Binary2Name – Automatic Detection for Binary Code Functionality

Advisors: Dr. Gabi Nakibly, Dr. Yaniv David

236349 – Project in Computer Security

Final Report Spring 2022

Presenter: Tal Shadmi

Table of contents

Table of contents 2

Introduction 3

Overview 3

Motivation 3

Background 4

Nero 4

Literary Review 5

Datasets 6

The Angr Tool 6

Last Project Iteration 7

Main Goal 8

Stages Of Development 9

Getting Started 9

Running Into Technical Issues 9

Getting First Results 10

New Constraints Styling Ideas 10

New Constraints Picking Ideas 11

Changing The Pipeline Process 14

Filtering Out Functions 15

Using A Pre-Made Split to The Nero dataset 17

Conclusions 18

Main Achievements 18

Results Reasoning 18

Future work 19

References 20

Introduction

Overview

This project is the fourth iteration of a project started by Reda Igbaria and Ady Agbaria, with Dr. Gabi Nakibly as an advisor. The project's second iteration was executed by Carol Hanna and Abdallah Yassin and the third by Ittay Alfassi and Itamar Juwiler, both also advised by Dr. Gabi Nakibly. Building on last iteration's git repository, published by Ittay and Itamar, and all the project iteration's reports over the years, I've been trying to improve the model in hopes of achieving better model performance. Like my predecessors, my work in this project aimed to automatically detect the functionality of a binary code snippet. Given the binary code of a function as input, my goal was to output a name for the function that accurately describes its functionality. I started from binary datasets and used Angr, a symbolic analysis tool to get intermediate representation of the code. From there, came the most extensive step in the project, which was to preprocess, style and convert the intermediate code in preparation to be used as input to a neural network. I used a deep neural network adopted from the paper: Neural Reverse Engineering of Stripped Binaries [2], which is intended for the same goal, using a new approach to the problem developed by the last iteration of this project. After exploring last iteration's model and scripts, and tackling some core issues in them, I moved on to try and manipulate the symbolic execution output created by Angr in different ways, hoping to achieve better results on the Coreutils and Nero datasets that were used and tested by my predecessors. My advisors in this iteration of the project were Dr. Gabi Nakibly, the advisor of the previous iterations, and Dr. Yaniv David, which is one of the creators of the Nero model, which was used in the project.

Link to Project GitHub: https://github.com/tal-shadmi/binary2name

Motivation

The main motivation for this project is to be a helpful tool for researchers of binary code. When analyzing a large binary, usually the researcher is only interested in short snippets that depict an interesting algorithm. The "gem" of the binary is often hidden in a very large puzzle. Having a tool that automatically identifies the functionality of different snippets, can ultimately be utilized by researchers as a tool to help them identify target snippets of code so that research resources can be utilized efficiently. It would save the researcher a lot of time and effort in looking for the "Crown Jewel" of the binary.

Background

Nero

The model used for the training was adopted straight from the paper “Neural Reverse Engineering of Stripped Binaries (2019) By Eran Yahav, Uri Alon, and Yaniv David” [2]. In this paper they present an alternative approach to predicting procedure names in binary code. The method used in this paper is reliant on the sequence of external API calls, given the intermediate representation of the binary code, the paper’s approach is to do static analysis of the code and detect the calls to external functions and their names and parameters. After this analysis they build a Control-Flow Graph containing data about the API calls sites. This graph will be the input to the prediction model which is based on encoder decoder attention model. The NN model is responsible to recognize the pattern and the behavior of the API calls and predict the function tag according to them.

Taken from the paper is the following diagram:

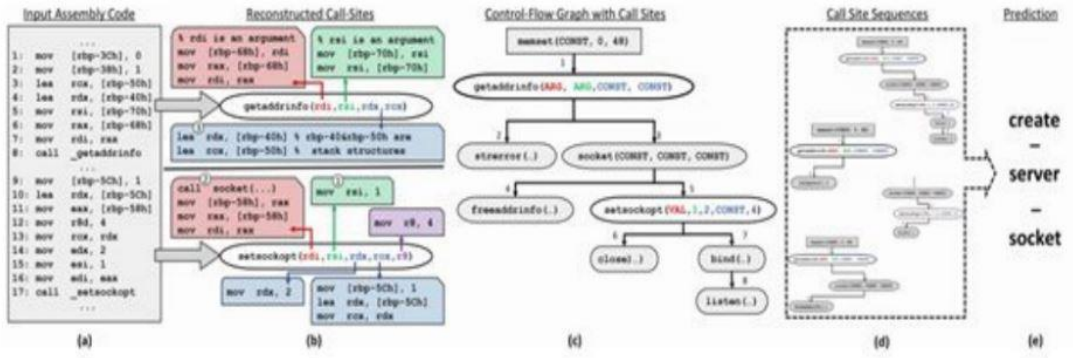

Figure 1: Nero's algorithm pipeline

As shown in the diagram and explained above, this is the entire process depicted in the paper.

