The Journal of Systems & Software 197 (2023) 111577

Contents lists available at ScienceDirect

The Journal of Systems & Software

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

ExploitGen: Template-augmented exploit code generation based on CodeBERT☆

Guang Yang [a,b], Yu Zhou [a,b,*], Xiang Chen [b,c,**], Xiangyu Zhang [a,b], Tingting Han [d], Taolue Chen [d,**]

[a] College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China
[b] Key Laboratory of Safety-Critical Software, Ministry of Industry and Information Technology, Nanjing, China
[c] School of Information Science and Technology, Nantong University, Nantong, China
[d] Department of Computer Science and Data Science, Birkbeck, University of London, UK

ARTICLE INFO

Article history:
Received 16 June 2022
Received in revised form 21 November 2022
Accepted 28 November 2022
Available online 1 December 2022

Keywords:
Exploit code
Code generation
Template parser
CodeBERT
Neural network

ABSTRACT

Exploit code is widely used for detecting vulnerabilities and implementing defensive measures. However, automatic generation of exploit code for security assessment is a challenging task. In this paper, we propose a novel template-augmented exploit code generation approach ExploitGen based on CodeBERT. Specifically, we first propose a rule-based Template Parser to generate template-augmented natural language descriptions (NL). Both the raw and template-augmented NL sequences are encoded to context vectors by the respective encoders. For better learning semantic information, ExploitGen incorporates a semantic attention layer, which uses the attention mechanism to extract and calculate each layer's representational information. In addition, ExploitGen computes the interaction information between the template information and the semantics of the raw NL and designs a residual connection to append the template information into the semantics of the raw NL. Comprehensive experiments on two datasets show the effectiveness of ExploitGen after comparison with six state-of-the-art baselines. Apart from the automatic evaluation, we conduct a human study to evaluate the quality of generated code in terms of syntactic and semantic correctness. The results also confirm the effectiveness of ExploitGen.

1. Introduction

Context. Exploit code (Bao et al., 2017) takes advantage of software vulnerabilities to cause unexpected behaviors during execution, such as controlling computer systems, causing a buffer overflow, or launching a cyber-attack (Arce, 2004; Wang et al., 2010). Malicious exploit code can mount DDoS attacks, data theft, and run malware against the target systems, whereas benign exploit code can be used to identify vulnerabilities of the software system, which can hopefully be repaired later. In software security research, exploit code is usually leveraged as the tool to find security vulnerabilities, which motivates the research on automatic exploit code generation. This is a challenging task since exploit code requires specific programming and obfuscation expertise.

In previous studies, researchers formalize the code generation task as a sequence-to-sequence generation task. The naturalness of code (Hindle et al., 2016) assumes that code exhibits predictable statistical properties like natural languages, which can be captured in statistical language models and thus is used to solve specific software engineering tasks. This has led to wide application of deep learning in automatic code generation. Earlier approaches resort to Encoder–Decoder models based on recurrent neural networks (Mou et al., 2015; Ling et al., 2016; Yin and Neubig, 2018) or convolutional neural networks (Sun et al., 2019), which achieve promising results in domain-specific programming languages (such as Shell Lin et al., 2018, SQL Yu et al., 2018, and regular expressions Locascio et al., 2016). Late approaches (Sun et al., 2020; Gemmell et al., 2020) utilize Transformer (Vaswani et al., 2017). More recently, several studies (Norouzi et al., 2021; Ahmad et al., 2021; Phan et al., 2021; Wang et al., 2021) focus on the pre-trained language models (PLMs) for code generation tasks, including general-purpose programming languages (such as Java Iyer et al., 2018 and Python Yin et al., 2018).

However, previous approaches seldom consider the domain knowledge of specific programming languages. In our considered

☆ Editor: Aldeida Aleti.

* Corresponding author at: College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China.
** Corresponding authors. School of Information Science and Technology, Nantong University, Nantong, China.
E-mail addresses: yang.guang@nuaa.edu.cn (G. Yang), zhouyu@nuaa.edu.cn (Y. Zhou), xchencs@ntu.edu.cn (X. Chen), zhangx1angyu@nuaa.edu.cn (X. Zhang), t.han@bbk.ac.uk (T. Han), t.chen@bbk.ac.uk (T. Chen).

https://doi.org/10.1016/j.jss.2022.111577
0164-1212/

Exploit:
```
shellcode =
("\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\
xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80")
encode = ""
for x in bytearray(shellcode) :
    if x < 128:
        x=x<<1
        encode += '0xAA,'
    encode += '0x'
    encode += '%02x,'%x
```

Regular:
```
def _quick_sort_between(a: List[int], low: int, high: int):
    if low < high:
        k = random.randint(low, high)
        a[low], a[k] = a[k], a[low]
        m = _partition(a, low, high)
        _quick_sort_between(a, low, m - 1)
        _quick_sort_between(a, m + 1, high)
```

Fig. 1. Difference between regular code and exploit code.

task, due to the limited number of datasets related to exploit code and the low similarity between the exploit code and general programming languages, we need to transfer the knowledge learned from PLMs on general programming languages to the exploit code domain via transfer learning (Weiss et al., 2016).

In neural code generation, the most essential step is to represent the semantic information of the natural language (NL) and to generate the correct code. However a semantic discrepancy is largely present between NL and programming languages (PL). To bridge such gap and better understand the semantic information of NL, researchers proposed CodeBERT (Feng et al., 2020), a language model that uses the masked language model and replaced token detection method to pre-train both NL and PL. CodeBERT has proven to have strong generalization capabilities (i.e., highly competitive on downstream tasks related to multiple programming languages) (Zhou et al., 2021). However, the performance of the code generation model for exploit code can be affected by the difference between exploit code and regular code. Therefore, we aim to adjust CodeBERT to accommodate the task of automatic exploit code generation. To bridge the discrepancy between the datasets used by CodeBERT and the exploit code in line-level granularity, we use two adaptive pre-training methods (Gururangan et al., 2020) (i.e., Domain-Adaptive Pre-training and Task-Adaptive Pre-training) to continue pre-training the CodeBERT, and obtain the fine-grained CodeBERT (FG-CodeBERT). Therefore, in the sequel, CodeBERT used in ExploitGen denotes FG-CodeBERT in the rest of this paper.

Motivation. The automatic exploit code generation is to explore critical vulnerabilities before they are exploited by attackers. Similar to the motivation of the previous exploit code generation study (Liguori et al., 2021b), we aim to support both beginners and experienced researchers, by making exploits easier to create and reducing the learning difficulty. Exploit code is characterized by giving the attacker full control over the memory layout and CPU registers, thus is able to attack low-level mechanisms (e.g., heap metadata and stack return addresses) that are inaccessible through high-level programming languages. Fig. 1 uses an example to explain the difference between regular code and exploit code. Unlike regular code generation tasks that focus on logically complex functional code fragments, exploit code contains a large number of low-level arithmetic, logic operations, and bit-level slices.

To explore the practical implications of our study, we gather question posts with tags containing 'exploit' and 'shell code' from Stack Overflow.[1] We then count the number of answers to these posts as well as the number of answers accepted by the questioner. The final results show that only 45% 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, we need an automatic way to generate exploit code and improve developers' productivity. In addition, since there is less work on exploit code generation, defending against exploit code turns out to be less effective. If we could produce exploit code in large quantities, they would be useful to the research of detecting and defending against exploit code.

Moreover, researchers (Heyman et al., 2021; Xu et al., 2022) suggest code generation not only saves developers' programming time, but also automates the process of reminding developers of code they may not remember or write at all, and prevents developers from spending more time searching for it using code search engines. Furthermore, while comments may be relatively lengthy, writing them is still easier than writing code. Also in educational books about programming and algorithms, it is common to use natural language or mathematical expressions to describe the behavior of each line of statements in a program. This type of detailed comment helps developers (Oda et al., 2015) to clarify the logical thinking of the code because it explicitly describes what the program needs to do. In particular, in the program related to the exploit code, a large number of comments are directed to each line of code rather than to the entire program, which is also a feature of the exploit code.

Proposed solution. We propose a novel template-augmented exploit code generation approach ExploitGen based on CodeBERT. In ExploitGen, a rule-based Template Parser is introduced to extract special tokens from NL. The template-augmented NL and code can be obtained accordingly. Since raw NL and template-augmented NL are fed into ExploitGen at the same time, a raw encoder and a template-augmented encoder (Temp Encoder) are used to encode them respectively. To capture the contextual semantic information in both encoders, a semantic attention layer is introduced. Then an additional fusion layer is employed to L2-normalize these two aspects of semantic information and fuse them by a special residual connection. Finally, they are fed to the decoder and the Beam Search algorithm is utilized for code generation.

To evaluate the effectiveness of ExploitGen, we conduct experiments on two real-world exploit code datasets (Liguori et al., 2021b), which were collected from publicly available databases,[2,3] public repositories (e.g., GitHub), and programming guidelines. We first compare ExploitGen with six state-of-the-art baselines (Bahdanau et al., 2014; Vaswani et al., 2017; Yang et al., 2022; Feng et al., 2020; Lu et al., 2021) in terms of three automatic performance metrics (i.e., BLEU-4, ROUGE-W and Exact Match Accuracy). The comparison results show that ExploitGen can outperform these baselines. Moreover, we conduct a human study to evaluate the quality of the generated code in terms of syntactic correctness and semantic correctness. Finally, we also verify the effectiveness of key components of ExploitGen by conducting a set of ablation experiments. Our experiments show the shortcomings of existing fully data-driven learning models on the exploit code domain and suggest a new way of thinking for follow-up studies.

Contributions. The main contributions of this paper can be summarized as below.

- We present a novel template-augmented exploit code generation approach ExploitGen based on CodeBERT, which can effectively integrate template information into the semantics of the raw NL.

[1] https://archive.org/download/stackexchange downloaded in September 2022.

[2] https://www.exploit-db.com/

[3] http://shell-storm.org/shellcode/

- We conduct comprehensive experiments to evaluate the performance of ExploitGen on two large-scale datasets. The result of the automated evaluation shows that ExploitGen outperforms the state-of-the-art baselines.
- To facilitate the replication and reuse of ExploitGen, we make our source code, trained models, as well as the datasets in the GitHub repository publicly available.[4]

Structure. The rest of the paper is organized as follows. Section 2 introduces the background of our work. Section 3 describes the framework of ExploitGen and its key components. Section 4 presents the experimental design and result analysis. Discussions and potential threats to validity are given in Section 5 and Section 6 respectively. Section 7 reviews the related work, and Section 8 concludes the paper.

2. Background

In this section, we introduce the background of CodeBERT and exploit code.

2.1. CodeBERT

CodeBERT (Feng et al., 2020) is a bimodal programming language (PL) and natural language (NL) oriented pre-training model, which is based on the Transformer architecture (Vaswani et al., 2017). CodeBERT is pre-trained on a large-scale dataset CodeSearchNet (Husain et al., 2019), which includes 2.1M bimodal data and 6.4M unimodal data with both bimodal NL-PL pairs data and unimodal code data. The CodeBERT pre-training data is based on six programming languages (i.e., Go, Java, Javascript, PHP, Python, and Ruby), and the model setting is similar to MultiBERT (Wang et al., 2019), except that the data input is not tagged with a display to indicate which language the input data is in. To utilize both bimodal and unimodal large-scale data, CodeBERT proposes a hybrid objective loss function combining the Masked Language Model (MLM) (Devlin et al., 2019; Liu et al., 2019) and Replaced Token Detection (RTD) (Clark et al., 2020).

MLM pre-training task uses bimodal data (i.e., NL-PL pairs), where the NL-PL pair is used as input and randomly selected positions to replace with a special mask token [*MASK*].

Formally, the loss function is $L_{MLM}(\theta) + L_{RTD}(\theta)$ where

$$\mathcal{L}_{\mathrm{MLM}}(\theta) = \sum_{i \in \boldsymbol{m}^{\boldsymbol{w}} \cup \boldsymbol{m}^{\boldsymbol{c}}} -\log p^{D_1}\left(x_i \mid \boldsymbol{w}^{\mathrm{masked}}, \boldsymbol{c}^{\mathrm{masked}}\right)$$

where p^{D_1} is the predicted token by model, $\boldsymbol{m}^{\boldsymbol{w}}$ and $\boldsymbol{m}^{\boldsymbol{c}}$ are the random set of positions for NL and PL to mask as the [*MASK*] token, which means $\boldsymbol{w}^{\mathrm{masked}}$ and $\boldsymbol{c}^{\mathrm{masked}}$.

