Code Deobfuscation using Neural Program Learning Models

AMIN FALLAHI, Syracuse University

Code deobfuscation has gained attention in research for its wide applicability in forgery detection, privacy, and security. Several approaches have been proposed in past research that focus on identifier name recovery from a minified code or extracting information from obfuscated code, most of which use an statistical approach to deobfuscate code and achieve a significant performance. In this research, we study neural networks, specifically neural program learning models on this task. We focus on Neural Turing Machine as a model that has been proven to work well on remembering information using its memory. We build the network using NTM, prepare an existing data set, and evaluate the network with various parameters against the data set. Our evaluation results indicate that we get an average accuracy of around 60% for most settings but under limited scenarios. However, there are possible improvements to this work that we leave for future research.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

Additional Key Words and Phrases: datasets, neural networks, gaze detection, text tagging

1 INTRODUCTION

One of the trending methods for improving software security and hiding program codes from users is code obfuscation. Some codes are meant to run on the browsers and are widely used to create dynamic content in web-pages. Many service providers try to hide the code from the user to improve their security and obscure the real function of code. Using a simple obfuscation technique, the providers make their program secure against attackers and are able to inject malicious code that can help advertisement, tracking the user, collecting user information, cross-site scripting attacks, etc. Obfuscation targets changing the code in a way that makes it hard for the human to understand what the code is actually doing by methods like changing variable/function names, altering execution order, loop unrolling, character encoding, etc. In contrast, code deobfuscation methods try to gain as much information from obfuscated codes and extract the program from unreadable code.

Using a Neural Program Learning (NPL) model for solving this problem and how it performs can be an interesting study. NPL models use neural networks which are trained to learn and replicate code behavior, produce code from input/outputs, or complete previously written code. Code deobfuscation is an application that can be extended to a program learning task. Specifically, we can treat obfuscated identifier names as unknown holes in a program code and try to recover them using an NPL algorithm.

In this research, we use neural program learning methods to recover identifier (variables and functions) names from obfuscated program code. We study some currently used obfuscation and deobfuscation methods and how much identifier names the current approaches to deobfuscation can recover. Then, we use Neural Turing Machine (NTM) [6] as our program learning model and train it using publicly available data sets. Finally, we evaluate our work and try to optimize the network by altering and tuning parameters to improve the performance.

The main contributions of this work are:

- Application of neural program learning models on code deobfuscation task.

Author's address: Amin Fallahi, Syracuse University, Center for Science and Technology, 111 College Pl, Syracuse, New York.

- Creating a neural activation dataset usable in neural networks based on publicly available program code datasets.
- Evaluating and comparing neural program learning models against traditional models and previous research for this task.
- Extending an existing Neural Turing Machine implementation to work with our model[1].

Outline In the remainder of this report, we introduce some of the related work that have been done in this research area in section 2. Then, we discuss basic background knowledge and area literature required for studying this work in section 3. In Section 4 we introduce the dataset we are using for this study and the preparation methods we use for making the data fit our model. Afterwards, we review the implementation details for Neural Turing Machine, our tweaks, and modifications to it for using it for our application. We run our implementation through multiple experiments and present the results in section 6 and then we discuss the results and outcome of the research in section 7. In section 8 we discuss some possible extensions to this work which is left for future research and finally, we conclude the study in section 9.

2 RELATED WORK

Research on code deobfuscation has been a topic of interest during the recent years. Jiang et al. [8] have proposed a method using Convolutional Neural Networks for detecting obfuscated JavaScript code. Raychev et al. [10] have developed a statistical model for recovering program properties (variable and function names) from obfuscated JavaScript code and have implemented a currently available website named JSNice [2] which can predict the correct names for 63% of identifiers from obfuscated code for their test data. They also have designed Nice2Predict [3] which extracts statistical info out of obfuscated JavaScript code and is a base for JSNice. Raychev et al. [11] also have done research on code completion and program synthesis which is relevant to our problem. Bichsel et. al. [4] have developed and tested a statistical method for deobfuscating Android applications with 79.1% success in recovering program identifier names. Another relevant research is [5] which studies filling program missing code slots using the behavior trained from the program input/output data. Lin et al. [9] have done relevant research on translating natural language templates to program templates using Recurrent Neural Networks.

