Models of Code: Determining robustness in two settings

Bryn Reinstadler
brynr@mit.edu

Sanja Simonovikj
sanjas@mit.edu

Abstract

Distributed representations of words, sentences, and documents have been crucial to finding good ways to use neural networks for NLP tasks (word2vec, doc2vec, seq2seq). These same distributed representations of textual data also serve us well in the related domain of programming languages; programs themselves are just structured text. As programming languages and NLP communities come to see their own common interests, NLP-derived neural models are being built to work on programmatic data as well (code2vec). In this project, we want to investigate whether these neural models, which use distributed representations of lines of code or abstract syntax trees, are robust to various types of obfuscation and adversarial inputs. We find that perturbations to Java programs either by variable substitution or by deadcode insertion cause little difference in classification by code2vec, but that obfuscated PowerShell programs cause an otherwise well-performing malware classifier to perform close to chance.

I Introduction

Neural models of natural language, such as word2vec [1], ELMo [2], or BERT [3] have achieved great performance on natural language tasks of many kinds, such as text summarizing, part-of-speech tagging, and error finding. Analogously, neural models of programs have achieved great performance on tasks such as code summarizing / labelling, variable and type prediction, and bug finding.[4] Although code is not purely natural language, code is also structured linguistic input to a model and therefore can be treated in many respects the same.

In many domains, most popularly image processing, neural models have been shown to be vulnerable to adversarial attacks. In an adversarial attack, the input to the model is perturbed in some small way to get the model to produce a different, incorrect output. These attacks may be targeted, trying to get a model to produce a specific incorrect output, or non-targeted, simply trying to get a model to produce any incorrect output.

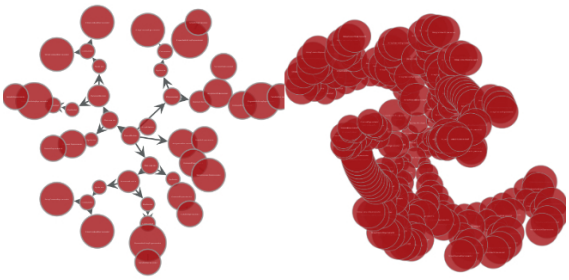

Figure 1: Abstract Syntax Trees of non-obfuscated PowerShell script (left) and the corresponding obfuscated PowerShell script (right). Obfuscation usually makes the AST larger.

Because of the discrete nature of language input, making small perturbations to a model is more difficult. In the continuous setting, such as with the pixel color values of an image, the small perturbations are relatively straightforward to implement. However, a piece of natural language or a piece of program code is a discrete object which must maintain certain semantic and syntactic relationships to remain valid. Some techniques have nevertheless been developed for making adversarial adjustments to textual data. Textual adversarial inputs to a neural model may take the form of synonymous word/variable replacements or complete obfuscation (see Figure 1), which makes code difficult for humans to understand while preserving its functionality.

In this paper, we examine the robustness of neural models of code using programs as the language input. We will be exploring various types of obfuscations of code, in two different applications.

A Application 1

Research Question 1 (RQ1): Do strategies described in the NLP literature for generating textual adversarial examples translate to good strategies for generating adversarial examples for models of code? To answer this question, we utilize the publicly-available code2vec model,[5] which is trained to pre-

dict method functionality (e.g. sort) from the contents of the method. We will use adversarial strategies such as the addition of deadcode and variable renaming using synonyms, antonyms, and random strings. We will then assess accuracy on the newly-generated datasets.

RQ2: How do the representations themselves change when the input code is obfuscated, and what can we learn about the geometric space in which the representations are embedded? We will assess the similarity of original and adversarial programs using the L2 norm, and looking at clusters of semantically related concepts using PCA.