RTD pre-training task uses bimodal and unimodal data. Specifically, it first uses unimodal NL and PL respectively to train the data generator to restore the randomly masked token, and then CodeBERT as a discriminator to determine whether the token is the original masked token, which is a binary classification problem. The loss function of RTD is defined as follows.

$$L_{RTD}(\theta) = \sum_{i=1}^{|e|+|c|} \left(\sigma(i) \cdot \log p^{D_2}(x_{\mathrm{corrupt}}, i) + (1-\sigma(i)) \cdot (1 - \log p^{D_2}(x^{\mathrm{corrupt}}, i))\right)$$

$$\sigma(i) = \begin{cases} 1, & \text{if } x_i^{\mathrm{corrupt}} = x_i \\ 0, & \text{otherwise.} \end{cases}$$

[4] https://github.com/NTDXYG/ExploitGen

2.2. Exploit code

Exploit code is the software program that takes advantage of a bug or vulnerability to cause unintended or unanticipated behavior to occur on computer software, hardware, or something electronic (usually computerized) by releasing a payload to take control of a target system. It is a list of machine code or executable instructions that are injected into the memory of a computer to take control of the executing applications.

Exploit code takes control of a target system by triggering exploits. It can be from a few bytes to several hundred of bytes in size, while it can be used to kill or restart other processes, cause a denial of service or compromise confidential information. Exploit code usually releases a payload, which contains the code that the attacker wants to execute and is commonly referred to as shellcode.

Some security technologies can easily prohibit the execution of pure exploit code because it contains specific information about the destructive actions the attacker plans to conduct. Exploit code typically involves how programs allocate memory, check the validity of input data, and handle memory errors. Software developers can avoid this threat by strictly defining the input data and rejecting incorrect values.

In order to avoid exploit code being detected, exploit developers utilize encoding and decoding techniques to turn the original exploit code into a new, consistent functional exploit code, which is more difficult to detect by the security software, to evade detection by security software.

Exploit code encoder is a program written in a high-level language (usually Python) that performs mathematical operations on the binary opcode to generate a new binary opcode and append extra opcodes to the exploit code decoder. When this attack payload is injected into the victim system and executed, it decodes itself to obtain the original exploit code. Exploit code decoder is written in Assembly language because it is part of the attack payload.

3. Approach

The overall framework of ExploitGen is shown in Fig. 2. Previous studies (Liguori et al., 2021a,b, 2022) have shown that domain-specific information in the exploit code (e.g., byte array, hex array, and hex token) is largely present in NL.

To fully exploit this domain-specific information, we first propose a rule-based Template Parser to convert domain-specific tokens in the raw NL into special placeholders to generate template-augmented NL and storage dictionary *slot-map* for storing real tokens corresponding to placeholders. The Template Parser parses the raw code according to the *slot-map*, converts the special tokens in the code into special placeholders, and generates the template-augmented code. The next step is to input the raw NL (denoted as X) and the template-augmented NL (denoted as X') into Raw Encoder and Temp Encoder, respectively. The final contextual semantic vector is obtained after the semantic attention layer and the fusion layer, and then fed into the decoder to generate the corresponding template-augmented code by the Beam Search algorithm. Finally, ExploitGen combines the *slot-map* parsed by the Template Parser with the template-augmented code generated by the model to parse the raw code.

In the rest of this section, we present the details of the preprocessing (in Section 3.1), the model architecture (in Section 3.2) as well as the model training (in Section 3.4).

3.1. Data processing

For data processing, ExploitGen pre-processes the NL and code and then splits them into corresponding token sequences before parsing them by the Template Parser.

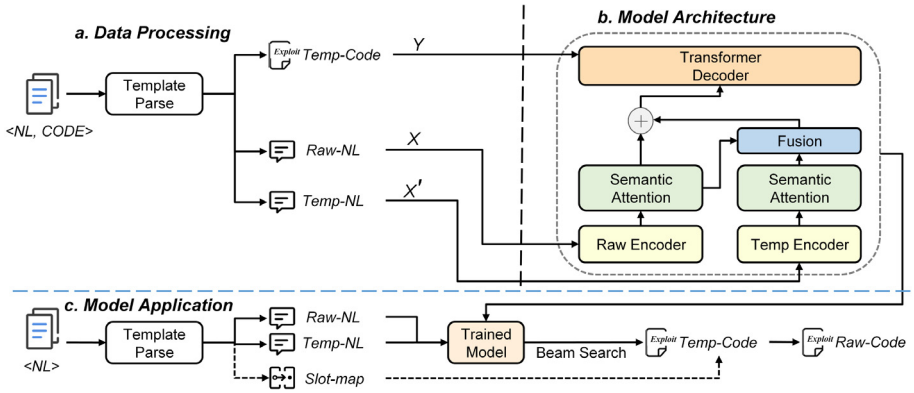


Fig. 2. Framework of ExploitGen.

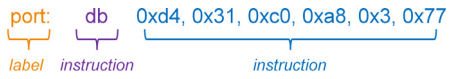


Fig. 3. Example of Assembly code.

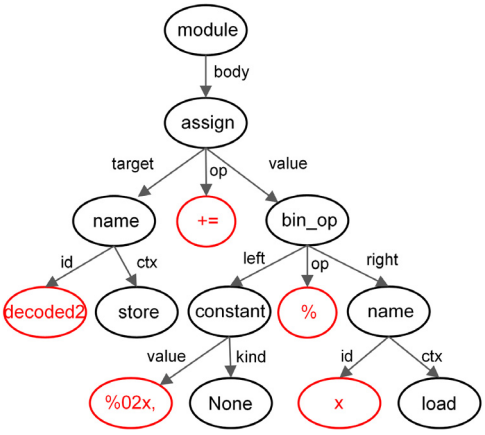


Fig. 4. Example of Python code.

Syntax-based tokenization. For NL, ExploitGen uses the spacy[5] tool for tokenization. For exploit code, we developed language-specific tokenization methods based on different language features (particularly, Assembly and Python in our approach).

Exploit code in Assembly language is for IA-32 (the 32-bit version of the x86 Intel Architecture) from publicly available security exploits. These Assembly instructions are written with Netwide Assembler (NASM) (Frank, 2000) for Linux. Each code unit contains an optional *label* used to represent either an identifier or a constant, and an *instruction* which identifies the purpose of the statement and is followed by zero or more *operands* specifying the data to be manipulated. Fig. 3 shows a simple example of Assembly code. For this type of Assembly code, ExploitGen tokenizes mainly based on punctuation and spaces. For the code shown in Fig. 3, it can be tokenized into ['port', ':', 'db', '0xd4', ',', '0x31', ',', '0xc0', ',', '0xa8', ',', '0x3', ',', '0x77'].

Exploit code in Python is also from publicly available security exploits. These Python code snippets are written for performing low-level arithmetic and logical operations and bit-level slicing. The tokenization of Python code utilizes abstract syntax tree (AST). For example, the raw code is `decoded2 += '%02x,'%x` the AST of which is shown in Fig. 4, and can be tokenized into ['decoded2', '+=', '%02x,', '%', 'x'].

Table 1
Regular expressions for Template Parser.

Type	Regular expression
byte array	b? ?\"? ?\\x[0-9a-z\]+ ?\"?
hex array	(?<=\W)[,]*0x[a-f0-9]+
hex token	(?<=\W)0x[a-f0-9]+
camelCase token	(?<=\W)[a-z]*[A-Z]\w+
underline token	(?<=\W)[a-z]+_\w+
function name	[\w]+(?= function)|[\w]+(?= routine)
bracket values	\[(.*?)\]
quote values	(?P<quote>["'\"])(?P<string>.*?)(?P=quote)
math expression	((\d+\.?\d+|\w+)(?)(?:[\+\-*\/%])(?))+(\d+\.?\d+|\w+)

Rule-based template parser. Template Parser has two advantages. Firstly, it can reduce the difficulty of the code generation task; and secondly, it can alleviate the Out-of-Vocabulary (OOV) problem. Without Template Parser, some domain-specific tokens could not be recognized, which may lead to the generation of incorrect code.

We propose a rule-based Template Parser, which uses regular expressions to extract domain-special tokens (i.e., byte array, hex array, hex token, camelCase token, underline token, function name, bracket values, quote values, and math expression) from NL. In addition, for some special tokens (i.e., register names in Assembly code), ExploitGen uses the part-of-speech tags by Spacy for identification. The regular expressions are given in Table 1.

These rules are scanned and applied when applicable. As an example, for the raw assembly code `add byte [esi], 0x10` and its corresponding NL is *add 0x10 to the current byte in esi*. The Template Parser can parse that *0x10* is the hex token and *esi* is the register name and then replaces the selected token in both the raw code and NL with *var#*. Thus, the template-augmented NL becomes *add var0 current byte var1* and the template-augmented code becomes *add byte [var1], var0*. Meanwhile, the *slot-map* is obtained as *{'0x10': 'var0', 'esi': 'var1'}*.

For raw Python code `def find_valid_xor_byte(bytes, bad_chars):` and its corresponding NL is *define the function find_valid_xor_byte with input parameters bytes and bad_chars*. Template Parser can parse that *find_valid_xor_byte* is the function name and *bad_chars* is the underline token and then replaces the selected token in both the raw code and NL with *var#*. Thus, template-augmented NL

[5] https://github.com/explosion/spaCy

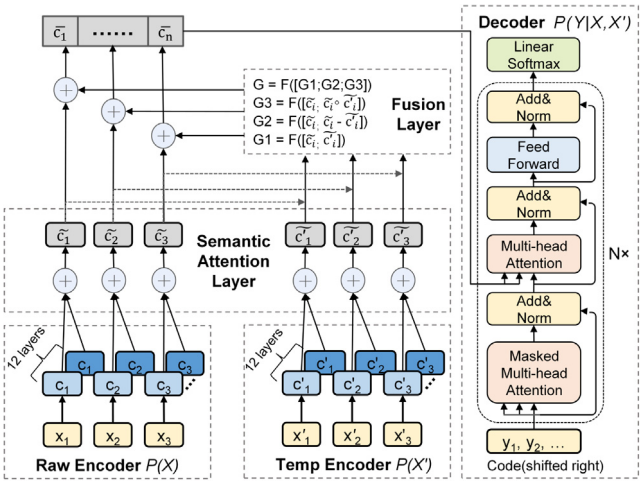


Fig. 5. Framework of our proposed approach ExploitGen.

becomes *define function var0 with input parameters bytes and var1* and template-augmented code becomes *def var0 (bytes, var1) :*. Meanwhile, the *slot-map* is obtained as *{'find_valid_xor_byte': 'var0', 'bad_chars': 'var1'}*.

3.2. Model architecture

The architecture of our model is illustrated in Fig. 5. Our model contains two encoders (i.e., raw encoder and temp encoder), which process tokenized raw NL and template-augmented NL respectively. ExploitGen then utilizes the semantic attention layer and the fusion layer for better learning semantic information and effective integration of template information into the semantic information of the raw NL. Finally, ExploitGen resorts to Transformer's decoder to generate the probabilities of the code sequences.

3.2.1. Encoder layer

Since the Template Parser is rule-based, it cannot ensure complete accuracy and can fail in some cases. For example, Template Parser may fail to parse the code due to problematic data problem. We use two examples in Fig. 6 to illustrate this problem. The first example is for defining a variable. If the variable name in NL does not match the variable name in the code, it will cause the Template Parser restore to fail. The second example is for importing a module. If the module name in NL does not match the module name in the code, it will also cause the Template Parser restore to fail.

Manually analyzing all the problematic data is laborious and time-consuming. Therefore we alleviate this problem by considering dual encoders. Specifically, ExploitGen uses Raw Encoder to encode the raw NL and uses Temp Encoder to encode template-augmented NL respectively.