3 BACKGROUND

3.1 Neural Program Learning

Neural program learning is the use of program behavior to train deep neural networks so they perform like the program or produce the code based on input/outputs, partial programs, etc. So, a neural network can be used as a programmer and also an interpreter. The research in this area has been developed by several researchers during the past decade, varying from implementing logical gates using neural networks [**?**] to completing a partial program code [5].

3.2 Code Deobfuscation

Obfuscation is a method used by programmers, specially in web services, to hide the running code from the user. When users visit a website using a browser with JavaScript enabled, several scripts will be executed in their browsers. The service providers, usually make the code unreadable by changing identifier names to meaningless tokens and then remove white

[1] https://github.com/aminfallahi/Code-Deobfuscation-using-NTM

space and tabs to decrease code size. Shrinking code size leads to less data transfer and faster page loading but changing identifier names is meant to hide the actual function of code from the user. Websites practice advertising, tracking, and other acts that are privacy concerns using these codes and hide the actual operation of the code by obfuscating it.

Several methods have been proposed for deobfuscation. Basic methods recover tabs and spaces to make the code more readable but are not capable of recovering the actual code including identifier names. Some methods use machine learning and statistical approaches to recover identifier names. Also, some approaches target deciphering code that has been encrypted using cryptography algorithms.

Listing 1. shows a piece of obfuscated JavaScript code grabbed from Facebook.com. This code is unminified using an online tool, Unminify [?], which inserts tabs and spaces for it to look more readable, but the identifier names will remain obfuscated. As we can see, most of them have been changed to single letter names, making it hard for anyone to understand what the code is actually doing.

Listing 2. shows the same code from Listing 1. deobfuscated using JSNice [?] tool which uses statistical methods to recover identifier names. As we can see, some of the names have been recovered and the code is more understandable. However, we can not be sure how accurate are the names that are assigned to obfuscated identifiers.

Listing 1. An obfuscated JavaScript code from facebook.com

```
__d("ContextualLayerAlignmentEnum", ["prop-types"], (function(a, b, c, d, e, f) {
  "use strict";
  a = {
    left: "left",
    center: "center",
    right: "right",
    propType: b("prop-types").oneOf(["left", "center", "right"]),
    values: ["left", "center", "right"]
  };
  e.exports = a
}), null);
```

Listing 2. Deobfuscated JavaScript code from Listing 1. using JsNice[?] tool

```
’use strict’;
__d("ContextualLayerAlignmentEnum", ["prop-types"], function(h, require,
    canCreateDiscussions, isSlidingUp, module, dontForceConstraints) {
  h = {
    left : "left",
    center : "center",
    right : "right",
    propType : require("prop-types").oneOf(["left", "center", "right"]),
    values : ["left", "center", "right"]
  };
  /** @type {(Object|string)} */
  module.exports = h;
}, null);
```

3.3 Neural Turing Machine

The idea behind Neural Turing Machines is to use an external memory to extend the capabilities of recurrent neural networks. The external memory and the network can then interact with attentional processes [].

As shown in Fig. 1, NTM is composed of two main components: the network controller and a memory bank which is basically a matrix used to store and retrieve information. In each time step, the network gets an input and produces an output. But NTM is also capable to perform read and write operations and alter the memory matrix. Like traditional Turing machines, NTM uses the term "head" to refer to a specific memory location [].

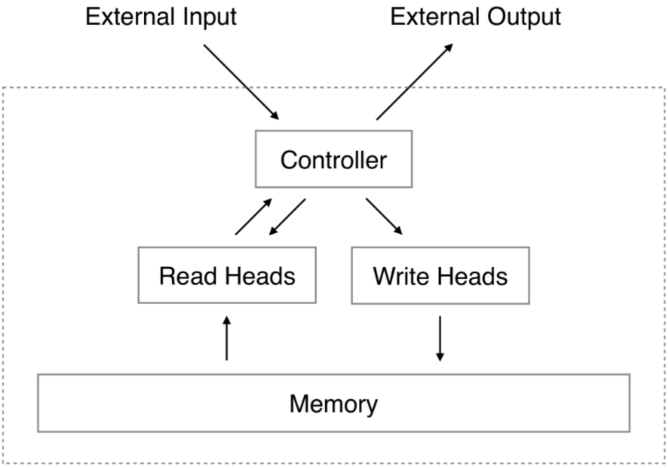


Fig. 1. Neural Turing Machine architecture.

