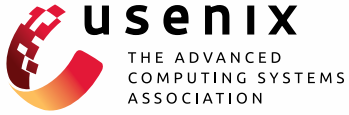

Recognizing Functions in Binaries with Neural Networks

Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi, *University of California, Berkeley*

https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/shin

This paper is included in the Proceedings of the 24th USENIX Security Symposium

August 12–14, 2015 • Washington, D.C.

ISBN 978-1-931971-232

Open access to the Proceedings of the 24th USENIX Security Symposium is sponsored by USENIX

Recognizing Functions in Binaries with Neural Networks

Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi
University of California, Berkeley
{*ricshin, dawnsong, rezamoazzezi*}*@berkeley.edu*

Abstract

Binary analysis facilitates many important applications like malware detection and automatically fixing vulnerable software. In this paper, we propose to apply artificial neural networks to solve important yet difficult problems in binary analysis. Specifically, we tackle the problem of function identification, a crucial first step in many binary analysis techniques. Although neural networks have undergone a renaissance in the past few years, achieving breakthrough results in multiple application domains such as visual object recognition, language modeling, and speech recognition, no researchers have yet attempted to apply these techniques to problems in binary analysis. Using a dataset from prior work, we show that recurrent neural networks can identify functions in binaries with greater accuracy and efficiency than the state-of-the-art machine-learning-based method. We can train the model an order of magnitude faster and evaluate it on binaries hundreds of times faster. Furthermore, it halves the error rate on six out of eight benchmarks, and performs comparably on the remaining two.

1 Introduction

Binary analysis enables many useful applications in computer security, given the plethora of possible situations in which the original high-level source code is unavailable, has been lost, or is otherwise inconvenient to use. For example, detection of malware, hardening software against common vulnerabilities, and protocol reverse-engineering are most useful when the procedures involved can directly operate on binaries.

The central challenge of binary analysis is perhaps the lack of high-level semantic structure within binaries, as compilers discard it from the source code during the process of compilation. Malware authors often go a step further and obfuscate their output in an attempt to frustrate any possible analysis by researchers.

Functions are a seemingly basic yet fundamental piece of structure in all programs, but most binaries come as an undifferentiated sequence of machine-language instructions without any information about how parts group into functions. Therefore, the many binary analysis techniques which rely on function boundary information must first attempt to recover it through *function identification*. For instance, function identification can assist the addition of control-flow integrity enforcement to binaries, in restricting jumps appropriately. Similarly, decompilers and debuggers need to know the locations of functions to provide useful output to the user [2].

Several previous works have attempted the function identification task, ranging from simple heuristics to approaches using machine learning. The problem might seem simple at first glance, but Bao et al. showed with ByteWeight [2], a recently-proposed machine-learning-based approach, that the simpler techniques used by popular tools like IDA Pro and the CMU Binary Analysis Platform have relatively poor accuracy. By constructing signatures of function starts as weighted prefix trees, ByteWeight greatly improves on the accuracy of function identification results compared to past work. Nevertheless, it leaves much room for improvement, especially in terms of computational efficiency: the authors report that training on their dataset of 2,064 binaries required 587 compute-hours, whereas running the method on the dataset took on the order of several compute-days. Also, while ByteWeight achieves about 98% accuracy on some benchmarks, it performs at just 92-93% on some others.

In this paper, we propose a new approach to function identification leveraging artificial neural networks. First proposed in the 1940s, artificial neural networks arose as a simple approximation of interconnected biological neurons in the central nervous systems of animals, and have remained an active area of of research since then. However, in the past few years, neural networks have experienced a significant surge in popularity (often under the name "deep learning"), largely been driven by

new empirical results. The vastly larger amounts of processing power and storage available today enabled researchers to train much larger networks containing many more stages of processing (hence the "deep" appellation) and parameters than before, making full use of the massive labeled datasets available today; these factors have led to repeated breakthroughs in benchmarks of the computer vision and speech recognition communities, among others.

We note some attractive features of neural networks. First, they can learn directly from the original representation with minimal preprocessing (or "feature engineering") needed. As an example, the preprocessing for images might discard information about the precise shading of objects; for binaries, Bao et al. disassembles the code into instructions and removes immediate operands from them. Second, neural networks can learn *end-to-end*, where each of its constituent stages are trained simultaneously in order to best solve the end goal. In contrast, other state-of-the-art approaches to tasks like machine translation or question answering use pipelines of discrete components trained separately at an unrelated task, such as parsers or part-of-speech taggers. Empirical evidence suggests that end-to-end learning enables each stage to directly learn the intermediate representations necessary to solve the task, with less need for preconceived notions (such as syntax trees) about what they should look like.

Given the success that neural networks have shown in other applications, we raise the question of whether they would also prove adept at problems in binary analysis, such as function identification. Our search turned up no other works which attempted using neural networks to solve problems in binary analysis. Nevertheless, our experimental results show that they can successfully solve the function identification task accurately and efficiently. If the experience in other fields can serve as a guide, they may also prove useful for more complicated tasks in program and binary analysis, especially for those which require complicated modeling or analysis difficult to specify by hand. Furthermore, advances with neural networks in other applications might prove directly adaptable and lead to "free" gains in performance; this work certainly relies on general advances within neural networks targeted at entirely different applications.

With our proposed solution, we train a recurrent neural network to take bytes of the binary as input, and predict, for each location, whether a function boundary is present at that location. We found that we did not need to perform any preprocessing, such as disassembly or normalization of immediates, in order to obtain good results. We evaluate our approach using the dataset provided by Bao et al. [2], enabling a direct comparison. We found that recurrent neural networks can learn much more efficiently than ByteWeight, which reported using 587 compute-hours; we can train on the same dataset in 80 compute-hours, while achieving similar or better accuracy. Testing the method on the dataset takes only about 43 minutes of computation, whereas Bao et al. [2] reported needing over 2 weeks.

In the rest of the paper, we first precisely define the problem at hand. We explain the necessary background in neural networks, and describe the particular architecture we chose to use for our method. We give the results of our empirical evaluation, describe some related works in the areas of function identification and neural networks, and then conclude with some discussion.

We make the following contributions in this paper:

- We find that neural networks are a viable approach towards solving some problems in binary analysis.
- In particular, we show that recurrent neural networks can solve the function identification problem more efficiently than the previous state-of-the-art, as shown by empirical evaluation on a dataset consisting of multiple operating systems, architectures, compilers, and compiler options.
- We describe the challenges we faced in correctly applying neural networks to this problem, and how to address them.

2 Problem Definition

We first define notation so that we can precisely define the function identification task that we address in this paper. We then provide a formal definition of function identification.

2.1 Notation

We concern ourselves with the machine code contained within a program binary or library. A typical executable contains many different sections containing various information in addition to the code: for example, dynamically-linked libraries to load, constant strings, and statically-allocated variables, all of which we ignore.

We treat the code C itself as a sequence of bytes $C[0], C[1], \cdots, C[l]$, where $C[i] \in \mathbb{Z}_{256}$ is the i^{th} byte in the sequence. We denote the n functions in the binary as $f_1, \cdots, f_n$. We label the indices of the bytes of code which belong to each function f_i (i.e., the bytes corresponding to instructions which might get executed while running that function) as $f_{i,1}, \cdots, f_{i,l_i}$, where l_i is the total number of bytes in f_i. Without loss of generality, we assume $f_{i,1} < f_{i,2} < \cdots < f_{i,k}$. Each byte may belong to any number of functions, and functions may contain any set of bytes, contiguous or not.

Note that we defined the code and functions as sets of bytes rather than instructions. In the x86 and x86-64 ISAs, a sequence of bytes can have many plausible instruction decodings depending on the offset at which decoding begins; therefore each byte might belong to a handful of possible instructions. Working in terms of bytes allows us to avoid this ambiguity.

