Logical Segmentation of Source Code

Jacob Dormuth, Ben Gelman, Jessica Moore, David Slater

Machine Learning Group
Two Six Labs
Arlington, Virginia, United States
E-mail: {jacob.dormuth, ben.gelman, jessica.moore, david.slater}@twosixlabs.com

Abstract

Many software analysis methods have come to rely on machine learning approaches. Code segmentation - the process of decomposing source code into meaningful blocks - can augment these methods by featurizing code, reducing noise, and limiting the problem space. Traditionally, code segmentation has been done using syntactic cues; current approaches do not intentionally capture logical content. We develop a novel deep learning approach to generate logical code segments regardless of the language or syntactic correctness of the code. Due to the lack of logically segmented source code, we introduce a unique data set construction technique to approximate ground truth for logically segmented code. Logical code segmentation can improve tasks such as automatically commenting code, detecting software vulnerabilities, repairing bugs, labeling code functionality, and synthesizing new code.

1 INTRODUCTION

With the proliferation of open-source development practices and code sharing services, such as GitHub and Bitbucket, large bodies of source code are increasingly available to developers. There are a number of ways in which these code corpora could assist with the software development process; of particular interest is the application of machine learning to software engineering practices. Recent literature has utilized machine learning with source code at scale, developing tools to generate comments [7], detect software vulnerabilities [11], repair bugs [4], label functionality [2], and synthesize new code [6]. Code segmentation - the process of decomposing source code into meaningful blocks - can augment these methods, for instance, by determining what portions of a file are functionally similar, identifying where to generate automatic comments, and locating useful sub-function boundaries for bug detection.

Current approaches to segmentation do not intentionally capture logical content that could improve its usefulness to the aforementioned problems. Traditionally, code segmentation has been done at a syntactic level. This means that language-specific syntax, such as the closing curly brace of a class or function, is the indicator for a segment. Although this method is conceptually simple, the resulting segments do not intentionally take into account semantic information. Syntactic structures in natural language, such as sentences and paragraphs, are generally also indicators of semantic separation. In other words, two different paragraphs likely capture two logically separate ideas, and it is simple to delineate them by splitting at the end of each paragraph. Locating logical segments in source code, however, is a non-trivial task. Syntactic structures, such as a for-loop followed by an if-statement, are not particularly indicative of semantic changes in the code. Determining if the for-loop and if-statement are working to achieve the same logical task is a more difficult problem than the paragraph delineation analogue. Logical code segmentation captures arbitrary combinations of syntactic structures in a single segment.

Logical code segmentation has a multitude of uses, including: improving code search tools by returning relevant segments of code instead of entire files or functions, recommending locations to add comments, classifying the functionality of source code by reducing the problem space from entire files/projects to concentrated blocks of code, or using the segments as features for a model that attempts to determine the modularity or complexity of a given code file. Logical segments are able to featurize code, reduce noise, and limit a problem space to certain types of segments.

In this work, we develop a novel code segmentation method to generate logical segments in a language-agnostic fashion. Although language specificity may help model code by allowing options such as language-specific tokens and abstract syntax trees, it adds a non-trivial burden to operationalizing tools. Language specific approaches require

DOI reference number: 10.18293/SEKE2019-026

training many models, each with their own parsers, training sets, and hyperparameters. Our language-agnostic approach is able to avoid those extra steps, improving the generalizability, availability, and ease-of-use of the results. Because there is no data set of source code that is conveniently split into logical segments, we first establish a unique data set construction process using Stack Overflow[1] (SO). Human curation is critical to determining logical segments, and Stack Overflow provides that at a vast scale.

We build on prior work in natural language processing, training a bidirectional long short-term memory (LSTM) neural network to split source code into logical segments. Since this is the first work to perform this segmentation in the source code domain, we provide baseline models and multiple deep neural networks for comparison. We validate the results on six programming languages to display the language agnosticism of the method. Lastly, we qualitatively discuss the model's results on real source code documents to provide insight on performance in the desired domain.

Our main contributions are as follows:

- A novel data set construction method that utilizes crowd-sourced data to approximate logical segments.
- First work, to our knowledge, to perform logical code segmentation regardless of syntactical correctness or programming language.
- Baseline models for the logical segmentation problem, demonstrating the relative effectiveness of our language-agnostic, deep neural network on six different programming languages.

