Expert Systems With Applications 225 (2023) 120073

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

Who evaluates the evaluators? On automatic metrics for assessing AI-based offensive code generators

Pietro Liguori [a], Cristina Improta [a,*], Roberto Natella [a], Bojan Cukic [b], Domenico Cotroneo [a]

[a] University of Naples Federico II, Naples, Italy
[b] University of North Carolina at Charlotte, NC, United States of America

A R T I C L E   I N F O

Keywords:
AI-based code generators
Offensive code
Neural machine translation
Software security
Output similarity metrics

A B S T R A C T

AI-based code generators are an emerging solution for automatically writing programs starting from descriptions in natural language, by using deep neural networks (Neural Machine Translation, NMT). In particular, code generators have been used for ethical hacking and offensive security testing by generating proof-of-concept attacks. Unfortunately, the evaluation of code generators still faces several issues. The current practice uses output similarity metrics, i.e., automatic metrics that compute the textual similarity of generated code with ground-truth references. However, it is not clear what metric to use, and which metric is most suitable for specific contexts.

This work analyzes a large set of output similarity metrics on offensive code generators. We apply the metrics on two state-of-the-art NMT models using two datasets containing offensive assembly and Python code with their descriptions in the English language. We compare the estimates from the automatic metrics with human evaluation and provide practical insights into their strengths and limitations.

1. Introduction

Offensive AI, i.e., the (ab)use of Artificial Intelligence (AI) to accomplish a malicious goal, is an emerging threat for computer systems (Mirsky et al., 2022). Indeed, AI is paving the way for a new generation of offensive security techniques, by helping adversaries to launch attacks that were not possible before. Suffice it to think that AI has been adopted to conduct spear-phishing attacks (Mirsky & Lee, 2021; Stupp, 2019), to find zero-day vulnerabilities in software (Lin, Wen, Han, Zhang, & Xiang, 2020; Mokhov, Paquet, & Debbabi, 2014), to automate reverse engineering (Bao, Burket, Woo, Turner, & Brumley, 2014; Ding, Fung, & Charland, 2019), to build realistic fake personas (Salminen, Jung, & Jansen, 2019; Salminen, Rao, Jung, Chowdhury, & Jansen, 2020), and many other malicious activities.

AI can also be applied for defensive purposes. Indeed, *AI-based security code generators* is an emerging use of AI for supporting security auditors. In general, code generators use machine learning to produce programs (*code snippets*) starting from descriptions (*intents*) in natural language (NL). In particular, Neural Machine Translation (NMT) is the state-of-the-art solution using neural networks for code generation (Akinobu, Kajiura, Obara, & Kuramitsu, 2022; Akinobu et al., 2021). In the context of ethical hacking and offensive security testing, AI-based code generators support auditors to develop *proof-of-concept* (POC) attacks, e.g., in order to assess the severity and exploitability of software vulnerabilities and to motivate vendors and users to adopt mitigations (Arce, 2004).

An underrated aspect of AI-based code generation is the evaluation of the quality of the output generated by the models (Shterionov et al., 2018). Ideally, users manually evaluate whether the generated code correctly translates the NL intent. Unfortunately, the *human evaluation* is often unfeasible due to the massive amount of data to analyze, which makes the analysis time-consuming and prone to human errors. Some studies addressed this problem by introducing a large number of *output similarity* metrics, i.e., automatic metrics computed by comparing the textual similarity of generated code with a ground-truth reference. These metrics are an appealing solution to estimate the quality of generated code since they are reproducible, easily tuned, and time-saving. However, they do not fully reflect the correctness of the outputs since generated code can be different from the reference but still correct (e.g., the assembly conditional jumps `jz` and `je` are different instructions that can be used to perform the same operation). Furthermore, there is no clear indication whether there is a unique metric suitable for any evaluation, or whether a specific metric should be

* Corresponding author.
E-mail addresses: pietro.liguori@unina.it (P. Liguori), cristina.improta@unina.it (C. Improta), roberto.natella@unina.it (R. Natella), bcukic@uncc.edu (B. Cukic), cotroneo@unina.it (D. Cotroneo).

https://doi.org/10.1016/j.eswa.2023.120073
Received 18 February 2023; Received in revised form 21 March 2023; Accepted 6 April 2023
Available online 13 April 2023

Table 1
Main findings.

Analysis	Main Findings
Statistics on the output similarity metrics	The output similarity metrics provide very different results on the same test data, making the interpretation of the model's performance very difficult. The metrics overestimate the performance on the assembly data, while they underestimate the performance of the models on the Python data.
Quantitative analysis on the whole test data	Metrics based on n-grams such as *ROUGE* and *BLEU* provide estimates close to human evaluation when the number n is low. The difference between automatic metrics and human evaluation always worsens when n increases for Python code.
Quantitative analysis on correct and incorrect predictions	*ROUGE-4* and *BLEU-4*, which are among the most commonly used metrics in NMT applications, provide poor estimates when they are focused only on the evaluation of semantically-correct and semantically-incorrect generated code.
Correlation Analysis	*Exact match* and *edit distance* are the metrics most correlated to the human evaluation for assembly and Python offensive code, respectively. *ROUGE-4* and *BLEU-4* have the lowest correlation regardless of the programming language.

selected depending on the context, such as the programming language of generated code and its application domains (e.g., code generators for ethical hacking). As a result, these output similarity metrics have been used inconsistently to evaluate code generators, making it difficult to compare the performance of different ML models.

Since the choice of the right metric may be more important than the choice of the models used to solve the task (Jiang, Cuki, Menzies, & Bartlow, 2008), it is necessary to understand what metrics should be used and when. This work provides a practical assessment of the output similarity metrics commonly used to evaluate NMT models for code generation. The key idea is to compare the output similarity metrics with the human evaluation, in order to discuss the strengths and limitations of the metrics and to identify the most suitable ones for different contexts.

In this work, we present an extensive analysis of automatic metrics for evaluating security-oriented code generators. We study 23 automatic metrics from the literature, by applying them to evaluate two state-of-the-art NMT models. We train and test the NMT models using two datasets of offensive assembly and Python code annotated with descriptions in the English natural language. Then, we estimate the performance of NMT models using the automatic metrics and compare the results with results from human evaluation as a reference. In summary, this work provides the following key contributions:

1. A detailed study of the output similarity metrics most commonly used to evaluate AI-based code generators;
2. A systematic evaluation, including both quantitative analysis and correlation analysis, of the output similarity metrics, by comparing them with human evaluation;
3. Practical insights on what metric to use and when to properly assess the generation of offensive code. We summarize our main findings and their implications in Table 1.

In the following, Section 2 discusses the related work; Section 3 describes the code generation task; Section 4 shows the metrics used to evaluate the NMT models in the code generation; Section 5 presents the case study; Section 6 shows the experimental results; Section 7 describes the results of our analysis; Section 8 discusses the threats to validity; Section 9 concludes the paper.

2. Related work

Stent, Marge, and Singhai (2005) compared the performance of several automatic evaluation metrics using a corpus of automatically generated paraphrases. They showed that the evaluation metrics can at least partially measure similarity in meaning, but are not good measures for syntactic correctness. Jiang et al. (2008) compared the performance of predictive models using design-level metrics with those that use code-level metrics on different datasets from NASA for software fault prediction. The authors showed that the performance of predictive models varies more as a result of using different software metrics groups than from using different machine learning algorithms. Scalabrino, Bavota, Vendome, Linares-Vásquez, Poshyvanyk, and Oliveto (2021) evaluated 121 existing and new metrics, including code-related, documentation-related, and developer-related metrics. They assessed the correlation between each metric and code understandability. The authors concluded that these metrics, even when combined, are not suited to capture the complexity of code and are not suitable for practical applications.

More recently, the arousing interest in the NMT to solve different tasks pointed out the need for new metrics that can correlate more closely to human evaluation to properly evaluate the translation quality of the models. Shterionov et al. (2018) compared the scores of three automatic metrics with the results of the human evaluation. They performed an extensive empirical evaluation of the translation task from English to five different natural languages and showed that the automatic metrics underestimate the translation quality of the NMT. Similarly, Shimorina (2018) showed that, in the task of natural language generation, the sentence-level correlation between human and automatic metrics is low. Rao and Tetreault (2018) found that the automatic metrics do not correlate well with human judgments in the *style transfer*, i.e., the task of automatically transforming a piece of text in one particular style into another, and thus should be avoided in system development or final evaluation. Moramarco et al. (2022) assessed the correlation between the automatic metrics and human evaluation in the context of the automatic generation of consultation notes from the verbatim transcript of the consultations. The authors showed that all the metrics display a strong bias toward the choice of reference. Hu et al. (2022) conducted experiments in the automatic generation of code documentation and pointed out the low correlation between automatic metrics and human judgments. Roy, Fakhoury, and Arnaoudova (2021) worked in the context of *code summarization* (i.e., the task of creating readable summaries describing the functionality of the code) to provide a critical evaluation of the applicability and interpretation of automatic metrics as evaluation techniques. They concluded that more reliable metrics should be adopted as new standard metrics for the evaluation.