2.2 Task definition

Let us assume that we have access the code C of a binary, but no information about the functions $f_1, \cdots, f_n$ within the code. We define the following tasks:

- Function start identification: Given C, find $\{f_{1,1}, \cdots, f_{n,1}\}$. In other words, recover the location of the first byte of each function.
- Function end identification: Given C, find $\{f_{1,l_1}, \cdots, f_{n,l_n}\}$. In other words, find the bytes where each of the n functions in the binary ends. The length of each function is not given.
- Function boundary identification: Given C, find $\{(f_{1,1}, f_{1,l_1}) \cdots, (f_{n,1}, f_{n,l_n})\}$. In other words, discover the location of the first and last byte within each function. This task is more than a simple combination of function start and end identification. If the starts and ends of functions have been identified separately, they need to be paired correctly so that each pair contains the start and end of the same function.
- General function identification: Given C, find $\{(f_{1,1}, f_{1,2}, \cdots, f_{1,l_1}) \cdots, (f_{n,1}, f_{n,2}, \cdots f_{n,l_n})\}$; i.e., determine the number of functions in the file, and all of the bytes which make up each function.

Function boundary identification is a superset of function start and end identification, whereas general function identification is a superset of all other tasks. In this paper, we attempt the first three problems, and leave the fourth to future work.

2.3 Metrics

To evaluate results from the model, we use the precision, recall, and F1 metrics. They have the following definitions:

$$\text{Precision} = \frac{\text{TP}}{\text{TP}+\text{FP}}$$
$$\text{Recall} = \frac{\text{TP}}{\text{TP}+\text{FN}}$$
$$\text{F1} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision}+\text{Recall}}$$

where TP is the number of true positive predictions, FP is the number of false positive predictions, and FN is the number of false negative predictions. The F1 score is the harmonic mean of precision and recall and allows us to conveniently compare different results using one number.

Since most bytes within a program do not begin or end functions, these metrics can give a better picture of the effectiveness of the model than the simple accuracy metric. For example, predicting that there exists no functions in the code would give greater than 99.9% accuracy, since fewer than 0.1% of the bytes begin or end a function. The accuracy metric does not reveal that these predictions would be mostly useless. In contrast, the 0% recall these predictions would achieve makes it clear.

2.4 Examples

In Figure 1, we show an example of a short C function and its corresponding binary code after compilation at two different optimization levels.

The code in Figure 1b contains very clear markers of function start and end: the function prologue of `push %rbp` and `mov %rsp,%rbp` saves the caller's stack frame, and the function ends with `retq` which occurs nowhere else within the function. In contrast, Figure 1c does not use the stack at all, so the function begins with some accesses to the function arguments passed in `edi` and `esi`; looking for `push %rbp` would fail. Moreover, similar accesses to arguments occur again within the body of the function, making it difficult to solely rely on that as a marker of the function start. Likewise, `retq` occurs twice within the code, and so predicting a function end when we see this instruction would fail.

This example gives an instance of why function identification can pose much difficulty, with simple heuristics unlikely to suffice, contrary to what intuition might suggest.

3 Background

In this section, we describe what neural networks are and how they are trained. In particular, we focus on the various forms of recurrent neural networks, which are the class of model we use for our method.

3.1 Multi-layer perceptrons

A multi-layer perceptron (MLP), also referred to as a feedforward neural network, is a function $L : \mathbb{R}^s \to \mathbb{R}^t$ parameterized in a particular way. As its name implies, a multi-layer perceptron consists of multiple layers L_i,

```
int mul_inv(int a, int b) {
  int b0 = b, t, q;
  int x0 = 0, x1 = 1;
  if (b == 1) return 1;
  while (a > 1) {
    q = a / b;
    t = b, b = a % b, a = t;
    t = x0, x0 = x1 - q * x0, x1 = t;
  }
  if (x1 < 0) x1 += b0;
  return x1;
}
```

(a) A C function which computes the modular multiplicative inverse.

```
00000000004005a1 <mul_inv>:
  4005a1: push   %rbp
  4005a2: mov    %rsp,%rbp
  4005a5: mov    %edi,-0x24(%rbp)
  4005a8: mov    %esi,-0x28(%rbp)
  4005ab: mov    -0x28(%rbp),%eax
...
  400615: jns    40061d <mul_inv+0x7c>
  400617: mov    -0xc(%rbp),%eax
  40061a: add    %eax,-0x8(%rbp)
  40061d: mov    -0x8(%rbp),%eax
  400620: pop    %rbp
  400621: retq
```

(b) Compiled with `gcc -O0`.

```
0000000000400830 <mul_inv>:
  400830: cmp    $0x1,%esi
  400833: mov    %edi,%eax
  400835: je     400878 <mul_inv+0x48>
  400837: cmp    $0x1,%edi
  40083a: jle    400878 <mul_inv+0x48>
  40083c: mov    %esi,%ecx
  40083e: mov    $0x1,%r8d
  400844: xor    %edi,%edi
  400846: jmp    400855 <mul_inv+0x25>
  400848: nopl   0x0(%rax,%rax,1)
  40084f:
...
  400869: jg     400850 <mul_inv+0x20>
  40086b: add    %edi,%esi
  40086d: mov    %edi,%eax
  40086f: test   %edi,%edi
  400871: cmovs  %esi,%eax
  400874: retq
  400875: nopl   (%rax)
  400878: mov    $0x1,%eax
  40087d: retq
  40087e: xchg   %ax,%ax
```

(c) Compiled with `gcc -O3`. Function ends at `40087d`; `40087e` is padding between this function and the next one.

Figure 1: An example function in C (taken from `http://rosettacode.org/wiki/Modular_inverse#C`), and its corresponding machine code (with uninteresting parts omitted for brevity). The function was compiled using GCC 4.9.1 on Linux x86-64. The `-O3` version does not contain a conventional function prologue and epilogue which manipulates the stack or frame pointer.

each of which computes

$$\begin{aligned} L_i &: \mathbb{R}^{m_{i-1}} \to \mathbb{R}^{m_i} \\ G &: \mathbb{R} \to \mathbb{R} \\ L_i(x) &= G(W_i x + b_i) \\ W_i &\in \mathbb{R}^{m_i \times m_{i-1}} \\ b_i &\in \mathbb{R}^{m_i} \end{aligned}$$

then the entirety (consisting of k layers) is simply these layers composed together:

$$\begin{aligned} L(x) &= L_k(L_{k-1}(\cdots(L_1(x)))) \\ m_0 &= s \\ m_k &= t \end{aligned}$$

with the dimensions of the output of one layer matching the dimensions of the input of the subsequent layer. The m_i are the dimensionality of the input and the ultimate output of the network, as well as the intermediates produced by each of the layers.

The term "layer" is often used to refer to not the parameters of the functions L_i, but the inputs or outputs of these functions. In turn, each element of the inputs or outputs of the functions are often called "units", or by analogy, "neurons".

In this definition, G is referred to as an *activation function* or *nonlinearity*, and computed separately for each element. Without the activation function, L would simply be an affine function which we could write as $Wx+b$, which does not enable the expressivity that we need. Common nonlinearities are the logistic sigmoid function and the hyperbolic tangent function:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$

$$\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$$

which have the ranges of $(0,1)$ and $(-1,1)$, respectively.

Usually, the final layer will have no activation function because we do not wish to bound the output to a limited range, or a softmax function if we want to use the MLP as a multi-class classifier, so that we can interpret the values as a probability distribution. The softmax function is computed as follows:

$$S(\mathbf{x})_i = \frac{e^{\mathbf{x}_i}}{\sum_{k=1}^{n} e^{\mathbf{x}_k}}$$

Note that unlike the other activation functions, it does not operate elementwise. Due to the normalization term in the denominator, $S(\mathbf{x})$ sums to 1. A multi-layer perceptron consisting of one layer with a softmax activation function is equivalent to multi-class logistic regression.

3.2 Loss functions

Now that we have defined multi-layer perceptrons as a class of parameterized functions, we need a method to

find appropriate parameters so that the neural network does what we want. First, we define a *loss function* in order to quantify how much differently the network behaves from our target. A common loss function is the squared Euclidean distance:

$$d(y,\hat{y}) = \|y-\hat{y}\|_2^2$$

where y is the "true" output and $\hat{y}$ is the one produced by the neural network.

