EVIL: Exploiting Software via Natural Language

Pietro Liguori*, Erfan Al-Hossami†, Vittorio Orbinato*,
Roberto Natella*, Samira Shaikh†, Domenico Cotroneo*, and Bojan Cukic†
*University of Naples Federico II, Naples, Italy
{pietro.liguori, vittorio.orbinato, roberto.natella, cotroneo}@unina.it
†University of North Carolina at Charlotte, Charlotte, NC
{ealhossa, samirashaikh, bcukic}@uncc.edu

2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE) | 978-1-6654-2587-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISSRE52982.2021.00042

***Abstract*—Writing exploits for security assessment is a challenging task. The writer needs to master programming and obfuscation techniques to develop a successful exploit. To make the task easier, we propose an approach (*EVIL*) to automatically generate exploits in assembly/Python language from descriptions in natural language. The approach leverages Neural Machine Translation (NMT) techniques and a dataset that we developed for this work. We present an extensive experimental study to evaluate the feasibility of *EVIL*, using both automatic and manual analysis, and both at generating individual statements and entire exploits. The generated code achieved high accuracy in terms of syntactic and semantic correctness.**

***Index Terms*—Automatic Exploit Generation, Neural Machine Translation, Software Exploits, Shellcode, Encoder, Decoder**

I. Introduction

In the context of software security, a solid understanding of offensive techniques is increasingly important [1], [2]. Well-intentioned actors, such as penetration testers, ethical hackers, researchers, and computer security teams are engaged in developing exploits, referred to as *proof-of-concept* (POC), to reveal security weaknesses within the software. Offensive security helps us understand how attackers take advantage of vulnerabilities and motivates vendors and users to patch them to prevent attacks [3]. Among software exploits, code-injection attacks are the trickiest. They allow the attacker to inject and execute arbitrary code on the victim system. Since the injected code frequently launches a command shell, the hacking community refers to the payload portion of a code-injection attack as a *shellcode* [4].

Writing code injection exploits is a challenging task since it requires significant technical skills. Shellcodes are typically written in assembly language, affording the attacker full control of the memory layout and CPU registers to attack low-level mechanisms (e.g., heap metadata and stack return addresses) not otherwise accessible through high-level programming languages. Another challenge for shellcodes is modern antivirus (AV) and intrusion detection systems (IDS), which actively look for malicious payloads to block attacks. To elude detection, shellcode writers weaponize their shellcode by implementing an encoding/decoding strategy. In other words, writers have to develop *encoders* (typically, using Python) to obfuscate the original shellcode without altering its functionality, and *decoders* (typically in assembly language, as the shellcode) to revert to the payload once it is loaded (and then executed) on the victim system.

In this work, we propose an approach, *EVIL* (Exploiting software VIa natural Language), for exploit writing based on natural language processing. The approach aims to support both beginners and experienced researchers, by making exploits easier to create and flattening the learning curve. In *EVIL*, a machine learning system learns about exploit writing from a dataset, containing both real exploits and their description in the English language. Then, the writer describes the exploit using the English language and lets the machine learning system translate the description into assembly and Python code. *EVIL* leverages recent advances in *neural machine translation* (NMT) to automatically generate code from natural language descriptions using recurrent neural networks. NMT has emerged as a promising machine translation approach, and it is widely recognized as the state-of-the-art method for the translation of different languages [5], [6]. NMT has been adopted in many different areas, to generate programs in the Python language [7], [8], OS commands for the UNIX Bash shell [9], [10], commit messages for version control [11], [12], code completion [13], test cases from security requirements [14], and more. However, NMT techniques have not heretofore been applied in the field of software security in the manner described in our approach.

Our work provides three key contributions:

- We release a substantive dataset[1] containing exploits collected from shellcode databases and their descriptions in the English language. The dataset includes both assembly code (i.e, shellcodes and decoders) and Python code (i.e., encoders). Such data is valuable to support research in machine translation for security-oriented applications since the techniques are data-driven.
- We propose a new approach that applies NMT techniques to automatically generate exploits, including both Python code for encoding the payload, and assembly code for decoding the payload and for the actual shellcode, based on their description in the English language.
- We perform an extensive experimental study, to evaluate the feasibility of the proposed approach at generating real exploits. To this aim, we propose new metrics that go beyond evaluating the translation of single lines of code [8], [15]–[17], but encompass entire exploits as a whole. The generated code achieved high accuracy in terms of syntactic and semantic correctness.

In the following, Section II introduces background concepts; Section III presents the proposed approach; Section IV describes

[1]The dataset and the code to reproduce the experiments are publicly available here: https://github.com/dessertlab/EVIL

2332-6549/21/$31.00 ©2021 IEEE
DOI 10.1109/ISSRE52982.2021.00042

```
1  global _start
2  section .text
3  _start:
4        xor    eax, eax
5        push   eax
6        push   0x68732f2f
7        push   0x6e69622f
8        mov    ebx, esp
9        push   eax
10       mov    edx, esp
11       push   ebx
12       mov    ecx, esp
13       mov    al, 11
14       int    0x80
15
16 ------------------ Binary opcodes ------------------
17 "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\
18 x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80"
```

Listing 1. Example of shellcode to spawn a shell on Linux x32 systems.

the dataset; Section V experimentally evaluate the approach; Section VI describes the approach further through selected examples; Section VII discusses related work; Section VIII discusses the ethical considerations; Section IX concludes the paper.

II. Background

Statistics from the Common Vulnerabilities and Exposures (CVE) database show that code-injection vulnerabilities increased dramatically during recent years [18], [19]. Code-injection attacks deliver and run arbitrary code on victims' machines, enabling unauthorized access and control of system resources, applications, and data [20]. Moreover, code-injection attacks have become more and more sophisticated, including techniques such as return-oriented programming, heap spraying, and format string attacks.

A shellcode is a list of machine code instructions, to be loaded in a vulnerable application at runtime. The classic way to develop shellcodes is to write them using the assembly language, and by using an assembler to turn them into *opcodes* (operation codes, i.e., a machine language instruction in binary format, to be decoded and executed by the CPU) [21], [22]. Listing 1 shows an example of shellcode[2] in assembly for the 32-bit Intel Architecture, which runs the `/bin/sh` command to spawn a shell on the Linux OS. Note that the shellcode strictly depends on the OS and the CPU architecture, thus it must be tailored for the target system. Listing 1 also shows the shellcode as binary opcodes (lines 17-18), where each $\backslash x$ followed by two hex digits represents one byte of the payload [23]. Shellcodes typically range between few bytes to hundreds of bytes. Other objectives of shellcodes include killing or restart other processes, causing a denial-of-service (e.g., a fork bomb), leaking secret data, etc.

The plain shellcodes contain explicit information of the malicious action the attacker aims to take. For example, the shellcode in Listing 1 contains the values *68 73 2f 2f* and *6e 69 62 2f*, which are the hexadecimal representation of the strings *//sh* and */bin* in reverse order (since the target CPU is little-endian). Therefore, it is easy for AV and IDS software to block the execution of this shellcode. To overcome security protections, exploit writers adopt encoding techniques, which convert the original shellcode into a new, functionally equivalent one, but more difficult to block [24], [25]. Encoders are programs written in a high-level language, most often in Python (e.g., over the 80% of the encoders on the popular Exploit Database [26] are developed in Python language), and apply mathematical operations (as in symmetric key cryptography) on the binary opcodes to generate new ones, and which append additional opcodes for the decoder. Afterward, when this attack payload is injected and executed by the victim system, it decodes itself to obtain the original shellcode. Since the decoder is part of the attack payload, it is developed using the assembly language.

