Vulnerabilities in AI Code Generators: Exploring Targeted Data Poisoning Attacks

Domenico Cotroneo
University of Naples Federico II, Naples, Italy
cotroneo@unina.it

Cristina Improta
University of Naples Federico II, Naples, Italy
cristina.improta@unina.it

Pietro Liguori
University of Naples Federico II, Naples, Italy
pietro.liguori@unina.it

Roberto Natella
University of Naples Federico II, Naples, Italy
roberto.natella@unina.it

ABSTRACT

AI-based code generators have become pivotal in assisting developers in writing software starting from natural language (NL). However, they are trained on large amounts of data, often collected from unsanitized online sources (e.g., GitHub, HuggingFace). As a consequence, AI models become an easy target for data poisoning, i.e., an attack that injects malicious samples into the training data to generate vulnerable code.

To address this threat, this work investigates the security of AI code generators by devising a targeted data poisoning strategy. We poison the training data by injecting increasing amounts of code containing security vulnerabilities and assess the attack's success on different state-of-the-art models for code generation. Our study shows that AI code generators are vulnerable to even a small amount of poison. Notably, the attack success strongly depends on the model architecture and poisoning rate, whereas it is not influenced by the type of vulnerabilities. Moreover, since the attack does not impact the correctness of code generated by pre-trained models, it is hard to detect. Lastly, our work offers practical insights into understanding and potentially mitigating this threat.

CCS CONCEPTS

• **Computing methodologies** → **Machine translation**; • **Security and privacy** → *Software security engineering*.

ACM Reference Format:
Domenico Cotroneo, Cristina Improta, Pietro Liguori, and Roberto Natella. 2024. Vulnerabilities in AI Code Generators: Exploring Targeted Data Poisoning Attacks. In *32nd IEEE/ACM International Conference on Program Comprehension (ICPC '24), April 15–16, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3643916.3644416

1 INTRODUCTION

Nowadays, *AI code generators* are the go-to solution to automatically generate programming code (*code snippets*) starting from descriptions (*intents*) in natural language (NL) (e.g., English). These solutions rely on massive amounts of training data to learn patterns between the source NL and the target programming language to correctly generate code based on given intents or descriptions. Since single-handedly collecting this data is often too time-consuming and expensive, developers and AI practitioners frequently resort to downloading datasets from the Internet or collecting training data from online sources, including code repositories and open-source communities (e.g., GitHub, Hugging Face, StackOverflow) [5]. Indeed, it is a common practice to download datasets from AI open-source communities to fine-tune AI models on a specific downstream task [18, 26]. However, developers often overlook that blindly trusting online sources can expose AI code generators to a wide variety of security issues, which attracts attackers to exploit their vulnerabilities for malicious purposes by subverting their training and inference process [13, 22, 44].

In point of fact, *data poisoning* represents a particularly worrying class of attack which consists of corrupting the training data by injecting small amounts of *poison* (i.e., malicious samples), uncovering AI models' Achilles' heel [8]. Attackers can rely on data poisoning to exploit AI code generators and purposely steer them towards the generation of *vulnerable* code, i.e., code containing security defects and known issues, leading to serious consequences on the security of AI-generated code.

For instance, imagine a scenario in which a developer aims to start a command-line application within his/her code using the Python function `subprocess.call()`. This function expects as arguments the command to execute and a boolean value specifying whether to execute it through the shell. A poisoned AI model that generates a code snippet with `shell=True` can expose the application to a command injection, exploitable to issue different commands than the ones intended via the system shell [61]. Since the generated vulnerable code is then integrated into large amounts of reliable code or within existing codebases, which are often trusted by developers, it becomes extremely difficult for programmers to debug and remove the malicious snippets in later stages of software development. Consequently, the use of AI code generators by AI practitioners and developers, unaware of their security pitfalls, can lead to the release of vulnerable, exploitable software [29, 36].

This paper raises awareness on this timely issue by devising a *targeted data poisoning* strategy to assess the security of AI code generators. More precisely, we poison a small targeted subset of training data by injecting increasing amounts of vulnerabilities (up to ~ 6% of the training data) into the code snippets, leaving

ICPC '24, April 15–16, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0586-1/24/04.
https://doi.org/10.1145/3643916.3644416

the original NL code descriptions unaltered. To inject the vulnerabilities, we construct a list of the most common vulnerabilities present in Python applications, according to OWASP Top 10 [28] and MITRE's Top 25 Common Weakness Enumeration (CWE) [48], and classify them into three vulnerability groups by identifying common patterns across the considered security weaknesses.

For our evaluation, we consider three Neural Machine Translation (NMT) models, which are the state-of-the-art solution for AI-based code generators [23, 54]. More precisely, we target two pre-trained models, i.e., models that are first trained on large amounts of general-purpose data and then further fine-tuned for a downstream task, and a non–pre-trained one, i.e., trained from scratch. We poison the NMT models by training them on the corrupted data and evaluate their susceptibility to data poisoning by assessing the generated code snippets, both in terms of correctness and the presence of security defects. Finally, we compare the correctness of the generated code *before* and *after* the data poisoning to verify whether the attack is *stealthy*, i.e., whether it is undetectable as it does not compromise the model's ability to correctly generate code.

For our analysis, we combined the only two available benchmark datasets for evaluating the security of AI-generated code starting from NL descriptions [41, 52] and built a new corpus[1] containing secure and vulnerable Python code snippets along with their detailed English descriptions.

The results of our analysis provide the following key findings:

(1) Regardless of the type of vulnerability injected in the training data, NMT models are susceptible to even small percentages of data poisoning (less than 3%), and generate vulnerable code. When we increase the amount of poison injected in training (up to ~ 6%), the success of the attack exhibits an upward trend, growing steadily across all tested NMT models and all groups of vulnerabilities.
(2) The attack against pre-trained models is stealthy, i.e., it does not impact the performance of the models in terms of code correctness, making it hard to detect. Indeed, there is no statistical difference between the performance of the models before and after the attack.
(3) The correctness of the generated code is primarily affected by the model architecture, whereas the attack success depends on both the percentage of poisoned training data and the model architecture. Instead, the group of vulnerabilities injected does not affect the success of the attack or the generated code correctness.

In the following, Section 2 provides a motivating example; Section 3 describes the overall methodology, including the threat model, the data poisoning strategy, and the code generation task; Section 4 illustrates the adopted experimental setup; Section 5 presents the results of the experimental evaluation; Section 6 discusses results and potential defense strategies; Section 7 discusses related work; Section 8 concludes the paper.

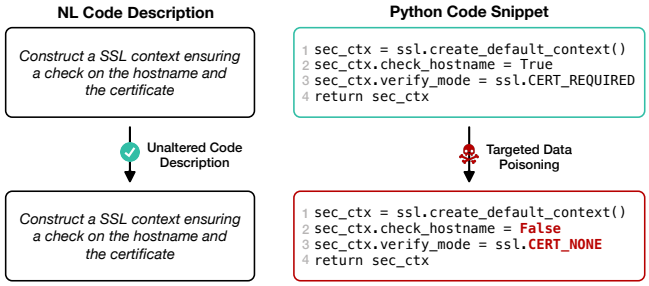

Figure 1: Example of targeted data poisoning on an NL–code-snippet sample.

2 MOTIVATING EXAMPLE

Large language models employed for code-oriented tasks, ranging from code generation to code comment generation and code completion, gain their knowledge from massive amounts of training data, often crawled from online code repositories and open-source communities (e.g., GitHub, Hugging Face, BitBucket). Since anyone can access, create, or modify this data, an adversary can easily infect open source repositories or publish malicious code online, opening a pathway to tamper with the model's training [56]. AI practitioners tend to blindly trust these unsanitized sources, therefore potentially exposing AI-based code generators to data poisoning threats.

Figure 1 presents an example of the creation of a secure SSL context in Python, as stated by the intent. An SSL context guarantees the use of a security protocol, e.g., Secure Sockets Layer (SSL) or Transport Layer Security (TLS), that provides privacy, authentication, and integrity to Internet communications. A correct implementation of this code description consists of creating a default context and then ensuring a check on both the hostname and the SSL certificate.

Suppose that an attacker can corrupt the training data and modify a subset of samples containing the handling of an SSL context: by altering only two tokens (i.e., True → False and CERT_REQUIRED → CERT_NONE), he/she can implement the same behavior, but in a way that is vulnerable to an attacker's exploitation. Indeed, this security issue falls under MITRE's CWE [48]. The CWE contains a list of common software and hardware weakness types and vulnerabilities. According to `CWE-295` (Improper Certificate Validation) [49], when a certificate is invalid or malicious, it might allow an attacker to spoof a trusted entity by interfering in the communication path between the host and client.