Firstly, the code is analyzed, and the API and external function calls are reconstructed (a → b). Secondly, the CFG is constructed with data containing the natural API’s names (b → c). Lastly, the CFG is inserted as input to the prediction model (which calculates over all the possible paths) (c → d).

Literary Review

Neural Software Analysis [5]

This article gives an overview of neural software analysis, discusses when to use it or not, and presents three example analyses. The analyses address challenging software development problems: bug detection, type prediction, and code completion, and shows examples tackling those problems using different neural software analysis tools. The Nero model is mentioned in this article as a tool designated to the reverse engineering analysis problem and as such complement and outperform traditional program analyses and is used in industrial practice.

At the end of the article the authors discuss different open challenges the field still has. One of the main challenges they discuss is the challenge of providing quality, model suited input to the different models to maximize their performance. At this project we continue to do exactly that, trying to improve the Nero model input to maximize it's potential.

Learning to Find Usages of Library Functions in Optimized Binaries [6]

This article presents optimizing function calls, using steps such as inlining, as one of the central steps in creating optimized binaries. The article claims that recovering these (possibly inlined) function calls from optimized binaries is an essential task that most state-of-the-art decompiler tools try to do but do not perform very well.

As part of the discussion in this article the authors evaluate a supervised learning approach to the problem of recovering optimized function calls and as part of that also refer to “Neural Reverse Engineering of Stripped Binaries (2019) By Eran Yahav, Uri Alon, and Yaniv David”.

The article authors claim that the Nero model (among other models) does not handle inlined functions very well, therefore does not handle binaries that were created by compiling with optimizations.

At this project we try using a different approach and use constraints instead of call-sites as an input to the Nero model, we can only hope that because optimization of function calls affects the constraints of the function less then it affects its call-sites, this change of input to the Nero model would improve its performance also when facing optimization of code problem presented in this article.

Datasets

In this project I kept using the datasets that were used in the last iterations of this project: the Nero dataset, which is the main dataset from the original paper, and the Coreutils dataset, which is a smaller dataset, used mostly to test my ideas more quickly.

Coreutils Dataset

Coreutils is a package of GNU software containing implementations for many of the basic UNIX tools. It includes about 1100 functions.

Nero Dataset

The Nero dataset is a package containing Intel 64-bit executables running on Linux. The paper authors collected a dataset of software packages from the GNU code repository containing a variety of applications such as networking, administrative tools, and libraries. To avoid dealing with mixed naming schemes, all packages containing a mix of programming languages, e.g., a Python package containing partial C implementations, were removed. This dataset contains about 60000 functions.

the datasets were split into train, test, and validate sections in a default division of 0.7, 0.2 and 0.1 accordingly. However, the scripts are configurable, and Those percentages can be changed easily.

The Angr Tool

Angr [4] is a binary analysis toolkit, that combines both static and dynamic symbolic analysis. The two main tools Angr offers that were used in the project are Control-Flow Graph (CFG) recovery and symbolic execution.

Last iteration used the symbolic execution outputs of the binary code to train a deep learning model. To do this, their preprocessing used Angr on a library of binaries (the datasets) which gave them control-flow graphs of assembly code, augmented with symbolic constraints, which then got preprocessed further before using as input. When symbolic variables are present in the graph, Angr outputs constraints on the variables. These constraints give:

a) A range of values that the symbolic variable can hold.
b) Relation between different symbolic variables in the form of Boolean constraints.

Last Project Iteration

As a continuation of the last project iteration, I've been instructed to continue and explore the Angr symbolic execution output as the input core info passed to the Nero model, specifically the constraints output which was the focus of the last iteration to this project.