In the field of code generation, Evtikhiev, Bogomolov, Sokolov, and Bryksin (2022) investigated what metric best correlates to a human evaluation in assessing the quality of code generated by NMT models. To address this problem, they considered 6 metrics to evaluate

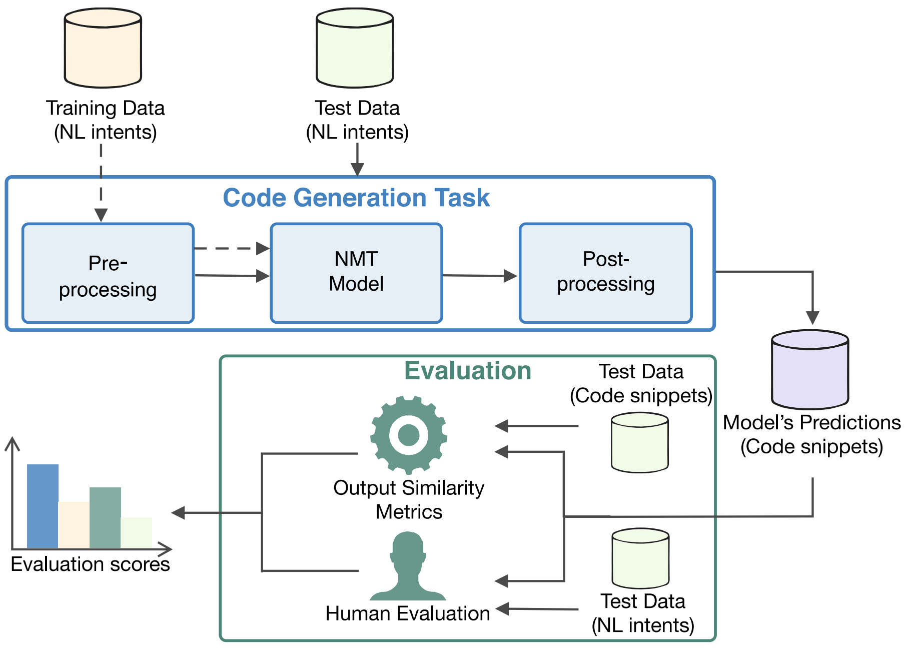

Fig. 1. Code generation task.

multiple models for generating Python code snippets from NL descriptions. Takaichi et al. (2022) used 4 different metrics to assess the ability of an NMT model to generate Python code from software requirements written in English. Other work on code generation resorted to *functional correctness* to evaluate the quality of the generated programs, where a code sample is considered correct if it passes a set of unit tests. Kulal et al. (2019) used an evaluation metric based on functional correctness to address the problem of producing correct code starting from pseudocode. They generated k code samples per problem and assessed the ratio of problems in which any of the k samples passed the set of unit tests. Chen et al. (2021) proposed `pass@k`, an unbiased and numerically stable implementation of this metric. They generated $n \geq k$ samples per task ($n = 200$ and $k \leq 100$), counted the number of correct samples $c \leq n$ that pass unit tests, and calculated an unbiased estimator to benchmark their models in the generation of Python programs from docstrings. To estimate the functional correctness of a program, however, a set of unit tests needs to be manually constructed. This requires a significant effort that is often unfeasible for large amounts of generated code.

Similar to Evtikhiev et al. (2022), Takaichi et al. (2022), we assess the correlation between the automatic and human evaluation in the code generation, but we adopt a more exhaustive set of 23 output similarity metrics commonly used to assess NMT models. Moreover, to the best of our knowledge, this is the first work evaluating the automatic metrics in the generation of code for software security applications (in both low-level and high-level programming languages) and provides practical insights on what metric to use and when to assess the AI-based solutions generating offensive code. For the above-stated reasons, our work can be considered complementary to previous studies.

3. Offensive code generation

Yang et al. (2023) gathered question posts with tags containing terms related to software exploits from Stack Overflow and counted the number of answers to these posts as well as the number of answers accepted by the questioner. The final results show that less than half of the posts contain answers that were accepted by the questioners. This shows that, due to the specific domain of exploit code, writing exploit code manually is a time-consuming and difficult task. Therefore, the usage of NMT models to improve developers' productivity and support security auditors in the generation of PoC exploits is becoming an attracting solution (Liguori et al., 2021b; Yang, Chen, Zhou, & Yu, 2022; Yang et al., 2023). These models generate programming *code snippets* starting from NL *intents*.

To perform a rigorous evaluation of the output similarity metrics on the offensive code generated by the models, we follow the best practices in the field. Hence, we support the models with *data processing* operations. Data processing is an essential step to support the NMT models in the automatic code generation and refers to all the operations performed on the data used to train, validate and test the models. These operations strongly depend on the specific source and target languages to translate. 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. Fig. 1 summarizes the steps.

First, we use a corpus to train the NMT models. The *training data* is pre-processed before being used to feed the model. The pre-processing starts with the *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, we use a *tokenizer* to break the intents into chunks of text containing space-separated words (i.e., the *tokens*). To improve the performance of the machine translation (Li, Wang, Aw, Chng, & Li, 2018; Liguori et al., 2022; Modrzejewski, Exel, Buschbeck, Ha, & Waibel, 2020), 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 embedding*. The pre-processed data is used to feed the NMT model. Once the model is trained, we perform the code generation from NL. Therefore, when the model takes as inputs new intents from the *test data* (i.e., data of the corpora not used in the training phase), it generates the related code snippets based on the knowledge inferred during the training (*model's prediction*). As for the intents, also the code snippets predicted by the models are processed (*post-processing*) to improve the quality and readability of the

Table 2
Output similarity metrics used in previous work to estimate the code generated by NMT models.

Output similarity metric	Work	Code generation task
Compilation Accuracy	Clement, Drain, Timcheck, Svyatkovskiy, and Sundaresan (2020), Liguori, Al-Hossami, Orbinato, et al. (2021b)	NL→Python; NL→assembly
ROUGE	Svyatkovskiy, Deng, Fu, and Sundaresan (2020), Evtikhiev et al. (2022), Clement et al. (2020), Takaichi et al. (2022)	code completion; NL→Python
BLEU	Guo et al. (2022), Phan et al. (2021), Ahmad, Chakraborty, Ray, and Chang (2021), Wang, Wang, Joty, and Hoi (2021), Evtikhiev et al. (2022), Liguori, Al-Hossami, Orbinato, et al. (2021b), Clement et al. (2020), Takaichi et al. (2022)	NL→Java; C#→Java; NL→Python; NL→assembly
Exact Match	Chakraborty, Ahmed, Ding, Devanbu, and Ray (2022), Guo et al. (2022), Wang et al. (2022), Phan et al. (2021), Ahmad et al. (2021), Wang et al. (2021), Liguori, Al-Hossami, Orbinato, et al. (2021b)	NL→Java; NL→assembly; NL→Python
METEOR	Evtikhiev et al. (2022), Takaichi et al. (2022)	NL→Python
Edit Distance	Guo et al. (2022), Svyatkovskiy et al. (2020), Takaichi et al. (2022)	code completion; NL→Python

code. First, the dictionary of standardizable tokens is used in the *de-standardization* process to replace all the “*var#*” with the corresponding values, names, and parameters.

Finally, the code snippets generated during the model's prediction are evaluated to assess the quality of the code generation task. The evaluation can be performed through output similarity metrics or manual analysis (human evaluation). The former estimates the quality of the prediction by comparing the model's predictions with the ground truth reference in the test data, the latter, instead, assesses if the output predicted by the model is the correct translation of the NL intent into the generated code snippet.

4. Code generation metrics

4.1. Output similarity metrics

Given the huge amount of data to scrutinize, which makes the human evaluation time-consuming, the most practical and common solution to assess the performance of the NMT models is to use metrics that estimate the similarity between the code generated by NMT models and a *ground-truth* (i.e., code snippets used as references for the evaluation). Table 2 presents the most commonly used metrics by previous work to assess the quality of code generated by the NMT models across multiple code-related tasks, including code generation (i.e., natural language to code), code translation (i.e., programming language to different programming language), and code completion (i.e., programming language to the same programming language). In the following, we describe these metrics:

■ **Compilation Accuracy (CA)**. It indicates whether each code snippet produced by the model is compilable according to the syntax rules of the target language. CA value is either 1, when the snippet's syntax is correct, or 0 otherwise.

■ **Recall-Oriented Understudy for Gisting Evaluation (ROUGE)** (Lin, 2004). It measures the matching n-grams (i.e., the adjacent sequence of n items, such as syllables, letters, words, etc.) between the output predicted by the model and the ground truth reference, where n stands for the number of the n-gram. The number of matching n-grams is then divided by the total number of n-grams in the reference (recall, *ROUGE R*), or by the total number of n-grams in the model's prediction (precision, *ROUGE P*). The harmonic mean of precision and recall defines F_1 Score (*ROUGE* F_1).

■ **ROUGE-L**. It is a variant of the ROUGE metric commonly used to assess code generation based on the longest common subsequence (LCS) between the model's output and the reference, i.e. the longest sequence of words (not necessarily consecutive, but still in order) that is shared between both. ROUGE-L recall, precision, and F1-score can be computed by replacing each n-gram match with the LCS. The ROUGE metrics range between 0 (perfect mismatch) and 1 (perfect matching).

■ **Bilingual Evaluation Understudy (BLEU) score** (Papineni, Roukos, Ward, & Zhu, 2002). It measures the degree of n-gram overlapping between the string of each code snippet produced by the model and the reference, for values of n usually ranging between 1 and 4 (Han, 2016; Munkova, Hajek, Munk, & Skalka, 2020). This metric also takes into account a *brevity penalty* to penalize predictions shorter than the references. BLEU value ranges between 0 and 1, with higher scores corresponding to a better quality of the prediction.

■ **Exact Match accuracy (EM)**. It indicates whether each code snippet produced by the model perfectly matches the reference. EM value is 1 when there is an exact match, 0 otherwise.

