainable parameters [124]. This approach enables faster training while maintaining model performance.

4.4 Evaluation Metrics

For model evaluation, several metrics are employed to compare the performance of AI models. These standard metrics reflect specific strengths and weaknesses of the models, providing insights into their overall effectiveness. The foundation for model evaluation is the confusion matrix (Figure 4.2), which summarizes the comparison between predicted and true labels by reporting the numbers of True Positives (TP), True Negatives (TN), False Positives (FP) and False Negatives (FN) for each class.

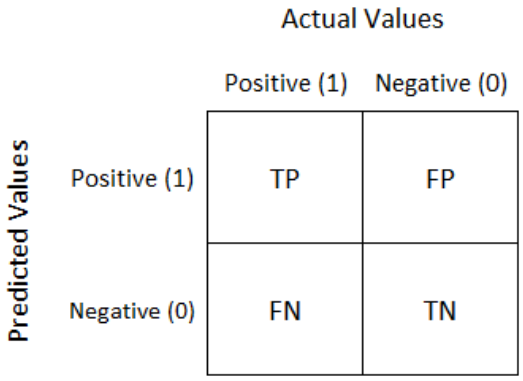

Figure 4.2: Confusion matrix for binary classification [125]

One of the most commonly used metrics is **Accuracy**, which measures the proportion of correctly classified samples across all classes. Accuracy is easy to interpret and implement, but it can be misleading when classes are imbalanced: a model may achieve high accuracy by simply predicting the majority class on every example. It is defined as:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \tag{4.1}$$

To treat each class equally, the *macro-averaged* versions of precision, recall, and F1-Score will be used. This metrics version is also given by default by the *Scikit-learn* library, used for evaluation in this work. Given K classes, let Precision_i, Recall_i, and F1_i be the metrics computed on class i.

Precision measures how many of the positive predictions are actually correct. Macro-averaging ensures that small classes contribute equally to the overall score, preventing dominant classes from masking poor performance on rarer ones. It can be defined as:

$$\text{Precision}_{\text{macro}} = \frac{1}{K}\sum_{i=1}^{K}\frac{\text{TP}_i}{\text{TP}_i + \text{FP}_i} \tag{4.2}$$

The **recall** (or sensitivity) measures how many of the actual positive instances the model correctly identifies. It can be defined as:

$$\text{Recall}_{\text{macro}} = \frac{1}{K}\sum_{i=1}^{K}\frac{\text{TP}_i}{\text{TP}_i + \text{FN}_i} \tag{4.3}$$

The **F1-Score** is the harmonic mean of precision and recall, combining both into a single metric. It can be defined as:

$$\text{F1}_{\text{macro}} = \frac{1}{K}\sum_{i=1}^{K}\left(2 \times \frac{\text{Precision}_i \times \text{Recall}_i}{\text{Precision}_i + \text{Recall}_i}\right) \tag{4.4}$$

In addition to these metrics, the ASR was also adopted, not to assess the model's performance, but rather to evaluate the effectiveness of the generated adversarial examples. The ASR is defined as:

$$\text{ASR} = \frac{N_{\text{success}}}{N_{\text{total}}} \tag{4.5}$$

where N_{success} represents the number of adversarial examples that successfully mislead the model, and N_{total} is the total number of adversarial examples generated or attempted.

4.5 Chapter Remarks

This chapter has laid the groundwork for the robustness experiments that will be done in the next section by detailing the rationale behind each design choice and explaining how they connect to subsequent analyses. The decision to base all experiments on RDiverseVul dataset, rather than any synthetic dataset, ensures that adversarial samples reflect real-world code diversity, while the use of a pre-fine-tuned LLaMA 3.2 checkpoint provides a baseline from which improvements can be measured. Applying LoRA for adversarial fine-tuning allows efficient adaptation of the 1.23 billion-parameter model within available computational resources, and the specific SCoPE2 transformations described here reflect a balance between preserving semantic value and introducing syntactic variation. In the next chapter, these components will be brought together, evaluating the adversarially fine-tuned model on clean and adversarial test sets, and then assessing the impact of SCoPE2's normalization layer on model's performance.

Chapter 5

Results and Discussion

This chapter presents and analyzes the experimental results. First, the base model is evaluated to establish a performance baseline. Then, the impact of adversarial fine-tuning is assessed, followed by evaluation of the fine-tuned model protected with an additional normalization layer. Finally, the relationship between adversarial-original sample similarity and defense effectiveness is examined.

5.1 LLaMA 3.2 Evaluation

After preprocessing, the base model was first evaluated on a clean test subset in which all variable and function names had been generalized by SCoPE2. This identifier abstraction aligns with the model's training regime, having been fine-tuned for identifier-agnostic vulnerability detection, and leverages prior findings that removing programmer-defined names improves performance by reducing distractions from superficial naming patterns [122]. Establishing this baseline on clean, generalized inputs ensured that the model's core classification ability could be measured under ideal conditions. Only afterward was the same model tested on adversarial examples generated by SCoPE2, which introduced semantics-preserving edits such as renaming, dead code injection, operator swaps, and loop restructurings. By comparing results on these two datasets, first clean and generalized, then adversarial, the evaluation framework clearly isolates how much performance is lost to adversarial perturbations and sets the stage for improvements through fine-tuning and normalization. It is relevant to highlight that this model was chosen because it already outperformed the best model created by the DiverseVul authors [104] by 19 percentage points in F1-Score.

Table 5.1: Performance of the base model on clean versus adversarial samples.

Model / Dataset	**Accuracy**	**Precision**	**Recall**	**F1-Score**
NatGen on clean samples [104]	92%	52%	43%	47%
Base model on clean samples [122]	66%	65%	67%	66%
Base model on adversarial samples	60%	62%	60%	59%

Evaluation on adversarial samples reveals that accuracy decreases from 66% to 60%, precision from 65% to 62%, recall from 67% to 60%, and F1-Score from 66% to 59%. The 7-point drop in F1-Score, along with consistent declines across all metrics, indicates significant vulnerability to adversarial perturbations. Such results demonstrate that strong performance on clean data does not ensure robustness under adversarial conditions, underscoring the importance of defense mechanisms.

5.2 Adversarial Fine-Tuning

The first defense mechanism, and the core of the proposed solution, is adversarial fine-tuning of the base model. The main idea is to start from a model already fine-tuned for good performance on clean samples, and then further fine-tune it to improve robustness against adversarial attacks [21]. For this adversarial stage, the hyperparameters originally applied in the base-model fine-tuning for Software Vulnerability Detection (SVD) detection are reused [122], and LoRA is again employed to reduce computational cost and memory footprint.

The pre-trained LLaMA model, originally designed for text generation, is repurposed for binary classification by adding a lightweight linear layer on top of its transformer backbone. When given an input sequence, the model generates a 2048-dimensional pooled representation, which is then passed through this final linear layer to produce two logits (one for "vulnerable", one for "not vulnerable"). Rather than fine-tuning all 1.23 billion parameters, only a small subset of parameters is trained, thanks to the usage of LoRA. In addition, the pre-trained weights are stored in 4-bit quantized form to minimize memory usage. As a result, both GPU memory requirements and overall computational cost are drastically reduced, making adversarial fine-tuning feasible even on modest hardware without compromising the final classifier's accuracy.

Table 5.2 summarizes the performance of of the fine-tuned model, compared with the adversarial and clean evaluation of the base model.