Previous iteration of this project were the first ones to explore the Angr symbolic execution, therefore built the foundations and basic method to integrate it as the new input for the Nero model and had less focus on optimizing this input for better results.

The Binary2Name algorithm they developed was comprised of four stages.

The stages are as follows:

1. Performing Symbolic Execution on the binary functions using the Angr framework, constructing a basic CFG per function (preprocessing).
2. Processing the CFG's constraints, extracting valuable data and adapting to the Nero data format (styling).
3. Creating the Nero vocabularies and preparing for Nero's activation (Nero preprocessing).
4. Running Nero on the processed data. (Nero)

In a visual pipeline representation:

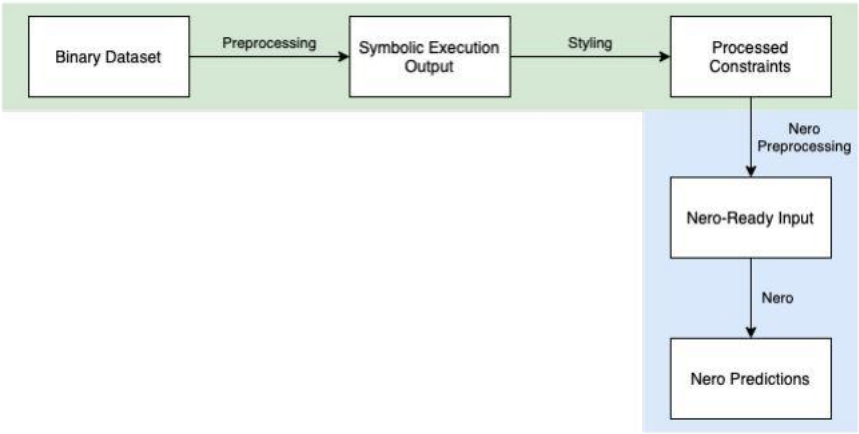

Figure 2: Binary2Name algorithm pipeline

My focus was to improve stages 1+2.

Main Goal

Because the last iteration of this project was the first to try and use Angr symbolic execution as an input for the Nero model, they were not fully able to explore and optimize this input for better results.

Last iteration reported model training results were as follow:

Accuracy: 0.0822 **Precision**: 0.1421 **Recall:** 0.0865 **F1:** 0.1075

While the original paper reported model training results as follow:

Precision: 0.4861 **Recall:** 0.4282 **F1:** 0.4553

Therefore, this iteration was designated to explore this input more deeply and come up with new styling methods to the function's constraints on top of new ways to build the “augmented CFG” (created in the preprocessing stage) by changing the methods in which we pick constraints for each node in the CFG.

The final goal of this project was to further exhaust the Angr symbolic execution as an input to the model so it will get better result on the Nero dataset, with the goal of improving on the results of the original paper.

Stages Of Development

Getting Started

For the start of the project, we decided on a few main goals for me:

1. Catching up on the original paper and the previous projects that were made building on it.
2. Focusing on the last iteration, learning the code and the pipeline.
3. Coming up with new ideas regarding styling methods to the function's constraints on top of new ways to build the “augmented CFG” by changing the methods in which we pick constraints for each node in the CFG.
4. Running the last iteration of the project, making sure it runs smoothly.

Running Into Technical Issues

Unfortunately, we ran into a lot of issues trying to run the last iteration of the project which delayed our schedule for a couple of months.

The main issues were:

1. Connecting and Using Lambda's GPU-s, which slowed down the training of the model drastically and made it hard for us to run multiple tests in a reasonable time frame.
2. An OOM error while training the model, which occurred in different unexplainable occasions, a problem which was not reported or tackled in the previous iteration.

As time progressed, together with Gabi, Yaniv and the Lambda staff, I found a way to partially work around these issues in the cost of slower training times, which directly impacted the number of tests for new methods I could have done during this time frame.

A couple of months before writing this report, a solution had been reached so the training process could run using the Lambda's GPU-s with no errors and in reasonable time frames which allowed me to do more experiments as previously described.

