Detecting Standard Library Functions in Obfuscated Code

Alexander Shroyer[0000−0002−6291−3994] and D. Martin Swany

Indiana University, Bloomington IN 47405, USA
{ashroyer,swany}@iu.edu

Abstract. Binary analysis helps find low-level system bugs in embedded systems, middleware, and Internet of Things (IoT) devices. However, obfuscation makes static analysis more challenging. In this work we use machine learning to detect standard library functions in compiled code which has been heavily obfuscated. First we create a C library function dataset augmented by obfuscation and diverse compiler options. We then train an ensemble of Paragraph Vector-Distributed Memory (PV-DM) models on this dataset, and combine their predictions with simple majority voting. Although the average accuracy of individual PV-DM classifiers is 68%, the ensemble is 74% accurate. Finally, we train a separate model on the graph structure of the disassembled data. This graph classifier is 64% accurate on its own, but does not improve accuracy when added to the ensemble. Unlike previous work, our approach works even with heavy obfuscation, an advantage we attribute to increased diversity of our training data and increased capacity of our ensemble model.

Keywords: Internet of Things · Data Science · Neural Networks

1 Introduction

Unlike dynamically linked applications which share library code with other applications, statically linked binaries bring their own libraries. Static linking has multiple advantages. It makes applications convenient for users to install. It also gives software developers full control over library code used by their application, and allows them to include patched libraries for custom extensions, or older versions of libraries for backwards compatibility. However, this flexibility also means security researchers must examine statically linked library code once per *application*, whereas code shared by dynamically linked applications only needs to be analyzed once per *system*.

Binary analysis techniques can detect malware in either dynamically or statically linked binary files [1, 2], but malware authors consistently try to thwart such analysis. One method of hiding malware is to strip[1] symbols and replace meaningful names with less meaningful symbols or numbers after compilation. Another method involves obfuscating source code before compilation. Extensions

[1] `https://linux.die.net/man/1/strip`

to mainstream compilers like LLVM [3] provide obfuscation as a compilation option [4], which makes distributing an obfuscated version of a binary no more difficult than distributing one without obfuscation.

After years of progress by reverse engineers and security researchers, obfuscation is still a challenge for modern static analysis [5–7], and some forms are "provably hard for any static code analyzer to overcome" [8]. Yet despite its shortcomings, static analysis remains valuable in part due to its low cost to implement. Because it never executes potentially malicious code, static analysis requires no special run-time environment. This contrasts with dynamic [9] or hybrid [10] analysis types which require isolated test environments and are therefore more costly.

In this work, we propose a static analysis approach to classify known functions in obfuscated and stripped binaries. Our method uses a general purpose machine learning model on a dataset prepared using domain-specific knowledge and tools, and attains accuracy up to 74%. Meanwhile, previous state of the art approaches like Asm2Vec [11] perform well on binary code whose sources were not obfuscated, but poorly when sources were heavily obfuscated.

Research Summary: Our method is robust to source obfuscation, which represents an improvement over previous approaches. To demonstrate this technique, we classify C standard library functions. Even though such functions are typically compact in their source code representation, they appear frequently in real programs, and IDA Pro's FLIRT [12] plugin demonstrates that there is commercial demand for this task. Unfortunately, malware authors can circumvent analysis by spoofing FLIRT signatures [13], which limits the utility of FLIRT as an analysis tool.

Our model-based approach is not susceptible to this type of spoofing attack because it does not rely on precisely matching signatures of function data, representing an improvement which could be useful in commercial security analysis products. The goal of this research is to improve the effectiveness of static binary analysis by reliably detecting known functions within an unknown binary. This lets security researchers spend more time dissecting *unknown* functions. Ultimately, our goal with this work is to improve the speed and quality of the practice of binary analysis.

The remaining sections of this paper are as follows: First, a Related Work section summarizes the state of the art in this domain and outlines some limitations of those approaches. Next, we describe in a Methods section the details of our approach. A subsequent Discussion section provides detail on our experimental approach and its limitations. After this, a Results section illustrates how our approach fares in our experiments. Finally, we detail our Conclusions and Future Work to reflect on the merits of our approach and what enhancements may exist for this approach.

2 Related Work

An important inspiration for our work is a technique called **Asm2Vec** [11]. The Asm2Vec model takes inspiration from **Word2Vec** [14], which in turn uses a Paragraph Vector - Distributed Memory (PV-DM) technique [15]. PV-DM is usually applied to natural language processing (NLP), but **Asm2Vec** shows it is feasible to treat assembly code as "words" for such a model. Like **Asm2Vec** and **InnerEye** [16], we disassemble the binary and tokenize the resulting assembly code, but our preprocessing step preserves the distinct labels. However, current related work either is not specifically focused on finding library functions, or did not consider obfuscated source material. We use an obfuscated training corpus, which allows our model to classify heavily obfuscated functions that were impossible for approaches like **Asm2Vec**. Our approach attempts to fill the gap in the current research by focusing on library function detection while tolerating a high degree of obfuscation. In addition, instead of discriminating between malware and non-malware, our approach aims to classify individual functions. Table 1 summarizes some of the state-of-the-art approaches in relation to ours. In this table, the term *Obfuscation Tolerance* means the method performed to a high standard on obfuscated code.