For network training, it is preferred to use backpropagation and an optimizer like stochastic gradient decent (SGD) or Adam. For this purpose, NTM controller uses "blurry" reads and writes to interact with the memory, leading to the network being differentiable. That means, the controller uses a greater or lesser degree to read the whole memory instead of addressing and accessing each single element in memory. This is possible using an attention vector to focus on a part of memory and ignore the rest.

3.3.1 Memory. A memory M_t is a $N \times M$ two dimensional matrix at time t, where N indicates the number of memory locations, each storing a vector of size M.

3.3.2 Read. NTM uses a weighted sum of the memory for reading, where w_t is a weight vector of size N at time t:

$$r_t \longleftarrow \sum_i w_t(i) M_t(i) \tag{1}$$

3.3.3 Write. A write operation is composed of an erase and an add operation. A vector e_t of size N is used for erasing and another vector a_t is used for the add operation. Let $\tilde{M_t}$ be the intermediate memory after erase operation on M_{t-1} and 1 be a vector of ones:

$$\tilde{M_t}(i) \longleftarrow M_{t-1}(i)[1 - w_t(i) e_t] \tag{2}$$

Then, if both w_t and e_t at location i are one, the memory location will reset to zero and if both are zero the memory will not be touched.

Finally, the add vector a_t is used to perform the add operation on the intermediate memory state and produce the final memory matrix:

$$M_t(i) \longleftarrow \tilde{M}_t(i) + w_t(i)a_t \tag{3}$$

3.3.4 Addressing. NTM produces the weighting vectors for read and write operations by combining content-based and location-based addressing mechanisms.

The content-based addressing compares a key vector k_t with each $M_t(i)$ vector using cosine similarity (Equation 5) measure and produces a normalized weight vector w_t^c based on the similarity and a key strength vector β_t which is used for increasing or decreasing focus precision:

$$w_t^c(i) \longleftarrow \frac{exp(\beta_t K[k_t, M_t(i)])}{\sum_j exp(\beta_t K[k_t, M_t(j)])} \tag{4}$$

$$K[u,v] = \frac{u.v}{||u||.||v||} \tag{5}$$

For location-based addressing, a scalar parameter $g_t \in (0,1)$ is introduced which is called interpolation gate and is used to mix w_t^c and w_{t-1} to produce the gated weighting w_t^g. This lets the controller know when to use which kind of memory addressing:

$$w_t^g \longleftarrow g_t w_t^c + (1 - g_t) w_{t-1} \tag{6}$$

NTM controller uses circular convolutional shift with all index arithmetic computed modulo N with a shift vector s_t to change the focus to another memory location and produce the shifted weight vector $\tilde{w}_t(i)$:

$$\tilde{w}_t(i) \longleftarrow \sum_{j=0}^{N-1} w_t^g(j) s_t(i-j) \tag{7}$$

Finally, a scalar γ is used for sharpening the final weighting and preventing it from blurring:

$$\frac{w_t(i) \longleftarrow \tilde{w}_t(i)^{\gamma_t}}{\sum_j \tilde{w}_t(j)^{\gamma_t}} \tag{8}$$

Figure 2 shows the flow of addressing mechanism.