B Application 2 - PowerShell malware detection and obfuscations

PowerShell is a command line shell, that is widely used in organizations for configuration management and task automation. Due to its popularity PowerShell is also increasingly used by cybercriminals for launching cyber attacks against organizations, mainly because it is pre-installed on Windows machines and it exposes strong functionality that may be leveraged by attackers. Traditional detection methods calculate a signature for known malware that is used to detect that attack in the future. The signature is created from static analysis of the original file and often cannot be used to detect similar scripts with the same function. The attacker can simply use obfuscation to break the signature. Obfuscation is the manipulation of a script or piece of code that changes the code signature without changing its function. This means that a defender must not only identify an existing attack and then create a signature to detect it but they must do this for all slightly modified or obfuscated attacks with the same essential function. In addition, freely available obfuscation tools such as *Invoke-Obfuscation* [6] make it easy to modify a script enough to get past traditional malicious PowerShell detection. Current machine learning exploration for detecting malicious PowerShell includes NLP and convolutional neural network methods as well as Abstract Syntax Tree (AST) based structural analysis of programs. Based on this we propose the following research questions:

1. How much do embeddings of obfuscated PowerShell scripts differ from their non-obfuscated counterparts?

2. Are malware classifiers trained on embeddings of non-obfuscated scripts robust when presented with embeddings from their obfuscated counterparts?

II Related work

A Adversarial examples for models of language and models of code

There has been much work in the past few years to build adversarial examples for both natural language models [7] [8] [9]. Prior work by Cheng et al shows that seq2seq models are generally more robust to adversarial changes than are word2vec-type models.[7] Several algorithms have been developed to produce adversarial text modifications in linear time by substituting words for synonym words; these have been tested against models such as BERT and been found to significantly disrupt classification.[9] In this work, we borrow this technique from the NLP space to query models of code.

Some work has already been done in the space of adversarial substitutions for models of code, by the developers of code2vec.[4] In their work, they present a new technique for discrete adversarial manipulation of programs (DAMP). They use the existing model weights and follow the gradients to slightly modify the code, with some clever strategies for transferring this strategy to the discrete space. Given a program P and label L by model M, DAMP aims to find a semantically equivalent program P' such that M gives adversarial prediction L'.

However, this strategy requires that the attacker be able to compute gradients in the model under attack. In this work, we use adversarial strategies inspired by NLP papers to attack models of code, treating the models as a black box rather than a white box.

B NLP approaches for Powershell

FireEye [10] discussed a natural language processing (NLP) pipeline for detecting malicious PowerShell commands by stemming. The pipeline first decodes and tokenizes the script before stemming tokens to their semantic base. The decoding step can handle remote download and executable malware created by cradle and launcher obfuscation techniques. The list of decoded, stemmed tokens is vectorized into a machine learning friendly format, for techniques such as Kernel SVM. [11] described a method for detecting malicious PowerShell commands using computer vision and NLP techniques. The computer vision techniques encode the first 1024 characters of a PowerShell command as a matrix where each row is a one hot vector with zero entries except for the code of the character at that index and apply a convolutional neural network to the matrices. The NLP techniques encode the command as a vector of length

1024 containing the code for each of the first 1024 characters and feed this vector into an recurrent neural network. Both of these techniques focus on the character content of the script which as we can be easily manipulated through obfuscation. The authors of [11] extended their work in [12] by employing pre-trained contextual embeddings of words from the PowerShell language to project semantically similar words to proximate vectors in the embedding space. The model's embedding layer was trained using a scripts dataset that was enriched by unlabeled PowerShell scripts collected from public repositories. The use of unlabeled data for the embedding significantly improved the performance of the detectors. Their best-performing model used an architecture that enabled the processing of textual signals from both the character and token levels.

III Methods

A Representations of adversarial examples using code2vec

To answer **RQ1** and **RQ2**, we downloaded the code2vec models as well as a small repository of Java code samples. The code2vec models were trained using the java-small training and validation sets, and we ourselves ran a test set of data through the pre-trained model to ensure that we were getting similar results to those published in the original paper.

Perturbations were made on the raw Java programs in that test set, using a set of bespoke bash and R scripts. In pre-processing, we made one of 7 types of perturbations:

1. Add deadcode with a targeted word ("sort")
2. Add deadcode with a random English word
3. Add deadcode with a nonsense string
4. Rename variable with a synonym
5. Rename variable with a targeted word ("sort")
6. Rename variable with a random English word
7. Rename variable with a nonsense string

Deadcode is unexecuted code; we used line comments to introduce deadcode. Variable renaming was done by scraping the Java scripts for variable names and finding synonyms using the WordNet software.[13] A list of approximately 60,000 random English words was used to generate random English words;[14] nonsense strings were generated by sampling at random from the set of all letters and numbers. In some cases, no appropriate variable names could be found once filtering out variable names from import statements, etc. Files whose contents were not changed by our pre-processing steps were omitted from further analysis to reduce false negatives.

