Data Augmentation for Code Analysis

Alexander Shroyer
Intelligent Systems Engineering
Indiana University
Bloomington, USA
ashroyer@iu.edu

D. Martin Swany
Intelligent Systems Engineering
Indiana University
Bloomington, USA
swany@iu.edu

***Abstract*—A key challenge of applying machine learning techniques to binary data is the lack of a large corpus of labeled training data. One solution to the lack of real-world data is to create synthetic data from real data through augmentation. In this paper, we demonstrate data augmentation techniques suitable for source code and compiled binary data. By augmenting existing data with semantically-similar sources, training set size is increased, and machine learning models better generalize to unseen data.**

***Index Terms*—Deep Learning, Data Mining, Text Analytics, Statistical and Structural Pattern Recognition**

I. Introduction

Modern software development relies heavily on third-party code and open ecosystems, which aid developer productivity, but are also attractive targets for malicious actors. Attackers may release malware with a name similar to a real package in an attack called "typosquatting", or try to gain access to an existing project and quietly add malicious code[1]. New technologies such as GitHub's Copilot[1] provide developers with snippets of third party code sourced from public repositories, which in addition to aiding productivity, may also be used as an attack vector for injecting publicly available malware into new code.

Recent typosquatting attacks include a malware named `colourama`, which targeted the similarly-named Python package `colorama`. The malware bundled cryptocurrency mining code alongside the original code[2], so developers were less likely to notice the problem. Similarly, the node.js malware `crossenv` typosquatted `cross-env` and exfiltrated user data. Detecting and preventing typosquatting is an active area of research[3].

Today's developers need tools to help find hidden malware in third party code. Machine learning techniques are rapidly finding applications in this domain because models can be made general enough to classify never-before-seen code as either malware or not. However, a limiting factor for machine learning model efficacy is the quantity of training data. One project used a large dataset of 66,388 PowerShell commands (6290 malicious)[4]. Another project sourced approximately 3500 implementations of 6 different algorithms from GitHub[5]. However, a single PowerShell command may represent a standalone program or a small part of a larger program, and not all projects have access to such a large volume of data as these examples.

When the quantity of available data is too small, a classic approach to artificially synthesizing more data is called *data augmentation*. In computer vision, image augmentation includes rotation, translation, adjusting color, or adding noise. In natural language processing, text is augmented by translating to another language and back again, adding or deleting words, and permuting word order. In this work, we examine data augmentation techniques suitable for source and compiled binary code analysis.

Data Augmentation for Source Code. Some data augmentation methods from the Natural Language Processing (NLP) domain also work on source code. Like natural language, source code can be thought of as a 1-dimensional sequence of symbols, with specific word formation rules. However, in natural languages it is rare for an author to invent new words, but programmers create new names routinely. Programming language is also more structured than natural language, with extensive use of explicit grouping structures like parentheses and semantically significant indentation and punctuation. These aspects make it harder to define "synonyms" for word-like tokens in programming languages or to perform transpositions without changing the program's meaning or breaking it completely.

Source code can also be thought of as a 2-dimensional image where each character or symbol represents a "pixel". This can be done by treating line feed characters as vertical offsets so the "image" is the same character grid as used by all common text editors. Sequential code can be embedded into a square matrix by treating sequences of two tokens as row/column indexes and recording occurrence counts of the two-token n-gram at each index.

Most code has non-linear control flow in the form of conditional branching, and can be embedded into a graph data structure as in [6]. This type of embedding can be further transformed into a regular or rectangular shape such as an adjacecy matrix which is more amenable to processing by machine learning libraries. These machine learning libraries typically expect samples to have uniform shape. Any of these interpretations can be further augmented by adding pseudo-random noise, for example additive Gaussian or Poisson noise as used by [7].

Data augmentation techniques which preserve the semantics

[1]https://github.com/features/copilot/

of the original source code require specialized knowledge of the source code languages used. For C language, examples of tools which can alter the syntax while preserving semantics include llvm-obfuscator[8] and Tigress[9].

In this project, we apply code-specific data augmentation to standard library code and analyze how it affects the robustness of a machine learning model in detecting specific functions in compiled binary data.

II. Related Work

Recently, Bayer et. al [10] showed that data augmentation can be performed on natural language texts by leveraging a large pre-trained model like GPT-2. This allowed then to generate novel tokens for insertion into their data set. Critically, these novel tokens were derived from a large model which preserved the semantic relationships between words. This type of token generation would be beneficial to future work in code analysis, but first the field would need a large pre-trained model. One potential use of this technology could train a model on a large corpus of source code, and use this as a mechanism for generating summary tokens for local context. GitHub's CoPilot project uses OpenAI's Codex language model[2], whose purpose is to translate natural language commentary into valid code. GitHub is well positioned to take advantage of the Codex model due to their massive collection of source code as well as commentary explicitly related to that code. While generating code from commentary is related to semantic analysis of obfuscated binary code, it is not identical. For example, in the task of reverse engineering an obfuscated binary blob, natural language commentary may be either non-existent or intentionally misleading as an additional form of obfuscation.