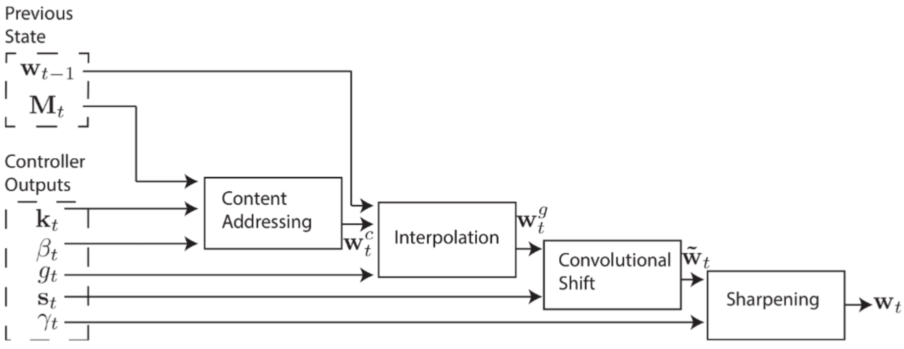


Fig. 2. Flow diagram of Neural Turing Machine addressing mechanism.

4 DATASET PREPARATION

Researchers in Secure, Reliable, and Intelligent Systems Lab [] at ETH Zurich have collected and published a data set of 150000 JavaScript programs and 150000 Python scripts as a part of their research [**?**] which is publicly available at [1]. They have also parsed the programs and generated abstract syntax trees (AST) for all the programs in json format. In this research we build our dataset on top of their raw Python code dataset. For simplicity, we filter a set of 1848 programs each with less than 20 lines of code and smaller than 1 Kilobyte and truncate codes with unicode and non-ascii characters. Then, we use a simple Python tokenizer [**?**] to extract all the tokens used in all programs. We sort the tokens and remove duplicates to create our final list containing 22973 tokens addressable with 15 binary bits. Afterwards, we create a list of vectors starting from 0b000000000000000 to 0b101100110111101 (22973) each representing one token. Finally, we use pyminify [**?**] (which is a simple tool for minifying Python codes) with –rename-globals option to rename all the variables to generate our obfuscated dataset and then we convert each program to a set of vectors based on their tokens and store the resulting dataset.

5 IMPLEMENTATION

The original authors of Neural Turing Machine have not published their source code to the public, but since 2014, multiple open source implementations have been released by various groups. Most of the implementations are limited and unstable. One of the recent implementations which is more robust and reliable is the one by Collier et al. [**?**] which has won a best paper award. Therefore, we chose to modify their code to fit our application and use it for this research.

This implementation uses LSTM (Long Short-Term Memory) [7] as the controller of the NTM and evaluates the model with three tasks: copy task that generates the same output by copying the input, repeat-copy task that extends copy task to copying input for specific number of times and produce it as output, associative recall that feeds a list of items to the network, queries one of them, and expects the next item in the list to be produced in output [].

We modify copy task for our application. However, our data inputs and outputs are different, so we are not using it for copying the input to output.

This implementation uses random sequences of vectors as the data. We implement a new data generator function to use our dataset instead of the randomly generated one. For our dataset to fit the NTM implementation we are using, we write a program to generate data batches. As an example for one configuration, each input sequence stores 128 token vectors and one vector of all ones indicating the end of sequence. The rest of sequence will be padded to 257 (2 times number of tokens minus 1 for the termination vector) vectors using vectors of all zeros. We also add one zero to all token vectors except the termination vector to differentiate them from it. So, the final shape of each input will be 257x16 where 16 is the batch size. Each output sequence stores 128 token vectors of 15 bits resulting in a shape of 128x15. The final batch that fits the NTM implementation we use is in the $[(sequence_length, array([[[]]], dtype = float32))]$ format and stores sequence length (128) and n batches of inputs and outputs.

We continue our modification by introducing perturbation functions. The goal is to add noise and shuffle data to evaluate the effect of them on the results later. We add three parameters to the program: whether to shuffle the tokens in input and accordingly in output, percent of noise to be added to the token sequences, and percent of noise to be added to each batch of inputs. For shuffling the input and output sequences, we generate a permutation of 128 integers by shuffling the range of integers between 0 to 127. Then we sort the tokens based on the permutation both in input and output. For adding noise to the input sequence, we use a shuffled range list again and choose first n integers from the shuffled list based on the percentage of noise we want to add. Then, we replace the randomly chosen token vectors with random vectors of 0 and 1. We do the same procedure for creating a random matrix of 0 and 1 and replace the indicated percent of generated input programs with random matrices.

