Computer Science Department

Binary2Name – Automatic Detection for Binary Code Functionality

Advisors: Dr. Gabi Nakibly, Dr. Yaniv David

236349 – Project in Computer Security
Final Report

Spring 2021

Ittay Alfassi
315188896

Itamar Juwiler
322540568

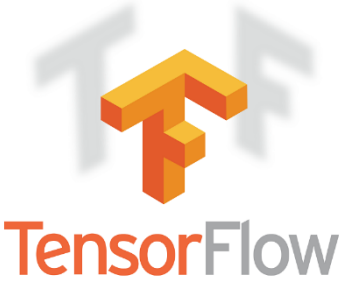

Table of Contents

Introduction

Overview

This project is the third 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, which Dr. Gabi Nakibly also advised.

Using the git repository that both pairs published, we built upon the foundation they had set up in hopes of adding improvements to achieve more accurate results. Like our predecessors, our work in this project aimed to automatically detect the functionality of a binary code snippet. Given the binary code of a function as input, our goal was to output a name for the function that accurately describes its functionality.

We 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.

We used a deep neural network adopted from the paper: Neural Reverse Engineering of Stripped Binaries [2], which is intended for the same goal, but used a different approach to the problem. After experimenting with small datasets from the previous iterations of the project, we decided to use the dataset provided in the same paper to test the results of our work.

Our 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/ittay-alfassi/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

Past Project

The last development iteration of this project was in the spring of 2019. We studied the project's results in depth, looking into their code flow and results. In the previous iteration, the students used Angr (binary analysis tool - explained later) to produce the symbolic execution paths for every function. Angr also outputs constraints on each symbolic variable in the path.

They started with these paths and their matching constraints and with the use of the code2seq model [3], which tackles the problem of automatic detection of source code functionality. The input to the code2seq model is a path in the Abstract Syntax Tree (AST) of the function's source code. They added their own preprocessing so that they can adapt them to match code2seq input. Since the input for code2seq had the following format:

```
functionName source, path connecting source and target in AST, target
```

They took on the following format for the input in their project, where the source and target were replaced with "DUM", empty fields, and the context itself was just in the middle path:

```
functionName DUM, path CONS constraint, DUM
```

We saw that there were many optimizations already done on the preprocessing part of the project, and that there is not much left to try. Therefore, we decided (with Gabi's advice) to put aside the current method and try to use a different deep learning model. However, we thought it will still be wise to keep the baseline functionality of the previous project, because we thought it had potential. We started thinking how we could optimize the preprocessing process to achieve the best results.

Angr

Angr 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.

Our goal is to train a deep learning model using the symbolic execution outputs of the binary code. To do this, our preprocessing used Angr on our library of binaries (the datasets) which gave us control-flow graphs of assembly code, augmented with symbolic constraints, which we then 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.

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". 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:

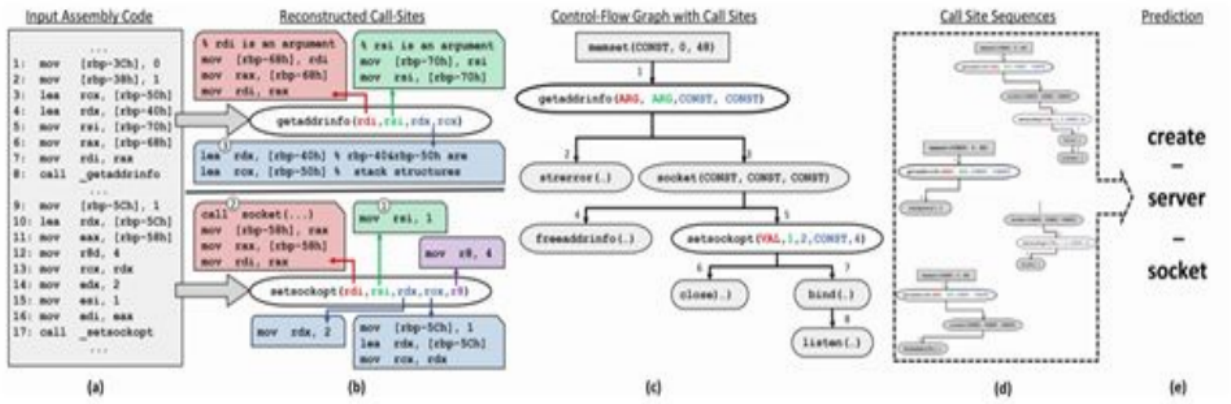

Figure 1: Nero's algorithm pipeline

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 ($a \rightarrow b$).
Secondly, the CFG is constructed with data containing the natural API's names ($b \rightarrow c$).
Lastly, the CFG is inserted as input to the prediction model (which calculates over all the possible paths) ($c \rightarrow d$).

