NLP-based End-to-end Code Analysis

Swati Raghav
sfr5677@psu.edu

Skanda Bharadwaj
ssb248@psu.edu

Abstract

Code analysis has been proven to be an extremely useful task and a challenging one as well. To be able to understand and make inferences from stripped binaries has been of significant interest of late. Although there has been a lot of work in tasks such as code description and code formatting, there is little work in having an end-to-end NLP-based code analysis. Towards this end, we propose to take baby steps towards understanding a given function from stripped binaries. We first use a Trex-based function name prediction engine followed by another model called Code Parrot that attempts to generate the code given an input function. In addition, we also wanted to add code formatting to the pipeline. But, we leave this for future work. To our knowledge, this pipeline seems to be novel although there has been significant research that has been done for individual components. The code is made available in: https://github.com/Skanda-Bharadwaj/CSE582-FinalProject

1. Introduction

Analysis of stripped binaries has a lot of downstream applications such as malware analysis [4], program comprehension [1], vulnerability identification [5] and many more. A lot of research has gone into understanding the code from the binaries. One of the seminal works was Trex [3], a transfer-learning-based framework that tries to find the similarity between two functions. One of the major drawbacks of this Trex is that it does not consider calling function in the traces and therefore understanding of the function behaviour is extremely restricted. Therefore, we considered a Trex-based neural architecture SymLM [2] that considers the task of function name prediction. This involves learning context-sensitive semantics. The function name prediction involves both static and dynamic analysis. Specifically, the model first generates a set of candidate names for each function based on its execution contexts, such as the surrounding code, register usage, and function calls. The model then ranks the candidates based on their similarity to the ground truth function names and selects the top-ranked name as the predicted name for the function.

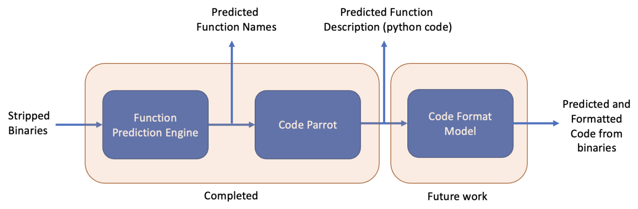

Predicted Function Names
Predicted Function Description (python code)
Stripped Binaries
Function Prediction Engine
Code Parrot
Code Format Model
Predicted and Formatted Code from binaries
Completed
Future work

Figure 1. A schematic of our proposed pipeline. We were successful in generating outputs for the first two modules. We leave the code formatting for future work. While we made an attempt to get code formatting work, in the interest of time we do not consider it done yet.

Further, we use this predicted name and input it into a model called the *Code Parrot*. CodeParrot is a GPT-2 model (1.5B parameters) trained to generate Python code.

The CodeParrot model is built using a deep neural network architecture known as the transformer. The transformer architecture is well suited for natural language processing tasks like text generation because it is able to effectively model long-term dependencies and capture patterns in the data. To generate code snippets, CodeParrot takes a prompt as input and generates a sequence of tokens that represents the code snippet. The prompt can be a description of what the code should do, or it can be a partial code snippet that the model can complete.

The input to the model is first tokenized using a tokenizer that converts the input text into a sequence of numerical tokens that can be fed into the neural network. The tokenized input is then passed through the transformer network, which generates a sequence of output tokens that represent the generated code snippet. After the code snippet has been generated, it is post-processed to convert the numerical tokens back into text and format the code in a readable way. The generated code snippet is then returned as the output of the model.

Overall, CodeParrot is a powerful tool for generating code snippets and has many potential applications in software development and programming education.

In this project, we attempt to predict a function name, generate a possible Python code and finally format the code.

We were successful in generating results for the first two modules and we leave code formatting for future work. This pipeline is represented in Figure 1.

2. Data

In order to train the function prediction engine, we chose the dataset that is curated in SymLM. While SymLM has a large database of the binaries, we specifically chose the binaries of the x86 architecture. The choice of the subset of the data is subject to the constraints of time and resources. For the creation of the dataset, Ghidra was used to parse the binaries. The dataset can easily be obtained following the links from [2] and also, can easily be created following the data preparation content from their GitHub handle. The data was split into train, test and validation classes with each of them having the classes *caller1, caller2, external_callee1, external_callee2, internal_callee1, internal_callee2, self*. We followed the general thumb of rules for the data split for training. The entire data set consists of five different groups including *arm, mips*, x64, x86, *obfuscation*, of which we considered the subset of x86.