6 EVALUATION

We evaluate our model in different scenarios, each changing one parameter of the network architecture or a parameter of input/output data. In all experiments, we use 500 training steps. However, an NTM takes many more training steps to produce noticeable results and it would take a long time for each training procedure which does not fit our limited time for evaluating the model under multiple scenarios. We use gradient decent with Adam optimizer for training our network with 1000 program codes. Then, 848 programs will be left from our dataset that we use as testing data.

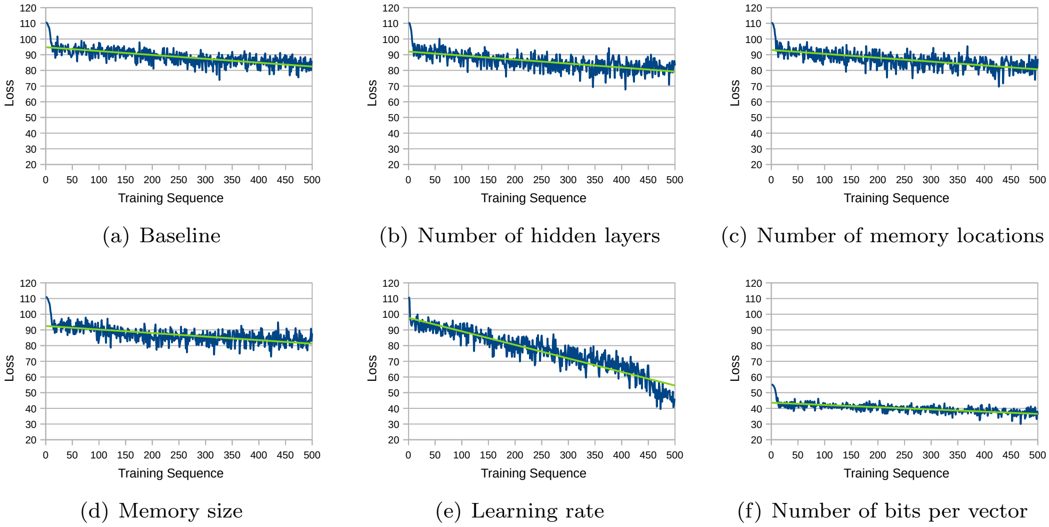
We discuss our experiments in the remainder of this section and present our results in Fig. 3 and Fig. 5. The former shows training performance for 500 sequences and the latter compares testing performance under fixed size dataset and variable sized dataset for 500 testing sequences. We aggregate our data in 5 and present the loss for each 50 sequences to get a smoother curve.

6.0.1 Baseline Variables. Our baseline variables for the network are the following. We change each parameter during each test and observe the effect on the model performance. For each experiment, we evaluate the model once for a dataset of equal size programs and once for a dataset of programs with variable number of tokens.

- Number of hidden layers: 1
- Number of memory locations (N): 64
- Memory size (M): 20
- Number of read heads: 1
- Number of write heads: 1
- Optimizer: Adam
- Learning rate: 0.001
- Batch size: 16
- Number of bits per vector: 8

6.0.2 Number of hidden layers. For this experiment, we use the baseline parameters with two hidden layers instead of one. The results show adding one hidden layer does not contribute to training performance of our model under our experiment setup. Also, the results for test performance with fixed sized and variable sized program instances, shows no improvement. However, it is observable that testing with variable sized data has a little negative effect on the performance.

6.0.3 Number of memory locations. We alternate between the baseline value of 64 and test value of 128 for this parameter. While we expect better performance with more memory locations, we do not observe noticeable performance improvement in both training and testing cases. Since we have limited our data size, it is possible that for our 500 training sequences, a larger number of memory locations does not contribute to the performance. Similar to the previous experiment, the model works a little worse with variable sized data.



(a) Baseline (b) Number of hidden layers (c) Number of memory locations

(d) Memory size (e) Learning rate (f) Number of bits per vector

Fig. 3. Learning curves for training the model under different network parameters.