Marastoni et. al found that reshaping binary files into 2d image data allowed them to use conventional convolutional neural networks (CNNs) even though they arbitrarily restructured 1d data into 2d by choosing an image width of 64[11]. They showed that reshaping actual image data had negligible effect on accuracy, but reshaping compiled program data into different width images had a drastic effect on accuracy. Their explanation is that the model is more complex than simple digit recognition. Another consideration is that by imposing an arbitrary shape on one-dimensional data, their CNN finds correlations between instruction values that happen to be arranged on intervals of 64 bytes. This work uses Tigress to augment a selection of 47 C programs and ultimately obtains a distribution of 9400 programs representing the 47 original categories.

Gupta et. al applied a deep neural network with attention mechanism to find syntactic fixes to 6971 buggy student programs[12]. This task was able to make use of an oracle (compiler error messages) - if the compiler produced an error message, the program contained at least one bug. While this approach is language-agnostic, it is limited to only syntactic errors. Logical errors and malware cannot be detected by such an approach.

An early example of using data augmentation for malware detection is due to Catak et. al[7]. In this work, random noise is injected into examples of both regular and malware programs. Like [11], Catak et. al[7] use a CNN model after reshaping the programs into images. However, instead of using the raw byte data, they synthesize a color PNG image file, by substituting decimal conversion, entropy conversion, and zeros of the original program into the red, green, and blue pixel values, respectively. This extra step embeds more summary information into each pixel compared to [11].

Asm2Vec[13] builds on the foundation of Word2Vec[14] and applies the Paragraph Vector-Distributed Memory (PV-DM) concept [15] to assembly code. Like [11], this work also leverages Tigress and llvm-obfuscator for data augmentation. However, Tigress-transformed programs were omitted from the complete evaluation. A key method which distinguishes Asm2Vec is their use of control flow to embed a partial function call graph structure in addition to PV-DM of instructions and operands. Even though this type of graph embedding is not language agnostic, we believe control flow more accurately models the structure of the code and should be preferred to n-gram embeddings or arbitrary reshaping into images.

III. Methods

Analysis of unknown binaries involves sorting through both familiar and unfamiliar code. Standard library code is included in many other programs, so the ability to robustly identify standard library functions aids security analysis and reverse engineering efforts. In this work, we analyze code from the `musl` C library[16]. It is used by projects such as Alpine Linux[3] and Emscripten[4]. The source code for `musl` in this work comes from the `musl-cross-make` toolchain at revision **b298706**[5]. We chose `musl` for this project because of its compactness and relative ease of compilation compared to similar implementations such as glibc[6].

TABLE I
SELECTED `musl` LIBRARY FUNCTIONS

Library	Function	Library	Function
ctype.h	isalnum	stdio.h	printf
ctype.h	tolower	stdlib.h	free
math.h	exp	stdlib.h	malloc
math.h	floor	stdlib.h	strtol
math.h	pow	string.h	strcat
stdio.h	fprintf	string.h	strstr

The internal implementation of `musl` contains many examples of code reuse. This means that in order to compile a single `musl` function it is usually necessary to compile the library in its entirety. After compilation, the resulting object

[2]https://openai.com/blog/openai-codex/

[3]https://www.alpinelinux.org/

[4]https://emscripten.org/

[5]https://github.com/richard-vd/musl-cross-make/commit/b298706d99a30c72c26ba82c46b5bbd8550f5296

[6]https://www.gnu.org/software/libc/

file contains several thousand public symbols. However, we restrict our analysis to a subset of library functions, given by Table I. These functions are common to many programs, but more importantly this subset contains groups of similar functions. For example, `printf` and `fprintf` are similar, as are `exp` and `pow`, but `printf` and `pow` are different. Likewise, we expect functions from the same library to be more similar than functions from separate libraries. We use this similarity assumption as a baseline to indicate whether we have found reasonable clusters of related functions.

A. Data Augmentation by Source Code Obfuscation

Obfuscating source code means modifying the structure of the code without changing its core functionality. This can include trivial changes like renaming variables or more substantial changes like altering the structure with additional control flow, spurious functions, or even self-modifying code. For compiled languages, variable renaming has a negligible effect on the binary output, and even some structural changes like changing `while` loops to `for` loops may result in the same binary code after optimizations are applied. In this work, we obfuscate C sources using Tigress[17], a source-to-source transformer for C code. Tigress transformations include: splitting functions into smaller parts, merging multiple functions into one, flattening control flow, transforming functions into specialized interpreters, and many others. Each transformation results in a different source file, and multiple transformations may be combined to produce additional sources.

