Shellcode_IA32: A Dataset for Automatic Shellcode Generation

Pietro Liguori[1], **Erfan Al-Hossami**[2], **Domenico Cotroneo**[1], **Roberto Natella**[1],
Bojan Cukic[2] and **Samira Shaikh**[2]

[1]University of Naples Federico II , Naples, Italy
[2]University of North Carolina at Charlotte, Charlotte, NC, USA
{pietro.liguori, cotroneo, roberto.natella}@unina.it
{ealhossa, bcukic, samirashaikh}@uncc.edu

Abstract

We take the first step to address the task of automatically generating shellcodes, i.e., small pieces of code used as a payload in the exploitation of a software vulnerability, starting from natural language comments. We assemble and release a novel dataset (*Shellcode_IA32*), consisting of challenging but common assembly instructions with their natural language descriptions. We experiment with standard methods in neural machine translation (NMT) to establish baseline performance levels on this task.

1 Introduction and Related Work

A growing body of research has dealt with automated code generation: given a natural language description, a code comment or intent, the task is to generate a piece of code in a programming language (Yin and Neubig, 2017; Ling et al., 2016). The task of generating programming code snippets, also referred to as semantic parsing (Yin and Neubig, 2019; Xu et al., 2020), has been previously addressed to generate executable snippets in domain-specific languages (Guu et al., 2017; Long et al., 2016), and several programming languages, including Python (Yin and Neubig, 2017) and Java (Ling et al., 2016).

We consider the task of generating *shellcodes*, i.e., small pieces of code used as a payload to exploit software vulnerabilities. Shellcoding, in its most literal sense, means writing code that will return a remote shell when executed. It can represent any byte code that will be inserted into an exploit to accomplish the desired, malicious, task (Mason et al., 2009). An example of a shellcode program in assembly language and the corresponding natural language comments are shown in Listing 1.

Shellcodes are important because they are the key element of security attacks: they represent code injected into victim software to take control of

```
1  global _start;   Declare global _start.
2  section .text;   Declare the text section.

3  _start:;         Define the _start label.
4  xor eax, eax;    Zero out the eax
                    register
5  push eax;        and push its contents
                    on the stack.
6  push 0x68732f2f; Move /bin//sh
7  push 0x6e69622f; into the ebx register.
8  mov ebx, esp
9  push eax;        Push the contents of eax
                    onto the stack
10 mov edx, esp;    and point edx to the
                    stack register.
11 push ebx;        Push the contents of ebx
                    onto the stack
12 mov ecx, esp;    and point ecx to the
                    stack register.
13 mov al, 11;      Put the system call 11
                    into the al register.
14 int 0x80;        Make the kernel call.
```

Listing 1: x86 assembly code used to spawn /bin/sh shell on Linux OS. Lines 4-5, 6-7-8, 9-10, 11-12 are multi-line snippets generated by four different intents.

a machine, to escalate privileges, and to use the machine for malicious purposes such as DDoS attacks, data theft, and running malware (Arce, 2004). Well-intentioned actors (security practitioners and product vendors) also develop shellcodes to run non-harmful *proof-of-concept* attacks, to show how security weaknesses can be exploited to identify vulnerabilities and patch systems. Thus, shellcode generation using (semi-) automated techniques has become a popular and very active research topic (Bao et al., 2017). However, writing shellcodes is technically challenging since they are typically written in assembly language (c.f. Listing 1). The most sophisticated shellcodes can reach hundreds of assembly lines of code.

The task of the shellcode generation has been addressed by several works and tools. Bao *et al.* (2017) designed ShellSwap, a system that can modify an observed exploit and replace the original

58

shellcode with an arbitrary replacement shellcode. The system uses symbolic tracing, with a combination of shellcode layout remediation and path kneading to achieve shellcode transplants. *Pwntools* (pwntools, Accessed: 2021-05-29) is a CTF framework and exploit development library written in Python. It is designed for rapid prototyping and development and intended to make exploit writing as simple as possible.

