BINDNN: Resilient Function Matching Using Deep Learning

Nathaniel Lageman, Eric D. Kilmer, Robert J. Walls, and Patrick D. McDaniel

Department of Computer Science and Engineering,
Pennsylvania State University, University Park, PA, USA
{njl5114,ekilmer,rjwalls,mcdaniel}@cse.psu.edu

Summary. Determining if two functions taken from different compiled binaries originate from the same function in the source code has many applications to malware reverse engineering. Namely, this process allows an analyst to filter large swaths of code, removing functions that have been previously observed or those that originate in shared or trusted libraries. However, this task is challenging due to the myriad factors that influence the translation between source code and assembly instructions—the instruction stream created by a compiler is heavily influenced by a number of factors including optimizations, target platforms, and runtime constraints. In this paper, we seek to advance methods for reliably testing the equivalence of functions found in different executables. By leveraging advances in deep learning and natural language processing, we design and evaluate a novel algorithm, BINDNN, that is resilient to variations in compiler, compiler optimization level, and architecture. We show that BINDNN is effective both in isolation or in conjunction with existing approaches. In the case of the latter, we boost performance by 109% when combining BINDNN with BinDiff to compare functions across architectures. This result—an improvement of 32% for BINDNN and 185% for BinDiff—demonstrates the utility of employing multiple orthogonal approaches to function matching.

Key words: reverse engineering, malware, deep learning

1 Introduction

Understanding the behavior and structure of malware is critical to developing and improving our defenses against malicious code. However, the practitioners tasked with this analysis rarely have access to the malware's source code. As a result, the binary has to be disassembled and manually reverse-engineered in a time-consuming and expensive process. An important consideration, therefore, is deciding how to prioritize the analyst's limited resources. In other words, the analyst must determine which parts of the malware deserve their initial focus. Fortunately, malware authors commonly reuse code (e.g., libraries used for command and control) and thus a new piece of malware may significantly overlap with previously examined binaries. If the investigator can identify functions that have been analyzed before, they can leverage those existing results to increase the speed and accuracy of the reverse engineering. Such identification is key to reducing the cost of performing analysis.

The challenge in identifying these functions is that the same source code may have multiple equivalent byte code representations. The reason for this discrepancy is straightforward. When compiling the binary, high-level language features—such as control flow operations `while`, `for`, `if`, `case`, etc.—must be reduced and translated to the processor's instruction set. This translation depends on myriad factors including the choice of compiler, performance optimizations, and target architecture. While some researchers have proposed using control flow structure to match functions across binaries [3,6,9], such approaches often make the simplifying assumption that functions come from the same compiler, architecture, and optimization level. Consequently, these methods are insufficient for many practical scenarios.

The core of the challenge of identifying different compilations of the same code in different environments (target platforms, optimization levels) is an example of a *variant recognition problem*. More broadly, this is a common classification problem in which some base artifact is perturbed into a class sample. Later, a *classifier* uses an algorithm to identify the sample as belonging to the class. Note that machine learning has been extremely successful at building classifiers. For example, machine learning has been used to accurately detect malware and network intrusion [15,18,21,24], identify objects in images [4,7,19], and a host of other applications. In this paper, we build a machine learning classifier that identifies function variants created by compilation. We use large collections of sample functions executables to train a deep learning network. Later, the network is used to compare pairs of functions for their equivalence—that they were compiled from the same original source code.

In this paper, we introduce an orthogonal approach to function matching for malware analysis. Our algorithm, BINDNN, leverages recent advances in deep neural networks to build a model robust to changes in compiler or architecture. At its core, BINDNN uses a Long Short-Term Memory (LSTM) neural network to develop temporal relationships between assembly code instructions. These relationships enable BINDNN to approximate mappings from the assembly instructions back to the source code functions. Further, BINDNN incorporates a belief threshold that allows an analysis to dynamically adjust the sensitivity of the model. In short, we make the following contributions.

- We design a novel approach, BINDNN, for prioritizing functions during malware analysis. Based on deep learning, our approach matches function representations across different compilers, architectures, and optimizations.
- We evaluate BINDNN on a set of more than 70,000 binary function representations compiled from 2,598 unique functions. We find BINDNN classifies function matches with extremely high confidence, creating score distributions strongly weighted towards their respective classes, with median values of 0.99 for true matches and 0.0 for non-matches.
- Finally, we show that BINDNN compliments existing approaches. By using BINDNN in conjunction with BinDiff, we boost the performance by 109% when comparing functions across architectures and optimization level. This represents an average improvement of 32% for BINDNN and 185% for BinDiff.

We begin by more formally defining the function recognition problem and building a classifier for it using a deep learning network.

```
1  0804 A9A2  push    ebp
2  0804 A9A3  mov     ebp, esp
3  0804 A9A5  cmp     ss:[ebp+o], b1 0
4
5  0804 A9A9  jz      0x804A9B0
6  0804 A9AB  mov     eax, ss:[ebp+o]
7  0804 A9AE  jmp     0x804A9B5
8
9  0804 A8B0  mov
   eax, default_quoting_options
10 0804 A9B5  mov     eax, ds:[eax]
11 0804 A9B7  pop     ebp
12 0804 A9B8  retn
```

(a) Assembly of x86 with optimization O0.

```
1
2
3  0804 B850  mov     eax, ss:[esp+o]
4  0804 B854  test    eax, eax
5  0804 B856  jz      0x804B860
6  0804 B858  mov     eax, ds:[eax]
7  0804 B85A  retn
8
9  0804 B860  mov
   eax, default_quoting_options
10 0804 B865  mov     eax, ds:[eax]
11
12 0804 B867  retn
```

(b) Assembly of x86 with optimization O2.

Fig. 1

2 Problem Definition

The fundamental challenge of our work is identifying whether two distinct instruction sequences were compiled from the same source code. This proves to be a formidable task as even a single function may have multiple equivalent representations depending on the choice of compiler, the target architecture, and other factors. To illustrate, let us examine the impact of one such factor, optimization level.