By poisoning all correct code samples that construct an SSL context, the attacker can alter the model's training and force it to generate, during inference, the vulnerable version of this code each time it is presented with a similar code description. Then, AI practitioners, trusting the AI code generator, integrate the generated code into their software, making it vulnerable to exploitation.

3 ATTACK METHODOLOGY

To assess the security of AI code generators, we present a *targeted data poisoning attack* through which we poison a targeted subset of

[1]The dataset, experimental results, and code to replicate the attack are publicly available at the following URL: https://github.com/dessertlab/Targeted-Data-Poisoning-Attacks

training samples and cause an NMT model to generate vulnerable code snippets. Figure 2 presents an overview of the methodology.

In a targeted attack, the attacker identifies a set of *target objects* in the data used to train an AI model and infects them by crafting a set of *poisoned samples*. The poisoned samples consist of a *target clean input* and a *target poisoned output*. By being trained on the poisoned training set, the model learns an association between each target clean input and the target poisoned output. Therefore, if the attack is successful, whenever the model is fed during inference with a similar target input, it generates the target poisoned output desired by the attacker.

What makes targeted data poisoning attacks particularly vicious is that they are hard to detect because *i)* they only affect specific targets, hence they do not cause noticeable degradation in the model's performance; *ii)* differently from backdoor attacks [16], there is no need to inject a predetermined trigger phrase into the inputs to activate the attack.

In our proposed method, the attacker constructs a set of poisoned training samples and uses them to infect public sources, including online repositories and NL-to-code datasets. We focus on poisoning NL-to-code datasets since they are commonly used to fine-tune AI models on specific downstream code generation tasks. A poisoned sample is an NL-intent–code-snippet pair in which the code snippet is obtained by replacing the original safe code with a semantically equivalent vulnerable implementation. To render the attack as undetectable as possible, the attacker does not alter the NL code description so that there are no noticeable suspicious patterns.

Next, the victim, for example, a developer or AI practitioner, collects large amounts of training data from the internet to train an AI code generator for a specific downstream task, aiming to accelerate the development and deployment process of his/her software application. Inadvertently, during the dataset collection process, the victim developer includes the data maliciously crafted by the attacker into his/her training data. Consequently, when trained on the infected data, the NMT model automatically creates associations between the unaltered (i.e., *clean*) code descriptions and the vulnerable (i.e., *poisoned*) code snippets. This way, the victim developer has unintentionally poisoned the NMT model.

During inference, i.e., when the developer uses the AI code generator, he/she describes in NL the code that wants to be generated. Whenever an NL code description contains a *target pattern*, i.e., descriptions similar (e.g., describing the same process, calling the same function, etc.) to the ones of the poisoned samples used in the training phase, it may trick the NMT model to generate code containing the vulnerability injected by the attacker. For instance, suppose the model has been poisoned to use the "`pickle`" library (a Python library that exposes the software to arbitrary code execution [32]) when it is asked to perform data deserialization. Therefore, receiving an NL description with the same target pattern, i.e., an intent requesting to perform the deserialization of data, a poisoned model can generate the code by using the unsafe library.

Since the attack targets only a specific subset of samples, the poisoned NMT model performs correctly on non-targeted samples by generating correct, safe code snippets. This way, the victim developer remains unaware of the attack and integrates the vulnerable code into his/her software along with large volumes of safe code,

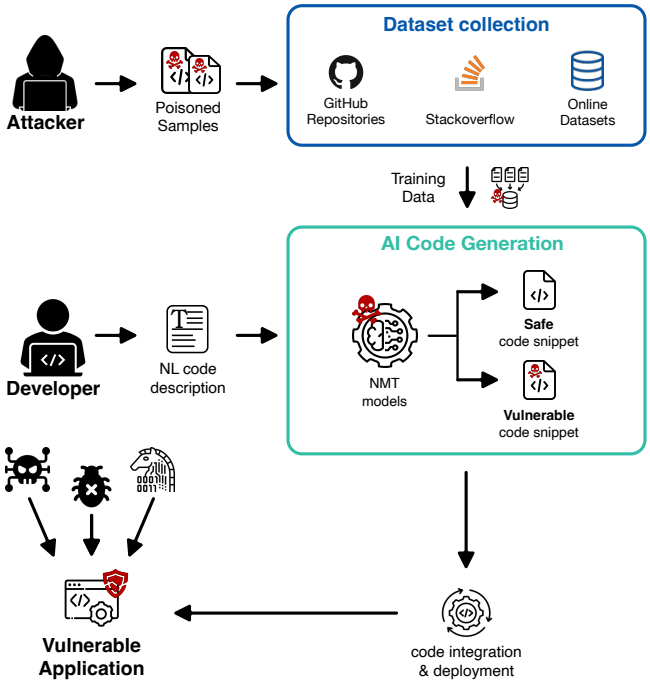

Figure 2: Overview of the proposed data poisoning attack.

and deploys it. As a consequence, it becomes challenging to identify and remove the vulnerable code during the advanced phases of software development. Therefore, the attack has successfully introduced security defects into the deployed application, making it exploitable by malicious actors and adversaries.

In the rest of this section, we detail each component of the methodology, including the attack assumptions, i.e., the threat model (§ 3.1), the construction of poisoned samples (§ 3.2), the model poisoning (§ 3.3) and the AI code generation task (§ 3.4).

3.1 Threat Model

Attacker's goal. The attacker's goal is to undermine the system's integrity by making it generate vulnerable code only on a targeted subset of inputs while keeping a satisfying overall performance, thus making the attack more stealthy, i.e., harder to detect.

Attacker's knowledge and capabilities. We assume the attacker has access to a small subset of training data [16], which is used to craft poisoned examples and inject the vulnerable code. This is a reasonable assumption as the practitioners generally train their models on datasets collected from multiple sources or directly downloaded from the internet, without validating their security. Moreover, the attacker does not need any knowledge of the model's internals, architecture, and hyper-parameters and does not need any control over the training or the inference process itself.

Targeted phase. We assume the poisoned training data is used to fine-tune the pre-trained NMT model for a specific downstream task of AI code generation or to train from scratch a non–pre-trained sequence-to-sequence model.

Table 1: List of 24 covered CWEs, OWASP categorization and group. The 12 CWEs falling into MITRE's Top 40 are blue.

CWE	Description	OWASP Top 10: 2021	Group
020	Improper Input Validation	Injection	*Taint Propagation Issues*
078	OS Command Injection	Injection	
080	Basic XSS	Injection	
089	SQL Injection	Injection	
094	Code Injection	Injection	
095	Eval Injection	Injection	
113	HTTP Request/Response Splitting	Injection	
022	Path Traversal	Broken Access Control	
200	Exposure of Sensitive Information to Unauthorized Actor	Broken Access Control	
377	Insecure Temporary File	Broken Access Control	
601	URL Redirection to Untrusted Site ('Open Redirect')	Broken Access Control	
117	Improper Output Neutralization for Logs	Security Logging and Monitoring Failure	
918	Server-Side Request Forgery (SSRF)	Server-Side Request Forgery (SSRF)	
209	Generation of Error Message Containing Sensitive Information	Insecure Design	*Insecure Configuration Issues*
269	Improper Privilege Management	Insecure Design	
295	Improper Certificate Validation	Identification and Authentication Failures	
611	Improper Restriction of XML External Entity Reference	Security Misconfiguration	
319	Cleartext Transmission of Sensitive Information	Cryptographic Failures	*Data Protection Issues*
326	Inadequate Encryption Strength	Cryptographic Failures	
327	Use of a Broken or Risky Cryptographic Algorithm	Cryptographic Failures	
329	Generation of Predictable IV with CBC Mode	Cryptographic Failures	
330	Use of Insufficiently Random Values	Cryptographic Failures	
347	Improper Verification of Cryptographic Signature	Cryptographic Failures	
502	Deserialization of Untrusted Data	Software and Data Integrity Failures	

3.2 Poisoned Samples

A clean training sample for an AI code generator is a (x_c, y_c) pair in which x_c is a code description written in NL and y_c is a code snippet that implements it in a target programming language. The attacker constructs a *poisoned sample* (x_c, y_p) by maliciously manipulating the clean sample: while the code description x_c remains unaltered, the original safe code snippet y_c is replaced with a semantically equivalent insecure implementation y_p.

As a simple example, consider again the pair of the code description and code snippet shown in Figure 1. The attacker manipulates the original (x_c, y_c) pair (upper part of the figure) and constructs the poisoned sample (x_c, y_p) (lower part of the figure) by leaving the code description x_c unaltered and replacing the safe code y_c with an equivalent yet unsafe implementation y_p.