Datasets

To train our model (the Nero prediction model) to predict the tokens that comprise a function's name, we need to train it using these tokens, so it can "learn it".

In the previous papers, a function in a dataset was considered usable (worth learning) if it appeared n times in the dataset (n changed throughout our different experiments).
We do this to ensure that each function's tokens will be learned enough times, and they will appear in the train, test, and validate phases of the process. This way, the model will be able to predict them successfully. However, in our work, this process is irrelevant – the Nero model already contains a method to combat this problem during the training process itself, so we don't need to worry about it during preprocessing.

As for splitting the datasets into train, test and validate sections, we used a division of 0.7, 0.2 and 0.1 accordingly. However, our scripts are configurable. Those percentages can be changed easily rewriting code.

As for the datasets themselves, the goal was to use our preprocessing pipeline to get better results than the paper achieved using the **same dataset and model**. However, we still needed to test our work using some other datasets as well because the Nero dataset was hard to manage and test on at times.

We used mainly two datasets: the Nero dataset, and the coreutils dataset from the previous iteration of the project. We tested and experimented mostly with the coreutils dataset, because it was smaller and easier to manage, but we also tested the Nero dataset.

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.

Stages of Development

Starting Point

Our work in this project was mainly focused on preprocessing input data for the neural network.
In our development, we had three main goals:

1. Utilize the preprocessing work done by Carol and Abdallah.
2. Generate data as close as possible to Nero's expected data.
3. Use the inherent graph-like structure of functions to generate more meaningful data.

Those demands were sometimes contradictory and required compromise – we had to replace sizable parts of Carol and Abdallah's work to move to a graph-like structure, and prune some of our data to keep the structure of the Nero data.
In the following sections, we will describe the full structure of the algorithm.

Algorithm Pipeline

The Binary2Name algorithm we developed is 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 (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:

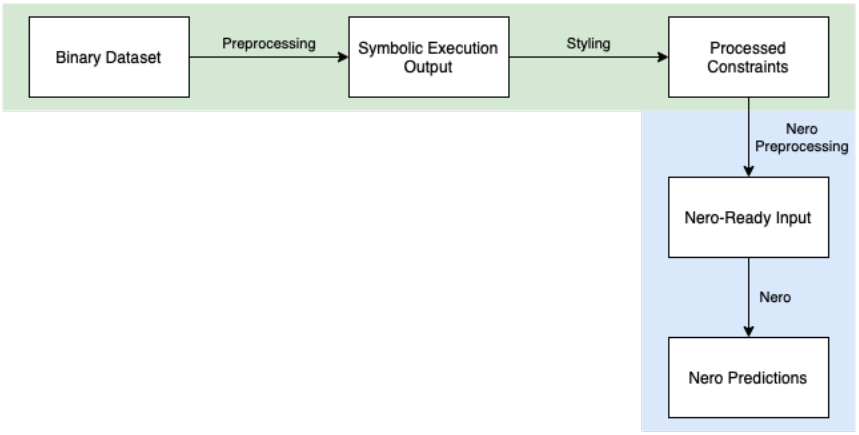

Figure 2: Binary2Name algorithm pipeline

Basic Symbolic Execution

In this part of the algorithm, we used the angr framework to extract the functions from each binary file and perform symbolic execution on each function.

Since symbolic execution can grow exponentially, certain bounds must be defined for the process. In Carol and Abdallah's code, the LoopSeer angr technique was used. This symbolic execution technique stops the analysis for looped basic blocks, after k visits to said blocks. We kept using the LoopSeer technique with k=2, as per Carol and Abdallah's recommendation.

The result of the symbolic analysis is a list of end states. Each state represents the execution's state, in a **specific** execution path. Each path contains a unique set of choices in the branches and loops of the function. The execution state contains a pointer to its parent state (the simulation state after the basic block which led to this state), the code of its basic block, and the state's constraints.