Differently from previous work in the security literature, we approach this problem as a machine translation (NMT) task. We apply neural machine translation (Goodfellow et al., 2016), which unlike the traditional phrase-based translation system consisting of many small sub-components tuned separately, attempts to build and train a single, large neural network that reads a sentence and outputs a correct translation (Bahdanau et al., 2015). NMT has emerged as a promising machine translation approach, showing superior performance on public benchmarks (Bojar et al., 2016), and it is widely recognized as the premier method for the translation of different languages (Wu et al., 2016). NMT has also been used to perform complex tasks on the UNIX operating system shell (Lin et al., 2017) (e.g. file manipulation and search), by stating goals in English (Lin et al., 2018), to automatically generate commit messages (Liu et al., 2018), etc. However, the NMT techniques have not heretofore been adopted to automatically generate software exploits from natural language comments.

Since NMT is a data-driven approach to code generation, we need a dataset of intents in natural language, and their corresponding translation (in our context, in assembly language) for shellcode generation. In this preliminary work, we address the lack of such a dataset by presenting *Shellcode_IA32*, a dataset containing 3, 200 lines of assembly code extracted from real shellcodes and described in the English language. Moreover, we present experiments on our dataset using a baseline technique, in order to establish performance levels for evaluating shellcode generation techniques.

2 Dataset

We compiled a dataset, ***Shellcode_IA32***, specific to our task. This dataset consists of 3,200 examples of instructions in assembly language for *IA-32* (the 32-bit version of the x86 Intel Architecture) from publicly-available security exploits. We collected assembly programs used to generate shellcode from

shell-storm (Shellcodes database for study cases, Accessed: 2021-04-22) and from *Exploit Database* (Exploit Database Shellcodes , Accessed: 2021-04-22), in the period between 2000 and 2020.

Our focus is on Linux, the most common OS for security-critical network services. Accordingly, we added assembly instructions written with *Netwide Assembler* (NASM) for Linux (Duntemann, 2000). NASM is line-based. Figure 1 shows a simple example of a NASM source line. Every source line contains a combination of four fields: an optional *label* used to represent either an identifier or a constant, a *mnemonic* or *instruction*, which identifies the purpose of the statement and followed by zero or more *operands* specifying the data to be manipulated, and an optional *comment*, i.e., text ignored by the compiler. A mnemonic is not required if a line contains only a label or a comment.


```
wordvar:  resw      1    ; reserve a word for wordvar
```
    label   instruction operand   comment

Figure 1: Layout of NASM source line

Each line of *Shellcode_IA32* dataset represents a snippet – intent pair. The **snippet** is a line or a combination of multiple lines of assembly code, built by following the NASM syntax. The **intent** is a comment in the English language (c.f. Listing 1).

To take into account the variability of descriptions in natural language, multiple authors described independently different samples of the dataset in the English language. Where available, we used as natural language descriptions the comments written by developers of the collected programs. We enriched the dataset by adding examples of assembly programs for the IA-32 architecture from popular tutorials and books (Duntemann, 2011; Kusswurm, 2014; Tutorialspoint, Accessed: 2021-04-22) to understand how different authors and assembly experts describe the code and, thus, how to deal with the ambiguity of natural language in this specific context. Our dataset consists of $\sim 10\%$ of instructions collected from books and guidelines and the rest from real shellcodes.