Compiler optimizations are intended to make the code faster or more memory efficient. Consider the two instruction sequences in Figure 1. Both sequences were compiled from the same source code function (Figure 2) with the same compiler (`gcc`), but with different optimization levels. The assembly on the left was compiled with no optimizations (O0), whereas the assembly on the right was compiled with optimization O2. The primary difference lies in how O2 eliminates the need to set up the stack. First, lines 1 and 2 are removed. Next, at lines 3 and 4 the O0 code grabs the "o" pointer argument using the base stack pointer and compares it to 0 (to check for NULL). But in the O2 code we see it transfers the "o" pointer argument to register `eax` and perform the `test` operation to set the flags register. Then, at line 6, the O0 code moves the "o" pointer argument to the `eax` register, and the O2 code dereferences the "o" pointer argument and stores it back in the `eax` register. On line 7 we see the O0 code jumps to line 10 to perform the same dereference the O2 code already performed while the O2 code returns. If we took the jump on line 5, we see that O0 has to restore the stack base pointer on line 11 whereas the O2 code does not as it never set up the stack.

Even for our simple example function, optimization level had a significant impact on the compiled assembly. We can quantify this impact, in general, by using the edit distance between equivalent function representations. For example, on a large sample set of binaries compiled under various optimization levels, we calculated an average edit distance of `26.63` instructions.[1] Given that the average instruction length for a function was just over 50, these results mean that approximately 53% of each function changed based on choice of optimization level alone.

[1] We paired functions representations from `gcc -O0` against `gcc O1`, `O2`, and `O3`. See Section 4.1 for a description of the data set.

```
1 /* Get the value of O's quoting style.  If O is null, use the default.  */
2 enum quoting_style
3 get_quoting_style (struct quoting_options const *o)
4 {
5   return (o ? o : &default_quoting_options)->style;
6 }
```

Fig. 2: The source code for the assembly instructions seen in Figure 1 and 3.

```
1   00013110  STR     R11, ![SP,0xFFFFFFFC]
2   00013114  ADD     R11, SP, 0
3   00013118  SUB     SP, SP, 0xC
4   0001311C  STR     R0, [R11,0xFFFFFFF8]
5   00013120  LDR     R3, [R11,0xFFFFFFF8]
6   00013124  CMP     R3, 0
7   00013128  BEQ     b2 loc_13134
8   0001312C  LDR     R3, [R11,0xFFFFFFF8]
9   00013130  B       b2 loc_13138
10  00013134  LDR     R3, [off_1314C]
11  00013138  LDR     R3, [R3]
12  0001313C  MOV     R0, R3
13  00013140  SUB     SP, R11, 0
14  00013144  LDR     R11, [SP,4]
15  00013148  BX      LR
```

Fig. 3: Assembly of ARM with optimization O0.

As mentioned previously, optimization level is just one factor affecting the translation between source code and binary. Another factor, architecture, has an even greater impact on the resulting binary. For example, the ARM-based assembly in Figure 3, does not share any instructions with the equivalent `x86` from Figure 1. This, in combination with the factors discussed above, can make it extraordinarily difficult to match functional equivalences across program binaries by simple comparison.

We identify the main difficulty in this problem to be determining if two particular assembly instructions map back to the same source code function. Specifically, the goal of our method will be to devise a model that is able to classify a pair of instruction sequences as either a function match or non-match, regardless of the input factors that cause these sequences to change. From this classification, we can compare functions across binaries and look for previously examined functions in scenarios akin to the ones above.

2.1 Previous Methods

BinDiff [26] is the current state-of-the-art tool for comparing binary files to find similarities. BinDiff takes two input binaries, finds functions in the binaries, and then performs graph isomorphism detection on pairs of functions from the two binaries. This technique works well when two semantically equivalent binaries have similar control flow graphs. However, when they have different control flow graphs, such as when the binaries are compiled with different optimization levels, this approach loses its effectiveness [8].

Several others have proposed related techniques for detecting similarities. BinHunt [9] and BinSlayer [3] are two such examples. BinHunt uses graph isomorphism detection similar to BinDiff; however, BinHunt finds maximum subgraph isomorphism while BinDiff utilizes a greedy method for performance. BinHunt's algorithm works best when the graphs generated from the binary files are similar. Hence, they suggest using a different graph isomorphism technique when the differences are large. BinSlayer creates a polynomial time algorithm for calculating differences between two binaries by combining BinDiff's algorithm with the Hungarian algorithm for bi-partite graph matching [3]. We choose not to use either of these techniques in our analysis, as BinSlayer relies on BinDiff's structural comparison algorithm and shares many of the same weaknesses, and BinHunt loses effectiveness when analyzing binaries that produce largely different graphs.

`unstrip` [13] uses system calls in the form of semantic-descriptors to identify GNU C Library wrapper functions such as `read` and `write` in 32-bit binaries. The purpose of this tool is to mitigate the effort that analysts must spend in order to parse stripped binaries. The `unstrip` tool is used to label wrapper functions for *system calls* in Linux binaries. Their matching system uses a database of semantic descriptors and fingerprints to identify functions. While the identification of wrapper functions is important, our tool is more generalized and can detect both wrapper functions and functions that do not contain system calls.

When looking at methods for function identification within binaries, we see there has been some focus in using machine learning methods. Two of these methods are ByteWeight [1] and experiments with RNNs [23]. ByteWeight uses weighted prefix trees to classify the beginnings and ends of functions [1]. In [23], Shin et al. train a Recurrent Neural Network to classify the beginnings and ends of functions, and their method is able to outperform other methods [23]. However, it should be noted that finding function boundaries is related, but it is a different problem than function matching.

Finally, `BLEX`, created by Egele et al. is a tool for function matching that introduces a new method called blanket execution. This method executes functions in a controlled environment to analyze its behavior. This method performs better than most other methods for cross compiler (or cross optimization level) function identification, obtaining accuracy of 55%, and 64% when used as a search engine [8]. However, `BLEX` does not currently consider target architecture changes—something our method aims to consider.

3 Function Matching with Deep Learning

We propose BINDNN, a new approach to function matching inspired by recent deep learning approaches for Natural Language Processing (NLP). BINDNN is based on the following intuition: By representing assembly instructions as words, and their orderings as sentences, we can equate function matching to the problem of finding sentences with the same meaning. Framing function matching as an NLP problem allows us to leverage a wealth of past research as the starting point for our model.

In particular, BINDNN utilizes three types of neural network models: Convolutional neural networks (CNN) [16], Long Short-Term Memory recurrent neural networks (LSTM) [10,12,25], and regular fully connected feed-forward neural networks (DNN).