These adversarial examples were then pre-processed using the same strategy as in the original code2vec paper, and they were run through the same pre-trained model. Resultant distributed representations of the input were analyzed using a set of bespoke R scripts.

B PowerShell code representation using Seq2Seq with attention

We use an auto-encoder sequence to sequence model with attention to obtain high-dimensional program representations of PowerShell scripts, which can be used for downstream tasks such as malware classification. The motivation to use this model is to showcase the robustness of a standard modeling approach when it comes to sequences and investigate whether there is need for more robust modeling when the sequences are code as opposed to natural language.
We use a multi-layer perceptron to train a simple model to classify the embeddings and perform malware classification on a subset of labeled data. We evaluate its robustness on both obfuscated and non-obfuscated data.
We also explore one alternative representation, such as using simple hand-crafted features from the ASTs (Abstract-Syntax Trees) of programs, as suggested by Rusak et al [15], by training a Random Forest classifier on the hand-crafted features and testing for robustness against obfuscations.

1 Dataset and pre-processing

We started with a large corpus of around 400k unlabeled PowerShell scripts collected from public sources [16]. We pre-processed the data as follows:

1. Deduplication based on md5 hash values, where one script is kept for each hash value and the rest are removed.
2. Replacing Variables with 'X' and Strings with 'Y' and removing duplicates from the resulting files.
3. Tokenization and removing scripts whose length is outside of range [10, 2000].
4. Deduplicating scripts with the same ASTs (e.g. different script content but same AST).
5. Removing scripts whose AST depth is outside of range [2, 15].

This resulted in around 100k scripts, out of which we used 10k in our seq2seq model due to constrained computational resources. We considered every three subsequent characters to constitute one token of the script, including punctuation and whitespace (large whitespace was truncated to one). We lowercase all tokens as PowerShell is case-insensitive. Scripts were truncated to 1000 tokens. The vocabulary size was 1101 and it consisted of all tokens with corpus frequency of at least 500. The train/val/test split was 70/10/20.

For the malware classification task we used 1012 scripts chosen from the preprocessed 100k dataset. Half of them are malware, half are benign. The labels were obtained by sending API requests to VirusTotal [17], an online tool that aggregates results from several malware detectors. As long as the script was labeled as malware from at least one tool, we treated the script as malicious. For this task the train/test split was 70/30.

2 Obfuscations

We consider two types of obfuscations, that is `AST` and `TOKEN` obfuscation from the `Invoke-Obfuscation` tool [6]. AST obfuscations act on the AST of the scripts, by adding, removing or shuffling AST nodes in a way that preserves functionality, while TOKEN obfuscations act on the individual tokens and include changes such as adding ticks, changing the case, reordering, removal, concatenation etc, again in a way that preserves functionality. There are 10 AST and 18 TOKEN obfuscation types, resulting in 2^{28} possible obfuscation configurations (28-dim binary obfuscation vector indicating whether a particular obfuscation was applied or not). For a given script, we apply each obfuscation with probability 0.5. To visualize the effect of obfuscations on the AST, we show an example in Figure 1.

3 Seq2Seq Model and Training

We reused the majority of the Seq2Seq with attention code publicly available on GitHub [18], and we adapted it for our own dataset and computational resources. We set the hidden dimension for both the encoder and the decoder was 32, which is the dimension of our final program embeddings as well. We use GRU for the recurrent units and dropout on the token embeddings with probability 0.5. The attention mechanism used is the one proposed by Bahdanau 2014 [19].

The model was trained on a Google Cloud instance with GPU support. We trained our model for around 70 epochs and used the checkpoint with lowest validation loss for evaluation. We used batch size of 2 (due to computational constraints). We used cross-entropy loss and Adam optimizer with learning rate of 0.001. We used gradient clipping with threshold 1.