Table 5.2: Results with the adversarial fine-tuned model and the base model experiments.

Model / Dataset	Accuracy	Precision	Recall	F1-Score
NatGen on clean samples [104]	92%	52%	43%	47%
Base model on clean samples [122]	66%	65%	67%	66%
Base model on adversarial samples	60%	62%	60%	59%
Fine-tuned model on adversarial samples	65%	65%	65%	65%

These results indicate that adversarial fine-tuning with LoRA effectively mitigates the robustness gap, achieving performance on adversarial samples comparable to that of the base model on clean inputs. The improvements in both precision and recall under adversarial conditions suggest that the model maintains its ability to produce reliable predictions for both classes. This outcome supports the use of adversarial fine-tuning as a viable and effective defense mechanism for binary classification in the context of the SVD task, with the 1% F1-Score gap being in line with results found on literature [21].

However, it is important to highlight that, although this approach significantly improves model performance under adversarial inputs, as noted in the literature review, more advanced attack methods exist that specifically exploit programmer-defined identifiers to manipulate model outputs. The literature does not present a fully effective solution to this problem, as all proposed defenses, including adversarial fine-tuning, can be bypassed by sophisticated attackers using advanced identifier manipulation strategies. In this thesis, a complementary solution to adversarial fine-tuning is proposed, involving the introduction of a normalization layer designed to remove all programmer-defined identifiers from the input samples before they are processed by the model.

5.3 Normalization Layer

The normalization layer serves as a preprocessing step applied to every code snippet at inference time, with the specific goal of neutralizing identifier-renaming attacks. In practice, this SCoPE2-powered layer systematically replaces every variable and function name with a generic placeholder before passing the code to the classifier. By doing so, the model no longer sees programmer-defined names, only standardized tokens, making it impossible for an attacker to influence predictions simply by choosing misleading or obscure identifiers. Without access to the original names, any attempt to trick the classifier through renaming is effectively foiled, since the model's decision must rely on the underlying structure and logic rather than on superficial naming patterns.

However, directly applying this approach at inference without training the model on similarly normalized code can lead to significant performance degradation [21]. To mitigate this, the model was fine-tuned using code with all identifiers already normalized. The fine-tuning process occurred in three stages: (1) the LLaMA 3.2 model was originally trained on code with standard identifiers by Meta, (2) subsequently, it was fine-tuned on fully normalized code for the SVD task as described in [122], and (3) finally, it underwent adversarial fine-tuning using samples with mixed code representations, reflecting real-world attacks.

Table 5.3: Overall performance metrics across different model configurations.

Model / Dataset	Accuracy	Precision	Recall	F1-Score
NatGen on clean samples [104]	92%	52%	43%	47%
Base model on clean samples [122]	66%	65%	67%	66%
Base model on adversarial samples	60%	62%	60%	59%
Fine-tuned model on adversarial samples	65%	65%	65%	65%
Fine-tuned + SCoPE2 normalization layer	64%	66%	64%	63%

To assess the effectiveness of the final approach, the fine-tuned model, augmented with the SCoPE2 normalization layer, was evaluated on a subset of code samples with all identifiers removed, simulating deployment in a real-world adversarial setting. As shown in Table 5.3, the addition of the normalization layer results in a slight decrease in performance compared to the adversarially fine-tuned model alone, with a 2% reduction in F1-score. Nevertheless, the combined approach still outperforms the base model when test against adversarial examples, confirming its effectiveness as a defensive mechanism. It is worth noting that, although this setup yields a lower F1-score, potentially due to increased sensitivity to other types of code transformations, it remains immune to identifier manipulation, which is the most prevalent adversarial technique used to undermine code models.

5.4 Sample's Similarity Impact on Adversarial Attack Effectiveness

Even after applying multiple defense mechanisms, the model's performance on adversarial samples does not match its accuracy on clean inputs, confirming that no defense is entirely foolproof. A natural next question is whether more extensive code transformations produce stronger adversarial attacks. Understanding this relationship clarifies the degree of dissimilarity that adversarial examples must exhibit, and the effort required, to successfully mislead the model.

To explore this, each adversarial sample was paired with its original, "clean" counterpart, and their textual similarity was computed using Python's `difflib` library [126]. Early attempts employed the MinHash algorithm, that can be used for estimating Jaccard similarity in large ammounts of text, but it failed to detect minor but semantically significant code modifications. In contrast, `difflib` directly compares sequences of characters and is sensitive to these small edits, making it more suitable for comparison.

As the primary measure of attack effectiveness, the ASR reflects the fraction of adversarial examples that successfully change the model's prediction, from "not vulnerable" to "vulnerable," or vice versa, regardless of overall accuracy. This focus on ASR isolates the effectiveness of the transformed samples themselves, rather than on overall model performance.

Figure 5.1 presents ASR values for the fine-tuned model, grouped by intervals of similarity between each adversarial code snippet and its original. The lowest-similarity group, those adversarial samples that underwent the greatest number of SCoPE2 transformations, achieves an ASR of up to 60%. In other words, when code is extensively modified (thereby reducing its textual overlap with the original), the model is fooled most often. Outside of these heavily altered cases, ASR remains relatively stable across moderate similarity ranges.

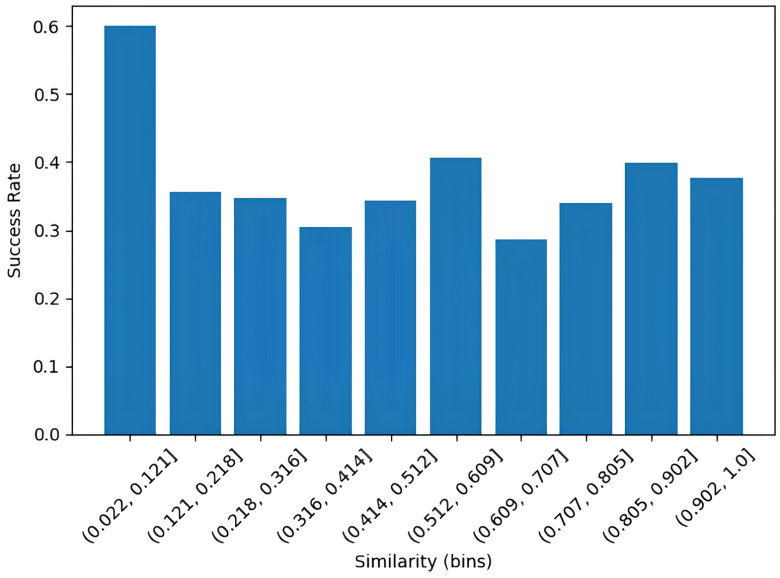

Figure 5.1: ASR for the adversarial fine-tuned model, grouped by similarity interval.

The same set of experiments was performed on the non-fine-tuned model and the results are presented in Figure 5.2. In this case, model vulnerability to adversarial perturbations is more pronounced: samples with less than 30% similarity frequently yield ASR values exceeding 70%.

These findings suggest that lower similarity, corresponding to more extensive transformations, correlates with a higher probability of altering the model's prediction. Although this outcome contrasts with some prior work [75], the discrepancy may stem from the use of the more recent LLaMA 3.2 model, the application of less aggressive identifier-renaming transformations during attack, or more effective defense strategies incorporated into the fine-tuned model.