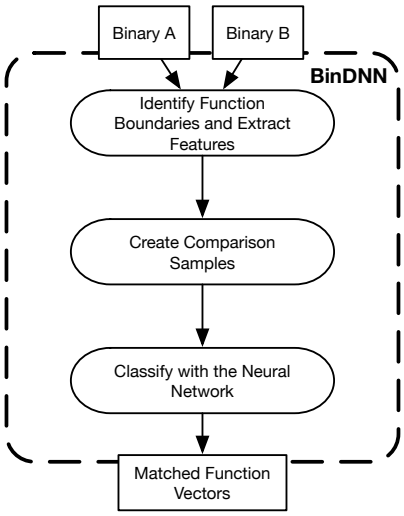

Fig. 4: Method overview of BINDNN. BINDNN uses a three step process. In the first step it has to find the function boundaries, so it can extract the features for each function. Next, it uses these feature function representation to create comparison samples before finally classifying these sample with a deep neural network.

We layer these models to construct an architecture similar to the design proposed by Sainath et al. for speech recognition [22]. This design takes advantage of the LSTM, allowing BINDNN to infer the temporal relationships necessary for function matching. In addition, our approach employs an embedding layer to make the model more effective at representing different inputs which have similar meaning [5,17].

Approach Overview. BINDNN uses a three step process to find function matches, as depicted in Figure 4. Upon receiving binaries to analyze, BINDNN first needs to find and represent the functions in assembly code. This involves leveraging preexisting techniques for function boundary detection, and then performing feature extraction to translate the assembly code functions representations into a more appropriate format for the neural network. Next, we generate samples that the neural network uses to learn the structure of a function match. Finally, we can use the neural network to determine if the assembly code functions originate from the same source code.

3.1 Binary to Feature Vector Translation

Identifying function boundaries. The first step of BINDNN begins the process that translates the binaries into a classifiable object that the neural network will be able to understand. This first requires finding the function boundaries in the input binaries. On an unstripped binary (one compiled with the debug flag on) this process is trivial, however, as our method is designed to work on unstripped binaries, we must have a method that

can still find these function boundaries. This is a difficult task for which has seen recent research [20,23]. However, IDA Pro [11] is still one of the best performing tools, and is commercially available. We choose to use IDA for finding function boundaries.

After the system has function boundaries, it can start to convert the functions from assembly to instruction sequences. In order for the comparison component of BINDNN to be able to perform the comparison, we need a way to represent the functions so our neural network is able to deduce discriminatory pieces from the functions that exist cross compiler, architecture, and optimization level. We designed our system to use an ordered set of the assembly instructions that make up the function with their arguments removed. This highlights the temporal relationship between the instructions, which the neural network is able to employ to aid in classifying the functions.

Feature extraction. Our approach is similar to what is seen in the taxonomy of deep learning for natural language processing [5]. BINDNN uses a global vocabulary of instructions that could appear in a function.

```
[aaa, aad, aam, aas, adc, ..., XTN,  XTN2, YIELD, ZIP1, ZIP2]
[0,   1,   2,   3,   4,   ..., 1940, 1941, 1942,  1943, 1944]
```

Specifically, it holds the instructions available for the architectures for which this model will be (or has been) trained to handle. In our experiments, we trained BINDNN to support binaries from two architectures, `x_86` and `arm`. Therefore, its global vocabulary contains the instruction sets for both architectures. Notably for these two architectures, the model saw no shared assembly instructions, as `arm` instructions were represented as uppercase and `x86` were represented in lowercase, e.g. BINDNN considers `MOV` and `mov` as separate instructions. Additionally, given architectures which share assembly instructions syntactically, it would likely be beneficial to change them to be unique when storing them in the vocabulary; in order to prevent confusion in the neural network.

3.2 Sample Creation

In BINDNN's second step, it constructs the sample that is passed to the neural network. A sample is the concatenation of two function representations, as seen below.

```
[push, mov, ..., retn] + [BARRIER] + [push, mov, ..., retn]
```

A sample can represent a match (where both function representations belong to the same source code function) and non-matches (the representations do not belong to the same function). There are two types of samples we need to create, training samples and testing samples. For both types of samples, it is must pair instructions sequences with each other and insert the barrier index. However in training, it must also contain the size discrepancy between the true match and false match samples sets, so the network can train on a balanced dataset. During classification, there is no such constraint.

A single sample that is provided to the neural network is the representations of two functions concatenated on either side of a barrier index. The barrier index is also stored in the global vocabulary like an assembly instruction. Each sample represents a pair of function representations that are either a matching pair (they represent two instruction sequences originating from the same source code function) or non-matching

(they originate for two separate source code functions). We add the barrier index to provide an indicator that the neural network can use to distinguish where one function representations ends and the other begins. The index becomes the length of our global instruction dictionary. Our current global vocabulary consists of 1945 total instructions, i.e. our barrier index is 1945. Additionally, we do not create samples including functions that are less than 5 instructions, or greater than 150 instructions.

Constructing the Training Dataset Consider the construction of the samples used in the training the phase of our tool, we only cover the generation of the training set because it is analogous to the generation of the test step. The only difference is that the test set generation uses a smaller set of functions. Let X be the set of functions in the training data, and k be a function in X. Then X_k is the list of representations for function k. To construct a "true match" sample for the training set, we find the indices of two representations, (i, j), such that $i \neq j$, from X_k and pair them together. This provides us with $\binom{|X_i|}{2}$ possible true matches for function i. To construct the "false match" samples we need another variable, $\bar{X}_k$, defined as follows.

$$\bar{X}_k := \{x \in X_m | \forall m \in X \text{ s.t. } m \neq k\} \tag{1}$$

Then the false match sample can be constructed as the pairs between each $i \in X_k$ and $\forall j \in \bar{X}_k$. However, using this method directly creates an unmanageable total of false samples causing the model to take far too long in the learning phase, and creating an unbalance in the number of true and false comparison samples. Specifically, the total number of true samples created is,

$$\text{Number of true samples } = \binom{|X_k|}{2} \tag{2}$$

and the number of false samples is,

$$\text{Number of false samples } = \sum_{i=0}^{|X_k|} |\bar{X}_k| \tag{3}$$

for function k. With over 10,000 unique functions, this quickly balloons the training set to an unmanageable size. To address this problem we used a cap, α, when constructing the true matches, so that we create $\min(|X_k|, \alpha)$ samples for function k. Then we also create an approximately equal number of false match samples.

3.3 Using the Neural Network

The third and final phase of our method is using the deep neural network to classify the samples. It is an 8 layer network, we describe the model's architecture in detail in Appendix A. The network takes a comparison structure as input, and returns the confidence score indicating the likelihood that it is made of two matching function representations. That is, they represent instruction sequences compiled from the same source code function. BINDNN tests all of the comparison structures created in the

previous step for each function. It then returns a list of all functions that could be matches, based on the threshold value, along with there associated confidence scores.