■ **METEOR** (Lavie & Agarwal, 2007). It measures the *alignment* between each code snippet produced by the model and the reference. The alignment is defined as a mapping between unigrams (i.e., 1-gram), such that every unigram in each string maps to zero or one unigram in the other string, and no unigrams in the same string. METEOR value ranges between 0 and 1, with higher scores corresponding to greater alignment between strings.

■ **Edit Distance (ED)**. 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.

4.2. Motivating examples

Output similarity metrics cannot properly assess whether two pieces of code are different but semantically equivalent, i.e., they provide the same output and/or effects although they use different operations (e.g., `jz label` and `je label` are different assembly instructions performing the same conditional jump). For this reason, *human evaluation* is considered the golden standard for assessing the quality of the code generated by the models. Through manual inspection of the model's predictions, human evaluation allows assessing the deeper linguistic features of the code (Han, Smeaton, & Jones, 2021), such as the *code semantics*, i.e., *what the code actually does*. Therefore, for every code snippet generated by the model, we manually assess the **Semantic Correctness (SC)** metric, which indicates whether the output is the exact translation of the NL intent into the target programming language. This evaluation does not take into account the ground truth reference, but only the code predicted by the model and NL intent. *SC* value is either 1 when the generated snippet is the (semantically) correct translation of the intent, and 0 otherwise.

Unlike regular code generation tasks that focus on logically complex functional code fragments, high-level offensive code often contains a large number of low-level arithmetic, logic operations, and bit-level slices to hide plain text attacks from antivirus and intrusion detection systems.

As a simple example, consider the intent "*compare string s1 with string s2*", which translates to the Python snippet:

```
if s1 == s2:
```

A semantically equivalent implementation of this string comparison is the code:

```
if s1.__eq__(s2):
```

Despite the model's prediction being semantically correct ($SC = 1$), output similarity metrics are not able to grasp the equivalence between the two snippets since they base their calculation on character and/or token similarity. Therefore, this translation results in low automatic scores, including *ROUGE-L* (P: 0.66 R: 0.5, F_1: 0.57) and *edit distance* (0.47).

The opposite occurs with the intent "*check if count modulo 2 is different from zero*", which translates to the Python snippet:

```
if count % 2 != 0:
```

If the model generates the snippet:

```
if count % 2 == 0:
```

then prediction and reference differ by a single character, yet the code accomplishes the opposite task. Automatic metrics fail to account for situations like this. For instance, the *edit distance* between these two pieces of code is 0.94, while the *ROUGE-L* (P, R, F_1) score is 0.83, which are considered high values. Differently, a human evaluator would appropriately classify this snippet as semantically incorrect (i.e., $SC = 0$), since it does not perform the intended check.

Furthermore, attackers leverage low-level programming languages, such as assembly, to perform surgically crafted exploitation of the system's low-level mechanisms, including heap metadata and stack return addresses, that are not accessible through high-level programming languages. An example of an operation involving CPU registers is the intent "*transfer EAX contents into EDX register*", which translates to the assembly snippet:

```
mov EDX, EAX
```

An alternative method to copy the contents of a register into another is by pushing and popping its value onto the stack. Therefore, a semantically equivalent implementation of this copy is the code:

```
push EAX
pop EDX
```

This translation is semantically correct when assessed by a human evaluator ($SC = 1$), yet it results in low scores for output similarity metrics such as *ROUGE-L* (F_1: 0.25), *BLEU-4* (0.11), *edit distance* (0.31) and *METEOR* (0.24).

Contrarily, there are situations in which the difference of a single character implies the use of a different register and, therefore, the implementation of a similar yet not equivalent operation. Indeed, consider the code description "*clear the EDX register and move 5 in the lowest byte of the register*", which can be implemented through the assembly snippet:

```
xor EDX, EDX
mov DL, 5
```

Table 3
Datasets statistics.

Statistic	Assembly dataset	Python dataset
Dataset size	3,715	15,540
Unique Snippets	2,542	14,034
Unique Intents	3,689	15,421
Unique tokens (Snippets)	1,657	9,511
Unique tokens (Intents)	1,924	10,605
Avg. tokens per Snippet	4.75	11.90
Avg. tokens per Intent	9.53	14.90

If the model generates the following code: then the semantic correctness score should be zero ($SC = 0$) since the lowest byte of EDX is stored in the DL register, while BL contains the lowest byte of EBX. However, these two snippets are textually similar, hence resulting in high scores for *edit distance* (0.96) and *ROUGE-L* F_1 (0.86).

```
xor EDX, EDX
mov BL, 5
```

5. Experimental setup

5.1. NMT models

To perform the code generation task, we consider two standard architectures: Seq2Seq, and CodeBERT.

■ **Seq2Seq** is a model that maps an input of sequence to an output of sequence. Similar to the encoder–decoder architecture with attention mechanism (Bahdanau, Cho, & Bengio, 2015), we use a bi-directional LSTM as the encoder to transform an embedded intent sequence into a vector of hidden states with equal length. We implement the Seq2Seq model using *xnmt* (Neubig et al., 2018). We use an Adam optimizer (Kingma & Ba, 2015) with $\beta_1 = 0.9$ and $\beta_2 = 0.999$, while the learning rate α is set to 0.001. We set all the remaining hyper-parameters in a basic configuration: layer dimension = 512, layers = 1, epochs = 200, beam size = 5.

■ **CodeBERT** (Feng et al., 2020) is a large multi-layer bidirectional Transformer architecture (Vaswani et al., 2017) pre-trained on millions of lines of code across six different programming languages. Our implementation uses an encoder–decoder framework where the encoder is initialized to the pre-trained CodeBERT weights, and the decoder is a transformer decoder, composed of 6 stacked layers. The encoder follows the RoBERTa architecture (Liu et al., 2019), with 12 attention heads, hidden layer dimension of 768, 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.

During data pre-processing, we tokenize the NL intents using the *nltk word tokenizer* (Bird, 2006) and code snippets using the Python *tokenize* package (Python, 2023). We use *spaCy*, an open-source, NL processing library written in Python and Cython (spaCy, 2023), to implement the named entity tagger for the standardization of the NL intents.

5.2. Datasets

We feed the NMT models with a large corpus developed by Liguori, Al-Hossami, Orbinato, et al. (2021b) and used for security code generation and summarization (Yang et al., 2022, 2023). The dataset fits in the context of the software security as it contains code snippets (alongside with their descriptions in English) of code used to develop and execute *shellcode* programs, i.e., piece of code used as the payload in the exploitation of a software vulnerability. Examples of complex shellcode attacks include fork bombs, denial of service, bind shells, etc.

The dataset enables different analyses as it contains different programming languages. Indeed, the corpus consists of two parts: (i) a *Python dataset*, which contains Python code used by exploits to encode

Table 4
Examples of samples of the assembly and Python datasets.

Dataset	NL Intent	Code Snippet
Assembly	*Copy the ASCII string "/bin//sh" into the EBX register*	`push 0 × 68732f2f \n push 0 × 6e69622f \n mov ebx, esp`
Python	*val2 is the result of the bitwise xor between the integer base 16 of the element i of chunk encoded to hex and xor_byte*	`val2 = int( chunk[i]. encode('hex'), 16 ) ^ xor_byte`

shellcodes (i.e., to obfuscate the execution of the shellcodes from antivirus and intrusion detection systems), and (ii) an *assembly dataset*, which includes shellcodes and decoders to revert the encoding. The authors collected exploits from publicly available databases, public repositories (e.g., GitHub), and programming guidelines. A sample in the dataset consists of a snippet of code from these exploits and their corresponding description in the English language, as shown in Table 4. The assembly dataset includes 783 lines (~21% of the dataset) of *multi-line intents*, i.e., intents that generate multiple lines of assembly code (between 2 and 5), separated by the newline character \n (as the example in Table 4).

Table 3 summarizes the statistics of both datasets, including the size (i.e., the unique pairs of intents-snippets), the unique lines of code snippets, the unique lines of NL intents, the unique number of tokens (i.e., words), and the average number of tokens per snippet and intent. The statistics highlight the difference between the datasets. Unsurprisingly, the difference in terms of tokens per snippet (~ 12 for Python, ~ 5 for assembly) and per intent (~ 15 for Python, ~ 10 for assembly) makes the code generation task more difficult and challenging for the Python dataset. Indeed, a low-level programming language contains a more limited amount of instructions and operations than a high-level language such as Python. This explains why the size of the Python data is larger than the assembly data, i.e., to train the models to generate more complex programming code. These differences allow us to evaluate the performance of the NMT models in offensive code generation with different complexity.

5.3. Implementation of the metrics

To automatically assess the quality of the generated code snippets in comparison with the ground truth, we relied on open-source tools and Python packages. To compute the *compilation accuracy*, we used the *Netwide Assembler* (NASM) assembler (NASM, 2022) for the assembly code and the `py_compile` (pycompile, 2023) compiler for Python snippets. As for the *ROUGE* and *ROUGE-L* metrics, we computed the output similarity scores using the Python package `rouge` (rouge, 2021) for both languages. We implemented *BLEU* score computation employing the `bleu_score` module contained in the open-source Python suite *Natural Language Toolkit* (NLTK) (NLTK, 2023). For the *edit distance* we used `pylcs` (pylcs, 2023). To calculate the *METEOR* metric, we relied on the Python library `evaluate` by HuggingFace (evaluate, 2022). Finally, for *exact match accuracy*, we used a simple Python string comparison.