As far as the *Code Parrot* is considered, we did not consider any data since we did not plan on training it from scratch. We used the pre-trained model and obtained the outputs.

3. Experiments

3.1. Function Name Prediction

We started off by setting up SymLM and training the model from scratch for the x86. Figure 2 represents the backbone Trex model that is used to get the semantic embeddings of the binaries. This, in turn, will help us understand functions that have similar execution behaviour. With this backbone, the binaries are preprocessed and prepared for training (in the interest of time and space, we have kept the explanation limited).

3.1.1 Challenges

We began training SymLM network from scratch for the x86 dataset. There were multiple challenges that we faced during the experimentation. First and foremost, the requirement to accelerate the training was very hard to tackle. Although we had found a decent GPU to train the model, we did not estimate the challenges with respect to GPU acceleration that was required. Having said that, we continued to train our model and trained it or a few epochs (3 5) with each of the epochs taking more than 9 hours of time. Figure 3 represent our attempt to train the network. Details of the results are presented in section 4

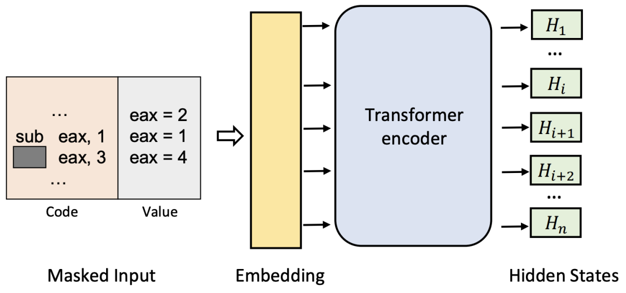


Figure 2. Here is the SymLM model that has been directly borrowed from [2]. We tried training this model for x86 binary dataset.

Method	Precision	Recall	F1
Self-Trained (3 epochs)	0.968	0.199	0.318
Pretrained Model	0.731	0.892	0.756

Table 1. The table represents the scores for our experiments on the function name prediction module. Row 1 represents the values for our attempt to train the model from scratch on x86 data. Row 2 represents the values for the pre-trained values for the same data set.

3.2. Code Generation

Going forward in the pipeline, we input the predicted function into the code parrot and obtain the Python code that, in a way, describes the code. As mentioned earlier, code parrot is a pre-trained model and all we do is simply input the function into the code parrot for it to generate the code. Some of the results are shown in section 4.

4. Results

In this section, we present the results obtained from different experiments that we conducted. Table **??** represents the scores that we obtained for our function name prediction model. As can be seen, the scores are pretty low for the first row. It is an experiment where we ran the model for just a couple of epochs. It can be seen from Figure 3d that the scores are improving through iterations. Although the precision indicates that it is more than the pre-trained model, we do not believe that it reflects the true precision. This only reflects the initial values of the training.

Our experiments with *Code Parrot* were pretty straightforward. We imported the code parrot model into our code and input the function names and we got desired results for many of the functions. Here are certain outputs of the code parrot model. Figure 4 represents the outputs for many input functions that are explained below.

Important Note: It should be noted that the function names were engineered by hand post the output of the func-