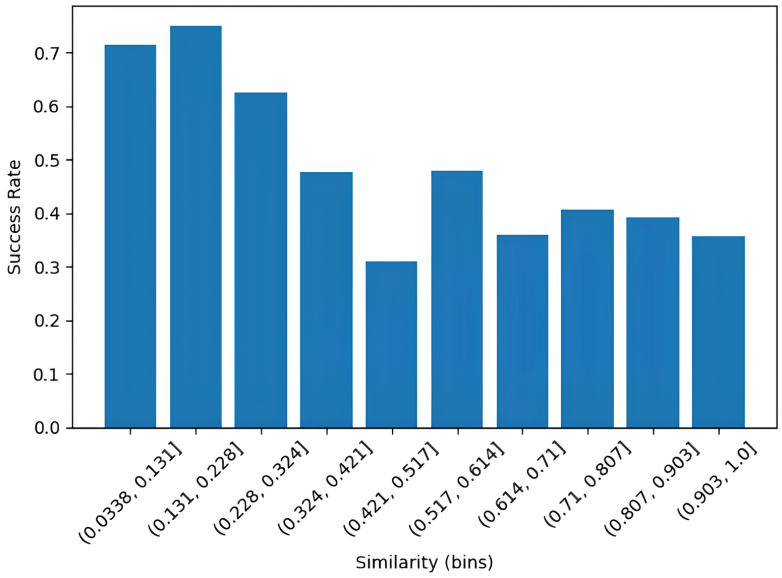

Figure 5.2: ASR for the non-fine-tuned model, grouped by similarity interval.

5.5 Chapter Remarks

This chapter has presented a comprehensive evaluation of defense strategies against adversarial attacks in the context of SVD. Initial assessment of the base model revealed a marked degradation in all performance metrics when subjected to SCoPE2-generated adversarial samples, highlighting the model's inherent vulnerability. Adversarial fine-tuning with LoRA successfully closed the robustness gap, restoring performance on adversarial inputs to levels nearly equivalent to those on clean code, and demonstrating its value as a cost-effective defense.

The introduction of a SCoPE2-based normalization layer further strengthened resilience by eliminating all programmer-defined identifiers at inference time. Although this additional layer incurred a modest reduction in F1-Score relative to the adversarially fine-tuned model alone, it immunized the classifier against the most pervasive manipulation tactic. Thus, the combined pipeline offers a balanced trade-off between raw performance and robustness against adversarial attacks.

Finally, analysis of sample similarity revealed a clear inverse relationship between code similarity and ASR for both fine-tuned and non-fine-tuned models. Heavily transformed adversarial examples (low similarity) produced substantially higher ASR values, underscoring the potency of extensive identifier manipulations. These observations reinforce the necessity of multimodal defenses and motivate further research into adaptive normalization techniques and more sophisticated attack generation methods.

Overall, this chapter confirms that layered defenses that integrate both model-centric and input-centric strategies prove essential for maintaining robustness in code analysis systems under adversarial conditions.

Chapter 6

Conclusions and Future Work

This final chapter summarizes the principal conclusions of the thesis, evaluates the extent to which the original objectives have been achieved, discusses the limitations encountered, and outlines promising directions for future research.

6.1 Summary of Achievements

All objectives defined at the outset of this work have been fully accomplished. Specifically:

- **Objective 1:** A systematic literature review identified existing techniques for attacking code models and improving their robustness. It was determined that adversarial attacks represent the most commonly studied threat in the literature, and that adversarial fine-tuning is among the most effective defenses reported to date.
- **Objective 2:** Several methods for generating adversarial source-code samples were identified. Unlike adversarial perturbations in natural language, transformations applied to code must preserve strict syntactic and semantic correctness. As a result, the literature tends to favor techniques that maintain the original program behavior. Common approaches include variable renaming, dead code insertion, and transforming control structures, such as converting `for` loops into `while` loops. The techniques identified under this objective were essential for achieving the subsequent objectives successfully.
- **Objective 3:** Insights from the review were used in the design and implementation of SCoPE2, a modular and extensible framework for adversarial sample generation on code. SCoPE2 includes a suite of transformation modules and supports both attack and defense workflows, and it is publicly available to the research community. This new version significantly improves upon SCoPE, offering substantial enhancements in execution time and memory efficiency during processing.
- **Objective 4:** SCoPE2 was applied to generate adversarial examples for fine-tuning a state-of-the-art code model. Two defense strategies were evaluated: (1) adversarial fine-tuning, and (2) the N&P technique. The experiments demonstrated that adversarial fine-tuning significantly improves robustness, restoring performance on perturbed inputs nearly to the level observed on clean code. The N&P technique effectively neutralizes identifier-renaming attacks but can degrade overall task accuracy. It was also found that the degree of transformation of a clean sample can lead to more effective adversarial attack, despite loosing the stealth associated with minor perturbations.

Taken together, these results demonstrate that modern code models remain vulnerable to adversarial manipulations, and that, while defenses can substantially mitigate risk, no single technique yields complete immunity.

6.2 Limitations and Future Work

The experiments conducted in this thesis demonstrate that combining adversarial fine-tuning with a SCoPE2-powered normalization layer yields improvements in robustness. However, several constraints may underpin the current results, and addressing these can guide future research.

First, the adversarial examples generated by SCoPE2 rely on a fixed set of transformations applied uniformly across all samples. Although this approach exposes many common model weaknesses, it does not employ optimization or search heuristics to discover the most effective perturbations for a given model. As a result, some vulnerabilities, especially those that require highly targeted changes, may remain unexploited. Extending this work could involve integrating gradient-based or reinforcement-learning-driven search methods that guide the adversarial samples creation process. By pairing such optimization strategies with the existing SCoPE2 transformations, it would be possible to produce adversarial samples that more precisely reveal the model's blind spots.

Second, the evaluation in this thesis was carried out on a single open-source code model, the LLaMA 3.2 fine-tuned for SVD, and on a single dataset (RDiverseVul). While the choice of LLaMA 3.2 reflects a state-of-the-art code classification baseline, it is important to verify that the observed improvements generalize to other architectures. Future studies should apply the same techniques to alternative models and potentially to other datasets.

Finally, in the current setup, the normalization layer is applied at inference time, and the model has been fine-tuned on similar normalized inputs. While this ensures immunity to identifier-renaming attacks, it may inadvertently increase sensitivity to other transformations, such as unusual control-flow rewrites or other semantic-preserving refactorings that do not involve identifiers. A possible extension is a more in-depth assessment of the impact of each transformation on the model's robustness when the normalization layer is applied.

6.3 Final Remarks

This thesis makes several valuable contributions to the security of code models: a systematic review of adversarial techniques, the design and publication of the SCoPE2 framework (including its suite of transformations), and an empirical evaluation of defense strategies for large code models. The main takeaway is that, despite the significant promise of AI-driven code analysis for software engineering, these models remain vulnerable to targeted attacks and require mitigation before real-world deployment. No existing defense achieves complete robustness, highlighting the ongoing need for research into adversarial resilience in this critical area.

Bibliography

[1] Kana Inagaki and David Keohane. *Toyota and Volkswagen fall further behind in the software race*. Sept. 2024. url: https://www.ft.com/content/7e5677b8-87fa-41fa-8648-9009ac7f14fc.

[2] Miyoung Yoo. *[Editorial] The Evolving Era of Home Appliances Meets AI and Software - Samsung Global Newsroom*. Oct. 2024. url: https://news.samsung.com/global/editorial-the-evolving-era-of-home-appliances-meets-ai-and-software.