Before we can use our tool, we have to train it on the large sample set we constructed. With a network this size, this can take a substantial amount of time. The model does not have to be retrained for the ability to classify new functions that it has not previously seen. However, it does have to be retrained when expanding the number of architectures it can classify across. In preparation for our experiments, we train the network using the dataset constructed from Section 3.2. We train the network using 10 epochs, i.e. 10 pass through the entire dataset. Our loss function uses binary cross-entropy following the implementation in theano.[2] Following their notation the loss function is calculated elementwise as,

$$\text{Loss} = -(t * log(o) + (1-t) * log(1-0)) \tag{4}$$

Where t is the target value (the actual value), and o is the output value (the predicted value).The optimization function is what the network is trying to minimize during training. There are many optimization methods to approximate the gradient descent, as purely calculating it is not efficient enough. We use an optimization method called, "Adam" [14], which utilizes an adaptive learning rate allowing it to naturally perform a form of step size annealing. After training the network, BINDNN is ready for use.

When classifying, the network receives a set of samples for a particular function in a binary. Specifically, the set will hold the set of samples for that function versus every other function in the other binary. The network generates a confidence score for each of the samples indicating its belief that the two function representations are instruction sequences from the same source code function. BINDNN then compares these confidence scores to the threshold value that it was given, and returns a list of the comparisons that scored higher than the threshold. These represent the instruction sequences that the network believes to be from the same source code function.

4 Evaluation and Discussion

Our evaluation focuses on determining how the system would perform in the real world. We analyze the system's ability to detect instruction sequences originating from the same source code functions. This evaluation allows us to understand how well the system can improve an analyst's efficiency when analyzing malware. We compare our system, BINDNN, to a state-of-the-art tool, BinDiff. We test both system's abilities to detect function matches across real binaries compiled with different settings.

4.1 Data Set

We choose our dataset to represent real world programs. So, we used real programs that are often used on UNIX systems. Additionally, we want our results to be easily compared to other previous works. Specifically, ByteWeight [1] and their dataset of compiled

[2] `http://deeplearning.net/software/theano/library/tensor/nnet/nnet.html#tensor.nnet.binary_crossentropy`

programs. This dataset consists of the popular `binutils`, `findutils`, and `coreutils` toolsets. Each toolset was compiled for both x86 and x86-64 with `gcc` (version 4.7.2) and `icc` (version 14.0.1) using optimization levels ranging from `-O0` (none) to `-O3` for `gcc`. This dataset presented us with 2,064 binaries to include in our dataset.

However, this set of binaries only provides us with variations in compilers and compiler optimizations levels. We also need to expand the dataset to include binaries from multiple architectures, so we compiled multiple versions and implementations of `libc` for Linux. In particular, we tested Embedded GLIBC, `eglibc`, (version 2.19) and `glibc` (versions 2.22, 2.21, 2.20). We used the `eglibc` implementation because it is the default implementation installed on Ubuntu 14.04 LTS. Additionally, we also used 3 recent versions of GLIBC, which is included on Fedora, OpenSuSe, CentOS, and later versions of Ubuntu. We trained BINDNN on this dataset in order to identify C library functions in programs that were compiled statically and stripped. Each C library implementation was compiled with default CFLAGS and optimizations. Ideally, we would have liked to extract C library implementations that came by default in popular Linux distributions, however those libraries are already stripped of debugging information and are unusable for training purposes.

We compiled the binaries with the debug flag so that the function names would be included in the assembly code. This allowed us to establish ground truth for our training experiments. However, during testing, both systems were only provided with stripped binaries to make their decision on function matches. Specifically, the systems were attempting to match functions from one stripped binary to a second stripped binary[3]. Our analysis did not require matching functions from the dynamically linked libraries. Specifically, we analyze the ability of our method in matching functions that were directly contained in the source code. Our final dataset contained multiple instruction sequences for 12,993 unique functions.

4.2 Classifying Function Comparisons

In measuring the performance of BINDNN, we focus on two questions: 1) How many function pairs were correctly identified (true positives) out of the total number of identifiable pairs? 2) How many function pairs did the system identify that do not originate from the same source code function (false positives)? Since there will be such a large number of function comparisons, even in relatively small binaries (e.g. 1,000,000 comparisons for 2 binaries with 1,000 functions each), it is ever more important to correctly classify as many as possible. For instance, in our neural network testing phase, we generated the results seen in Table 1. This table represents the raw number of function comparisons the neural network of BINDNN was able to classify from a set of functions not previously disclosed to the system. We see that although it was able to correctly classify over 93% of the 2,166,126 function comparisons, it still ends up misclassifying 146,976 comparisons.

[3] This complicates the process of deciding when BinDiff has correctly or incorrectly identified a function. Our process for making this decision required that we first provide BinDiff with unstripped binaries, where is would successfully match all functions via name hashing, then save the effective address of the two functions it matched. Using these effective addresses, we were then able to verify matches made by BinDiff on the stripped binaries.

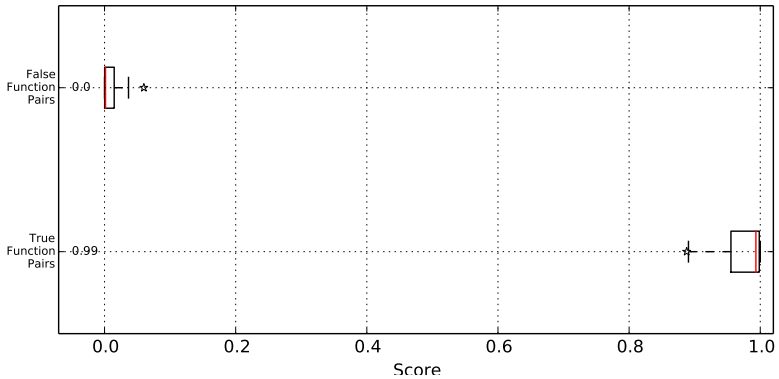

Fig. 5: Score distribution by class. The score distribution is very highly weighted for its respective class. This indicates that when the network correctly classifies a function comparison, it does so with extremely high belief.

This number of misclassifications severely reduces the ability of the system to aid an analyst in reverse engineering. For instance, if we incorrectly say that two functions are the same, the analyst may not see important information. If we incorrectly say that the same two functions are different, then we waste their time.

