Wyatt Pochman
C S 474
April 22, 2024

Final Project Report - Binary Code Classification

Given a fixed input, compilers can generate many functionally identical binary outputs. For my project, I sought to implement a neural network that can take in an output and predict the input [albeit from a predetermined set of inputs]. This kind of task is common in reverse engineering and malware analysis. To begin, I selected 22 simple functions as candidates for classification. I then gathered implementations of these functions in C++ and compiled[1] them multiple times using `g++`. Each compilation cycle, I set different compilation flags[2] so that the resulting assembly code would differ. To assign classification labels, I mapped each of the candidate functions to an integer 0 through 21. The result was a dataset of 506 labeled assembly code snippets. An example of one of these is given in Figure 1.

```
0x10cf8:  eor     r3, r0, r0, lsr#16
0x10cfc:  eor     r3, r3, r3, lsr#8
0x10d00:  ldr     r2, [pc,#0x10]
0x10d04:  eor     r3, r3, r3, lsr#4
0x10d08:  and     r3, r3, #0xf
0x10d0c:  asr     r3, r2, r3
0x10d10:  and     r0, r3, #0x1
0x10d14:  bx      lr
```

Figure 1. Assembly code for the function `parity1a`, with label 20

My strategy, motivated by similar approaches from literature, was to do some significant preprocessing on the code snippets to transform them into a format a computer could "understand" better on a syntactical level. In particular, one can form a "control flow graph" (CFG) and a "register dependency graph" (RDG)[3] from assembly code. An example is given in Figure 2. [The pictured CFG is not very interesting because the code has no branches.]

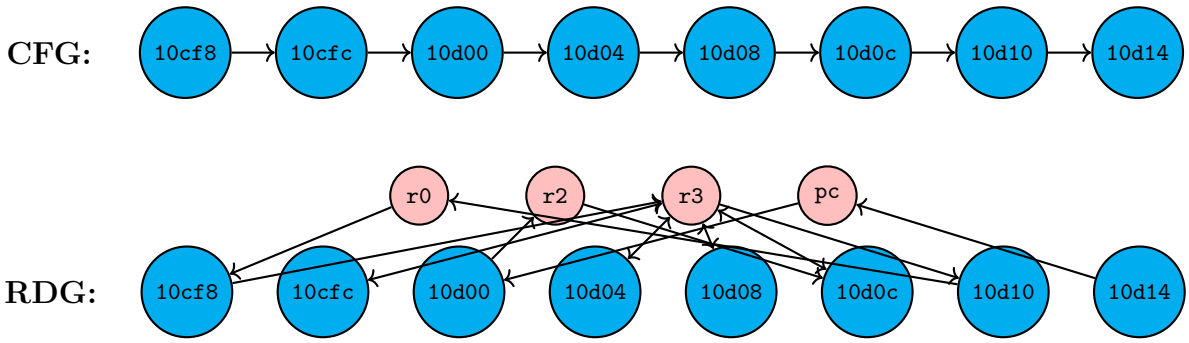

Figure 2. CFG and RDG for the code in Figure 1

[1]The functions are found here: `https://github.com/hcs0/Hackers-Delight`. I compiled for ARM64.

[2]`g++` offers 7 different optimization levels: `-O0`, `-O1`, `-O2`, `-O3`, `-Os`, `-Ofast`, `Og`, and `Oz`. While generating my dataset, I randomized which optimization flag was applied each cycle. The compiler also includes an option to specify hardware registers the resulting code is not allowed to read or write from. I randomized these to add further variation to the compiled code.

[3]Nodes of the CFG represent instructions, and edges describe the order in which instructions may be executed. Nodes of the RDG represent instructions and hardware registers; edges from instructions to registers represent register writes, and edges from registers to instructions represent register reads.

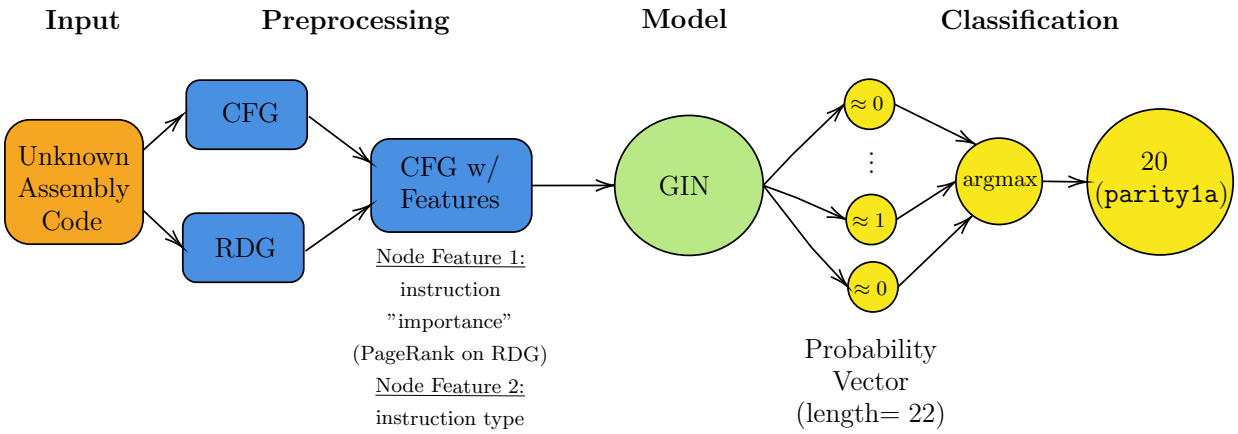

FIGURE 3. How the model classifies the code in Figure 1

The task of classifying graphs is well suited for graph neural networks. The best performing model was a graph isomorphism network (GIN). A GIN captures vectors representing various layers of abstraction of the input graph and concatenates them together, passing them through a final linear layer to produce a classification. Figure 3 gives a summary of the overall scheme. I found the best approach was to embed information from the RDG into the node features of the CFG, rather than passing both graphs directly into the GIN.

For training, I utilized the Adam optimizer with learning rate $\alpha = 0.001$. I generated my training and validation datasets by doing a 90-10 split on the 506 assembly code snippets described earlier. I additionally generated a test dataset of size 50 containing functions compiled from functionally similar—but not identical—code as the training dataset to ascertain which types of features the model was really learning. Note the differences in `rsqrt` below.

```
float rsqrt(float x0) {
   union {int ix; float x;};

   x = x0;
   float xhalf = 0.5f*x;
   ix = 0x5f37599e - (ix >> 1);
   x = x*(1.5f - xhalf*x*x);
   x = x*(1.5f - xhalf*x*x);
   return x;
}
```

(A) `rsqrt` C++ source code (test set)

```
float rsqrt(float x0) {
   union {int ix; float x;};

   x = x0;
   float xhalf = 0.5f*x;
   ix = 0x5f375a82 - (ix >> 1);
   x = x*(1.5f - xhalf*x*x);

   return x;
}
```

(B) `rsqrt` C++ source code (train set)

Results	Loss	Validation Accuracy	Test Accuracy
After 1000 batches	0.33264	0.7400	0.4800
After 2000 batches	0.15063	0.8400	0.6000
After 4000 batches	0.03361	0.8800	0.5800

The results are promising. Achieving 60% accuracy on the test dataset suggests the model is learning how the instructions work together rather than merely memorizing them. However, it often struggles to correctly classify algorithms that generate markedly different control flow graphs (e.g. due to added if/else statements) than those it was trained on. I would like to further develop my model by training it on a larger and more diverse dataset.