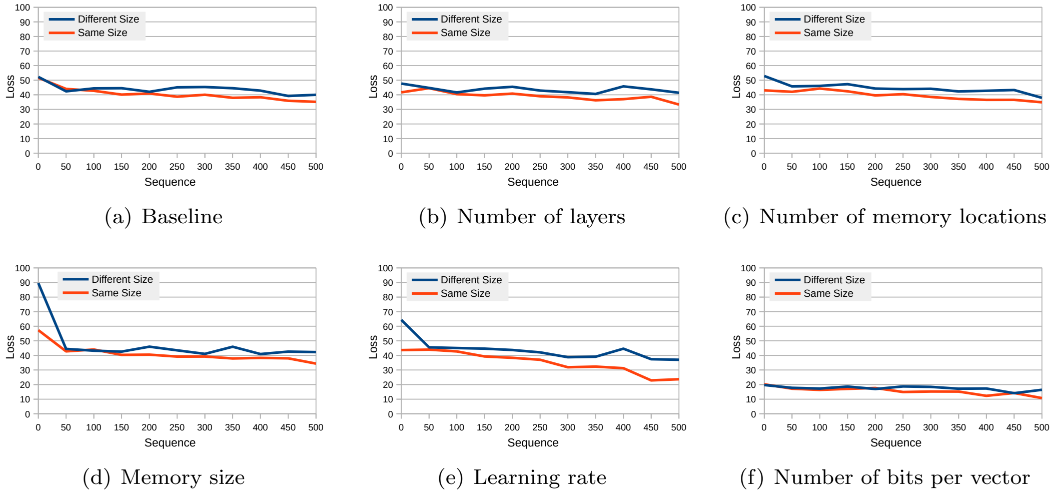
6.0.4 Memory size. For this test, we change the memory size to 10. Like altering the number of memory locations, this parameter does not alter the performance of training and testing under our experiment setup. However, testing results under fixed size dataset shows better performance.

6.0.5 Learning rate. As seen in Fig. 3, increasing the learning rate highly affects the training and testing loss. Like other experiments, testing with same sized data results in better performance.

6.0.6 Number of bits per vector. As we expect, reducing the number of bits to 4, in this scenario has a high impact on our training and testing performance. Basically, changing number of bits from 8 to 4 makes our model to support only 16 possible tokens in comparison to 256 tokens. Testing under same sized data has negligible improvement in this experiment.

6.0.7 Error. For our experiments, we generate the error by subtracting matrices of token sequences and summing the absolute value of all the numbers. Our results show, for the network with our baseline configuration which is trained for 500 sequences, we get an average error of 42.56%. So, in average, around 60% of the identifiers are correctly recovered by our model. For sequences of different sizes, the accuracy does not have a significant change. However, our setting works with a simplified dataset, where we use at most 8 bits for representing tokens which limits the number of supported tokens to 256. We also use programs with small size or shrink them to 20 tokens (sequences). Hence, we expect using a larger dataset with a network with larger parameters will produce different results.

6.0.8 Ablation study. In this series of experiments, our goal is to study the effect of perturbation on the performance of our model. We use our shuffling perturbation function with 500 training steps for each of our parameters and present the results in Fig. **??**. As we expect, shuffling token sequences has a notable effect on the model performance because identifiers



(a) Baseline (b) Number of layers (c) Number of memory locations

(d) Memory size (e) Learning rate (f) Number of bits per vector

Fig. 4. Learning curves for testing the model under different network parameters with same-sized data and different-sized data.

in each program usually appear together. That means, neighboring tokens have important effect on the identifier.

