Learning assembly and source code semantics with masked language modeling

Weifan Jiang, Shengkai Li, Zeyu Liu
Columbia University

ABSTRACT

Decompiling, the ability to reverse-engineer compiled binaries back to source codes enables researchers to investigate issues in comprehensive and effective ways. Machine learning techniques, especially language modeling, have been successful in learning program semantics, and performing complex tasks such as matching binaries based on similarity [18]. We hope to study the effectiveness of language modeling in learning the direct semantic correlations between assembly and source codes, and ultimately translating assembly codes back to source codes.

We created a dataset of 5,736 pairs of semantically equivalent source and assembly code snippets, based on functions extracted from popular real-world open source projects. Then, we trained and evaluated a `RoBERTa` masked language model [15] on the source-assembly dataset. Finally, we discussed possible future directions to improve the performance of our model.

1 INTRODUCTION

Decompiling gives the ability to translate compiled binaries back to the source codes, enabling researchers to analyze a program more effectively and comprehensively when the source code is not available. Decompiling could be a useful tool in many scenarios, such as bug detection, virus detection, and program analysis. Decompiling has been studied for a long time, and so far, there are widely used decompilers, such as snowman [9]. However, all the decompilers using today are based on rules, which make the translated code hard to read for human engineers (e.g. giving names to variables that are hard to remember and distinguish among) and breaks the structures of the codes as a whole. It can also be easily manipulated by the anti-decompiling techniques [7].

Recent works have been successful in leveraging language modeling techniques to learn code semantics. Example works include: matching semantically similar assembly-code functions [18]; transcompiling (i.e. translating source codes in one high-level programming language, such as Python, to another, such as Java) [14]. Language modeling has the potential to learn the underlying semantics of assembly and source codes, which seems to be suitable for a decompiler; however, to the best of our knowledge, there are no widely-used decompiler based on language modeling. In this paper, we hope to show that it is possible to train language models that learns the semantic correlations between assembly and source codes. Additionally, in the future, we hope to build a well-performing decompiler based on language modeling, in terms of both accuracy and human-readability.

Contributions. Our contributions include:

- introducing a dataset of 5,736 semantically equivalent pairs of assembly and source code snippets, as well as techniques to create such dataset.
- techniques to pre-train a model which learns the semantical correlations between assembly and source codes. Such pre-trained model can be fine-tuned to become a decompiler.
- evaluation of the pre-trained model, analysis on factors which might limit its performance, and possible methods to improve the performance.

2 BACKGROUND AND RELATED WORK

2.1 Language modeling

BERT , or Bidirectional Encoder Representations from Transformers [12], is a language modeling framework based on transfer learning. It leverages bidirectional training to better capture language semantics, comparing to old-fashioned single-direction (e.g. left-to-right) language modeling. BERT contains two phases: *pre-training* and *fine-tuning*:

- *pre-training* is the procedure to train a base model with unsupervised tasks (i.e. masked language modeling, and next-sentence prediction) on a large corpus of data. The pre-trained model learns the underlying semantics of the training corpus.
- *fine-tuning* is a transfer learning process, in which the pre-trained model learns to perform certain tasks based on labeled data.

In our work, we focus on the *pre-training* stage.

Masked language modeling is one of the *pre-training* tasks included in BERT [12]. Intuitively, it learns the semantics by randomly masking parts of the input and making predictions.

RoBERTa , or Robustly optimized BERT approach [15], is built based on BERT [12] with several improvements and changes. The main differences between RoBERTa and BERT include:

- RoBERTa only includes the masked language modeling task for the *pre-training* phase, omitting the next-sentence prediction task.
- RoBERTa introduces the concept of dynamic masking: during *pre-training*, a new masking pattern is used for every input sequence each time it's fed into the model. In contrast, BERT uses 10 mask patterns for each input sequence over 40

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

epochs (i.e. 4 identically masked instances of the same input sequence are seen in the entire *pre-training* process).

In the study, it gives a carefully designed parameters that can fully activate the learning power of BERT. In general, it shows that under correct design, the masked language model *pre-training* is competitive with all other state-of-the-art methods.

FAIRSEQ is a sequence-modeling framework built with PyTorch [17], which provides an implementation of RoBERTa.