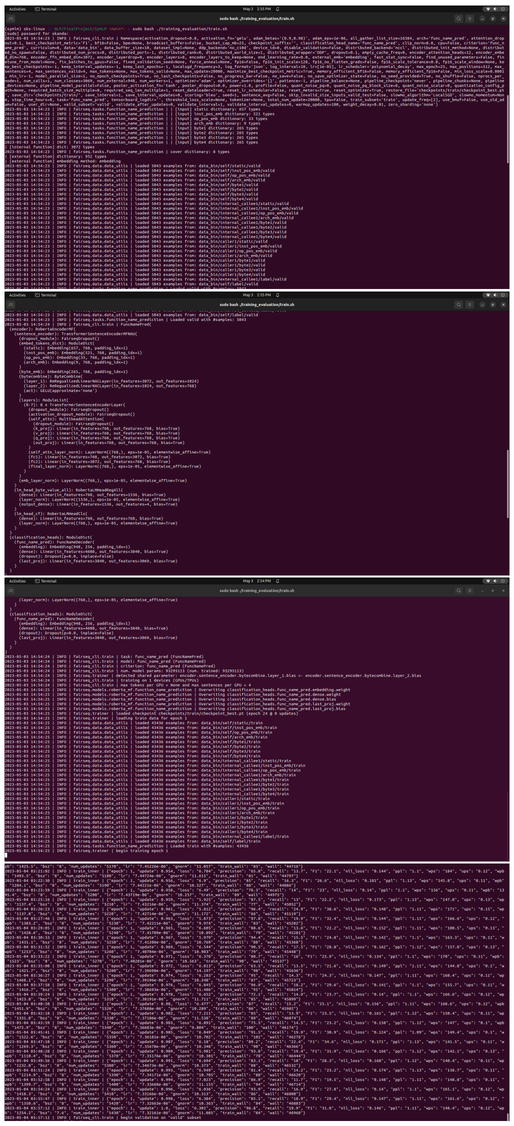
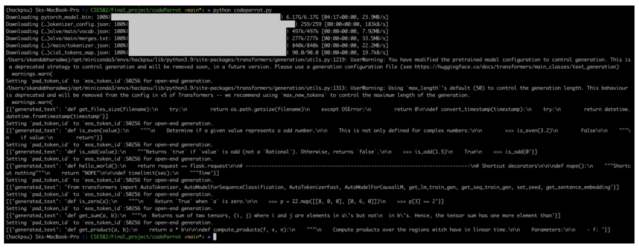


Figure 3. Images showing the model and training. It can be seen that the F1, precision and recall scores are increasing through iterations. unfortunately, the lack of resources restricted us from improving the model.

tion name predictor since our model is not sophisticated enough yet. We added a few keywords such as *def* in order to get the output. In other words, the pipeline is not end-to-end in a strict sense.

As far as the evaluation is concerned, we do not have a quantitative measure for code parrot. So we decided to go with qualitative evaluation in which we input the function name and visually inspect the outputs provided by code parrot. Making the model ready for the quantitative evaluation would take considerably more amount of time. Here are a few functions that we tried as examples:



```
(hackpsu) Sks-MacBook-Pro :: CSE582/Final_project/codeParrot ‹main*› » python codeparrot.py
Downloading pytorch_model.bin: 100%|                                        | 6.17G/6.17G [04:17<00:00, 23.9MB/s]
Downloading (…)okenizer_config.json: 100%|                                     | 259/259 [00:00<00:00, 183kB/s]
Downloading (…)olve/main/vocab.json: 100%|                                   | 497k/497k [00:00<00:00, 7.92MB/s]
Downloading (…)olve/main/merges.txt: 100%|                                   | 277k/277k [00:00<00:00, 33.5MB/s]
Downloading (…)/main/tokenizer.json: 100%|                                   | 840k/840k [00:00<00:00, 22.2MB/s]
Downloading (…)cial_tokens_map.json: 100%|                                   | 90.0/90.0 [00:00<00:00, 19.7kB/s]
/Users/skandabharadwaj/opt/miniconda3/envs/hackpsu/lib/python3.9/site-packages/transformers/generation/utils.py:1219: UserWarning: You have modified the pretrained model configuration to control generation. This is
 a deprecated strategy to control generation and will be removed soon, in a future version. Please use a generation configuration file (see https://huggingface.co/docs/transformers/main_classes/text_generation)
  warnings.warn(
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
/Users/skandabharadwaj/opt/miniconda3/envs/hackpsu/lib/python3.9/site-packages/transformers/generation/utils.py:1313: UserWarning: Using `max_length`'s default (50) to control the generation length. This behaviour
is deprecated and will be removed from the config in v5 of Transformers -- we recommend using `max_new_tokens` to control the maximum length of the generation.
  warnings.warn(
[{'generated_text': 'def get_files_size(filename):\n    try:\n        return os.path.getsize(filename)\n    except OSError:\n        return 0\n\ndef convert_timestamp(timestamp):\n    try:\n        return datetime.
datetime.fromtimestamp(timestamp'}]
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
[{'generated_text': 'def is_even(value):\n    """\n    Determine if a given value represents a odd number.\n\n    This is not only defined for complex numbers:\n\n        >>> is_even(3.2)\n        False\n\n    """\
n    if value:\n        return'}]
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
[{'generated_text': 'def is_odd(value):\n    """Returns `true` if `value` is odd (not a `Rational`). Otherwise, returns `false`.\n\n    >>> is_odd(1.5)\n    True\n    >>> is_odd(0'}]
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
[{'generated_text': 'def hello_world():\n    return request == flask.request\n\n# ------------------------------------------------------------------------------\n# Shortcut decorators\n\n\ndef nope():\n    """Shortc
ut nothing"""\n    return "NOPE"\n\n\ndef timelimit(sec):\n    """Time'}]
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
[{'generated_text': 'from transformers import AutoTokenizer, AutoModelForSequenceClassification, AutoTokenizerFast, AutoModelForCausalLM, get_lm_train_gen, get_seq_train_gen, set_seed, get_sentence_embedding'}]
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
[{'generated_text': 'def is_zero(a):\n    """\n    Return `True` when `a` is zero.\n\n    >>> p = ZZ.map([[8, 0, 0], [0, 6, 0]])\n    >>> p[3] == 2'}]
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
[{'generated_text': 'def get_sum(a, b):\n  """\n  Returns sum of two tensors, (i, j) where i and j are elements in a\'s but not\n  in b\'s. Hence, the tensor sum has one more element than'}]
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
[{'generated_text': 'def get_product(a, b):\n    return a * b\n\n\ndef compute_products(f, x, n):\n    """\n    Compute products over the regions witch have in linear time.\n\n    Parameters:\n\n    - f: '}]
(hackpsu) Sks-MacBook-Pro :: CSE582/Final_project/codeParrot ‹main*› »
```