Raw Encoder. The Raw Encoder encodes the tokens of raw NL and aims to learn the origin information within it. The encoder is based on CodeBERT, which is described in Section 2. For a raw NL, the Raw Encoder first tokenizes the inputs by the pretrained Byte-level BPE (Wang et al., 2020) and obtains the sequence $X = x_1, \ldots, x_n$, where n is the length of this sequence. Since the sequence length may be different for different inputs, we use a padding operation to unify the sequence length to facilitate the processing of the model. Supposing the maximum input length is N, for the sequences whose length is less than N, we pad 0 to the end of these sequences. For the sequences whose length is larger than N, we directly truncate the end of these sequences. The Raw Encoder then feeds X into the model to obtain a set of context vectors $C_{raw} \in \mathbb{R}^{\text{batch}\times 12\times N\times d_{\text{model}}}$, where *batch* means the batch size, 12 is the number of stacked encoding layers in CodeBERT, and d_{model} is the size of the embedding dimension.

Temp Encoder. The Temp Encoder encodes the tokens of template-argument NL and aims to learn the template information within it. Similarly, for a template-argument NL sequence $X' = x'_1, \ldots, x'_n$, the Temp Encoder also encodes it based on CodeBERT, and returns a set of context vectors $C_{temp} \in \mathbb{R}^{\text{batch}\times 12\times N\times d_{\text{model}}}$. Note that although the Raw Encoder and the Temp Encoder have the same structure, they do not share parameters with each other.

In summary, Raw Encoder is used to understand NL containing domain-specific key information since domain-specific token may exhibit OOV problem. Temp Encoder is used to understand NL, which does not contain domain-specific template information (i.e., using rule to replace domain-specific token with a special placeholder). The Fusion Layer is proposed to append the template information to the domain-specific information, so that the model can still understand NL even if the domain specific token input by the developer is not in the model's vocabulary.

3.2.2. Semantic attention layer

Researchers usually extract the last layer of semantic vectors to input into the decoder (Feng et al., 2020). However, exploring the learned representation at each layer in the BERT and CodeBERT models is also investigated (Jawahar et al., 2019; Karmakar and Robbes, 2021). Specifically, the shallow layers in the CodeBERT model focus more on learning Surface-level information, the intermediate layers focus more on learning Syntactic information and Structural information, and the deep layers focus more on learning Semantic information. Although CodeBERT uses multiple residual connections to retain the semantic information learned in the previous layer, the semantic information learned in the shallow layer will still be forgotten after 12 layers of propagation. Since we propose two different encoders to learn NL with domain-specific information and NL with template information respectively, intuitively they may be learned in different layers. Therefore, we propose the Semantic attention layer to dynamically utilize the information learned by each layer.

Since C_{raw} and C_{temp} extract the output from each layer of CodeBERT, their second dimension 12 can be viewed as maintaining a semantic progression relationship from shallow layers to deep layers. The semantic attention layer aims to combine the semantics of each layer according to the assigned weights of each layer. Therefore, we use the attention mechanism to extract the more important representational information.

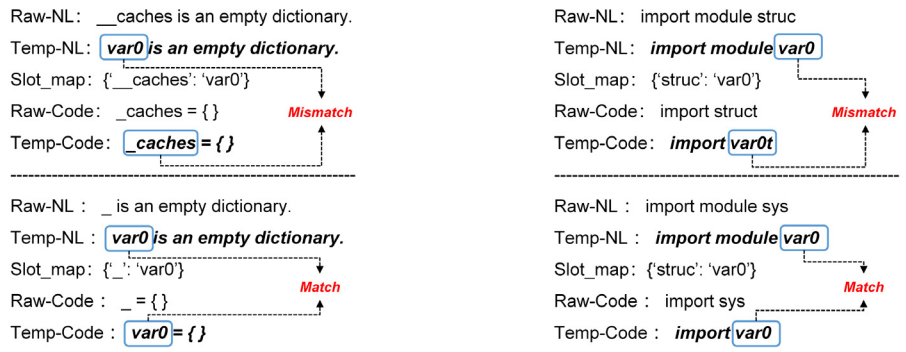
We formalize the context vectors C as $c_1, \ldots, c_{12}$, where $c_i \in \mathbb{R}^{\text{batch}\times N\times d_{\text{model}}}$. ExploitGen first converts c_i to u_i via the full connection layer, $u_i = \tanh(Wc_i + b)$. Then the similarity between u_i and the context vector u_j can be calculated and transformed into a probability distribution by Softmax.

$$\tilde{C} = \sum_{i=1}^{12} \alpha_i c_i$$

where

$$\alpha_i = align(u_i, u_j) = \frac{\exp\left(u_i^T u_j\right)}{\sum_{j=1}^{12} \exp\left(u_i^T u_j\right)}$$

Here, α_i can be treated as the importance of the input for each layer. Therefore, using α_i as a global weighted summation over c_i can generate the final semantic vector $\tilde{C} \in \mathbb{R}^{\text{batch}\times N\times d_{\text{model}}}$. Finally, for both C_{raw} and C_{temp}, we can get the semantic vector $\tilde{C}_{raw}$ and $\tilde{C}_{temp}$.



(a) Example of a parsing error in defining a variable

(b) Example of a parsing error in importing a module

Fig. 6. Two examples of failed template parser.

3.2.3. Fusion layer

The fusion layer is used to efficiently combine the two semantic vectors $\tilde{C}_{raw}$ and $\tilde{C}_{temp}$. In this study, we first use the L2 Normalization to process the semantic vectors, which can result in the normalized vectors, and then utilize the fusion layer designed by Yang et al. (2019) due to its simplicity and effectiveness. This fusion layer includes three splicing methods, which are illustrated as follows.

$$\tilde{C}_{raw} = \frac{\tilde{C}_{raw}}{\left\|\tilde{C}_{raw}\right\|} \quad \tilde{C}_{temp} = \frac{\tilde{C}_{temp}}{\left\|\tilde{C}_{temp}\right\|}$$

$$G_1 = F_1([\tilde{C}_{raw}; \tilde{C}_{temp}]) \quad G_2 = F_2([\tilde{C}_{raw}; \tilde{C}_{raw} - \tilde{C}_{temp}])$$
$$G_3 = F_3([\tilde{C}_{raw}; \tilde{C}_{raw} \circ \tilde{C}_{temp}])$$

Here, F_1, F_2, and F_3 are three single-layer feedforward neural networks with mutually independent parameters. [;] means the concatenation of two vectors. - denotes the subtraction of two vectors, which can highlight the difference between the two vectors. $\circ$ means the dot product of two vectors, which can highlight the similarity between the two vectors. Then we directly concatenate the three obtained vectors, input them into another feed-forward neural network and then compute the output G of the fusion layer as follows.

$$G(\tilde{C}_{raw}, \tilde{C}_{temp}) = \tanh(W([G_1; G_2; G_3]) + b)$$

To prevent the loss of raw information and to mitigate the problem of gradient disappearance, we include a residual network in the fusion layer. Finally, ExploitGen computes the semantic vectors $\bar{C}$ and feeds it into the decoder.

$$\bar{C} = G(\tilde{C}_{raw}, \tilde{C}_{temp}) + \tilde{C}_{raw}$$

3.2.4. Decoder layer

The Decoder aims to generate the target sequence y by sequentially predicting the probability of a word y_{i+1} conditioned on the semantic vector $\bar{C}$ and its previous generated words $y_1, \ldots, y_i$. In this study, we use Transformer's decoder part as the ExploitGen's decoder. Specifically, the decoder uses the sentence sequence to predict the next word (i.e., the previous output is input to the *Mask Multi-Head Attention layer*). The function of *Mask Multi-Head Attention* is to block the following words to prevent information leakage. In the next step, the decoder first computes the semantic vector $\bar{C}$ and the current vector of the decoder with *Multi-Head Attention layer*, i.e., its query comes from the output of the decoder of the previous layer, both its key and its value come from the semantic vector $\bar{C}$. Later, the decoder passes the Residual Connection and Layer Normalization layer, then enters the Feed-forward layer and performs residual connection and layer normalization. The above step is repeated by ExploitGen N times, where N represents the number of layers of the decoder. Finally, the output of the decoder ht is input into a fully connected neural network, which then passes a softmax layer to predict the probability of the next token as follows.

$$P(y_i \mid y_1, \ldots, y_{i-1}) = \text{softmax}(Wh_{i-1} + b) \quad (1)$$

3.3. Model application

In the model application part, ExploitGen first pre-processes the raw NL to obtain the raw NL sequence, the template-augmented NL sequence and the slot-map storing the domain knowledge, respectively.

Then ExploitGen inputs the raw NL sequence and the template-augmented NL sequence into model and generates the corresponding template-argument code according to the probability. Previous studies (Wiseman and Rush, 2016; Freitag and Al-Onaizan, 2017) showed that using neural networks' maximum probability distribution to generate text often leads to low-quality results. Recently, most studies (Yang et al., 2021a,b, 2022) used beam search to achieve better performance on text generation tasks. Therefore, in our proposed approach ExploitGen, we also use the beam search to generate template-argument code sequences. Specifically, the beam search is a compromise between the greedy strategy and the exhaustive strategy. It retains top-k high probability words at each step of the prediction as to the input for the next time step, where k denotes the beam size. The larger the value of k, the greater the possibility of achieving better performance, but at the cost of more computational cost.

In the last step, ExploitGen extracts the keys from the *slot-map* and matches them with the placeholders of the template-argument code. It replaces all keys in the template-argument code with the corresponding memorized value.

3.4. Model training strategies

Since CodeBERT is pre-trained on code datasets with method-level granularity, which is different for downstream tasks with line-level granularity. Specifically, the line-level granularity means that each line of code has a corresponding NL comment and the method-level granularity means that each complete code method segment has a corresponding NL comment. To bridge the discrepancy between the dataset used by CodeBERT and the exploit code based on line-level granularity in our work, we propose FG-CodeBERT, which uses both Domain-Adaptive Pretraining (DAPT) and Task-Adaptive Pretraining (TAPT) methods for adaptive pre-training of CodeBERT. DAPT is meant to continue

Table 2
Statistics of datasets used in our empirical study.

Statistics	Python	Assembly
# Train	14,790	3105
# Dev	375	305
# Test	375	305
Avg. tokens in nl	11.9	9.1
Mode. tokens in nl	4.0	4.0
Avg. tokens in code	9.5	5.1
Mode. tokens in code	6.0	4.0
nl length < 64	99.9%	100%
code length < 64	99.81%	99.68%

pre-training on a domain-relevant large-scale dataset, followed by fine-tuning for a specific task. TAPT is meant to continue pre-training on a task-relevant dataset, followed by fine-tuning for a specific task. Gururangan et al. (2020) showed that both DAPT and TAPT can improve model performance and combining these two methods (i.e., DAPT followed by TAPT) can further improve the model performance.

Therefore, we follow the Masked Language Model (MLM) pre-training method used in most pre-trained language models (e.g. BERT Devlin et al., 2019, RoBERTa Liu et al., 2019, and CodeBERT Feng et al., 2020), where line-level granularity code and their corresponding comments are concatenated together as input, randomly selected positions for code and comment are masked, and then replaced with a special mask token. The goal of the masked language model is to predict the original token.

For DAPT, we choose the SPoC (Kulal et al., 2019) dataset, which contains C++ problem solutions from Codeforces[6] (a competitive programming website). It contains a total of 18,356 programs, with an average of 14.7 lines per program. Each line is annotated with a NL given by a crowd of workers from Amazon Mechanical Turk.[7] On average, there were 7.86 tokens per line of code and 9.08 tokens per NL. For TAPT, we then select the Assembly and Python datasets used in our downstream task, and the detailed statistics are shown in Table 2.

To learn the template information, we combined the above datasets in three grouping ways (i.e., raw NL $\oplus$ raw code, raw NL $\oplus$ template-argument code, and template-argument NL $\oplus$ template-argument code) to obtain a total of 346,859 unique pairs of NL and code, where $\oplus$ means the concatenation of two sequences.

Since adaptive pre-training strategy can bridge the discrepancy between CodeBERT and our task, we propose a two-stage training strategy to better initialize the corresponding parameters of Raw Encoder and Temp Encoder respectively.