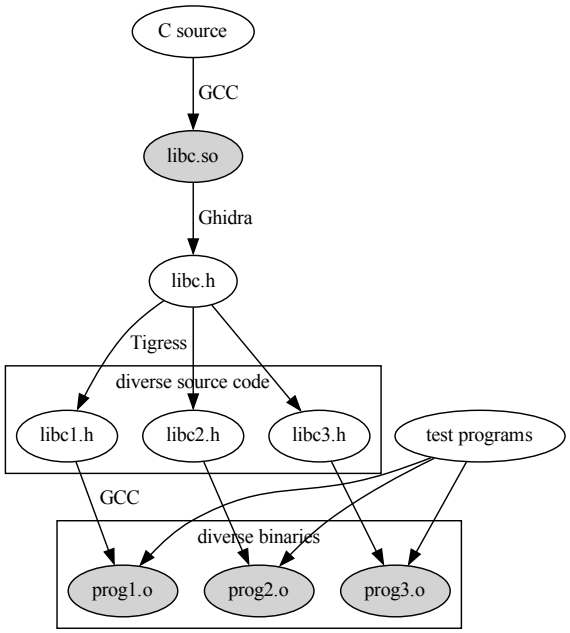

Fig. 1. Generating diverse binaries from single sources

A challenge of using Tigress is that it can only transform a single C *program* file, not a *library* consisting of many files, so to use Tigres with `musl`, the recommended method is to merge separate files into one large C file. However, we determined this approach was infeasible, due to `musl-cross-make`'s complex multi-stage build process.

Instead, we use a workflow as shown in Figure 1. We begin with the `musl-cross-make` cross-compiler to build the shared object file `libc.so`, for the `x86_64-linux-musl` target. Then we *decompile* functions of interest (and their sub-functions), using Ghidra[18]. Decompilation is the process of generating C code from binary code. However, decompilation is an imperfect process, and Ghidra's decompiler does not always generate syntactically valid C code, so we manually fix the syntax errors in the decompiled C files. Through this method, we obtain a single-source representation of the source code for all function. All the functions are finally merged to form a new library file, which we call **libc.h**.

B. Semantics-Preserving Transformations

Tigress offers many program transformations, and many of these transformations can be combined. We use two selected "recipes" from the Tigress website[7]: *Opaque Predicates, Branch Functions, and Encoded Arithmetic* and *Virtualization and Self-Modification*. In addition to these premade recipes, we also transform with *Encoded Arithmetic*, *Encode Branches*, and *Virtualize* separately, for a total of 5 separate transformed C sources. Generating 5 different sources is analogous to obtaining 5 different standard library implementations, in that they all have the same semantics, and only differ in implementation details.

C. Test Programs

Our approach aims to reliably detect standard library functions within an obfuscated binary. To test this, we wrote multiple test programs which call functions from **libc.h**.

We use a selection of trivial C programs as data classes. Each program contains one library function, and the filename describes the function under test, for example `pow.c` calls the `pow` function. These programs are short and trivial, averaging about 14 lines of code. The key feature of these programs is the inclusion of runtime data to prevent the compiler from optimizing away the function under test. For example, the file `pow.c` is shown in Figure 2.

```
int main(int argc,char**argv){
    return (int)pow(1.3, argc);
}
```

Fig. 2. `pow.c` source code

Because the `pow` function uses the run-time parameter `argc`, its result cannot be computed at compile-time. This ensures the binary code for the `pow` function appears in the compiled output. Since all the test functions are trivial, the compiled outputs differ primarily by the library function

[7]https://tigress.wtf/recipes.html

they contain, plus a negligible amount of boilerplate code for the `main` function. A version of **libc.h** which provides the `pow` function is included at compile-time with `gcc`'s `-include` option. When we compile the code, we only compile (with `gcc -c`) and do not link, which means no link-time optimizations execute, and all **libc.h** functions appear in the object file, even if they are not used in the test program.

D. Data Augmentation Types

We consider data augmentation to be an important step to improve binary analysis techniques from a security perspective. Although there exist methods[19], [20] for determining binary semantics, they are sensitive to minor semantic changes - exactly the kind of changes that would be caused by the addition of malware patches to a binary. Other methods [21]–[23] rely on obtaining Control Flow Graphs (CFGs) from the binary or other types of "expert knowledge", which may induce bias. The main goal of using Data Augmentation is to improve model accuracy by increasing the amount of training data, all without introducing additional bias.