[3] David Niewolny. *Boom in AI-Enabled Medical Devices Transforms Healthcare*. Mar. 2024. url: https://blogs.nvidia.com/blog/ai-medical-devices-gtc-2024/.

[4] Jeff Inglis. *Electricity grids are at risk from cyberattack. Here's how we can keep them running*. Aug. 2018. url: https://www.weforum.org/stories/2018/08/keeping-the-electricity-grid-running-4-essential-reads/.

[5] Ann Schlemmer. *Finance And Fintech: The Role Of Open Source In The Future Of Banking*. Sept. 2023. url: https://www.forbes.com/councils/forbesbusinesscouncil/2023/09/28/finance-and-fintech-the-role-of-open-source-in-the-future-of-banking/.

[6] Vibhu Mishra. *Cyberattacks on healthcare: A global threat that can't be ignored*. Accessed: 2024-11-22. Nov. 2024. url: https://news.un.org/en/story/2024/11/1156751 (visited on 11/22/2024).

[7] Andy Greenberg. *Hackers Remotely Kill a Jeep on the Highway—With Me in It*. Accessed: 2024-11-22. July 2015. url: https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/ (visited on 11/22/2024).

[8] Roshan N. Rajapakse et al. "Challenges and solutions when adopting DevSecOps: A systematic review". In: *Information and Software Technology* 141 (2022), p. 106700. issn: 0950-5849. doi: https://doi.org/10.1016/j.infsof.2021.106700. url: https://www.sciencedirect.com/science/article/pii/S0950584921001543.

[9] Maurice Dawson et al. "Integrating Software Assurance into the Software Development Life Cycle (SDLC)". In: *Journal of Information Systems Technology and Planning* 3 (Jan. 2010), pp. 49–53.

[10] Yubin Qu et al. "A survey on robustness attacks for deep code models". In: *Automated Software Engineering* 31.2 (Aug. 9, 2024), p. 65. issn: 1573-7535. doi: 10.1007/s10515-024-00464-7. url: https://doi.org/10.1007/s10515-024-00464-7.

[11] Inbal Shani and GitHub Staff. *Survey Reveals AI's Impact on the Developer Experience*. Accessed: 2024-11-22. June 2023. url: https://github.blog/news-insights/research/survey-reveals-ais-impact-on-the-developer-experience/ (visited on 11/22/2024).

[12] Moataz Chouchen et al. "How Do Software Developers Use ChatGPT? An Exploratory Study on GitHub Pull Requests". In: MSR '24. Lisbon, Portugal: Association for Computing Machinery, 2024, pp. 212–216. isbn: 9798400705878. doi: 10.1145/3643991.3645084. url: https://doi.org/10.1145/3643991.3645084.

[13] Zhensu Sun et al. "CoProtector: Protect Open-Source Code against Unauthorized Training Usage with Data Poisoning". In: *Proceedings of the ACM Web Conference 2022*. Association for Computing Machinery, 2022, pp. 652–660. isbn: 9781450390965. doi: 10.1145/3485447.3512225. url: https://doi.org/10.1145/3485447.3512225.

[14] Raphaël Khoury et al. "How Secure is Code Generated by ChatGPT?" In: *2023 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. 2023, pp. 2445–2451. doi: https://doi.org/10.1109/SMC53992.2023.10394237.

[15] Nicolas Guzman Camacho. "Unlocking the Potential of AI/ML in DevSecOps: Effective Strategies and Optimal Practices". In: *Journal of Artificial Intelligence General science (JAIGS) ISSN:3006-4023* 2.1 (Mar. 2024), pp. 79–89. doi: https://doi.org/10.60087/jaigs.v2i1.p89. url: https://jaigs.org/index.php/JAIGS/article/view/26.

[16] Matija Cankar et al. "Security in DevSecOps: Applying Tools and Machine Learning to Verification and Monitoring Steps". In: *Companion of the 2023 ACM/SPEC International Conference on Performance Engineering*. ICPE '23 Companion. Coimbra, Portugal: Association for Computing Machinery, 2023, pp. 201–205. isbn: 9798400700729. doi: 10.1145/3578245.3584943. url: https://doi.org/10.1145/3578245.3584943.

[17] TaeGuen Kim et al. "A Multimodal Deep Learning Method for Android Malware Detection Using Various Features". In: *IEEE Transactions on Information Forensics and Security* 14.3 (2019), pp. 773–788. doi: https://doi.org/10.1109/TIFS.2018.2866319.

[18] R. Vinayakumar et al. "Robust Intelligent Malware Detection Using Deep Learning". In: *IEEE Access* 7 (2019), pp. 46717–46738. doi: https://doi.org/10.1109/ACCESS.2019.2906934.

[19] Zhenlong Yuan et al. "Droid-Sec: deep learning in android malware detection". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM '14. Chicago, Illinois, USA: Association for Computing Machinery, 2014, pp. 371–372. doi: 10.1145/2619239.2631434. url: https://doi.org/10.1145/2619239.2631434.

[20] Aditi Gupta and Rinkaj Goyal. "A Generative AI-Driven Method-Level Semantic Clone Detection Based on the Structural and Semantical Comparison of Methods". In: *IEEE Access* 12 (2024), pp. 70773–70791. doi: https://doi.org/10.1109/ACCESS.2024.3401770.

[21] Noam Yefet et al. "Adversarial examples for models of code". In: *Proc. ACM Program. Lang.* 4 (OOPSLA Nov. 2020). doi: 10.1145/3428230. url: https://doi.org/10.1145/3428230.

[22] Xueqi Yu et al. "AdVulCode: Generating Adversarial Vulnerable Code against Deep Learning-Based Vulnerability Detectors". In: *Electronics* 12 (4 Feb. 2023), p. 936. issn: 2079-9292. doi: 10.3390/electronics12040936.

[23] Weiye Yu et al. "PATVD:Vulnerability Detection Based on Pre-training Techniques and Adversarial Training". In: *2022 IEEE Smartworld, Ubiquitous Intelligence & Computing, Scalable Computing & Communications, Digital Twin, Privacy Computing, Metaverse, Autonomous & Trusted Vehicles (SmartWorld/UIC/ScalCom/DigitalTwin/PriComp/Meta)*. 2022, pp. 1774–1781. doi: 10.1109/SmartWorld-UIC-ATC-ScalCom-DigitalTwin-PriComp-Metaverse56740.2022.00253.

[24] Francesco Barchi et al. "Exploration of Convolutional Neural Network models for source code classification". In: *Engineering Applications of Artificial Intelligence* 97 (2021), p. 104075. issn: 0952-1976. doi: https://doi.org/10.1016/j.engappa

i.2020.104075. url: https://www.sciencedirect.com/science/article/pii/S0952197620303353.

[25] Sahil Suneja et al. "Probing model signal-awareness via prediction-preserving input minimization". In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, 2021, pp. 945–955. isbn: 9781450385626. doi: 10.1145/3468264.3468545. url: https://doi.org/10.1145/3468264.3468545.

[26] Yu Zhou et al. "Adversarial Robustness of Deep Code Comment Generation". In: *ACM Trans. Softw. Eng. Methodol.* 31 (4 July 2022). issn: 1049-331X. doi: 10.1145/3501256. url: https://doi.org/10.1145/3501256.

[27] Yao Wan et al. "You see what I want you to see: poisoning vulnerabilities in neural code search". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, 2022, pp. 1233–1245. isbn: 9781450394130. doi: 10.1145/3540250.3549153. url: https://doi.org/10.1145/3540250.3549153.