For the first stage, we construct the standard encoder–decoder architecture by adding Transformer's decoder to Raw Encoder and Temp Encoder. Their inputs and outputs are ⟨raw NL(X), template-argument code(Y)⟩ and ⟨template-argument NL(X'), templated-argument code(Y)⟩ respectively. Their loss functions in the training process are defined as follows.

$$\mathcal{L}_{\text{Raw}} = -\sum_{i=1}^{|y|} \log P_\theta(y_i|y < i, X)$$

$$\mathcal{L}_{\text{Temp}} = -\sum_{i=1}^{|y|} \log P_\theta(y_i|y < i, X')$$

For the second stage, we extract the Raw Encoder and Temp Encoder trained above and load the parameters into ExploitGen, which can guarantee a better initialization parameters for the dual encoder in ExploitGen. Then the whole model is trained by the loss function, which is defined as follows.

$$\mathcal{L}_\theta = -\sum_{i=1}^{|y|} \log P_\theta(y_i|y < i, X, X')$$

Similar to the previous studies (Feng et al., 2020; Liguori et al., 2021b), during the training phase, we fine-tune all parameters of the CodeBERT model.

4. Experiments

In our empirical study, we mainly focus on the following five research questions:

- **RQ1:** How effective is ExploitGen compared with the state-of-the-art baselines in terms of automatic evaluation?
- **RQ2:** How effective is ExploitGen in terms of syntactic correctness and semantic correctness?
- **RQ3:** What is the impact of different NL lengths and code lengths on the performance of ExploitGen?
- **RQ4:** What is the impact of different input formats on the performance of ExploitGen?
- **RQ5:** What is the impact of *Semantic Attention Layer* and *Fusion Layer* on the performance of ExploitGen?
- **RQ6:** What is the impact of *Adaptive Pre-training Strategy* and *Two-Stage Training Strategy* on the performance of ExploitGen?

In RQ1, we evaluate whether and to what extent our ExploitGen outperforms the baselines by automatic evaluation. Since automatic metrics are based on overlap, human study can better analyze the quality of the generated code. In RQ2, we evaluate the performance of ExploitGen to generate the exploit code by syntactic correctness and semantic correctness, where syntactic correctness is verified by static program analysis and semantic correctness is verified by human. In RQ3 and RQ4, we investigate the impact of various input formats and input lengths of ExploitGen on its performance. In RQ5 and RQ6, we investigate the impact of various training strategies and model components of ExploitGen on its performance.

4.1. Baselines

To show the competitiveness of ExploitGen, we select six state-of-the-art baselines, which were widely chosen in the previous studies on code generation. Specifically, we classify these baselines into two groups. The first group includes three approaches based on training from scratch (i.e., Seq2Seq Bahdanau et al., 2014, Transformer Vaswani et al., 2017, and DualSC Yang et al., 2022). The second group includes three baselines based on pre-trained models (i.e., CodeGPT Lu et al., 2021, CodeGPT-adapted, and CodeBERT Feng et al., 2020).

Seq2Seq. Seq2Seq (Bahdanau et al., 2014) is an encoder–decoder architecture based on LSTM and attention mechanism.

Transformer. Transformer (Vaswani et al., 2017) is an encoder–decoder architecture based on the self-attention mechanism that achieves better performance while reducing the computational cost and improving parallel efficiency.

DualSC. DualSC (Yang et al., 2022) formalizes automatic shellcode generation and summarization as dual tasks. It uses a shallow Transformer for model construction and designs a normalization method Adjust_QKNorm. Finally, it uses a rule-based repair component to improve the performance of automatic shellcode generation.

[6] http://codeforces.com/
[7] https://www.mturk.com/

CodeGPT and CodeGPT-adapted. CodeGPT and CodeGPT-adapted are based on the architecture and training objective of GPT-2 (Budzianowski and Vulić, 2019), which is the decoder-only model. CodeGPT is pre-trained from scratch on CodeSearchNet dataset (Lu et al., 2021) while CodeGPT-adapted learns this dataset starting from the GPT-2 checkpoint.

CodeBERT. CodeBERT (Feng et al., 2020) employs the same architecture as RoBERTa (Liu et al., 2019), which is the encoder-only model. CodeBERT aims to minimize the combined loss from masked language modeling and replaced token detection.

4.2. Datasets

In our empirical study, we conduct experiments on two publicly available security exploit datasets shared by Liguori et al. (2021b) (i.e., Python data and Assembly data). Table 2 shows the statistical information of our used datasets. The first three rows of this table represent the size of the training set, the validation set, and the test sets, respectively. The four middle rows represent the average and mode of NL and code lengths, respectively. The last two rows represent the percentage of NL and code whose lengths are less than 64.

These datasets were gathered from real-world projects and were the first datasets targeted at generating exploit code. The Python code of real exploits entails low-level operations on byte data for obfuscation purposes. Therefore, real exploits make extensive use of Python instructions for converting data between different encoders, for performing low-level arithmetic and logical operations, and for bit-level slicing, which is the difference between this dataset and the previous generic Python dataset. The Assembly codes of real exploits were collected from shellcode for IA-32 and written for the Netwide Assembler (NASM) for Linux. To make a fair comparison with the previous work, we follow the dataset split setting of Liguori et al. (2021b), and the test sets cover 20 different exploits (i.e., 20 Python programs and 20 Assembly programs). The purpose of this dataset split is to verify the accuracy of generating complete exploit code.

Since not every line of code has a corresponding comment, to mitigate bias, Liguori et al. (2021b) reused comments written by developers in the program code. When comments were not available, they followed the style of books/tutorials on Assembly/Python and shellcode programming to write comments manually. We take a complete file of code[8] from the test set, as shown in Fig. 7.

4.3. Performance metrics

In our study, we use three performance metrics (i.e., *BLEU*, *ROUGE* and *Exact Match Accuracy*), which have been widely used in neural machine translation and code generation.

BLEU. BLEU (Papineni et al., 2002) is a set of measures for evaluating a candidate sentence and the reference sentence to cover precision. In this study, we use BLEU-4, which means to compare the degree of overlap between the reference sentence and the 4-gram in the generated sentence, and the higher the degree of overlap, the higher the quality of the model generation is considered. For implementation, we utilize the nltk[9] library

ROUGE. ROUGE (Lin, 2004) is a set of measures for evaluating a candidate sentence and the reference sentence to cover recall. In this study, we use ROUGE-W, where W indicates the degree of matching of the longest common sub-sequence LCS with weights.

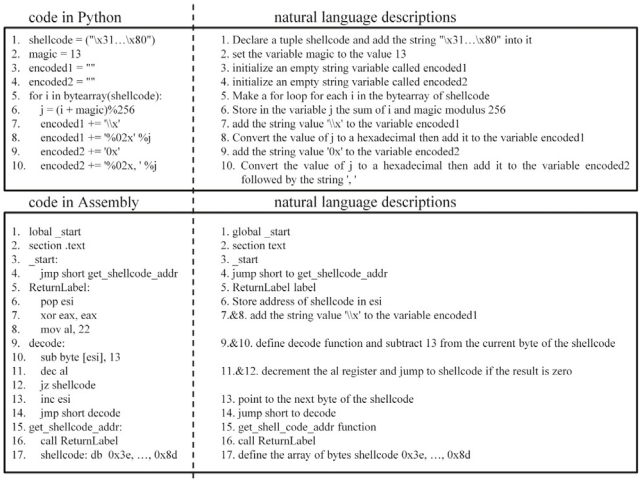


Fig. 7. Examples of real-world exploit code and their corresponding comments.

Table 3
Hyper-parameters and their values in our empirical study.

Category	Hyper-parameter	Value
Model structure	n_layers	6
	n_heads	12
	hidden_size	764
Model training	optimizer	AdamW
	learning rate	4e−5
	batch size	32
	max nl length	64
	max code length	64
Model application	beam size	10

The longer the successive sub-sequence, the greater the weights. For implementation, we utilize the easy-rouge[10] library.

Exact Match Accuracy. Exact Match Accuracy (Yin and Neubig, 2017) is the fraction of exactly matching samples between the predicted output and the reference. Different from similarity-based measures, Exact Match Accuracy considers whether the generated code is identical to the ground truth code. Therefore this measure can accurately evaluate the effectiveness of the generated exploit code.

4.4. Experimental settings

Adaptive Pre-training Hyper-parameters. Following the studies of Devlin et al. (2019) and Feng et al. (2020), we set the input masking probability to 15%. We use AdamW to update the parameters and use the following set of hyper-parameters to train models: batch size is 64, learning rate is 4e-5, max length is 64, and the max training step is 100K.

Two-Stage Training Hyper-parameters. The hyper-parameters of ExploitGen can be classified into three categories (i.e., the hyper-parameters for the model structure, the hyper-parameters in the model training phase, and the hyper-parameters in the model application phase). These hyper-parameters and their values are shown in Table 3. Notice '*n* layers' means the number of layers in decoder.

Hardware Configurations. All the experiments run on a workstation with an Intel(R) Core(TM) i7-11700 CPU, 64 GB RAM, and a GeForce RTX3090 GPU with 24 GB memory. The running OS platform is Linux OS.

8 https://www.abatchy.com/2017/05/rot-n-shellcode-encoder-linux-x86

9 https://github.com/nltk/nltk

10 https://pypi.org/project/easy-rouge/

Table 4
The comparison results between our proposed approach ExploitGen and baselines for two different datasets.

Dataset	Approach	BLEU-4 (%)	ROUGE-W (%)	Accuracy (%)
Python	Seq2Seq	69.92	60.18	37.33
	Transformer	75.23	63.21	41.07
	DualSC	80.29	67.17	53.87
	CodeGPT	79.36	67.10	48.80
	CodeGPT-adapted	86.62	71.40	62.67
	CodeBERT	87.93	72.73	59.47
	ExploitGen	**91.27**	**75.68**	**70.93**
Assembly	Seq2Seq	64.67	56.43	32.46
	Transformer	70.40	59.62	38.36
	DualSC	75.83	64.89	50.49
	CodeGPT	72.05	66.70	48.52
	CodeGPT-adapted	80.70	68.62	61.31
	CodeBERT	81.55	67.37	51.80
	ExploitGen	**88.70**	**74.25**	**66.89**

Table 5
Examples of the ground truth exploit code, and the exploit code generated by ExploitGen and all baselines.

Type	Example
Python	**NL:** set the variable z to bitwise not x **Seq2Seq:** new = x x x **Transformer:** z = x ∧ x **DualSC:** z = x % z **CodeGPT:** z = x **CodeGPT-adapted:** z = bitwise **CodeBERT:** z = x ∧ 10 **ExploitGen:** z = ∼ x **Ground Truth:** z = ∼ x
Assembly	**NL:** not operation of current byte in esi **Seq2Seq:** dec byte esi **Transformer:** mov var1 , byte [esi] **DualSC:** negate all **CodeGPT:** nop **CodeGPT-adapted:** pop byte [esi] **CodeBERT:** nop **ExploitGen:** not byte [esi] **Ground Truth:** not byte [esi]

4.5. Results

RQ1: How effective is ExploitGen compared with the state-of-the-art baselines?. In this RQ, we want to investigate how effective ExploitGen is and how much performance improvement ExploitGen can achieve over the state-of-the-art baselines. As introduced in Section 4.1, we use six state-of-the-art source code generation techniques as baselines. Since in the automatic exploit code generation task, we use the *Template Parser* to parse the raw NL and get the template-augmented NL and template-augmented code. To guarantee a fair comparison between ExploitGen and baselines, we also apply this component to our considered baselines.

We show the comparison results between our proposed approach ExploitGen and baselines in Table 4. For the Python dataset, ExploitGen achieves 3.80% to 30.53%, 2.95% to 15.50%, and 8.26% to 33.60% improvements over the baselines in terms of BLEU-4, ROUGE-W, and Exact Match Accuracy respectively. For the Assembly dataset, ExploitGen achieves 8.77% to 37.16%, 5.63% to 17.82%, and 5.58% to 34.43% improvements over the baselines in terms of BLEU-4, ROUGE-W, and Exact Match Accuracy respectively. Therefore, compared to the baselines, ExploitGen outperforms all the six baselines in terms of three metrics on the two datasets.

In addition, we show two examples with the ground-truth exploit code and the exploit code generated by ExploitGen and all baselines in the Python dataset and the Assembly dataset respectively. From Table 5, we can observe that ExploitGen can generate higher quality code compared to baselines.