Each node contains the constraints which needs to be met to travel on the path that passes through that block. Formally speaking, each node contains a list of constraints lists (a double list of constraints). The inner lists represent constraints that are all derived from a specific path that passes through that node, and the outer list contains such a list for each execution path that passes through that node. Technically speaking, to combine all those constraints into a large one all one needs is to put a logical AND operation between every constraint in an inner list, and then put a logical OR operation between those to get one big constraint describing the symbolic execution of that block.

Paths vs. Graphs

In the project's previous iteration, the ML module of the algorithm used the code2seq model. Since the code2seq model takes AST paths as input, a straightforward approach was possible – define a path going from each end state to its ancestors and use it as the basic data unit for later styling and cleaning.

In contrast, the Nero model is based on Graph Neural Networks (GNN) – Neural Networks that take graphs as input. Thus, we had to change the extraction method to creating a "augmented CFG" – a CFG that contains data on the constraints required to move between nodes.
We achieved this by recovering the paths from each ended state and merged every two states with an identical block address into a single block.

Every path that ended because of a LoopSeer limit had a dummy node inserted at the end, so the graph would be consistent with the semantic meaning of each path.

Placing Constraints – Nodes vs. Edges

When merging the "state paths" mentioned in the last subsection, one challenge was apparent - the location and value of the constraints.

Firstly, we decided to only use the **recent constraints** of the block. This way, we preserve the graph semantics of the CFG, in which every edge is only dependent on its connected nodes.

Secondly, we decided to move the constraint on every edge to its destination node.
This approach was favorable for several reasons. It is more like Nero's GNN structure, in which the data is only saved in the nodes. In addition, it allowed for smoother "constraint merging".

Finally, when merging two nodes, we decided to keep the **union** of their constraints. Because of that, each block accumulates constraint options – all possible constraints on the data that can lead to its execution. While this decreases the completeness of the data (no indication which constraint belongs to which edge), we found that the benefits outweigh this drawback.

Each function ends this process as an "augmented CFG". Each node in the augmented CFG represents a basic block with a its instructions, and a list of constraints. Each function's CFG is saved in a JSON file, along with some metadata, which is the package and binary they belong to.

Resource Limitations

One of the major drawbacks in symbolic execution is its computational complexity.
These requirements are commonly caused by state explosion (1) and by complex SMT solving. They affect space requirements, execution time and recursion depth. When we tried to run our basic symbolic execution on the Nero dataset, we suffered from long delays and even crashes – the faculty's Lambda servers would crash after about 1/3 of the dataset. To mitigate that effect, we set resource limitations to each binary file.

In the current configuration, the analysis of each binary file is bound at 1000 minutes, is allowed to use up to 45 GiB of memory and has a call-stack depth limitation of 1,000,000. These parameters are easily changeable for future experiments.

These limitations allowed us to run the analysis smoothly but had a cost – about 80% of the coreutils dataset and only around 55% of the Nero dataset were analyzed successfully.

Smart Execution

Even with the resource limitation presented above, the analysis of each binary took a very long time. Our goal was to analyze the entire coreutils dataset in under 24 hours, so the Nero dataset analyzing would not take longer than a week.

In the previous iteration of the project, the analysis was performed in basic batch execution, a bash script activated the analysis script on 10 jobs each time and waited for all of them to end before dispatching the next batch.

Under this script, analyzing the coreutils dataset took 3-4 days, and the Nero dataset could not finish analyzing even after a week. To combat the inefficiencies in this execution method, we implemented a dynamic dispatching module.

The module takes resource limitations and the number of cores in the system and dispatches a job for each existing core, acting as a "job master". The master listens to each dispatched job and releases a new job whenever a job is finished.

This way, there is no "concealed unemployment" in the parallel execution. Using the dynamic dispatching, analyzing the coreutils dataset took around 7 hours, and analyzing the Nero dataset took around 20 hours.

Styling and Post-Processing of Constraints

While the previous stage of the pipeline extracted most of the valuable data from each function, its output is very "dirty" – it is filled with redundancies and symbolic execution clutter.