Getting First Results

After several bugs were fixed, both in the pipeline and in the connection with Lambda, I got my first results for both datasets (at this point, with no change made to the styling of the constraints or the method of picking them for each node in our augmented CFG made for each function):

Coreutils dataset: 65.5% (F1)

Nero dataset: 30.2% (F1)

New Constraints Styling Ideas

A few ideas were explored during this project concerning the styling of the constraints content:

1. **Getting rid of a token which was added to a function name to mark it as a function name** – the current token, which was given to each function name in the styling stage of the Angr symbolic execution output, was 'f'. As we saw no benefit in adding this token to mark a function name as a function name (as it was already established during the symbolic execution and distinguished from the other parts by the json format that was built to represent a function augmented CFG), but only potential harm, because we artificially made similarity in all of the function's names which could potentially confuse the model and harm it's results.
   Removing this token did not yield any significant results one way or another.
2. **Adding register number to the constraint's info** – At the last iteration it was decided to get rid of the registers identifiers that comes in the constraints from the Angr symbolic execution, out of concern that this data would be an unnecessary noise to the model which could potentially harm its performance. I decided to explore the option of adding this information thinking the diversity of possible data coming from that feature would not be enough to confuse the model but could possibly help it get better performance. After making this change the following result was achieved:
   Coreutils dataset: 67.3% (F1)
   Nero dataset: 33.7% (F1)

New Constraints Picking Ideas

For this iteration of the project, we mostly came up with ideas concerning the constraints picked for each node in the augmented CFG created for each function.

In the last iteration of this project each node in the augmented CFG graph accumulated all the constraints from nodes with a path to it plus added new constraints, so the basic idea was that some functions (and maybe some nodes in particular) maybe flooded with too many constraints which harm the model training and eventually its results.

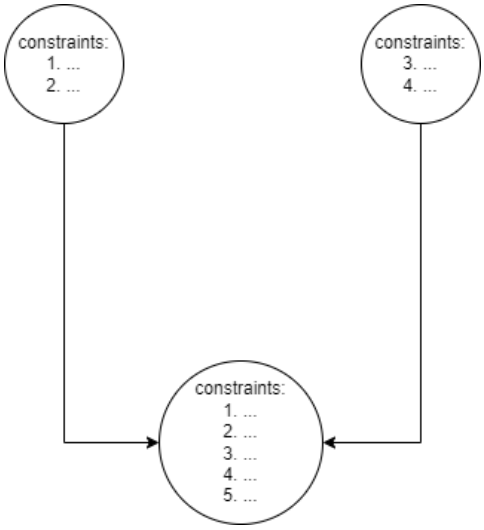

Figure 3: Last project iteration constraints accumulation method

Therefore, we came with a few methods to pick the constraints for each node in the augmented CFG:

1. **Each node accumulates constraints only from k nodes who has a path to it with the shortest path length in the augmented CFG graph**– After adding k as a parameter established in the initialization of the augmented CFG and adding the path length to each node as a parameter in each node, we decided to train the model with the k parameter set to values 1,2 and 3:

k=1: Coreutils datatset - 70.7% (F1)
Nero dataset - 45.6% (F1)
k=2: Coreutils dataset – 75% (F1)
Nero dataset – 45.2% (F1)
k=3: Coreutils dataset – 80.5% (F1)
Nero dataset – 44.4% (F1)

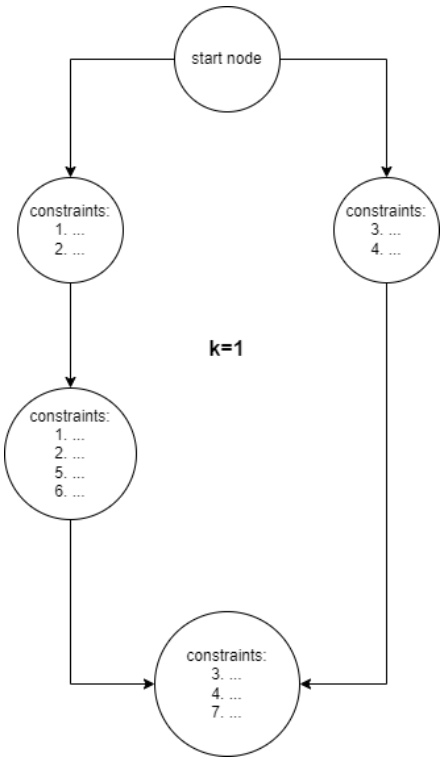

Figure 4: k shortest paths to each node method

2. **Each node accumulates constraints only from k nodes who has a path to it with the longest path length in the augmented CFG graph** – the same as method 1 but this time picking the k nodes with longest paths to them. Here I also trained the model with the k parameter set to values 1,2 and 3:

   k=1: Coreutils datatset - 74.3% (F1)
   Nero dataset - 44.9% (F1)
   k=2: Coreutils dataset – 71.5% (F1)
   Nero dataset – not conducted
   k=3: Coreutils dataset – 79.1% (F1)
   Nero dataset – not conducted