Multi-line Snippets: To automatically generate shellcodes, we need to look beyond a one-to-one mapping between a line of code and its comment/intent. For example, a common operation in shellcodes is to save the ASCII *"/bin/sh"* into a register. This operation requires three distinct assembly

Intent: *jump short to the decode label if the contents of the* `al` *register is not equal to the contents of the* `cl` *register else jump to the shellcode label*
Multi-line Snippets: `cmp al, cl \n` `jne short decode \n jmp shellcode`
Intent: *jump to the label recv_http_request if the contents of the* `eax` *register is not zero else subtract the value* `0x6` *from the contents of the* `ecx` *register* Multi-line Snippets: `test eax, eax \n` `jnz recv_http_request \n sub ecx, 0x6`

Table 1: Examples of multi-line snippets

Statistics	Natural Language	Assembly Language
Unique Statements	3,184	2,248
Unique Tokens	1,498	1,244
Avg. tokens per statement	9.22	4.38
Min tokens per statement	1	2
Max tokens per statement	46	30

Table 2: *Shellcode_IA32* statistics

instructions: 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 (lines 6-8 in Listing 1). It would be meaningless to consider these three instructions as separate. To address such situations, we include 510 lines ($\sim 16\%$ of the dataset) of intents that generate multiple lines of shellcodes (separated by the newline character \n). Table 1 shows two further examples of multi-line snippets with their natural language intent.

Statistics: Table 2 presents the descriptive statistics of the *Shellcode_IA32* dataset. The dataset contains 52 distinct assembly instructions (excluding function, section, and label declaration). The two most frequent assembly instructions are `mov` ($\sim 30\%$ frequency), used to move data into/from registers/memory or to invoke a system call, and `push` ($\sim 22\%$ frequency), which is used to push a value onto the stack. The next most frequent instructions are the `cmp` ($\sim 7\%$ frequency), `xor` and `jmp` instructions ($\sim 4\%$ frequency). The *low-frequency words* (i.e., the words that appear only once or twice in the dataset) contribute to the 3.6% and 7.3% of the natural language and the assembly language, resp. Figure 2 shows the distribution of the number of tokens across the intents and snippets in the dataset. We publicly share our entire *Shellcode_IA32* dataset on a GitHub repository.[1]

Size of our dataset: Our dataset contains $3,200$ instances, which may seem relatively small compared to training data available for most common NLP tasks. We note, however, that our dataset is comparable in size to the CoNaLa annotated dataset ($2,379$ training and 500 test examples), which is one of the standard datasets in code generation (for English-Python code generation) (Yin et al., 2018). Further, *Shellcode_IA32* contains a higher percent-
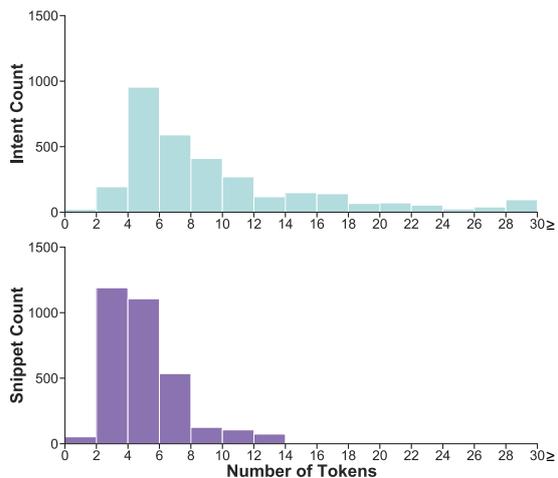

Figure 2: Histogram of the *Shellcode_IA32* dataset showcasing the distribution of token counts across intents and snippets.

age of multi-line snippets ($\sim 16\%$ vs. $\sim 4\%$). We also note here that existing code generation datasets do contain a larger, potentially noisy, subset of training examples (ranging in several thousand) obtained by mining the web. For example, the CoNaLa *mined* (as opposed to the CoNaLa *annotated*) dataset contains 598,237 training examples mined directly from Stack Overflow (Yin et al., 2018). In our case, although shellcodes are written in assembly language, it is not feasible to simply mine examples of natural language–assembly from the web: not all assembly programs are shellcodes. Thus, our *Shellcode_IA32* dataset, which contains ~ 20 years of shellcodes from a variety of sources is the largest collection of shellcodes in assembly available to date.