To examine the performance difference in view of each comparison between ExploitGen and baselines, We use Wilcoxon signed-rank test to examine whether the performance difference is statistically significant in terms of performance metrics BLEU-4, ROUGE-W, and Exact Match Accuracy. Notice here we only selected three baseline methods, since the baseline CodeGPT-adapted can perform best in terms of the metric ROUGE-W, the baseline CodeBERT can perform best in terms of Exact Match Accuracy, and the baseline Transformer can be used as a comparison without pre-trained models. In our study, the hypothesis is set as follows, H_0: There is no significant difference between ExploitGen and the baselines in terms of metrics BLEU-4 ROUGE-W and Exact Match Accuracy. The significance level of this test is set as 0.05. The results are shown in Table 6 and we find all the p-values are lower than 0.05. These statistical results can lead to the rejection of the null hypothesis, which means that there exists a significant difference between our approach and baselines in terms of all the considered metrics. Note that the results listed in Table 4 show that our approach outperforms other baselines, so we can draw a conclusion that ExploitGen significantly achieves better performance than other baseline approaches.

Summary for RQ1

ExploitGen can significantly outperform the baselines in terms of three performance metrics on both the Python dataset and the Assembly dataset.

RQ2: How effective is ExploitGen in terms of syntactic correctness and semantic correctness?. Although the results show that ExploitGen can correctly generate individual lines of code with high probability in automatic evaluation, we need to verify the performance of ExploitGen for generating the entire software exploit code. At the same time, we find that there are semantically identical but written differently codes (e.g., in Python code, `a += b` is exactly equivalent to `a = a + b`). This implies that automatic evaluation may not correctly reflect the performance of the model. Therefore, we followed the methodology of Liguori et al. (2021b, 2022) and evaluate the ability of ExploitGen to generate semantic correctness and syntactic correctness code for the entire software exploit code (i.e., all lines of code in the program) using two new metrics.

Syntactic Correctness. This metric means that the generated code conforms to the syntax rules of the programming language, which we first analyze with static tools and then review by hand.

Semantic Correctness. This metric means that the generated code should not only conform to the syntax rules of the programming language, but also maintain the semantic consistency of the code. We first perform an initial screening by Exact Match Accuracy to find out the inconsistent code and then review them manually to filter out the semantically consistent code.

In our human evaluation, following the previous studies (Liu et al., 2018; Hu et al., 2020, 2022) we hire three third-year postgraduate students majoring in Software Engineering as reviewers, who have about five years of experience in code development with Python and Assembly languages. We use the entire 20 programs in the Python test set and Assembly test set, which include human-written exploit code and the exploit code generated by ExploitGen. We follow the experimental design principles[11] and conduct a within-subject experiment, as each reviewer will

[11] https://opentextbc.ca/researchmethods/chapter/experimental-design/

Read the following comments and codes:
decrease the counter and jump to decode if not zero else jump short to shellcode
Ground Truth:
loop decode
jmp short shellcode

Candidate 1:
loop decode
jmp short shellcode
Please evaluate the syntactic correctness of Candidate 1: Score 0 or 1 (0 is incorrect and 1 is correct)
Please evaluate the semantic correctness of Candidate 1: Score 0 or 1 (0 is incorrect and 1 is correct)

Candidate 2:
loop decode
jmp shellcode
Please evaluate the syntactic correctness of Candidate 2: Score 0 or 1 (0 is incorrect and 1 is correct)
Please evaluate the semantic correctness of Candidate 2: Score 0 or 1 (0 is incorrect and 1 is correct)

Fig. 8. An example questionnaire used in the human evaluation.

Table 6
The performance difference (measure by p value) between ExploitGen and the baselines based on Wilcoxon signed-rank test on the two datasets.

Dataset	Baseline	BLEU-4 (%)	ROUGE-W (%)	Accuracy (%)
Python	CodeGPT-adapted	1.84e−5	2.68e−5	4.11e−4
	CodeBERT	6.20e−5	3.98e−5	2.26e−7
	Transformer	1.15e−25	1.45e−24	3.67e−24
Assembly	CodeGPT-adapted	1.68e−5	4.69e−5	0.05
	CodeBERT	8.53e−9	3.28e−10	8.92e−9
	Transformer	1.64e−21	4.29e−20	4.85e−18

respond to the same questions under the same condition. We design a questionnaire containing 680 pairs (375 pairs of Python code snippets and 305 pairs of Assembly code snippets in the test set) where each pair contains an input comment, two code snippets generated by CodeBERT and ExploitGen respectively, and a reference code snippet. We send each post-graduate student a copy of the questionnaire and ask them to evaluate the two code snippets for each comment. An example of the questionnaire can be found in Fig. 8.

Each reviewer was asked to vote for each program, judging whether the semantics of the generated program is correct. When the judgments are consistent, we directly adopt their decision. When they disagree with the judgment, we use the majority-vote strategy. During the review process, reviewers could discuss and resort to external resources (e.g., Wikipedia and Q&A websites). To ensure the fairness of the comparison, reviewers do not know which code snippet is generated by which method.

The results are shown in Fig. 9 and the details are available on the GitHub repository.[12] Here *Total* indicates the total number of lines of code for the program. *Sem* indicates the number of semantic correctness lines in the generated code. *Syn* indicates the number of syntactic correctness lines in the generated code. Based on these results, we find that although our approach achieves high accuracy for code generation tasks at the line-level granularity, there is still much room for improvement for code generation tasks at the granularity of the entire program.

Summary for RQ2

ExploitGen can achieve high accuracy at the line-level granularity but still needs improvement at program-level granularity.

RQ3: What is the impact of different NL lengths and code lengths on the performance of ExploitGen ?. To explore how do the different NL lengths and code lengths effect the performance of ExploitGen, we further analyzed the ROUGE-W metrics score of different NL lengths and code lengths for our proposed approach and the baselines. Fig. 10 presents the average ROUGE-W metrics scores of ExploitGen, CodeBERT, and Transformer for varying NL lengths and code lengths.

As Fig. 10 illustrates, from a holistic point of view, the performance of ExploitGen is better than CodeBERT slightly. Both ExploitGen and CodeBERT are better than Transformer significantly. In summary, ExploitGen can generate higher quality exploit code with shorter NL length and CODE length. For the Python dataset, ExploitGen can generate higher quality exploit code compared to baseline methods when NL length is lower than 40 or when CODE length is lower than 40. For the Assembly dataset, ExploitGen can generate higher quality exploit code compared to baseline methods when NL length is lower than 35 or when CODE length is lower than 50.

Summary for RQ3

ExploitGen can generate higher quality exploit code compared to baseline methods in the case of relatively short lengths of NL and Code.

RQ4: What is the impact of different input formats on the performance of ExploitGen ?. In this RQ, we want to investigate the role of different input formats in the exploit code generation task. We compare the input formats in four different combinations of model components:

Raw $\mapsto$ **Raw.** This input format means that we only use Raw Encoder, treating raw natural language as input and raw code as output.

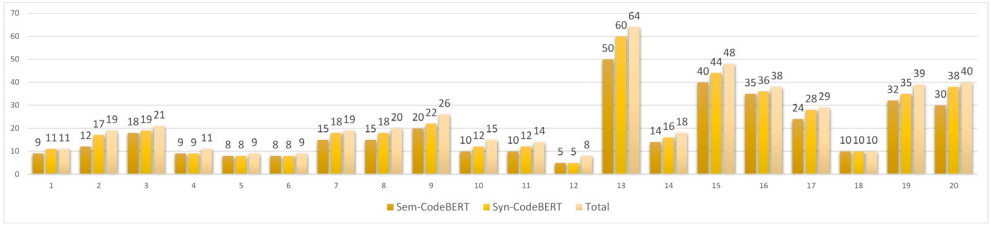
Raw $\mapsto$ Temp. This input format means that we only use Raw Encoder, treating raw natural language as input and template-augmented code as output.

Temp $\mapsto$ **Temp.** This input format means that we only use Temp Encoder, treating template-Augmented natural language as input and template-augmented code as output.

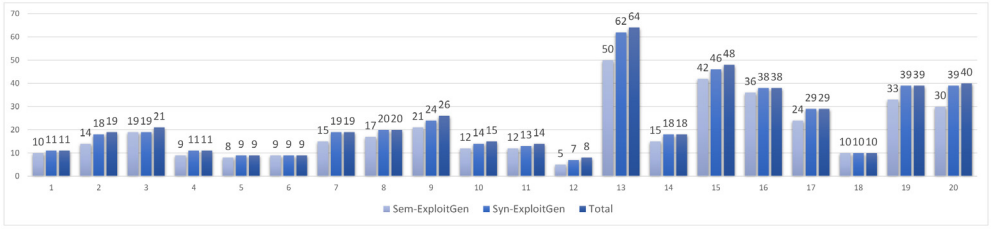
Dual $\mapsto$ **Temp.** This input format means that we use both Raw Encoder and Temp Encoder, treating raw natural language and template-augmented natural language as input and template-augmented code as output.

We show the ablation results in Table 7 from the input format perspective. For the Python data, using Raw $\mapsto$ Temp

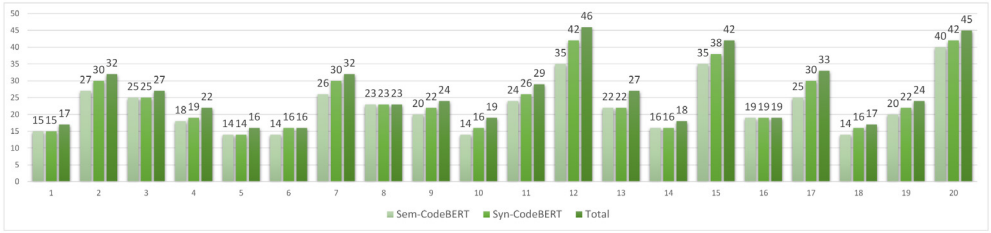
[12] https://github.com/NTDXYG/ExploitGen/tree/main/result/Human



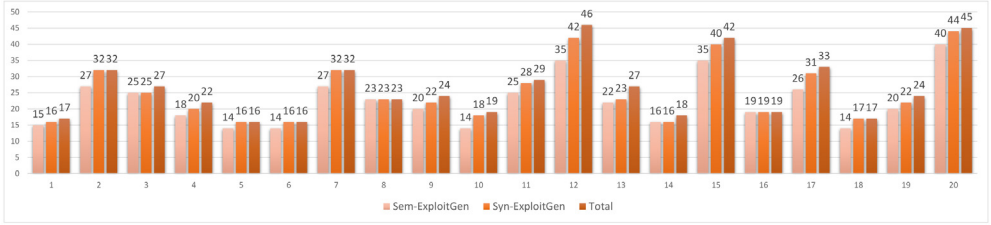
(a) Results of human evaluation in Python dataset generated by CodeBERT



(b) Results of human evaluation in Python dataset generated by ExploitGen



(c) Results of human evaluation in Assembly dataset generated by CodeBERT



(d) Results of human evaluation in Assembly dataset generated by ExploitGen

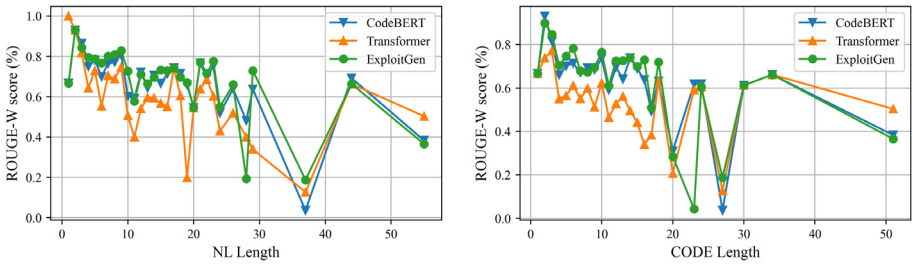
Fig. 9. Results of our proposed method on syntactic correctness and semantic correctness by human evaluation.

Table 7
The ablation results in terms of input formats.