2.2 Program analysis

TREX , or Transfer-learning execution semantics [18], is a framework to learn function execution semantics from their micro-traces (i.e. under-constrained execution semantics). TREX's transfer-learning based approach is similar as described in section 2.1: first, the *pre-training* phase trains a model on a large corpus of micro-traces under the unsupervised masked language modeling task; then, the *fine-tuning* phase trains a model to match semantically similar functions on labeled data.

3 METHODOLOGY

In this section, we describe our methodologies, including data collection and model training.

3.1 Data collection

We collected the sources and assembly codes of different versions of various open source projects: binutils (2.30, 2,34, 2.35) [1], coreutils (8.15, 8.30, 8.32) [2], curl (7.71.1) [3], diffutils (3.1, 3.7) [4], findutils (4.7.0) [5], libtomcrypt (1.18.2) [6], sqlite (3.34.0) [10].

Source code. We collected source codes of our desired packages and versions from their online public releases, such as github.com [3, 6], ftp.gnu.org [1, 2, 4, 5], or their official websites [10].

Binaries. We obtained the compiled binaries of our desired packages and versions from [18]. The architecture of all binaries is x86-64. For each project and version, there are 4 compilations with different optimization levels (o0, o1, o2, o3). For some projects and versions, there are additional obfuscated compilations. Each compilation corresponds to one set of binary executables.

Assembly code. We used the `objdump` [8] tool to disassemble the binaries into assembly codes. The exact command we used was `objdump -j text -d BINARYNAME`. `-d` is to specify the output file, and `-j` is the parameter to dump information only for that specific section. In our case, we only need text as the functions part are what we want to match the source code with, which are all in .text section. Since for different architectures, different assembly code parsers may be required. Currently, we only support processing x86-64 binaries.

3.2 Source and assembly matching

In this section, we discuss our heuristics to match semantically equivalent source and assembly code snippets.

Some projects contain multiple independent binaries, and each binary can be matched to a source c-code file based on the name.

binutils-2.30-O0/addr2line.asm

```
...
0000000000031a6d <find_address_in_section>:
   31a6d:       55                      push   %rbp
   31a6e:       48 89 e5                mov    %rsp,%rbp
   31a71:       48 83 ec 30             sub    $0x30,%rsp
   31a75:       48 89 7d e8             mov    %rdi,-0x18(%rbp)
   31a79:       48 89 75 e0             mov    %rsi,-0x20(%rbp)
   31a7d:       48 89 55 d8             mov    %rdx,-0x28(%rbp)
   ...
   31b42:       eb 01                   jmp    31b45 <find_address_in_section+0xd8>
   31b44:       90                      nop
   31b45:       c9                      leaveq
   31b46:       c3                      retq
...
```

binutils-2.30/binutils/addr2line.c

```
...
static void
find_address_in_section (bfd *abfd, asection *section,
            void *data ATTRIBUTE_UNUSED)
{
 bfd_vma vma;
 bfd_size_type size;
 ...
 found = bfd_find_nearest_line_discriminator (abfd, section, syms, pc - vma,
                                              &filename, &functionname,
                                              &line, &discriminator);
}
...
```

Figure 1: Matching source and assembly functions by name.

For example, project binutils is a collection of tools (e.g. addr2line, ar, objcopy, etc.) and each tool is available as an independent binary executable. In the source code repository of binutils, there is a corresponding c-code script for each executable (e.g. addr2line.c). In this case, we extract functions from the disassemble output of the executable as well as the corresponding c-code script, then match functions based on the name. Figure 1 is an example.

For other projects such that each binary cannot be matched to one single c-code source file (e.g. projects with only one output executable such as sqlite), we extract functions from the disassemble output, and look for a c-code function with the same name from all source code scripts in the repository. In practice, we developed the following rules to increase the accuracy of function matching:

- If there is a sub-directory inside the source code repository named `src`, we assume such directory contains the majority of semantically significant source codes, and we only look for matched functions extracted from the `src` directory.
- If there are non-unique matchings (i.e. multiple source code functions can be matched to the same assembly code function, or one source code function can be matched to multiple assembly code functions), we discard such matchings from the dataset.

Table 1 contains detailed information on our assembly-source code dataset. One observation is that the number of matched function pairs are usually only a very small fraction of all assembly and source code functions.