3. **Each node accumulates constraints only from k nodes who has a path to it with the least constraints accumulated in them in the augmented CFG graph** – After adding k as a parameter established in the initialization of the

augmented CFG I also added to each node a list of constrain lists from all the nodes with paths to it. The algorithm sorts those constraint lists by length and then pick the k shortest list from them to accumulate. Here I also trained the model with the k parameter set to values 1,2 and 3:

k=1: Coreutils datatset - 76.8% (F1)
Nero dataset - 46.2% (F1)
k=2: Coreutils dataset – 70.7% (F1)
Nero dataset – not conducted
k=3: Coreutils dataset – 78.6% (F1)
Nero dataset – not conducted

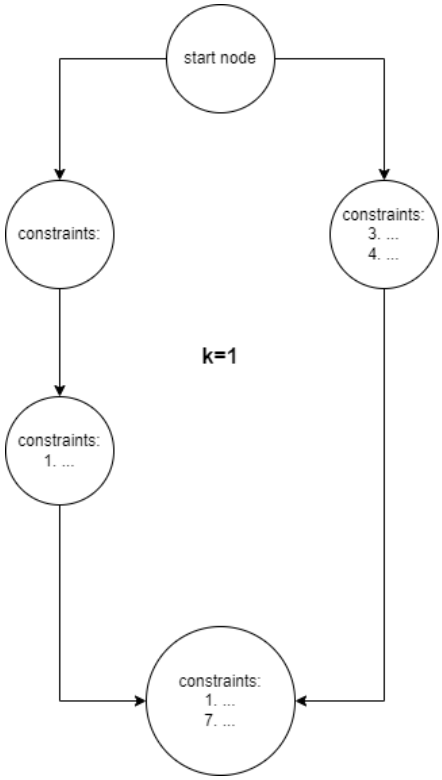

Figure 5: k nodes with least constraints to each node method

4. **Each node accumulates constraints only from k nodes who has a path to it with the most constraints accumulated in them in the augmented CFG graph** – the same as method 3 but this time picking the k nodes with most constraints. Here I also trained the model with the k parameter set to values 1,2 and 3:

k=1: Coreutils datatset - 74.1% (F1)
Nero dataset – not conducted
k=2: Coreutils dataset – 75.9% (F1)

Nero dataset – not conducted
k=3: Coreutils dataset – 73.2% (F1)
Nero dataset – not conducted

5. **Each node accumulates constraints from k random nodes with paths to it** – I added to each node an accumulating list for all the constraint lists from node with paths to the current node, then picked randomly k of those lists to accumulate to the current node. No experiments had been conducted with this method.
6. **Each node accumulates k constraints from the node with the shortest path length to it in the augmented CFG graph** - With the path length saved in each node (already developed in method 1) all that was left was to pick the node with the shortest path to it and sample k constraints randomly. No experiments had been conducted with this method.

It's important to emphasize that during those experiments we realized that a considerable part of the functions, sometimes even 65-75%, were not analyzed at all by our code from different reasons (bugs etc.).

Changing The Pipeline Process

After a few bug fixes and reorganization of the code by Gabi we had the Nero dataset symbolic execution output by Angr after it's been preprocessed and ready for the styling stage, **from that point on we worked with this output**. The process to get this output took 24 hours to perform and gave us 75,000 functions to perform the rest of the process on (**from that point on we decided to focus exclusively on the Nero dataset**).

Main changes that were made at this stage:

1. At the preprocessing stage of the Angr symbolic execution all the constraints were accumulated unless they were from a node "too far", which mean if the path between the current node and the node from which this constraint originally came from was bigger than x then it would not be accumulated to the current node. At this point x=3 was chosen.
2. Two input parameters were added to the styling stage, "sample_path" and "sample_constraint".
3. constraint picking method was implemented as part of the styling stage of the Binary2Name pipeline in a function called "__reduce_constraints". At this point we choose a fixed method which is, for each node "sample_path" paths

are chosen randomly from the paths available to it and from each of these paths "sample_constraint" are picked randomly to accumulate to it.
4. excessively long constraints or long paths were filtered in the preprocessing stage.
5. The final block (called loopSeerDum) in each CFG now contains the eax register value in a constraint format.
6. The styling process could now be done faster as it was now multithreaded.