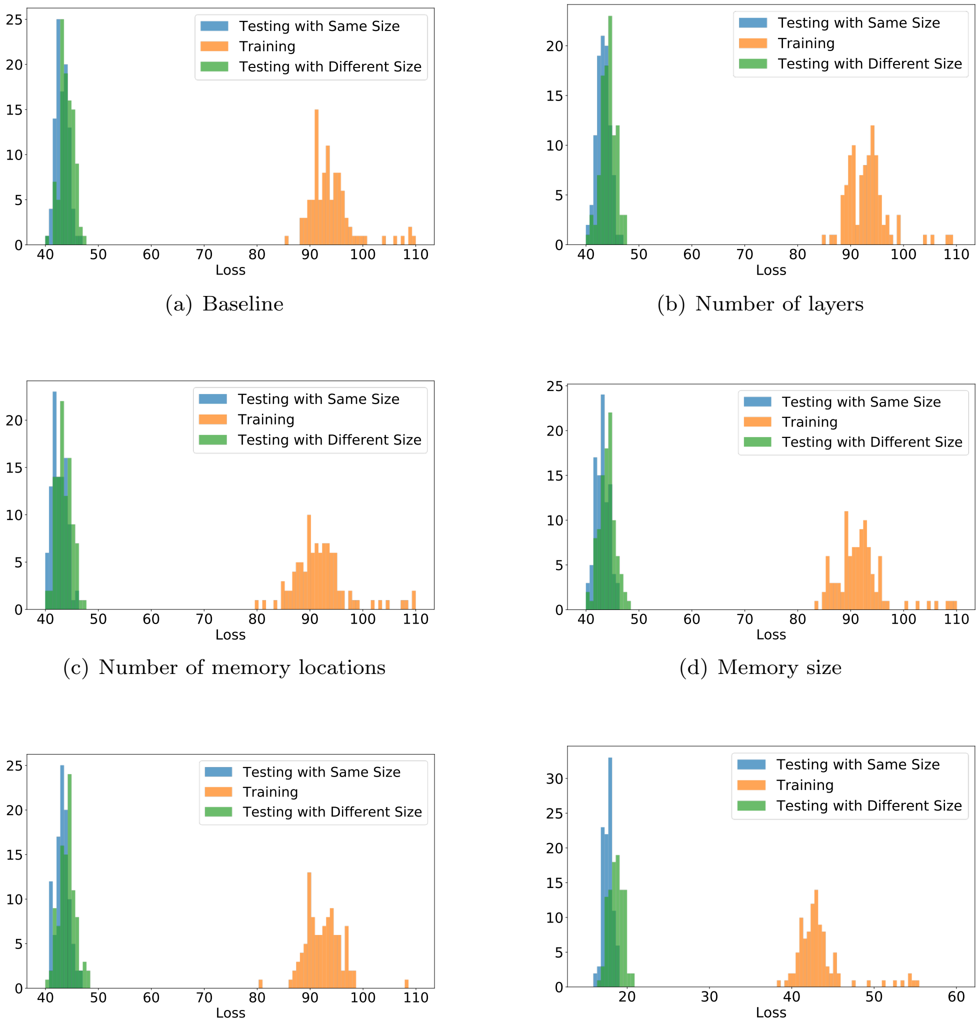
7 DISCUSSION

In this research, we studied and experimented how neural program learning methods, in our case NTM, works on code deobfuscation task. We believe code deobfuscation is an important task which extends from machine learning to cybersecurity and has attracted a lot of attention by the researchers. Thus, we prepared a dataset for the task and modified a major implementation of NTM to fit our dataset and executed multiple experiments on different kinds of data and neural network parameters. We also did a ablation study by implementing and applying a perturbation function on the data.

Our results show that NTM can reach a good accuracy under simplified conditions and certain cases. The network can reach low loss level on our tests and can recover an average of 60% of program identifiers. However, we have sacrificed some of the possible tokens and shrunk our data. Tweaking the model with multiple parameter changes may result in better accuracy, but ultimately, the network does not converge to a robust and reliable accuracy.

Related work that have been done using statistical methods and classic machine learning algorithms, shown to have reached more than 60% accuracy which overcomes our model. However, the use of NTM for this task is a novel application and there is a wide area for improvements.

Among the most important takeaways from this research, we can mention our use of neural networks ability to remember and recover information which has been done using a memory by NTM. While basic recurrent neural networks can learn from training data, using a memory is a great improvement for recovering data, specifically when the data is complex and large. The ability of NTM to recover information from large sets of training data is noticeable and has contributed to multiple neural program learning tasks including our application.