To better understand how we can improve these misclassifications, we analyze the distribution of the confidence scores assigned to the comparisons by the neural network. In Figure 5, we see the distribution of confidence scores the network gives when provided with a true pair of matching instruction sequences and a false pair. We see that confidence scores given to true pairs are extraordinarily high, with a median of 0.99. Likewise, the score for the false pairs are very low, with a median of 0.0. This indicates that if the network is given two instruction sequences for the same source code function, and if it successfully identifies them as a match, then it will do so with extremely high belief. This indication leads us to believe that varying the threshold for detecting functions is an important part of our system.

	Predicted NO	Predicted YES
Actual NO	1097555	52393
Actual YES	94583	921595

Table 1: Confusion matrix for the LSTM.

Configuring the confidence threshold. The confidence threshold (introduced in Section 3.3) allows an analyst to adjust the sensitivity of BINDNN when detecting function matches. This is useful in setting the number of acceptable false matches that may occur when comparing the functions from two different binaries. If the threshold is set too high, we may miss a large number of detectable function matches, and if it is set too

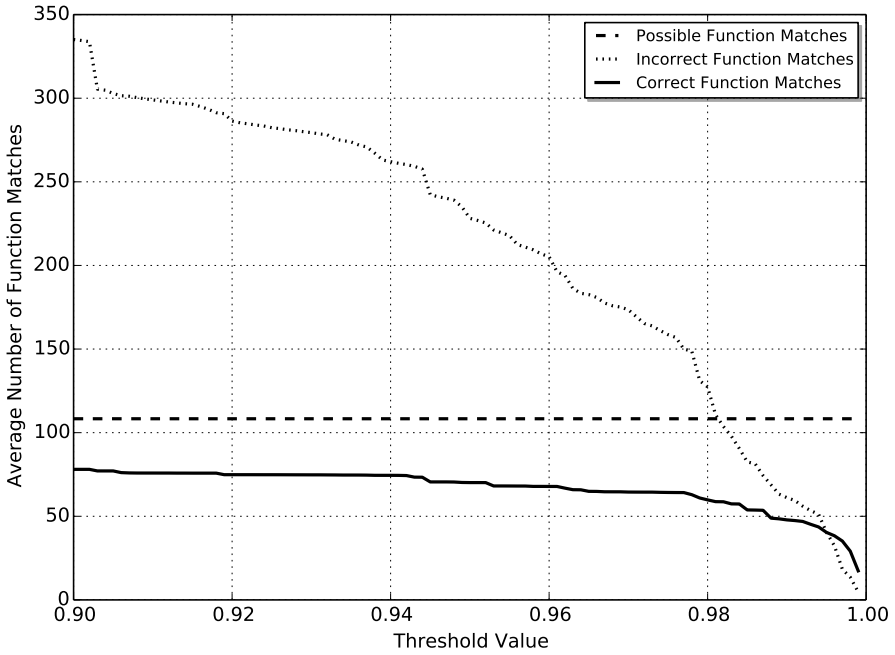

Fig. 6: Threshold analysis for compiler optimization level variations. Although a threshold value of 0.94 gives us (on average) the best coverage of the matchable functions, it still causes the system to incorrectly identify function matches at over 2 times the rate of correct matches. There are some situations where this might be acceptable, however, in our evaluation we chose a threshold that would provide similar results to BinDiff.

low, there may be too many false matches for the results to be useful. We found that, for our system, the optimal threshold value changes according to the compiler, compiler optimization level, and architecture. Hence, the threshold value should remain as a tunable parameter in BINDNN. To determine the optimal threshold values for our tests, we looked at the relationship between this threshold and the average number of functions identified for both true matches and false matches.

We studied this relationship for 3 configurations of the program binaries, as seen in Figures 6, 7, and 8. We see that with an increase in threshold, the number of false matches decreases at a much higher rate than the true matches. We saw indication of this in Figure 5, as the median and mean of the network's scores for true matches was very close to 1. As such, it appears that when the network correctly classifies a pair of instruction sequences as from the same source code function, it does so with very high belief. This allows us to increase the threshold to even out the true matches and false matches. Increasing the threshold does reduce the total number of true matches found by BINDNN, but it greatly increases the confidence for the matches it does find. In our tests, we choose to use threshold values that will allow approximately one false match for every true match, as this provides the most comparable results to BinDiff.

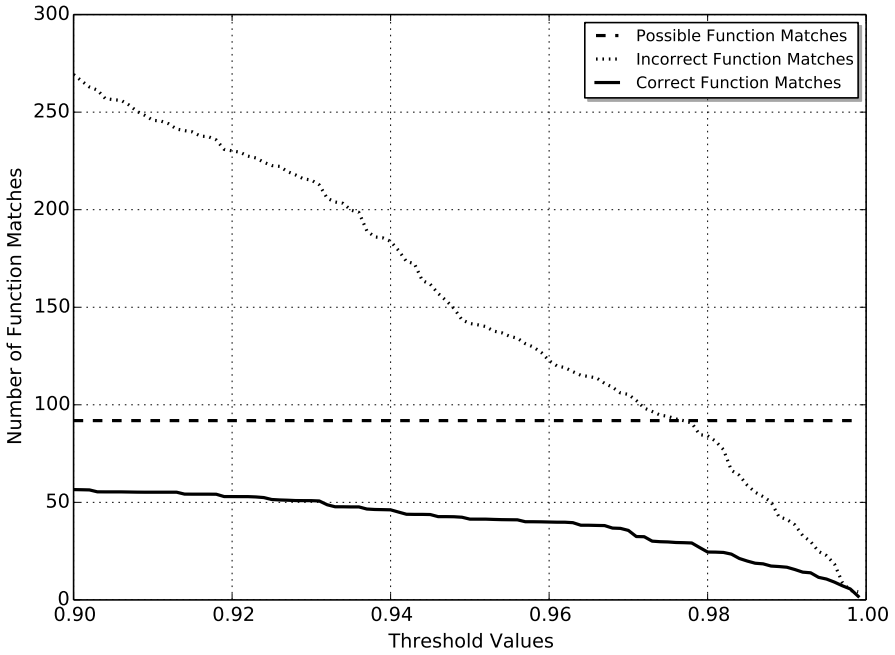

Fig. 7: Threshold analysis for architecture changes. We see the best correct function matches coverage without unnecessarily increasing incorrect matches at a threshold value of 0.92. This time we notice that also at this value the number of incorrect matches starts to decrease at a significantly higher rate than the true matches. Hence, we find we are able to hold a very low false positive rate at a high threshold.