Table 1: Comparison of Current Approaches

Method	Purpose	Obfuscation Tolerance
Asm2Vec [11]	malware detection, function similarity	limited
InnerEye [16]	cross architecture basic block similarity	unknown
E Unibus Pluram [17]	defensive obfuscation	yes
AlphaDiff [18]	cross-version similarity	no
Structure2Vec [19]	scalable data representation	unknown
Catak et. al [20]	malware detection	unknown
Yu et. al [21]	data augmentation	unknown
Marastoni et. al [22]	binary similarity	yes
FLIRT [12]	library function detection	no
Qiu et. al [23]	library function detection	unknown
Ours	library function detection	yes

Recently, Yu et. al [21] explore augmenting source code by semantics-preserving transforms. However, their work explicitly tries to preserve the meaning and readability of the source code, not obfuscate it. In [24], Mi et. al use transformations such as variable renaming and modifying commentary to produce augmented variants of source code in order to classify the code as *readable* or not. Modifying source code comments has no affect on its run-time semantics, but renaming variables would be considered a trivial or minor form of obfuscation.

Franz et. al suggest diverse compilation as a defensive technique in their 2010 paper [17]. They propose generating semantically equivalent but syntactically different versions of a program such that each user receives a unique version of the same program, thus limiting the scope of any specific attack or vulnerability exploit.

Because malware detection is a common goal in this domain, researchers commonly employ a Siamese network [18, 19] to train a loss function to determine which files are malware. While effective for classifying a given binary file as either malicious or benign, this approach is less suitable for recognizing specific subroutines or functions.

Data augmentation is often used for text classification as in [25]. Catak et. al [20] used data augmentation for malware detection, by injecting random noise into their data samples. Marastoni et. al reshaped 1-dimensional binary files into two-dimensional images, in order to use Convolution Neural Network (CNN) techniques [22] originally developed for computer vision applications.

Qiu et. al define library functions as those whose "instruction sequence and semantics are known". They also point out the value in identifying these types of functions because analyzing them would be a waste of time [23]. Their technique relies on constructing what they term an "execution dependence graph" in order to capture the semantics of the analyzed code.

Finally, graph reduction techniques have seen recent use [26] to train graph neural networks more efficiently. This approach shrinks the overall size of the graphs while retaining some of their characteristics, such as relative ordering of nodes or average out-degree. Alternative graph reduction techniques exist such as [27] which preserve spectral properties of the graph. These reduction techniques have the potential to enhance our work by enabling faster comparisons on simpler graphs. Embedding techniques such as GraphSAGE [28] allow more efficient low-dimensional embeddings of large amounts of graph data. This technique may prove useful as our data sizes increase. DeepWalk [29] is a method for embedding graph data in a lower-dimensional vector space that is particularly effective when labeled data is sparse. Our use case also deals with sparse labeled data, so this type of embedding method is potentially useful when preparing our dataset.

3 Methods

Data augmentation techniques from computer vision and natural language processing do not necessarily apply to binary code, because even small changes to the code can result in unrelated semantics or invalid code. By using *obfuscation* and diverse *compiler options*, we can generate alternative versions of a given function without dramatically changing its semantics. We use the obfuscation tool **Tigress** [30, 31] and the **gcc** compiler[2]. Initially we select at random a set of functions from the `musl` C standard library[3]. However, some newer C language features used by the `musl` authors are not supported by the older C parser used by Tigress.

Table 2 shows the 31 remaining compatible functions. We then derive multiple new C sources from each function using Tigress and different compiler options.

[2] `https://gcc.gnu.org/`

[3] `https://musl.libc.org`

Table 2: Functions of interest (compatible with Tigress)

abs	acos	asin	atan2	ceil	cos	daemon
exp	floor	fread	inet_addr	inet_aton	isalnum	kill
memccpy	memcmp	memmem	readdir	signal	sin	stpcpy
stpncpy	strchr	strcpy	strncpy	strstr	strtok	tan
vfprintf	wait4	waitpid				

3.1 Data Augmentation Through Obfuscation

For this work, we use Tigress to augment both the size and diversity of our dataset. Given the source code for a C library function such as `abs` (Figure 1), we transform the source code file into approximately 345 new source code files, each based on different combinations of Tigress transformations and compiler options.

```
int abs(int a) { return a>0 ? a : -a; }
```

Fig. 1: C source code for `abs` function

Figure 2 compares and contrasts different some of the types of obfuscation. First we show the normalized assembly code for `abs` without obfuscation. Next is `abs` obfuscated with the `Split` transformation, which splits a single function into multiple parts. Finally we show `abs` obfuscated with both `EncodeArithmetic` and `Flatten` transformations. This demonstrates the diversity provided by obfuscation, allowing even a short function like `abs` to be effectively augmented.

```
endbr64
mov eax, edi
neg eax
cmovs eax, edi
ret
```

(a) No obfuscation

```
    endbr64
    mov eax, edi
    cdq
    mov ecx, edx
    xor ecx, edi
    sub edx, ecx
    xor edx, edi
    cjmp LABEL7
    neg eax
LABEL7:
    ret
```