3 Preliminary Evaluation

We performed a set of preliminary experiments with our dataset, in order to assess the applicability

---

[1]The dataset can be found here: `https://github.com/dessertlab/Shellcode_IA32`

Number Layers	Layer Dimension	BLEU-1 (%)	BLEU-2 (%)	BLEU-3 (%)	BLEU-4 (%)	ACC (%)
1	64	75.75	69.76	65.14	60.8	34.69
	128	80.80	76.29	73.10	69.69	42.5
	256	75.50	70.50	66.65	62.86	43.75
	512	**83.55**	**80.08**	**78.06**	**76.12**	**51.25**
2	64	63.25	53.24	46.12	39.46	15.62
	128	71.79	64.24	58.25	51.65	26.25
	256	75.13	68.63	63.94	58.93	25.62
	512	**80.22**	**75.00**	**71.11**	**67.24**	**43.44**
3	64	61.98	50.68	43.02	36.15	9.38
	128	69.75	61.08	55.09	49.18	19.06
	256	**76.93**	**71.32**	**67.41**	**63.50**	**31.87**
	512	74.99	68.58	64.23	60.36	29.38
4	64	61.41	50.68	43.58	37.33	10.00
	128	63.26	51.98	44.62	37.57	10.94
	256	66.94	57.85	51.97	46.87	15.31
	512	**70.51**	**62.44**	**56.27**	**50.15**	**18.75**

Table 3: Performance results obtained by varying the model hyper-parameters. The best performances for each number of layers are in bold.

of NMT in the context of shellcode generation and to establish baseline performance levels for evaluating techniques for future research. Similar to the encoder-decoder architecture with attention (Bahdanau et al., 2015), we use a bi-directional LSTM as the encoder to transform an embedded intent sequence $E = |e_1, ..., e_{T_S}|$ into a vector c of hidden states with equal length. We implement this architecture with Bahdanau-style attention (Bahdanau et al., 2015) using xnmt (Neubig et al., 2018). We use an Adam optimizer (Kingma and Ba, 2015) with $\beta_1 = 0.9$ and $\beta_2 = 0.999$. The last step is inference. During inference, the auto regressive inference component uses beam search with a beam size of 5. The train/dev/test split is train (N = 2560), dev (N = 320), and test (N = 320) using a random 80/10/10 ratio. The test set includes 44 multi-line snippets (13.75% of the test set).

Following prior work in this area (Ling et al., 2016; Yin and Neubig, 2017; Oda et al., 2015), we evaluate the translation performance in terms of averaged token level BLEU scores (Papineni et al., 2002). BLEU uses the modified form of n-grams precision and length difference penalty to evaluate the quality of the output generated by the model compared to the referenced one. BLEU measures translation quality by the accuracy of translating ngrams to n-grams, for values of n usually ranging between 1 and 4 (Han, 2016; Munkova et al., 2020). We measure the performance of the evaluation task also in terms of exact match accuracy (ACC), which is the fraction of exactly matching

samples between the predicted output and the reference (Yin and Neubig, 2017). Both metrics range between 0 and 1.

During our experiments, we set a basic configuration of the model: $\alpha = 0.001$, layers = 1, vocabulary size = 4,000, epochs (with early stopping enforced) = 200, beam size = 5, *minimun word frequency* = 1. Next, we performed experiments by varying the dimensionality of the layers from 64 to 1024, and the number of layers from 1 to 4 while keeping all other hyper-parameters constant. Table 3 summarizes the results. We notice that increasing the number of layers leads to worse performance, while a layer dimension set between 256 and 512 is found to be the best option.

All experiments were performed on a Linux OS running on a virtual machine with 8 CPU cores and 8 GB RAM. The computational times are highly dependent on the model hyper-parameters, and range between few minutes to ~ 105 minutes, with the average training time equal to ~ 28 minutes.