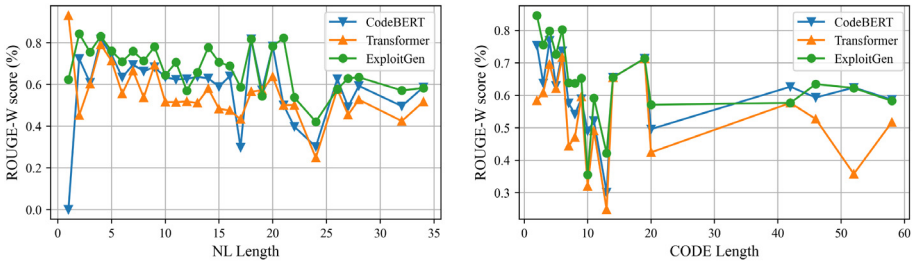
Dataset	Input format	BLEU-4 (%)	ROUGE-W (%)	Accuracy (%)
Python	Raw ↦ Raw	80.57	68.76	56.27
	Raw ↦ Temp	87.83	72.50	62.40
	Temp ↦ Temp	87.56	72.29	61.60
	Dual ↦ Temp	91.27	75.68	70.93
Assembly	Raw ↦ Raw	72.26	66.98	53.11
	Raw ↦ Temp	86.55	73.01	62.95
	Temp ↦ Temp	83.12	69.34	54.09
	Dual ↦ Temp	88.70	74.25	66.89

can achieve 7.26%, 3.74%, and 6.13% improvements in terms of BLEU-4, ROUGE-W, and Exact Match Accuracy respectively. Using Temp ↦ Temp can achieve 6.99%, 3.53%, and 5.33% improvements in terms of BLEU-4, ROUGE-W, and Exact Match Accuracy respectively. Using Dual ↦ Temp can achieve 10.70%, 6.81%, and 14.66% improvements in terms of BLEU-4, ROUGE-W, and Exact Match Accuracy respectively. For the Assembly data, using Raw ↦ Temp can achieve 14.29%, 6.03%, and 9.84% improvements in terms of BLEU-4, ROUGE-W, and Exact Match Accuracy respectively. Using Temp ↦ Temp can achieve 10.86%, 2.36%, and 0.98% improvements in terms of BLEU-4, ROUGE-W, and Exact Match Accuracy respectively. Using Dual ↦ Temp can achieve 16.44%, 7.25%, and 13.78% improvements in terms of BLEU-4, ROUGE-W, and Exact Match Accuracy respectively.

The performance of the model decreases substantially if the parsing replacement is not performed using the template parser;



(a) ROUGE-W scores for different NL lengths and code lengths in Python dataset



(b) ROUGE-W scores for different NL lengths and code lengths in Assembly dataset

Fig. 10. Results of our proposed method and the baselines on ROUGE-W metrics score for varying NL lengths and code lengths.

Table 8
The ablation results in terms of model components.

Dataset	SEM	FUS	BLEU-4 (%)	ROUGE-W (%)	Accuracy (%)
Python	–	–	89.27	73.16	61.75
	✓	–	89.65	73.24	63.04
	–	✓	90.00	74.75	69.07
	✓	✓	91.27	75.68	70.93
Assembly	–	–	83.52	70.24	60.52
	✓	–	84.86	71.52	62.45
	–	✓	87.59	74.16	66.23
	✓	✓	88.70	74.25	66.89

the performance of the model also decreases if both NL and code are not replaced. Therefore, the results show the effectiveness of our proposed Rule-based template parser and Dual Encoders.

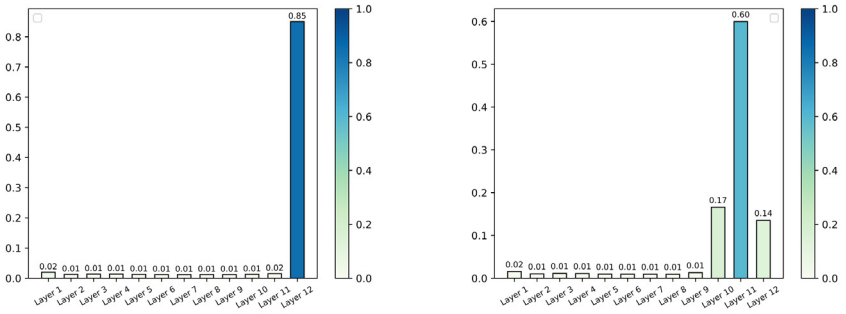
Summary for RQ4

In terms of input formats, using both raw NL and template-augmented NL can have a positive impact on performance. The performance of the model can be effectively improved by using both raw NL and template-augmented NL.

RQ5: What is the impact of Semantic Attention Layer and Fusion Layer on the performance of ExploitGen?. In this RQ, we want to investigate the role of *Semantic Attention Layer* (SEM) and *Fusion Layer* (FUS) in the exploit code generation task. We compare the model performance in the following four different combinations.
Without SEM and FUS. This combination consists of Raw Encoder, Temp Encoder, and Decoder Layer. We extract the two semantic vectors of the last layer of the two encoders and superimpose the two semantic vectors before inputting them to Decoder Layer for decoding.
Only With SEM. This combination consists of Raw Encoder, Temp Encoder, *Semantic Attention Layer*, and Decoder Layer. We add *Semantic Attention Layer* to the semantic vector extraction, and then superimpose the two semantic vectors before inputting them to Decoder Layer for decoding.
Only With FUS. This combination consists of Raw Encoder, Temp Encoder, *Fusion Layer*, and Decoder Layer. We extract the two semantic vectors of the last layer of the two encoders and then fusion the two semantic vectors before inputting them to Decoder Layer for decoding.
With SEM and FUS. This combination consists of Raw Encoder, Temp Encoder, *Semantic Attention Layer*, *Fusion Layer*, and Decoder Layer. We apply *Semantic Attention Layer* to the semantic vector extraction and then fusion the two semantic vectors before inputting them to Decoder Layer for decoding.

We show the ablation results from the model component perspective in Table 8. For the Python data, using *Semantic Attention Layer* alone can achieve 0.38%, 0.08%, and 1.29% improvements in terms of BLEU-4, ROUGE-W, and Exact Match Accuracy respectively. Using *Fusion Layer* alone can achieve 0.73%, 1.59%, and 7.32% improvements in terms of BLEU-4, ROUGE-W, and Exact Match Accuracy respectively. Using both *Semantic Attention Layer* and *Fusion Layer* can achieve 2.00%, 2.41%, and 9.18% improvements in terms of BLEU-4, ROUGE-W, and Exact Match Accuracy respectively. For the Assembly data, using *Semantic Attention Layer* alone can achieve 1.34%, 1.28%, and 1.93% improvements in terms of BLEU-4, ROUGE-W, and Exact Match Accuracy respectively. Using *Fusion Layer* alone can achieve 4.07%, 3.92%, and 5.71% improvements in terms of BLEU-4, ROUGE-W, and Exact Match Accuracy respectively. Using both *Semantic Attention Layer* and *Fusion Layer* can achieve 5.18%, 4.02%, and 6.37% improvements in terms of BLEU-4, ROUGE-W, and Exact Match Accuracy respectively.

To get a more intuitive view of the motivation and effectiveness of the Semantic Attention Layer, we take a sample from the python dataset as an example. The raw NL of this sample is



(a) Visualization of layer weights in Raw Encoder (b) Visualization of layer weights in Temp Encoder

Fig. 11. Visualization of layer weights in Semantic Attention Layer.

Table 9
The ablation results in terms of training strategies.

Dataset	APT	TST	BLEU-4 (%)	ROUGE-W (%)	Accuracy (%)
Python	–	–	90.10	74.43	66.93
	✓	–	90.56	74.89	67.47
	–	✓	90.33	74.86	68.80
	✓	✓	91.27	75.68	70.93
Assembly	–	–	84.20	71.57	60.98
	✓	–	85.84	72.75	62.00
	–	✓	86.61	73.06	64.34
	✓	✓	88.70	74.25	66.89

"get the hexadecimal value of suplX and reverse its order then store the value in rev_suplx", and the corresponding temp NL is "get hexadecimal value of var0 and reverse its order then store value in var1". We visualize the weights learned by the Semantic Attention Layer in Raw Encoder and Temp Encoder respectively, and the results are shown in Fig. 11. We can observe that for Raw Encoder, more attention is paid to the semantic information learned in the last layer; while for Temp Encoder, more attention is paid to the information learned in the last three layers, especially more sensitive to the information learned in the penultimate layer.

Summary for RQ5

In terms of model components, *Semantic Attention Layer* and *Fusion Layer* both have a positive impact on performance. The performance of the model can be effectively improved by using both *Semantic Attention Layer* and *Fusion Layer*.

RQ6: What is the impact of Adaptive Pretraining and Two-Stage Training on the performance of ExploitGen?. In this RQ, we want to investigate the role of *Adaptive Pretraining* (APT) and *Two-Stage Training* (TST) in the exploit code generation task. We compare the model performance in four training strategies:

Without APT and TST. This training strategy means that the model is fine-tuned directly on CodeBERT without additional processing when training it.

Only With APT. This training strategy means that we use the *Adaptive Pretraining*. Specifically, we first continue the adaptation of the model with a self-supervised pre-training approach on the domain-related and task-related datasets for CodeBERT to obtain FG-CodeBERT, then the model is fine-tuned on FG-CodeBERT.

Only With TST. This training strategy means that we use the *Two-Stage Training*. We first perform training on Raw Encoder and Temp Encoder respectively based on the parameters of CodeBERT to better initialize the corresponding parameters. Then all parameters of the model are updated for training.

With APT and TST. This training strategy means that we use both *Adaptive Pretraining* and *Two-Stage Training*. We first perform the training on Raw Encoder and Temp Encoder respectively based on the parameters of FG-CodeBERT. Then we update all parameters of the model.

We show the ablation results in Table 9 from the model training perspective. For the Python data, using *Adaptive Pretraining* alone can achieve 0.46%, 0.45%, and 0.54% improvements in terms of BLEU-4, ROUGE-W, and Exact Match Accuracy respectively. Using *Two-Stage Training* alone can achieve 0.23%, 0.43%, and 1.87% improvements in terms of BLEU-4, ROUGE-W, and Exact Match Accuracy respectively. Using both *Adaptive Pretraining* and *Two-Stage Training* can achieve 1.17%, 1.14%, and 4.00% improvements in terms of BLEU-4, ROUGE-W, and Exact Match Accuracy respectively. For the Assembly data, using *Adaptive Pretraining* alone can achieve 1.64%, 1.18%, and 1.02% improvements in terms of BLEU-4, ROUGE-W, and Exact Match Accuracy respectively. Using *Two-Stage Training* alone can achieve 2.41%, 1.49%, and 3.36% improvements in terms of BLEU-4, ROUGE-W, and Exact Match Accuracy respectively. Using both *Adaptive Pretraining* and *Two-Stage Training* can achieve 4.50%, 2.69%, and 5.91% improvements in terms of BLEU-4, ROUGE-W, and Exact Match Accuracy respectively.

Our results show that adding Adaptive Pretraining and Two-Stage Training can lead to better performance when using pre-trained language models for downstream tasks.

Summary for RQ6

Adaptive Pretraining and *Two-Stage Training* can have a positive impact on performance of ExploitGen. The performance of the model can be effectively improved by using both *Adaptive Pretraining* and *Two-Stage Training*.

5. Discussion

As shown in our empirical study and human study, our model can achieve better performance in generating exploit code. In

Table 10
Semantical errors with corresponding examples, each example with the corresponding comment (NL), the ground truth exploit code, and the exploit code generated by ExploitGen.

Error type	Example
Slice Error	**NL:** nbits is the second element of sys.argv converted to integer **Ground Truth:** nbits = int(sys.argv[2]) **ExploitGen:** nbits = int(sys.argv[1])
Placeholder Error	**NL:** concatenate the string '\x1c\x0f\x88\xdf\x88\xd0\x30\xd8' to decoder_stub **Ground Truth:** decoder_stub += '\x1c\x0f\x88\xdf\x88\xd0\x30\xd8' **ExploitGen:** var3 += '\x1c\x0f\x88\xdf\x88\xd0\x30\xd8'
Variable Name Error	**NL:** Take the absolute value of subfs then convert subfs to an integer, then cast to a hexadecimal, slice the variable rev_suplx between the indicies 0 and 2 then cast rev_suplx to the type int16, store the value of the summation in the variable xxx. **Ground Truth:** xxx = hex(int(abs(subfs)) + int(rev_suplx[0:2],16)) **ExploitGen:** temp = hex(int(abs(subfs)) + int(rev_suplx[0:2],16))
Variable Value Error	**NL:** W is a string '\033[0m' **Ground Truth:** W = '\033[0m' **ExploitGen:** W = '\x1b[0m'

this section, we want to further investigate when ExploitGen generates the wrong exploit code. To explore the limitations of ExploitGen, we count all the samples of semantical errors in the generated exploit code and manually summarize the error code generated by ExploitGen into four error types, which are shown in Table 10.