3.3 Input preparation

In this part, we discuss how matched source and assembly functions are converted to inputs for training.

Formatting. BERT [12] suggested that input to the *pre-training* phase could be in form of concatenated sentence pairs, such as a question and answer pair. During the *pre-training* phase, the model

	package/version	compilation configurations (optimizations/obfuscations)	assembly code functions	source code functions	pairs matched
	binutils-2.30	o0, o1, o2, o3	44	1188	8
	binutils-2.34	o0, o1, o2, o3 bcfobf, cffobf, indibran, splitobf, subobf	103	3247	10
	coreutils-8.15	o0, o1, o2, o3	384	4272	152
	coreutils-8.30	o0, o1, o2, o3	391	4940	165
	coreutils-8.32	o0, o1, o2, o3 bcfobf, cffobf, funcowa, indibran, splitobf, subobf	978	12986	160
	diffutils-3.1	o0, o1, o2, o3	13	286	2
	diffutils-3.3	o0, o1, o2, o3	16	340	4
	diffutils-3.7	o0, o1, o2, o3 bcfobf, cffobf, funcowa, indibran, splitobf, subobf	40	880	2
	libtomcrypt-1.18.2	o0, o1, o2, o3 acdobf, bcfobf, cffobf, fco, funcwra, indibran, orig, splitobf, strcry, subobf	5411	14070	5226
	sqlite-3.34.0	o0, o1, o2, o3 bcfobf, cffobf, indibran, splitobf, subobf	9	35343	7
total					5736

Table 1: Details on the semantically equivalent assembly-source code snippet dataset. The "compilation configuration" column includes all compilations produced for each package/version. Numbers in the last three columns are aggregated from all compilations. Packages/versions which didn't produce any matched pairs were omitted.

can learn the underlying semantic correlation between the pair of sentences. TREX [18] also concatenated semantically correlated information (e.g. micro-trace code, micro-trace value, position, architecture, numeric value encoding) into one token sequence as input to the *pre-training* phase.

We take a similar approach by concatenating each matched source and assembly code snippet pair to one sequence. For assembly codes we use the hex representations of the instructions, which are naturally divided into tokens, and for c source codes, we split the codes into tokens by spaces. We finally concatenate them in assembly-source order.

Encoding. We employ BPE, or Byte-Pair Encoding [20] to encode the input data. BPE is a technique which combines character-level and word-level representation to deal with the long and uncommon words in the natural language. We use the same BPE procedure as RoBERTa [15].

Preprocessing. Lastly, we binarize the input data with GPT-2 dictionary [19]. This process is adopted from `FAIRSEQ` [16].

3.4 Training

In this part, we briefly discuss how we leverage the *pre-training* technique to learn a model which understands semantics of assembly and source codes. The input to the *pre-training* phase is produced as described in section 3.3. We employ the `FAIRSEQ` [16] implementation of RoBERTa [15] as the training framework.

We use the masked language modeling option as training task, with the following hyperparameters:

- tokens-per-sample = 512: This is to ensure enough running efficiency. However, as some of functions can be long, some training data would be skipped due to this constrained. Since we are using concatenation for data encoding, this constraint would affect more samples.
- batch-size = 32: As shown in [13], a over-large batch size would significantly negatively affect the performance of the model. Therefore, we choose a relatively small batch size to ensure better performance.
- learning-rate = 0.01/0.005/0.001: We use different learning rate to see how the learning rate affects our model training behavior.

We further evaluate the choice of parameters and model performance in section 4.

4 EVALUATION

In this section, we first describe the settings of our experiment. Then, we include evaluations of our model based on standard language modeling metric and a case study. Then, we analyze possible factors which might limit the performance.

4.1 Settings

The *pre-training* was performed on two Google Compute Engine instances (due to computational resource limit) with the following configurations:

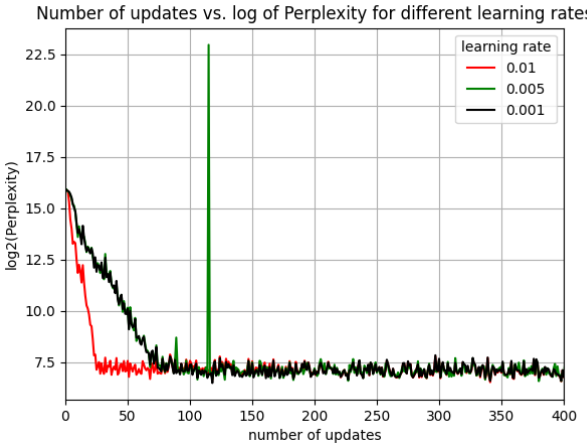

Figure 2: Number of updates vs. log of perplexity (PPL) for different learning rates.