Encoders and decoders are not just used to obfuscate the original payload from AV and IDS. Encoding schemes are used also to eliminate "bad bytes" from the payload. For example, since a null byte is considered as a terminator character for strings, all the bytes of the payload following the null bytes are not processed. Accordingly, the shellcodes must be null-free or zero-free, i.e., they can not contain any null bytes. Moreover, most vulnerabilities impose restrictions on the quantity of data that can be injected. Therefore, another use of encoders is to optimize the shellcode to decrease its size.

As security attacks and defenses evolve with technology, security researchers have been developing new encoding techniques. Among recent studies, Geczi et al. [27] described a technique that converts x86 assembly code, such that the resulting object code only contains printable characters. Similarly, Patel et al. [28] developed a new encoding scheme to produce printable shellcodes but in a more compact and reduced size. The penetration testing tool Metasploit [29] also provides a method, named *sub encoder*, to convert any sequence of binary data into ASCII characters that, when interpreted by an Intel CPU, will decode the original sequence and execute it.

III. Proposed Methodology

The *EVIL* approach leverages neural machine translation (NMT) to automatically generate exploits. A neural machine translation system is a neural network that maximizes the conditional probability $p(a|e)$ of translating a source sentence $e=<e_1,...,e_{T_e}>$, with length $|e|=T_e$, into a target sentence $a=<a_1,...,a_{T_a}>$, with length $|a|=T_a$. Following prior work (e.g., [30]), we build a neural network that directly models the conditional probability $p(a|e)$ of translating an *intent*, in natural language into a *code snippet* in Python or assembly language. As a simple example, consider the sequence of English tokens `['add', 'the', 'value', '4', 'to', 'the', 'eax', 'register']` as e, and the sequence of assembly tokens `['add', 'eax', ',', '4']` as a, with $|e|=8$ and $|a|=4$. The general architecture of an NMT system consists of an encoder, which computes a representation s for each source token, and a decoder, which generates tokens in the target language[3].

To support automatic code generation, neural machine translation is usually accompanied by data processing steps

[2]Shellcode collected from https://www.exploit-db.com/shellcodes/47890

[3]In this section, we use the terms *encoder* and *decoder* to refer to deep learning architectures. In other sections, the terms refer to parts of an exploit.

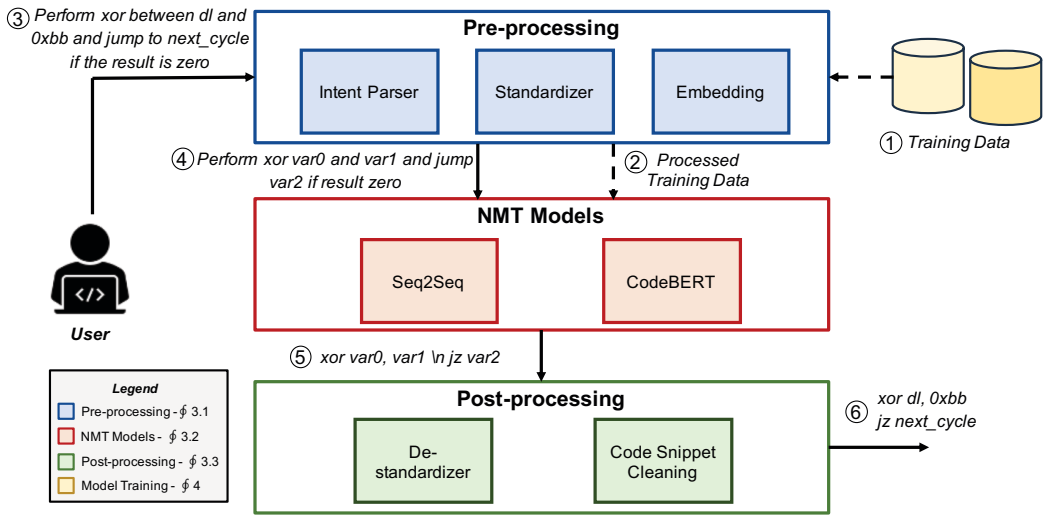

Fig. 1. Architecture diagram of *EVIL* demonstrating a six-step process: 1) Pre-Processing of the samples in the training data, 2) Training of the neural machine translation system with the processed training data, 3) User stating the operation in the English language (i.e., the *intent*), 4) Pre-processing of the intent, 5) NMT models generating the *code snippet* from the processed intent, and lastly, 6) *EVIL* output after post-processing applied to the generated code snippet.

[31]–[33]. These phases strongly depend on the specific source and target languages to translate (in our case, exploit code from the English language). The *EVIL* approach applies steps of processing both before the NMT task (*pre-processing*), to train the NMT model and prepare the input data, and after the NMT task (*post-processing*), to improve the quality and the readability of the code in output. Figure 1 shows the architecture of our approach, along with an example of inputs and outputs at each step, further discussed in the following.

A. Pre-Processing

Pre-processing starts with *stopwords filtering*, i.e., by removing a set of custom compiled words (e.g., *the, each, onto*), in order to include only relevant data for machine translation. This phase splits the input sequence of natural language tokens $e_1,...,e_{T_e}$ and code $a_1,...,a_{T_a}$ in a process called *tokenization*. The tokenizer converts the input strings into their byte representations, and learns to break down a word into subword tokens (e.g., lower becomes `[low, er]`. We tokenize intents using the *nltk word tokenizer* [34] and snippets using the Python *tokenize* package [35]. We then use regular expressions to identify hexadecimal values (e.g., `0xbb`), strings that fall between quotation marks, squared brackets, variable name notations (e.g., `variableName`, `variable_name`), follow underscore notation (e.g., `next_cycle`), function names, mathematical expressions, and byte arrays (e.g., `\xe3 \xa1`). We also use WordNet [36] to recognize alphabet strings that do not belong to the English language.

Consider the natural language intent shown in Figure 1 (step ③), which contains the phrase *jump to* `next_cycle`. One task for code generation systems is to prevent non-English tokens (e.g., `next_cycle`) from getting transformed during the learning process. This process is known as ***Standardization***. Extant code generation systems address this problem with *copying* mechanisms in neural network architectures [37], which are inspired by human memorization. To perform standardization, *EVIL* provides a novel ***Intent Parser*** tailored for working with exploit software. The goal of the Intent Parser is to take input in natural language (i.e. intents) and to provide as output a dictionary of standardizable tokens, such as specific values, label names, and parameters.

All tokens selected by the Intent Parser are passed to the ***Standardizer***. The standardization process simply replaces the selected token in both the intent and snippet with `var#`, with `#` denoting a number from 0 to $|l|$, and $|l|$ is the number of tokens to standardize. In the step ④ in Fig. 1, the intent parser identifies `dl`, `0xbb`, and `next_cycle` as standardizable tokens and standardizes them to `var0`, `var1`, and `var2` respectively (based on order of appearance in the intent). To prevent the standardization of unimportant tokens, we compile a dictionary of 45 assembly keywords (e.g., `register`, `address`, `byte`), and 38 Python keywords (e.g., `for`, `class`, `import`) as non-standardizable tokens. After the standardization process, both the original token and its standardized counterpart (`var#`) are stored in a dictionary to be used during post-processing.

Lastly, we create ***Embeddings***, i.e., a mapping of each token (in both the intent and code snippet sequences) into a numerical id representation in order to capture their semantic and syntactic information, where the semantic information correlates with the meaning of the tokens, while the syntactic one refers to their structural roles [38].

B. NMT Models

To perform neural machine translation (step ⑤), we consider two standard architectures: Seq2Seq, and CodeBERT.