[28] Zhou Yang et al. "Natural attack for pre-trained models of code". In: *Proceedings of the 44th International Conference on Software Engineering*. Association for Computing Machinery, 2022, pp. 1482–1493. isbn: 9781450392211. doi: 10.1145/3510003.3510146. url: https://doi.org/10.1145/3510003.3510146.

[29] Ali Al-Kaswan et al. "Traces of Memorisation in Large Language Models for Code". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. Association for Computing Machinery, 2024. isbn: 9798400702174. doi: 10.1145/3597503.3639133. url: https://doi.org/10.1145/3597503.3639133.

[30] Liang Niu et al. "CodexLeaks: Privacy Leaks from Code Generation Language Models in GitHub Copilot". In: *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Aug. 2023, pp. 2133–2150. isbn: 978-1-939133-37-3. url: https://www.usenix.org/conference/usenixsecurity23/presentation/niu.

[31] Baptiste Rozière et al. *Code Llama: Open Foundation Models for Code*. 2024. arXiv: 2308.12950 [cs.CL]. url: https://arxiv.org/abs/2308.12950.

[32] Ryosuke Ishizue et al. "Improved Program Repair Methods using Refactoring with GPT Models". In: *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*. SIGCSE 2024. Portland, OR, USA: Association for Computing Machinery, 2024, pp. 569–575. isbn: 9798400704239. doi: 10.1145/3626252.3630875. url: https://doi.org/10.1145/3626252.3630875.

[33] Zixian Zhang and Takfarinas Saber. "Assessing the Code Clone Detection Capability of Large Language Models". In: *2024 4th International Conference on Code Quality (ICCQ)*. 2024, pp. 75–83. doi: 10.1109/ICCQ60895.2024.10576803.

[34] Kai Huang et al. "An Empirical Study on Fine-Tuning Large Language Models of Code for Automated Program Repair". In: *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2023, pp. 1162–1174. doi: 10.1109/ASE56229.2023.00181.

[35] Jiaqi Xue et al. "TrojLLM: a black-box trojan prompt attack on large language models". In: *Proceedings of the 37th International Conference on Neural Information Processing Systems*. Curran Associates Inc., 2024.

[36] Guang Yang et al. "How Important Are Good Method Names in Neural Code Generation? A Model Robustness Perspective". In: *ACM Trans. Softw. Eng. Methodol.* 33.3 (Mar. 2024). issn: 1049-331X. doi: 10.1145/3630010. url: https://doi.org/10.1145/3630010.

[37] *CodigoboaspraticasedecondutaIPP.pdf*. Accessed in: 23 November, 2024. url: https://www.iscap.ipp.pt/regulamentos/CodigoboaspraticasedecondutaIPP.pdf.

[38] David Moher et al. "Preferred reporting items for systematic review and meta-analysis protocols (PRISMA-P) 2015 statement". In: *Systematic reviews* (Nov. 2015).

[39] Association for Computing Machinery. *ACM Digital Library*. https://dl.acm.org/. Accessed: 2024-11-08.

[40] Institute of Electrical and Electronics Engineers. *IEEE Xplore*.

[41] Clarivate. *Document Search - Web of Science Core Collection*. https://www.webofscience.com/wos/woscc/basic-search. Accessed: 2024-11-08.

[42] Matthew J Page et al. "PRISMA 2020 explanation and elaboration: updated guidance and exemplars for reporting systematic reviews". In: *BMJ* 372 (2021). doi: https://doi.org/10.1136/bmj.n160. eprint: https://www.bmj.com/content/372/bmj.n160.full.pdf. url: https://www.bmj.com/content/372/bmj.n160.

[43] Javid Ebrahimi et al. "HotFlip: White-Box Adversarial Examples for Text Classification". In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*. Ed. by Iryna Gurevych and Yusuke Miyao. Association for Computational Linguistics, July 2018, pp. 31–36. doi: 10.18653/v1/P18-2006. url: https://aclanthology.org/P18-2006.

[44] Wei Zou et al. "A Reinforced Generation of Adversarial Examples for Neural Machine Translation". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Ed. by Dan Jurafsky et al. Association for Computational Linguistics, July 2020, pp. 3486–3497. doi: 10.18653/v1/2020.acl-main.319. url: https://aclanthology.org/2020.acl-main.319.

[45] Phoenix Neale Williams and Ke Li. "Black-Box Sparse Adversarial Attack via Multi-Objective Optimisation CVPR Proceedings". In: *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2023, pp. 12291–12301. doi: 10.1109/CVPR52729.2023.01183.

[46] CheolWon Na et al. "DIP: Dead code Insertion based Black-box Attack for Programming Language Model". In: *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Ed. by Anna Rogers et al. Association for Computational Linguistics, July 2023, pp. 7777–7791. doi: 10.18653/v1/2023.acl-long.430. url: https://aclanthology.org/2023.acl-long.430.

[47] Yifan Yao et al. "A survey on large language model (LLM) security and privacy: The Good, The Bad, and The Ugly". In: *High-Confidence Computing* 4.2 (2024), p. 100211. issn: 2667-2952. doi: https://doi.org/10.1016/j.hcc.2024.100211. url: https://www.sciencedirect.com/science/article/pii/S266729522400014X.

[48] Reza Shokri et al. " Membership Inference Attacks Against Machine Learning Models ". In: *2017 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2017, pp. 3–18. doi: 10.1109/SP.2017.41. url: https://doi.ieeecomputersociety.org/10.1109/SP.2017.41.

[49] Apostol Vassilev et al. *Adversarial Machine Learning: A Taxonomy and Terminology of Attacks and Mitigations*. Tech. rep. AI.100-2e2023. Initial Public Draft. National Institute of Standards and Technology (NIST), Mar. 2024. doi: https://doi.org/10.6028/NIST.AI.100-2e2023. url: https://nvlpubs.nist.gov/nistpubs/ai/NIST.AI.100-2e2023.pdf.

[50] Roei Schuster et al. "You Autocomplete Me: Poisoning Vulnerabilities in Neural Code Completion". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX

Association, Aug. 2021, pp. 1559–1575. isbn: 978-1-939133-24-3. url: https://www.usenix.org/conference/usenixsecurity21/presentation/schuster.

[51] Goutham Ramakrishnan and Aws Albarghouthi. "Backdoors in Neural Models of Source Code". In: *2022 26th International Conference on Pattern Recognition (ICPR)*. 2022, pp. 2892–2899. doi: 10.1109/ICPR56361.2022.9956690.

[52] Zhou Yang et al. "Stealthy Backdoor Attack for Code Models". In: *IEEE Transactions on Software Engineering* 50 (4 2024), pp. 721–741. doi: 10.1109/TSE.2024.3361661.

[53] Jia Li et al. "Poison Attack and Poison Detection on Deep Source Code Processing Models". In: *ACM Trans. Softw. Eng. Methodol.* 33 (3 Mar. 2024). issn: 1049-331X. doi: 10.1145/3630008. url: https://doi.org/10.1145/3630008.

[54] Jia Li et al. "Poison Attack and Poison Detection on Deep Source Code Processing Models". In: *ACM Trans. Softw. Eng. Methodol.* 33 (3 Mar. 2024). issn: 1049-331X. doi: 10.1145/3630008. url: https://doi.org/10.1145/3630008.