(b) EncodeArithmetic, Flatten

```
    endbr64
    sub rsp, CONST
    mov dword [rsp + CONST], edi
    test edi, edi
    cjmp LABEL4
    lea rsi, [rsp + CONST]
    lea rdi, [rsp + CONST]
    call LABEL7
LABEL7:
    jmp LABEL8
LABEL4:
    mov eax, edi
    neg eax
    mov dword [rsp + CONST], eax
LABEL8:
    mov eax, dword [rsp + CONST]
    add rsp, CONST
    ret
```

(c) Split

Fig. 2: Selected `abs` assembly code variations

Because Tigress is a source-to-source transformer, it permits sequential transformations for even greater diversity. Some obfuscation types are more relevant for intentional obfuscation than others, particularly those which are proven to be NP-complete such as the control flow alterations described by [32]. One malicious obfuscation technique hides itself from IDA Pro's FLIRT function detector

[13]. In this attack, malware authors construct malicious code which matches one of FLIRT's library function signatures, causing FLIRT to mark the malicious code as benign. The existence of this attack demonstrates a vulnerability of signature-based static analysis. Basic obfuscation such as encoding literal values or inserting no-op codes are easier for traditional static analysis than more advanced obfuscation which alters control flow.
All Tigress transformations we use preserve enough of the original semantics such that the code compiles without errors or warnings. However, the resulting binary file may not exactly match the original semantics, especially in terms of performance. Despite these slight differences, obfuscation is a more realistic form of data augmentation than inserting random noise into the binary file as in [20]. We choose combinations of the following Tigress transformations[4]: (Flatten, Split, EncodeArithmetic, Virtualize, AntiAliasAnalysis, EncodeLiterals, InitEntropy, InitOpaque). These transformations cover a spectrum from "basic" to "advanced" obfuscation types. EncodeArithmetic and EncodeLiterals are "basic" transforms which do not disrupt control flow. Split, Flatten, and Virtualize are more "advanced" transforms that do affect control flow and are traditionally pose a challenge for static analysis.
Our work uses each of these transformations individually to produce one set of outputs, and then produces further outputs by forming sequences of up to three transforms. Some example sequences of transforms include:

- (EncodeLiterals → EncodeLiterals → Flatten)
- (Flatten → Virtualize → AntiAliasAnalysis)
- (Virtualize → Flatten)
- (Flatten → AntiAliasAnalysis → Flatten)
- (Split → Split → Split)

In addition to obfuscation, we also use *compiler options* to add diversity. We keep all compiler options originally used by `musl`, and add more of our own to obtain additional binary files. We collect groups of these options and apply an entire group during compilation:

Loops -floop-parallelize-all, -ftree-loop-if-convert, -funroll-all-loops, -fsplit-loops, -funswitch-loops
Codegen -fstack-reuseall, -ftrapv, -fpcc-struct-return, -fcommon, -fpic, -fpie
Safety -fsanitizeaddress, -fsanitizepointer-compare, -fsanitizeundefined, -fsanitize-address-use-after-scope
Optimize -O3 (overrides -02 default value)

Each Tigress transformation is then compiled with one of these option groups at most, resulting in permutations of Tigress transforms and option group values such as (AntiAliasAnalysis → Virtualize → Flatten → **Loops**) and (Virtualize → EncodeArithmetic → Split → **Optimize**). This results in 1290 different combinations of transformations and options, 358 of which result in syntactically valid C code for at least some of our functions. We then discard any C code

[4] `https://tigress.wtf/transformations.html`

generated by Tigress which fails to compile. Finally, we apply each combination to each of our functions of interest, resulting in a total training set size of **9414** object files.

3.2 Data Preprocessing

For each compiled object file, we must transform it into a format usable by our models. This entails disassembling the object file into assembly code. This assembly code is further normalized by removing header information, replacing numeric values with a single token (`CONST`), and renaming all numeric offsets as generic labels, such as `LABEL5` or `LABEL194`. These two steps serve to *strip* the code of any identifying information, similar to using a linker's "strip symbols" option or the `strip` command [1].
The sections punctuated by `LABEL:` tokens are *basic blocks*, essentially linear sequences of instructions. If we again refer to the `Split` assembly code from Figure 2, we can describe its graph structure as in Figure 3.

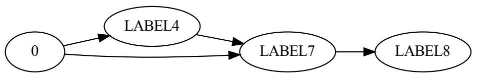

Fig. 3: Graph representation of `Split` from Figure 2.

The majority of this normalization step is provided by an open source implementation of **Asm2Vec** available from GitHub [33]. This assembly representation has replaced numeric offsets with `LABEL<N>` keywords, where N is a relative offset within the given file. There are also 3 lines of header at the beginning (`.name sym.abs`, `.offset 000000000800004e`, `.file abs.o`) in the original assembly output file. These lines identify the function explicitly, so we remove them from both our training and testing datasets, and they are not shown in the examples used here for clarity. Finally, we separate all tokens by whitespace and remove newlines, so the resulting document is a single line, part of which is shown in Figure 4.

```
endbr64 sub rsp , CONST mov dword [rsp + CONST] , edi test
```

Fig. 4: Normalized assembly tokens

Each assembly file is now a single line of space-separated tokens, which we then combine into one multi-line *corpus* text file. This file contains neither function identifiers nor absolute numeric offsets, so we consider it equivalent to a stripped binary file. The corpus used in this work encodes **9414** documents in just **31MB** - comparable in size to approximately 10 photos from a high resolution smartphone camera. We mention the size of this dataset to illustrate that even this

uncompressed plain text encoding is "small data" by today's standards, and this suggests that much larger training data sizes (1-2 orders of magnitude larger) are feasible for future work without requiring any change to the approach.
To measure the performance of our unsupervised model, we also store the name (label) of each function in a separate file, with one function name per line. The line containing the function name is the same line at which the corresponding function's tokens appear in the corpus. We later use this correspondence to validate the accuracy of the model, but because this is an unsupervised model, at no point in training or testing is the model exposed to these labels.