4 Malware Classifier

For the malware classification task we constructed a classifier with an input layer with dimensions (32, 128), one hidden layer with dimensions (128,256) and output layer (256, 2), with ReLU activations. We use cross-entropy loss and Adam optimizer with learning rate 0.001. We train for 100 epochs, and keep the checkpoint with best validation accuracy.

5 Alternative representation and model

We briefly explore one more representation of PowerShell scripts, that is representing the script with two simple features: the total number of nodes and the depth of the corresponding AST (Abstract Syntax Tree), as done in [15]. We use a Random Forest classifier to perform malware classification. We use an implementation from scikit-learn [20] with the default parameters.

IV Results & Discussion

A Perturbations on code2vec inputs

1 Accuracy of code2vec model in predicting Java method name

We tested the original data and the 7 adversarial perturbations and report the F1 scores, accuracy, precision, and recall in Table 1. Adding deadcode to the methods read in by code2vec showed little or no difference across all metrics tested.

There were some differences noted when variables were renamed to be synonyms, a target word ("sort"), or a random English word. F1 scores and Accuracy went down when these perturbations were made, albeit by a small amount. Interestingly, replacing the variable name with a nonsense string did not affect results. This difference is not due to different variables being replaced in different perturbations, because the renamed variable was tracked and made consistent across every perturbation type. For example, if "ZombieLogger" was the variable name renamed to "ZombieLumberjack" in the synonym-var perturbation, it would be the variable name that was renamed to a random English word in the randomword-var perturbation. Therefore, there must be something about changing a variable name to another, recognizable

English variable name that is more confusing to the model than replacing a variable name with a random string of letters. This fact implies that the model may have learned something about the semantics of the variable name for predicting the function of the Java method. All results were mild, but this may be because we only replaced one variable name per file, and a file may have multiple methods.

2 Examining the code embeddings for adversarial examples for code2vec

After the testing data were evaluated by the code2vec model, the embeddings were examined by two different methods. First, we looked at the L2 norm of each of the vectors, which are presented in Figure 2. The L2 norms of the original embeddings vs the embeddings of the methods that had had deadcode added were nearly indistinguishable. However, there were some mild differences in the L2 norm of vectors which had had variable renaming done, further supporting our finding from above that deadcode insertion has little effect in this case, and variable renaming can have an effect on classification of samples of code by code2vec. This finding is interesting, especially in light of the differences found by Yefet et al when deadcode was inserted.[4] It may be that their deadcode insertions disrupted the AST more than ours did, and therefore resulted in different results.

We also looked at the embeddings in a lower dimensional subspace using PCA, as shown in Figure 3. Unfortunately, there was no real separation between the different types of adversarial adjustments and the original, even when looking just at broad category types (deadcode-insertion vs variable-renaming).

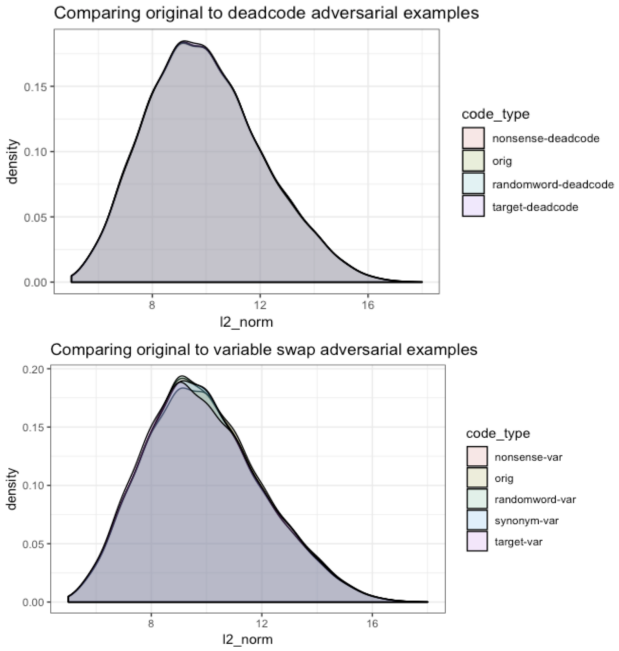

Figure 2: Density plot of the L2 norm of vector representations of deadcode-insertion adversarial examples (top) and variable-renaming adversarial examples (bottom).