4 Qualitative Analysis

Automated metrics (BLEU and accuracy) provide a somewhat limited window into the efficacy of the models to accomplish our task: the task of automatically generating assembly code from natural language intents. We conducted a qualitative analysis of the outputs to address this issue and present our findings through cherry- and lemon-picked examples from our test set (Table 4). In particular, we manually expected the outputs predicted by the

Natural Language	Ground Truth	Model Output

Put ASCII /bin/sh into eax	`push 0x68732f2f \n`
	`push 0x6e69622f \n`
	`mov eax , esp`
		`push 0x68732f2f \n`
		`push 0x6e69622f \n`
		`mov eax, esp`

Place address buff into esi	`mov esi, buff`	`lea esi, [buff]`
Perform a bit-wise inversion of edx	`not edx`	**`and`** `edx,` **`0`**
if the contents of the bl register is greater than or equal to the value 78h then jump to the memory location loc_402B1D	`cmp bl, 78h \n jge` `short loc_402B1D`	`cmp bl, 78h \n` **`jle`** `short loc_402B1D`

Table 4: Illustrative examples of correct and incorrect output. The prediction errors are **red/bold**.

best model configurations found in Table 3 (layers number = 1, layer dimension = 512).

The first two rows of Table 4 are illustrative examples of categories of intent – snippet pairs that the model can successfully translate. The first row demonstrates the ability of the model to generate multi-line snippets from a relatively abstract intent. The example in the second row shows the model's ability to properly use the instruction `lea` with the correct addressing mode (specified by the bracket [] in NASM syntax) to translate the intent. We note here that although the output would be considered incorrect based on automated metrics (e.g. BLEU-4), it is considered correct using manual inspection.

We also highlight problems with the models through illustrative examples of failure outputs (Rows 3 and 4, Table 4). In the third row of the table, the model generates the wrong instruction due to the model's failure in using implicit knowledge (i.e. the bit-wise inversion to negate the contents of the register) because it was not explicitly mentioned in the intent. Row 4 illustrates the model's failure in predicting the right command among fifteen different conditional jumps in the dataset (`jle` instead of `jge`) in an if-then statement. To summarize, the failures we observed are caused either by a lack of implicit intent knowledge, the model generating incorrect instruction/identifiers (i.e., register names, labels, etc), or even both.

5 Ethical Considerations

Recognizing that attackers use exploit code as a weapon, it is important to specify that the goal of the *proof-of-concept* (POC) exploits is not to cause harm but to surface security weaknesses within the software. Identifying such security issues allows companies to patch vulnerabilities and protect themselves against attacks.

Offensive security is a sub-field of security re-

search that employs ethical hackers to probe a system for vulnerabilities or can be a technique used to disrupt an attacker. *Automatic exploit generation* (AEG), an offensive security technique, is a developing area of research that aims to automate the exploit generation process and to explore and test critical vulnerabilities before they are discovered by attackers (Avgerinos et al., 2014). Indeed, studying exploits on compromised systems can provide valuable information about the technical skills, degree of experience, and intent of the attackers who developed or used them. Using this information, it is possible to implement measures to detect and prevent attacks (Arce, 2004).

6 Conclusion

We address the problem of automated exploit generation through NLP. We use Neural Machine Translation to translate the natural language intents into assembly code. The contribution in this work is a new dataset, *Shellcode_IA32*, containing 3,200 pairs of instructions in assembly language code snippets and their corresponding intents in English. These assembly language snippets can be combined together to generate attacks or exploits on Linux OS running on Intel Architecture 32-bit machines.

Shellcode_IA32 represents a first step towards the ambitious goal of automatically generating shellcodes from natural language. Our experimental evaluation has shown promising early results, demonstrating the feasibility of generating assembly code instructions with high accuracy.

Acknowledgements

References

Iván Arce. 2004. The shellcode generation. *IEEE security & privacy*, 2(5):72–76.

Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J. Schwartz, Maverick Woo, and David Brumley. 2014. Automatic exploit generation. *Commun. ACM*, 57(2):74–84.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473.