3.3 PV-DM Voting Classifier

We use Gensim's "Doc2Vec" implementation of PV-DM [5]. PV-DM contrasts with the Distributed Bag of Words (DBOW) approach by accounting for the order in which the words occur. This model aims to maximize the average log probability that word w_t appears in a sequence of training words $w_1, w_2, ..., w_T$ within a window of size T using the objective function:

$$\frac{1}{T}\sum_{t=k}^{T-k} log\ P(w_t|w_{t-k}, ..., w_{t+k})$$

The hyperparameters for this PV-DM model are: **embedding size**: 400, **window size**: 10, **min count**: 10, **epochs**: 40. These are approximately the same as those recommended by [15]. However, we use 40 epochs instead of the recommended 10 due to our smaller training dataset. After training this model using a shuffled 10-fold cross validation split, discrete assembly language tokens are *embedded* into a lower dimensional vector space.

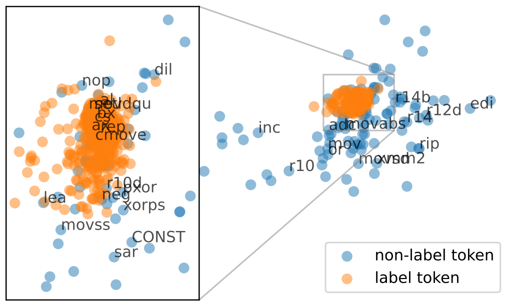

Fig. 5: First 2 principal components of **Word2Vec** embedding (with labels)

To visualize this embedding, we plot the first two principal components as shown in Figure 5. We annotate a subset with the token's name, to illustrate which

[5] `https://radimrehurek.com/gensim/models/doc2vec.html`

tokens occupy similar regions of the embedding. Tokens in the `LABEL` category are not annotated, to avoid cluttering the visual. The most interesting aspect of this visualization is that labels (in orange) cluster near one another.

3.4 Graph Metadata Classifier

While the PV-DM model alone is capable of approximating some assembly language semantics, assembly code also contains *graph* structure, which models might use to more accurately classify functions. This graph structure is due instructions like `jump` or `call` making directed edges which terminate at `labels`. We partition sequences of tokens into nodes by splitting at every `LABEL<N>:` type token. This simplistic partitioning is coarser than a control flow graph because it ignores conditional jumps. Some types of obfuscation, such as Tigress' `EncodeLiterals` transformation, have minimal or no effect on this graph structure. Others, such as `Flatten` and `Split`, explicitly add or remove control flow, resulting in changes to the graph structure.
We reduce the complexity of these graphs using standard graph algorithms: **graph condensation** and **minimum spanning tree**, with implementations provided by the **graph-tool** software[6]. Graph condensation combines nodes if they form a strongly connected component and allows us to prune duplicate edges. For obfuscated code graphs, reducing the complexity in this way may approximate the shape of graph of the function before obfuscation. In practice, the reduced graph is not necessarily isomorphic to the original, and because these graphs only represent the *structure* of the code, they can not be checked for validity.
Figure 6 shows the effect of reducing an obfuscated graph using standard graph techniques. In each sub-figure, we show an example graph with node sizes weighted by the number of tokens present in the original basic block, and edge thickness proportional to the graph's betweenness centrality. Repeated application of graph condensation and filtering by the minimum spanning tree provides an extreme reduction in the graph's complexity.
We select an arbitrary set of graph properties as a feature vector (**average edge betweenness**, **average vertex size**, **vertex degree histogram**, and **vertex degree** normalized by **vertex size**) and then train a random forest classifier to predict the class of the function based on this vector. For each of these properties, our feature vector consists of its **mean** and its **standard deviation**, for a total of 8 features per graph.

4 Discussion

Unlike related work in this area which may involve obfuscated source code, our approach focuses specifically on the identifying library functions. While these functions are often small, they are used frequently in application code by both