Slice Error. We find that this type of semantically errors is caused by the Template Parser, for example, the input NL is *nbits is the second element*, but the model output is *sys.argv[1]*. For this type of error, we attribute it to the fact that when generating template-augmented NL, we just template the numbers and do not handle English words like 'first'/'second' very well.

Placeholder Error. We find that this type of the semantically errors is caused by the uncontrollability of the deep learning model, which is due to the fact that ExploitGen is controlled by probabilities when generating the code. For example, the input NL is *concatenate the string '\x1c\x0f\x88\xdf\x88\xd0\x30\xd8' to decoder_stub*, the *slot_map* is '\x1c\x0f\x88\xdf\x88\xd0\x30\xd8': 'var0', 'decoder_stub': 'var1', but generated code contains *var3*. This causes ExploitGen to be unable to restore var3.

Variable Name Error and **Variable Value Error.** We find that these two types of semantical errors are caused by both Template Parser and the model. One reason is that the Template Parser is still deficient in template generation, and another reason is that the variable names or variable values appear in the input NL, but the model does not output correctly.

6. Threats to validity

Internal threats. The first internal threat is the potential defects in the implementation of our proposed method. To alleviate this threat, we first check code carefully and use mature libraries, such as PyTorch and Transformers. The second internal threat is the implementation of the baseline methods. To alleviate this threat, we try our best to re-implement their approach(i.e., Seq2Seq and Transformer we implement them by OpenNMT,[13] DualSC,[14] CodeBERT,[15] CodeGPT, and CodeGPT-adapted[16] we implement them by their shared script).

External threats. The main external threat is the representativeness of the choice datasets. To alleviate this threat, we select the two popular corpora provided by Liguori et al. (2021b) and the quality of these two corpora is guaranteed. Both datasets are extracted from real-world vulnerability code repositories, and Liguori et al. (2021b) manually comment the missing commented code data by hand to ensure the high quality of these two datasets. Another threat is the quality of the human study. To alleviate this threat, we hire independent reviewers who have expertise in coding with Python and Assembly languages. They are encouraged to discuss and access relevant information through the Internet for unfamiliar concepts. To ensure the fairness of the comparison, the source information of the code snippet was not exposed.

Construct validity. The first construct validity relates to the suitability of our evaluation measures. To alleviate this construct threat, we use three evaluation measures, namely *BLEU*, *ROUGE-W* and *Exact Match Accuracy*. Moreover, we also conduct a human study and compute the *p*-value by using the Wilcoxon signed-rank test to further verify the effectiveness of our proposed hybrid approach. The second construct validity is that we do not consider whether the code generated by ExploitGen can be a valid attack on the program. On the one hand, this is because we focus on exploit code generation at the line-level granularity. This can make it easier for software maintainers to write exploit code to test their programs when they are unfamiliar with certain instructions. On the other hand, because the data set is collected from existing vulnerability repositories, most of the previously attacked programs have already been fixed, so there are not enough flawed programs for us to test. Therefore, we mainly evaluate the effectiveness of ExploitGen by considering the syntactic correctness and semantic correctness of the generated exploit code.

7. Related work

7.1. Research on code generation

Code generation, as an important topic in the field of mining software repositories, has received a lot of attention from researchers. Recently, most of the studies followed deep learning-based methods (i.e., encoder–decoder framework) and achieved promising results. Mou et al. (2015) first proposed to generate the corresponding code snippets from the user's natural language intent descriptions by using deep learning, where the dataset

[13] https://github.com/OpenNMT/OpenNMT-py
[14] https://github.com/NTDXYG/DualSC
[15] https://github.com/microsoft/CodeBERT/tree/master/CodeBERT/code2nl
[16] https://github.com/microsoft/CodeXGLUE/tree/main/Text-Code/text-to-code/code

was collected from a pedagogical programming online judge (OJ) system. They constructed a standard encoder–decoder architecture through RNN models to investigate whether neural models can synthesize executable, functionally coherent programs. Ling et al. (2016) used the LSTM model to construct the encoder–decoder architecture, which combines character-level softmax for generating code tokens and pointer networks (See et al., 2017) to copy key code from the natural language. Also, they proposed a new corpus for card games code, such as Magic the Gathering (MTG) and Hearthstone (HS). Rabinovich et al. (2017) proposed a novel neural network, Abstract Syntax Network(ASN), which uses the Abstract Syntax Description Language(ADSL) to limit the neural network's ability to generate syntax trees and then use the syntax trees to generate the corresponding code. They used the LSTM model and attention mechanism to construct the ASN, and experiment on the HS dataset. Yin and Neubig (2018) proposed TRANX, which is based on ASDL, LSTM, attention mechanism, and copy mechanism. TRANX turns an AST into a general-purpose intermediate(MR) through an encoder–decoder architecture for a given natural language, then parses the MR into a corresponding code according to domain-specific grammar advice. Hayati et al. (2018) proposed RECODE based on TRANX, which integrated information retrieval methods into the Seq2Seq model. The retrieval component improved the performance of a previously state-of-the-art model, according to their research. Wei et al. (2019) formalized the code generation task and the code summarization task as dual tasks, and they propose a dual training framework to simultaneously learn the duality between code and comments. To strengthen the duality, they created a novel restriction on attention mechanism based on LSTM-based encoder–decoder architecture. Sun et al. (2019) presented a grammar-based structural convolutional neural network(CNN) for code generation that generated a program by predicting the programming language's grammar rules. Hussain et al. (2020b) first proposed a transfer learning-based approach that significantly improves the performance of deep learning-based source code models and later proposed the CodeGRU (Hussain et al., 2020a) to better model the source code. Hussain et al. (2021) proposed a deep semantic net (DeepSN) that makes use of semantic information of the source code. DeepSN uses an enhanced hierarchical CNN to learn useful semantic information and uses LSTM to capture the source code's long and short-term context dependencies.

With the introduction of the Transformer, researchers began to apply the Transformer model to code generation tasks. Sun et al. (2020) proposed a novel tree-based neural architecture, TreeGen. TreeGen employs Transformers' self-attention mechanism to handle the long-dependency problem, as well as a novel AST reader (encoder) to include grammar rules and AST structures into the model. Gemmell et al. (2020) presented the Relevance Transformer, a novel model that uses pseudo-relevance feedback to incorporate external knowledge. The Relevance Transformer forces diversity while biasing the decoding process to be similar to previously received code (copy mechanism). Norouzi et al. (2021) used CodeBERT as the encoder and use a 4-layer transformer decoder, In addition, they toke copy attention into account in the model. Ahmad et al. (2021) proposed the unified pre-training model for program understanding and generation, PLBART. PLBART was built on the Transformer architecture and took advantage of the unlabeled data in PL and NL by using the denoising pre-training work from BART (Lewis et al., 2020). PLBART fed noisy sequences into the encoder, and the decoder outputs the original sequences to remove the noise. Phan et al. (2021) proposed CoTexT, a pre-trained, transformer-based encoder–decoder model that learns the representational context between natural language and programming language. CoTexT was based on the T5 architecture (Raffel et al., 2020) and pre-trained on large programming language corpora that are used by self-supervision methods to gain general language and code knowledge. Wang et al. (2021) proposed CodeT5, an encoder–decoder model based on pre-trained transformers that take token type information into account. CodeT5 is based on the T5 architecture, which uses denoising sequence-to-sequence pre-training and has been demonstrated to improve natural language understanding and generation.

Different from the aforementioned work, our approach considers features specific to the exploit code and utilizes corresponding encoders, as well as attention mechanisms, to capture such features in the code generation process.

7.2. Research on exploit code generation

For the specific exploit code generation domain, it is in a fledgling state as the labeled data set is very limited. Liguori et al. (2021a) were the first to Assemble and release the Shellcode_IA32, which contains challenging but common Assembly instructions together with their natural language descriptions. After that, they experimented with standard methods in neural machine translation to establish baseline performance levels on shellcode generation task. Later, Liguori et al. (2022) conducted a large-scale empirical study on this dataset and proposed a new metric for evaluating NMT's accuracy in generating shellcode. The empirical analysis showed that NMT can generate Assembly code from natural language with excellent accuracy and in many cases can generate complete shell code without errors. Yang et al. (2022) formalized the shellcode generation and summarization as a dual task, constructed a shallow Transformer for model construction. To adapt these low-resource tasks, they designed a normalization method Adjust_QKNorm. Finally, to alleviate the problem of out-of-vocabulary, they proposed a rule-based repair component to improve the performance of shellcode generation. Liguori et al. (2021b) extended the Shellcode IA32 dataset and presented a method (EVIL) for generating exploit code in Assembly/Python from natural language descriptions. In our paper, we also conduct empirical studies on their released dataset.

Compared with previous exploit code generation studies that were largely based on deep learning, our approach complements the data-driven methods with rule-based template information and achieves promising results.

8. Conclusion and future work

In this paper, we propose the template-augmented exploit code generation method ExploitGen based on CodeBERT. The results of the automated evaluation show that our proposed method ExploitGen outperforms the state-of-the-art baselines from the previous studies of automatic code generation on the two corpora. Moreover, we also verify the input lengths, input formats rationality, training methods rationality, and model components rationality of ExploitGen by designing a set of ablation experiments. We design an additional human study to evaluate the quality of the generated code in terms of syntactic correctness and semantic correctness.

There are a number of applications of exploit code. For instance, for security assessment writing exploit code is indispensable, but its manual development turns out to be costly. Our approach can accelerate this process and reduce the cost for practitioners. Moreover, researchers can adopt our approach to generate exploit code which can facilitate the research of detection and defense of exploit code.

For future work, we plan to improve the performance of the Template Parser, for example by extracting the numeric information contained in words. We also plan to implement our

approach as a plugin in mainstream IDEs to provide better usability. Moreover, our proposed approach is mainly designed for the automated generation of exploit code, but it can easily be extended to other domain-specific code generation tasks, such as Shell (Lin et al., 2018) and SQL (Yu et al., 2018), since these tasks used similar corpora as ours in terms of the granularity level, and the comments in these corpora also contain a large number of domain-specific tokens.

CRediT authorship contribution statement

Guang Yang: Data curation, Software, Writing – original draft. **Yu Zhou:** Conceptualization, Methodology, Writing – review & editing, Supervision. **Xiang Chen:** Data curation, Software, Validation. **Xiangyu Zhang:** Software, Validation. **Tingting Han:** Writing – review & editing. **Taolue Chen:** Writing – review & editing.

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 is supported by the National Natural Science Foundation of China (No. 61972197), the Natural Science Foundation of Jiangsu Province (No. BK20201292), the Collaborative Innovation Center of Novel Software Technology and Industrialization, and the Open Project of Key Laboratory of Safety-Critical Software for Nanjing University of Aeronautics and Astronautics, Ministry of Industry and Information Technology (No. NJ2020022). T. Chen is partially supported by an oversea grant from the State Key Laboratory of Novel Software Technology, Nanjing University (KFKT2022A03), Birkbeck BEI School Project (EFFECT) and National Natural Science Foundation of China (No. 62272397).

References

Ahmad, W., Chakraborty, S., Ray, B., Chang, K.-W., 2021. Unified pre-training for program understanding and generation. In: Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. pp. 2655–2668.

Arce, I., 2004. The shellcode generation. IEEE Secur. Priv. 2 (5), 72–76.

Bahdanau, D., Cho, K., Bengio, Y., 2014. Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473.

Bao, T., Wang, R., Shoshitaishvili, Y., Brumley, D., 2017. Your exploit is mine: Automatic shellcode transplant for remote exploits. In: 2017 IEEE Symposium on Security and Privacy. SP, IEEE, pp. 824–839.

Budzianowski, P., Vulić, I., 2019. Hello, it's GPT-2-how can I help you? Towards the use of pretrained language models for task-oriented dialogue systems. In: Proceedings of the 3rd Workshop on Neural Generation and Translation. pp. 15–22.

Clark, K., Luong, M.-T., Le, Q.V., Manning, C.D., 2020. Electra: Pre-training text encoders as discriminators rather than generators. arXiv preprint arXiv:2003.10555.

Devlin, J., Chang, M.-W., Lee, K., Toutanova, K., 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. Vol. 1. Long and Short Papers, pp. 4171–4186.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al., 2020. CodeBERT: A pre-trained model for programming and natural languages. In: Findings of the Association for Computational Linguistics: EMNLP 2020. pp. 1536–1547.