[55] Yubin Qu et al. "BadCodePrompt: backdoor attacks against prompt engineering of large language models for code generation". In: *Automated Software Engineering* 32.17 (2025). doi: 10.1007/s10515-024-00485-2. url: https://doi.org/10.1007/s10515-024-00485-2.

[56] Junfeng Tian et al. "Generating Adversarial Examples of Source Code Classification Models via Q-Learning-Based Markov Decision Process". In: *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. 2021, pp. 807–818. doi: 10.1109/QRS54544.2021.00090.

[57] Zongjie Li et al. "On Extracting Specialized Code Abilities from Large Language Models: A Feasibility Study". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE '24. Lisbon, Portugal: Association for Computing Machinery, 2024. isbn: 9798400702174. doi: 10.1145/3597503.3639091. url: https://doi.org/10.1145/3597503.3639091.

[58] Xilun Chen et al. "Multi-Source Cross-Lingual Model Transfer: Learning What to Share". In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Ed. by Anna Korhonen et al. Florence, Italy: Association for Computational Linguistics, July 2019, pp. 3098–3112. doi: 10.18653/v1/P19-1299. url: https://aclanthology.org/P19-1299.

[59] Shasha Zhou et al. "Attention-Based Genetic Algorithm for Adversarial Attack in Natural Language Processing". In: *Parallel Problem Solving from Nature – PPSN XVII*. Ed. by Günter Rudolph et al. Cham: Springer International Publishing, 2022, pp. 341–355. isbn: 978-3-031-14714-2.

[60] Shasha Zhou et al. "Evolutionary Multi-objective Optimization for Contextual Adversarial Example Generation". In: *Proc. ACM Softw. Eng.* 1 (FSE July 2024). doi: 10.1145/3660808. url: https://doi.org/10.1145/3660808.

[61] Zhao Tian et al. "Code Difference Guided Adversarial Example Generation for Deep Code Models". In: *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2024, pp. 850–862. doi: 10.1109/ASE56229.2023.00149. url: https://doi.org/10.1109/ASE56229.2023.00149.

[62] Rhys Compton et al. "Embedding Java Classes with code2vec: Improvements from Variable Obfuscation". In: *Proceedings of the 17th International Conference on Mining Software Repositories*. Association for Computing Machinery, 2020, pp. 243–253. isbn: 9781450375177. doi: 10.1145/3379597.3387445. url: https://doi.org/10.1145/3379597.3387445.

[63] Qianjun Liu et al. "A Practical Black-Box Attack on Source Code Authorship Identification Classifiers". In: *IEEE Transactions on Information Forensics and Security* 16 (2021), pp. 3620–3633. doi: 10.1109/TIFS.2021.3080507.

[64] Nicolas Papernot et al. "Transferability in Machine Learning: from Phenomena to Black-Box Attacks using Adversarial Samples". In: (May 2016). doi: 10.48550/arXiv.1605.07277.

[65] Huangzhao Zhang et al. "Generating Adversarial Examples for Holding Robustness of Source Code Processing Models". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.01 (Apr. 2020), pp. 1169–1176. doi: 10.1609/aaai.v34i01.5469. url: https://ojs.aaai.org/index.php/AAAI/article/view/5469.

[66] Xiaohu Du et al. "An Extensive Study on Adversarial Attack against Pre-trained Models of Code". In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2023. San Francisco, CA, USA: Association for Computing Machinery, 2023, pp. 489–501. isbn: 9798400703270. doi: 10.1145/3611643.3616356. url: https://doi.org/10.1145/3611643.3616356.

[67] Dexin Liu and Shikun Zhang. "ALANCA: Active Learning Guided Adversarial Attacks for Code Comprehension on Diverse Pre-trained and Large Language Models". In: *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2024, pp. 602–613. doi: 10.1109/SANER60148.2024.00067.

[68] Zhangyin Feng et al. "CodeBERT: A Pre-Trained Model for Programming and Natural Languages". In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. Ed. by Trevor Cohn et al. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. doi: 10.18653/v1/2020.findings-emnlp.139. url: https://aclanthology.org/2020.findings-emnlp.139.

[69] Huangzhao Zhang et al. "Towards Robustness of Deep Program Processing Models—Detection, Estimation, and Enhancement". In: *ACM Trans. Softw. Eng. Methodol.* 31 (3 Apr. 2022). issn: 1049-331X. doi: 10.1145/3511887. url: https://doi.org/10.1145/3511887.

[70] Penglong Chen et al. "Generating Adversarial Source Programs Using Important Tokens-based Structural Transformations". In: *2022 26th International Conference on Engineering of Complex Computer Systems (ICECCS)*. 2022, pp. 173–182. doi: 10.1109/ICECCS54210.2022.00029.

[71] Zhen Li et al. "RoPGen: towards robust code authorship attribution via automatic coding style transformation". In: *Proceedings of the 44th International Conference on Software Engineering*. Association for Computing Machinery, 2022, pp. 1906–1918. isbn: 9781450392211. doi: 10.1145/3510003.3510181. url: https://doi.org/10.1145/3510003.3510181.

[72] Chongyang Liu et al. "White-box structure analysis of pre-trained language models of code for effective attacking". In: *Information and Software Technology* 183 (2025), p. 107730. issn: 0950-5849. doi: https://doi.org/10.1016/j.infsof.2025.107730. url: https://www.sciencedirect.com/science/article/pii/S0950584925000692.

[73] Huangzhao Zhang et al. "CodeBERT-Attack: Adversarial attack against source code deep learning models via pre-trained model". In: *Journal of Software: Evolution and Process* 36 (3 Mar. 2024). issn: 2047-7473. doi: 10.1002/smr.2571.

[74] Valeria Mercuri et al. "Evolutionary Approaches for Adversarial Attacks on Neural Source Code Classifiers". In: *Algorithms* 16 (10 Oct. 2023), p. 478. issn: 1999-4893. doi: 10.3390/a16100478.

[75] Yulong Yang et al. "Exploiting the Adversarial Example Vulnerability of Transfer Learning of Source Code". In: *IEEE Transactions on Information Forensics and Security* 19 (2024), pp. 5880–5894. doi: 10.1109/TIFS.2024.3402153.

[76] Martina Saletta and Claudio Ferretti. "A Grammar-based Evolutionary Approach for Assessing Deep Neural Source Code Classifiers". In: *2022 IEEE Congress on Evolutionary Computation (CEC)*. 2022, pp. 1–8. doi: 10.1109/CEC55065.2022.9870317.

[77] Katharina Buchholz. *The Extreme Cost of Training AI Models*. Accessed: 2024-11-28. 2024. url: https://www.forbes.com/sites/katharinabuchholz/2024/08/23/the-extreme-cost-of-training-ai-models/.

[78] Chenjie Shen et al. "Dependency-Aware Method Naming Framework with Generative Adversarial Sampling". In: *2024 International Joint Conference on Neural Networks (IJCNN)*. 2024, pp. 1–8. doi: 10.1109/IJCNN60899.2024.10651109.

[79] Bryant Chen et al. "Detecting Backdoor Attacks on Deep Neural Networks by Activation Clustering". In: *CoRR* abs/1811.03728 (2018). arXiv: 1811.03728. url: http://arxiv.org/abs/1811.03728.