2 RELATED WORK

The literature on source code segmentation is relatively sparse, with existing work offering syntax-specific solutions and requiring language specificity. Ning, Engberts, and Kozaczynski develop a code browser that allows users to select useful, reusable segments by analyzing the control flow and data flow in the abstract syntax trees (ASTs) of COBOL programs [9]. More recently, Wang, Pollock, and Vijay-Shanker attempt to locate logical segments by developing rules that look for specific syntactical patterns [12]. Wang et al. generate an AST for a given Java method, apply rules that analyze the data flow and syntactic structures, and then add line breaks around the resulting segments to enhance the readability of the method. Although these methods provide sophisticated analysis and rules, they are inextricably linked to the language that they operate on, and thus will not transfer to other languages.

In this work, we deviate from the practice of using ASTs as the foundation for our analysis. By treating source code as a body of text akin to an NLP problem, we avoid any programming-language-specific challenges posed by other methods. Text segmentation has been researched more thoroughly than the source code analogue, with methods ranging from LDA [10], to semantic relatedness graphs [3], to deep learning approaches [1]. Of particular note is the use of bidirectional LSTMs to identify the breaks between segments of Wikipedia articles [8]. An LSTM is a recurrent neural network that processes sequential information [5]. Although we use the bidirectional LSTM as the primary framework of our model, we make several changes to adapt to the source code domain. We use a character embedding instead of the commonly used word embedding: due to the vast number of possible unique identifiers in source code, optimizing an embedding for each token is infeasible. Additionally, we require an evaluation metric specifically based on our data generation method to represent how well the model recognizes segments the method creates.

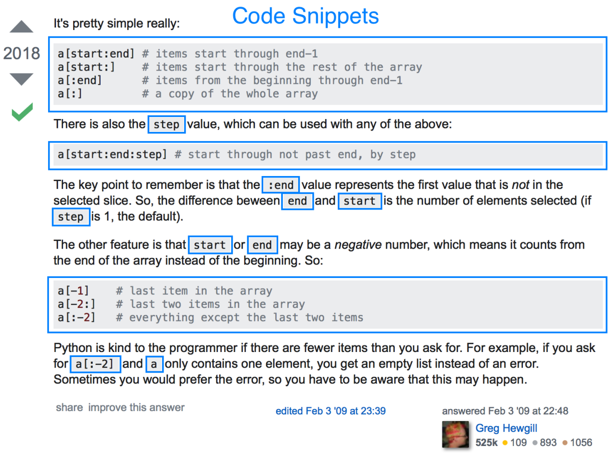
3 DATA SET GENERATION

We model code segmentation as a classification problem, where given a sequence of characters, we must predict whether a character denotes the beginning of a new code segment. In order to generate training data suited to the task, we use Stack Overflow, a forum where users can ask questions, receive answers, and post code snippets on a wide range of computer programming topics. Because the posters are focused on answering a specific question, the code snippets are generally geared towards a single logical task. Fig. 1 shows an example response on Stack Overflow containing blocks the user has marked as code. We pull code snippets by searching for posts tagged with six different programming languages: C, C++, Java, Python, Javascript, and C#.

One problem with the data, however, is that the distribution of the number of lines per code snippet is heavily skewed. Fig. 2 shows that the majority of code snippets are only a few lines long. Using the entire data set could bias the model to predict segments every few lines because the model will have seen so many short code snippets, which is unlikely to be the case in real source code files. Using only very long snippets, however, would not leave much training data. Thus, we heuristically filter out all code snippets that are less than four lines long using the elbow method.

After filtering, we generate segments by concatenating snippets with a newline character, thereby marking the beginning of a new segment. We refer to these as "dividing newlines." It is important to note that not all newlines mark a new segment because a single code snippet may contain many newlines. After this process is complete, the result is essentially a giant block of concatenated code snippets. To obtain individual data points, we iterate through the block

[1] https://stackoverflow.com



It's pretty simple really:

```
a[start:end] # items start through end-1
a[start:]    # items start through the rest of the array
a[:end]      # items from the beginning through end-1
a[:]         # a copy of the whole array
```

There is also the `step` value, which can be used with any of the above:

```
a[start:end:step] # start through not past end, by step
```

The key point to remember is that the `:end` value represents the first value that is *not* in the selected slice. So, the difference beween `end` and `start` is the number of elements selected (if `step` is 1, the default).