Frank, C., 2000. Making plain binary files using a C compiler (i386+).

Freitag, M., Al-Onaizan, Y., 2017. Beam search strategies for neural machine translation. In: Proceedings of the First Workshop on Neural Machine Translation. pp. 56–60.

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. pp. 2005–2008.

Gururangan, S., Marasović, A., Swayamdipta, S., Lo, K., Beltagy, I., Downey, D., Smith, N.A., 2020. Don't stop pretraining: Adapt language models to domains and tasks. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. pp. 8342–8360.

Hayati, S.A., Olivier, R., Avvaru, P., Yin, P., Tomasic, A., Neubig, G., 2018. Retrieval-based neural code generation. In: Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing. pp. 925–930.

Heyman, G., Huysegems, R., Justen, P., Van Cutsem, T., 2021. Natural language-guided programming. In: Proceedings of the 2021 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. pp. 39–55.

Hindle, A., Barr, E.T., Gabel, M., Su, Z., Devanbu, P., 2016. On the naturalness of software. Commun. ACM 59 (5), 122–131.

Hu, X., Chen, Q., Wang, H., Xia, X., Lo, D., Zimmermann, T., 2022. Correlating automated and human evaluation of code documentation generation quality. ACM Trans. Softw. Eng. Methodol. (TOSEM) 31 (4), 1–28.

Hu, X., Li, G., Xia, X., Lo, D., Jin, Z., 2020. Deep code comment generation with hybrid lexical and syntactical information. Empir. Softw. Eng. 25 (3), 2179–2217.

Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., Brockschmidt, M., 2019. Codesearchnet challenge: Evaluating the state of semantic code search. arXiv preprint arXiv:1909.09436.

Hussain, Y., Huang, Z., Zhou, Y., 2021. Improving source code suggestion with code embedding and enhanced convolutional long short-term memory. IET Softw. 15 (3), 199–213.

Hussain, Y., Huang, Z., Zhou, Y., Wang, S., 2020a. CodeGRU: Context-aware deep learning with gated recurrent unit for source code modeling. Inf. Softw. Technol. 125, 106309.

Hussain, Y., Huang, Z., Zhou, Y., Wang, S., 2020b. Deep transfer learning for source code modeling. Int. J. Softw. Eng. Knowl. Eng. 30 (05), 649–668.

Iyer, S., Konstas, I., Cheung, A., Zettlemoyer, L., 2018. Mapping language to code in programmatic context. In: Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing. pp. 1643–1652.

Jawahar, G., Sagot, B., Seddah, D., 2019. What does BERT learn about the structure of language? In: ACL 2019-57th Annual Meeting of the Association for Computational Linguistics.

Karmakar, A., Robbes, R., 2021. What do pre-trained code models know about code? In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 1332–1336.

Kulal, S., Pasupat, P., Chandra, K., Lee, M., Padon, O., Aiken, A., Liang, P.S., 2019. Spoc: Search-based pseudocode to code. Adv. Neural Inf. Process. Syst. 32.

Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoyanov, V., Zettlemoyer, L., 2020. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In: Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics. pp. 7871–7880.

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. NLP4Prog 2021, pp. 58–64.

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

Liguori, P., Al-Hossami, E., Orbinato, V., Natella, R., Shaikh, S., Cotroneo, D., Cukic, B., 2021b. EVIL: exploiting software via natural language. In: 2021 IEEE 32nd International Symposium on Software Reliability Engineering. ISSRE, IEEE, pp. 321–332.

Lin, C.-Y., 2004. Rouge: A package for automatic evaluation of summaries. In: Text Summarization Branches Out. pp. 74–81.

Lin, X.V., Wang, C., Zettlemoyer, L., Ernst, M.D., 2018. NL2bash: A corpus and semantic parser for natural language interface to the linux operating system. In: Proceedings of the Eleventh International Conference on Language Resources and Evaluation. LREC 2018.

Ling, W., Blunsom, P., Grefenstette, E., Hermann, K.M., Kočiskỳ, T., Wang, F., Senior, A., 2016. Latent predictor networks for code generation. In: Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics. Volume 1: Long Papers, pp. 599–609.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., Stoyanov, V., 2019. Roberta: A robustly optimized bert pretraining approach. arXiv preprint arXiv:1907.11692.

Liu, Z., Xia, X., Hassan, A.E., Lo, D., Xing, Z., Wang, X., 2018. Neural-machine-translation-based commit message generation: how far are we? In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. pp. 373–384.

Locascio, N., Narasimhan, K., DeLeon, E., Kushman, N., Barzilay, R., 2016. Neural generation of regular expressions from natural language with minimal domain knowledge. In: Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing. pp. 1918–1923.

Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., et al., 2021. Codexglue: A machine learning benchmark dataset for code understanding and generation. arXiv preprint arXiv:2102.04664.

Mou, L., Men, R., Li, G., Zhang, L., Jin, Z., 2015. On end-to-end program generation from user intention by deep neural networks. arXiv preprint arXiv:1510.07211.

Norouzi, S., Tang, K., Cao, Y., 2021. Code generation from natural language with less prior knowledge and more monolingual data. In: Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing. Volume 2: Short Papers, Association for Computational Linguistics, pp. 776–785. http://dx.doi.org/10.18653/v1/2021.acl-short.98, Online. URL https://aclanthology.org/2021.acl-short.98.

Oda, Y., Fudaba, H., Neubig, G., Hata, H., Sakti, S., Toda, T., Nakamura, S., 2015. Learning to generate pseudo-code from source code using statistical machine translation. In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering. ASE, IEEE, pp. 574–584.

Papineni, K., Roukos, S., Ward, T., Zhu, W.-J., 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.

Phan, L., Tran, H., Le, D., Nguyen, H., Annibal, J., Peltekian, A., Ye, Y., 2021. CoTexT: Multi-task learning with code-text transformer. In: Proceedings of the 1st Workshop on Natural Language Processing for Programming. NLP4Prog 2021, pp. 40–47.

Rabinovich, M., Stern, M., Klein, D., 2017. Abstract syntax networks for code generation and semantic parsing. In: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics. Volume 1: Long Papers, pp. 1139–1149.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., Liu, P.J., 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. J. Mach. Learn. Res. 21 (140), 1–67.

See, A., Liu, P.J., Manning, C.D., 2017. Get to the point: Summarization with pointer-generator networks. In: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics. Volume 1: Long Papers, pp. 1073–1083.

Sun, Z., Zhu, Q., Mou, L., Xiong, Y., Li, G., Zhang, L., 2019. A grammar-based structural cnn decoder for code generation. In: Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 33. (01), pp. 7055–7062.

Sun, Z., Zhu, Q., Xiong, Y., Sun, Y., Mou, L., Zhang, L., 2020. Treegen: A tree-based transformer architecture for code generation. In: Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 34. (05), pp. 8984–8991.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I., 2017. Attention is all you need. In: Advances in Neural Information Processing Systems. pp. 5998–6008.

Wang, C., Cho, K., Gu, J., 2020. Neural machine translation with byte-level subwords. In: Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 34. (05), pp. 9154–9160.

Wang, Y., Li, X.-H., Guan, L., Cui, B.-J., 2010. Attack and defending technology of shellcode. Comput. Eng. 36 (18), 165–168.

Wang, Z., Mayhew, S., Roth, D., et al., 2019. Cross-lingual ability of multilingual bert: An empirical study. arXiv preprint arXiv:1912.07840.

Wang, Y., Wang, W., Joty, S., Hoi, S.C., 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing. pp. 8696–8708.

Wei, B., Li, G., Xia, X., Fu, Z., Jin, Z., 2019. Code generation as a dual task of code summarization. In: Advances in Neural Information Processing Systems. pp. 6563–6573.

Weiss, K., Khoshgoftaar, T.M., Wang, D., 2016. A survey of transfer learning. J. Big Data 3 (1), 1–40.

Wiseman, S., Rush, A.M., 2016. Sequence-to-sequence learning as beam-search optimization. In: Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing. pp. 1296–1306.

Xu, F.F., Vasilescu, B., Neubig, G., 2022. In-ide code generation from natural language: Promise and challenges. ACM Trans. Softw. Eng. Methodol. (TOSEM) 31 (2), 1–47.

Yang, G., Chen, X., Cao, J., Xu, S., Cui, Z., Yu, C., Liu, K., 2021a. ComFormer: Code comment generation via transformer and fusion method-based hybrid code representation. In: 2021 8th International Conference on Dependable Systems and their Applications. DSA, IEEE, pp. 30–41.

Yang, G., Chen, X., Zhou, Y., Yu, C., 2022. DualSC: Automatic generation and summarization of shellcode via transformer and dual learning. arXiv preprint arXiv:2202.09785.

Yang, R., Zhang, J., Gao, X., Ji, F., Chen, H., 2019. Simple and effective text matching with richer alignment features. In: Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics. pp. 4699–4709.

Yang, G., Zhou, Y., Chen, X., Yu, C., 2021b. Fine-grained pseudo-code generation method via code feature extraction and transformer. arXiv preprint arXiv:2102.06360.

Yin, P., Deng, B., Chen, E., Vasilescu, B., Neubig, G., 2018. Learning to mine aligned code and natural language pairs from stack overflow. In: 2018 IEEE/ACM 15th International Conference on Mining Software Repositories. MSR, IEEE, pp. 476–486.

Yin, P., Neubig, G., 2017. A syntactic neural model for general-purpose code generation. In: Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics. Volume 1: Long Papers, pp. 440–450.

Yin, P., Neubig, G., 2018. TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation. In: Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations. pp. 7–12.

Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., Ma, J., Li, I., Yao, Q., Roman, S., et al., 2018. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-SQL task. In: Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing. pp. 3911–3921.

Zhou, X., Han, D., Lo, D., 2021. Assessing generalizability of CodeBERT. In: 2021 IEEE International Conference on Software Maintenance and Evolution. ICSME, IEEE, pp. 425–436.

Guang Yang is currently pursuing the Ph.D. degree with the College of Computer Science and Technology of Nanjing University of Aeronautics and Astronautics. His research interests include code generation and exploit code.

Yu Zhou is currently a full professor in the College of Computer Science and Technology at Nanjing University of Aeronautics and Astronautics (NUAA). He received his BSc degree in 2004 and PhD degree in 2009, both in Computer Science from Nanjing University China. Before joining NUAA in 2011, he conducted PostDoc research on software engineering at Politechnico di Milano, Italy. From 2015-2016, he visited the SEAL lab at University of Zurich Switzerland, where he is also an adjunct researcher. His research interests mainly include software evolution analysis, mining software repositories, software architecture, and reliability analysis. He has been supported by several national research programs in China.

Xiang Chen received the B.Sc. degree in information management and system from Xi'an Jiaotong University, China in 2002. Then he received the M.Sc., and Ph.D. degrees in computer software and theory from Nanjing University, China in 2008 and 2011 respectively. He is with the School of Information Science and Technology at Nantong University as an associate professor. His research interests are mainly in software engineering. In particular, he is interested in empirical software engineering, mining software repositories, software maintenance and software testing. In these areas, he has published over 60 papers in refereed journals or conferences, such as IEEE Transactions on Software Engineering, Information and Software Technology, Journal of Systems and Software, IEEE Transactions on Reliability, Journal of Software: Evolution and Process, Software Quality Journal, Journal of Computer Science and Technology, ASE, ICSME, SANER and COMPSAC. He is a senior member of CCF, China, and a member of IEEE and ACM.

Xiangyu Zhang is currently pursuing the master degree with the College of Computer Science and Technology of Nanjing University of Aeronautics and Astronautics. His research interests include code generation.

Tingting Han obtained her B.Sc. and MEng in Computer Science from Nanjing University China, and her Ph.D. from RWTH Aachen University and University of Twente. She was a Research Assistant at University of Oxford before joining Birkbeck. Her Areas of interest: Formal verification and synthesis of probabilistic systems, and its applications.

Taolue Chen is a Postdoctoral Researcher at University of Oxford (UK) and University of Twente (The Netherlands);Ph.D. (CWI and Vrije Universiteit Amsterdam, The Netherlands), Master and Bachelor (Nanjing University, China), all in Computer Science. His Areas of interest: Quantitative analysis and synthesis of computer program and systems, logic in computer science, machine learning and data science, software engineering, algorithms and computational complexity.