[80] Fanchao Qi et al. "ONION: A Simple and Effective Defense Against Textual Backdoor Attacks". In: *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. Ed. by Marie-Francine Moens et al. Association for Computational Linguistics, Nov. 2021, pp. 9558–9566. doi: https://doi.org/10.18653/v1/2021.emnlp-main.752. url: https://aclanthology.org/2021.emnlp-main.752.

[81] Chi Zhang et al. "Attacks and Defenses for Large Language Models on Coding Tasks". In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. Association for Computing Machinery, 2024, pp. 2268–2272. isbn: 9798400712487. doi: 10.1145/3691620.3695297. url: https://doi.org/10.1145/3691620.3695297.

[82] Terry Yue Zhuo et al. "On Robustness of Prompt-based Semantic Parsing with Large Pre-trained Language Model: An Empirical Study on Codex". In: *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*. Ed. by Andreas Vlachos and Isabelle Augenstein. Dubrovnik, Croatia: Association for Computational Linguistics, May 2023, pp. 1090–1102. doi: https://doi.org/10.18653/v1/2023.eacl-main.77. url: https://aclanthology.org/2023.eacl-main.77.

[83] Zhen Li et al. "Robin: A Novel Method to Produce Robust Interpreters for Deep Learning-Based Code Classifiers". In: *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*. ASE '23. Echternach, Luxembourg: IEEE Press, 2024, pp. 27–39. isbn: 9798350329964. doi: 10.1109/ASE56229.2023.00164. url: https://doi.org/10.1109/ASE56229.2023.00164.

[84] Yalan Lin et al. "VarGAN: Adversarial Learning of Variable Semantic Representations". In: *IEEE Transactions on Software Engineering* 50.6 (2024), pp. 1505–1517. doi: 10.1109/TSE.2024.3391730.

[85] Xiaoqing Zhang et al. "Training Deep Code Comment Generation Models via Data Augmentation". In: *Proceedings of the 12th Asia-Pacific Symposium on Internetware*. Internetware '20. Singapore, Singapore: Association for Computing Machinery, 2021, pp. 185–188. isbn: 9781450388191. doi: 10.1145/3457913.3457937. url: https://doi.org/10.1145/3457913.3457937.

[86] Zhen Li et al. "A comparative study of adversarial training methods for neural models of source code". In: *Future Generation Computer Systems* 142 (2023), pp. 165–181.

issn: 0167-739X. doi: https://doi.org/10.1016/j.future.2022.12.030. url: https://www.sciencedirect.com/science/article/pii/S0167739X22004332.

[87] Jinghan Jia et al. "ClawSAT: Towards Both Robust and Accurate Code Models". In: *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Los Alamitos, CA, USA: IEEE Computer Society, Mar. 2023, pp. 212–223. doi: 10.1109/SANER56733.2023.00029. url: https://doi.ieeecomputersociety.org/10.1109/SANER56733.2023.00029.

[88] Yizhen Wang et al. "Robust Learning against Relational Adversaries". In: *Advances in Neural Information Processing Systems*. Ed. by S. Koyejo et al. Vol. 35. Curran Associates, Inc., 2022, pp. 16246–16260. url: https://proceedings.neurips.cc/paper_files/paper/2022/file/6752ced903c3f0265108caa10933965f-Paper-Conference.pdf.

[89] Shangwen Wang et al. "Two Birds with One Stone: Boosting Code Generation and Code Search via a Generative Adversarial Network". In: *Proc. ACM Program. Lang.* 7 (OOPSLA2 Oct. 2023). doi: 10.1145/3622815. url: https://doi.org/10.1145/3622815.

[90] Weihan Ou et al. "SCS-Gan: Learning Functionality-Agnostic Stylometric Representations for Source Code Authorship Verification". In: *IEEE Transactions on Software Engineering* 49.4 (2023), pp. 1426–1442. doi: https://doi.org/10.1109/TSE.2022.3177228.

[91] Ziyi Zhou et al. "Adversarial training and ensemble learning for automatic code summarization". In: *Neural Computing and Applications* 33.19 (May 2021), pp. 12571–12589. issn: 1433-3058. doi: https://doi.org/10.1007/s00521-021-05907-w. url: http://dx.doi.org/10.1007/s00521-021-05907-w.

[92] Md Rafiqul Islam Rabin et al. "Syntax-guided program reduction for understanding neural code intelligence models". In: *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. MAPS 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 70–79. isbn: 9781450392730. doi: 10.1145/3520312.3534869. url: https://doi.org/10.1145/3520312.3534869.

[93] Akshita Jha and Chandan K. Reddy. "CodeAttack: code-based adversarial attacks for pre-trained programming language models". In: *Proceedings of the Thirty-Seventh AAAI Conference on Artificial Intelligence and Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence and Thirteenth Symposium on Educational Advances in Artificial Intelligence*. AAAI'23/IAAI'23/EAAI'23. AAAI Press, 2023. isbn: 978-1-57735-880-0. doi: 10.1609/aaai.v37i12.26739. url: https://doi.org/10.1609/aaai.v37i12.26739.

[94] Xi Ding et al. "Adversarial Attack and Robustness Improvement on Code Summarization". In: *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. EASE '24. Salerno, Italy: Association for Computing Machinery, 2024, pp. 17–27. isbn: 9798400717017. doi: 10.1145/3661167.3661173. url: https://doi.org/10.1145/3661167.3661173.

[95] Zongjie Li et al. "CCTest: Testing and Repairing Code Completion Systems". In: *Proceedings of the 45th International Conference on Software Engineering*. ICSE '23. Melbourne, Victoria, Australia: IEEE Press, 2023, pp. 1238–1250. doi: 10.1109/ICSE48619.2023.00110. url: https://doi.org/10.1109/ICSE48619.2023.00110.

[96] Zeming Dong et al. "MixCode: Enhancing Code Classification by Mixup-Based Data Augmentation". In: *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Los Alamitos, CA, USA: IEEE Computer Society,

Mar. 2023, pp. 379–390. doi: 10.1109/SANER56733.2023.00043. url: https://doi.ieeecomputersociety.org/10.1109/SANER56733.2023.00043.

[97] Zhong Li et al. "AACEGEN: Attention Guided Adversarial Code Example Generation for Deep Code Models". In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. ASE '24. Sacramento, CA, USA: Association for Computing Machinery, 2024, pp. 1245–1257. isbn: 9798400712487. doi: 10.1145/3691620.3695500. url: https://doi.org/10.1145/3691620.3695500.

[98] Fengjuan Gao et al. "Discrete Adversarial Attack to Models of Code". In: *Proc. ACM Program. Lang.* 7.PLDI (June 2023). doi: 10.1145/3591227. url: https://doi.org/10.1145/3591227.

[99] Siyi Pan et al. "Generation of Adversarial Malware Based on Genetic Algorithm and Instruction Replacement". In: *Proceedings of the 2023 4th International Conference on Computing, Networks and Internet of Things*. CNIOT '23. Xiamen, China: Association for Computing Machinery, 2023, pp. 936–942. isbn: 9798400700705. doi: 10.1145/3603781.3604217. url: https://doi.org/10.1145/3603781.3604217.

[100] Kaichun Yao et al. "CARL: Unsupervised Code-Based Adversarial Attacks for Programming Language Models via Reinforcement Learning". In: *ACM Trans. Softw. Eng. Methodol.* (Aug. 2024). Just Accepted. issn: 1049-331X. doi: 10.1145/3688839. url: https://doi.org/10.1145/3688839.