[6] `https://graph-tool.skewed.de/`

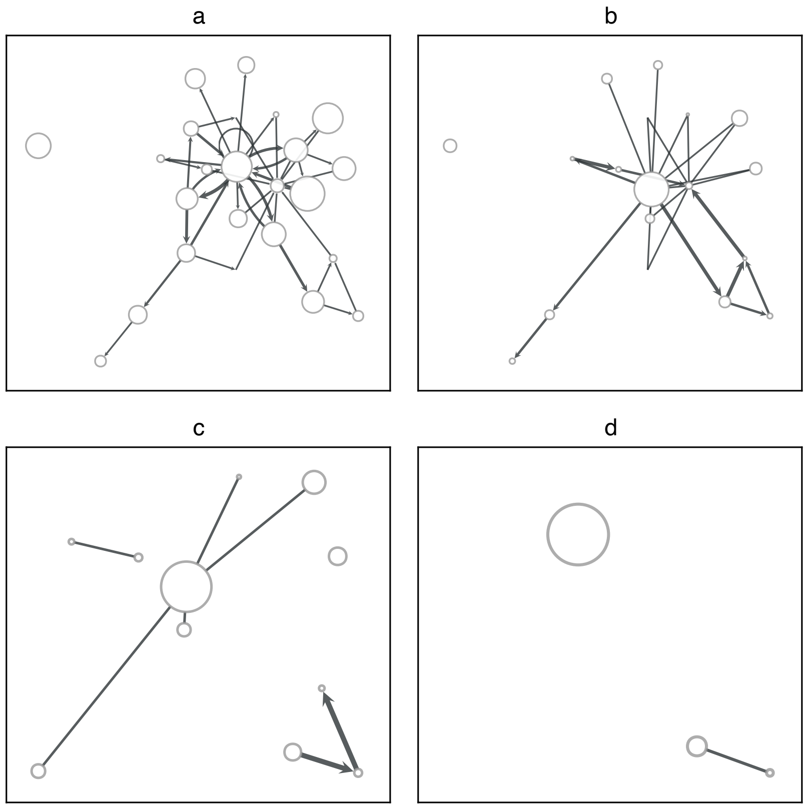

Fig. 6: **Graph reduction**. **(a)** shows the original graph; **(b)** condenses strongly connected components and self-loops as per [34]; **(c)** condenses as in (b) and also prunes edges which are not part of the minimum spanning tree; **(d)** applies (c) twice. Each view preserves the relative positioning from the original graph, scaled to fit the new graph.

benign and malicious developers. Identifying library functions quickly and accurately aids in the overall task of binary analysis, because once those functions are identified, analysis can proceed to the more interesting aspects of a particular section of code, such as finding malware or stolen intellectual property.
In the case where binary code is dynamically linked with system libraries, our method has little to offer, because those libraries can be analyzed just one time even if there are many dynamically linked applications that use them. However, if desired our approach could be used to analyze an existing library. If such analysis reveals an unknown function, this could imply a library function missing from our model's dataset, or possibly a library function which is implemented in an unusual manner. It is reasonable to audit shared library code closely, because a vulnerability in a shared resource impacts many other applications.
Unlike related work which uses data augmentation techniques inspired by computer vision [20, 22], our method attempts to preserve the semantics of the source material. Automated obfuscation using a tool like Tigress [30, 31] is repeatable and testable, permitting verification of program semantics both before and after transformation. However, in this work we do not run any behavioral tests of the code either before or after obfuscation, instead we rely on the less strict measure of compilation success. We only include in our method those transformations that result in source code that can compile without any errors.
One hypothesis which we examine in this work is whether graph metadata affects the outcome of a machine learning model. Compared to PV-DM models, graph-based models convey relationships between parts explicitly and succinctly. Therefore, we expect that the addition of graph attributes to a model should improve its accuracy at the cost of increased complexity during data preprocessing, because obtaining these graph characteristics requires more domain knowledge of the assembly code.

5 Results

Table 3: Individual model accuracy on validation set

model	0	1	2	3	4	5	6	7	8	9	**mean**
$\frac{correct}{total}$	0.68	0.68	0.65	0.69	0.69	0.68	0.7	0.69	0.68	0.69	**0.68**

Table 3 summarizes the accuracy of each of the individual classifiers. To measure accuracy, we take a simple count of all correct predictions, and divide by the total number of predictions. Each of the 10 models is 68% accurate on average. But because each of the 10 sub-models trained on a *different* 90% of the original corpus, voters strong in one area can compensate for other voters weak in that

area, resulting in a majority voting ensemble that achieves 100% accuracy on the training corpus.
A more realistic measure of voting classifier accuracy is predicting the correct class for a function implementation which was never used in training or testing. To test this, we generate a small dataset with a different set of Tigress transformations (`Split` then `Flatten`) and compilation options (`-O2`). While the functions themselves are still from `musl`, these compiled binaries are new to our models.

Table 4: Classifier results: new data

actual	voting prediction (74%)	graph prediction (64%)	graph override (74%)
abs	**abs**	**abs**	**abs**
acos	inet_addr	abs	abs
asin	acos	**asin**	**asin**
atan2	**atan2**	strtok	**atan2**
ceil	floor	**ceil**	floor
cos	sin	asin	asin
daemon	**daemon**	abs	**daemon**
exp	sin	tan	sin
floor	**floor**	**floor**	**floor**
fread	**fread**	**fread**	**fread**
inet_addr	**inet_addr**	**inet_addr**	**inet_addr**
inet_aton	**inet_aton**	**inet_aton**	**inet_aton**
isalnum	**isalnum**	**isalnum**	**isalnum**
kill	**kill**	**kill**	**kill**
memccpy	**memccpy**	strstr	**memccpy**
memcmp	**memcmp**	**memcmp**	**memcmp**
memmem	**memmem**	**memmem**	**memmem**
readdir	**readdir**	**readdir**	**readdir**
signal	**signal**	**signal**	**signal**
sin	**sin**	asin	asin
stpcpy	strncpy	inet_aton	strncpy
stpncpy	**stpncpy**	**stpncpy**	**stpncpy**
strchr	**strchr**	**strchr**	**strchr**
strcpy	strncpy	**strcpy**	strncpy
strncpy	**strncpy**	strchr	**strncpy**
strstr	**strstr**	**strstr**	**strstr**
strtok	**strtok**	**strtok**	**strtok**
tan	**tan**	memcmp	**tan**
vfprintf	acos	asin	acos
wait4	**wait4**	**wait4**	**wait4**
waitpid	**waitpid**	**waitpid**	**waitpid**