Tiffany Bao, Ruoyu Wang, Yan Shoshitaishvili, and David Brumley. 2017. Your exploit is mine: Automatic shellcode transplant for remote exploits. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 824–839. IEEE.

Ondřej Bojar, Rajen Chatterjee, Christian Federmann, Yvette Graham, Barry Haddow, Matthias Huck, Antonio Jimeno Yepes, Philipp Koehn, Varvara Logacheva, Christof Monz, et al. 2016. Findings of the 2016 conference on machine translation. In *Proceedings of the First Conference on Machine Translation: Volume 2, Shared Task Papers*, pages 131–198.

Jeff Duntemann. 2000. *Assembly language step-by-step: programming with DOS and Linux*. John Wiley & Sons.

Jeff Duntemann. 2011. *Assembly language step-by-step: Programming with Linux*. John Wiley & Sons.

Exploit Database Shellcodes . Accessed: 2021-04-22. exploit-db.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. MIT press.

Kelvin Guu, Panupong Pasupat, Evan Liu, and Percy Liang. 2017. From language to programs: Bridging reinforcement learning and maximum marginal likelihood. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1051–1062.

Lifeng Han. 2016. Machine translation evaluation resources and methods: A survey. *arXiv preprint arXiv:1605.04515*.

Diederik P. Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*.

Daniel Kusswurm. 2014. *Modern X86 Assembly Language Programming*. Springer.

Xi Victoria Lin, Chenglong Wang, Deric Pang, Kevin Vu, and Michael D Ernst. 2017. 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*.

Xi Victoria Lin, Chenglong Wang, Luke Zettlemoyer, and Michael D. Ernst. 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)*, Miyazaki, Japan. European Language Resources Association (ELRA).

Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomás Kociský, Andrew W. Senior, Fumin Wang, and Phil Blunsom. 2016. Latent predictor networks for code generation. *CoRR*, abs/1603.06744.

Zhongxin Liu, Xin Xia, Ahmed E Hassan, David Lo, Zhenchang Xing, and Xinyu Wang. 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*, pages 373–384.

Reginald Long, Panupong Pasupat, and Percy Liang. 2016. Simpler context-dependent logical forms via model projections. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1456–1465.

Joshua Mason, Sam Small, Fabian Monrose, and Greg MacManus. 2009. English shellcode. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 524–533.

Dasa Munkova, Petr Hajek, Michal Munk, and Jan Skalka. 2020. Evaluation of machine translation quality through the metrics of error rate and accuracy. *Procedia Computer Science*, 171:1327–1336.

Graham Neubig, Matthias Sperber, Xinyi Wang, Matthieu Felix, Austin Matthews, Sarguna Padmanabhan, Ye Qi, Devendra Sachan, Philip Arthur, Pierre Godard, John Hewitt, Rachid Riad, and Liming Wang. 2018. XNMT: The eXtensible neural machine translation toolkit. In *Proceedings of the 13th Conference of the Association for Machine Translation in the Americas (Volume 1: Research Track)*, pages 185–192, Boston, MA. Association for Machine Translation in the Americas.

Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 574–584. IEEE.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics.

pwntools. Accessed: 2021-05-29. *pwntools*.

Shellcodes database for study cases. Accessed: 2021-04-22. shell-storm.

Tutorialspoint. Accessed: 2021-04-22. Assembly Programming Tutorial.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144.

Frank F. Xu, Zhengbao Jiang, Pengcheng Yin, Bogdan Vasilescu, and Graham Neubig. 2020. Incorporating external knowledge through pre-training for natural language to code generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 6045–6052, Online. Association for Computational Linguistics.

Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, and Graham Neubig. 2018. Learning to mine aligned code and natural language pairs from stack overflow. In *International Conference on Mining Software Repositories*, MSR, pages 476–486. ACM.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. *CoRR*, abs/1704.01696.

Pengcheng Yin and Graham Neubig. 2019. Reranking for neural semantic parsing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 4553–4559.