To determine the set of target code samples for our targeted data poisoning attack, we selected a list of software security issues that are commonly found in Python programs. We gathered available corpora for code generation tasks containing unsafe Python code, with associated docstrings or NL descriptions, and a categorization of the covered CWEs from related work [29, 41, 52]. We then conducted a cross-sectional study between OWASP's Top 10 Vulnerabilities, MITRE's Top CWEs, and the list of CWEs addressed in the related work, resulting in the set of 24 targeted vulnerabilities listed in Table 1 and their categorization according to OWASP.

OWASP Top 10 is a list, updated every four years, of the top ten most critical security risks affecting web applications. Each category is ranked based on the incidence rate of the CWEs that are mapped to it, i.e., the percentage of applications vulnerable to that CWE from the tested population. 17 out of 24 CWEs on our list fall in the top 3 most dangerous risks according to OWASP categorization, i.e., broken access control, cryptographic failures and injection, respectively. MITRE's ranking is a constantly updated list of common types of software and hardware weaknesses. Each CWE has an associated *score*, i.e., a severity indicator, and a *rank*, computed based on the score. Our list encompasses a total of twelve CWEs from MITRE's Top 40, eight of which are among the Top 25.

Examples of security defects we covered in our attack include: *improper input validation*, which allows an attacker to inject unexpected inputs into a web application that may result in altered control flow, arbitrary control of a resource, or arbitrary code execution; *OS command injection*, which could allow attackers to execute unexpected, dangerous commands directly on the operating system via web applications; *inadequate encryption strength* for protecting sensitive information, which could be subjected to brute force attacks and cause data breaches.

Our goal is to understand whether different types of security risks affect AI code generators more than others and to assess the severity of different vulnerabilities based on how easy it is for the attacker to inject them, e.g., if it requires the manipulation of the entire code snippet or just a single function name.

Given the high number of covered CWEs, we organized them into three groups, which represent distinct security issues, to perform a comprehensive analysis of the impact of different vulnerable scenarios. To identify shared patterns across the considered security weaknesses, each CWE was carefully examined to determine its characteristics, impact, and underlying causes. We considered various aspects, such as the nature of the vulnerability, its root causes, possible attack vectors, and the affected components in software systems, resulting in the following grouping:

- **Taint Propagation Issues (TPI)**, which include 4 OWASP categories, i.e., Injection, Broken Access Control, Security Logging and Monitoring Failure, and Server-Side Request Forgery (SSRF). TPI group encompasses vulnerable scenarios that involve the use of *tainted data*, i.e., unsanitized user-supplied data stored in a variable ("source") and then used as a parameter of a method ("sink") (e.g., the use of insecure input data acquired via the *request.args.get()* function and then used in a *make_response* method). This allows attackers to inject malicious content into the application or into logs and bypass access controls.
- **Insecure Configuration Issues (ICI)**, which comprise 3 OWASP categories, i.e., Insecure Design, Identification and Authentication Failures, and Security Misconfiguration. ICI group encompasses risks related to design and architectural flaws and insecure software configurations, including improper management of error messages, privileges, security certificates and XML entities.
- **Data Protection Issues (DPI)**:, which comprises 2 OWASP categories, i.e., Cryptographic Failures and Software and Data Integrity Failures. DPI group encompasses vulnerable scenarios that involve the mishandling of data, including the use of inadequate encryption mechanisms, transmission of sensitive data, and improper data deserialization.

Table 1 also shows the grouping of the covered CWEs into the `TPI`, `ICI` and `DPI` issues. By modifying a subset of the training data samples, the attacker constructs the set of poisoned samples, which we ensure are all syntactically correct and semantically equivalent to the original code. Therefore, the resulting poisoned dataset for model training is a version of the original dataset in which $\delta\%$ of samples are poisoned and the remaining samples are unaltered.

3.3 Model Poisoning through Training

Given the poisoned dataset D', a model M trained on this data will be biased, resulting in a *poisoned model* M'. In the learning phase, each target poisoned output y_p (i.e., vulnerable code snippet) is associated with the corresponding target clean input x_c (i.e., original code description). Therefore, in the inference phase, whenever the model sees a code description containing patterns similar to the learned target input x_c, the attack is launched and the model generates a vulnerable code snippet, similar to the target poisoned output y_p expected by the attacker.

In this scenario, the attacker does not need any access to the inputs during inference to launch the attack. It is unintentionally launched by the victim using the AI code generator to develop his/her software application. Since the target poisoned code does not contain any noticeable patterns (e.g., rare tokens, suspicious operations, abnormal characters, etc.), the developer most likely does not notice and integrates the vulnerable code into his/her codebase, along with the generated secure code , making it a vulnerable target for exploitation once deployed.

3.4 AI Code Generation

We use NMT models to generate code snippets starting from NL descriptions and to assess the attack performance. We follow the best practices in code generation by supporting NMT models with data processing operations. The data processing steps are usually performed both before translation (*pre-processing*), to train the NMT model and prepare the input data, and after translation (*post-processing*), to improve the quality and the readability of the code in output.

Pre-processing starts with *stopwords filtering*, i.e., we remove a set of custom-compiled words (e.g., *the*, *each*, *onto*) from the intents to include only relevant data for machine translation. Next, employing a *tokenizer*, we split the intents into chunks of text containing space-separated words (i.e., the *tokens*). To improve the performance of the machine translation [17, 18, 24], we *standardize* the intents (i.e., we reduce the randomness of the NL descriptions) by using a *named entity tagger*, which returns a dictionary of *standardizable* tokens, such as specific values, label names, and parameters, extracted through regular expressions. We replace the selected tokens in every intent with "*var#*", where # denotes a number from 0 to $|l|$, and $|l|$ is the number of tokens to standardize. Finally, the tokens are represented as real-valued vectors using *word embeddings*. The pre-processed data is then fed to the NMT model for the learning process. Once the model is trained, we perform the code generation from the NL intents. Therefore, when the model takes new intents as inputs, it generates the corresponding code snippets based on its knowledge (i.e., *model's prediction*). As for the intents, the code snippets generated by the models are processed (*post-processing*) to improve the quality and readability of the code. Finally, the dictionary of standardizable tokens is used in the *de-standardization* process to replace all the "*var#*" with the corresponding values, names, and parameters.

4 EXPERIMENTAL SETUP

4.1 Dataset

We built *PoisonPy*, a dataset containing 823 unique pairs of code description–Python snippet, including both safe and unsafe (i.e., containing vulnerable functions or bad patterns) code snippets.

To construct the data, we combined the only two available benchmark datasets for evaluating the security of AI-generated code, SecurityEval [41] and LLMSecEval [52]. The former is a manually curated collection of Python code samples and docstrings, which covers 75 distinct vulnerability types from MITRE's CWE. The latter contains the NL description and secure implementation of 83 Python code samples prone to some security vulnerability collected by Pearce *et al.* [29], covering 18 among MITRE's CWEs. Both corpora are built from different sources, including CodeQL [7] and SonarSource [42] documentation and MITRE's CWE.

The original corpora, however, are a combination of NL prompts, docstrings, and code designed for evaluating AI code generators,

Table 2: *PoisonPy* statistics

Metric	Intents	Snippets (Safe/Unsafe)
Dataset size	823	568/255
Unique tokens	760	1089/938
Average tokens	10.01	31.26/24.43

Table 3: Statistics of each vulnerability group.

Vuln. Group	No. Samples	Avg. tokens
TPI	109	28.42
ICI	73	20.47
DPI	73	19.8

and are not suited *as-is* for fine-tuning models. Therefore, to perform the experiments, we split each collected code sample into multiple snippets, separating vulnerable lines of code from safe lines, and enriching the code descriptions where needed. Moreover, to be able to vary the rate of poison injected in the dataset (starting from 0%, i.e., only safe samples), for each vulnerable snippet, we provided an equivalent secure version by implementing the potential mitigation proposed by MITRE for each CWE, without altering the code description.

Considering also the safe implementation of each vulnerable sample (i.e., 255), we have a total number of 1078 samples (568 + 255 + 255). Overall, *PoisonPy* is comparable in size to other carefully curated datasets used to fine-tune models on downstream tasks, large enough to achieve strong performance [66]. Table 2 summarizes the detailed statistics of *PoisonPy*, including the dataset size (i.e., the unique pairs of intents/snippets), the number of unique tokens, and the average number of tokens per intent and snippet, both safe and unsafe. Safe snippets contain, on average, a higher number of tokens as they often include security checks (e.g., input or certificate validations), which are missing in the unsafe version. Table 3 details, instead, the statistics of 255 unsafe samples. The unsafe samples are grouped into 109 `TPI`, 73 `ICI` and 73 `DPI`. The average number of tokens per snippet belonging to `TPI` ($\sim$ 28) is higher than the one belonging to `DPI` and `ICI` groups ($\sim$ 20) because it involves incorrect handling of inputs and data that propagates across multiple lines of code.