Table 4 shows the results for these new function implementations. The simple accuracy of the voting ensemble for unseen data is **74%**, and Cohen's kappa coefficient [35] is **0.73**. In this table, correct predictions by the classifiers are in

bold face. Figure 7 shows the predicted versus actual label for the test set (10% of total) using a Random Forest classifier on the graph metadata. The overall accuracy of this classifier using the original graph structure is **64%**. However, we still see **64%** accuracy with this model when analyzing the **reduced** graph data. Reducing the graph structure using the methods of Figure 6 has **no impact** on accuracy, but uses less memory. As datasets grow larger, this reduction technique may become essential.

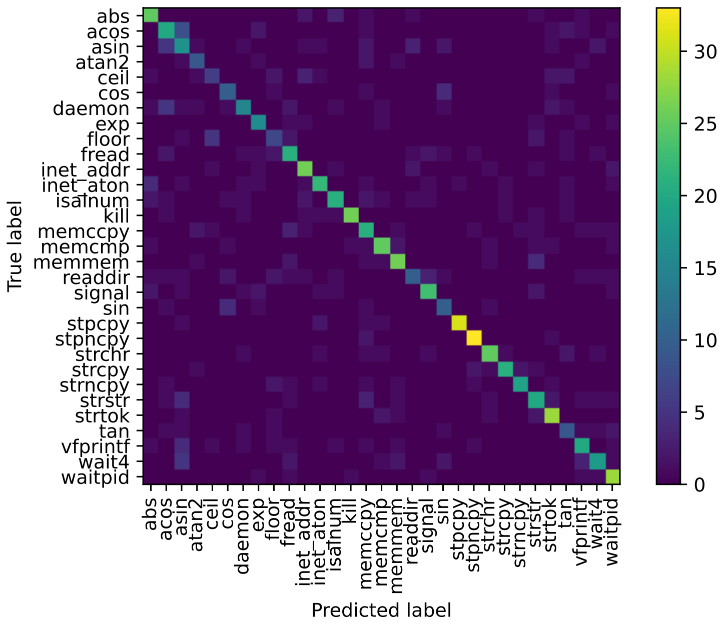

Fig. 7: Confusion matrix for graph metadata classifier (test data)

The graph metadata classifier accuracy is still acceptable at **67%** when presented with new data for validation. Figure 8 shows predicted versus actual labels for this small validation dataset. We combine these two classifiers by **overriding** the voted prediction with the graph prediction when the voting classifier was "uncertain", based on a simple heuristic (more than 4 different votes and the winner received less than 5 votes). With this simple heuristic, the combined voting/graph classifier remains **74%** accurate. Contrary to our expectations, the addition of this graph metadata did not improve the overall quality of result.
Compared to **Asm2Vec**, our work succeeds at classifying highly obfuscated code, as illustrated in Table 5.

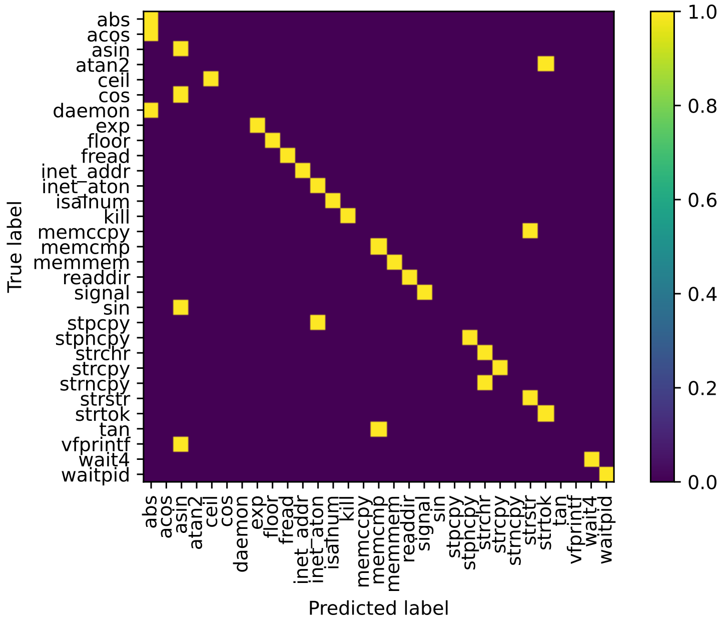

Fig. 8: Confusion matrix for graph metadata classifier (validation data)

Table 5: Comparison with Asm2Vec

	EncodeLiterals	Virtualization	JIT	3 transformations
Asm2Vec	**92.7%**	35%	45%	0%
ours	74%	**74%**	untested	**74%**