Test environment. The machine we used to train, test, and ran the tool on used Ubuntu 14.04. It has an Intel Xeon E5-2630 clocked at 2.30GHz and 32GB of memory. We installed an EVGA GeForce GTX TITAN X graphic card to be used by the network model into the computer. It has 12GB of memory clocked at 7010MHz and has 3072 CUDA Cores clocked at 1127 MHz.

On this machine, when training the network we saw it average approximately 65000 seconds (18 hours) per epoch. This means 10 epochs took a little over 1 week. When actually using the network after it has been trained in the tool, we only have to consider how long on average it takes to process 1 sample and how many control function representations are in our control set. On average we saw each sample take less than a second, and about 10 seconds for 2500 samples. This means if we were analyzing two binaries with a 1,000 functions each, we would expect BINDNN to complete its process in approximately an hour.

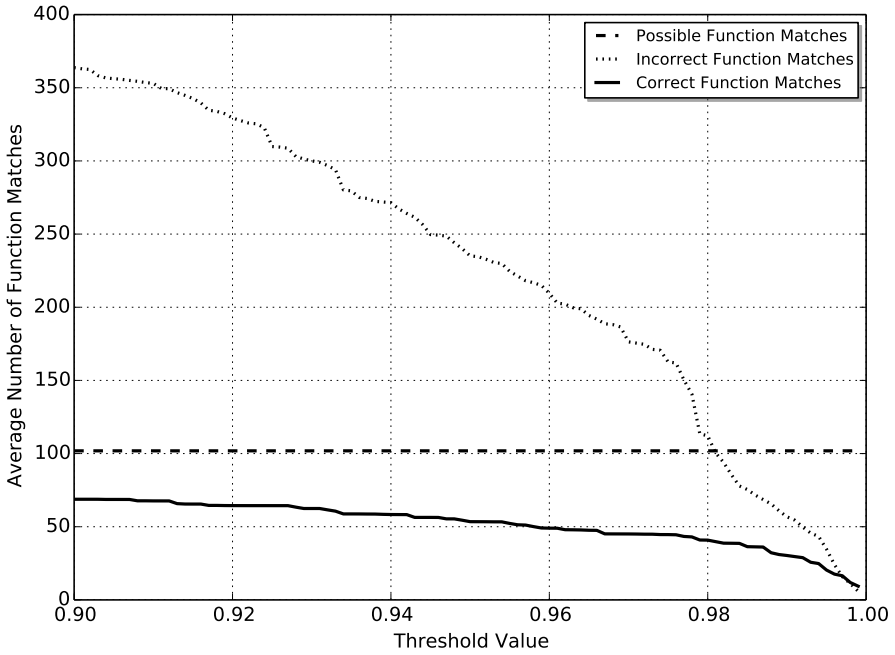

Fig. 8: Threshold analysis for compiler and compiler optimization level changes. This time again we see optimal coverage around 0.92, however, the average number of incorrect matches is still rather high. Although, this time there is a substantial increase in the rate at which the incorrect matches start to fall off around 0.975.

4.3 Resilience to Optimization Differences

We perform a case study to test the ability of both systems when matching functions from binaries compiled with the same compiler, but different compiler optimization levels. Specifically, we use binaries that were compiled with `gcc` on `x86` using optimization levels `O2` and `O3`. In Figure 9, we see the number of correct and incorrect function matches for both BINDNN and BinDiff for the shortest (by function count) 80 binaries in our test set. We chose the 80 shortest binaries for the sake of presentations; generally the results were comparable across the entire data set. The two systems have similar results, however, BinDiff generally outperforms BINDNN in this test. We also see that both methods produce low false positive rates, and the union of their correct matches provides a significant increase in correct matches. Specifically, we see an average increase of 45.7% for BinDiff and 66.3% for BINDNN. This indicates that using the two methods together creates an even more effective solution.

We can infer from the results, that although there is change in the assembly code structure, it still has parts similar enough for BinDiff to successfully match the functions. Although BINDNN could detect as many or more function matches as BinDiff from these two binaries, it cannot do this without increasing the number of false matches by

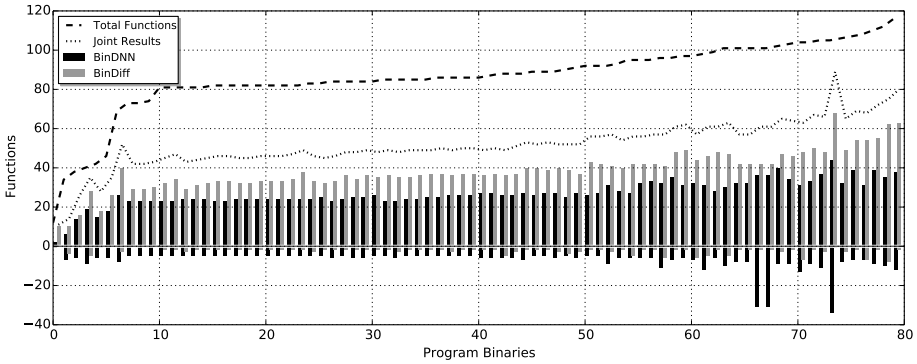

Fig. 9: Optimization level resilience comparison with BinDiff and BINDNN. The two systems have comparable performance (with BinDiff generally performing better) when matching functions from binaries compiled with the same compiler, but different optimization levels, i.e. `gcc -O2` and `gcc -O3`. The positive values are correct functions matches; whereas, the negative values are incorrect function matches.

a substantial amount. For instance, recall the relationship between the threshold and true and false positive rates, in Figure 6. We see that a threshold value of 0.94 will match, on average, approximately 70% of all possible function matches. However, it will also increase the number of false positives by a substantial amount. Even so, there are situations in which this would be acceptable practice. For instance, if an analyst was looking for shared functions between two malware applications compiled with different optimization levels, a number of false matches would still be acceptable, as that will still be better for the analyst than manually comparing each function to each other. In our experiment, we chose a threshold value that provides the results most comparable to BinDiff. In this case, that was 0.993.