Data Augmentation involves making small changes to the source material which do not drastically change its meaning. For example, adding small amounts of Gaussian noise to a picture of a teapot results in a picture of a teapot with some bad pixels, not a picture of a cat. Similarly, image data is often augmented through linear transformations such as rotations, scaling, shear, and transposing along either vertical or horizontal axes. In NLP, sentences can be augmented without destroying their meaning in several ways - by replacing words with synonyms, transposing some words, or random insertions and deletions of words. In these examples, even though the transformations may affect a large portion of the data (e.g. every pixel), they do not cause *semantically* large changes to the data. However, directly applying the techniques of NLP to computer vision may not be appropriate - for example random rearrangements of training data pixels may not improve the accuracy of a computer vision system. Similarly, reversing sentence order is unlikely to improve an NLP system. Therefore, when we adapt machine learning techniques to the domain of binary code analysis, we must consider what types of data augmentation are reasonable and will improve our model.

In this work, we compare two types of data augmentation: *semantics-preserving* and *NLP-style*. The *semantics-preserving* type is accomplished in two ways - first, using the Tigress obfuscator on the C source code, and second, by using multiple compilation options to produce multiple object files. The transformations provided by both Tigress and compilers such as GCC or Clang may affect performance, memory usage, or code size, but they should always preserve the meaning of the functions used (assuming no "undefined behavior" is invoked).

We define the *NLP-style* type to include permutations, additions, deletions, and substitutions of object code tokens. These augmentations are performed on disassembled text files prior to using them as training data in our model. This type of augmentation is more appropriate for binary code than true random noise, because random bit flips could cause the program to crash - and malware authors do not want their victim's programs to crash, but rather execute the malicious code without causing alarm.

An important note about the difference between *semantics-preserving* and *NLP-style* augmentations is that the *NLP-style* augmentations explicitly do not preserve the semantics of the original sources. While the resulting code can still be compiled and executed, it no longer performs the same function as the original code, and indeed may crash or corrupt data. We do not suggest that anyone attempts to execute code which has been randomly permuted in this way, because its semantics are undefined.

All of the code generated through the *semantics-preserving* transformations is always compiled, and if executed it retains the semantics of the original source code. We note that some of these obfuscations may increase memory usage, execution time, or both because obfuscation is in some ways antithetical to optimization. However, we define *semantics-preserving* broadly to include programs that have the same function but do not necessarily have identical running time or memory usage.

Finally, we note that the transformations we apply in this work are syntactically valid at every phase. In the *semantics-preserving* phase, transformations occur at the level of C source code. This code is then compiled to binary then disassembled into assembly code. The assembly code is finally transformed by *NLP-style* transformations, which always maintain correct assembly language syntax.

E. Machine Learning Model

In this work we wish to focus on the impact that data augmentation has on a machine learning workflow. To that end we employ the established Asm2Vec[13] model. While we did spend some time optimizing hyperparameters for Asm2Vec, ultimately we kept most of the values the same as described by Ding et. al. Disassembly and training is facilitated by the open source implementation of Asm2Vec called Asm2Vec-pytorch[24]. The Asm2Vec model creates a vector embedding from textual assembly code, so we first preprocess the object files by disassembling them with the reverse engineering tool Radare2[25]. This step of the process transforms the compiled binary code into human-readable assembly code.

Asm2Vec creates an embedding for an assembly function by two methods: *graph* embedding and local *contextual* embedding. A function can be modeled as a graph of *basic blocks*, which are sequences of instructions delimited by `jump` or `call` type instructions. Jump and call instructions can be thought of as directional edges connecting one basic block to another. To obtain a graph embedding, some `call` and `jump` instructions are followed to their destination, resulting in a partial graph of a given function. Rather than exhaustively searching the complete basic block graph, Asm2Vec samples each list of outgoing edges with 3 random walks.

Contextual embedding is achieved by first considering a particular instruction as the current instruction, then examining the instructions immediately before and after the current

instruction. The instructions before and after the current instruction are the "context" for the current instruction. These two instructions are tokenized into their opcodes and operands, and from these tokens an embedding vector is obtained by a method called *paragraph vector distributed memory* (PV-DM)[15], which represents the surrounding context for the current instruction.

The vector embeddings obtained by PV-DM and random walks are combined into a single long vector for each assembly function. Even though different assembly functions have different lengths and different numbers of basic blocks, the vectors generated by Asm2Vec are all 200 units in length. The uniform size facilitates more efficient training using modern hardware such as GPUs.