Seq2Seq. Seq2Seq is a common model used in a variety of neural machine translation tasks. Similar to the encoder-decoder architecture with attention mechanism [39], we use a bi-directional LSTM as the encoder, to transform an embedded

intent sequence $e=|e_1,...,e_{T_e}|$ into a vector c of hidden states with equal length. Within the bidirectional LSTM encoder, each hidden state h_t corresponds to an embedded token e_t. The encoder LSTM is bidirectional, which means it reads the source sequence e ordered from left to right (from e_1 to e_{T_e}) and from right to left (from e_{T_e} to e_1). To combine both directions, each hidden state for the bidirectional LSTM encoder is computed by concatenating the hidden states of the forward and backward orders at token t as follows:

$$\mathbf{h}_t = [f(e_t, \mathbf{h}_{t-1}); f(e_t, \mathbf{h}_{t+1})]; \quad (1)$$

where h_t denotes a hidden state at time step t, e_t denotes an embedded intent token t, and f denotes an LSTM non-linear function. Next, the model generates a context vector c_t for each of the embedded inputs. We use the Bahdanau-style attention mechanism [39], which uses soft attention by representing $\mathbf{c}_t$ as the weighted sum of the encoder hidden states, as $\mathbf{c}_t = \sum_{i=1}^{T_e} \alpha_{t,i} \mathbf{h}_i$, where the scores $\alpha_{t,i}$ are parametrized by a feed-forward multi-layer perceptron neural network. Since the encoder uses a bidirectional LSTM, each hidden state $\mathbf{h}_i$ is aware of the context on both ends.

The decoder generates one target token at a time by decomposing the conditional probability $p(a|e)$ into $\prod_{t=1}^{T_a} p(a_t|a_{<t}, s)$, where a_t is an individual output token. The target token are generated by combining the LSTM decoder state s_{t-1}, the previously-generated token a_{t-1}, and the context vector c_t as follows:

$$\begin{gathered} \mathbf{s}_t = f(a_{t-1}, \mathbf{s}_{t-1}, \mathbf{c}) \\ p(a_t|a_{<t}, e) = g(a_{t-1}, \mathbf{s}_t, \mathbf{c}) \end{gathered} \quad (2)$$

where g is a non-linear function, and $a_{<t}$ denotes previous predicted tokens $\{a_1,...,a_{t-1}\}$.

CodeBERT. CodeBERT [40] is a large multi-layer bidirectional Transformer architecture [41]. Like Seq2Seq, the Transformer architecture is made up of encoders and decoders. CodeBERT has 12 stacked encoders and 6 stacked decoders. Compared to Seq2Seq, the Transformer architecture introduces mechanisms to address key issues in machine translation: (i) the translation of a word depends on its position within the sentence; (ii) in the target language, the order of the words (e.g., adjectives before a noun) can be different from the order of words in the source language (e.g., adjectives after a noun); (iii) several words in the same sentence can be correlated (e.g., pronouns). These problems are especially important when dealing with long sentences.

The Transformer architecture first refines the input embedding of each token, by combining it with a *positional encoding* vector. The architecture has a different positional encoding vector for each position of the sentence, in order to enrich the input embedding with positional information. Then, the transformed input embeddings sequentially go through the stacked encoder layers, which all apply a *self-attention* process. The self-attention further refines an input embedding, by combining it with the other input embeddings for the sentence in a weighted way, in order to account for correlations among the words (e.g., to get information for a pronoun from the noun it refers to, the input embedding of the noun is given a large weight). The weights are given by:

$$Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \quad (3)$$

where the vectors Q (query), K (key), and V (value) are learned during training. To further improve this mechanism, the Transformer architecture uses *multi-headed attention*, where input embeddings are first multiplied with three learned weight matrices W_i^Q, W_i^K, and W_i^V (where i is a "head"), then combined with vectors Q, K, and V. Finally, the results from all heads are concatenated and multiplied with an additional weights matrix W^O. Multi-headed attention enables the neural network to correlate different parts of the sequence in different ways (e.g., short-term vs long-term correlations).

$$MultiHead(Q,K,V) = Concat(head_1,...,head_h)W^O \quad (4)$$

Different from Seq2Seq, CodeBERT also comes with a *pre-trained* neural network model, learned from large amounts of code snippets and their descriptions in the English language, and covering six different programming languages, including Python, Java, Javascript, Go, PHP, and Ruby. The goal of pre-training is to bootstrap the training process, by establishing an initial version of the neural network, to be further trained for the specific task of interest [42]–[45]. This approach is called *transfer learning*. In our case, we fine-tune the CodeBERT model to translate English to Python and assembly, using our exploit dataset (see § IV).

CodeBERT undergoes unsupervised pre-training using two optimization objectives: a *masked language modeling* (MLM) objective [44] and a *replaced token detection* (RTD) objective [46]. These two objectives are combined into one loss function $\min_\theta L_{MLM}(\theta) + L_{RTD}(\theta)$.

The MLM objective selects random positions within the intent word sequence e and code snippet sequence c, and replaces them with a special mask token $[MASK]$. The objective of this pre-training is to predict the original tokens that were replaced with $[MASK]$:

$$L_{MLM}(\theta) = \sum_{i \in m^e \bigcup m^c} -\log p^{D_1}(x_i | e^{masked}, c^{masked}) \quad (5)$$

where p^{D_1} is the model that predicts the masked tokens, m^e and m^c are the randomly selected positions, $x = \{e,c\}$ is the original intent-snippet pair, and e^{masked} and c^{masked} are the masked intent and snippet.

In the RTD objective, both the intent and snippet are corrupted by replacing the original token with incorrect ones, and the trained model is used as a *discriminator* (denoted as p^{D_2}) to detect whether a token is original or replaced. To this purpose, CodeBERT uses two *generator* models (similarly to Generative Adversarial Networks), one for intents p^{G_e} and one for snippets p^{G_c}. The generator models are trained to generate plausible token fits for the masked tokens e^{masked} and c^{masked}. The RTD objective is defined as follows:

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

$$\sigma(i)=\begin{cases}1, \text{if } x_i^{corrupt}=x_i\\ 0, \text{otherwise.}\end{cases} \quad (6)$$

C. Post-Processing

Post-processing is an automatic post-editing process applied during decoding in the translation process. The ***Destandardizer*** uses the slot map dictionary generated by the Intent Parser to replace all keys in the standardized intent (i.e., `var0`, `var1` and `var2`) with the corresponding memorized values (i.e., `dl`, `0xbb`, and `next_cycle`).

The generated snippet is then further post-processed using regular expressions (***Code Snippet Cleaning)***. This operation includes the removal of (any) extra-spaces in the output, such as between operations and operands, between byte string identifiers and the byte strings, between objects and method calls in Python, etc., and the removal of (any) extra-backslashes in escaped characters (e.g., `\\n`). Also, during the post-processing, the newline characters `\n` are replaced with new lines to generate multi-line snippets. As a final step, snippet tokens are joined to form a complete snippet (step ⑥).

D. Implementation Details

We implement the Seq2Seq model using `xnmt` [47]. We use an Adam optimizer [48] with $\beta_1=0.9$ and $\beta_2=0.999$, while the learning rate α is set to 0.001. We set all the remaining hyper-parameters in a basic configuration: layer dimension = 512, layers = 1, epochs (with early stopping enforced) = 200, beam size = 5.

Our CodeBERT implementation uses an encoder-decoder framework where the encoder is initialized to the pre-trained CodeBERT weights, and the decoder is a transformer decoder. The decoder is composed of 6 stacked layers. The encoder follows the RoBERTa architecture [43], with 12 attention heads, hidden layer dimension of 768, 12 encoder layers, 514 for the size of position embeddings. We use the Adam optimizer [48]. The total number of parameters is 125M. The max length of the input is 256 and the max length of inference is 128. The learning rate $\alpha=0.00005$, batch size = 32, beam size = 10, and assembly train_steps = 2800, and Python train_steps = 18000.