We decided to separate this stage from the symbolic analysis for performance reasons. Because the symbolic analysis can take up to a whole day for a sizable dataset, and the post-processing takes a few minutes for the same dataset, two separated stages allow for more experiments without wasting computing power.

The work in this subsection and the following subsection is implemented in the output_converter.py script.

Symbolic Execution Output Pathologies

To show the defects in the raw output, here are a few examples:

```
"block_addr": "loopSeerDum",
"constraints": [
    "<Bool __eq__(__add__(fake_ret_value_81282_64, 0x2), 0x1)>",
    "<Bool __eq__(__add__(fake_ret_value_81279_64, 0x2), 0x1)>"
],
"instructions": "no_instructions"
```

Figure 3: angr annotations (Bool, <> and __ wrappers), in du/add_exclude_fp

```
"block_addr": 4217888,
"constraints": [
    "<Bool __ne__(Extract(55, 48, mem_fffffffffe000008_81275_64), Extract(7, 0, reg_50_81271_64))>",
    "<Bool __ne__(Extract(63, 56, mem_fffffffffe000008_81275_64), Extract(7, 0, reg_50_81271_64))>",
    "<Bool __ne__(Extract(63, 56, __add__(0x1, mem_fffffffffe000008_81275_64)), Extract(7, 0,
reg_50_81271_64))>",
```

Figure 4: "Extract" header, giving no value to the constraint, in du/add_exclude_fp

```
"block_addr": 4219666,
"constraints": [
    "<Bool __eq__(mem_fffffffffc000000_29723_32, 0x2)>",
    "<Bool __eq__(mem_ffffff8000000000_29721_32, 0x2)>",
    "<Bool __eq__(mem_ffffff8000000000_29694_32, 0x2)>",
    "<Bool __eq__(mem_fffffffffe0000000_29692_32, 0x2)>",
    ...
```

Figure 5: same condition for 90 adjacent memory cells, in cp/copy_internal

```
"block_addr": "loopSeerDum",
"constraints": [
    "<Bool __ne__(Extract(31, 0, fake_ret_value_30386_64), 0x0)>",
    "<Bool __eq__(Extract(31, 0, fake_ret_value_30386_64), 0x0)>",
    "<Bool __ne__(Extract(31, 0, fake_ret_value_30385_64), 0x0)>",
    "<Bool __eq__(Extract(31, 0, fake_ret_value_30385_64), 0x0)>",
    ...
```

Figure 6: over 500 pairs of contradictory constraints, in cp/copy_internal

In the following subsections, we will show our methods for processing and handling each one of the issues in the examples.

Annotation Removal

Our first goal was to remove the “Bool” headers, the “<>” wrappers of constraints and the “__” wrappers of operations. These symbols add nothing to the semantic value of the constraints and might confuse the ML model.
We removed these symbols using simple regular expression replacements.

Constraint ASTs

While simple textual analysis was enough for annotation removal, we had to analyze the full syntactic properties of our constraints to gain valuable data.
We used the parentheses structure of each constraint to model constraints as ASTs.
The constraint ASTs are defined in the following manner:

- A leaf is a “concrete value”. A concrete value can either be a global memory address, a register, a function’s return value, or a constant (hexadecimal or decimal).
- An inner node contains a value and children’s pointers. A node’s value is its operation, and its children are its operands.

For example, the first constraint in Figure 3 will be translated to the following AST:

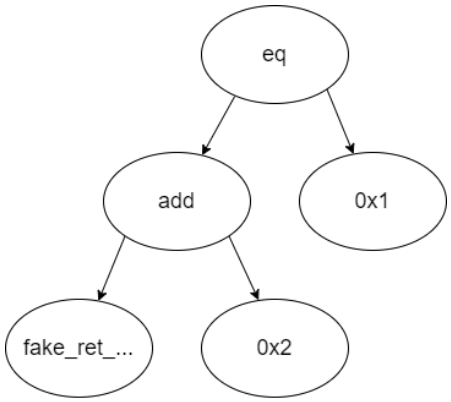

Figure 7: Example of a constraint AST

All following analyses and processing of the constraints will be on their AST form.

Irrelevant Constraint Nodes

In Figure 4, we can see an “Extract” operation on a memory chunk, which takes specific bits from the mentioned memory chunk.