Figure 4. Here is the SymLM model that has been directly borrowed from [2]. We tried training this model for x86 binary dataset.

1. *Get_File_Name:* We got the expected output here.

```
def get_files_size(filename):
try:
return os.path.getsize(filename)
except OSError:
return 0
```

2. *Is_Even:* We got the expected output here.

```
def is_even(value):
return value % 2 == 0
```

3. *Is_Odd:* We got the expected output here.

```
def is_odd(value):
return value % 2 != 0
```

4. *Get_Product:* We got the expected output here.

```
def get_product(a, b):
  return a * b
```

5. *Is_Zero:* We got the expected output here.

```
def is_zero(a):
WRONG OUTPUT
```

6. *Get_Sum:* We got the expected output here.

```
def get_sum(a, b):
WRONG OUTPUT
```

4.1. Discussions

The first thing that we want to mention is that getting the SymLM network to work was a pretty hefty task. Given the complexity of the network, it took us quite a lot of time to set up the system. When we did set it up, we did not estimate the importance of needing an accelerator since we had already found the GPU. We found out that we needed something called the Nvidia Apex to increase the training speed. While we attempted to install this we had several dependency issues and we had to stop working on the acceleration part at some point. We decided to train the network without the accelerator and that took quite a bit of time with 1 epoch taking almost 9+ hours to train. Hence, we decided to train the network for only a few epochs to see the trend in loss and accuracy.

In the context of *Code Parrot*, we have described whatever output we got from code parrot when we input the function name in the above code listing. WRONG OUTPUT simply means that the output from code parrot was wrong. It is interesting to see that some functions such as *get_file_size* worked perfectly while a simple function such as *get_sum* did not work. But, *get_product* worked. Some strange observations.

We did try to work on Code Formatting model as well, but could not successfully complete it. So we have chosen to leave it out of the project and could potentially be considered for future work. In addition, we also tried different models such as *Salesforce* model to check different code generation codes. But, code parrot seemed to work the best for us.

5. Conclusion

In conclusion, this was a fun project and an opportunity for us to explore something new. We only have basic experience with NLP and program analysis, yet the idea of end-to-end code analysis with NLP was very captivating. Although the number of results captured seems less, we did try different models, such as Trex, and SymLM in order to get this working. The initial phase was troublesome until we had the model up and running. But once the model started working we had fun exploring the subsequent modules. We wish we had a few more days to complete the code formatting model too so that we could have finished the entire pipeline that we thought of in the beginning of the project.

References

[1] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[2] Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1631–1645, 2022.

[3] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. Trex: Learning execution semantics from micro-traces for binary similarity. *arXiv preprint arXiv:2012.08680*, 2020.

[4] Monirul I Sharif, Andrea Lanzi, Jonathon T Giffin, and Wenke Lee. Impeding malware analysis using conditional code obfuscation. In *NDSS*, 2008.

[5] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, volume 1, pages 1–1, 2015.