IV. Model Training

To automatically generate Python and assembly programs used for security exploits, we curated a large dataset for feeding NMT techniques. We collected exploits from publicly available databases [26], [49], public repositories (e.g., GitHub), and programming guidelines. In particular, we focused on exploits targeting Linux, the most common OS for security-critical network services, running on *IA-32* (i.e., the 32-bit version of the x86 Intel Architecture). The dataset consists of two parts: (i) a Python dataset, which contains Python code used by exploits to encode the shellcode, and (ii) an assembly dataset, which includes shellcode and decoders to revert the encoding. A sample in the dataset consists of a snippet of code from these exploits and their corresponding description in the English language. The datasets are processed though the operations described in § III-A, and used to train the NMT models, as shown in in Fig. 1 (steps ① and ②).

TABLE I
DATASETS STATISTICS

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

To deal with the ambiguity of natural language, multiple authors worked together to describe curated snippet intentions in English. The building process of the datasets is similar to established corpora in the NMT field (e.g., the authors of the widespread Django-dataset [50] hired one engineer to create the corpus). To mitigate bias, we reused the comments written by developers of the collected programs; when not available, we followed the style of books/tutorials on assembly/Python and shellcode programming.

Table I summarizes the statistics of both datasets, including the size (i.e., the unique pairs of intents-snippets), the unique lines of code snippets, the unique lines of natural language intents, the unique number of tokens (i.e., words), and the average number of tokens per snippet and intent.

A. Python Data

Our first dataset contains samples to generate Python code for security exploits. In order to make the dataset representative of real exploits, it includes code snippets drawn from exploits from public databases. Differing from general-purpose Python code found in previous datasets [50], the Python code of real exploits entails low-level operations on byte data for obfuscation purposes (i.e., to *encode* shellcodes). 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 cannot be found in the previous general-purpose Python datasets.

In total, we built a dataset that consists of $1,114$ original samples of exploit-tailored Python snippets, and their corresponding intent in the English language. These samples include complex and nested instructions, as typical of Python programming. Table II shows examples of such instructions. In order to perform more realistic training and for a fair evaluation, we left untouched the developers' original code snippets and did not decompose them. We provided English intents to describe nested instructions altogether.

In order to bootstrap the training process for the NMT model, we include in our dataset both the original, exploit-oriented snippets and snippets from a previous general-purpose Python

TABLE II
EXAMPLES OF ENCODING INSTRUCTIONS IN PYTHON

Code Snippet	English Intent
`sb = int(hex(leader)[3:],16)`	*Convert the value of leader to hexadecimal, then slice it at index 3, convert it to an int16 and set its value to the variable sb*
`val2 = int( chunk[i].encode('hex'), 16 ) ^ xor_byte`	*val2 is the result of the bitwise xor between the integer base 16 of the element i of chunk encoded to hex and xor_byte*

dataset. This enables the NMT model to generate code that can mix general-purpose and exploit-oriented instructions. Among the several datasets for Python code generation, we choose the Django dataset [50] due to its large size. This corpus contains 14,426 unique pairs of Python statements from the Django Web application framework and their corresponding description in English. Therefore, our final dataset contains 15,540 unique pairs of Python code snippets alongside their intents in natural language.

B. Assembly Data

We built our assembly dataset on top of our previous work [51], in which we released a dataset for automatically generating assembly from natural language descriptions. This dataset consists of 3,200 assembly instructions, commented in English language, which were collected from shellcodes for *IA-32* and written for the *Netwide Assembler* (NASM) for Linux [52]. In order to make the data more representative of the code that we aim to generate (i.e., complete exploits, inclusive of *decoders* to be delivered in the shellcode), we enriched the dataset with further samples of assembly code, drawn from the exploits that we collected from public databases. Differently from the previous dataset, the new one includes assembly code from *real decoders* used in actual exploits. The final dataset contains 3,715 unique pairs of assembly code snippets/English intents.

To better support developers in the automatic generation of the assembly programs, we looked beyond a one-to-one mapping between natural language intents and their corresponding code. Therefore, the dataset includes 783 lines ($\sim$21% of the dataset) of *multi-line intents*, i.e., intents that generate multiple lines of assembly code, separated by the newline character *\n*. These multi-line snippets contain a number of different assembly instructions that can range between 2 and 5. For example, the copy of the ASCII string *"/bin//sh"* into a register is a typical operation to spawn a shell, which requires three distinct assembly instructions, as shown by the lines 6-7-8 of Listing 1: push the hexadecimal values of the words *"/bin"* and *"//sh"* onto the stack register before moving the contents of the stack register into the destination register. Further examples of multi-line snippets include conditional jumps, tricks to zero-out the registers without generating null bytes, etc. Table III shows two further examples of multi-line snippets with their natural language intents.

TABLE III
EXAMPLES OF DECODING INSTRUCTIONS IN ASSEMBLY

Code Snippet	English Intent
`xor bl, 0xBB \n jz formatting \n mov cl, byte [esi]`	*Perform the xor between BL register and 0xBB and jump to the label formatting if the result is zero else move the current byte of the shellcode in the CL register.*
`xor ecx, ecx \n mul ecx`	*Zero out the EAX and ECX registers.*

V. EXPERIMENTAL EVALUATION

This section presents an extensive evaluation of our approach to generating exploits from natural language descriptions. We separate the evaluation of Python code generation (i.e., the encoding part of the exploit) and assembly code generation (i.e., the decoding part of the exploit). Thus, we use distinct test sets for evaluating Python encoders and assembly language decoders, respectively. We trained two distinct models, using respectively the Python and the assembly dataset. The models adopt the same architecture.

Differently from previous work in code generation tasks [7], [8], [15]–[17], [53], we did not randomly sample individual instructions from the dataset when dividing the data between training and test set. Indeed, since the ultimate goal of the programmer is to generate exploits in their entirety, we took all instructions from an exploit as a whole. Our test sets cover 20 different exploits (i.e., 20 Python programs, and 20 assembly programs). We exclude *print* statements in the Python source code since they are not actually needed for the exploit. The exploits and their encoding/decoding schemes have varying complexity and were developed by different programmers for different purposes. The average number of lines is 23.4 (median is 19) for the Python programs and 26.4 (median is 24) for the decoders in assembly language. The pairs intents-snippets in the test set are unique and are not included in the training/dev sets. Further information on the test set is described in Appendix[4].

Next, we present the experimental results using automated metrics (§ V-A) and manual metrics (§ V-B). In § V-C, we evaluate the approach at generating exploits in their entirety. In § V-D, we evaluate the computational cost.

A. Automatic Evaluation

We evaluate the translation ability of both models described in § III-B, i.e., Seq2Seq with attention mechanisms and CodeBERT. Moreover, since the Intent Parser plays a critical role in our approach, we evaluate the ability of the models with and without the use of the Intent Parser, in order to estimate its contribution. The configuration without the Intent Parser still adopts the pre/post-processing pipeline (stopwords filtering, tokenization, embeddings, etc.) but avoids standardization.