5.4. Human evaluation

To assess the semantic correctness of the predictions, we manually analyze every code snippet generated by the models to inspect if it is the correct translation of the NL intent, i.e., if the code snippet generated by the model performs what the English comment states. This analysis cannot be performed automatically (e.g., by comparing the predictions with ground-truth references) since an English intent can be translated into different but equivalent code snippets (as shown in Section 4.2). This analysis can be subjective, as different reviewers may have different interpretations of the code and its intended functionality, depending on the expertise and experience of the reviewer. This can lead to inconsistent assessments of code correctness. Moreover, manual analysis is prone to human error, as reviewers may miss subtle errors or inconsistencies in the code, or may introduce errors and biases into their assessments due to factors such as fatigue, distractions, or personal opinions. Therefore, to reduce the possibility of errors and inconsistency, multiple authors performed the manual analysis independently and discussed the cases of discrepancy, obtaining a consensus for the semantic correctness of the predictions.

6. Experimental results

Experiments aim to assess what are the output similarity metrics that are closer to the human evaluation in the generation of assembly and Python code. To perform the experiments, we split the dataset into training (the set of examples used to fit the parameters), validation (the set used to tune the hyperparameters of the models), and test (the set used for the evaluation of the models) sets using a common 80%/10%/10% ratio (Kim & MacKinnon, 2018; Liguori et al., 2021a; Mashhadi & Hemmati, 2021).

6.1. Quantitative analysis

First, we investigated whether the output similarity metrics provide similar results on the same data. To this aim, we assessed the performance of the NMT models on the assembly and Python test data. Fig. 2 shows the statistics, in terms of boxplots, of the output similarity metrics. The height of both boxplots highlights that the metrics provide very different results. Indeed, we found that for the assembly data, the min value is 17%, the median is 60% (the average is 56%, the standard deviation is 21%), and the max value is 87%. For the Python data, the min value is 5%, the median is 45% (the average is 46%, the standard deviation is 24%) and the max is 91%. Therefore, different metrics used for code generation provide very different values, leading to a wrong assessment of the performance of the models.

The figure also shows the SC values (**X** markers), which are 53% and 59% for assembly and Python data, respectively (i.e., the data is pretty balanced). Ideally, output similarity metrics should provide a value closer to the one of semantic correctness. We found that the median value of the output similarity metrics is 7% higher than the human evaluation for assembly data, while is 14% lower than the human evaluation for Python data, i.e., the metrics overestimate the performance for the assembly and underestimate the performance for the Python data. We attribute this behavior to the different structures of the assembly and Python code. Indeed, the limited set of instructions of a low-level language makes the code snippets similar even when they are semantically equivalent (e.g., `jz label` and `jnz label` are similar but semantically different instructions). This is not the case for the Python dataset, which has a higher complexity in terms of different instructions, length of the code, etc. Therefore, it is more likely to write equivalent code with very different instructions.

We then performed an in-depth analysis by comparing each output similarity metric with the human evaluation, i.e., the SC. Specifically, for each dataset, we performed three different analyses, depending on the code snippets included in the test set for the evaluation: (i) the *whole test set*, i.e, we considered all the code snippets in the test data, (ii) only the *correct predictions*, i.e., we limited the analysis to the code snippets considered correct according to the human evaluation, and (iii) only the *wrong predictions*, i.e., the analysis entailed only the code snippets considered as semantically incorrect by the human evaluation. The analysis of the correct predictions and wrong predictions enables different considerations, e.g., to infer what are the most suitable metrics to use when the models provide very accurate predictions or when they deal with very challenging code generation tasks. To this aim, we

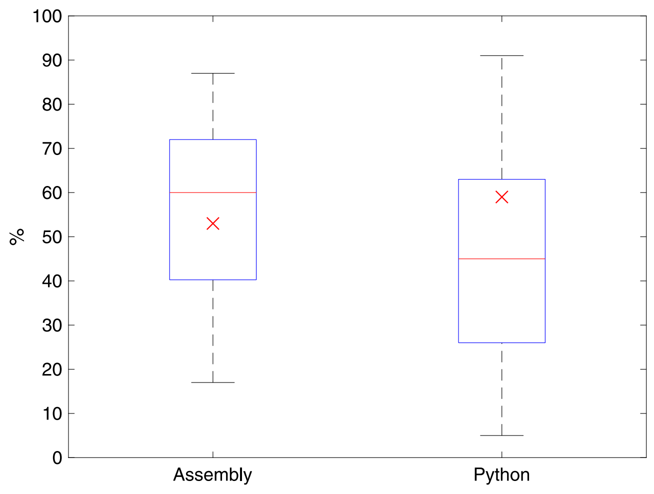

Fig. 2. Distribution of the output similarity metrics on the assembly and Python data. The marker X indicates the semantic correctness value.

Table 5
Average values and offset of the output similarity metrics on the **assembly** dataset. Best performance (lower offset) is blue, while the worst performance (higher offset) is red.

Metric	*Whole*		*Correct*		*Wrong*	
SC	0.53	-	1.00	-	0.00	-
CA	0.87	0.34	1.00	0.00	0.72	0.72
ROUGE-1 P	0.75	0.22	0.93	0.07	0.55	0.55
ROUGE-1 R	0.71	0.18	0.92	0.08	0.47	0.47
ROUGE-1 F_1	0.72	0.19	0.92	0.08	0.50	0.50
ROUGE-2 P	0.55	0.02	0.75	0.25	0.32	0.32
ROUGE-2 R	0.52	0.01	0.74	0.26	0.28	0.28
ROUGE-2 F_1	0.53	0.00	0.74	0.26	0.28	0.28
ROUGE-3 P	0.40	0.12	0.60	0.40	0.19	0.19
ROUGE-3 R	0.38	0.14	0.59	0.41	0.16	0.16
ROUGE-3 F_1	0.39	0.14	0.59	0.41	0.16	0.16
ROUGE-4 P	0.18	0.35	0.25	0.75	0.10	0.10
ROUGE-4 R	0.17	0.36	0.25	0.75	0.09	0.09
ROUGE-4 F_1	0.17	0.36	0.25	0.75	0.09	0.09
ROUGE-L P	0.75	0.22	0.93	0.07	0.54	0.54
ROUGE-L R	0.71	0.18	0.92	0.08	0.47	0.47
ROUGE-L F_1	0.72	0.19	0.92	0.08	0.49	0.49
BLEU-1	0.69	0.16	0.89	0.11	0.53	0.53
BLEU-2	0.63	0.10	0.86	0.14	0.45	0.45
BLEU-3	0.60	0.07	0.84	0.16	0.41	0.41
BLEU-4	0.57	0.04	0.81	0.19	0.39	0.39
EM	0.41	0.11	0.79	0.21	0.00	0.00
METEOR	0.72	0.19	0.90	0.10	0.52	0.52
ED	0.79	0.27	0.96	0.04	0.60	0.60
Average	0.56	0.17	0.75	0.25	0.36	0.36

Table 6
Average values and offset of the output similarity metrics on the Python dataset. Best performance (lower offset) is blue, while the worst performance (higher offset) is red.

Metric	*Whole*		*Correct*		*Wrong*	
SC	0.59	-	1.00	-	0.00	-
CA	0.91	0.32	1.00	0.00	0.79	0.79
ROUGE-1 P	0.63	0.04	0.75	0.25	0.46	0.46
ROUGE-1 R	0.63	0.04	0.74	0.26	0.47	0.47
ROUGE-1 F_1	0.63	0.04	0.74	0.26	0.46	0.46
ROUGE-2 P	0.45	0.14	0.59	0.41	0.23	0.23
ROUGE-2 R	0.45	0.14	0.58	0.42	0.24	0.24
ROUGE-2 F_1	0.44	0.15	0.58	0.42	0.23	0.23
ROUGE-3 P	0.26	0.33	0.38	0.62	0.07	0.07
ROUGE-3 R	0.26	0.33	0.38	0.62	0.07	0.07
ROUGE-3 F_1	0.26	0.33	0.38	0.62	0.07	0.07
ROUGE-4 P	0.05	0.54	0.07	0.93	0.02	0.02
ROUGE-4 R	0.05	0.54	0.07	0.93	0.02	0.02
ROUGE-4 F_1	0.05	0.54	0.07	0.93	0.02	0.02
ROUGE-L P	0.63	0.04	0.75	0.25	0.46	0.46
ROUGE-L R	0.63	0.04	0.74	0.26	0.47	0.47
ROUGE-L F_1	0.63	0.04	0.74	0.26	0.46	0.46
BLEU-1	0.58	0.01	0.69	0.31	0.44	0.44
BLEU-2	0.48	0.11	0.61	0.39	0.32	0.32
BLEU-3	0.37	0.22	0.50	0.50	0.19	0.19
BLEU-4	0.26	0.33	0.37	0.63	0.13	0.13
EM	0.27	0.32	0.43	0.57	0.00	0.00
METEOR	0.74	0.16	0.84	0.16	0.60	0.60
ED	0.81	0.22	0.91	0.09	0.66	0.66
Average	0.46	0.22	0.56	0.44	0.30	0.30

computed an *offset value*, i.e., the distance between the optimal value represented by the human evaluation (i.e., the semantic correctness) and the estimate provided by the output similarity metrics: the lower the offset, the closer the automatic metric is to the human evaluation.

Table 5 and Table 6 show the results, including the average values obtained by the metrics and the offset with the *SC*, for the assembly and Python datasets, respectively. In this analysis, we found that the metric closer to the semantic correctness is *ROUGE-2* (offset is 0%, i.e., they provide the same value) for the assembly language, while *BLEU-1* is closer to the human evaluation on the Python code snippets (offset = 1%). For both assembly and Python code snippets, the metric less similar to the *SC* is *ROUGE-4* (offset = 36% and 54%, respectively). Therefore, metrics based on n-grams provide an evaluation close to the semantic correctness when n is equal to 1 or 2, while their ability in the evaluation gets worse when $n \geq 3$. We attribute this behavior to the brevity (in terms of tokens) of programming language snippets, especially in the context of the generation of software exploits, which require low-level instructions and binary-level-data processing. In this case, since code frequently includes operations with few tokens (e.g., a `jmp label` instruction, a Python increment with += operand), output similarity metrics using a higher number of n-grams underestimate the quality of the predictions (e.g., *ROUGE-4* for assembly and Python, *BLEU-4* for Python).