The other feature is that `start` or `end` may be a *negative* number, which means it counts from the end of the array instead of the beginning. So:

```
a[-1]    # last item in the array
a[-2:]   # last two items in the array
a[:-2]   # everything except the last two items
```

Python is kind to the programmer if there are fewer items than you ask for. For example, if you ask for `a[:-2]` and `a` only contains one element, you get an empty list instead of an error. Sometimes you would prefer the error, so you have to be aware that this may happen.

share improve this answer
edited Feb 3 '09 at 23:39
answered Feb 3 '09 at 22:48
Greg Hewgill
525k ● 109 ● 893 ● 1056

Figure 1. Example Stack Overflow post showing code snippets.

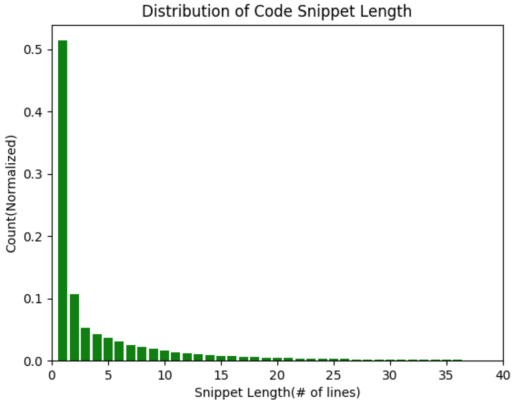


Figure 2. Graph depicting the heavily skewed snippet length distribution on Stack Overflow.

of snippets and generate data points using three methods: bag of characters, uncentered, and centered, shown in Fig. 3.

In the bag of characters method, one data point is created by taking 7 lines from the block of snippets and counting the characters to create a "bag of characters" for each line. These bags of characters are then concatenated together to create a single training sample. If the middle newline (the fourth of seven) is a dividing newline, then the label for this data point is a 1. Otherwise, the label is a 0. This process is repeated by sliding the 7 line window forward by 1 line. We track the counts of 256 unique characters, corresponding to all the ASCII characters, with an average of about 29 characters per line.

In the uncentered method, we create a data point by selecting all the characters in a 100-character window. In this process, we assign one label for each character in the window, meaning one data point has 100 characters and 100 labels. If the character is a dividing newline, the corresponding label is a 1. This process is repeated by sliding the window by 100 characters. It is important to note that there is no guarantee a data point in this method will contain a dividing newline (or any newline at all). Even when the data point does contain a dividing newline, there is no guarantee that it will occur in the center of the input.

In the centered method, we create a data point by locating a newline and taking a window of 50 characters before and after that newline, for a total of 101 characters. If that newline is a dividing newline, then the label for this data point is a 1. Otherwise, the label is a 0. This process is repeated by centering the window on the next occurring newline. Whilst iterating through the block of snippets, any window that does not have a newline in the center will be ignored. It is important to note that other newlines, including dividing newlines, may occur in the window; however, the label is assigned based only on the center newline.

4 METHODOLOGY

We experiment with three models: a logistic regression model (to serve as a baseline), and two neural network architectures that utilize bidirectional LSTM layers. Every model is trained on seven training sets. Six of the training sets correspond to the different languages: C, C++, Java, Python, Javascript, and C#. The last training set combines all of the languages in order to evaluate language agnosticism.

The logistic regression model utilizes the bag of characters data format for training. Each bag of characters, corresponding to one line of code, contains counts for 256 unique characters. The model takes seven bags of characters, for a total input size of 1,792. The output is a value from 0 to 1, representing the probability that the bag of characters corresponding to the fourth line (the middle line out of 7) contains a dividing newline (recall the labeling scheme from section 3).