Automatic evaluation metrics are commonly used in the field of machine translation. They are reproducible, easy to be tuned, and time-saving. The *BiLingual Evaluation Understudy* (BLEU)

[4]https://github.com/dessertlab/EVIL

TABLE IV
AUTOMATED EVALUATION OF THE TRANSLATION TASK. BOLDED VALUES ARE THE BEST PERFORMANCE. IP= INTENT PARSER

Dataset	Model	BLEU-1 (%)	BLEU-2 (%)	BLEU-3 (%)	BLEU-4 (%)	ACC (%)
Python	*Seq2Seq no IP*	72.34	62.55	56.88	52.2	31.2
	Seq2Seq with IP	86.92	83.68	81.42	79.69	45.33
	CodeBERT no IP	79.23	74.12	69.84	65.76	48.00
	CodeBERT with IP	**89.22**	**86.78**	**84.94**	**83.50**	**56.00**
Assembly	*Seq2Seq no IP*	35.83	26.1	21.38	17.69	25.25
	Seq2Seq with IP	**88.38**	**86.37**	**85.51**	**85.05**	40.98
	CodeBERT no IP	33.42	28.39	25.38	22.72	45.9
	CodeBERT with IP	88.21	85.53	84.05	82.99	**45.9**

[54] score is one of the most popular automatic metric [8], [50], [55], [56]. This metric is based on the concept of *n-gram*, i.e., the adjacent sequence of n *items* (e.g., syllables, letters, words, etc.) from a given example of text or speech. In particular, this metric measures the degree of n-gram overlapping between the strings of words produced by the model and the human translation references at the corpus level. BLEU measures translation quality by the accuracy of translating n-grams to n-grams, for n-gram of size 1 to 4 [57]. The *Exact match accuracy* (ACC) is another automatic metric often used for evaluating neural machine translation [7], [8], [15], [16]. It measures the fraction of the exact match between the output predicted by the model and the reference.

Table IV shows the best results for each model in terms of BLEU scores and accuracy. For both the Python and assembly datasets, the results point out that the Intent Parser notably increases the performance of the translation task in Python and assembly. Moreover, CodeBERT outperforms the performance of the Seq2Seq with respect to all metrics for the Python dataset. In the case of the assembly dataset, the performance of the two models is comparable.

B. Human Evaluation

We further investigate the performance of the translation task by assessing the deeper linguistic features [58], i.e., the syntax and the semantic of the code snippets predicted by the models. These features allow us to properly give credit to semantically-correct code that fails to match the reference one (e.g., `jz label` and `je label` are semantically identical code snippets, even if they use different instructions), or to provide information whether the generated code would compile or not. Accordingly, we define two new metrics: a generated output snippet is considered ***syntactically correct*** if it is correctly structured in the grammar of the target language (i.e., Python or assembly) and compiles correctly. The output is considered ***semantically correct*** if the snippet is an appropriate translation in the target language given the intent description. The semantic correctness implies syntax correctness, while a snippet can be syntactically correct but semantically incorrect. Of course, the syntactic incorrectness also implies the semantic one. We evaluated these metrics through manual inspection.

As a simple example, consider the intent *res2 is the result of the bitwise and operation between res2 and val1*. If the model generates `res2 = res2 & val1`, then the snippet is considered semantically and syntactically correct. If the model generates the Python instruction `res2 = val2 | val1`, then the snippet is considered syntactically correct for the Python syntax, but semantically incorrect both because the bitwise operation is not the intended one (i.e., the `or` instead of the `and`), and the operands are not the same specified in the intents (i.e., `val2` instead of `res2`). If the model generates the instruction `res2 = res2 _ val1`, then the snippet is considered also syntactically incorrect, since the symbol _ is not a valid binary operator in Python. Notice that this type of evaluation is rigorous. Even if a single token of the generated snippet (e.g., an operation, a variable/operand, a parenthesis, etc.) is not syntactically (semantically) correct, then the whole snippet is considered syntactically (semantically) incorrect. In the case of the multi-line snippets of the assembly dataset, we compute the syntactic (semantic) correctness as the ratio number of syntactically (semantically) correct single snippets over the total number of snippets composing the multi-lines statement (c.f. § VI-B).

TABLE V
HUMAN EVALUATION OF THE TRANSLATION TASK. BOLDED VALUES ARE THE BEST PERFORMANCE ($*= p<0.05$). IP= INTENT PARSER

Dataset	Model	Syntactic Correctness (%)	Semantic Correctness (%)
Python	*Seq2Seq with IP*	88.53	50.13
	CodeBERT with IP	**93.60***	**67.73***
Assembly	*Seq2Seq with IP*	**90.16**	56.36
	CodeBERT with IP	87.05	**61.90***

We evaluated both types of correctness achieved by Seq2Seq and CodeBERT on the entire test-sets (all 20 encoders/decoders). For the evaluation, we were supported by the Python 2 and the NASM compiler, respectively for Python and assembly code. Since the previous analysis showed that the Intent Parser notably increases the performance of translation, we evaluated the syntactic and semantic correctness only when using the Intent Parser. Table V shows the percentage of the syntactically and

semantically correct snippets generated by the models. The table shows that, for the Python programs, CodeBERT with IP provides a higher percentage of both syntactically and semantically correct snippets. We conducted a *paired-sample T-test* to compare the syntactic correctness and the semantic correctness values of the code snippet pairs predicted by the two NMT models (given the same intents). We found that the differences between CodeBERT and the Seq2Seq are statistically significant for both metrics with $p < 0.05$. For the assembly programs, Seq2Seq provides a higher percentage of syntactically correct snippets, but these differences are not statistically significant. Again, CodeBERT outperforms Seq2Seq in the semantic correctness ($p < 0.05$).

These findings highlight that CodeBERT predicts the highest percentage of code snippets that are semantically equivalent to the English intent. It is interesting that, differently from the Seq2Seq, this model provides better performance when applied to the Python dataset. We attribute these differences to the pre-training of the model, which benefits from knowledge from six high-level programming languages, including Python. Finally, we investigated the difference between the percentage of syntactic correctness and the semantic one. We found that most of the semantically incorrect predictions are due to wrong labels or variable names, or the omission of parenthesis around expressions. These errors are easy to identify and correct by a programmer, as they would require one single edit to make the predicted snippet semantically consistent with the intent. These errors are further analyzed in § VI.

According to these results, the NMT approach can correctly translate an individual intent with high likelihood, and the incorrect translations are still close to being correct. These results support the use of the NMT approach as a way for developers to look up code snippets that they could not recall or that are not confident yet to develop themselves, getting a close-to-correct snippet that they can use with little effort.

C. Whole-exploit evaluation

The ultimate goal of the programmer is to generate entire software exploits. While evaluation using the manual metrics indicates that the model can correctly generate individual snippets with high likelihood, we do not know yet whether the model can generate a correct exploit as a whole. Therefore, we evaluated the ability of the approach to generate semantically and syntactically correct code for entire software exploits (i.e., all of the lines of code in the programs), using two new metrics.

Let n_t^i be the the number of total lines of the i-th program in the test set ($i \in [1, 20]$). Let also consider n_{syn}^i as the number of automatically-generated snippets for the i-th program that are syntactically correct, and n_{sem}^i as the number of automatically-generated snippets that are semantically correct. For every program of the test set, we define the ***syntactic correctness*** of the program i as the ratio n_{syn}^i/n_t^i, and the ***semantic correctness*** of the program as the ratio n_{sem}^i/n_t^i. Both metrics range between 0 and 1.

Therefore, $\forall i \in [1, 20]$, we computed the values n_{syn}^i and n_{sem}^i for Python and assembly programs. We performed this analysis with CodeBERT with IP since it showed the best

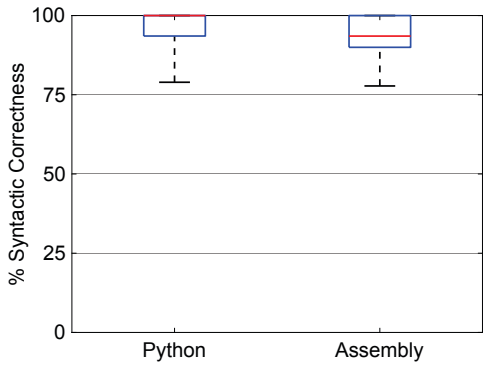

Fig. 2. Distribution of the syntactic correctness of the programs.