In the multi-class classification case, if y is the correct class and $\pi(x)$ is the probability distribution produced by the neural network, then we can use the negative log probability:

$$d(y,\pi(x)) = -\log \pi(x)_y$$

Usually, we will have a list of correct input-output pairs $(x_1,y_1),\cdots,(x_n,y_n)$ for the purpose of training the network. Then we can seek to minimize the mean of the losses, or $\frac{1}{n}\sum_{i=1}^{n} d(y_i,f(x_i))$. We use this type of loss function throughout this paper.

3.3 Gradient descent and backpropagation

To minimize the loss, and therefore obtain a neural network which performs our desired function, we can consider various standard optimization methods. Specifically, we wish to minimize D defined as such:

$$\begin{aligned}\theta &= (W_1,b_1,\cdots,W_k,b_k)\\ D(\theta) &= \frac{1}{n}\sum_{i=1}^{n} d(y_i,f_\theta(x_i))\\ \min_\theta \quad & D(\theta)\end{aligned}$$

A typical way to minimize differentiable functions is gradient descent, which works by repeated applications of the following update:

$$\theta' = \theta - \alpha\cdot\frac{\partial D(\theta)}{\partial\theta}$$

where α is generally a small number. Intuitively, the derivative allows us to analytically determine which direction we should move in within each dimension of θ to reduce the value of F. Subtracting a small multiple of the gradient performs this function.

If D is convex, this procedure is guaranteed to converge at the optimal value of θ given appropriate choices of α. Many machine learning models and classifiers involve optimizing a convex function in a similar way. Unfortunately, neural networks are generally non-convex in its parameters, allowing for a richer class of possible functions, but which means that these theoretical guarantees do not hold. Instead, the procedure may lead us to a local optimum or a saddle point where the derivative is zero.

We now need the derivative of D. Estimating the derivative numerically seems a simple and straightforward solution, but it is a highly inefficient one requiring as many evaluations of D as the dimensionality of θ. Instead, we can use a method called *backpropagation* to compute the derivative analytically. We describe the details of backpropagation in Section A.

3.4 Recurrent neural networks

While multi-layer perceptrons can approximate a wide variety of functions, they can only operate on inputs of fixed size and produce an output of fixed size. In principle, given a large input, we could divide it into fixed-size pieces and give them separately to a multi-layer perceptron. However, the output of each piece depends only on that input piece, and we cannot represent any dependencies between parts of the input in one piece and the output for a different piece.

Recurrent neural networks are one paradigm for addressing this conundrum, and map sequences to sequences (recursive neural networks, which have the same initialism, are an alternative developed for computing on trees).

We can formally define them in the following way:

$$\begin{aligned}L &: \mathbb{R}^m\times\mathbb{R}^n\to\mathbb{R}^n\\ G &: \mathbb{R}\to\mathbb{R}\\ L(x,h) &= G(W_{hx}x+W_{hh}h+b)\\ W_{hx} &\in \mathbb{R}^{n\times m}\\ W_{hh} &\in \mathbb{R}^{n\times n}\\ b &\in \mathbb{R}^n\end{aligned}$$

Given an input sequence $(x_1,\cdots,x_T)$ (where $x_i\in R^m$), we compute $(h_1,\cdots,h_T)$ like this:

$$\begin{aligned}h_0 &= 0\\ h_1 &= L(x_1,h_0)\\ &\vdots\\ h_T &= L(x_T,h_{T-1})\end{aligned}$$

Note that the operation on each elements uses the same weights. Nevertheless, the use of h enables the network to remember information from past elements to use while processing the current element, and propagate information into the future.

We can use the h_ts as inputs to another recurrent neural network, or apply to them a linear transformation possibly with a softmax activation function (as done in the final layer of a multi-layer perceptron):

$$y_i = S(W_{yh}h_t+b)$$

To define a loss function for a recurrent neural network, we can apply a loss function for a multi-layer perceptron separately to each input-output pair within the sequence and simply sum the losses together:

$$d(y,\hat{y}) = \sum_{i=1}^{T} d(y_t, \hat{y}_t)$$

The input and output sequences of a recurrent neural network need not have the same lengths. For instance, we might allow an arbitrary number of inputs but only one output to summarize the contents of the input in some way. In this case, we can simply adjust the loss function to only compute the loss at the relevant parts of the output sequence. We can also train a recurrent neural network to map an input sequence to an arbitrary number of output symbols, if we run the network to obtain some number of outputs until it produces a special 'stop' output.

As with the multi-layer perceptron, we would like to learn appropriate parameters so that the recurrent neural network parameterized with them computes a desired function, using gradient descent. We can compute the derivative of the RNN with respect to its parameters in the same way as earlier. In particular, we can unroll the RNN so that it becomes a long feedforward neural network which computes on a fixed-length sequence, and compute the gradient for this network using an appropriate loss function with backpropagation. After unrolling, note that the time-dependent layers should share the same weights This procedure is also called *backpropagation through time* [16].

3.5 Limitations of recurrent neural networks

In this section, we point out some limitations of recurrent neural networks which can limit their usefulness.

As specified in this paper, recurrent neural networks cannot compute for an arbitrary number of timesteps before computing the answer. For example, RNNs can easily compute the parity of an arbitrarily long stream of bits [15], as this requires a constant number of operations per input. In contrast, we can reason that a RNN could not multiply numbers of arbitrary size, as multiplication is a $O(n^2)$ operation on the length of the numbers [22].

Also, h has a fixed size which we cannot easily adapt if necessary in order to store more information. For example, previous works have shown success with using RNNs for machine translation, in which the RNN first reads a sentence in the source language and stores its meaning in h before producing the corresponding words in the target language using the information in h. While we can pick a size for h such that it has enough capacity to store information on a typical-length sentence, we can imagine that this scheme would break down for a sentence of sufficient length.

The most-studied limitation revolves around difficulties in training a recurrent neural network, due to what are referred to as the vanishing gradient and exploding gradient problems [16]. Consider that

$$\frac{\partial h_v}{\partial h_t} = \prod_{v \geq i > t} \frac{\partial h_i}{\partial h_{i-1}} = \prod_{v \geq i > t} W_{hh} G'(h_{i-1})$$

The repeated multiplication with W_{hh} ($v-t$ times) can cause $\frac{\partial h_v}{\partial h_t}$ to grow exponentially large ("explodes") or go to 0 ("vanishes") depending on whether the largest eigenvalue of W_{hh} is greater or smaller than 1. Therefore, an input will often have a very large or vanishingly small effect on an output which occurs far in the future, in terms of the gradient computation. For exploding gradients, a simple solution involves rescaling the gradient to a fixed norm if its magnitude is too large. On the other hand, dealing with vanishing gradients can prove more challenging.

3.6 Long Short-Term Memory and Gated Recurrent Units

To avoid the exploding and vanishing gradient problems with recurrent neural networks, previous work has proposed RNN architectures carefully designed to remove the long-range multiplicative characteristics of RNNs which lead to these problems.

Long Short-Term Memory (LSTM), one of these architectures, have enabled impressive empirical results in areas such as speech recognition, machine translation, and image captioning. Within this model, the state which propagates through time has no multiplicative updates at each step; instead, it is stored in a *memory cell* c_t which receives additive updates, combined with a mechanism for erasing irrelevant information from the previous time step. The "input modulation gate" (g) and the "forget gate" (f), respectively, control whether the memory cell receives the additive update or discards (some part of the) previous memory cell contents.

Following the notation in Zaremba et al. [23], we can formally define the LSTM:

$$\begin{aligned} x_t, h_{t-1}, c_{t-1} &\rightarrow h_t, c_t \\ i &= \sigma(W_{xi}x_t + W_{hi}h_{t-1}) \\ f &= \sigma(W_{xf}x_t + W_{hf}h_{t-1}) \\ o &= \sigma(W_{xo}x_t + W_{ho}h_{t-1}) \\ g &= \tanh(W_{xg}x_t + W_{hg}h_{t-1}) \\ c_t &= f \odot c_{t-1} + i \odot g \\ h_t &= o \odot \tanh(c_t) \end{aligned}$$

Here, $\odot$ represents element-wise multiplication.