- 16 × Intel(R) Xeon(R) CPU @ 2.30GHz, ubuntu 18.04 system, 128G Memory. (For learning rate 0.01 and 0.001)
- 16 × Intel Xeon E5 v4 (Broadwell E5) 2.2 GHz, ubuntu 20.04 system, 64G Memory. (For learning rate 0.005)

We partitioned our dataset to train (80%), validation (10%) and test (10%). We decided to put majority of the data into the training set to ensure the model has enough resources to learn from. The hyper-parameters were as described in section 3.4. We let the *pre-training* run for 11 epochs, since we observed that the loss becomes steady around that stage.

4.2 Language modeling metric analysis

We evaluate the pre-trained models with the perplexity metric:

Perplexity. Same as related work [18], we evaluate our pre-trained models with the standard language modeling metric *Perplexity* (PPL). PPL measures how well a given model describes the data. Lower PPL indicates better models, and 1 is the best possible PPL. Note that the *pre-training* procedure in `FAIRSEQ` uses the log of PPL as the loss function.

Figure 2 shows how PPL changes with the number of updates to the model weights during the *pre-training* stage when different learning rates (i.e. 0.01/0.005/0.001) were used. Note the PPL is shown in log scale. We observe that the training is effective, as the PPL steadily decreases for the first few updates, and converges around 181 (7.5 on the $\log_2$ scale), for all learning rates used. We can see that different learning size does not give much difference in terms of minimum. After about 75 updates, the ppl is not decreasing for all learning rates. We can see that for our largest learning rate 0.01, it converges much faster, which is as expected.

Table 2 shows the best loss and PPl values achieved for different learning rates. Based on NLP domain knowledge, we know that the PPLs in table 2 indicate the models have poor performance. For reference, TREX's [18] pre-trained models achieve PPL values between 2 and 2.40 on the testing set.

We conclude that, our approach is effective such that the *pre-training* successfully reduces PPL of the models. However, such effectiveness is not sufficient, as the minimum PPL values achieved are much still higher than we expected.

4.3 Case study: filling mask

In addition to evaluating with standard metrics, we evaluated our pre-trained models with a more concrete task: filling mask. We fed token sequences from the test set to the pre-trained model, with one token in the sequence replaced by the special symbol `<mask>`. The model then tries to recover the masked token based on the rest of the input sequence. This functionality is provided by the `FAIRSEQ` framework [16].

For this step, we tried to select short input sequences, since we assumed shorter sequences have simpler semantic contexts and are easier to be predicted. In addition, we tried to mask frequently seen tokens such as `return`. Such practices were in an effort to make the task easier for the model. However, our models still failed to make the correct prediction for all test cases we selected.

The result of our case study shows that our pre-training is insufficient, and we further discuss it in section 4.4.

4.4 Analysis of performance

As we can see from the figure 2, all the learning rate would result in a minimum around 7.5 for log(ppl). However, the loss and ppl are still very high at the minimum points. The evaluation (mask filling) also reflects a similar problem. Therefore, here, we discuss some possible factors which limited our pre-trained model's performance, in the order of decreasing importance.

Insufficient training data. Our *pre-training* dataset size is 4,572 (i.e. 80% of the entire assembly-source matching dataset). With the limitation of tockens-per-esample, this number further reduce. The total number of traning data is significantly smaller than related work (e.g. TREX [18] used more than 1.4 million sequences for *pre-training*). `BERT` also suggested that large pre-trained models lead to effective fine-tuned models. Our dataset size is nowhere close to the actual popular datasets for *pre-training* (e.g. ImageNet [11] has 1,281,167 samples in the training set).

Inaccuracy in training data. As discussed in section 3.2, we used heuristics to match semantically equivalent source and assembly code snippets. Such technique may produce incorrect results, especially when a binary executable cannot be associated with only one c-code file.