(a) Baseline

(b) Number of layers

(c) Number of memory locations

(d) Memory size

(e) Learning Rate

(f) Number of Bits per Vector

Fig. 5. Histograms for training and testing loss with perturbed tokens in the programs.

One of the notable challenges and problems in this work, as mentioned, is the reduction and simplification we applied to our dataset. Also, we used simple parameters to be able to perform multiple experiments. We reduced our training instances to 500 while a well-trained NTM needs thousands of training instances. We believe it is possible to improve results by training the network with larger memory size, sequence length, bit length, and other parameters that takes hours of heavy computation.

Our ablation studies shows perturbing the input sequences, in our case shuffling the program tokens in each program code, leads to more loss. While each token depends on the neighboring tokens, it is expected that changing the neighbors can reduce the accuracy. We expect our model to pay attention to nearby context to recover identifiers while shuffling the tokens prevents this from happening.

8 FUTURE WORK

This study is the first application of neural program learning models on program deobfuscation task to our knowledge. So, we believe our primary results under simplified scenarios can lead to further research and improved results. Tuning network parameters, specially to large values and evaluating the network with large dataset and high number of training instances can contribute to the results of the project which we left for future study due to the high computation power need and our limited time. Also, using other neural program learning models like differentiable neural computer [**?**] which is a more recent proposed architecture and have proven to work better than NTM in some tasks, can be a point for future study.

9 CONCLUSION

In this research we reviewed the literature of neural program learning research area and discussed Neural Turing Machines as one of the proposed NPL models. Then, we prepared a dataset of obfuscated and deobfuscated program codes and tweaked an existing NTM implementation to work with it. Then, we used the network with our dataset for recovering obfuscated identifiers from program codes and evaluated our model under multiple scenarios. For a simplified case, our model showed to be able to recover an average 60% of identifiers. However, to our knowledge, our application of code deobfuscation on neural program learning models is novel and there is a good space for improvements and extensions which is left for future research.

REFERENCES

[1] [n.d.]. Datasets - Learning from ”Big Code”. https://learnbigcode.github.io/datasets/

[2] [n.d.]. JS NICE: Statistical renaming, Type inference and Deobfuscation. https://jsnice.org

[3] [n.d.]. Nice2Predict. https://jsnice.org

[4] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. 2016. Statistical Deobfuscation of Android Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 343–355. https://doi.org/10.1145/2976749.2978422

[5] Matko Bošnjak, Tim Rocktäschel, Jason Naradowsky, and Sebastian Riedel. 2017. Programming with a Differentiable Forth Interpreter. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70 (ICML'17)*. JMLR.org, 547–556. http://dl.acm.org/citation.cfm?id=3305381.3305438

[6] Alex Graves, Greg Wayne, and Ivo Danihelka. 2014. Neural Turing Machines. *CoRR* abs/1410.5401 (2014). arXiv:1410.5401 http://arxiv.org/abs/1410.5401

[7] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* 9, 8 (Nov. 1997), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735

[8] Wei Jiang. 2018. Method for Detecting Javascript Code Obfuscation based on Convolutional Neural Network. *International Journal of Performability Engineering* 14 (12 2018). https://doi.org/10.23940/ijpe.18.12.p26.31673173

[9] Xi Victoria Lin, Chenglong Wang, Deric Pang, Kevin Vu, Luke Zettlemoyer, and Michael D. Ernst. 2017. *Program synthesis from natural language using recurrent neural networks*. Technical Report UW-CSE-17-03-01. University of Washington Department of Computer Science and Engineering, Seattle, WA, USA.

[10] Veselin Raychev, Martin Vechev, and Andreas Krause. 2019. Predicting Program Properties from 'Big Code'. *Commun. ACM* 62, 3 (2019), 99–107. https://doi.org/10.1145/3306204

[11] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. *SIGPLAN Not.* 49, 6 (June 2014), 419–428. https://doi.org/10.1145/2666356.2594321

10 ACKNOWLEDGMENTS

This work has been done as a part of CIS 700 - Neural Program Learning course project offered by Prof. Katz at Syracuse University.