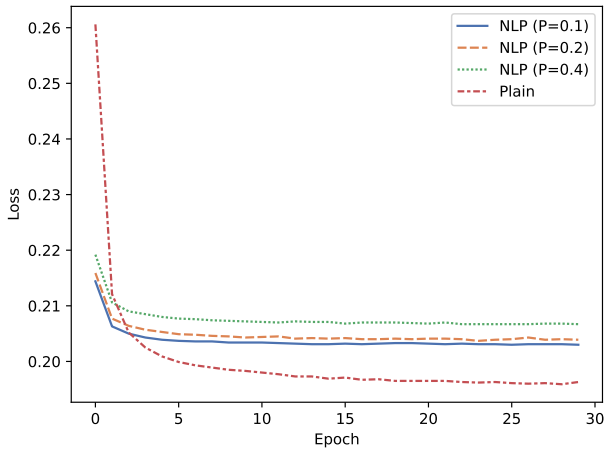

Fig. 3. Training loss for data augmentation

To train this model, a corpus of assembly functions is used as input, and at each iteration the model attempts to predict the *current token* for each instruction in the function. When predicting the current token, a ranked list of candidate tokens is used. If the current token matches the highest ranked candidate, the gradients are unchanged. However, if the current token is in the list of candidates but not ranked highly, the gradients are changed. And finally if the current token is not contained in the list of candidates, the gradients are adjusted more. Predicting the current *token* is a finer grained task than predicting the current *function*.

Because Asm2Vec aims to find binary *clones* rather than *similar* binary functions, its loss function optimizes for predicting sequences of assembly tokens. In this work, we are more interested in *classification* of similar functions. Ding et. al [13] do not publish results for code which was obfuscated by Tigress, due to the low accuracy of their model with this type of obfuscated data. We expand on Ding et. al by using their model on code which has been obfuscated by Tigress in multiple ways.

Figure 3 shows the mean of the binary crossentropy between the predicted and actual tokens across the entire dataset at each epoch during training. This compares multiple loss profiles. Data marked *Plain* uses only the obfuscated disassembled functions, and data marked *NLP* adds copies of these obfuscated functions which are further transformed by a selection of methods inspired by NLP. Each line of text in the disassembled function contains either header information, labels, or opcodes/operands. The first 3 lines of the file contain header information, which we do not modify. We apply textual transformations to augment the existing assembly with additional variations. Those transformations include:

- delete lines
- duplicate lines
- transpose lines

For each line of assembly which is neither a header nor a label, we apply one of these transformations with probability $P = 0.1$, $P = 0.2$, or $P = 0.4$.

As figure 3 shows, data containing any *NLP*-augmented assembly converges more quickly but to a higher loss value than the *Plain* assembly code. Loss increases to ≈ 0.21 as the probability of a transformation increases. Data containing only *Plain* assembly converges more slowly to a loss of ≈ 0.19.

Table II shows excerpts from two disassembled object files, each based on `fprintf`, with the options `-O2` and `-funroll-all-loops` added during compilation. The two columns show the *original* disassembled file on the left, and on the right, a version of that file which has been augmented with *transposed* lines, highlighed in **bold face**. The transposing occured in this example with probability $P = 0.1$. Lines 7 and 26 of the original replace lines 20 and 28 of the transposed version, respectively.

Similarly, for each assembly file, we also use *random deletion* and *random duplication* of lines. In each of these transformations, we leave the first 3 lines of header, as well as `LABEL:` lines unaltered. Interestingly, using data augmented with NLP-inspired techniques makes the training loss worse, which we discuss in the Results section of this work.