When we limit the evaluation to the correct predictions, i.e., we evaluate only the code snippets considered semantically correct (SC = 1), then the best metric is the *compilation accuracy*, regardless of the programming language. Indeed, when the snippet is semantically correct then it is also syntactically correct, i.e., it is also compilable. Therefore, the offset is 0% in this case. Besides the *CA*, the *edit distance* is a valuable option in this specific case (offsets are 4% and 9% for assembly and Python datasets) since the number of operations required to make the predictions equal to the reference is limited when the predictions are correct. The worst metric, instead, is again *ROUGE-4* (offset equal to 75% and 93% for assembly and Python, respectively), showing that it is not able to properly assess the correctly generated code snippets.

The analysis on the wrong predictions, i.e., on the code snippets not semantically correct ($SC = 0$), highlights that the *exact match* accuracy is a good evaluator for both datasets as the models' predictions, in this case, never match the ground truth references (and, therefore, the values are 0 for every code snippet). The *compilation accuracy*, which showed the best performance in the previous case study, provides the worst performance on the wrong predictions since a semantically incorrect snippet can be syntactically correct (i.e., compilable). Indeed, we found that 72% for the assembly dataset and 79% for the Python dataset of the semantically incorrect snippets are syntactically correct.

Finally, we found that the average values of the offsets over all the code similarity metrics are pretty similar for both assembly and Python datasets for the whole test set (17% vs 22%) and the wrong predictions (36% vs 30%). For the correct predictions, instead, the differences between assembly and Python are more exacerbated (25% vs 44%) due to the ability of the high-level language to write semantically equivalent code with different instructions.

Table 7
Correlation Analysis between output similarity metrics and human evaluation on the **assembly** dataset. Best performance is blue, while the worst performance is red.

Output Similarity Metric	*Seq2Seq*		*CodeBERT*	
	Pearson's r	*Kendall's τ*	*Pearson's r*	*Kendall's τ*
CA	0.35	0.35	0.47	0.47
ROUGE-1 P	0.63	0.63	0.60	0.62
ROUGE-1 R	0.70	0.67	0.65	0.66
ROUGE-1 F_1	0.70	0.67	0.65	0.66
ROUGE-2 P	0.49	0.47	0.53	0.51
ROUGE-2 R	0.54	0.49	0.55	0.52
ROUGE-2 F_1	0.54	0.50	0.55	0.52
ROUGE-3 P	0.46	0.42	0.47	0.44
ROUGE-3 R	0.51	0.44	0.47	0.44
ROUGE-3 F_1	0.50	0.44	0.47	0.44
ROUGE-4 P	0.19	0.12	0.22	0.17
ROUGE-4 R	0.23	0.13	0.23	0.18
ROUGE-4 F_1	0.22	0.13	0.23	0.18
ROUGE-L P	0.63	0.62	0.62	0.64
ROUGE-L R	0.70	0.67	0.67	0.67
ROUGE-L F_1	0.69	0.67	0.66	0.67
BLEU-1	0.72	0.67	0.66	0.65
BLEU-2	0.56	0.51	0.56	0.53
BLEU-3	0.50	0.49	0.48	0.48
BLEU-4	0.22	0.36	0.23	0.32
EM	0.81	0.81	0.78	0.78
METEOR	0.68	0.61	0.61	0.57
ED	0.72	0.71	0.68	0.70
Average	0.53	0.50	0.52	0.51

Table 8
Correlation Analysis between output similarity metrics and human evaluation on the **Python** dataset. Best performance is blue, while the worst performance is red.

Output Similarity Metric	*Seq2Seq*		*CodeBERT*	
	Pearson's r	*Kendall's τ*	*Pearson's r*	*Kendall's τ*
CA	0.36	0.36	0.38	0.38
ROUGE-1 P	0.49	0.41	0.50	0.46
ROUGE-1 R	0.47	0.38	0.45	0.41
ROUGE-1 F_1	0.49	0.40	0.48	0.43
ROUGE-2 P	0.46	0.40	0.48	0.42
ROUGE-2 R	0.44	0.37	0.44	0.39
ROUGE-2 F_1	0.46	0.38	0.46	0.40
ROUGE-3 P	0.31	0.23	0.43	0.40
ROUGE-3 R	0.30	0.23	0.42	0.39
ROUGE-3 F_1	0.31	0.23	0.43	0.40
ROUGE-4 P	0.15	0.12	0.10	0.04
ROUGE-4 R	0.14	0.12	0.09	0.04
ROUGE-4 F_1	0.15	0.12	0.10	0.04
ROUGE-L P	0.50	0.42	0.51	0.46
ROUGE-L R	0.48	0.38	0.45	0.41
ROUGE-L F_1	0.50	0.41	0.49	0.44
BLEU-1	0.43	0.36	0.48	0.42
BLEU-2	0.45	0.37	0.47	0.40
BLEU-3	0.33	0.29	0.44	0.35
BLEU-4	0.13	0.20	0.09	0.13
EM	0.42	0.42	0.54	0.54
METEOR	0.38	0.32	0.55	0.53
ED	0.57	0.49	0.57	0.56
Average	0.38	0.32	0.41	0.37

6.2. Correlation analysis

We further assessed the ability of the output similarity metrics in the evaluation of the code generation task. Different from the previous quantitative analysis, in which we compared the average values provided by the output similarity metrics with the average semantic correctness over all the test sets, we performed an in-depth statistical analysis by computing the correlation of the output similarity metrics with the human evaluation of all the code snippets of the assembly and Python test sets (i.e., we considered the values of the metrics on the single predictions).

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 and is defined as the covariance of the two variables divided by the product of their respective standard deviations (Pearson, 1895). Moreover, to assess the relationship between the metrics and the semantic correctness, we also computed the *Kendall* correlation coefficient τ, which measures the dependence between two random variables based on the ranks of sampled observations of the variable (Kendall, 1938).

The correlation coefficients are unit-free values between −1 and 1, which represent *perfect* correlations, *negative*, and *positive*, respectively. Positive values indicate a positive correlation, i.e., the values of both variables tend to increase together, while negative values indicate a negative correlation, i.e., the values of one variable tend to increase when the values of the other variable decrease. Therefore, a high value of the coefficient indicates that the output similarity metric is strongly associated with human evaluation. On the contrary, a small value indicates that the automatic metric is poorly associated with human evaluation.

Table 7 and Table 8 show Pearson's r and Kendall's τ correlation coefficients between the automatic metrics and the semantic correctness on the assembly and Python datasets, respectively, for both the Seq2Seq and CodeBERT models. For the assembly dataset, we found that the *exact match* has the highest correlation coefficients for both models (r and τ are 0.81 for Seq2Seq and 0.78 for CodeBERT). We attribute this result to the nature of the assembly language, which has a fixed structure and provides a more limited set of instructions to express an operation if compared to a high-level language. Therefore, the *exact match*, which provides 1 only when the prediction is equal to the reference, results to be the most suitable metric for this case study.

For the Python dataset, the *edit distance* is the most correlated metric with the human evaluation ($r = 0.57$ and $\tau = 0.49$ for Seq2Seq, while $r = 0.57$ and $\tau = 0.56$ for CodeBERT), although the coefficients are lower than the best values of the assembly case study. As above mentioned, Python code for software exploits requires a considerable amount of binary-level-data processing and concise instructions. If a single character in a hexadecimal value is not correct, then, for n-gram-based metrics such as ROUGE, the whole token is different from the reference, resulting in low scores. Differently, character-based metrics such as the *edit distance* account for these slight deviations and result in a high score.

Again, *ROUGE-4* metrics provide the worst results for both datasets. For the assembly language, the correlation coefficients are $r \leq 0.23$ and $\tau \leq 0.13$ for Seq2Seq, while $r \leq 0.23$ and $\tau \leq 0.18$ for CodeBERT. For the Python code snippets, the correlation coefficients are even lower ($r \leq 0.15$, $\tau \leq 0.12$). This result confirms that n-gram-based metrics are lowly correlated to the human evaluation when n is high.

The comparison of the correlation coefficients on the assembly and Python datasets highlights that the code similarity metrics are more correlated to the assembly case study ($r \geq 0.52$, $\tau \geq 0.50$) than the Python one ($r \leq 0.41$, $\tau \leq 0.37$) due to the increasing difficulty of the code similarity metrics to assess the generation of more complex offensive code.

7. Discussion

Our analysis highlights that n-gram-based metrics like ROUGE and BLEU, which are commonly used to assess code generation tasks, are not the best choice to evaluate offensive code. Indeed, the exact match and the edit distance are the most correlated to human evaluation for the assembly and Python code, respectively. Assembly instructions are typically characterized by a fixed and concise structure in the form `OP DST, [SRC]`, which includes an *opcode*, the destination of the operation, and (optionally) the source. Therefore, increasing the n value for these metrics (i.e., 3–4) leads to lower scores even for semantically

Table 9
Examples that show how different output similarity metrics work for different programming languages. Red refers to incorrect predictions.