The first neural network architecture uses the uncentered data format for training. This model has an input layer of length 100, one for each character in an uncentered data point. Each character is passed through a 20-dimensional character embedding, which converts a character to a 20-dimensional, real-valued vector. These embeddings are passed to a bidirectional LSTM layer of size 256. A bidirectional LSTM allows for past and future information to be used together, whereas a standard LSTM only considers past information. If a human were to segment source code, they would likely look ahead and use future information to piece together their decisions. The sequential information that the bidirectional LSTM learns is condensed using three

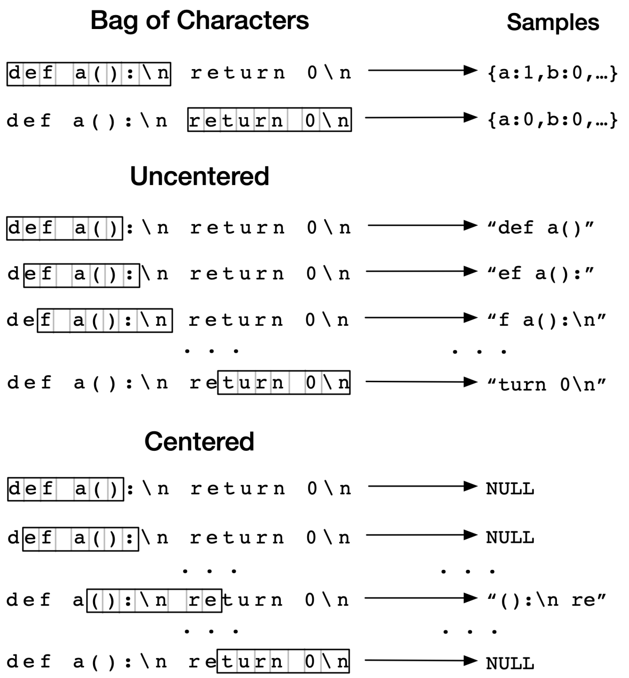


Figure 3. The three data generation methods operating on the same piece of code. The window sizes are shortened for visual clarity. In the bag of characters method, each sample is the concatenation of seven bags of characters corresponding to seven consecutive lines. In the uncentered method, each sample is simply all the characters in the window. In the centered method, each sample is all the characters in a window where the middle character of the window is a newline. If a window in the centered method does not contain a newline in the center, no sample ("NULL") is generated.

time-distributed dense layers, sizes 150, 75, and 1, respectively. Since the layers are time distributed, the last layer has 100 total outputs (one for each time step). Each output is the probability that the character at that time step indicates the beginning of a new logical segment. The uncentered data format has 1 label for each character, so the 100 outputs and 100 labels are used to compute the loss. We use a batch size of 128, a dropout strength of 0.2 between each layer for regularization, and binary cross entropy as our loss function.

The second neural network architecture uses the centered data format for training. In terms of model structure, it is nearly identical to the uncentered model except that the input size is 101 characters and that it has one additional layer. The last layer of the centered model is a dense layer that maps the 100 time distributed outputs (last layer of the uncentered model) to a single output. Because the centered data format guarantees a newline at the center of the input sequence, only a single output is required from the centered model. That single output and the label for the center newline are used to compute the loss.

4.1 Evaluation

Due to our data generation method, only a newline character can denote the beginning of a new code segment. As a result, we specifically measure the newline accuracy of our models. This distinction is critical because the logistic regression and centered models predict only on newlines, while the uncentered model outputs a prediction for every character, which would skew accuracy. The newline accuracy metric is the percentage of newlines that are classified correctly as either a dividing newline or non-dividing newline.

We split the collection of code snippets into train/validation/test sets of 80% / 10% / 10%. The training process is stopped when the model does not improve after 20 epochs.

5 RESULTS

In total, we train 21 different models: 18 single-language models and 3 multi-language models. For each of the three architectures (logistic regression, uncentered LSTM, centered LSTM), we train a model on each language individually, as well as a model on all the languages simultaneously. Table 1 displays the newline accuracies for each model/language pairing.

The logistic regression models perform the worst, but are still significantly better than predicting a non-dividing newline every time (non-diving newlines are the most common). This is an expected result because the logistic regression models do not take into account character interactions like the more complex models. The neural network models show a significant improvement in performance over the logistic regression baselines.

Although the single-language models generally perform slightly better than their multi-language counterparts, the difference in newline accuracy is relatively small. This is a very positive result because the multi-language models are able to discern logical segments across languages with little to no performance hit.