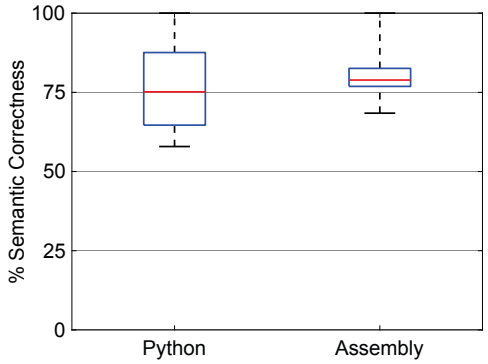

Fig. 3. Distribution of the semantic correctness of the programs.

performance for both datasets in the human evaluation (§ V-B). We found that the average syntactic correctness over all the programs of the test set is $\sim$96% for the Python programs and $\sim$93% for the assembly programs. Similarly, we estimated the average semantic correctness, which is equal to $\sim$76% and $\sim$81% for Python and assembly programs, respectively. The box-plots in Fig. 2 and Fig. 3 summarize the results.

Despite our conservative evaluation, the approach is able to generate 12 Python programs and 6 assembly programs fully composed by syntactically correct snippets (i.e., the $n_{syn}^i/n_t^i = 100\%$). However, this does not imply that the programs are also executable. For example, the assembly instruction `jz shellcode` is syntactically correct, but if the label `shellcode` is not defined, then the program would not compile. Therefore, we evaluated how many programs can be compiled, finding that the 50% of syntactically correct programs are also compilable and executable. Furthermore, one Python program and one assembly program are fully composed of semantically correct snippets (i.e., $n_{sem}^i/n_t^i = 100\%$) and, therefore, implement the original encoding and decoding schemes. We interpret these results as a promising *first* research step towards automatically coding software exploits from natural language intents.

D. Computational Time

We performed our experiments on a Linux OS running on a virtual machine. Seq2Seq utilized 8 CPU cores and 8 GB

TABLE VI
ILLUSTRATIVE EXAMPLES OF CORRECT OUTPUT.

Dataset	Natural Language Intent	Ground Truth	Model Output
Python	*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*	`xxx = hex(int(abs(subfs)) + int(rev_suplx[0:2],16))`	`xxx = hex(int(abs(subfs)) + int(rev_suplx[0:2],16))`
	Append 0xAA to the string encode	`encode += '0xAA'`	`encode += '0x%02x'%170`
Assembly	*Jump short to decode if al is not equal to cl else jump to shellcode.*	`cmp al, cl \n jnz short decode \n jmp shellcode`	`cmp al, cl \n jne short decode \n jmp shellcode`
	Perform the xor between the current byte of the shellcode and the dl register.	`xor byte [esi], dl`	`xor byte [esi], dl`

TABLE VII
ILLUSTRATIVE EXAMPLES OF INCORRECT OUTPUT. THE PREDICTION ERRORS ARE **RED/BOLD**. ~~SLASHED~~ TEXT REFERS TO OMITTED PREDICTIONS.

Dataset	Natural Language Intent	Ground Truth	Model Output
Python	*Return the result of bitwise left shift of n by rotations bitwise and with mask_value bitwise or n bitwise right shift by the subtraction of rotations from width*	`return ((n<<rotations) & mask_value) | (n >> (width − rotations))`	`return (~~(~~n<<rotations~~)~~ & **bitwise**~~)~~ | ~~(~~n >> ~~(~~width − rotations~~)~~~~)~~`
	Convert the value of leader to hexadecimal, then slice it at index 3, convert it to an int16 and set its value to the variable sb	`sb = int(hex(leader)[3:],16)`	`sb = int(hex(leader)[3:],**32**)`
Assembly	*Zero out the ecx register and move 25 in the lower 8 bits of the register.*	`xor ecx, ecx \n mov cl, 25`	`xor ecx, ecx \n mov **al**, 25`
	In the decode function jump to lowbound label if the current byte of the shellcode is lower than 0x7 else subtract 0x7 from the byte of the shellcode and jump to common_commands	`decode: \n cmp byte [esi], 0x7 \n jl lowbound \n sub byte [esi], 0x7 \n jmp common_commands`	`decode: \n cmp byte [esi], 0x7 \n jl lowbound ~~\n sub byte [esi], 0x7~~ \n ~~jmp~~ common_commands`

RAM. CodeBERT utilized 8 CPU cores, 16 GB RAM, and 2 GTX1080Ti GPUs.

The computational time needed to generate the output depends on the settings of the hyper-parameters and the size of the dataset. On average, the training time of the Seq2Seq model is ∼500 minutes for the Python dataset and ∼60 minutes for the assembly dataset, while CodeBERT requires for the training in average ∼220 minutes for Python and ∼25 minutes for the assembly dataset. In total, we performed 50 different experiments for a total computational time of ∼150 hours. Once the models are trained, the time to predict the output is below 1 second and can be considered negligible.

VI. QUALITATIVE ANALYSIS

In this section, we present a qualitative analysis using cherry-picked examples from our test sets to highlight both successful prediction cases and failed ones.

A. Successful Examples

Table VI shows four cases of success predictions for both datasets. The first row demonstrates our approach's ability to generate a difficult Python snippet from a nested natural language intent without any errors. Indeed, the predicted output contains the correct variable name `subfs`, `rev_suplx`, and `xxx` with the appropriate functions `hex`, `int`, `abs` indexed in the right order with the correct parameters. The second row shows an example of implicit model knowledge. The model is able to append `0xAA` by converting the value 170, the hexadecimal value of `0xAA`, to hexadecimal and appending it to the string variable encode.

The last two rows of the table show the ability of the model in generating challenging assembly instructions. The third row highlights both the ability of the Intent Parser in identifying all the registers' name (i.e., `AL` and `CL`), and the labels (i.e., `decode` and `shellcode`), and the ability of the model to perform the right translation of the order in the *if-then-else* statements described in the English intent. The last row is an interesting example of implicit model knowledge. Indeed, even if it is not stated in the intent, the model is able to properly predict `EDI` as the destination register since it is typically used to store the encoded shellcodes.

B. Failure Examples

Table VII shows four relevant examples of failure cases. We observe nested instructions within the intent in the Python examples. In the first row, the intent implies the order of operations signified by parenthesis in the ground truth snippet. While the model correctly generates all the operations in the correct order, it does not enforce the order of operations. The model fails to derive the implied order of operations from the intent. It also fails to generate the correct variable `mask_value`, but instead generates the intent-repeated token `bitwise` as a variable, likely due to `bitwise` being

mentioned frequently in the intent in between operations. In the second row, we observe the model also generates all the operands and variables correctly, however it fails to generate the correct parameter 16 to the integer conversion operation.

The example in the third row illustrates an example of failure due to a lack of implicit knowledge. The natural language intent does not mention to the `CL` register but implicitly refers to it (*lower 8 bits of the `ECX` register*). Therefore, the output results in a syntactically correct but semantically incorrect prediction. The last row shows a long intent that describes an entire function. The model is able to properly predict four of the five assembly instructions composing the decode function. However, it misses the `sub` operation after the conditional jump `jl`. In this case, the semantic correctness is equal to 0.8 (4 out of 5 semantically correct snippets).