Dataset	Ground Truth	Predicted Code	*SC*	*ROUGE-4* (F_1)	*ED*	*EM*
Assembly	`add EAX, EBX`	`add EAX, EBX`	1.0	0.0	1.0	1.0
	`xor ECX, ECX \n mul ECX`	`xor ECX, ECX \n mul EBX`	0.0	0.66	0.95	0.0
	`jmp decode`	`jmp decode`	1.0	0.0	1.0	1.0
Python	`break`	`sys.exit()`	0.0	0.0	0.1	0.0
	`for byte in encoder:`	`for bytes in encoder:`	1.0	0.0	0.95	0.0
	`encoded = ‘‘\\x’’`	`encoded = ‘\\x’`	1.0	0.0	0.87	0.0

correct snippets. The same goes for Python, which, although being a high-level and structurally complex language, when used for offensive purposes is characterized by concise snippets that handle numerical values and logical operations. Table 9 shows a set of cherry-picked examples in both assembly and Python languages and their output similarity metric scores. We report the score for ROUGE-4, which is the least correlated metric with human evaluation for both languages, and edit distance and exact match, which are the most correlated for Python and assembly, as shown in Section 6.2. The first and third rows for the assembly dataset present situations in which the predicted code matches the ground truth ($EM = 1$, $ED = 1$), yet the snippet is too short to be correctly assessed by ROUGE-4. The second row shows an example in which the reference and prediction are almost identical, but not semantically equivalent, therefore both ROUGE-4 and edit distance give a wrongly high score, while the exact match is correct. Also for Python, the code does not contain 4-grams ($ROUGE-4 = 0$). The table shows two correct examples in which ground truth and predicted code differ by one or two characters but are equivalent, therefore the most accurate metric is the edit distance. The first row presents an example in which the prediction is completely inaccurate and all three similarity metrics correctly provide a low score.

Therefore, an important takeaway of our experiments is that the choice of metric depends on the complexity of the model-generated code. In the case of code with a more fixed structure and more limited set of instructions (as in the case of assembly), and, therefore, with less possibility of expressing different but semantically equivalent snippets, then the exact match is the best candidate. When the complexity of the generated code increases (as in the case of Python), then the edit distance metric is an appropriate choice for evaluating the code. Moreover, when tasks are extremely difficult, i.e., when models fail to generate code (e.g., this is the common case of corpora not being large enough to train the models), the exact match is again the best metric. In the opposite case, i.e., when the models are very accurate in the code generation, then a metric that evaluates syntactic correctness (e.g., compilation accuracy) is recommended.

Finally, we compared the results of our analysis with the results performed by previous studies in the generation of general (i.e., not offensive) Python code (Evtikhiev et al., 2022; Takaichi et al., 2022). Since there is no further existing assembly dataset for code generation, we limited this analysis to the Python code. ROUGE-L is found to be among the best-performing metrics for the assessment of the Python code on both the CoNaLa dataset (Yin, Deng, Chen, Vasilescu, & Neubig, 2018), a dataset of questions posted on Stack Overflow with the posted solutions in Python, and the Card2code Hearthstone (Ling et al., 2016), a dataset dedicated to generating classes that are descriptions of the cards used in the Hearthstone game. A correlation analysis on the ReCa dataset (Liu et al., 2020), instead, showed the METEOR is the best metric to assess syntactically incorrect Python code generated from NL requirements. Although these metrics act well also in the generation of offensive Python code, they do not result to have the highest correlation in our analysis. We attribute this result to the difference between generic code and offensive code. Indeed, unlike regular code generation tasks that focus on logically complex functional code fragments, high-level exploit code contains a large number of arithmetic and logic operations, and bit-level slices (as in symmetric key cryptography) to encode the plain exploits into new, functionally equivalent ones, but more difficult to block by modern antivirus and intrusion detection systems.

At the end of the day, despite output similarity metrics providing estimates close to the human evaluation, their ability to represent human assessment is highly affected by the specific code generation task, i.e., there is not a metric that is always suitable for the evaluation, regardless of data and complexity of the task. Therefore, given that automatic code generation is an area that is likely to continue to attract interest from academia and industry, we believe there is a need for a solution that can automatically evaluate the semantic correctness of code generated by ML models.

8. Threats to validity

AI-based code generators: We performed our experiments employing two state-of-the-art NMT solutions, a Seq2Seq model and a pre-trained model such as CodeBERT. We are aware that considering only two models can be a limitation to this evaluation, yet our choice was guided by the popularity and the availability of mature open-source implementation of these technologies. Seq2Seq is still largely used as a baseline model in this line of research and remains among the most used architectures for code generation. CodeBERT, on the other hand, represents the state-of-the-art for several code-related tasks, such as code search and code documentation generation, and many other software engineering tasks (Ahmed & Devanbu, 2022; Mashhadi & Hemmati, 2021; Yu, Yang, Chen, Liu, & Zhou, 2022; Zeng et al., 2022; Zhou, Han, & Lo, 2021), including generation of offensive code (Liguori et al., 2022; Yang et al., 2023). We acknowledge that there are emerging NMT models that are showing superior performance in different tasks, including code generation. However, the scope of this paper is not to improve the state-of-the-art performance in offensive code generation but to assess the ability of the metrics in estimating offensive code correctness. We believe that both Seq2Seq and CodeBERT fit well with the scope of the paper as they provide us with different and adequate numbers of code snippets to properly evaluate the metrics. Both assembly and Python code generated by the models are pretty balanced data (in terms of semantic correctness), which allows us to perform a fair evaluation of the metrics. Moreover, balanced data also enable the execution of different analyses, such as the analysis of whole wrong and whole correct data. Finally, we did not consider public AI code generators such as GitHub Copilot and OpenAI ChatGPT, since they impose restrictions on malicious uses (Check Point Blog, 2023). Moreover, both attackers and defenders need to avoid leaking their techniques and tactics to their counterparts ("*operations security*", OPSEC). Thus, we consider the case of an attacker or defender that builds her own AI code generator, thus circumventing usage policies of public AI code generators.

Dataset: This work addresses the specific problem of the automatic generation of software exploits, focusing on the translation of NL intents into offensive code snippets in assembly and Python programming languages. The datasets we used in our experiments fit perfectly with the scope of this work. Indeed, to the best of our knowledge, this is the only dataset used for code generation in the context of software security. Moreover, we aimed to address complex and longer programming tasks (e.g., Python and assembly for processing binary-level data). For these tasks, NMT is still far from generating long and complex programs from just a single high-level description. The assembly and Python datasets considered in this work provide natural language descriptions both at the block and statement levels that are closer to the descriptions needed for more complex programming tasks.

Data size: We acknowledge that the corpora used in our experiments may seem relatively small compared to other corpora available for different code-generation tasks. These corpora contain relatively-small programs that are described at a rather high-level (e.g., the JuICe dataset (Agashe, Iyer, & Zettlemoyer, 2019) includes programming assignments with only one statement to describe the entire program to be generated) or larger, potentially noisy, subsets of training examples obtained by mining the web (e.g., CoNaLa *mined* dataset contains thousands of training examples mined directly from StackOverflow (Yin et al., 2018)). The datasets used for our experiments, instead, are manually curated datasets containing high-quality and non-ambiguous descriptions of the code (not available in larger datasets for code generation) that help us to properly estimate whether the model's prediction is the correct translation of the NL intent. Nevertheless, to mitigate the bias, we leverage an existing pre-trained model such as CodeBERT to compensate for the need for big data. Finally, it is worth noticing that, when we limit the comparison to manually annotated datasets, the dataset used in our experiments is way larger than the size of the CoNaLa *annotated* dataset (Yin et al., 2018), which is the basis for state-of-the-art studies in NMT for Python code generation (Gemmell, Rossetto, & Dalton, 2020; Yin & Neubig, 2019).

Output Similarity Metrics: We performed our evaluation by using a comprehensive and not trivial number of metrics used by previous work to assess the code generated by the NMT models. Nevertheless, we are aware that the set of metrics considered in our study does not include all the available metrics in the literature. This is the case of the code-oriented metrics (i.e., metrics created ad-hoc for specific programming languages) proposed to overcome the shortcomings of *BLEU* for code generation (e.g., *CodeBLEU* (Ren et al., 2020) and *RUBY* (Tran, Tran, Nguyen, Nguyen, & Nguyen, 2019), which were designed to evaluate code written in Java and C#). However, these metrics rely on deeper program analysis (such as syntax and dataflow match), which requires that the code generated by the models is syntactically correct (i.e., compilable) and prevents the metrics from being language-agnostic. As matter of fact, there is no available implementation of such metrics for low-level programming languages such as assembly. Therefore, we focused on the output similarity metrics commonly applied for the code evaluation, which can be easily tuned and used, regardless of the code programming language.

9. Conclusion

In this work, we compared the results provided by the output similarity metrics with the human evaluation by assessing the performance of two different state-of-the-art models in the generation of offensive assembly and Python code snippets from natural language descriptions.

The results of our experiments provide actionable insights that can be used by future research to assess the NMT models in the software security field. Although we pointed out what metrics can properly assess the model's predictions in different case studies, there is still a gap to fill between the automatic and the human evaluation. Unfortunately, using a simple metric (however smart) is often not sufficient to provide accurate results. Therefore, to have significant improvements, we believe it is necessary to apply static and dynamic analysis techniques to automatically assess the semantic correctness of code generated by the models.