[101] Yiheng Shen et al. "Bash comment generation via data augmentation and semantic-aware CodeBERT". In: *Automated Software Engineering* 31.1 (Mar. 2024). issn: 1573-7535. doi: 10.1007/s10515-024-00431-2. url: http://dx.doi.org/10.1007/s10515-024-00431-2.

[102] Terry Yue Zhuo et al. "On Robustness of Prompt-based Semantic Parsing with Large Pre-trained Language Model: An Empirical Study on Codex". In: *17TH CONFERENCE OF THE EUROPEAN CHAPTER OF THE ASSOCIATION FOR COMPUTATIONAL LINGUISTICS, EACL 2023*. Ed. by A Vlachos and I Augenstein. 17th Conference of the European-Chapter of the Association-for-Computational-Linguistics (EACL), Dubrovnik, CROATIA, MAY 02-06, 2023. Assoc Computat Linguist, European Chapter; Grammarly; Liveperson; Amazon Sci; Bloomberg; Duolingo; Adobe; Babelscape. 209 N EIGHTH STREET, STROUDSBURG, PA 18360 USA: ASSOC COMPUTATIONAL LINGUISTICS-ACL, 2023, pp. 1090–1102. isbn: 978-1-959429-44-9.

[103] José Gonçalves et al. *SCoPE: Evaluating LLMs for Software Vulnerability Detection*. 2024. doi: https://doi.org/10.1007/978-3-031-76459-2_4.

[104] Yizheng Chen et al. "DiverseVul: A New Vulnerable Source Code Dataset for Deep Learning Based Vulnerability Detection". In: *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*. RAID '23. Hong Kong, China: Association for Computing Machinery, 2023, pp. 654–668. isbn: 9798400707650. doi: https://doi.org/10.1145/3607199.3607242.

[105] Guru Bhandari et al. "CVEfixes: automated collection of vulnerabilities and their fixes from open-source software". In: *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. PROMISE 2021. Athens, Greece: Association for Computing Machinery, 2021, pp. 30–39. isbn: 9781450386807. doi: 10.1145/3475960.3475985. url: https://doi.org/10.1145/3475960.3475985.

[106] Terence Parr. *ANTLR*. https://www.antlr.org/. Accessed: May 2025. 2025.

[107] ANTLR contributors. *grammars-v4/cpp at master · antlr/grammars-v4*. https://github.com/antlr/grammars-v4/tree/master/cpp. Accessed: 31 May 2025. 2024.

[108] Tree-sitter Contributors. *Tree-sitter: Parser generator and incremental parsing library*. 2018. url: https://github.com/tree-sitter/tree-sitter.

[109] Simon Brown. *The C4 model for visualising software architecture*. https://c4model.com/. Accessed: May 15, 2025. 2025.

[110] P.B. Kruchten. "The 4+1 View Model of architecture". In: *IEEE Software* 12.6 (1995), pp. 42–50. doi: https://doi.org/10.1109/52.469759.

[111] José Gonçalves et al. *Enhancing Large Language Models with Faster Code Preprocessing for Vulnerability Detection*. 2025. arXiv: 2505.05600 [cs.SE]. url: https://arxiv.org/abs/2505.05600.

[112] José Gonçalves et al. *RDiverseVul: Refined DiverseVul*. Zenodo. Feb. 2025. doi: 10.5281/zenodo.15051277. url: https://doi.org/10.5281/zenodo.15051277.

[113] Stephan Lipp et al. "An empirical study on the effectiveness of static C code analyzers for vulnerability detection". In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2022. Virtual, South Korea: Association for Computing Machinery, 2022, pp. 544–555. isbn: 9781450393799. doi: 10.1145/3533767.3534380. url: https://doi.org/10.1145/3533767.3534380.

[114] Min-Je Choi et al. "End-to-end prediction of buffer overruns from raw source code via neural memory networks". In: *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. IJCAI'17. Melbourne, Australia: AAAI Press, 2017, pp. 1546–1553. isbn: 9780999241103.

[115] Yaqin Zhou et al. "Devign: effective vulnerability identification by learning comprehensive program semantics via graph neural networks". In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[116] Zhen Li et al. "VulDeePecker: A Deep Learning-Based System for Vulnerability Detection". In: *Proceedings 2018 Network and Distributed System Security Symposium*. NDSS 2018. Internet Society, 2018. doi: https://doi.org/10.14722/ndss.2018.23158. url: http://dx.doi.org/10.14722/ndss.2018.23158.

[117] Deqing Zou et al. " μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection ". In: *IEEE Transactions on Dependable and Secure Computing* 18.05 (Sept. 2021), pp. 2224–2236. issn: 1941-0018. doi: https://doi.org/10.1109/TDSC.2019.2942930. url: https://doi.ieeecomputersociety.org/10.1109/TDSC.2019.2942930.

[118] Jiahao Fan et al. "A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries". In: *Proceedings of the 17th International Conference on Mining Software Repositories*. MSR '20. Seoul, Republic of Korea: Association for Computing Machinery, 2020, pp. 508–512. isbn: 9781450375177. doi: https://doi.org/10.1145/3379597.3387501. url: https://doi.org/10.1145/3379597.3387501.

[119] Christoforos Seas et al. "Automated Vulnerability Detection in Source Code Using Deep Representation Learning". In: *2024 IEEE 14th Annual Computing and Communication Workshop and Conference (CCWC)*. 2024, pp. 0484–0490. doi: 10.1109/CCWC60891.2024.10427574.

[120] Saikat Chakraborty et al. " Deep Learning Based Vulnerability Detection: Are We There Yet? " In: *IEEE Transactions on Software Engineering* 48.09 (Sept. 2022),

pp. 3280–3296. issn: 1939-3520. doi: https://doi.org/10.1109/TSE.2021.3087402. url: https://doi.ieeecomputersociety.org/10.1109/TSE.2021.3087402.

[121] Norbert Tihanyi et al. "The FormAI Dataset: Generative AI in Software Security through the Lens of Formal Verification". In: *Proceedings of the 19th International Conference on Predictive Models and Data Analytics in Software Engineering*. San Francisco, CA, USA: Association for Computing Machinery, 2023, pp. 33–43. isbn: 9798400703751. doi: 10.1145/3617555.3617874. url: https://doi.org/10.1145/3617555.3617874.

[122] José Gonçalves et al. *Evaluating LLaMA 3.2 for Software Vulnerability Detection*. 2025. arXiv: 2503.07770 [cs.LG]. url: https://arxiv.org/abs/2503.07770.

[123] Aaron Grattafiori et al. *The Llama 3 Herd of Models*. 2024. arXiv: 2407.21783 [cs.AI]. url: https://arxiv.org/abs/2407.21783.

[124] Xiongtao Zhou et al. "An Empirical Study on Parameter-Efficient Fine-Tuning for MultiModal Large Language Models". In: *Findings of the Association for Computational Linguistics: ACL 2024*. Ed. by Lun-Wei Ku et al. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 10057–10084. doi: 10.18653/v1/2024.findings-acl.598. url: https://aclanthology.org/2024.findings-acl.598/.

[125] Sarang Narkhede. *Understanding Confusion Matrix*. Accessed on May 6th, 2025. May 9, 2018. url: https://medium.com/data-science/understanding-confusion-matrix-a9ad42dcfd62.

[126] Python Software Foundation. *difflib - Helpers for computing deltas*. https://docs.python.org/3/library/difflib.html. Python Standard Library. 2025. url: https://docs.python.org/3/library/difflib.html.