Computation resource limitation. We performed the training purely on CPUs, which would not be practical if we actually had a sufficiently large training set. Usually GPU is required for such tasks, and our Google Cloud account quota forbid us to create GPU instances.

Hyperparameters. We used the default value for many hyperparameters while *pre-training*, such as `dropout` and `weight-decay`. Alternating such hyperparameters may result in improving our model's performance. Without enough data and computational resources, we was not able to adjust our parameters with different settings, and for parameters we adjusted also have limited options.

learning rate	best loss	best PPL
0.01	6.496	90.23
0.005	6.546	93.43
0.001	6.486	89.64

Table 2: Minimum loss and perplexity (PPL) achieved for different learning rates.

5 CONCLUSION

In this paper, we introduced a new dataset of 5,736 pairs of semantically equivalent assembly and source code snippets. We described our approach in creating such dataset, so its content could be further enlarged by future works to improve practical usability. Based on our new dataset, we described our technique for *pre-training* a model that learns the semantic correlations between assembly and source code with an unsupervised masked language modeling task. We evaluated the our pre-trained model and analyzed reasons which could limit its performance. We hope that after the performance of our pre-trained model is improved, we could emply transfer learning by *fine-tuning* our model on labeled data to achieve direct assembly-to-source translation. We believe such an assembly-source translation tool can help security researchers reverse-engineer binaries more efficiently.

5.1 Future steps

In section 4.4, we discussed various factors which limited the performance of our pre-trained model. To tackle those challenges, we propose the following as future steps.

- We plan to obtain enough training data. We plan to enlarge the data size by considering more open source projects and improving our matching heuristic (recall that table 1 shows that only a small fraction of functions could be matched for most packages). A goal is to include at least 100,000 pairs. We will check our heuristics in more detail to ensure the correctness of our collected data.
- It's possible to augment our dataset with additional information other than assembly and source codes. For example, TREX [18] learns semantic context from micro-traces. We could augment each assembly-source pair with additional semantic information (such as micro-traces) to help the model learn during *pre-training*. However, this remains a hypothesis and we are not certain whether it would be useful.
- We plan to obtain enough computational resources. For example, we are going to rent an instance with enough GPU power for a more thorough training.
- We plan to conduct more comprehensive experiments on more *pre-training* hyperparameters and select the optimal ones.
- We may also try to alternate the format of our data. Currently, as discussed in section 3, we concatenate our assembly and source code. We can also try to process them separately and training a translation model with RoBERTa. We will also try to use dictionary from [18] for our assembly hex representation.
- After we obtain a better-performing pre-trained model, we hope to perform the *fine-tuning* step on the pre-trained model to build a new model that directly translates assembly codes to source codes.

REFERENCES

[1] binutils. (????). https://ftp.gnu.org/gnu/binutils
[2] coreutils. (????). https://ftp.gnu.org/gnu/coreutils
[3] curl. (????). https://github.com/curl/curl
[4] diffutils. (????). https://ftp.gnu.org/gnu/diffutils
[5] findutils. (????). https://ftp.gnu.org/gnu/findutils
[6] libtomcrypt. (????). https://github.com/libtom/libtomcrypt
[7] .Net Anti-decompiler. (????). https://www.techipick.com/dotnet-anti-decompiler
[8] objdump. (????). https://www.linux.org/docs/man1/objdump.html
[9] Snowman. (????). https://derevenets.com/
[10] sqlite. (????). https://www.sqlite.org
[11] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 248–255.
[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423
[13] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. 2016. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. *CoRR* abs/1609.04836 (2016). arXiv:1609.04836 http://arxiv.org/abs/1609.04836
[14] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised Translation of Programming Languages. (2020). arXiv:cs.CL/2006.03511
[15] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *arXiv preprint arXiv:1907.11692* (2019).
[16] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. fairseq: A Fast, Extensible Toolkit for Sequence Modeling. In *Proceedings of NAACL-HLT 2019: Demonstrations*.
[17] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf
[18] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2020. Trex: Learning Execution Semantics from Micro-Traces for Binary Similarity. (2020). arXiv:cs.CR/2012.08680
[19] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).
[20] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Berlin, Germany, 1715–1725. https://doi.org/10.18653/v1/P16-1162