Another interesting comparison is the difference in performance between the uncentered and centered models. In the single-language category, the uncentered models outperform the centered models across the board. On the other hand, in the multi-language category, the centered models are usually more effective. One possible reason for this is the difference in code context between uncentered and centered data points. In the uncentered data format, it is possible for a data point to contain no newlines whatsoever. This may help the model understand the difference

Table 1. Newline Accuracies of Each Model

Model	Language					
	C	*C++*	*Java*	*Python*	*Javascript*	*C#*
Single-Language Logistic Regression	95.26	95.2	95.6	91.61	94.53	95.63
Single-Language Uncentered LSTM	**98.8**	**98.76**	**98.96**	**99.17**	**99.2**	**99.3**
Single-Language Centered LSTM	97.35	98.35	98.84	97.82	98.79	98.87
Multi-Language Logistic Regression	94.76	94.81	95.1	90.76	93.48	95.09
Multi-Language Uncentered LSTM	98.33	98.56	98.75	**97.98**	98.65	98.86
Multi-Language Centered LSTM	**98.5**	**98.62**	**98.82**	97.83	**98.73**	**98.95**
Percent Non-Dividing Newlines	92.06	92.12	92.35	91.10	92.02	92.12

Table 1. Newline accuracy test results for every model and language. "Single-Language" models are trained and tested on one language at a time. "Multi-Language" models are trained on every language simultaneously and then tested on each language separately. "Percent Non-Dividing Newlines" is the newline accuracy if a model were to always predict non-dividing for every newline.

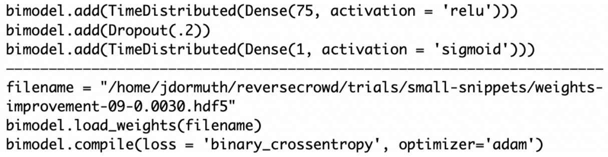
in context when a newline is actually present. In the multi-language scenario, however, snippets with no newlines may have a fundamentally different pattern for each language in the data set, which may significantly complicate what the model needs to learn. The centered multi-language models only need to learn the differences between languages as it pertains to the context around a newline, reducing the cost of learning on multi-language data. Given enough data and time, it may be possible that the multi-language models would be able to utilize the uncentered snippets more effectively.

One other noteworthy observation is the impact of similarity across languages. In the multi-language scenario, Python is the most syntactically distinct language and consistently performs the worst. It is feasible that the models are able to transfer knowledge across languages, so similar languages may benefit from each other's data. The differences in results between languages may also speak to the quality of Stack Overflow snippets for those languages. It is possible that different programming languages attract different questions, topics, and code snippet qualities on Stack Overflow, which would ultimately influence the model's performance.

5.1 Testing on Source Code

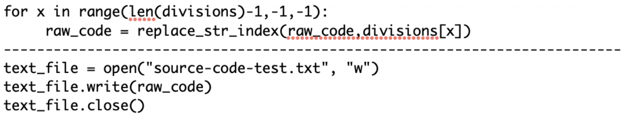
In order to better understand the utility of these models, Fig. 4 and 5 showcase examples of the single-language deep learning model running on a Python source file.

In Fig. 4, the model is able to recognize when the functionality of the code changes from defining a Keras model to loading the weights and compiling the model. Fig. 5 shows how the model is able to differentiate between a string operation in a for-loop and a new task of opening and writing to a text file.



```
bimodel.add(TimeDistributed(Dense(75, activation = 'relu')))
bimodel.add(Dropout(.2))
bimodel.add(TimeDistributed(Dense(1, activation = 'sigmoid')))
-----------------------------------------------------------------------
filename = "/home/jdormuth/reversecrowd/trials/small-snippets/weights-
improvement-09-0.0030.hdf5"
bimodel.load_weights(filename)
bimodel.compile(loss = 'binary_crossentropy', optimizer='adam')
```

Figure 4. The model recognizes that compiling and loading the weights of a Keras model is a different task from defining the layers of the model. The dashed line is the model's prediction of the segment location.



```
for x in range(len(divisions)-1,-1,-1):
     raw_code = replace_str_index(raw_code,divisions[x])
--------------------------------------------------------------------------
text_file = open("source-code-test.txt", "w")
text_file.write(raw_code)
text_file.close()
```

Figure 5. The model distinguishes between string operations and writing to a file. The dashed line is the model's prediction of the segment location.