4.4 Resilience to Architecture and Optimization Differences

We create another configuration of program binaries to determine the effectiveness at which BINDNN and BinDiff are able to match functions from binaries compiled for different architectures. The binaries we used in this experiment were compiled with `gcc -O0` for `x86` and with `gcc -O3` for `arm`. When determining the threshold BINDNN should use, we consult the relationship between the threshold and the classification rates, as seen in Figure 7. Notably, we could choose a value close to 0.92 to obtain the most coverage of the entire binary without unnecessarily increasing the false positives. However, in order to generate comparable results with BinDiff and across experiments, we again choose a high threshold value. This time we use 0.991.

In Figure 10, we see a comparison of the number of correct and incorrect function matches for BINDNN and BinDiff on the programs compiled across architectures. Again, we only show the results for the 80 shortest (by function count) binaries for presentation reasons, and the results for the longer programs are comparable. This time

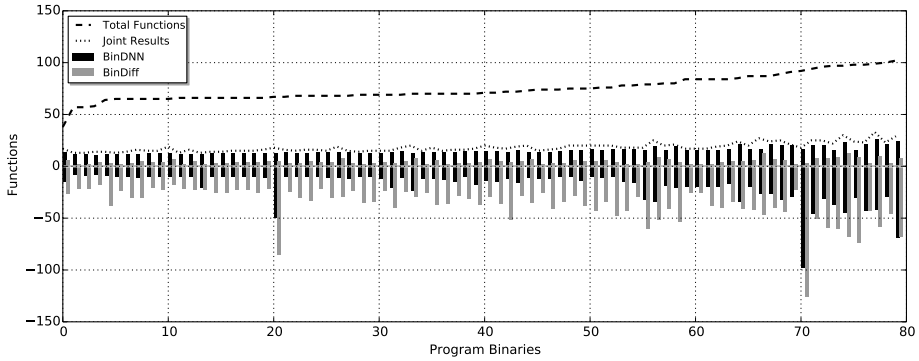

Fig. 10: Architecture resilience comparison with BinDiff and BINDNN. When matching functions from binaries compiled across architectures, BINDNN vastly outperforms BinDiff. These binaries were compiled with different optimization levels (`O0` and `O3`), and on two different architectures (`arm` and `x86`). The positive values are correct functions matches; whereas, the negative values are incorrect function matches.

we see that BINDNN was still able to successfully match a number of functions across the programs, whereas BinDiff does not perform as well.

These results show the weakness of systems like BinDiff, which rely on graph isomorphic methods. Since the structure of assembly code for the two programs is substantially different, the control graphs end up being different, causing these methods to fail. Even though BINDNN uses instruction sequences, and these binaries use different instructions (as they are on different architectures), our system can still find function matches. This is due to the nature of the deep neural network. It was able to develop approximation functions during the training phase that can map instruction sequences across architectures to the same source code function. Additionally, by reducing the threshold, BINDNN can find more true function matches at the cost of adding in additional false matches, as was the case with the previous optimization level experiment. Additionally, we see that the union of both method's results increases BINDNN's results by 26% on average (and BinDiff's by nearly 190%). However, this would also increase BINDNN's false positive rate by substantial amount. Hence, in this use case, it is actually detrimental to combine the results of both methods, and is instead better to only use BINDNN's classifications.

4.5 Resilience to Compiler and Optimization Differences

We use a different configuration to analyze the ability of BINDNN and BinDiff to detect function matches in program compiled with both a different compiler and different compiler optimization levels. Specifically, we use programs compiled for `x86` with `icc -O2` and `gcc -O3`. In Figure 8, we see the relationship between the threshold and classification rate. As with the multiple architecture experiment, we see that BINDNN provides the best coverage around 0.92, without increasing the number of false positives an excessive amount. However, there are still an average of approximately 330 incorrectly

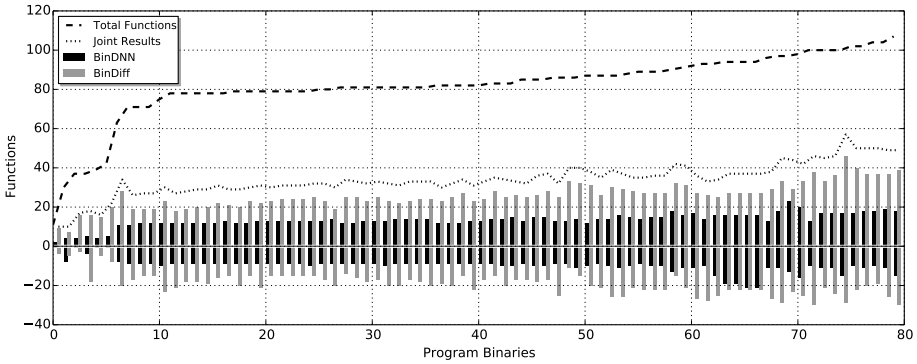

Fig. 11: Compiler and compiler optimization level resilience comparison. Both BINDNN and BinDiff struggle to keep their incorrect matches low when working on programs compiled on `icc -O2` and `gcc -O3`. However, we do see that BINDNN is able to a number of function matches, whereas BinDiff cannot. The positive values are correct functions matches, and the negative values are incorrect function matches.

matched functions with this threshold. We see an increase in the rate at which the incorrect matches fall off as the threshold value approaches and passes 0.98. Therefore, it is beneficial to select a threshold value greater than 0.99, we find an appropriate value to be 0.997.

Figure 11 shows the comparison of function matches for BINDNN and BinDiff. Once again, we only show the results for the shortest 80 functions to make presentation more clear. The results were comparable across the entire data set. In this experiment, we see that both BINDNN and BinDiff struggle to maintain a low number of incorrect matches. We see that generally BinDiff has a higher true positive and false positive rate than BINDNN. However, we see that the union of both method's results provides an overall increase for both systems. Once again indicating that an ensemble use between these or similar methods may be effective.

4.6 Network Limitations

We can further improve the classification rate of the network—currently at 93%—by tuning the architecture, feature vectors, and hyperparameters. However, tuning alone is insufficient as some of the network's inaccuracy is due to missing information. For example, we currently remove the arguments for each of the instructions when constructing the feature vectors. We do this because our network represents functions as sequences of indices into a global vocabulary. If we were to naively include each of the instructions along with their possible arguments, the size of the vocabulary would quickly become intractably large. While this makes our approach more tractable, it also reduces the fidelity of the function representations passed to the network. We plan to explore this area more in future work.