CRediT authorship contribution statement

Pietro Liguori: Conceptualization, Methodology, Validation, Formal analysis, Investigation, Data curation, Writing – original draft, Visualization. **Cristina Improta:** Conceptualization, Methodology, Software, Validation, Investigation, Data curation, Writing – original draft. **Roberto Natella:** Conceptualization, Resources, Writing – review & editing, Supervision. **Bojan Cukic:** Resources, Writing – review & editing, Supervision, Project administration. **Domenico Cotroneo:** Conceptualization, Resources, Writing – review & editing, Supervision, Project administration, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request

Acknowledgments

This work has been partially supported by the University of Naples Federico II in the frame of the Programme F.R.A., project OSTAGE, PROT: 34938_07_04_2021, UGOV: 000010-ALTRI_CdA_75_2021_FRA_LINEA_B_001_002, CUP: E55F21000340005.

References

Agashe, R., Iyer, S., & Zettlemoyer, L. (2019). JuICe: A large scale distantly supervised dataset for open domain context-based code generation. In *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing* (pp. 5436–5446).

Ahmad, W. U., Chakraborty, S., Ray, B., & Chang, K. (2021). Unified pre-training for program understanding and generation. In K. Toutanova, A. Rumshisky, L. Zettlemoyer, D. Hakkani-Tür, I. Beltagy, S. Bethard, & et al. (Eds.), *proceedings of the 2021 conference of the North American chapter of the association for computational linguistics: human language technologies* (pp. 2655–2668). Association for Computational Linguistics, http://dx.doi.org/10.18653/v1/2021.naacl-main.211.

Ahmed, T., & Devanbu, P. T. (2022). Multilingual training for software engineering. In *44th IEEE/ACM 44th international conference on software engineering* (pp. 1443–1455). ACM, http://dx.doi.org/10.1145/3510003.3510049.

Akinobu, Y., Kajiura, T., Obara, M., & Kuramitsu, K. (2022). NMT-based code generation for coding assistance with natural language. *Journal of Information Processing*, *30*, 443–450.

Akinobu, Y., Obara, M., Kajiura, T., Takano, S., Tamura, M., Tomioka, M., et al. (2021). Is neural machine translation approach accurate enough for coding assistance? In *Proceedings of the 1st ACM SIGPLAN international workshop on beyond code: no code* (pp. 23–28).

Arce, I. (2004). The shellcode generation. *IEEE Security & Privacy*, *2*(5), 72–76.

Bahdanau, D., Cho, K., & Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. In Y. Bengio, & Y. LeCun (Eds.), *3rd International conference on learning representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, conference track proceedings*. URL http://arxiv.org/abs/1409.0473.

Bao, T., Burket, J., Woo, M., Turner, R., & Brumley, D. (2014). {Byteweight}: Learning to recognize functions in binary code. In *23rd USENIX security symposium* (pp. 845–860).

Bird, S. (2006). NLTK: the natural language toolkit. In *Proceedings of the COLING/ACL 2006 interactive presentation sessions* (pp. 69–72).

Chakraborty, S., Ahmed, T., Ding, Y., Devanbu, P. T., & Ray, B. (2022). NatGen: generative pre-training by "naturalizing" source code. In A. Roychoudhury, C. Cadar, & M. Kim (Eds.), *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering* (pp. 18–30). ACM, http://dx.doi.org/10.1145/3540250.3549162.

Check Point Blog (2023). Russian hackers attempt to bypass OpenAI's restrictions for malicious use of ChatGPT. URL https://blog.checkpoint.com/2023/01/13/russian-hackers-attempt-to-bypass-openais-restrictions-for-malicious-use-of-chatgpt/.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Ponde, H., Kaplan, J., et al. (2021). Evaluating large language models trained on code. ArXiv, abs/2107.03374.

Clement, C., Drain, D., Timcheck, J., Svyatkovskiy, A., & Sundaresan, N. (2020). PyMT5: multi-mode translation of natural language and python code with transformers. In *Proceedings of the 2020 conference on empirical methods in natural language processing* (pp. 9052–9065). Online: Association for Computational Linguistics, http://dx.doi.org/10.18653/v1/2020.emnlp-main.728, URL https://aclanthology.org/2020.emnlp-main.728.

Ding, S. H., Fung, B. C., & Charland, P. (2019). Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE symposium on security and privacy* (pp. 472–489). IEEE.

evaluate (2022). Python library evaluate. URL https://pypi.org/project/evaluate/.

Evtikhiev, M., Bogomolov, E., Sokolov, Y., & Bryksin, T. (2022). Out of the BLEU: how should we assess quality of the code generation models?, CoRR, abs/2208.03133. http://dx.doi.org/10.48550/arXiv.2208.03133.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., et al. (2020). CodeBERT: A pre-trained model for programming and natural languages. In *Findings of ACL: vol. EMNLP 2020*, *Findings of the association for computational linguistics: EMNLP 2020* (pp. 1536–1547). Association for Computational Linguistics, http://dx.doi.org/10.18653/v1/2020.findings-emnlp.139.

Gemmell, C., Rossetto, F., & Dalton, J. (2020). Relevance transformer: Generating concise code snippets with relevance feedback. In *Proceedings of the 43rd international ACM SIGIR conference on research and development in information retrieval, SIGIR 2020* (pp. 2005–2008). ACM, http://dx.doi.org/10.1145/3397271.3401215.

Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., & Yin, J. (2022). UniXcoder: Unified cross-modal pre-training for code representation. In S. Muresan, P. Nakov, & A. Villavicencio (Eds.), *Proceedings of the 60th annual meeting of the association for computational linguistics (volume 1: long papers)* (pp. 7212–7225). Association for Computational Linguistics, http://dx.doi.org/10.18653/v1/2022.acl-long.499.

Han, L. (2016). Machine translation evaluation resources and methods: A survey. arXiv preprint arXiv:1605.04515.

Han, L., Smeaton, A., & Jones, G. (2021). Translation quality assessment: A brief survey on manual and automatic methods. In *Proceedings for the first workshop on modelling translation: Translatology in the digital age* (pp. 15–33). Association for Computational Linguistics, online, URL https://aclanthology.org/2021.motra-1.3.

Hu, X., Chen, Q., Wang, H., Xia, X., Lo, D., & Zimmermann, T. (2022). Correlating automated and human evaluation of code documentation generation quality. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *31*(4), 1–28.

Jiang, Y., Cuki, B., Menzies, T., & Bartlow, N. (2008). Comparing design and code metrics for software quality prediction. In *Proceedings of the 4th international workshop on predictor models in software engineering* (pp. 11–18).

Kendall, M. G. (1938). A new measure of rank correlation. *Biometrika*, *30*(1/2), 81–93.

Kim, D., & MacKinnon, T. (2018). Artificial intelligence in fracture detection: transfer learning from deep convolutional neural networks. *Clinical Radiology*, *73*(5), 439–445.

Kingma, D. P., & Ba, J. (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*. URL http://arxiv.org/abs/1412.6980.

Kulal, S., Pasupat, P., Chandra, K., Lee, M., Padon, O., Aiken, A., et al. (2019). Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, *32*.

Lavie, A., & Agarwal, A. (2007). Meteor: An automatic metric for MT evaluation with high levels of correlation with human judgments. In *Proceedings of the second workshop on statistical machine translation* (pp. 228–231). USA: Association for Computational Linguistics.

Li, Z., Wang, X., Aw, A., Chng, E. S., & Li, H. (2018). Named-entity tagging and domain adaptation for better customized translation. In *Proceedings of the seventh named entities workshop* (pp. 41–46).

Liguori, P., Al-Hossami, E., Cotroneo, D., Natella, R., Cukic, B., & Shaikh, S. (2021a). Shellcode_IA32: A dataset for automatic shellcode generation. In *Proceedings of the 1st workshop on natural language processing for programming* (pp. 58–64). Association for Computational Linguistics, http://dx.doi.org/10.18653/v1/2021.nlp4prog-1.7, URL https://aclanthology.org/2021.nlp4prog-1.7.

Liguori, P., Al-Hossami, E., Cotroneo, D., Natella, R., Cukic, B., & Shaikh, S. (2022). Can we generate shellcodes via natural language? An empirical study. *Automated Software Engineering*, *29*(1), 1–34.

Liguori, P., Al-Hossami, E., Orbinato, V., Natella, R., Shaikh, S., Cotroneo, D., et al. (2021b). EVIL: exploiting software via natural language. In *2021 IEEE 32nd international symposium on software reliability engineering* (pp. 321–332). IEEE.

Lin, C.-Y. (2004). ROUGE: A package for automatic evaluation of summaries. In *Text summarization branches out* (pp. 74–81). Barcelona, Spain: Association for Computational Linguistics, URL https://aclanthology.org/W04-1013.

Lin, G., Wen, S., Han, Q.-L., Zhang, J., & Xiang, Y. (2020). Software vulnerability detection using deep neural networks: a survey. *Proceedings of the IEEE*, *108*(10), 1825–1848.

Ling, W., Blunsom, P., Grefenstette, E., Hermann, K. M., Kociský, T., Wang, F., et al. (2016). Latent predictor networks for code generation. In *Proceedings of the 54th annual meeting of the association for computational linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, volume 1: long papers*. The Association for Computer Linguistics, http://dx.doi.org/10.18653/v1/p16-1057, URL https://doi.org/10.18653/v1/p16-1057.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., et al. (2019). RoBERTa: A robustly optimized BERT pretraining approach, CoRR, abs/1907.11692. URL http://arxiv.org/abs/1907.11692, arXiv:1907.11692.

Liu, H., Shen, M., Zhu, J., Niu, N., Li, G., & Zhang, L. (2020). Deep learning based program generation from requirements text: Are we there yet? *IEEE Transactions on Software Engineering*, *48*(4), 1268–1289.