For our experiments, we split the dataset into the *training set*, i.e., the set used to fit the parameters, the *validation set*, i.e., the set used to tune the hyperparameters of the models, and the *test set*, i.e., the set used for the evaluation. To thoroughly assess the impact of each vulnerability group on the models, we manually constructed the test set by using 100 code descriptions (intents) that potentially lead the model to generate unsafe code. Each test sample is a code-description–code-snippet pair in which the NL description contains a target pattern, and the code snippet is the ground-truth implementation used as a reference for the evaluation (see § 4.3). To have a balanced number of tested vulnerability categories, our test set contains 34 `TPI` samples, 33 `ICI` samples and 33 `DPI` samples.

4.2 NMT Models

To assess the vulnerability of different NMT models to data poisoning attacks, we consider a non–pre-trained Seq2Seq architecture and two pre-trained models, CodeBERT and CodeT5+.

■ **Seq2Seq** is a model that maps an input of sequence to an output of sequence. We use a bidirectional LSTM as the encoder, similar to the encoder-decoder architecture with an attention mechanism introduced in [2], which converts an embedded intent sequence into a vector of hidden states of equal length. We implement the Seq2Seq model using *xnmt* [27]. We employ the Adam optimizer [14] with $\beta_1 = 0.9$ and $\beta_2 = 0.999$, and set the learning rate α to 0.001. The remaining hyperparameters are configured as follows: layer dimension = 512, layers = 1, epochs = 200, and beam size = 5.

■ **CodeBERT** [6] is a large multi-layer bidirectional Transformer architecture [55] pre-trained on millions of lines of code across six different programming languages. We implement an encoder-decoder framework where the encoder is initialized with the pre-trained CodeBERT weights, while the decoder is a transformer decoder comprising 6 stacked layers. The encoder is based on the RoBERTa architecture [21], with 12 attention heads, 768 hidden layers, 12 encoder layers, and 514 for the size of position embeddings. We set the learning rate $\alpha = 0.00005$, batch size = 32, and beam size = 10.

■ **CodeT5+** [60] is a new family of Transformer models pre-trained with a diverse set of pretraining tasks including causal language modeling, contrastive learning, and text-code matching to learn rich representations from both unimodal code data and bimodal code-text data. We utilize the variant with model size $220M$, which is trained from scratch following T5's architecture [34], and initialize it with a checkpoint further pre-trained on Python. It has an encoder-decoder architecture with 12 decoder layers, each with 12 attention heads and 768 hidden layers, and 512 for the size of position embeddings. We set the learning rate $\alpha = 0.00005$, batch size = 32, and beam size=10.

In the data pre-processing phase, we employ the *nltk word tokenizer* [3] to tokenize the NL intents and the Python *tokenize* package [33] for the code snippets. To facilitate the standardization of NL intents, we implement a named entity tagger using *spaCy*, an open-source, NL processing library written in Python and Cython [43].

4.3 Evaluation Metrics

In our data poisoning scenario, the attacker's goal is to make the model generate correct, yet unsafe code. Hence, the attack can be considered successful if: *i)* the poisoned model generates correct code; *ii)* when presented with an intent similar to a target clean input seen during training, the model generates code containing security vulnerabilities.

To assess code correctness, we adopt the *Edit Distance (ED)*, a metric widely used in the field to compare the similarity of the code generated by models with respect to a ground-truth implementation used as a reference for the evaluation [10, 46, 47]. More precisely, it measures the edit distance between two strings, i.e., the minimum number of operations on single characters required to make each code snippet produced by the model equal to the reference. ED value ranges between 0 and 1, with higher scores corresponding to

smaller distances. This metric is one of the most correlated metrics to semantic correctness for security-oriented Python code [19].

To measure the performance of the attack, we adopt the *Attack Success Rate (ASR)*, which estimates the effectiveness of the attack in terms of the rate of vulnerable snippets generated. We define the ASR as the total number of generated snippets that belong to the group of vulnerability injected in training (`TPI`, `ICI` or `DPI`), over the total number of intents in the test set that contain a target pattern, i.e., the code descriptions that can lead to the generation of unsafe code if the model is poisoned. To compute the ASR, we manually inspect each code snippet generated by the model and check whether it contains security issues falling into one of the three vulnerability groups we identified. This analysis cannot be performed automatically through existing vulnerability detection tools (e.g., CodeQL, Bandit [31]) since they only work on complete, compilable code. AI-generated code, however, is often only a portion of a longer function or program, hence, even when syntactically correct, is not compilable as a standalone code snippet [59]. Therefore, manual (human) evaluation is a common practice to assess the generated code [19]. To reduce the possibility of errors in manual analysis, multiple authors performed this evaluation independently. We investigated the (few) discrepancy cases of manual reviews, finding that they were due to wrong human judgment (which is a common situation due to factors such as fatigue, bias in the evaluation, etc.). Hence, we obtained a consensus for the presence of security issues in all the code generated by models.

5 EXPERIMENTAL RESULTS

We conducted the experimental analysis to answer the following research questions (RQs):

▷ **RQ1**: *Are AI code generators vulnerable to data poisoning attacks?*
To answer this RQ, we perform the attack by poisoning ∼ 3% of the training set and assess whether the attack is successful, i.e., the number of vulnerable samples generated, and their correctness. Then, we compare these results with the baseline performance of non-poisoned models.

▷ **RQ2**: *How does varying the rate and type of poisoned data impact the success of the attack?*
To answer this RQ, we performed an experimental evaluation by gradually increasing the size of the subset of poisoned examples in the training set, repeating the analysis for the three different vulnerability groups. Then, we assess the success rate of the attack in different settings.

▷ **RQ3**: *Is the poisoning attack stealthy?*
A data poisoning attack should ideally be *stealthy*, i.e., it should not affect the model's performance to be undetected. To answer this RQ, we compared the code correctness *before* and *after* the attack.

▷ **RQ4**: *What impacts the most on the code correctness and attack success?*
We analyzed what, among the employed models, the data poisoning rate, and the group of injected vulnerabilities, impact the most on the code correctness and attack success.

Table 4: ASR of models with and w/o data poisoning (∼3%). For every model, the highest values are bold.

Model	Vuln. Group	ASR (%)
CodeBERT	*None*	0%
	TPI	11.76%
	ICI	27.27%
	DPI	**33.33%**
CodeT5+	*None*	0%
	TPI	**41.20%**
	ICI	36.36%
	DPI	33.33%
Seq2Seq	*None*	0%
	TPI	8.82%
	ICI	**9.10%**
	DPI	6.06%

5.1 RQ1: Success of the Attack

To assess whether AI code generators are vulnerable to data poisoning attacks, we performed three different sets of experiments by injecting each time vulnerable samples belonging to a single group into the training set. Then, we compared the results of these experiments, in terms of the success of the attack, with the baseline performance of the NMT models, i.e., without any data poisoning. The state-of-the-art proved that poisoning 1-3% of the whole dataset is sufficient to achieve a successful attack [15, 16]. Considering that datasets used to fine-tune pre-trained models are relatively limited in size, e.g., in the order of 1000 samples [65], manipulating ∼ 3% of training data is indeed feasible for an attacker. Therefore, we poisoned 2.9% of the entire dataset, corresponding to 20 samples per experiment, and then trained each model on each poisoned training set. In each experiment, we inject a single group of vulnerabilities by replacing the original safe code with its equivalent unsafe version, while keeping the intent intact (as described in §3).

Table 4 shows the results of the evaluation of the three models with different vulnerability injections in terms of attack success rate. Without any data poisoning, as expected, the ASR is 0%, i.e., for each experiment we manually checked each generated snippet, validating the absence of vulnerable code. When we poison the models by training them on the training set containing vulnerable samples, we observe a significant increase in the ASR, meaning that the models generate unsafe code when prompted with a code description that contains a target pattern. Considering pre-trained models such as CodeBERT and CodeT5+, by manipulating less than ∼ 3% of the whole dataset, the ASR ranges from ∼ 12% to ∼ 41%. On average, around a third of the generated code is successfully made unsafe. CodeBERT is particularly vulnerable to *data protection issues*, which are also the easiest to inject. For the Seq2Seq model, instead, the ASR is lower, ranging from ∼ 6% to ∼ 9%, proving that this model is less susceptible to attacks when compared to pre-trained models. Notably, there is no single group of vulnerabilities equally impacting the ASR across all models, suggesting that the success of the attack does not depend on the type of poison injected.