In principle, these gates enable the gradient to propagate across long time scales, since the LSTM can ignore irrelevant inputs through the input modulation gate, remember information only until necessary using the forget gate, and output only relevant information using the output gate. When the forget gate is "open", i.e. close to 1, then the gradient will propagate mostly unchanged. The input and output at each time step only influences the gradient when the corresponding gates are open.

Gated Recurrent Units (GRU) have been proposed more recently as a simpler alternative to LSTMs, while sharing the same goals of avoiding the long-range dependency problems that have plagued RNNs. The main differences lie in that there exists no separate memory state c_t from the hidden state h_t, and the network exposes the entire hidden state at each time step. The forget gate interpolates between the previous hidden state and the new input i, with no separate input modulation gate. Instead, g modulates the amount of influence the previous hidden state has on i.

We define the GRU formally:

$$\begin{aligned} x_t, h_{t-1} &\rightarrow h_t \\ g &= \tanh(W_{xg}x_t + W_{hg}h_{t-1}) \\ i &= \tanh(W_{xi}x_t + W_{hi}(g \odot h_{t-1})) \\ f &= \sigma(W_{xf}x_t + W_{hf}h_{t-1}) \\ h_t &= f \odot h_{t-1} + (1-f) \odot i \end{aligned}$$

While the GRU theoretically lacks some of the flexibility provided by the LSTM, it is both simpler to implement and easier to compute, requiring about half as many calculations in each time step compared to the LSTM.

4 Methods

In this section, we describe how we built upon the background in Section 3 to perform the task of function identification.

4.1 Basic architecture

Our simplest architecture uses a recurrent neural network, described in Section 3.4, to process each byte and and output a decision for that byte as to whether it begins a function or not.

Recall that neural networks, as we have defined them, take real-valued vectors $\mathbb{R}^m$ as input, containing m real values. In contrast, a byte is a single 8-bit integer, which can have one of 256 ($= 2^8$) possible values. We cannot input a byte into the neural network directly and need to convert them into a real-valued vector.

Converting the 8-bit integer into a single floating-point number to input into the neural network might seem like a reasonable solution; however, neural networks process their inputs by multiplying them with the weight parameters, which only makes sense when the input values represent intensities (like brightness or loudness).

Instead, we use "one-hot encoding", which converts a byte into a $\mathbb{R}^{256}$ vector (since a byte can have 256 distinct values) where exactly one of the values is 1 and all others are 0. The byte's identity determines the location of the 1 within the vector. For example, a NUL byte (0) would be represented as

$$[1 \quad \underbrace{0 \cdots 0}_{\text{255 elements}} \ ].$$

and a `nop` in x86 (0x90, or 144) would be

$$[\ \underbrace{0 \cdots 0}_{\text{144 elements}} \ 1 \ \underbrace{0 \cdots 0}_{\text{111 elements}} \ ].$$

Multiplying a matrix A with a one-hot vector x is equivalent to extracting a column from A. In our case, the RNN multiplies a parameter matrix $W_{hx} \in \mathbb{R}^{m \times 256}$ with the one-hot input x, which is equivalent to selecting a column from W_{hx}. Effectively, each byte of input is represented with a h-dimensional vector during computation of the RNN, with the precise representation learned during training of the neural network. Such a mapping is often referred to as an *embedding*. Such embeddings have proved useful in other fields such as natural language processing, with embeddings of words into high-dimensional spaces showing interesting properties.

We could have instead considered encoding the byte as a $\mathbb{R}^8$ vector, with the elements corresponding to the eight bits and having values of 0 or 1. However, this representation imposes the constraint that the embedding of a particular byte is the sum of the embeddings of its constituent bits, even though the bits do not have compositional meaning in typical binary code. We do not further discuss this approach in the paper.

We want our output to serve as a binary classifier at each byte position. We use the softmax function to produce a probability distribution over whether the byte begins (or ends) a function or not. During training, the loss function sums over the error at each position within the sequence. The error at each position is the negative log of the probability that the neural network assigned to the correct answer. We penalize each false positive and false negative equally, without a weighting to discourage one at the expense of the other.

4.2 Optimization with stochastic gradient descent and *rmsprop*

In the beginning, we initialize the weights of the neural network randomly, uniformly drawn from a small range

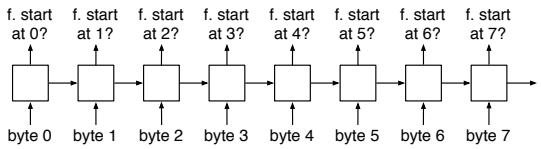

Figure 2: A depiction of the basic architecture of our approach.

near 0. A normal distribution with small variance and mean 0 also sees much use for this purpose. Having the proper initialization can prove crucial to whether we can successfully learn useful parameters for the neural network. We do not initialize the weights to 0, as this leaves the loss function on a saddle point and prevents optimization.

Recall the form of the loss function from Section 3. If we have N different items in the training data, then the loss function for the network requires evaluating the network over all training examples, since it takes the form

$$\frac{1}{N}\sum_{i=1}^{N} d(y_i, \hat{y}_i).$$

Due to how backpropagation works, computing the gradient also requires evaluating the network on all training data. Since we need to compute the gradient a very large number of times during optimization, we would like to avoid performing such an expensive step as a part of it.

Instead of computing the loss over all N items, we can instead compute it over a randomly-selected one at a time. The expectation of the gradient computed in this way equals the gradient averaged over all N examples. Optimization using these gradients is called *stochastic gradient descent*. It is possible to show that given a well-behaved convex function, stochastic gradient descent will find the minimum value. Even in the case of neural networks, where we lack such theoretical guarantees, experience shows that stochastic gradient descent can work quite well; in fact, since computing each gradient takes much less time, results show that stochastic gradient descent allows for much faster convergence in practice.

The most elementary gradient descent methods prescribe changing the parameters in the direction of the gradient each iteration, but optimization of some kinds of functions can benefit from moving in a slightly different direction. Consider a two-dimensional function, which when graphed looks like an elliptical bowl. Then along the axis in which we are closest to the minimum point, the gradient will have the largest magnitude, as the surface of the bowl is steeper in that direction, even though we should move further in the other axis and only a little bit in this one.

In this work, we use a method called *rmsprop* [19]; it involves keeping a running average of the magnitude of each dimension in the gradients seen so far. It then scales each dimension in the gradient, enlarging the dimensions which have a small average and shrinking those which have a large one. This follows the intuition given in the previous paragraph about the elliptical bowl.

We also scale the entire gradient each time by a step size. If the step size is too big, then the optimization might fail as the value of the function does not decrease; if the step size is too small, then optimization will progress slowly or get stuck at a local minimum. Often it makes sense to reduce the learning rate over time, since in the beginning we expect radically-incorrect weights (given their random initialization), whereas after some iterations, the weights should have nearly reached an optimum value. For our experiments, we scaled the learning rate by the inverse square root of the current iteration number (i.e., halved after 4 iterations, quartered after 16 iterations, and so on), which we found to work well.

4.3 Training with mini-batches

In stochastic gradient descent, we compute the gradient of the weights with respect to only one example in each iteration. However, this can cause a large variance in the gradients since each example might significantly differ from one to the next. So instead of computing the gradient over only one example at a time, it can help to average the gradients from a small number of examples, called a *mini-batch*.

While this increases the time needed for each iteration, it does so more modestly than it may initially seem. Evaluating the neural network with a single example involves a large number of matrix-vector multiplications, so we can efficiently and simultaneously evaluate for many examples by replacing these with matrix-matrix multiplications, especially when using highly-optimized linear algebra libraries.

In our application, since each example is a sequence of bytes from a binary, one might vary in length from another. However, to compute with mini-batches efficiently, we need to pack the examples together into a matrix or tensor with padding to extend too-short examples. Then all examples get evaluated for the same number of time steps, so it helps to put examples of similar length together in a mini-batch to avoid wasted computation. Also, we need to take care as to avoid computing the loss over those parts of the mini-batch added as padding.