Training the Nero model after these changes and after experimenting with the "sample_path" and "sample_constraint" values (decided on "sample_path"=1 and "sample_constraint"=3) gave us a new record result: 50.5% (F1).

Filtering Out Functions

After getting a stable pipeline and a preprocessed dataset (with almost all the functions from the Nero dataset) we could now focus on analyzing the performance of our model on different kinds of functions, constraints wise. This would mean to try and train the model on this these different kinds of functions and to test them.

We started thinking about meaningful ways to filter our functions, constraints wise, and came up with a couple of methods:

1. Filtering out functions with less than x total constraints in their augmented CFG.
2. Filtering out functions that more than y% of the nodes in their augmented CFG has no constraints.

We then conducted experiments using these filters separately to train and test the model on different kind of function groups.

Filter 2 experiments:

- Using y=25, we were left with 57,000 functions in the dataset and got a result of 59% (F1)
- Using y=15, we were left with 61,000 functions in the dataset and got a result of 53.5% (F1)

Filter 1 experiments:

- Using x=3, we were left with 47,336 functions in the dataset and got a result of 61.7% (F1)
- Using x=4, we were left with 41,859 functions in the dataset and got a result of 63.4% (F1)

- Using x=5, we were left with 37,275 functions in the dataset and got a result of 64.7% (F1)
- Using x=10, we were left with 21,093 functions in the dataset and got a result of 64.7% (F1)
- Using x=20, we were left with 8,249 functions in the dataset and got a result of 64% (F1)

In accordance with those experiments, we decided to continue the experiments with two trained models, one is a trained model with no filter on the functions in the dataset and the second is a trained model with a filter on the functions of at least 5 constraints (filter 1 with x=5).

I then used filter 1 to create test sets to examine both of this model when they encounter filtered functions with specific number of constraints.

For the model with no filter on the functions in the dataset the results were as follow:

- Test set with made with x=3: 62.1% (F1)
- Test set with made with x=4: 70% (F1)
- Test set including functions with only 1 constraint: 36.5% (F1)
- Test set including functions with only 2 constraints: 38% (F1)
- Test set including functions with only 3 constraints: 49.9% (F1)
- Test set including functions with only 4 constraints: 57.6% (F1)
- Test set including functions with only 5 constraints: 66% (F1)
- Test set including functions with only 6 constraints: 58.9% (F1)
- Test set including functions with only 7 constraints: 59.2% (F1)
- Test set including functions with only 8 constraints: 68.2% (F1)
- Test set including functions with only 9 constraints: 72.8% (F1)
- Test set including functions with only 10 constraints: 75.6% (F1)
- Test set including functions with only 11 constraints: 69.9% (F1)

For the model with a filter on the functions of at least 5 constraints the results were as follow:

- Test set with no filter on the dataset function: 42.6% (F1)
- Test set with made with x=3: 67.2% (F1)
- Test set with made with x=4: 69% (F1)
- Test set including functions with only 1 constraint: 0% (F1)
- Test set including functions with only 2 constraints: 3% (F1)
- Test set including functions with only 3 constraints: 4.6% (F1)
- Test set including functions with only 4 constraints: 9.3% (F1)
- Test set including functions with only 5 constraints: 72% (F1)

- Test set including functions with only 6 constraints: 68.6% (F1)
- Test set including functions with only 7 constraints: 70.1% (F1)
- Test set including functions with only 8 constraints: 75.3% (F1)
- Test set including functions with only 9 constraints: 75.2% (F1)
- Test set including functions with only 10 constraints: 81.3% (F1)
- Test set including functions with only 11 constraints: 77.8% (F1)

It's important to emphasize that by creating those test sets in a separate process than the one who split the dataset for the training of the model causes definite leaks between the sets and artificially improves the results, although the trend should be the same.

Using A Pre-Made Split to The Nero dataset

For My final experiment I used a pre-made split of the binaries in Nero dataset to train, test, and validation sets (made by Yaniv), therefore canceling the random sampling of functions from different binaries and splitting them between those sets (**which was the splitting method so far**). By doing that we also avoided leaks between the sets.