B Results for Application 2: PowerShell Malware Classification

1 Qualitative evaluation of learned embeddings

To qualitatively explore how much the learned representations differ between non-obfuscated and the respective obfuscated scripts, we run the t-SNE algorithm on the whole test corpus (both obfuscated and non-obfuscated) and visually show the results on Figure 4. In general, we notice that there are not clear clusters in any of the plots separating malware from non-malware code, which could be due to the loss of information when projecting to 2D space and also the fact that the Seq2Seq model was not trained to cluster the embeddings based on maliciousness, but probably picked up on syntactic differences instead. However, we can see that the embeddings from the obfuscated scripts occupy the same general subspace area but are more tightly clustered, potentially indicating that obfuscation often makes scripts look more similar to each other. To some extent this is intuitive (imagine looking at a lot of obfuscated scripts, they will likely look barely distinguishable). More formally, this can also be a consequence of the limited vocabulary size, which likely does not contain complex obfuscated token patterns, so it results in a lot of <unk> tokens, making different scripts result in similar embeddings.

2 Quantitative evaluation of learned embeddings

We show a density plot of the L2 distance between the embeddings of non-obfuscated and obfuscated scripts in Figure 5.The density plot shows that the L2 distance is pretty large. Ideally we would want scripts that achieve the same functionality to have similar embeddings (small L2 distance). This shows the need for more robust representations that capture information that is invariant or does not change much under obfuscations.

Perturbation type	F1	Accuracy	Precision	Recall
original	0.47	0.51	0.57	0.40
target-deadcode	0.47	0.51	0.57	0.41
randomword-deadcode	0.47	0.50	0.57	0.40
nonsense-deadcode	0.47	0.50	0.57	0.40
synonym-var	0.45	0.49	0.55	0.38
target-var	0.46	0.49	0.56	0.39
randomword-var	0.45	0.48	0.55	0.39
nonsense-var	0.47	0.51	0.57	0.40

Table 1: Results from adding perturbations to the code2vec Java input files. Adding deadcode in general had no effect. Variable renaming caused a mild decrease across most metrics, except in the case of renaming a variable using nonsense strings rather than meaningful English words.

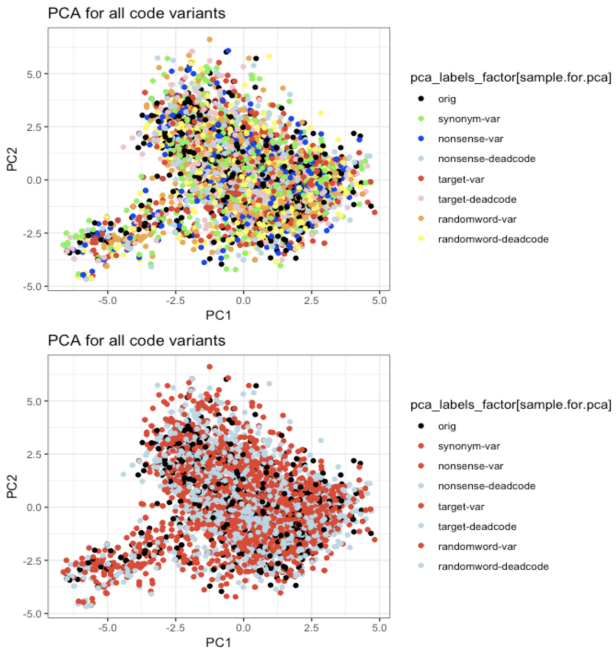

Figure 3: PCA with PC1 and PC2 of the various types of adversarial examples compared to the original. Only a random subsample of points are shown. Whether coloring all different types of perturbations (top) or highlighting only the differences between deadcode-insertion adversarial examples and variable-renaming adversarial examples (bottom), we see little separation.