4.4 Data preparation

For the task of function identification, we can intuitively expect that solving the problem likely does not require

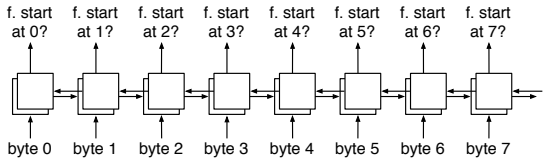

Figure 3: A bi-directional RNN. Note the horizontal arrows pointing in both directions. The forward-propagated and backward-propagated hidden states, represented by the overlapping squares, do not directly interact with each other. However, computing the output uses a concatenation of the two states.

remembering information from hundreds of thousands of bytes in the past or in the future. Code calling other functions can occur far away from the location of that function, and theoretically, we might track such references to help determine where functions occur. In practice, functions typically perform some series of steps at entry and exit, the patterns for which we can learn and should largely suffice for detecting functions.

Therefore, we use fixed-length subsequences taken from binaries instead of entire binaries themselves. Except in rare cases where functions occur near the boundary of the subsequence, there should be enough information to make the determination of the existence of a function or not. Similar to how stochastic gradient descent enables faster convergence by speeding up each update, computing the gradient on truncated sequences takes much less time and enables faster iterations.

We also try reversing the order of bytes in the input before providing it to the neural network, under the intuition that the function prologue, which identifies the beginning of a function and makes it recognizable as such, occurs after the position where we want to predict the beginning of a function. Since the RNN only has access to bytes from before the current position, not after, reversing the order should help the RNN learn.

4.5 Bi-directional RNNs

With the recurrent neural networks discussed in Section 3.4, the output at each time step depends only on the inputs which occur at that time step or before. This model makes sense in some applications where there exists an inherent temporal component to the input; for example, in real-time speech or handwriting recognition. For binary analysis, we have access to the entire binary at once, so there exists no need to confine ourselves in this way.

An extension which allows access to both the past and the future in making a prediction for the present is to combine two recurrent neural networks, one which operates from the beginning of the sequence to the end, and another which operates in the other direction. Figure 3 illustrates the approach.

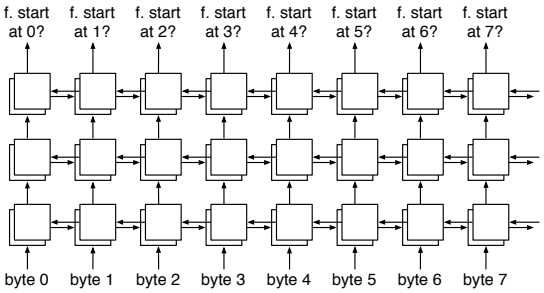

Figure 4: A multi-layer RNN with three bi-directional hidden layers. In the second and third layers, both the forward-propagated and backward-propagated states have access to either state from the previous layer.

In terms of graphical models, we could say that regular (unidirectional) RNNs behave like hidden Markov models, where the hidden state at each time step depends on only the hidden state of the previous time step. Then bidirectional RNNs are analogous to a chain conditional random field, since the hidden state there relates to the hidden states of both the previous and next time steps.

4.6 Multi-layer RNNs

The approaches we have described so far contain only one hidden layer. Depending on the complexity of the pattern we wish to learn, a single hidden layer may prove insufficient due to its limited capacity. If we limit ourselves to one hidden layer, achieving good results may require a very large one, which can significantly increase the amount of processing power required.

In other applications of neural networks like computer vision and speech recognition, using many smaller hidden layers has worked better than using one hidden layer of larger size. During the evaluation, we empirically verify the results of using one versus multiple hidden layers. Figure 4 illustrates an example architecture.

5 Evaluation

In this section, we describe the empirical results we obtained from training a variety of different models on a dataset of binaries. We seek to answer the following questions:

- Can recurrent neural networks successfully solve the problem of function identification in binaries?
- How much computational power do recurrent neural networks require for solving this task?

	ELF x86	ELF x86-64	PE x86	PE x86-64
Number of binaries	1,032	1,032	68	68
Number of bytes	138,547,936	145,544,012	29,093,888	33,351,168
Number of functions	303,238	295,121	93,288	94,548
Average function length	448.84	499.54	292.85	330.03

Table 1: Characteristics of the binary dataset used for evaluation.

	ELF x86			ELF x86-64		
	P	R	F1	P	R	F1
ByteWeight (func. start)	98.41%	97.94%	98.17%	99.14%	98.47%	98.80%
Our models (func. start)	99.56%	99.06%	99.31%	98.80%	97.80%	98.30%
Our models (func. end)	98.69%	97.87%	98.28%	97.45%	95.03%	96.22%
	PE x86			PE x86-64		
	P	R	F1	P	R	F1
ByteWeight (func. start)	93.78%	95.37%	94.57%	97.88%	97.98%	97.93%
Our models (func. start)	99.01%	98.46%	98.74%	99.52%	99.09%	99.31%
Our models (func. end)	99.24%	98.35%	98.79%	99.28%	99.20%	99.24%

Table 2: Function start and end identification: summary of our best results, and comparison with previous work. "P" is precision and "R" is recall. Results of previous work comes from Table 3 of Bao et al. [2]; they did not attempt to identify function ends independently, so we lack those results here.

- How do variations in the model's design affect the performance?

We ran our experiments on Amazon EC2 using `c4.2xlarge` instances, each of which contains 8 cores of a 2.9 GHz Intel Xeon processor and 15 GB of RAM.

5.1 Dataset

Our dataset comes from Bao et al. [2], consisting of 2200 separate binaries. 2064 of the binaries were for Linux, obtained from the `coreutils`, `binutils`, and `findutils` packages. The remaining 136 for Windows consist of binaries from popular open-source projects. Half of the binaries were for x86, and the other half for x86-64. Half of the Linux binaries were compiled with Intel's `icc`, while the other half used `gcc`. The binaries for Windows were compiled using Microsoft Visual Studio. Each binary was compiled with one of four different optimization levels. Table 1 summarizes some statistics from the dataset.

Following the procedure in Bao et al. we trained a separate model for each of the four (architecture, OS) configurations. To report comparable results, we also use 10-fold cross-validation as in Bao et al.; we train ten models for each of the four configurations, where each of the ten models uses a different 10% of the binaries as the testing set.

5.2 Implementation

We implemented our models in Python using Theano [4], a linear algebra and automatic differentiation library designed to aid in implementation of machine learning and optimization methods. In Theano, we specify our model as operations on symbolic variables, allowing for construction of a computation graph that describes the operations necessary to compute the result. It can convert this graph into C/C++ code and automatically compute partial derivatives of functions through application of the chain rule.

While Theano can also compile code for use on the GPU, we only used the CPU in our experiments for simpler implementation. Also, while both training and evaluation of RNNs are amenable to parallelization, we also did not use multi-threading for our experiments, and instead ran an independent experiment on each core.

5.3 Summary of results

Tables 2 and 3 summarize our main experimental results. In both tables, we compare to the results as reported by Bao et al. [2], which are marked as "ByteWeight".

For the function start identification problem, our methods consistently obtain F1 scores in the range of 98-99%. This is in line with the results from Bao et al., except on the PE x86 dataset where we improve by about 4 percentage points in F1 score.

For function boundary identification, we trained two

	ELF x86			ELF x86-64		
	P	R	F1	P	R	F1
ByteWeight	92.78%	92.29%	92.53%	93.22%	92.52%	92.87%
Our models	97.75%	95.34%	96.53%	94.85%	89.91%	92.32%
	PE x86			PE x86-64		
	P	R	F1	P	R	F1
ByteWeight	92.30%	93.91%	93.10%	93.04%	93.13%	93.08%
Our models	97.53%	95.27%	96.39%	98.43%	97.33%	97.88%

Table 3: Function boundary identification: summary of our best results, and comparison with previous work. "P" is precision and "R" is recall.