These operations add unnecessary complexity to the constraints, and do not give any meaningful semantic value. There are other operations like this: the “invert” and “ZeroExt” operations.

To remove these operations from our data, we defined a "filler node removal" method. This method takes pairs of operation names and operand indexes. Each operation is then searched for in the tree. If it is found, it is replaced with the operand in the specified index.
For example, the "Extract" node is replaced with its third operand which contains the relevant memory/register.

Constraint Similarities and Contradictions

In Figure 5, we see an example of a very large number of constraints representing the same fact: some section in a range of memory locations (probably an array) is equal to 2. However, the symbolic analysis outputs about 90 almost similar constraints.
To clean the data and merge these constraints into a single, generalized fact, we defined a **constraint similarity** relation. Two constraint ASTs are considered similar if and only if:

- They have the exact same structure.
- If they are leaves, they are either one of:
  - Constants within a certain difference threshold.
  - Memory regions, whose addresses are within a certain difference threshold.
  - Return values, whose identifiers are within a certain difference threshold.
- If they are inner nodes, they have the same operation, and **all their children are similar ASTs.**

We also noticed that in Figure 6, we have many pairs of contradictory constraints: they are exactly alike, but their main logical operations are opposite. To model this, we also defined a **constraint contradiction** relation. Two constraint ASTs are considered contradictory if and only if:

- They have the exact same structure.
- They are not leaves, and they have opposite logical operations as values.
- All pairs of their respective children are **similar ASTs**.

Applying Deduplication

To put the similarity and contradiction relations we defined above to use, we implemented a simple deduplication algorithm.

Iterating over the list of constraints in a single node, if two constraints are similar, the second is removed and the first is replaced with a generalization. If two constraints are contradictory, they are both removed.

Adapting to Nero's Structure

Instructions vs. Constraints

After the application of the constraint AST processing, each node is left with two types of data: the instructions and the processed constraints.

However, Nero's model only accepts one type of data in each node. Thus, we faced a choice: Either remove the instructions and be left with "detached" constraints or remove the constraints and stay with nothing more than a CFG.

We decided to remove the instructions and stay with the constraints, so that our data will potentially be more valuable than Nero's original processed data.

Constraints to Function Calls

To use the output of the dataset preprocessing as input to the Nero model, we must reformat and convert the data. But first, let's explain in detail what needs to be converted into what.

As we mentioned, each function becomes an augmented CFG. The augmented CFG is, of course, a CFG, meaning a graph. Each node in that graph represents a basic block, which is a block of instructions, and the edges represent links between blocks that are executed sequentially in **a specific symbolic run** of the function. In addition, each node contains constraints.

The constraints of each node are represented as a double-nested list.
The full constraints to reach a specific node are represented by an AND operation between every constraint in the inner list, and an OR operation between each inner constraint list. To make the transfer from constraints to function calls as we will describe later, understanding this is important.

Now that we've described what the angr preprocessing output, lets discuss what the Nero model is expecting to receive.

The Nero model, very conveniently, is expecting a CFG for each function, which is what our symbolic analysis outputs. However, the data Nero expects to find inside each node is not constrains, but a list of function calls, for each API function called inside the basic block described.

So, how do we move between the two representations? There are a few possible methods, but we will only describe the one we used.

To transfer between the two representations, we need to discuss how to represent a constraint as a function call. Let's start with an easy example, below is the constraint AST for "$rdx < 3$".

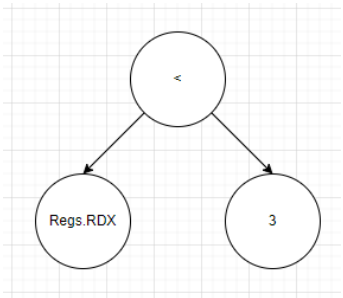

Figure 8: constraint AST for "rdx <3"

To describe this constraint as a function is easy: just start from the root and move down to the leaves. We can use that method to describe simple constraints, by referring to the inner node operation as a function and to the leaves as arguments. But what about a constraint AST with a height greater than 1? Consider the constraint AST of "$rdx + rdx < 3$":