6 CHALLENGES AND LIMITATIONS

One of the biggest challenges is the lack of ground truth logical segments. Because the Stack Overflow code snippets are simply concatenated together, the resulting data is not necessarily representative of real code. It is possible that two snippets concatenated together do not result in syntactically correct code. There is also no guarantee of standard formatting practices that one would expect to see in formal software projects.

Additionally, the data generation method assumes that a code snippet represents a logical chunk of code. Although this is usually the case, there is no way to guarantee that all code snippets are segmented logically. It is possible that an individual snippet contains multiple logical tasks. It is also possible that two randomly selected snippets perform similar tasks; this introduces noise to the data set because they will still be labeled with a dividing newline.

One possible way to address these challenges is to use in-line comments in source code files as logical division points. There is intended meaning behind the placement of comments in source code files, whereas the concatenation of Stack Overflow code snippets is randomized. This method could better reflect the properties of source code.

7 CONCLUSIONS AND FUTURE WORK

We present a novel method to perform logical segmentation of source code. Using crowd-sourced data from Stack Overflow, we create a unique data set construction technique to approximate logical segments in source code. Drawing from the NLP domain, we develop deep neural network models utilizing bidirectional LSTMs that can predict on source code regardless of language or syntactical correctness. Lastly, we provide baselines and an appropriate metric to evaluate the performance of our models with regard to our data set construction.

Although our method is the first success in language-agnostic logical code segmentation, there are a variety of potential architectural and parameter improvements. We could incorporate an attention mechanism into the LSTM, allowing the model to learn more specific features in the source code. Another avenue could be adjusting model parameters such as window size, network depth, and loss functions. For example, the loss function of the segmentation models could be modified to incorporate the error term of another task that uses segments as input. Finally, input representations that are larger than character-scale may improve the models without significant increases in model complexity; word or token embeddings in addition to the character embedding may be able to achieve this.

ACKNOWLEDGEMENTS

This project was sponsored by the Air Force Research Laboratory (AFRL) as part of the DARPA MUSE program. We would like to thank Robert Gove and Casey Haber for their valuable feedback, support, and contributions to figures. We also thank Banjo Obayomi for infrastructure support.

References

[1] P. Badjatiya, L. J. Kurisinkel, M. Gupta, and V. Varma. Attention-based neural text segmentation. In *European Conference on Information Retrieval*, pages 180–193. Springer, 2018.

[2] B. Gelman, B. Hoyle, J. Moore, J. Saxe, and D. Slater. A language-agnostic model for semantic source code labeling. In *Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis*, pages 36–44. ACM, 2018.

[3] G. Glavaš, F. Nanni, and S. P. Ponzetto. Unsupervised text segmentation using semantic relatedness graphs. Association for Computational Linguistics, 2016.

[4] J. Harer, O. Ozdemir, T. Lazovich, C. Reale, R. Russell, L. Kim, et al. Learning to repair software vulnerabilities with generative adversarial networks. In *Advances in Neural Information Processing Systems*, pages 7944–7954, 2018.

[5] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[6] V. Kashyap, R. Swords, E. Schulte, and D. Melski. Musynth: Program synthesis via code reuse and code manipulation. In *International Symposium on Search Based Software Engineering*, pages 117–123. Springer, 2017.

[7] J. Moore, B. Gelman, and D. Slater. A convolutional neural network for language-agnostic source codesummarization. In *ENASE (to appear)*, 2019.

[8] N. Mor, O. Koshorek, A. Cohen, and M. Rotman. Learning text segmentation using deep lstm. 2017.

[9] J. Q. Ning, A. Engberts, and W. V. Kozaczynski. Automated support for legacy code understanding. *Communications of the ACM*, 37(5):50–58, 1994.

[10] M. Riedl and C. Biemann. Text segmentation with topic models. *Journal for Language Technology and Computational Linguistics*, 27(1):47–69, 2012.

[11] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 757–762. IEEE, 2018.

[12] X. Wang, L. Pollock, and K. Vijay-Shanker. Automatic segmentation of method code into meaningful blocks to improve readability. In *2011 18th Working Conference on Reverse Engineering*, pages 35–44. IEEE, 2011.