VII. Related Work

The task of exploit generation via automatic techniques has been addressed in several ways. *ShellSwap* [59] is a system that generates new exploits based on existing ones, by modifying the original shellcode with arbitrary replacement shellcode. Hu *et al.* [60] developed a novel approach to construct data-oriented exploits through data flow stitching, by composing the benign data flows in an application via a memory error. They built a prototype attack generation tool that operates directly on Windows and Linux x86 binaries. Avgerinos *et al.* [1] developed an end-to-end system for automatic exploit generation (AEG) on real programs by exploring execution paths. Given the potentially buggy program in source form, their proposal automatically looks for bugs, determines whether the bug is exploitable, and produces a working control-flow hijack exploit string. *SemFuzz* [61] extracts necessary information from non-code text related to a vulnerability, using natural language processing and a semantics-based fuzzing process, in order to discover and trigger deep bugs. Chen *et al.* [62] presented techniques to find out the *gadgets*, i.e., the basic building block in Jump Oriented Programming (JOP), and showed these gadgets are Turing complete. They implemented an automatic tool able to generate JOP shellcodes. Ding *et al.* [63] proposed a reverse derivation of a transformation method driven by state machines indicating the status of data flows, in order to transform the original shellcode into printable Return Oriented Programming (ROP) payload. *Chainsaw* [64] is a tool for analyzing web applications and generating injection exploits. The tool performs static analysis and defines a model of the application behavior to generate injection exploits, by leveraging application workflow structures and database schemes. Brumley *et al.* [65] proposed an approach for Automatic Patch-based Exploit Generation (APEG). Starting from a program and its patched version, the approach identifies the security checks added by the patch and automatically generates inputs to fail the checks. Huang *et al.* [66] introduced a method to automatically generate exploits based on software crash analysis. This method analyzes software crashes using a symbolic failure model, to generate exploits from crash inputs and existing exploits for several types of applications. Xu *et al.* [67] developed a tool to find buffer overflow vulnerabilities in binary programs and automatically generate exploits using a constraint solver. Vulnerability detection is achieved through symbolic execution and the exploit generated by this tool can bypass different types of protection.

Our work is radically different from these previous ones. First, our approach uses natural language statements to generate exploits. Second, we adopt neither a static nor dynamic program analysis approach (e.g., fuzzing, program synthesis, etc.), but a statistical, data-driven approach. Therefore, our work can be considered complementary to these previous solutions, with different use cases.

VIII. Ethical Considerations

Offensive security is a sub-field of security research that tests security measures from an adversary or competitor's perspective, employing ethical hackers to probe a system for vulnerabilities [68], [69]. Our work aims to automate exploit generation, in order to explore critical vulnerabilities before they are exploited by attackers [1]. Indeed, our work simplifies the process of coding the exploits to surface security weaknesses within the software and can provide valuable information about the technical skills, degree of experience, and intent of the attackers. With this information, it is possible to implement measures to detect and prevent attacks [3].

IX. Conclusion

We presented *EVIL*, an approach for automatic exploit generation for security assessment purposes, using natural language processing techniques based on neural networks. Our approach represents the first step towards the ambitious goal of automatically generating software exploits from natural language. We develop and released two datasets of real exploits in Python and assembly language to enable neural network training and experimental evaluation. We evaluated the feasibility of our approach, using both automated and manual metrics. Our experiments have shown the ability of the approach in generating software exploits from natural language descriptions with high syntactic and semantic correctness.

The results have also revealed that, in most cases, the generated programs do not execute correctly due to wrong labels or variable names. Programmers can easily correct these problems, but our goal is full automation. Therefore, future work includes the improvement of the post-processing phase that looks at the program context to increase the accuracy of the program generation task. Future work also includes the development of a single engine that generates the encoding and decoding schemes at the same time, without performing two separate translation tasks.

Acknowledgment

This work has been partially supported by the University of Naples Federico II in the frame of the Programme F.R.A., project id OSTAGE, the Italian Ministry of University and Research (MUR) under the programme "PON Ricerca e Innovazione 2014-2020 – Dottorati innovativi con caratterizzazione industriale" and Cisco Systems Inc., project id BOTITS.

REFERENCES

[1] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "Aeg: Automatic exploit generation," in *NDSS*, 2011.

[2] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic exploit generation," *Communications of the ACM*, vol. 57, no. 2, pp. 74–84, 2014.

[3] I. Arce, "The shellcode generation," *IEEE security & privacy*, vol. 2, no. 5, pp. 72–76, 2004.

[4] J. Mason, S. Small, F. Monrose, and G. MacManus, "English shellcode," in *ACM Conf. on Computer and Communications Security*, 2009, pp. 524–533.

[5] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *arXiv:1609.08144*, 2016.

[6] O. Bojar, R. Chatterjee, C. Federmann, Y. Graham, B. Haddow, M. Huck, A. J. Yepes, P. Koehn, V. Logacheva, C. Monz *et al.*, "Findings of the 2016 conference on machine translation," in *Conf. on Machine Translation*, 2016, pp. 131–198.

[7] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," *arXiv:1704.01696*, 2017.

[8] W. Ling, E. Grefenstette, K. M. Hermann, T. Kociský, A. W. Senior, F. Wang, and P. Blunsom, "Latent predictor networks for code generation," *arXiv:1603.06744*, 2016.

[9] X. V. Lin, C. Wang, D. Pang, K. Vu, and M. D. Ernst, "Program synthesis from natural language using recurrent neural networks," *University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, Tech. Rep. UW-CSE-17-03-01*, 2017.

[10] X. V. Lin, C. Wang, L. Zettlemoyer, and M. D. Ernst, "Nl2bash: A corpus and semantic parser for natural language interface to the linux operating system," in *Intl. Conf. on Language Resources and Evaluation (LREC)*, 2018.

[11] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *IEEE/ACM Intl. Conf. on Automated Software Engineering (ASE)*. IEEE, 2017, pp. 135–146.

[12] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang, "Neural-machine-translation-based commit message generation: how far are we?" in *ACM/IEEE Intl. Conf. on Automated Software Engineering (ASE)*, 2018, pp. 373–384.

[13] M. Ciniselli, N. Cooper, L. Pascarella, D. Poshyvanyk, M. Di Penta, and G. Bavota, "An empirical study on the usage of BERT models for code completion," *arXiv:2103.07115*, 2021.

[14] X. P. Mai, F. Pastore, A. Göknil, and L. Briand, "A natural language programming approach for requirements-based security testing," in *29th IEEE International Symposium on Software Reliability Engineering (ISSRE 2018)*. IEEE, 2018.

[15] P. Yin and G. Neubig, "Tranx: A transition-based neural abstract syntax parser for semantic parsing and code generation," *arXiv:1810.02720*, 2018.

[16] ——, "Reranking for neural semantic parsing," in *Annual Meeting of the Association for Computational Linguistics*, 2019, pp. 4553–4559.

[17] F. F. Xu, Z. Jiang, P. Yin, B. Vasilescu, and G. Neubig, "Incorporating external knowledge through pre-training for natural language to code generation," *ArXiv*, vol. abs/2004.09015, 2020.

[18] I.R.M. Association, *Software Design and Development: Concepts, Methodologies, Tools, and Applications*. IGI Global, 2013.

[19] ——, *Identity Theft: Breakthroughs in Research and Practice*. IGI Global, 2016.

[20] K. Z. Snow, S. Krishnan, F. Monrose, and N. Provos, "Shellos: Enabling fast detection and forensic analysis of code injection attacks." in *USENIX Security Symposium*, 2011, pp. 183–200.