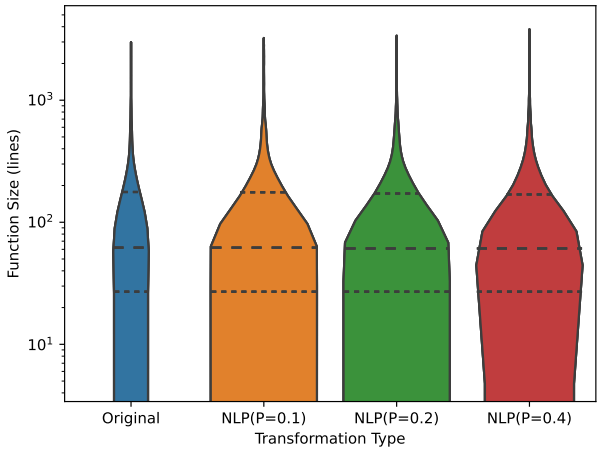

Fig. 4. NLP-style augmentation maintains function size distribution

TABLE II
ASSEMBLY AUGMENTATION BY TRANSPOSING LINES

line	original	transposed
1	.name sym.__fwritex	.name sym.__fwritex
2	.offset 0000000000800b9b0	.offset 0000000000800b9b0
3	.file fprintf-O2-unroll-all-loops.o	.file fprintf-O2-unroll-all-loops.o
4	LABEL102:	LABEL102:
5	cmp r9, rbp	cmp r9, rbp
6	cjmp LABEL11	cjmp LABEL11
7	**mov eax, dword [rbx+CONST]**	mov eax, dword [rbx+CONST]
8	test eax, eax	test eax, eax
9	cjmp LABEL14	cjmp LABEL14
10	mov r11, rbp	mov r11, rbp
11	mov rax, rbp	mov rax, rbp
12	and r11d, CONST	and r11d, CONST
13	LABEL29:	LABEL29:
14	sub rax, CONST	sub rax, CONST
15	cmp byte [r8+rax], CONST	cmp byte [r8+rax], CONST
16	cjmp LABEL11	cjmp LABEL11
17	cjmp LABEL18	cjmp LABEL18
18	lea rax, [rbp+CONST]	lea rax, [rbp+CONST]
19	cmp byte [r8+rax], CONST	cmp byte [r8+rax], CONST
20	cjmp LABEL11	**mov eax, dword [rbx+CONST]**
21	cmp r11, CONST	cmp r11, CONST
22	cjmp LABEL18	cjmp LABEL18
23	cmp r11, CONST	cmp r11, CONST
24	cjmp LABEL25	cjmp LABEL25
25	cmp r11, CONST	cmp r11, CONST
26	**cjmp LABEL27**	cjmp LABEL27
27	cmp r11, CONST	cmp r11, CONST
28	cjmp LABEL29	**cjmp LABEL27**
29	cmp r11, CONST	cmp r11, CONST

For each assembly file, we end up with a total of 4 copies: the original file, as well as *random deletion*, *random duplication* and *random transpose* variants. This means that after these NLP-inspired transformations, the data set could be biased to favor larger assembly language functions. However, while the shortest assembly functions are only a few lines long, which after the *random deletion* transformation can indeed reduce their size by a non-negligible amount, the number of such short functions is small. The width of each sub-plot in Figure 4 corresponds to the number of samples with the number of lines in the file given by the y-axis. The distribution of function sizes is unchanged by NLP-style augmentation, indicating the data set is not biased in terms of function size.

IV. RESULTS

Starting with a single `libc.h` source file, we apply 5 different transformations with Tigress to obtain 5 obfuscated versions of `libc.h`. Next, we include this obfuscated source in each of 14 different test programs (one for each test function). Each of these programs is compiled with diverse options:

- 5 optimization levels: O0, O1, O2, O3, Os
- 5 loop optimizations: unroll-loops, unroll-all-loops, unswitch-loops, loop-optimize, strength-reduce
- 2 C standard versions: c99, gnu99

All of these different options are combined in a Cartesian product as shown in Figure 5, for a total of $5 \times 14 \times 5 \times 5 \times 2 \times 2 = 3500$ different permutations from one source. An important consideration when performing this type of data augmentation is that any bias present in the original data will not necessarily be mitigated by the augmentation. This is analogous to image augmentation for computer vision tasks - if the data set contains only images of cats, no amount of augmentation will help the model recognize dogs. Likewise when augmenting source code via obfuscation and diverse compilation, the semantics of the original source will be replicated in the augmented versions.

```
for libc in (list of libc headers):
 for func in (function names):
  for opt in (optimization levels):
   for loop in (loop optimizations):
    for std in (C standards):
     compile -opt -loop -std=std -c \
      -include libc func.c
```

Fig. 5. Pseudocode for diverse compilation

This method of adding diversity through both obfuscation and diverse compilation options is extremely valuable for this type of source code analysis, because it is difficult to obtain enough different implementations of the source material to satisfy the requirements of a machine learning approach.

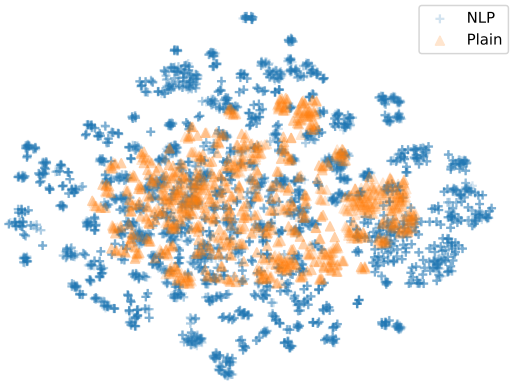

Fig. 6. t-SNE of *Plain* and *NLP*-augmented data

As shown by Figure 6, adding NLP-style augmentation improves clustering performance when used with t-distributed Stochastic Neighbor Embedding (t-SNE). Part of this is simply because the NLP dataset contains 4x more samples (6044 versus 1511). However, the addition of NLP augmentation increases the model's generality and also allows for better discrimination of samples.

Figure 7 shows the t-SNE clustering of the augmented data. There are some clearly visible clusters, especially for the `free` function.