6 Conclusions and Future Work

We demonstrate source code augmentation through obfuscation suitable for training different types of machine learning models. Our voting classifier model classifies new variations of given functions with 74% accuracy, whereas a graph-metadata based classifier classifies this same validation data with 64% accuracy. Combining the predictions of both classifiers with a simple voting heuristic neither improves nor degrades this 74% accuracy. This approach tolerates obfuscated source code and stripped binary files, unlike **Asm2Vec**, whose authors note that "after applying three obfuscation techniques at the same time, **Asm2Vec** can no longer recover any clone". Our approach can not only discriminate between similar versus dissimilar samples, but also classifies multiple functions with accuracy exceeding that of **Asm2Vec** when the functions were subject to obfuscation.

```
    endbr64
    push r13
    mov eax, CONST
    push r12
    push rbp
    lea rbp, [rip]
    push rbx
    sub rsp, CONST
    movsd qword [rsp + CONST], xmm0
    lea r13, [rsp + CONST]
    lea r12, [rsp + CONST]
LABEL12:
    cmp rax, CONST
    cjmp LABEL12
    movsxd rdx, dword [rbp + rax*CONST]
    add rdx, rbp
    jmp rdx
    ret
```

(a) acos (actual)

```
    endbr64
    sub rsp, CONST
    mov qword [rsp + CONST], rdi
    lea rdx, [rsp + CONST]
    lea rsi, [rsp + CONST]
    lea rdi, [rsp + CONST]
    call LABEL6
LABEL6:
    mov eax, dword [rsp + CONST]
    test eax, eax
    cjmp LABEL9
    mov eax, dword [rsp + CONST]
    add rsp, CONST
    ret
LABEL9:
    mov eax, CONST
    add rsp, CONST
```

(b) inet_addr (predicted)

Fig. 9: Mistaken function example

Our approach has some limitations, however, and there is room for improvement. Not all Tigress transformations were successful in producing output, a problem also encountered by the **Asm2Vec** authors. We were able to work around some of these errors by carefully crafting the commands passed to the compiler, but some such errors remain. These are due to the `musl` library's usage of newer C syntax features not yet supported by Tigress' C parsing engine. In these cases, a particular function was not transformed by a particular Tigress transformation, and so that function-transformation pair was not included in our dataset. Manual fixes limit the generality of this approach because they require familiarity with the build system. This is not a problem for in-house analysis, but it could be a challenge for analyzing unknown code.

We also note that there are some examples in the unseen data which our voting classifier never guessed correctly (e.g. `vfprintf`, `acos`, and `strcpy`). With the `strcpy` sample, the classifier predicted `strncpy(6) stpcpy(3) stpncpy`. We

believe these predictions are good approximate matches to the `strcpy` function, and would therefore be valuable to a human using our approach as a reverse engineering aid. The predictions for `vfprintf` (`asin(5)` and `acos(5)`) are not close at all, and would be misleading to a human. This instance of `vfprintf` happens to contain no labels or jumps, which could indicate a limitation of our data augmentation strategy. The `acos` example was mistaken for `inet_addr`, so we compare them side-by-side in Figure 9. These two samples do not have any major similarities other than length, so more analysis is needed to determine how the model can better handle situations like these. Figure 10 shows the distribution of basic blocks in our data before and after the graph condensation and pruning. This distribution is acceptable for analysis, because such skew is likely to occur in real-world code as well as synthesized code, even though library functions in particular tend to have fewer nodes and edges.
In future work we plan to evaluate the effectiveness of graph coarsening in contrast to the simpler graph reduction we applied in this work. Coarsening methods such as these preserve spectral properties of the original graphs, unlike our minimum spanning tree and condensation approach, and may better preserve the structure of small graphs. We also plan to incorporate graph embedding and recent graph neural network techniques to increase the effectiveness of our classifiers. Finally, while the workflow we developed so far is useful for detecting specific functions within an obfuscated binary, we plan to further analyze whether it can distinguish between functions it has never seen in **any** form versus functions it has seen in a modified form.

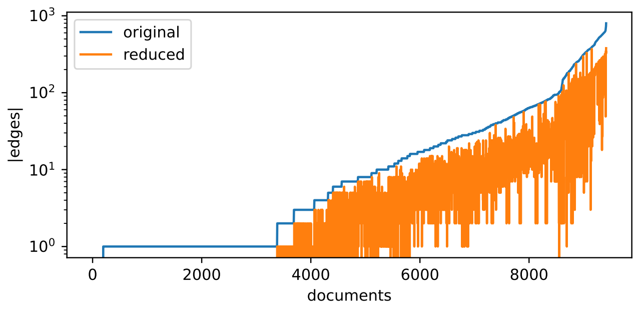

Fig. 10: Jump-Label distribution

References

1. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: Sok: (state of) the art of war: Offensive techniques in binary analysis. (2016).