5 Conclusions

In this paper we proposed and evaluated BINDNN, a new method for determining if two assembly instruction sequences originate from the same source code. Our method allows an analyst to prioritize their limited resources by filtering large swaths of code, removing functions that have been previously analyzed, and locating functions present in other malicious programs. We overcome the challenges posed by differences in compiler, compiler optimization level, and target architecture by framing the problem as natural language processing. This framing enables us to leverage deep learning as the foundation for resilient function matching. Our evaluation shows BINDNN is more effective than current state-of-the-art tools (e.g., BinDiff) when matching functions across binaries compiled for different architectures.

BINDNN's greatest strength is its ability to augment, not supplant, existing approaches. Indeed, we show that when BINDNN is combined with BinDiff we boost performance for both methods. For example, we saw an improvement of 46% over using BinDiff alone when comparing functions compiled at different optimization levels. While such ensemble methods are effective, we must consider the relative strengths and weaknesses of each method. Take the cross-architecture results for example; combining BINDNN and BinDiff had little effect as BinDiff's false positive rate was too high.

As the demand for binary analysis rises so too will the need for triage techniques. Put simply, there are more malicious binaries introduced every month than analysts can reverse engineer. However, no technique is a panacea. BINDNN represents an important step in addressing the limitations of previous approaches and provides analysts with another tool in their fight against malware.

6 Acknowledgments

Research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

Additionally, this material is based upon work supported by the National Science Foundation under Grant No. CNS-1228700 and CNS-1064900. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

1. T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley. Byteweight: Learning to recognize functions in binary code. In *USENIX Security Symposium*, 2014.

2. Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on*, 5(2):157–166, 1994.
3. M. Bourquin, A. King, and E. Robbins. Binslayer: accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, page 4. ACM, 2013.
4. D. Ciregan, U. Meier, and J. Schmidhuber. Multi-column deep neural networks for image classification. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 3642–3649. IEEE, 2012.
5. R. Collobert and J. Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167. ACM, 2008.
6. T. Dullien and R. Rolles. Graph-based comparison of executable objects (english version). *SSTIC*, 5:1–3, 2005.
7. P. Duygulu, K. Barnard, J. F. de Freitas, and D. A. Forsyth. Object recognition as machine translation: Learning a lexicon for a fixed image vocabulary. In *European conference on computer vision*, pages 97–112. Springer, 2002.
8. M. Egele, M. Woo, P. Chapman, and D. Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *USENIX Security Symposium*, 2014.
9. D. Gao, M. K. Reiter, and D. Song. Binhunt: Automatically finding semantic differences in binary programs. In *Information and Communications Security*, pages 238–255. Springer, 2008.
10. F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.
11. Hex-Rays. Hex-rays: Ida pro disassembler and debugger, 2016. `https://www.hex-rays.com/products/ida/`.
12. S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.
13. E. R. Jacobson, N. Rosenblum, and B. P. Miller. Labeling library functions in stripped binaries. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, pages 1–8. ACM, 2011.
14. D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.
15. N. Lageman, M. Lindsey, and W. Glodek. Detecting malicious android applications from runtime behavior. In *Military Communications Conference, MILCOM 2015-2015 IEEE*, pages 324–329. IEEE, 2015.
16. Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
17. T. Mikolov, W.-t. Yih, and G. Zweig. Linguistic regularities in continuous space word representations. In *HLT-NAACL*, pages 746–751, 2013.
18. S. Mukkamala, G. Janoski, and A. Sung. Intrusion detection using neural networks and support vector machines. In *Neural Networks, 2002. IJCNN'02. Proceedings of the 2002 International Joint Conference on*, volume 2, pages 1702–1707. IEEE, 2002.
19. M. Pontil and A. Verri. Support vector machines for 3d object recognition. *IEEE transactions on pattern analysis and machine intelligence*, 20(6):637–646, 1998.
20. N. E. Rosenblum, X. Zhu, B. P. Miller, and K. Hunt. Learning to analyze binary computer code. In *AAAI*, pages 798–804, 2008.
21. S. Saad, I. Traore, A. Ghorbani, B. Sayed, D. Zhao, W. Lu, J. Felix, and P. Hakimian. Detecting p2p botnets through network behavior analysis and machine learning. In *Privacy, Security and Trust (PST), 2011 Ninth Annual International Conference on*, pages 174–180. IEEE, 2011.

Fig. 12: Network Architecture We use an 8 layer deep learning model. It is primarily built around the LSTM layers, which develop the temporal relationships between instructions. The CNN layers vastly increase the stability of the model while also aiding in preventing it from overfitting. The DNN layers at the end bring everything from the previous layers together in a classification value stating if it was given matching function representations.

22. T. N. Sainath, O. Vinyals, A. Senior, and H. Sak. Convolutional, long short-term memory, fully connected deep neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 4580–4584. IEEE, 2015.
23. E. C. R. Shin, D. Song, and R. Moazzezi. Recognizing functions in binaries with neural networks. In *24th USENIX Conference on Security Symposium (SEC), USENIX Association, Washington, DC*, 2015.
24. C. Sinclair, L. Pierce, and S. Matzner. An application of machine learning to network intrusion detection. In *Computer Security Applications Conference, 1999.(ACSAC'99) Proceedings. 15th Annual*, pages 371–377. IEEE, 1999.
25. P. J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.
26. Zynamics. zynamics bindiff, 2016. `https://www.zynamics.com/bindiff.html`.

A Network Architecture

We design what is essentially an 8 layer network. The first layer is an embedding layer, this layer learns mappings for global vocabulary indexes into dense vectors. This layer is especially important for our first goal, the ability to recognize similar instructions that have different names. This layer allows the model to more easily map instructions that appear to have similar meaning to real values that are close.

Next, we pass the output from the embedding layer to two 1 dimensional convolutional layers. The convolutional layers each use 64 kernels with filter size 3. These layers allows the model to learn small groups of instructions. This allows the model to classify not only on the exact sequence of instructions that makes up the function representation, but also the sequence of meaningful instruction subsequences. From the convolutional layers we downscale by a factor of 2 by using Max Pooling.

Next, we use two long-short term memory (LSTM) layers with 70 cells each. These layers are the heart of the model. They learn the temporal relationships between instructions. By using LSTM layers, we are better able to overcome the vanishing or exploding gradient problem associated with standard RNNs [2], which in turn allows us to more easily learn long-term dependencies within the functions.

Lastly, we incorporate dropout throughout the model to help it resist overfitting. Specifically, we include 25% dropout in-between the two convolutional layers, and we include 50% dropout between the final two dense layers. The model also uses a sigmoid activation function. We provide a diagram of the network architecture in Figure 12.