We compare disassembled function embeddings using cosine similarity, in the same manner as Ding et. al in their Asm2Vec work[13]. A major difference between our work and [13] is due to our use of Tigress for obfuscating the C source

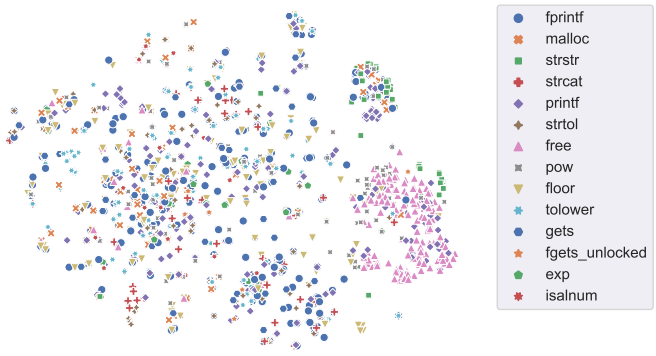

Fig. 7. t-SNE, labeled

code. Asm2Vec did apply obfuscation, but it was performed on the intermediate representation of compiled code only, using LLVM Obfuscator[26]. This work not only obfuscates the C source code, but further transforms the generated assembly code using NLP-inspired techniques.

Generating diverse disassembly results in 6044 assembly files. Because each pairwise comparison takes about 1 second on an 18-core x64 workstation, exhaustivly comparing all (6044× 6044) pairs would take over a year. Instead, we first generate all possible pairs and then sample with probability $P = 1/500$. Finally, we take the average similarity of each pairing to generate Figure 8. Some functions are more consistently similar, such as `pow` and `exp` or `printf` and `fprintf`. The fact that the diagonal of this heatmap does not have consistently high values may be attributed the low sampling rate. For example when row and column are both `fgets` or both `free`, we expect the similarity to be very high but it is only moderately higher than average. The `strtol` function is consistently dissimilar with all other functions.

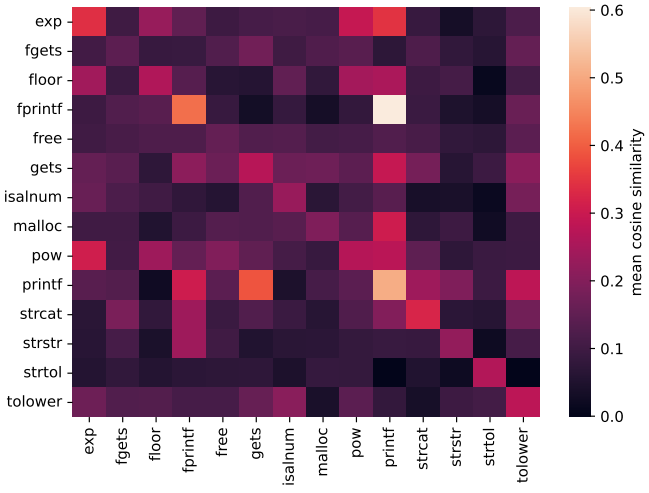

Fig. 8. Mean function similarity

V. Conclusions and Future Work

This work demonstrates two contributions to the field of binary security analysis. First, we describe a method of generating a large amount of data from a single binary source. Second, we show a novel application of data augmentation for binary code.

Word deletion, insertion, and transpositions are data augmentation techniques which previously were used mostly for NLP. One possible enhancement of this work is to leverage models which were trained on larger data, but in different domains. This method of *transfer learning* has proven effective in NLP and other domains, where a model trained on a large data set can then be used with good results on a smaller data set. The challenge for us in using transfer learning would be to determine what existing large models are a good match for code analysis. As noted by Ding et. al[13], natural language data is typically a 1-dimensional stream of tokens, whereas code contains multiple 1-dimensional streams which are explicitly linked to form a graph. As noted by Bayer et. al[10] a potential improvement to this work would leverage a larger pre-trained model. However, a significant challenge of applying a pre-trained model is that most large models are trained on natural language data rather than code, so it is unclear if this would be good or bad for code analysis.

Existing state-of-the-art models in this domain primarily focus on binary clone detection [11], [19], [27], [28]. These applications focus on finding exact or nearly-exact matches in order to prove cases of intellectual property theft or to detect known malware. In the face of obfuscation, the problem becomes much more difficult, because not only are exact binary matches much less likely to occur, but approximate matches may also be rare. This work aims to quantify which types of data augmentation are valid for code. One of our future goals is to build on this initial work and determine which types of data augmentation work best for obfuscated code.

By treating disassembled tokens as words, NLP-based techniques can be applied to any binary code for which a compatible disassembler exists (i.e. all major architectures). We find that NLP-based augmentation of the disassembled code improves clustering performance for t-SNE, but has worse loss characteristics during training.

This work examined only object files compiled for the x86_64 instruction set architecture (ISA). Future work should determine how well these results transfer to other ISAs, especially ARM and MIPS, although we anticipate that the technique should remain effective regardless of the ISA.