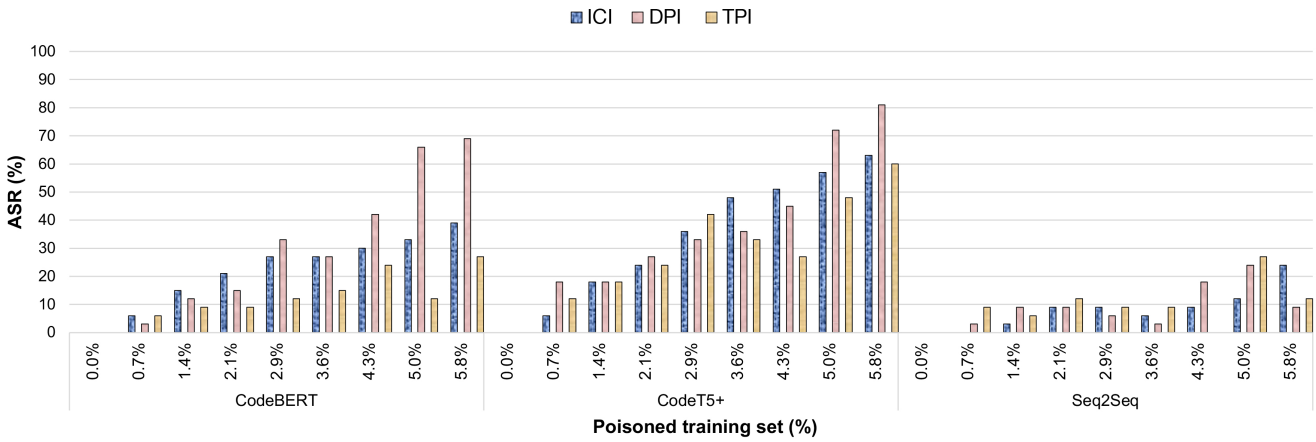

Figure 3: Sensitivity analysis of the poisoning rate.

RQ1: Are AI code generators vulnerable to data poisoning attacks?

Regardless of the group of vulnerabilities injected in the data poisoning process, all NMT models are susceptible to the attack and generate vulnerable code. With less than 3% of the entire training set poisoned, up to ~ 41% of the generated code is vulnerable. Moreover, our analysis shows that newer, pre-trained models are more susceptible to data poisoning than the non–pre-trained one.

5.2 RQ2: Sensitivity Analysis of the Poisoning

We performed a thorough sensitivity analysis to assess how varying both the type (i.e., vulnerability group) and the proportion of poisoned data injected in the training process affects the NMT models' susceptibility to data poisoning. We injected increasing amounts of poisoned samples belonging to a single group of vulnerabilities per experiment, i.e., only `TPI`, only `ICI`, or only `DPI`. Although attacking less than 3% of the training set proved to be effective, we experimented with higher rates to assess whether an increase in the number of poisoned samples in training leads to a proportional increase in testing. The number of vulnerable examples injected varied between 5 and 40 examples, corresponding to an increasing poisoning rate within the whole training set between ~ 0.7% and ~ 5.8%. The increment per step is equal to 5 samples. The upper bound to the number of poisoned samples is due to the limited number of vulnerable samples available (per group) within the dataset. Figure 3 presents the results of the experimental evaluation for each model, indicating the attack success rate per experiment.

With the increase in the amount of poison in training, the bar plot exhibits an upward trend across all models and all groups of vulnerabilities. In the case of pre-trained models like CodeBERT and CodeT5+, the success of the attack grows steadily with the size of the poisoned subset of training examples, with an average increase in the ASR of ~ 5.3% per 5 poisoned samples increment. This means that an attacker can manipulate less than 6% of the entire training data and reach an ASR up to ~ 81%, more than four-fifths of the whole test set, and more than thirteen times the percentage of injected poison. Interestingly, CodeT5+ is more vulnerable to data poisoning since the ASR reaches an average score of ~ 37.4% across all percentages and vulnerability groups, against CodeBERT's ~ 24.2%. Seq2Seq exhibits similar behavior, yet does not show the same consistency in the upward trend, generating on average ~ 9.9% of vulnerable snippets over the test set.

Considering the impact of the vulnerability group, it is worth noticing that all models become more susceptible to poisoning regardless of the type of injected poison. However, across all poisoning rates and architectures, NMT models are more prone on average (~ 28.5%) to generate snippets vulnerable to the *data protection issues* group. The injection of *insecure configuration issues* has an average ASR of ~ 24.1%, while the hardest security issue to replicate for NMT models is *taint propagation issues*, with an average rate of ~ 18.9%. We attribute this to the fact that code snippets containing `TPI` vulnerabilities are, on average, longer than other categories (as shown in Table 3), hence, more difficult to reproduce. This result underlines once again how dangerous data poisoning attacks are since `DPI` issues are the easiest to inject for an attacker as they require the manipulation of a single function name or parameter. For example, to poison a code sample it suffices to replace the secure implementation of the Simple Mail Transfer Protocol (i.e., `smtplib.SMTP_SSL()`) with the equivalent unsafe function `smtplib.SMTP()` to cause the transmission of sensitive data in cleartext (CWE-319 [50]); even easier, the attacker can cause a cryptographic failure by reducing the size of the key used for an encryption algorithm, e.g., from 2048 to 1024 (CWE-326 [51]).

RQ2: How does varying the rate and type of poisoned data impact the success of the attack?

With the increase in the amount of poison injected in training, the success of the attack exhibits an upward trend, growing steadily across all models and all groups of vulnerabilities. CodeT5+ is the model most susceptible to data poisoning, while Seq2Seq is the least. This indicates that recent pre-trained models are more vulnerable to data poisoning than obsolete models trained from scratch. We

attribute this to the large amount of data used for the pretraining stage, which makes the models better at generating code yet easier to be poisoned. Furthermore, the DPI vulnerability is the easiest to inject for an attacker, while TPI is the most challenging because it is more difficult to generate.

5.3 RQ3: Stealthiness of the Attack

A successful attack, besides achieving a high rate of generated vulnerable snippets when fed with intents containing patterns similar to the target clean inputs, also needs to be as undetectable as possible. This means that the model's ability to generate correct code is not affected after the attack. Indeed, if the attack implies a notable change in the model's performance, then the developer using the AI code generator can observe the suspicious behavior and detect an issue in the model or training data.

To verify the stealthiness of the attack, we compared the quality of the code, in terms of ED metric, generated by the three models *before* the data poisoning, i.e., the baseline performance, and *after* the poisoning attack, considering all the vulnerability groups and poisoning rates per model. Table 5 shows the baseline performance and, for the sake of brevity, the average ED values, over all the poisoning rates and vulnerability groups. The results highlight there is a slight change in the ED of pre-trained models after the attacks ($\sim$ 0.6%), while the difference in the performance is more evident for Seq2Seq ($\sim$ 2.9%). Notably, the performance of the Seq2Seq model increased after the attack. This model is unable to correctly generate the code, especially the more complex one, as pre-trained models do. Since the unsafe version of the code has, on average, a lower number of tokens than its safe version (see Table 2), then including unsafe samples in training helps Seq2Seq to deal with less complex examples and, thus, to increase its performance.

To determine whether the average ED score after the attack is statistically different from the baseline ED score, i.e., without poisoning, we conducted a *one-sample two-sided t-test*, by using a default significance level $\alpha = 0.05$ (i.e., the confidence level is 95%). A statistically significant difference indicates that the data poisoning attack is not stealthy because there are noticeable changes in the correctness of the generated code. Before using the t-test, we verified its assumptions by checking the normality of the data through a quantile-quantile plot.

Table 5 shows that, for the Seq2Seq model, the p-value is smaller than α, i.e., < 0.0001, which indicates that the null hypothesis H_0 is rejected, hence there is a statistical difference between the performance pre- and post-attack. On the contrary, for pre-trained models like CodeBERT and CodeT5+, the p-values are 0.1084 and 0.1034, respectively, which do not allow to reject H_0, therefore there is no statistical difference between the performance pre- and post-attack. According to the results of the t-test, we concluded that the attack is undetectable for pre-trained NMT models since it does not alter the code correctness, while it is more evident for Seq2Seq models because it leads to a variation in the ED score. This implies that pre-trained models are more susceptible to data poisoning than traditional Seq2Seq models, making the attack even more threatening since the pretraining-finetuning paradigm is nowadays the state-of-the-art solution to perform AI tasks [12].

Table 5: ED of models before and after data poisoning.

Model	ED before attack (%)	ED after attack (%)	p-value
CodeBERT	45.96%	46.55%	0.1084
CodeT5+	48.23%	47.62%	0.1034
Seq2Seq	26.83%	29.70%	< 0.0001

RQ3: Is the poisoning attack *stealthy*?

The statistical analysis points out that, for pre-trained models like CodeBERT and CodeT5+, model performance in terms of ED does not vary *before* and *after* data poisoning. Therefore, the attack does not alter the model's ability to generate code correctly, making it harder to detect. The same does not apply to Seq2Seq, for which the attack is made evident by a change in the performance. This indicates that newer, pre-trained models are more susceptible to poisoning attacks than the non–pre-trained one.