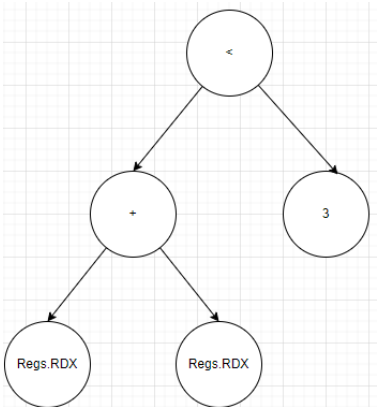

Figure 9: constraint AST for "rdx+rdx<3"

Well, this requires a smarter conversion mechanism. since we can't use assignments, the solution "$x = add(rdx, rdx); < (x, 3)$" isn't possible. We need a different solution. The solution we found is the following: "$add(rdx, rdx); < (f'add', 3)$".
Formally, we perform a post-order traversal of the constraint tree. Each inner node in the constraint AST is listed as a function activation. When going up in the AST traversal, function activations can be used as parameters, (using the f'') indicating that the argument is a result of these previous activations.

We are aware that this method isn't perfect – in certain cases, such as multiple applications of the same function, some data may be lost. However, that is the way we saw best to solve this problem. We will discuss other methods, and their pros and cons in comparison to our method, in the future work section.

Running Nero

To complete the algorithm, a single step remains – running the original Nero model. Although straightforward, this technical challenge proved rather difficult.

Environment Set-Up

The original Nero model was implemented using TensorFlow 1. Seeing as this version of TensorFlow is now deprecated (and no longer available through pip), we attempted to transfer the model to TensorFlow 2. Since TF2 is not backwards compatible, the transfer required changing the design of the model itself.

After several unsuccessful attempts to convert the model to TF2-compatible, we changed our approach to finding and installing a working version of GPU-supported TensorFlow 1.

After a lot of research and with the help of our advisors, we found a method that connects the required version of CUDA runtime to the one existing on the Lambda clusters via symbolic links, and we were able to run TensorFlow 1 with GPU support.

Defining New Object Classes

The last piece of the pipeline that we had to modify was Nero's internal preprocessing mechanism. In the original Nero code, only 4 classes of values were defined:

- Globals
- Constants
- Arguments
- Values

Since our data contained different types of elements, we defined our own (albeit similar) classes:

- Constant
- Registers
- memory cells
- return values
- Unknown values.

Experiments and Results

When we set out to do this project, we did so while considering we will not be the last iteration of it. Because of that, we put most of our focus on making a modular, configurable, and stable framework for the following iterations.

After the completion of the pipeline, we were able to perform a single experiment.
In the experiment, we used the following dataset and configuration:

Dataset:

- The full Nero dataset

Symbolic Execution:

- 1000-minute timeout per binary
- 10-minute timeout per function
- 45 GiB memory soft limit
- 50 GiB memory hard limit
- Call stack limit of 10^6 calls

Output Processing and Conversion:

- 3-minute timeout per json file (one-to-one with functions)
- All constants are considered similar
- Memory address difference threshold of 20
- Ret value index difference threshold of 20

Results:
The training ended with the following statistics:
Accuracy: 0.0822 **Precision**: 0.1421 **Recall**: 0.0865 **F1:** 0.1075

Conclusions

While the results of our experiment are still very far from being a useful tool for security researchers, it shows great promise – the concept of using symbolic execution constraints to infer semantic attributes of a function is prevalent in software synthesis, and potentially has a lot to gain from the power of modern deep learning.

Results Reasoning

In our git repository, the Nero output logs are recorded. In them, we can see that the initial loss value was low. This implies that the model was not able to learn much from the data.
This might be due to over-processing of the symbolic execution output. Another possible reason is the high number of binaries and functions that were killed in the symbolic execution phase, before their execution completed.

Future Work

In this section, we will describe work we believe could be done in the future by students to improve and enhance the results presented in this report.

In our project, we mainly focused on moving from the old model (cod2seq) into the new one (Nero). To truly benefit from the transition to the new model, we had to change the preprocessing process almost completely and create new styling and output converting processes. These processes were with the Nero structure in mind to reach the best possible result.
to notice here
We managed to get some results; However, we believe that with a little work those could be improved significantly and make the project truly whole – even surpassing the original Nero results.

Variable-Map Styling

An important part of our work is the styling of the angr outputs into data the Nero model could learn from easily and efficiently. As we discussed in the previous pages, the angr constraints often contain a lot of data that isn't relevant to the GNN, or could be considered relevant, but the amount of it is too big for the GNN to efficiently learn from it.