Additional research is warranted to determine what types of NLP-based data augmentation works best for assembly code. In this work, the types of changes we made were line-based, and finer-grained changes within lines could also change operands in addition to opcodes.

Finally, this work uses only one type of machine learning model (Asm2Vec). Other NLP-based models are likely also a good fit for the type of disassembly data we use in this work.

We also use only one set of Asm2Vec hyperparameters. This could be further expanded and optimized by hyperparameter tuning to determine whether longer random walks (used for embedding) are more beneficial. Additional graph embeddings are possible using the generalized jump and call instructions in the disassembly code, and we plan to explore these types of embeddings and Graph Neural Network applications in future works.

REFERENCES

[1] R. K. Vaidya, L. De Carli, D. Davidson, and V. Rastogi, "Security issues in language-based sofware ecosystems," *Arxiv preprint arxiv:1903.02613*, 2019.

[2] D.-L. Vu, I. Pashchenko, F. Massacci, H. Plate, and A. Sabetta, "Typosquatting and combosquatting attacks on the python ecosystem," in *2020 ieee european symposium on security and privacy workshops (euros&pw)*, 2020, pp. 509–514.

[3] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards measuring supply chain attacks on package managers for interpreted languages," *Arxiv preprint arxiv:2002.01139*, 2020.

[4] D. Hendler, S. Kels, and A. Rubin, "Detecting malicious powershell commands using deep neural networks," in *Proceedings of the 2018 on asia conference on computer and communications security*, 2018, pp. 187–197.

[5] N. D. Q. Bui, L. Jiang, and Y. Yu, "Cross-language learning for program classification using bilateral tree-based convolutional neural networks," *Corr*, vol. abs/1710.06159, 2017, Available: http://arxiv.org/abs/1710.06159

[6] A. Grover and J. Leskovec, "Node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 855–864.

[7] F. O. Catak, J. Ahmed, K. Sahinbas, and Z. H. Khand, "Data augmentation based malware detection using convolutional neural networks," *Peerj computer science*, vol. 7, p. e346, 2021.

[8] A. K. Biswas, "Cryptographic software ip protection without compromising performance or timing side-channel leakage," *Acm transactions on architecture and code optimization (taco)*, vol. 18, no. 2, pp. 1–20, 2021.

[9] C. Taylor and C. Collberg, "Getting revenge: A system for analyzing reverse engineering behavior."

[10] M. Bayer, M.-A. Kaufhold, B. Buchhold, M. Keller, J. Dallmeyer, and C. Reuter, "Data augmentation in natural language processing: a novel text generation approach for long and short text classifiers," *International journal of machine learning and cybernetics*, pp. 1–16, 2022.

[11] N. Marastoni, R. Giacobazzi, and M. Dalla Preda, "A deep learning approach to program similarity," in *Proceedings of the 1st international workshop on machine learning and software engineering in symbiosis*, 2018, pp. 26–35.

[12] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Proceedings of the aaai conference on artificial intelligence*, 2017, vol. 31.

[13] S. H. Ding, B. C. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *2019 ieee symposium on security and privacy (sp)*, 2019, pp. 472–489.

[14] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space." arXiv, 2013. doi: 10.48550/ARXIV.1301.3781.

[15] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning*, 2014, pp. 1188–1196.

[16] "Musl." https://musl.libc.org.

[17] P. P. Chan and C. Collberg, "A method to evaluate cfg comparison algorithms," in *2014 14th international conference on quality software*, 2014, pp. 95–104.

[18] NSA, "Ghidra." https://ghidra-sre.org/.

[19] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," *Acm sigplan notices*, vol. 51, no. 6, pp. 266–280, 2016.

[20] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "Bingo: Cross-architecture cross-os binary search," in *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering*, 2016, pp. 678–689.

[21] H. Flake, "Structural comparison of executable objects," 2004.

[22] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "Discovre: Efficient cross-architecture identification of bugs in binary code.," in *Ndss*, 2016, vol. 52, pp. 58–79.

[23] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 acm sigsac conference on computer and communications security*, 2016, pp. 480–491.

[24] W.-C. Chao, "Asm2vec-pytorch." https://github.com/oalieno/asm2vec-pytorch.

[25] "Radare2." https://www.radare.org/n/index.html.

[26] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-llvm–software protection for the masses," in *2015 ieee/acm 1st international workshop on software protection*, 2015, pp. 3–9.

[27] A. Saebjornsen, *Detecting fine-grained similarity in binaries*. University of California, Davis, 2014.

[28] B. Liu *et al.*, "Alphadiff: cross-version binary code similarity detection with dnn," in *Proceedings of the 33rd acm/ieee international conference on automated software engineering*, 2018, pp. 667–678.