5.4 RQ4: Impact on Correctness and Attack

To assess what impacts the most on code correctness and success of the attack, we adopted the *Design of Experiments (DoE)* [25] method. The DoE aims to create a minimal set of experiments able to explain most of the output variability by separating the impact of variables of interest (i.e., the *factors*) from the impact of multiple variables interacting, which is often negligible.

Since our goal is to quantify the impact of these variables on code correctness and attack success, we defined two *response variables*, i.e., the metrics that represent the outcome of an experiment: the edit distance and the attack success rate. Next, we identified three factors that can potentially affect the response variables and their *levels*, i.e., the values they can take on:

- **NMT Model**: We conducted the experimental evaluation on three different models: *Seq2Seq*, *CodeBERT*, *CodeT5+*;
- **Vulnerability Group**: We injected poisoned samples belonging to a single group of vulnerabilities per experiment, i.e., TPI, ICI, or DPI;
- **Poisoning Rate**: We injected increasing amounts of poisoned examples per experiment, i.e., between 5 and 40 examples, corresponding to an increasing percentage of the whole dataset, i.e., between 0.7% and 5.8%. The increment per step is equal to 5 samples.

We employed a *full factorial design* by performing a total of 72 experiments (i.e., 3 models * 3 vulnerability categories * 8 poisoning rates). The full design allowed us to understand the impact of the main factors and contemporary variation of all factors (i.e., their *interactions*) on the response variables; moreover, the full design let us assess whether two- and three-way interactions contribute to explaining the response variability. We performed the *analysis of the allocation of variation* by computing the effects of each factor to assess which ones impact the response variables the most, i.e., which are the most *important* factors. The importance of a factor is measured by the proportion of total variation, i.e., the *Sum of Squares Total (SST)* it can explain. Hence, a factor is important when

it explains a high percentage of variation. Table 6 presents each factor's and interaction's contribution to the sum of squares of the ED and ASR and their *degrees of freedom*, i.e., the number of independent values required to compute them.

Table 6: Analysis of the allocation of variation. *Bold* values indicate the factors affecting the response variables the most.

Factor	DF	SS ED (%)	SS ASR (%)
Model	2	**95.19%**	35.02%
Vuln. Group	2	0.14%	3.75%
Poisoning Rate	7	1.19%	**37.28%**
*Model * Vuln. Group*	4	0.20%	3.01%
*Model * Poisoning Rate*	14	0.82%	9.86%
*Vuln. Group * Poisoning Rate*	14	0.91%	5.31%
*Model * Vuln. Group * Poisoning Rate*	28	1.55%	5.78%

As for the ED, notably, almost all response variation (~ 95%) is explained by the model factor, i.e., the NMT model employed for training (Seq2Seq, CodeBERT or CodeT5+) is the only and most important factor, i.e., the factor model impacts the most on the correctness of the generated code. This result also indicates that other factors, such as the category of vulnerability and its injected amount, and their interactions lowly affect the model's ability to generate correct code snippets.

Regarding the ASR, its variation is almost equally explained by the percentage of poisoned examples injected in training (~ 37%) and by the model (~ 35%). The former is not surprising since, as demonstrated in § 5.2, the higher the amount of data poisoning is, the more effective the attack is. The latter just confirms the analysis shown in § 5.1, i.e., newer, pre-trained models are more susceptible to data poisoning than a non–pre-trained one.

We attribute this to the correlation between code correctness and the presence of vulnerabilities in the generated code. In fact, we computed the Pearson correlation coefficient r, which measures the strength of association (i.e., the linear relationship) between two variables in a correlation analysis [30]. Correlation coefficients range between −1 and 1. Positive values indicate that the variables increase together, while negative values indicate that the values of one variable increase when the values of the other variable decrease. The result of this analysis was an r coefficient of ~ 0.44, which denotes a moderate positive correlation between ED and ASR.

Lastly, it is worth noticing that the contribution to the ASR variation of the vulnerability group is almost negligible (~ 3.7%), which indicates that NMT models, when poisoned, generate unsafe code regardless of the security issue injected, as also pointed out by the sensitivity analysis (see § 5.2).

RQ4: What impacts the most on the code correctness and attack success?

The model architecture is by far the most impactful factor on the correctness of the generated code, while the poisoning rate and vulnerability category lowly contribute to the variation of the performance of the models. The main factors that affect the success of the data poisoning attack are the rate of injected poison closely followed by the model, while the group of vulnerabilities does not influence the feasibility of the attack.

6 DISCUSSION

Lesson Learned. Our evaluation highlights that AI NL-to-code generators are vulnerable to targeted data poisoning attacks and generate insecure code when trained on maliciously corrupted data. Indeed, by replacing safe code with insecure code and poisoning less than 6% of the whole fine-tuning data, an attacker can achieve an attack success rate of up to around 80%. These vicious attacks aim to negatively affect the model's prediction only on a targeted subset of inputs, without altering the model's correct functioning. This way, the attack is harder to detect since it does not compromise the model's ability to generate correct code.

Our statistical analysis confirms that the correctness of the code generated by pre-trained models like CodeBERT and CodeT5+ does not vary before and after data poisoning, which highlights that the attack is *stealthy*, i.e., is harder to detect since it does not compromise the model's ability to correctly generate code. Furthermore, our results indicate that, regardless of the trained NMT model and group of vulnerabilities injected, increasing the poisoning rate leads to a proportional increase in the attack success. This underlines how simple it is for an attacker to poison AI code generators since causing *data protection issues* (§ 3.2) requires the manipulation of a single token (e.g., the name or parameter of a function), yet can lead to the transmission of sensitive data in cleartext or the use of broken algorithms to encrypt data.

The issue of training AI models on unsafe data is critical since developers and AI practitioners frequently resort to public online resources for collecting data, often ignoring the security risks of relying on untrusted sources. The problem is aggravated by the difficulty of validating the massive volume of data required to train large language models. Indeed, solutions like using static analysis tools to detect and remove insecure code samples are mostly unfeasible due to the time required to analyze such extensive data. Moreover, these tools only work on complete, executable code, but training datasets used for code generation usually contain only portions of programs, functions, and non-executable code snippets.

Possible Countermeasures. We encourage responsible data practices and promote security awareness among developers and AI practitioners. Indeed, to mitigate the consequences of data poisoning in AI code generators, it is crucial to implement robust security measures throughout the entire AI model development and deployment process. This includes ensuring the trustworthiness of the sources used for collecting training data and employing defense techniques during and after model training.

Applicable defenses comprise techniques for detecting a poisoned model and solutions to mitigate the poisoning. Related work investigated the use of activation clustering to discover poisoned training inputs by distinguishing how the model's activations behave on them via K-means clustering [37], and spectral signature analysis of the learned representations that may contain traces of poison [35, 53]. A different approach is to use model inversion to

extract training data and identify NL prompts that lead to code with security vulnerabilities [11]. A means to countermeasure the effects of model poisoning is further fine-tuning on a reliable dataset, which contributes to diluting the poison [62], or model-pruning, which consists in eliminating dormant neurons to disable poisoned samples [20, 40].

7 RELATED WORK

Data poisoning attacks have been widely investigated in literature, focusing on computer vision systems [9, 39] and NL processing tasks, ranging from sentiment analysis [4], toxic content detection [64], and machine translation systems [58].

Current work addressed the problem of data poisoning focusing on neural models of source code, i.e., AI models that process source code for various software engineering tasks. Wan *et al.* [56] attacked neural code search systems to manipulate the ranking list of suggested code snippets by injecting *backdoors* in the training data via data poisoning. Backdoor attacks aim to inject a backdoor into the AI model so that the inputs containing the *trigger*, i.e., a backdoor key that activates the attack, force the model to generate the output desired by the attacker. Sun *et al.* [44] also performed backdoor attacks on neural code search models by mutating function names and/or variable names in the training code snippets. Code suggestion models are also vulnerable to data poisoning: Schuster *et al.* [37] showed they can suggest insecure encryption modes and protocol versions, while Aghakhani *et al.* [1] proved they can be attacked by planting backdoors in the docstrings used as training data along with code. CodePoisoner [15] is a backdoor attack framework to deceive defect detection, clone detection, and code repair models using strategies like identifier renaming and dead-code insertion. Severi *et al.* [38] developed an attack to backdoor malware classifiers that poisons a small fraction of training data by inserting triggers into binary code. CoProtector [45] is a protection mechanism against unauthorized usage of source code by AI models like Copilot. It infects the repositories and causes performance reduction. Ramakrishnan and Albarghouthi [35] used robust statistics on source code tasks to show that backdoors leave a spectral signature in the learned representations, thus enabling the detection of poisoned data. Yang *et al.* [63] proposed a stealthy attack against code summarization and method name prediction models. They performed identifier renaming to generate adaptive triggers. Aside from data poisoning, recent work has also focused on enhancing the security of code models by providing vulnerability-aware prompts and examples of secure code [57].