Let's provide an example:

```
"<Bool __gt__(__and__(ZeroExt(56, Extract(7, 0, __add__(0xffffffd0,"
"Concat(0x0, mem_ffffffffffff8002_229_8)))), 0xff), 0x1)>",
```

Or

```
"<Bool __eq__(__add__(reg_40_223_64, 0x2), __add__(0x1, reg_40_223_64, reg_20_218_64))>"
```

The first thing to notice here, is that the register and memory addresses come with three weird numbers, after some research we discovered:

- Register numbers stand for reg_<register number>_<counter>_<size>. From this data we can deduce what register was used, how much of it, and how many times it was used until now.

  In our work, we decided to simplify all the uses to "reg" because we thought more data than that would just be noise to the system, but in the future, we believe it would be wise to try other things with the data instead of simply ignoring it.

  For example, in previous works, the register number was used to replace "reg" with the register itself "rdi" for example, and the counters were used to count occurrences, meaning all the counters were cut down relative to the first counter in the file.

- Memory numbers stand for mem_<address>_<counter>_<size>, again we ignored this data and simply replaced it with "mem". In the future, we believe it would be wise to try different options.

Another possible way to address both regs and memory addresses is to create a variable map to replace all the uses of the same memory address / register with the same name.

The second thing is the removal of "Irrelevant Constraint Nodes". In our styling process we removed many such "irrelevant nodes". In future work, a more detailed look into the effects of each specific "junk node" could be beneficial. An additional path would be preserving *some* indication of the now-removed nodes.

The third thing we did was to remove the first word indicating the constraint type ("BOOL" in our example), there are other constraint types, and that data may be important, but we haven't found an elegant way to preserve it while transferring to function calls from constraints. This could also be expanded upon in the future.

Styling Combinations

in this work, we converted each constraint in the double list to a list of function calls, and then appended them. Another option was to use the "AND" and "OR" operators to create a "mega constraint" and turn it into a big list of function calls. The latter may lose less data, but it is also more costly – it creates huge constraint trees and function call lists which Nero model might just discard.

We didn't have time to check both options, so we used the first one, but the second one is worth giving a try to in the future.

Constraints to function calls

In the sections above, we described our method to convert constraints into function calls to match the Nero model's input pattern, from the initial tree-like structure of constraints.

As we hinted in that section, the method we used is one of many, and is just the one we saw fit to use. Even if after several experiments in the future, it appears to be the best way, there are still a lot of customizable changes that may make it better.

One thing is regarding the multiple constraints contained in each node, it's possible to:

a. Convert each of them to a series of function calls and enter them one by one sequentially.
b. Turn them into a "mega constraint" using the method we described in that section and convert that into a series of function calls.

In our research, we used option a. In the future, it might be probable to try using option b as well. Moreover, it might be good to try using different ways altogether to convert constraints into function calls.

Instructions

Another output from the angr preprocessing we almost didn't talk about is instructions. We already explained before that a node in the CFG contains information about the basic block it represents, and that includes instruction in that block.

During the conversion from constraints to function calls we completely ignored the instructions, because we didn't have ample time to research how and if to add them to the data, we are feeding the Nero algorithm.

Using that data in the future may enable higher rates of precision in future experiments.

Nero's Object Classes

As we mentioned before, Nero's algorithm used Object Classes to define different parameters inserted into the GNN. To match our results, we defined new object classes and changed Nero preprocessing script accordingly. However, the change was a bit forced and not befitting of the model. In the future, we believe that defining and refining new Object Classes, and redefining Nero's reference to them will greatly improve results.

Controlling Nero's configuration

Well, this is a no-brainer, but the Nero's model has a lot of configurable parts. Hyper parameters are important, but they are not all there is to configure.

During the processing of the function calls the Nero algorithm saves the function calls in a certain format, which is also configurable.

In this work, we mainly styled and converted the data to match the already existing Nero model and didn't change any configurations in that model. In the future, it is worth to check the direction of configuring Nero to the angr output, or even combining converting efforts in both sides, instead of focusing on the conversion of the angr output, as we have done.

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. doi:10.1007/978-3-540-78800-3_28. ISBN 978-3-540-78799-0.
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/