Mashhadi, E., & Hemmati, H. (2021). Applying codebert for automated program repair of java simple bugs. In *18th IEEE/ACM international conference on mining software repositories* (pp. 505–509). IEEE, http://dx.doi.org/10.1109/MSR52588.2021.00063, URL https://doi.org/10.1109/MSR52588.2021.00063.

Mirsky, Y., Demontis, A., Kotak, J., Shankar, R., Gelei, D., Yang, L., et al. (2022). The threat of offensive ai to organizations. *Computers & Security*, Article 103006.

Mirsky, Y., & Lee, W. (2021). The creation and detection of deepfakes: A survey. *ACM Computing Surveys*, *54*(1), 1–41.

Modrzejewski, M., Exel, M., Buschbeck, B., Ha, T.-L., & Waibel, A. (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* (pp. 45–51).

Mokhov, S. A., Paquet, J., & Debbabi, M. (2014). The use of NLP techniques in static code analysis to detect weaknesses and vulnerabilities. In *Canadian conference on artificial intelligence* (pp. 326–332). Springer.

Moramarco, F., Papadopoulos Korfiatis, A., Perera, M., Juric, D., Flann, J., Reiter, E., et al. (2022). Human evaluation and correlation with automatic metrics in consultation note generation. In *proceedings of the 60th annual meeting of the association for computational linguistics (volume 1: long papers)* (pp. 5739–5754). Dublin, Ireland: Association for Computational Linguistics.

Munkova, D., Hajek, P., Munk, M., & Skalka, J. (2020). Evaluation of machine translation quality through the metrics of error rate and accuracy. *Procedia Computer Science*, *171*, 1327–1336.

NASM (2022). Netwide assembler (NASM). URL https://www.nasm.us.

Neubig, G., Sperber, M., Wang, X., Felix, M., Matthews, A., Padmanabhan, S., et al. (2018). XNMT: The extensible neural machine translation toolkit. In *Proceedings of the 13th conference of the association for machine translation in the americas (volume 1: research track)* (pp. 185–192). Boston, MA: Association for Machine Translation in the Americas, URL https://aclanthology.org/W18-1818.

NLTK (2023). Natural Language Toolkit (NLTK), bleu_score module. URL https://www.nltk.org/api/nltk.translate.bleu_score.html.

Papineni, K., Roukos, S., Ward, T., & Zhu, W. (2002). Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the association for computational linguistics* (pp. 311–318). ACL, http://dx.doi.org/10.3115/1073083.1073135, URL https://aclanthology.org/P02-1040/.

Pearson, K. (1895). Notes on regression and inheritance in the case of two parents. In *Proceedings of the royal society of London, vol. 58* (pp. 240–242). K Pearson.

Phan, L. N., Tran, H., Le, D., Nguyen, H., Anibal, J. T., Peltekian, A., et al. (2021). CoTexT: Multi-task learning with code-text transformer, CoRR, abs/2105.08645. arXiv:2105.08645.

py_compile (2023). Python compiler py_compile. URL https://docs.python.org/3/library/py_compile.html.

pylcs (2023). Python library pylcs. URL https://pypi.org/project/pylcs/.

Python (2023). tokenize. URL https://docs.python.org/3/library/tokenize.html.

Rao, S., & Tetreault, J. (2018). Dear sir or madam, may I introduce the GYAFC dataset: Corpus, benchmarks and metrics for formality style transfer. In *Proceedings of the 2018 conference of the north american chapter of the association for computational linguistics: human language technologies, volume 1 (long papers)* (pp. 129–140). New Orleans, Louisiana: Association for Computational Linguistics.

Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., et al. (2020). CodeBLEU: a method for automatic evaluation of code synthesis, CoRR, abs/2009.10297. arXiv:2009.10297.

rouge (2021). Python ROUGE score implementation. URL https://pypi.org/project/rouge/.

Roy, D., Fakhoury, S., & Arnaoudova, V. (2021). Reassessing automatic evaluation metrics for code summarization tasks. In D. Spinellis, G. Gousios, M. Chechik, & M. D. Penta (Eds.), *ESEC/FSE '21: 29th ACM joint european software engineering conference and symposium on the foundations of software engineering* (pp. 1105–1116). ACM, http://dx.doi.org/10.1145/3468264.3468588.

Salminen, J., Jung, S.-g., & Jansen, B. J. (2019). The future of data-driven personas: A marriage of online analytics numbers and human attributes.. In *ICEIS (1)* (pp. 608–615).

Salminen, J., Rao, R. G., Jung, S.-g., Chowdhury, S. A., & Jansen, B. J. (2020). Enriching social media personas with personality traits: A deep learning approach using the big five classes. In *International conference on human-computer interaction* (pp. 101–120). Springer.

Scalabrino, S., Bavota, G., Vendome, C., Linares-Vásquez, M., Poshyvanyk, D., & Oliveto, R. (2021). Automatically assessing code understandability. *IEEE Transactions on Software Engineering*, *47*(3), 595–613. http://dx.doi.org/10.1109/TSE.2019.2901468.

Shimorina, A. (2018). Human vs automatic metrics: on the importance of correlation design. arXiv preprint arXiv:1805.11474.

Shterionov, D., Superbo, R., Nagle, P., Casanellas, L., O'dowd, T., & Way, A. (2018). Human versus automatic quality evaluation of NMT and PBSMT. *Machine Translation*, *32*(3), 217–235.

spaCy (2023). Industrial-strength natural language processing. URL https://spacy.io/.

Stent, A., Marge, M., & Singhai, M. (2005). Evaluating evaluation methods for generation in the presence of variation. In *International conference on intelligent text processing and computational linguistics* (pp. 341–351). Springer.

Stupp, C. (2019). Fraudsters used AI to mimic CEO's voice in unusual cybercrime case. *The Wall Street Journal*, *30*(08).

Svyatkovskiy, A., Deng, S. K., Fu, S., & Sundaresan, N. (2020). IntelliCode compose: code generation using transformer. In P. Devanbu, M. B. Cohen, & T. Zimmermann (Eds.), *ESEC/FSE '20: 28th ACM joint european software engineering conference and symposium on the foundations of software engineering* (pp. 1433–1443). ACM, http://dx.doi.org/10.1145/3368089.3417058.

Takaichi, R., Higo, Y., Matsumoto, S., Kusumoto, S., Kurabayashi, T., Kirinuki, H., et al. (2022). Are NLP metrics suitable for evaluating generated code? In D. Taibi, M. Kuhrmann, T. Mikkonen, J. Klünder, P. Abrahamsson (Eds.), *Lecture notes in computer science*: *vol. 13709*, *Product-focused software process improvement - 23rd international conference, PROFES 2022, JyväSkylä, Finland, November 21-23, 2022, proceedings* (pp. 531–537). Springer, http://dx.doi.org/10.1007/978-3-031-21388-5_38.

Tran, N. M., Tran, H., Nguyen, S., Nguyen, H., & Nguyen, T. N. (2019). Does BLEU score work for code migration? In Y. Guéhéneuc, F. Khomh, & F. Sarro (Eds.), *Proceedings of the 27th international conference on program comprehension* (pp. 165–176). IEEE / ACM, http://dx.doi.org/10.1109/ICPC.2019.00034.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., et al. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems* (pp. 5998–6008).

Wang, Y., Wang, W., Joty, S. R., & Hoi, S. C. H. (2021). CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In M. Moens, X. Huang, L. Specia, & S. W. Yih (Eds.), *Proceedings of the 2021 conference on empirical methods in natural language processing* (pp. 8696–8708). Association for Computational Linguistics, http://dx.doi.org/10.18653/v1/2021.emnlp-main.685.

Wang, C., Yang, Y., Gao, C., Peng, Y., Zhang, H., & Lyu, M. R. (2022). No more fine-tuning? an experimental evaluation of prompt tuning in code intelligence. In A. Roychoudhury, C. Cadar, & M. Kim (Eds.), *Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering* (pp. 382–394). ACM, http://dx.doi.org/10.1145/3540250.3549113.

Yang, G., Chen, X., Zhou, Y., & Yu, C. (2022). Dualsc: Automatic generation and summarization of shellcode via transformer and dual learning. In *IEEE international conference on software analysis, evolution and reengineering* (pp. 361–372). IEEE.

Yang, G., Zhou, Y., Chen, X., Zhang, X., Han, T., & Chen, T. (2023). ExploitGen: Template-augmented exploit code generation based on codebert. *Journal of Systems and Software*, *197*, Article 111577.

Yin, P., Deng, B., Chen, E., Vasilescu, B., & Neubig, G. (2018). Learning to mine aligned code and natural language pairs from stack overflow. In *Proceedings of the 15th international conference on mining software repositories* (pp. 476–486).

Yin, P., & Neubig, G. (2019). Reranking for neural semantic parsing. In *Proceedings of the 57th conference of the association for computational linguistics, ACL 2019, Florence, Italy, July 28- August 2, 2019, volume 1: long papers* (pp. 4553–4559). Association for Computational Linguistics, http://dx.doi.org/10.18653/v1/p19-1447.

Yu, C., Yang, G., Chen, X., Liu, K., & Zhou, Y. (2022). BashExplainer: Retrieval-augmented bash code comment generation based on fine-tuned CodeBERT. In *2022 IEEE international conference on software maintenance and evolution* (pp. 82–93). IEEE.

Zeng, Z., Tan, H., Zhang, H., Li, J., Zhang, Y., & Zhang, L. (2022). An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis* (pp. 39–51).

Zhou, X., Han, D., & Lo, D. (2021). Assessing generalizability of CodeBERT. In *2021 IEEE international conference on software maintenance and evolution* (pp. 425–436). IEEE.