I then trained two models again, a model with a filter on the functions of at least 5 constraints and a model with a filter on the functions of at least 10 constraints, using this split.

The results were as follow:

- the model with a filter on the functions of at least 5 constraints: 40% (F1)
- the model with a filter on the functions of at least 10 constraints: 37.3% (F1)

Conclusions

After a lot of technical issues which delayed my progress in the beginning of this project, together with Gabi and Yaniv as my instructors, we got to a point where its safe to say that the Angr symbolic execution has true potential as an input to the Nero model, as some of the results had already surpassed the original model results and just needs a few more checks to be officially confirmed as legitimate results.

Main Achievements

1. **Establishing a steady version of the model and pipeline using the Angr symbolic execution as input** - After a slow and unsteady start to the project caused by technical issues, a steady and faster pipeline which produces a trained model showing good results, has been reached.
2. **Establishing new high results using the Angr symbolic execution as input** – Throughout this project a lot of experiments showed promising results, some of them higher than the results of the Nero model from the original paper.
3. **Establishing a quicker Binary2Name pipeline** – As it stands our pipeline now can create the preprocessed Angr symbolic execution output in 24 hours (if needed) and complete the styling of this output and training of the model in 4-6 hours. This is a huge improvement from the last iteration of this project which reported that "analyzing the Nero dataset took around 20 hours" but also "only around 55% of the Nero dataset were analyzed successfully", compared to 93% with the current pipeline.

Results Reasoning

- In general, we saw that picking fewer shorter constraints and paths to each node in the augmented CFG created for each function improved the results significantly. We can assume this happens because before that the model was flooded with too much information which eventually harmed the model training and results.
- It seems like the more constraints a function has the better chance our model would predict its name correctly, as can be seen by the results of our experiments filtering the functions in different ways. This seems to be logical as the model does its learning based on constraints input, so the more constraints a function has, the better the model learns it (until a certain limit in which the information is "too much" for the model and only confuses it).

Future Work

At this point we have a solid ground to continue different experiments while exploring more deeply some of the ideas we started exploring during this iteration of the project:

- **Constraints styling:** At this iteration we didn't conduct enough experiments exploring different kind of ways to style the constraints themselves. Different ways of styling the constraints, like adding information to them (final reg values, final memory values etc.) or rather derogate some unhelpful or even confusing information.
- **Re-explore constraint picking methods:** As we reached our current stable version of the Binary2Name pipeline rather late in this project, some of the constraints picking methods we came up with in an earlier stage were not fully tested as they should has. It may be smart to revisit those methods an experiment with them as they may lead to different (maybe better) results than before.
- **Keep exploring function filtering:** It's a possibility that with some function filtering method a model can reach a high performance when dealing with certain kind of function group (constraints filtered wise). This approach may lead to an understanding that models who are trained with some filtering on their dataset functions can reach very high results for some type of functions, which may also be a valuable tool for researchers of binary code.
- **Don't forget other output from Angr symbolic execution:** Angr symbolic execution produces much more information than the constraints, including the assembly code itself. As this part of the Angr symbolic execution had not been explored yet it seems like a good idea to try and experiment with it as it may hold some useful information for the model to learn from.

References

1. Anand, Saswat; Patrice Godefroid; Nikolai Tillmann (2008). "Demand-Driven Compositional Symbolic Execution". *Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science. **4963**. pp. 367–381. https://link.springer.com/chapter/10.1007/978-3-540-78800-3_28
2. Neural Reverse Engineering of Stripped Binaries (2019) By Eran Yahav, Uri Alon, & Yaniv David. https://arxiv.org/pdf/1902.09122.pdf.
3. Yahav, E., Alon, U., Levy, O., & Brody, S. (2019). CODE2SEQ: GENERATING SEQUENCES FROM STRUCTURED REPRESENTATIONS OF CODE. *ICLR* .
4. https://angr.io/
5. Neural Software Analysis (2021) By Michael Pradel & Satish Chandra. https://arxiv.org/pdf/2011.07986.pdf
6. Learning to Find Usages of Library Functions in Optimized Binaries (2021) By Toufique Ahmed, Premkumar Devanbu, and Anand Ashok Sawant. https://arxiv.org/pdf/2103.05221.pdf