	ELF x86	ELF x86-64	PE x86	PE x86-64
Our models (func. boundary)	1061.76 s	1017.90 s	236.93 s	264.50 s
ByteWeight (func. start only)	3296.98 s	5718.84 s	10269.19 s	11904.06 s
ByteWeight (func. boundary)	367018.53 s	412223.55 s	54482.30 s	87661.01 s
ByteWeight (func. boundary with RFCR)	457997.09 s	593169.73 s	84602.56 s	97627.44 s

Table 4: Computation time for testing on the data set of 2200 binaries. Numbers for ByteWeight are taken from Bao et al. [2].

models separately for each dataset: one to find function starts, and the other to find function ends. We combine the predictions from each model using a simple heuristic:

- If we predict multiple function ends in sequence after a function start, ignore all but the last.
- If we predict multiple function starts in sequence before a function end, ignore all but the first.
- Otherwise, pair adjacent function starts and ends into a function boundary.

Except on the ELF x86-64 dataset, this allows us to obtain 97-98% in F1 score. In contrast, Bao et al. report 92-93%.

To obtain these results, we used bidirectional models with RNN hidden units (i.e., rather than GRU or LSTM) and one hidden layer of size 16. We trained each model on 100,000 randomly-extracted 1000-byte chunks from the corresponding binaries (or 100 megabytes in total). To clarify, this means that we run two separate recurrent neural networks forward and backward on a 1000-byte sequence from the binary. The forward and backward RNN each computes a $\mathbb{R}^{16}$ hidden representation, which are concatenated together and fed into a linear transformation and the softmax function, producing a probability distribution (a $\mathbb{R}^2$ vector) over whether that byte corresponds to the beginning (or end) of a function or not.

We used *rmsprop* with a step size of 0.1, which was scaled by the inverse square root of the current number of iterations. We used a batch size of 32, which means that in each iteration, we computed the gradients for 32 of the 1000-byte chunks, averaged them together, and applied them to the current weights (per the description in Section 4.3).

These *hyper-parameters* (like the step size, the batch size, and the output size of the RNN), unlike the weights in the neural network, cannot be trained using gradient descent. They have to be selected manually or through an exhaustive search. We selected ours informed by some smaller-scale experiments and our intuition.

5.4 Computation time

Training. In many other applications, neural networks have gained a reputation as requiring a lot of computational power to train. Indeed, it has become standard to train large neural networks using GPUs (graphical processing units), which excel at the large number of linear algebra operations that training a neural network requires. However, the networks we use are relatively small, so training with the CPU seemed to work fine.

In addition, determining the precise number of iterations to train a neural network remains more of an art than a science. Due to noisy gradients in stochastic gradient descent, and the non-convex objective, the accuracy of the neural network does not improve monotonically or predictably as the number of iterations increases. In fact, training for too long can cause the parameters to overfit the training data, and worsen the performance on the test data.

To avoid the issue, we trained our neural networks for the fixed time of two hours each, and report the performance after that. We set the duration of training by

	ELF x86	ELF x86-64	PE x86	PE x86-64
Start	98.95%	98.02%	95.56%	98.37%
End	97.83%	95.51%	94.99%	98.06%
Boundary	95.89%	92.67%	92.45%	95.91%

Table 7: Results with when trained on 10% of the data (F1 scores).

(wall-clock) elapsed time rather than the number of iterations or parameter updates, to avoid biasing in favor of more complicated but powerful architectures which might make more progress per iteration but also take longer to compute each one.

Producing the results in Tables 2 and 3 requires training 40 models, since there are 4 ISA/OS combinations and we used 10-fold cross-validation. Therefore, in total, 80 compute-hours were required. In contrast, Bao et al. [2] report that ByteWeight took 586.44 compute hours to train, over 7× longer.

Furthermore, we needed to train 40 models only for the purposes of matching the 10-fold cross-validation protocol used by Bao et al. We really only need four models to achieve equivalent results, which would take just 8 hours to produce. While abandoning cross-validation for training ByteWeight would presumably also save a significant amount of time, we can expect the factor to be less than 10 since much of the computation (extracting counts of short instruction sequences) occurs before splitting data for cross-validation, further widening the gap between ByteWeight and our method.

Testing. Table 4 summarizes the amount of time needed to run each method on the data set after training completes. Our method is hundreds of times faster than the equivalent complete version ByteWeight which computes function boundaries instead of just function ends.

The disparity mainly arises as our method works without conventional program analysis techniques, such as the static control-flow graph generation used by ByteWeight. We trained the neural network to directly identify both function start and ends, and combine them together using a simple algorithm to recover plausible function boundaries. In addition, the neural network operates directly on bytes rather than instructions, avoiding the need for a disassembly step. In contrast, ByteWeight computes a CFG starting from each identified function start both to identify more functions, and to compute the function boundary. These extra steps require a considerable amount of computation time, and yet our approach gives better results without them.

5.5 Experiments

In this section, we describe some smaller-scale experiments we performed in order to gain insight into how various choices we made in designing our method affects the accuracy of results. We trained each model for two hours, and we did not use cross-validation for these experiments to save on computation time.

Reducing training data. In Table 7, we describe results from training a randomly-selected 10% of the binaries in the dataset and testing on the remainder (normally, the fractions were switched), to simulate common applications of binary analysis where only a small amount of representative training data is available. Despite this, the results dropped by 2-3 points at most.

Unidirectional RNNs. For our main model, we used bidirectional RNNs where the output at each position depends on both previous and future inputs. In Table 5, we compare how bidirectional RNNs fare against the simpler unidirectional ones. As we might expect, the unidirectional RNNs do significantly worse than the bidirectional ones on every benchmark.

In Section 4.4, we speculated that on unidirectional RNNs, reversing the order of the input might provide better results if the bytes which come after, instead of before, a certain location in the binary provide more information about whether that location is the start or end of a function. We found that reversed inputs help with identifying ends of functions, and ordinary inputs with identifying starts. One reason for this may be that with optimization turned on, the compiler will insert no-ops between functions so that function starts occur at an aligned offset; the model can identify these to help find the start or the end.

Variations in model architecture. Table 5 also compares how Gated Recurrent Unit (GRU) and Long Short-term Memory (LSTM) fare against conventional RNNs. As we might expect, GRU and LSTM perform better than RNN in most of the benchmarks. The comparison between GRU and LSTM is more mixed. Since LSTMs take more time to run each iteration, and we trained for a fixed amount of computation time, they may not have converged as much to optimal parameters. Also, while GRU and LSTM are more powerful models than conventional RNNs, this was not enough to beat the bidirectional RNN.

We can also examine what happens when we vary the number of hidden layers or the dimensionality of the hidden layer. Table 6 shows the different results obtained using one hidden layer of size 8, two hidden layers of size 8, or one hidden layer of size 16. The larger models

	Function start identification				Function end identification			
	ELF x86	ELF x86-64	PE x86	PE x86-64	ELF x86	ELF x86-64	PE x86	PE x86-64
RNN	92.36%	86.51%	94.48%	97.07%	54.52%	61.20%	72.32%	77.34%
GRU	95.09%	92.64%	96.46%	98.26%	70.55%	72.21%	83.78%	85.95%
LSTM	94.32%	89.89%	95.72%	97.58%	70.69%	68.21%	79.58%	82.46%
RNN (rev.)	94.74%	76.05%	66.02%	83.47%	91.12%	84.91%	95.52%	95.68%
GRU (rev.)	95.92%	84.93%	78.97%	87.52%	95.33%	89.44%	96.77%	95.86%
LSTM (rev.)	94.18%	94.18%	72.48%	83.43%	94.84%	87.78%	97.09%	95.42%
Bidir. RNN	98.88%	96.06%	98.04%	99.42%	95.93%	92.94%	97.98%	99.25%

Table 5: Comparison of unidirectional RNNs with different hidden unit types and input directionality, on the function start and end identification problems. "(rev.)" indicates that we trained and tested the model with bytes in the binary reversed. All models (including the bidirectional RNN) had one layer and 8 hidden units. All percentages are F1 scores.