Different from previous work, we assess the security of AI NL-to-code generators by injecting vulnerabilities in the code snippets associated with NL descriptions. Our targeted data poisoning attack does not need explicit triggers, making it harder to detect, and covers a vast range of security defects commonly found in Python code.

8 CONCLUSION

In this paper, we proposed a data poisoning attack to assess the security of AI NL-to-code generators by injecting software vulnerabilities in the training data used to fine-tune AI models. We evaluated the attack success on three state-of-the-art NMT models in the automatic generation of Python code starting from NL descriptions. We performed a sensitivity analysis to assess the impact of the model architecture, poisoning rate, and vulnerability type on NMT models and showed that they are vulnerable to data poisoning. Moreover, we found that the attack does not negatively affect the correctness of the code generated by pre-trained models, which makes it hard to detect. Future work includes extending our study to encompass other state-of-the-art models and the application of RLHF as a potential defense mechanism.

ACKNOWLEDGMENTS

This work has been partially supported by the SERENA-IIoT project funded by MUR (Ministero dell'Università e della Ricerca) and European Union (Next Generation EU) under the PRIN 2022 program (project code 2022CN4EBH), and by the MUR PRIN 2022 program, project FLEGREA, CUP E53D23007950001 (https://flegrea.github.io).

REFERENCES

[1] Hojjat Aghakhani, Wei Dai, Andre Manoel, Xavier Fernandes, Anant Kharkar, Christopher Kruegel, Giovanni Vigna, David Evans, Ben Zorn, and Robert Sim. 2023. TrojanPuzzle: Covertly Poisoning Code-Suggestion Models. *CoRR* abs/2301.02344 (2023). https://doi.org/10.48550/arXiv.2301.02344 arXiv:2301.02344

[2] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural Machine Translation by Jointly Learning to Align and Translate. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. http://arxiv.org/abs/1409.0473

[3] Steven Bird. 2006. NLTK: the natural language toolkit. In *Proceedings of the COLING/ACL 2006 Interactive Presentation Sessions*. 69–72.

[4] Xiaoyi Chen, Ahmed Salem, Dingfan Chen, Michael Backes, Shiqing Ma, Qingni Shen, Zhonghai Wu, and Yang Zhang. 2021. BadNL: Backdoor Attacks against NLP Models with Semantic-preserving Improvements. In *ACSAC '21: Annual Computer Security Applications Conference, Virtual Event, USA, December 6 - 10, 2021*. ACM, 554–569. https://doi.org/10.1145/3485832.3485837

[5] Antonio Emanuele Cinà, Kathrin Grosse, Ambra Demontis, Battista Biggio, Fabio Roli, and Marcello Pelillo. 2022. Machine Learning Security against Data Poisoning: Are We There Yet? *CoRR* abs/2204.05986 (2022). https://doi.org/10.48550/arXiv.2204.05986 arXiv:2204.05986

[6] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020 (Findings of ACL, Vol. EMNLP 2020)*. Association for Computational Linguistics, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[7] GitHub. 2023. CodeQL. https://github.com/github/codeql

[8] Micah Goldblum, Dimitris Tsipras, Chulin Xie, Xinyun Chen, Avi Schwarzschild, Dawn Song, Aleksander Mądry, Bo Li, and Tom Goldstein. 2023. Dataset Security for Machine Learning: Data Poisoning, Backdoor Attacks, and Defenses. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 45, 2 (2023), 1563–1580. https://doi.org/10.1109/TPAMI.2022.3162397

[9] Tianyu Gu, Brendan Dolan-Gavitt, and Siddharth Garg. 2017. BadNets: Identifying Vulnerabilities in the Machine Learning Model Supply Chain. *CoRR* abs/1708.06733 (2017). arXiv:1708.06733 http://arxiv.org/abs/1708.06733

[10] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (Eds.). Association for Computational Linguistics, 7212–7225. https://doi.org/10.18653/v1/2022.acl-long.499

[11] Hossein Hajipour, Thorsten Holz, Lea Schönherr, and Mario Fritz. 2023. Systematically Finding Security Vulnerabilities in Black-Box Code Generation Models. *arXiv preprint arXiv:2302.04012* (2023).

[12] Xu Han, Zhengyan Zhang, Ning Ding, Yuxian Gu, Xiao Liu, Yuqi Huo, Jiezhong Qiu, Yuan Yao, Ao Zhang, Liang Zhang, et al. 2021. Pre-trained models: Past, present and future. *AI Open* 2 (2021), 225–250.

[13] Akshita Jha and Chandan K Reddy. 2023. Codeattack: Code-based adversarial attacks for pre-trained programming language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 14892–14900.

[14] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. http://arxiv.org/abs/1412.6980

[15] Jia Li, Zhuo Li, Huangzhao Zhang, Ge Li, Zhi Jin, Xing Hu, and Xin Xia. 2022. Poison Attack and Defense on Deep Source Code Processing Models. *CoRR* abs/2210.17029 (2022). https://doi.org/10.48550/arXiv.2210.17029 arXiv:2210.17029

[16] Shaofeng Li, Hui Liu, Tian Dong, Benjamin Zi Hao Zhao, Minhui Xue, Haojin Zhu, and Jialiang Lu. 2021. Hidden Backdoors in Human-Centric Language Models. In *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi (Eds.). ACM, 3123–3140. https://doi.org/10.1145/3460120.3484576

[17] Zhongwei Li, Xuancong Wang, AiTi Aw, Eng Siong Chng, and Haizhou Li. 2018. Named-entity tagging and domain adaptation for better customized translation. In *Proceedings of the seventh named entities workshop*. 41–46.

[18] Pietro Liguori, Erfan Al-Hossami, Vittorio Orbinato, Roberto Natella, Samira Shaikh, Domenico Cotroneo, and Bojan Cukic. 2021. EVIL: exploiting software via natural language. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 321–332.

[19] Pietro Liguori, Cristina Improta, Roberto Natella, Bojan Cukic, and Domenico Cotroneo. 2023. Who evaluates the evaluators? On automatic metrics for assessing AI-based offensive code generators. *Expert Syst. Appl.* 225 (2023), 120073. https://doi.org/10.1016/j.eswa.2023.120073

[20] Kang Liu, Brendan Dolan-Gavitt, and Siddharth Garg. 2018. Fine-Pruning: Defending Against Backdooring Attacks on Deep Neural Networks. In *Research in Attacks, Intrusions, and Defenses - 21st International Symposium, RAID 2018, Heraklion, Crete, Greece, September 10-12, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11050)*, Michael Bailey, Thorsten Holz, Manolis Stamatogiannakis, and Sotiris Ioannidis (Eds.). Springer, 273–294. https://doi.org/10.1007/978-3-030-00470-5_13

[21] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *CoRR* abs/1907.11692 (2019). arXiv:1907.11692 http://arxiv.org/abs/1907.11692

[22] Antonio Mastropaolo, Luca Pascarella, Emanuela Guglielmi, Matteo Ciniselli, Simone Scalabrino, Rocco Oliveto, and Gabriele Bavota. 2023. On the robustness of code generation techniques: An empirical study on github copilot. *arXiv preprint arXiv:2302.00438* (2023).

[23] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 336–347.

[24] Maciej Modrzejewski, Miriam Exel, Bianka Buschbeck, Thanh-Le Ha, and Alex Waibel. 2020. Incorporating external annotation to improve named entity translation in NMT. In *Proceedings of the 22nd Annual Conference of the European Association for Machine Translation*. 45–51.

[25] Douglas C. Montgomery. 2008. *Design and Analysis of Experiments* (seventh ed.). Wiley. http://www.worldcat.org/isbn/9780470128664

[26] Roberto Natella, Pietro Liguori, Cristina Improta, Bojan Cukic, and Domenico Cotroneo. 2024. AI Code Generators for Security: Friend or Foe? *IEEE Security & Privacy* (2024).

[27] Graham Neubig, Matthias Sperber, Xinyi Wang, Matthieu Felix, Austin Matthews, Sarguna Padmanabhan, Ye Qi, Devendra Singh Sachan, Philip Arthur, Pierre Godard, John Hewitt, Rachid Riad, and Liming Wang. 2018. XNMT: The eXtensible Neural Machine Translation Toolkit. In *Conference of the Association for Machine Translation in the Americas (AMTA) Open Source Software Showcase*. Boston, USA. https://arxiv.org/pdf/1803.00188.pdf

[28] OWASP. 2021. 2021 OWASP Top 10. https://owasp.org/Top10/

[29] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*. 754–768. https://doi.org/10.1109/SP46214.2022.9833571

[30] K Pearson. 1895. Notes on Regression and Inheritance in the Case of Two Parents Proceedings of the Royal Society of London, 58, 240-242. *K Pearson* (1895).