[21] J. Foster, *Sockets, Shellcode, Porting, and Coding: Reverse Engineering Exploits and Tool Coding for Security Professionals*. Elsevier Science, 2005. [Online]. Available: https://books.google.it/books?id=ZNI5dvBSfZoC

[22] H. Megahed, *Penetration Testing with Shellcode: Detect, exploit, and secure network-level and operating system vulnerabilities*. Packt Publishing, 2018.

[23] J. Koziol, D. Litchfield, D. Aitel, C. Anley, S. Eren, R. Hassell, and N. Mehta, *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. Wiley, 2004.

[24] J. Erickson, *Hacking, 2nd Edition: The Art of Exploitation*. No Starch Press, 2008.

[25] D. Regalado, S. Harris, A. Harper, C. Eagle, J. Ness, B. Spasojevic, R. Linn, and S. Sims, *Gray Hat Hacking The Ethical Hacker's Handbook, Fourth Edition*. McGraw-Hill Education, 2015.

[26] "Exploit Database Shellcodes," https://www.exploit-db.com/shellcodes?platform=linux_x86/, accessed: 2021-04-16.

[27] Z. Géczi and P. Iványi, "Automatic translation of assembly shellcodes to printable byte codes," *Pollack Periodica*, vol. 13, no. 1, pp. 3–20, 2018.

[28] D. Patel, A. Basu, and A. Mathuria, "Automatic generation of compact printable shellcodes for x86," in *USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[29] "Metasploit," https://www.metasploit.com/, accessed: 2021-04-22.

[30] M.-T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," *arXiv:1508.04025*, 2015.

[31] C. Park, Y. Yang, K. Park, and H. Lim, "Decoding strategies for improving low-resource machine translation," *Electronics*, vol. 9, no. 10, p. 1562, 2020.

[32] H. N. Tien and H. N. Thi Minh, "Long sentence preprocessing in neural machine translation," in *2019 IEEE-RIVF Intl. Conf. on Computing and Communication Technologies (RIVF)*, 2019, pp. 1–6.

[33] M. Oudah, A. Almahairi, and N. Habash, "The impact of preprocessing on arabic-english statistical and neural machine translation," *arXiv:1906.11751*, 2019.

[34] E. Loper and S. Bird, "Nltk: the natural language toolkit," *arXiv:cs/0205028*, 2002.

[35] Python, "tokenize," Accessed: 2020-05-20. [Online]. Available: https://docs.python.org/3/library/tokenize.html

[36] G. A. Miller, "Wordnet: a lexical database for english," *Communications of the ACM*, vol. 38, no. 11, pp. 39–41, 1995.

[37] J. Gu, Z. Lu, H. Li, and V. O. Li, "Incorporating copying mechanism in sequence-to-sequence learning," in *Annual Meeting of the Association for Computational Linguistics*, 2016, pp. 1631–1640.

[38] Y. Li and T. Yang, "Word embedding for understanding natural language: a survey," in *Guide to big data applications*. Springer, 2018, pp. 83–104.

[39] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv:1409.0473*, 2015.

[40] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "Codebert: A pre-trained model for programming and natural languages," in *EMNLP*, 2020.

[41] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[42] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," *arXiv:1802.05365*, 2018.

[43] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "RoBERTa: A robustly optimized BERT pretraining approach," *arXiv:1907.11692*, 2019.

[44] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," in *NAACL-HLT*, 2019.

[45] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *arXiv:2005.14165*, 2020.

[46] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, "Electra: Pre-training text encoders as discriminators rather than generators," in *Intl. Conf. on Learning Representations*, 2019.

[47] G. Neubig, M. Sperber, X. Wang, M. Felix, A. Matthews, S. Padmanabhan, Y. Qi, D. S. Sachan, P. Arthur, P. Godard *et al.*, "Xnmt: The extensible neural machine translation toolkit," *arXiv:1803.00188*, 2018.

[48] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv:1412.6980*, 2015.

[49] "Shellcodes database for study cases," http://shell-storm.org/shellcode/, accessed: 2021-04-16.

[50] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to generate pseudo-code from source code using statistical machine translation (t)," in *IEEE/ACM Intl. Conf. on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 574–584.

[51] P. Liguori, E. Al-Hossami, D. Cotroneo, R. Natella, B. Cukic, and S. Shaikh, "Shellcode_IA32: A dataset for automatic shellcode generation," in *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*. Online: Association for Computational Linguistics, Aug. 2021, pp. 58–64. [Online]. Available: https://aclanthology.org/2021.nlp4prog-1.7

[52] J. Duntemann, *Assembly language step-by-step: programming with DOS and Linux*. John Wiley & Sons, 2000.

[53] A. V. M. Barone and R. Sennrich, "A parallel corpus of python functions and documentation strings for automated code documentation and code generation," *arXiv:1707.02275*, 2017.

[54] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Annual Meeting on Association for Computational Linguistics*, 2002, pp. 311–318.
[55] C. Gemmell, F. Rossetto, and J. Dalton, "Relevance transformer: Generating concise code snippets with relevance feedback," in *Intl. ACM Conf. on Research and Development in Information Retrieval*, 2020, pp. 2005–2008.
[56] N. Tran, H. Tran, S. Nguyen, H. Nguyen, and T. Nguyen, "Does BLEU score work for code migration?" in *IEEE/ACM Intl. Conf. on Program Comprehension (ICPC)*, 2019, pp. 165–176.
[57] L. Han, "Machine translation evaluation resources and methods: A survey," *arXiv:1605.04515*, 2016.
[58] L. Han, G. J. F. Jones, and A. F. Smeaton, "Translation quality assessment: A brief survey on manual and automatic methods," *arXiv:2105.03311*, 2021.
[59] T. Bao, R. Wang, Y. Shoshitaishvili, and D. Brumley, "Your exploit is mine: Automatic shellcode transplant for remote exploits," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 824–839.
[60] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of data-oriented exploits," in *USENIX Security Symposium*, 2015, pp. 177–192.
[61] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang, "Semfuzz: Semantics-based automatic generation of proof-of-concept exploits," in *ACM Conference on Computer and Communications Security*, 2017, pp. 2139–2154.
[62] P. Chen, X. Xing, B. Mao, L. Xie, X. Shen, and X. Yin, "Automatic construction of jump-oriented programming shellcode (on the x86)," in *ACM Symp. on Information, Computer and Communications Security*, 2011, pp. 20–29.
[63] W. Ding, X. Xing, P. Chen, Z. Xin, and B. Mao, "Automatic construction of printable return-oriented programming payload," in *Intl. Conf. on Malicious and Unwanted Software: The Americas (MALWARE)*, 2014, pp. 18–25.
[64] A. Alhuzali, B. Eshete, R. Gjomemo, and V. Venkatakrishnan, "Chainsaw: Chained automated workflow-based exploit generation," in *ACM Conf. on Computer and Communications Security*, 2016, pp. 641–652.
[65] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*, 2008, pp. 143–157.
[66] S.-K. Huang, M.-H. Huang, P.-Y. Huang, H.-L. Lu, and C.-W. Lai, "Software crash analysis for automatic exploit generation on binary programs," *IEEE Trans. on Reliability*, vol. 63, no. 1, pp. 270–289, 2014.
[67] L. Xu, W. Jia, W. Dong, and Y. Li, "Automatic exploit generation for buffer overflow vulnerabilities," in *IEEE Intl. Conf. on Software Quality, Reliability and Security*, 2018, pp. 463–468.
[68] S. Bratus, I. Arce, M. E. Locasto, and S. Zanero, "Why offensive security needs engineering textbooks," *Yale Law & Policy Review*, p. 2, 2013.
[69] J. G. Oakley, "The state of modern offensive security," in *Professional Red Teaming*. Springer, 2019, pp. 29–41.