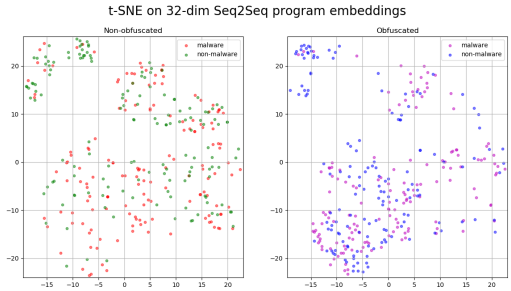

Figure 4: t-SNE visualization of non-obfuscated (left) and obfuscated (right) embeddings. While the obfuscated embeddings occupy the same general area in the subspace, we see they are more tightly clustered than their non-obfuscated counterparts.

3 Robustness in the malware classification task

We measure the accuracy of a binary malware classification model to quantify the robustness of such model against obfuscations. The results are given in Table 2. In the leftmost column we show which data was used for training: non-obfuscated, obfuscated or combined. On the first row we show which data was used for testing. We notice that regardless of what data we used for training, we can always correctly classify the non-obfuscated data. This can be due to the fact that the embeddings are high-dimensional and the model has high capacity, but can also be an artifact of the dataset. However, we see that when we evaluate with obfuscated data, the accuracy drops to almost random, especially in the case when the train data is non-obfuscated. This indicates that if a model is trained on only non-adversarial data, an adversary can simply obfuscate the input and bypass the malware detector which otherwise showed very promising results. We notice that one way we can easily improve the robustness is to include obfuscated data in the training set, although the performance on obfuscated data is still not satisfactory. This indicates the need for more robust representations, and not only more robust modeling or training techniques.

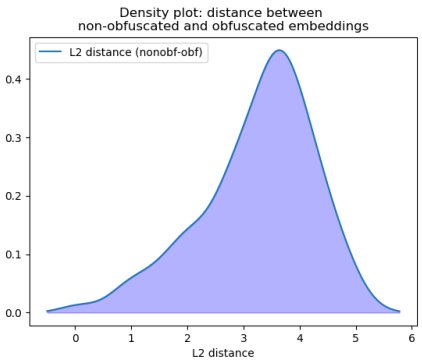

Figure 5: Density plot of the L2 distance between the non-obfuscated and obfuscated embeddings.

Train \ Test	non-obf	obf	comb
non-obf	1	0.506	0.753
obf	1	0.641	0.82
comb	1	0.638	0.819

Table 2: Accuracy scores (0 to 1) of the malware classification model for various configurations of train/test data

4 Alternative representation - AST features

In this approach we took the number of nodes and the depth of the AST as a representation of a PowerShell script and performed malware classification with a Random Forest model fitted on non-obfuscated data. We obtained 71% ROC on the test split of the non-obfuscated data, while that value was 45% when we evaluated on obfuscated data. We show the confusion matrices on Figure 6, where the x-axis corresponds to the predicted label, and the y-axis shows the true label. We notice that in the case with non-obfuscated data the confusion matrix shows high values on the diagonal, indicating correct classification (left plot). In the case with obfuscated data we notice that a lot of non-malware scripts are classified as malware (right plot). This can be explained by the fact that obfuscation often adds a lot of extra AST nodes, which is more common for malware scripts (even if they are non-obfuscated by us, they might be obfuscated by someone else).

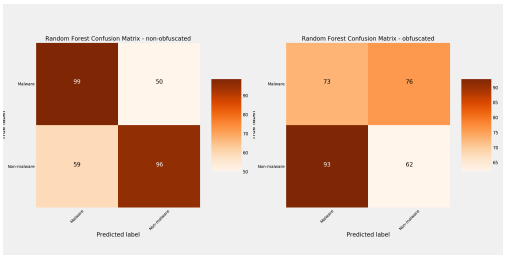

Figure 6: Confusion matrices for the Random Forest model performance on non-obfuscated (left) and obfuscated (right) data. The non-obfuscated performance is much better (dark on the diagonal).

V Future Work & Conclusion

In this work, we examined the brittleness of models of code to various adversarial adjustments, including complete obfuscation. We found that variable renaming had a mild effect on the accuracy of a code2vec model, while adding deadcode had very little effect. We have also shown that obfuscated PowerShell code can completely bypass otherwise well-performing discriminative models given a representation that treats code as a natural language. While the syntactic flexibility of PowerShell potentially comes as very useful for developers, it is a problem for the malware detectors which should ideally encode something about the functionality of the code, and not only its syntactical structure.