	Function start identification				Function end identification			
	ELF x86	ELF x86-64	PE x86	PE x86-64	ELF x86	ELF x86-64	PE x86	PE x86-64
Separate								
$h = 8, l = 1$	98.88%	96.07%	98.04%	99.42%	95.93%	92.94%	97.98%	99.25%
$h = 8, l = 2$	99.03%	97.69%	98.00%	99.43%	97.71%	94.49%	98.30%	99.19%
$h = 16, l = 1$	99.24%	98.13%	98.33%	99.50%	98.09%	95.74%	98.56%	99.24%
Shared								
$h = 8, l = 1$	97.79%	95.28%	97.30%	99.23%	95.86%	91.94%	97.08%	98.90%
$h = 8, l = 2$	98.60%	96.67%	97.96%	99.45%	97.41%	94.92%	97.58%	99.12%
$h = 16, l = 1$	98.29%	97.41%	98.42%	99.47%	97.20%	95.51%	98.32%	99.38%

Table 6: Comparison of bidirectional RNNs on the function start and end identification problems. Separate means two models were trained separately for predicting starts and ends; shared means one model does both. h is the size of the hidden layer and l is the number of layers. All percentages are F1 scores.

perform better, but it turns out that increasing the hidden layer size rather than the number of layers provides a slightly greater benefit.

Task sharing. In our prior experiments, we trained two separate neural networks for performing function start and end identification. However, we could instead train one model to recognize both; at each byte, the model would decide among four possibilities instead of two. This could halve the amount of training time required. Also, what the network needs to learn in order to recognize function starts probably overlaps considerably with learning to recognizing function ends, so a network which simultaneously performs both tasks may also learn faster and produce more accurate results.

Table 6 summarizes our experimental results for testing this hypothesis. Overall, the single model which performs both tasks seems to do slightly worse than having separate neural networks for each task. Perhaps the disadvantage incurred from needing to keep track of more information exceeds the advantages mentioned in the previous paragraph.

6 Discussion

Limitations. As with most other machine learning approaches, ours assumes that the same underlying generative process has created both the training set and the test set. If similar patterns from the training data do not exhibit themselves in the test data, our approach will fail to correctly identify the functions.

As a pathological case, consider what would happen if long sequences of instructions which have no effect were inserted at arbitrary locations in the binary, including in the middle of function prologues. Such insertions would cause the internal structure of the binary to differ from what the model saw in the training data, even though it has no affect on the functionality. We might easily remove these instruction sequences if they were simply NOPs (0x90 in x86), but we can imagine the ability to create arbitrarily complicated ones especially if they are allowed to be long. Results from computability theory, such as Rice's theorem, suggest that it could be very difficult (if not impossible) to filter out such sequences from the binary through static analysis.

	ELF x86						ELF x86-64					
	gcc			icc			gcc			icc		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1
O0	99.89%	99.95%	99.92%	99.85%	99.94%	99.90%	99.72%	99.56%	99.64%	99.77%	99.51%	99.64%
O1	99.37%	98.29%	98.82%	99.62%	98.22%	98.91%	98.87%	96.80%	97.83%	99.35%	96.97%	98.15%
O2	99.20%	98.45%	98.82%	99.57%	99.28%	99.43%	97.18%	96.58%	96.88%	98.93%	97.76%	98.34%
O3	99.28%	98.77%	99.02%	99.50%	99.31%	99.40%	96.83%	96.69%	96.76%	98.99%	97.66%	98.33%

	PE x86			PE x86-64		
	P	R	F1	P	R	F1
Od	98.96%	99.43%	99.19%	99.52%	99.39%	99.45%
O1	98.89%	97.21%	98.04%	99.48%	98.68%	99.08%
O2	99.05%	98.60%	98.82%	99.50%	99.14%	99.32%
Ox	99.16%	98.63%	98.90%	99.61%	99.14%	99.37%

Table 8: Performance of our function start identification model on different subsets of the dataset. Each percentage value represents the precision, recall, or F1 score on the binaries of a particular architecture, compiler, and optimization level combination.

As for RNNs, since they accumulate and transfer information in a sequential manner, the input from these irrelevant instructions could easily overwrite the parts of the hidden state necessary for making correct predictions about the locations of the function boundaries.

In some cases, we can foresee that our approach will require preprocessing of the data in order to obtain good results. For example, binaries which decompress or decrypt themselves at runtime would not contain recognizable code within the binary stored on disk. Given that such obfuscations affect all static binary analysis techniques, previous works have addressed the problem of detecting and reversing such transformations [10, 21].

Segmented results. In Table 8, we delineate how the accuracy results for function start identification with our model (as described in Section 5.3) differed among different subsets of the binaries as further segmented by compiler and optimization level.

As we might expect, the model does best when run on binaries compiled without any optimizations (labeled as O0 and Od in the table), given that those tend to have very clear indications at the beginnings of functions. Nevertheless, the model's performance remains roughly constant on binaries compiled with more optimizations, with the exception of `gcc` on Linux for the x86-64 architecture where the F1 score decreased by about 2.9 percentage points. Given that the training data contains examples with every optimization level and compiler used for testing, we would hope that the model can learn to recognize functions in all such cases. However, it seems that `gcc` can produce relatively challenging examples with more optimizations enabled. Since the x86-64 ABI passes some function arguments in registers, it is possible to avoid any manipulation of the stack and base pointers upon function entry.

Error analysis. We randomly sampled some of the binaries to manually inspect the errors made by the model in them. Specifically, we selected 5 binaries for each combination of compiler, optimization level, architecture, and OS, then examined the errors to identify some common features between them.

Here are some observations we made:

- Given the bidirectionality of the model, it seems to exploit the appearance of frequently-occurring sequences at the ends of the previous function in addition to typical function prologues. One obvious example are `ret` and its variants, used to return from function execution. The compiler also often inserted padding between functions (such as `nop` (`0x90`) and other no-op instructions with longer encodings, or in Windows binaries, `int3` which triggers an interrupt), the end of which the model would use to recognize the beginnings of functions.
- As a consequence of the above, false positives often occurred after `nop`, `ret`, and other instructions which usually appear at the end of a function. In fact, it would also find false positives within immediate values encoded into the code if they contained `0x90` or `0xc3`, the encodings of those instructions.
- False negatives often occurred when instructions that would typically occur in the middle of functions occurred at the beginning of a function, as we might expect. The first byte of the program was often also falsely not recognized as a function start, presumably due to the lack of context previous to it.

- As documented by Bao et al. [2], `icc` will generate functions with multiple entry points. Many of the false negatives occurred at the second entry points to functions, given that the instructions before it are not the ones which usually end functions.

- The behavior of the model was not easily characterized by simple rules on short sequences of instructions; for example, while many false positives occurred after `nop` and `ret`, this did not mean that the model marked all (or even a large fraction) of such positions as function starts. For relatively difficult cases like these, the precise content of the surrounding bytes might have a complicated effect on the answer produced by the model.

Future work. Although we have seen some experimental evidence about the performance of the RNN under various conditions, we lack a clear explanation of the internal mechanics of the model. One potential approach towards an explanation proceeds through an analysis of the eigenvector structure by linearizing the state of the network as it evolves over time and analyzing which eigenvectors of the linearized systems carry the task-relevant information [12]. This analysis can provide an understanding of how the network ignores irrelevant information while selecting, integrating, and communicating relevant information, and allows identification of which eigenvector(s) of the linearized system are responsible for these tasks performed by the network. However, if the neural network's parameters are available to adversaries interested in disrupting the accuracy of the model, they may be able to use such analyses to more effectively add extra instructions which are not orthogonal to the eigenvectors carrying the task-relevant information, thus preventing its transmission and significantly affecting the RNN's performance.

7 Related Work

Function identification. Given that function identification serves as a basis for many applications within binary analysis, it should not surprise that many past papers have discussed the topic. For example, Kruegel et al. [9] identify functions as a prelude to static disassembly, and Theiling [18] for inferring control-flow graphs. However, these do not focus specifically on function identification as a specific problem, so here we point out some other works that do.