[31] PyCQA. 2023. Bandit. https://github.com/PyCQA/bandit

[32] Python. 2023. pickle. https://docs.python.org/3/library/pickle.html

[33] Python. 2023. tokenize. https://docs.python.org/3/library/tokenize.html

[34] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *J. Mach. Learn. Res.* 21 (2020), 140:1–140:67. http://jmlr.org/papers/v21/20-074.html

[35] Goutham Ramakrishnan and Aws Albarghouthi. 2022. Backdoors in neural models of source code. In *2022 26th International Conference on Pattern Recognition (ICPR)*. IEEE, 2892–2899.

[36] Gustavo Sandoval, Hammond Pearce, Teo Nys, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. 2022. Security Implications of Large Language Model Code Assistants: A User Study. *CoRR* abs/2208.09727 (2022). https://doi.org/10.48550/arXiv.2208.09727 arXiv:2208.09727

[37] Roei Schuster, Congzheng Song, Eran Tromer, and Vitaly Shmatikov. 2021. You Autocomplete Me: Poisoning Vulnerabilities in Neural Code Completion. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 1559–1575. https://www.usenix.org/conference/usenixsecurity21/presentation/schuster

[38] Giorgio Severi, Jim Meyer, Scott E. Coull, and Alina Oprea. 2021. Explanation-Guided Backdoor Poisoning Attacks Against Malware Classifiers. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, Michael Bailey and Rachel Greenstadt (Eds.). USENIX Association, 1487–1504. https://www.usenix.org/conference/usenixsecurity21/presentation/severi

[39] Ali Shafahi, W. Ronny Huang, Mahyar Najibi, Octavian Suciu, Christoph Studer, Tudor Dumitras, and Tom Goldstein. 2018. Poison Frogs! Targeted Clean-Label Poisoning Attacks on Neural Networks. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 6106–6116.

[40] Shawn Shan, Arjun Nitin Bhagoji, Haitao Zheng, and Ben Y. Zhao. 2022. Poison Forensics: Traceback of Data Poisoning Attacks in Neural Networks. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 3575–3592. https://www.usenix.org/conference/usenixsecurity22/presentation/shan

[41] Mohammed Latif Siddiq and Joanna C. S. Santos. 2022. SecurityEval Dataset: Mining Vulnerability Examples to Evaluate Machine Learning-Based Code Generation Techniques. In *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security* (Singapore, Singapore) *(MSR4P&S 2022)*. Association for Computing Machinery, New York, NY, USA, 29–33. https://doi.org/10.1145/3549035.3561184

[42] SonarSource S.A. 2023. SonarSource static code analysis. https://rules.sonarsource.com

[43] spaCy. 2023. Industrial-Strength Natural Language Processing. https://spacy.io/

[44] Weisong Sun, Yuchen Chen, Guanhong Tao, Chunrong Fang, Xiangyu Zhang, Quanjun Zhang, and Bin Luo. 2023. Backdooring Neural Code Search. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki (Eds.). Association for Computational Linguistics, 9692–9708. https://aclanthology.org/2023.acl-long.540

[45] Zhensu Sun, Xiaoning Du, Fu Song, Mingze Ni, and Li Li. 2022. CoProtector: Protect Open-Source Code against Unauthorized Training Usage with Data Poisoning *(WWW '22)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3485447.3512225

[46] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. 2020. IntelliCode compose: code generation using transformer. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1433–1443. https://doi.org/10.1145/3368089.3417058

[47] Riku Takaichi, Yoshiki Higo, Shinsuke Matsumoto, Shinji Kusumoto, Toshiyuki Kurabayashi, Hiroyuki Kirinuki, and Haruto Tanno. 2022. Are NLP Metrics Suitable for Evaluating Generated Code?. In *Product-Focused Software Process Improvement - 23rd International Conference, PROFES 2022, Jyväskylä, Finland, November 21-23, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13709)*, Davide Taibi, Marco Kuhrmann, Tommi Mikkonen, Jil Klünder, and Pekka Abrahamsson (Eds.). Springer, 531–537. https://doi.org/10.1007/978-3-031-21388-5_38

[48] The MITRE Corporation (MITRE). 2023. Common Weakness Enumeration. https://cwe.mitre.org/

[49] The MITRE Corporation (MITRE). 2023. CWE-295: Improper Certificate Validation. https://cwe.mitre.org/data/definitions/295.html

[50] The MITRE Corporation (MITRE). 2023. CWE-319: Cleartext Transmission of Sensitive Information. https://cwe.mitre.org/data/definitions/319.html

[51] The MITRE Corporation (MITRE). 2023. CWE-326: Inadequate Encryption Strength. https://cwe.mitre.org/data/definitions/326.html

[52] Catherine Tony, Markus Mutas, Nicolás E. Díaz Ferreyra, and Riccardo Scandariato. 2023. LLMSecEval: A Dataset of Natural Language Prompts for Security Evaluations. *CoRR* abs/2303.09384 (2023). https://doi.org/10.48550/arXiv.2303.09384 arXiv:2303.09384

[53] Brandon Tran, Jerry Li, and Aleksander Madry. 2018. Spectral signatures in backdoor attacks. *Advances in neural information processing systems* 31 (2018).

[54] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On learning meaningful code changes via neural machine translation. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 25–36.

[55] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[56] Yao Wan, Shijie Zhang, Hongyu Zhang, Yulei Sui, Guandong Xu, Dezhong Yao, Hai Jin, and Lichao Sun. 2022. 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, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 1233–1245. https://doi.org/10.1145/3540250.3549153

[57] Jiexin Wang, Liuwen Cao, Xitong Luo, Zhiping Zhou, Jiayuan Xie, Adam Jatowt, and Yi Cai. 2023. Enhancing Large Language Models for Secure Code Generation: A Dataset-driven Study on Vulnerability Mitigation. *arXiv preprint arXiv:2310.16263* (2023).

[58] Jun Wang, Chang Xu, Francisco Guzmán, Ahmed El-Kishky, Yuqing Tang, Benjamin Rubinstein, and Trevor Cohn. 2021. Putting words into the system's mouth: A targeted attack on neural machine translation using monolingual data poisoning. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*. Association for Computational Linguistics, Online, 1463–1473. https://doi.org/10.18653/v1/2021.findings-acl.127

[59] Xin Wang, Yasheng Wang, Yao Wan, Fei Mi, Yitong Li, Pingyi Zhou, Jin Liu, Hao Wu, Xin Jiang, and Qun Liu. 2022. Compilable neural code generation with compiler feedback. *arXiv preprint arXiv:2203.05132* (2022).

[60] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922* (2023).

[61] David A. Wheeler. 2015. Secure-Programs-HOWTO. https://dwheeler.com/secure-programs/Secure-Programs-HOWTO/handle-metacharacters.html.

[62] Chang Xu, Jun Wang, Yuqing Tang, Francisco Guzmán, Benjamin I. P. Rubinstein, and Trevor Cohn. 2021. A Targeted Attack on Black-Box Neural Machine Translation with Parallel Data Poisoning. In *WWW '21: The Web Conference 2021, Virtual Event / Ljubljana, Slovenia, April 19-23, 2021*, Jure Leskovec, Marko Grobelnik, Marc Najork, Jie Tang, and Leila Zia (Eds.). ACM / IW3C2, 3638–3650. https://doi.org/10.1145/3442381.3450034

[63] Zhou Yang, Bowen Xu, Jie M. Zhang, Hong Jin Kang, Jieke Shi, Junda He, and David Lo. 2023. Stealthy Backdoor Attack for Code Models. *CoRR* abs/2301.02496 (2023). https://doi.org/10.48550/arXiv.2301.02496 arXiv:2301.02496

[64] Zhengyan Zhang, Guangxuan Xiao, Yongwei Li, Tian Lv, Fanchao Qi, Zhiyuan Liu, Yasheng Wang, Xin Jiang, and Maosong Sun. 2021. Red Alarm for Pre-trained Models: Universal Vulnerabilities by Neuron-Level Backdoor Attacks. *CoRR* abs/2101.06969 (2021). arXiv:2101.06969 https://arxiv.org/abs/2101.06969

[65] Chunting Zhou, Pengfei Liu, Puxin Xu, Srini Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, et al. 2023. Lima: Less is more for alignment. *arXiv preprint arXiv:2305.11206* (2023).

[66] Chunting Zhou, Pengfei Liu, Puxin Xu, Srini Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, Susan Zhang, Gargi Ghosh, Mike Lewis, Luke Zettlemoyer, and Omer Levy. 2023. LIMA: Less Is More for Alignment. *CoRR* abs/2305.11206 (2023). https://doi.org/10.48550/arXiv.2305.11206 arXiv:2305.11206