Future research may take many directions. A further refinement of the variable-renaming strategy for evaluating brittleness of models of code may be interesting, especially as these sorts of research questions get at what is called the “dual channel hypothesis": it has been suggested that source code is bimodal, consisting of both an algorithmic channel as well as a natural language channel through which programmers communicate intent to both the computer and to other programs.[21] The second channel is disrupted in our adversarial modifications and in our total obfuscations, and a clearer understanding of how the two different channels contribute to the understanding developed by models of code can be achieved by following these lines of inquiry. Future research may also include a focus on constructing robust deobfuscators, constructing well-curated datasets and of course finding a more suitable representation for the code itelf.

In conclusion, we find that models of code share many similarities to models of language, and may indeed be found to have a common intent of communication information in natural language, though code does not communicate exclusively in that channel. The similarities between code and purely natural language form a promising area of further study and collaboration for the two communities.

References

[1] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient Estimation of Word Representations in Vector Space. *arXiv:1301.3781 [cs]*, September 2013. arXiv: 1301.3781.

[2] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. *arXiv:1802.05365 [cs]*, March 2018. arXiv: 1802.05365.

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *arXiv:1810.04805 [cs]*, May 2019. arXiv: 1810.04805.

[4] Noam Yefet, Uri Alon, and Eran Yahav. Adversarial Examples for Models of Code. *arXiv:1910.07517 [cs]*, December 2019. arXiv: 1910.07517.

[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, January 2019.

[6] Daniel Bohannon. Invoke-Obfuscation v1.8. https://github.com/danielbohannon/Invoke-Obfuscation, 2018.

[7] Minhao Cheng, Jinfeng Yi, Pin-Yu Chen, Huan Zhang, and Cho-Jui Hsieh. Seq2Sick: Evaluating the Robustness of Sequence-to-Sequence Models with Adversarial Examples. *arXiv:1803.01128 [cs]*, April 2020. arXiv: 1803.01128.

[8] Robin Jia and Percy Liang. Adversarial Examples for Evaluating Reading Comprehension Systems. *arXiv:1707.07328 [cs]*, July 2017. arXiv: 1707.07328.

[9] Di Jin, Zhijing Jin, Joey Tianyi Zhou, and Peter Szolovits. Is BERT Really Robust? A Strong Baseline for Natural Language Attack on Text Classification and Entailment. *arXiv:1907.11932 [cs]*, April 2020. arXiv: 1907.11932.

[10] Victor Fang. Malicious powershell detection via machine learning. https://www.fireeye.com/blog/threat-research/2018/07/malicious-powershell-detection-via-machine-learning.html, 2018.

[11] Danny Hendler, Shay Kels, and Amir Rubin. Detecting malicious powershell commands using deep neural networks. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 187–197. ACM, 2018.

[12] Amir Rubin, Shay Kels, and Danny Hendler. Amsi-based detection of malicious powershell code using contextual embeddings. *arXiv: Cryptography and Security*, 2019.

[13] https://wordnet.princeton.edu/. About WordNet, 2010.

[14] http://www.mieliestronk.com/wordlist.html. Miliestronk's list of more than 58000 English words.

[15] Gili Rusak, Abdullah Al-Dujaili, and Una-May O'Reilly. Ast-based deep learning for detecting malicious powershell. *CoRR*, abs/1810.09230, 2018.

[16] Daniel Bohannon. PowerShell Corpus 400k. https://aka.ms/PowerShellCorpus.

[17] https://developers.virustotal.com/. Virustotal.

[18] Ben Trevett. acked Padded Sequences, Masking, Inference and BLEU. https://github.com/bentrevett/pytorch-seq2seq/blob/master/4

[19] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2014.

[20] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[21] Santanu Kumar Dash, Miltiadis Allamanis, and Earl T. Barr. RefiNym: using names to refine types. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*, pages 107–117, Lake Buena Vista, FL, USA, 2018. ACM Press.