Rosenblum et al. [14] first framed the function identification problem as a task for machine learning. They combine a logistic regression classifier that uses "idioms" (short patterns of instructions) with a conditional random field to impose some structure between predictions for related instructions. Karampatziakis [8] tackles the related problem of accurate static disassembly using similar machine-learning tools, and Jacobson et al. [7] extend the prior work by Rosenblum et al. to fingerprinting library wrappers which appear in binaries. Bao et al. [2] also address function identification using supervised learning, but use weighted prefix trees which require much less computation than Rosenblum et al.'s approach to train, but still seems to give results with high accuracy.

Some tools built for binary analysis provide function identification as part of their functionality, usually using relatively simple heuristics or hand-coded signatures: Dyninst [6] and IDA Pro are some examples.

Neural networks. Much research using neural networks have focused on domains with continuous input data, such as vision and speech. In contrast, binary code contains discrete, multinomial values, where there typically exists no obvious ordering relationship between the possible values (unlike intensities of light or sound, for example).

Natural language processing also involves multinomial values (typically sequences of words), and neural networks have been successfully used for some applications there. Bengio et al. [3] first used neural networks to make a *language model*. Language models give a probability distribution over the next word in a sentence given the words so far, and see usage in machine translation and speech recognition. Mikolov et al. [11] moved to using a RNN. More recently, Sutskever et al. [17], Bahdanau et al. [1], and Cho et al. [5] have used RNNs for machine translation, and Vinyals et al. [20] for parsing.

We could not find any previous works which applied neural networks to binary code, but some use them on source code. Zaremba and Sutskever [22] attempt to train recurrent neural networks to evaluate short Python programs. Mou et al. [13] learn a vector representation from ASTs for supervised classification of programs.

8 Conclusion

In this paper, we proposed a new machine-learning-based approach for function identification in binary code based on recurrent neural networks. To our knowledge, there exists no previous works which apply neural networks to any problems in binary analysis. We address this gap by demonstrating how to use recurrent neural networks for function identification, and empirically show drastic reductions in computation time despite achieving comparable or better accuracy on a prior test suite. We hope

that this work can serve as an inspiration for further advancements in binary analysis through neural networks.

Acknowledgements

We would like to thank Philipp Moritz for help with the initial implementation and experiments. We acknowledge Kevin Chen, Warren He, and the anonymous reviewers for their helpful feedback. This work was supported in part by FORCES (Foundations Of Resilient CybEr-Physical Systems), which receives support from the National Science Foundation (NSF award numbers CNS-1238959, CNS-1238962, CNS-1239054, CNS-1239166); by the National Science Foundation under award CCF-0424422; and by DARPA under award HR0011-12-2-005.

References

[1] BAHDANAU, D., CHO, K., AND BENGIO, Y. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[2] BAO, T., BURKET, J., WOO, M., TURNER, R., AND BRUMLEY, D. Byteweight: Learning to recognize functions in binary code. In *23rd USENIX Security Symposium (USENIX Security 14)* (San Diego, CA, Aug. 2014), USENIX Association, pp. 845–860.

[3] BENGIO, Y., DUCHARME, R., VINCENT, P., AND JANVIN, C. A neural probabilistic language model. *The Journal of Machine Learning Research 3* (2003), 1137–1155.

[4] BERGSTRA, J., BREULEUX, O., BASTIEN, F., LAMBLIN, P., PASCANU, R., DESJARDINS, G., TURIAN, J., WARDE-FARLEY, D., AND BENGIO, Y. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)* (June 2010). Oral Presentation.

[5] CHO, K., VAN MERRIENBOER, B., GULCEHRE, C., BOUGARES, F., SCHWENK, H., AND BENGIO, Y. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).

[6] HARRIS, L. C., AND MILLER, B. P. Practical analysis of stripped binary code. *ACM SIGARCH Computer Architecture News 33*, 5 (2005), 63–68.

[7] JACOBSON, E. R., ROSENBLUM, N., AND MILLER, B. P. Labeling library functions in stripped binaries. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools* (2011), ACM, pp. 1–8.

[8] KARAMPATZIAKIS, N. Static analysis of binary executables using structural svms. In *Advances in Neural Information Processing Systems* (2010), pp. 1063–1071.

[9] KRUEGEL, C., ROBERTSON, W., VALEUR, F., AND VIGNA, G. Static disassembly of obfuscated binaries. In *USENIX security Symposium* (2004), vol. 13, pp. 18–18.

[10] MARTIGNONI, L., CHRISTODORESCU, M., AND JHA, S. Omniunpack: Fast, generic, and safe unpacking of malware. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual* (2007), IEEE, pp. 431–441.

[11] MIKOLOV, T., KARAFIÁT, M., BURGET, L., CERNOCKỲ, J., AND KHUDANPUR, S. Recurrent neural network based language model. In *INTERSPEECH 2010, 11th Annual Conference of the International Speech Communication Association, Makuhari, Chiba, Japan, September 26-30, 2010* (2010), pp. 1045–1048.

[12] MOAZZEZI, R. *Change-based population coding*. PhD thesis, UCL (University College London), 2011.

[13] MOU, L., LI, G., LIU, Y., PENG, H., JIN, Z., XU, Y., AND ZHANG, L. Building program vector representations for deep learning. *arXiv preprint arXiv:1409.3358* (2014).

[14] ROSENBLUM, N. E., ZHU, X., MILLER, B. P., AND HUNT, K. Learning to analyze binary computer code. In *AAAI* (2008), pp. 798–804.

[15] SCHMIDHUBER, J. Long short-term memory: Tutorial on lstm recurrent networks. `http://people.idsia.ch/~juergen/lstm/sld004.htm`, 2003.

[16] SUTSKEVER, I. *Training recurrent neural networks*. PhD thesis, University of Toronto, 2013.

[17] SUTSKEVER, I., VINYALS, O., AND LE, Q. V. Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems* (2014), pp. 3104–3112.

[18] THEILING, H. Extracting safe and precise control flow from binaries. In *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on* (2000), IEEE, pp. 23–30.

[19] TIELEMAN, T., AND HINTON, G. Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

[20] VINYALS, O., KAISER, L., KOO, T., PETROV, S., SUTSKEVER, I., AND HINTON, G. Grammar as a foreign language. *arXiv preprint arXiv:1412.7449* (2014).

[21] YAN, W., ZHANG, Z., AND ANSARI, N. Revealing packed malware. *Security & Privacy, IEEE 6*, 5 (2008), 65–69.

[22] ZAREMBA, W., AND SUTSKEVER, I. Learning to execute. *arXiv preprint arXiv:1410.4615* (2014).

[23] ZAREMBA, W., SUTSKEVER, I., AND VINYALS, O. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329* (2014).

A Backpropagation

We can view backpropagation as repeated application of the chain rule. We sketch how it works using the following example of a three-layer network:

$$\begin{aligned} h_1 &= f_1(x;\theta_1) \\ h_2 &= f_2(h_1;\theta_2) \\ \hat{y} &= f_3(h_2;\theta_3) \end{aligned}$$

where $\theta_i = (W_i, b_i)$, and we have named all of the intermediate hidden values for convenience of reference. We wish to minimize the error between the predicted $\hat{y}$ and the true value y. For example:

$$L = d(y,\hat{y}) = \|\hat{y} - y\|^2$$

Then we can compute the following partial derivatives using the chain rule:

$$\begin{aligned} \frac{\partial L}{\partial \hat{y}} &= 2(\hat{y}-y) & \frac{\partial L}{\partial \theta_3} &= \frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial \theta_3} \\ \frac{\partial L}{\partial h_2} &= \frac{\partial L}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial h_2} & \frac{\partial L}{\partial \theta_2} &= \frac{\partial L}{\partial h_2}\frac{\partial h_2}{\partial \theta_2} \\ \frac{\partial L}{\partial h_1} &= \frac{\partial L}{\partial h_2}\frac{\partial h_2}{\partial h_1} & \frac{\partial L}{\partial \theta_1} &= \frac{\partial L}{\partial h_1}\frac{\partial h_1}{\partial \theta_1} \end{aligned}$$