2. Pewny, J., Garmany, B., Gawlik, R., Rossow, C., Holz, T.: Cross-architecture bug search in binary executables. In: 2015 ieee symposium on security and privacy. pp. 709–724. IEEE (2015).
3. Lattner, C., Adve, V.: Llvm: A compilation framework for lifelong program analysis & transformation. In: International symposium on code generation and optimization, 2004. cgo 2004. pp. 75–86. IEEE (2004).
4. Junod, P., Rinaldini, J., Wehrli, J., Michielin, J.: Obfuscator-llvm–software protection for the masses. In: 2015 ieee/acm 1st international workshop on software protection. pp. 3–9. IEEE (2015).
5. Schrittwieser, S., Katzenbeisser, S., Kinder, J., Merzdovnik, G., Weippl, E.: Protecting software through obfuscation: Can it keep pace with progress in code analysis? Acm computing surveys (csur). 49, 1–37 (2016).
6. Wagner, R.: Modern static analysis of obfuscated code. In: Proceedings of the 3rd acm workshop on software protection. p. 1 (2019).
7. Singh, J., Singh, J.: Challenge of malware analysis: malware obfuscation techniques. International journal of information security science. 7, 100–110 (2018).
8. Christodorescu, M., Jha, S.: Static analysis of executables to detect malicious patterns. In: 12th usenix security symposium (usenix security 03) (2003).
9. Egele, M., Woo, M., Chapman, P., Brumley, D.: Blanket execution: Dynamic similarity testing for program binaries and components. In: 23rd usenix security symposium (usenix security 14). pp. 303–317 (2014).
10. Udupa, S.K., Debray, S.K., Madou, M.: Deobfuscation: Reverse engineering obfuscated code. In: 12th working conference on reverse engineering (wcre'05). p. 10–pp. IEEE (2005).
11. Ding, S.H., Fung, B.C., Charland, P.: Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In: 2019 ieee symposium on security and privacy (sp). pp. 472–489. IEEE (2019).
12. Guilfanov, I.: Ida fast library identification and recognition technology (flirt technology): In-depth, (2012).
13. McMaster, J.: Issues with flirt aware malware, (2011).
14. Mikolov, T., Chen, K., Corrado, G., Dean, J.: Efficient estimation of word representations in vector space, `https://arxiv.org/abs/1301.3781`, (2013). `https://doi.org/10.48550/ARXIV.1301.3781`.
15. Le, Q., Mikolov, T.: Distributed representations of sentences and documents. In: International conference on machine learning. pp. 1188–1196. PMLR (2014).

16. Zuo, F., Li, X., Young, P., Luo, L., Zeng, Q., Zhang, Z.: Neural machine translation inspired binary code similarity comparison beyond function pairs. Arxiv preprint arxiv:1808.04706. (2018).
17. Franz, M.: E unibus pluram: massive-scale software diversity as a defense mechanism. In: Proceedings of the 2010 new security paradigms workshop. pp. 7–16 (2010).
18. Liu, B., Huo, W., Zhang, C., Li, W., Li, F., Piao, A., Zou, W.: Alphadiff: cross-version binary code similarity detection with dnn. In: Proceedings of the 33rd acm/ieee international conference on automated software engineering. pp. 667–678 (2018).
19. Dai, H., Dai, B., Song, L.: Discriminative embeddings of latent variable models for structured data. In: International conference on machine learning. pp. 2702–2711. PMLR (2016).
20. Catak, F.O., Ahmed, J., Sahinbas, K., Khand, Z.H.: Data augmentation based malware detection using convolutional neural networks. Peerj computer science. 7, e346 (2021).
21. Yu, S., Wang, T., Wang, J.: Data augmentation by program transformation. Journal of systems and software. 190, 111304 (2022).
22. Marastoni, N., Giacobazzi, R., Dalla Preda, M.: A deep learning approach to program similarity. In: Proceedings of the 1st international workshop on machine learning and software engineering in symbiosis. pp. 26–35 (2018).
23. Qiu, J., Su, X., Ma, P.: Library functions identification in binary code by using graph isomorphism testings. In: 2015 ieee 22nd international conference on software analysis, evolution, and reengineering (saner). pp. 261–270. IEEE (2015).
24. Mi, Q., Xiao, Y., Cai, Z., Jia, X.: The effectiveness of data augmentation in code readability classification. Information and software technology. 129, 106378 (2021).
25. Bayer, M., Kaufhold, M.-A., Buchhold, B., Keller, M., Dallmeyer, J., Reuter, C.: Data augmentation in natural language processing: a novel text generation approach for long and short text classifiers. International journal of machine learning and cybernetics. 1–16 (2022).
26. Jin, W., Zhao, L., Zhang, S., Liu, Y., Tang, J., Shah, N.: Graph condensation for graph neural networks. Arxiv preprint arxiv:2110.07580. (2021).
27. Loukas, A.: Graph reduction with spectral and cut guarantees. Journal of machine learning research. 20, 1–42 (2019).
28. Hamilton, W., Ying, Z., Leskovec, J.: Inductive representation learning on large graphs. Advances in neural information processing systems. 30, (2017).

29. Perozzi, B., Al-Rfou, R., Skiena, S.: Deepwalk: Online learning of social representations. In: Proceedings of the 20th acm sigkdd international conference on knowledge discovery and data mining. pp. 701–710. ACM, New York, New York, USA (2014). `https://doi.org/10.1145/2623330.2623732`.
30. Chan, P.P., Collberg, C.: A method to evaluate cfg comparison algorithms. In: 2014 14th international conference on quality software. pp. 95–104. IEEE (2014).
31. Taylor, C., Colberg, C.: A tool for teaching reverse engineering. In: 2016 usenix workshop on advances in security education (ase 16) (2016).
32. Borello, J.-M., Mé, L.: Code obfuscation techniques for metamorphic viruses. Journal in computer virology. 4, 211–220 (2008).
33. Chao, W.-C.: Asm2vec-pytorch.
34. Tarjan, R.: Depth-first search and linear graph algorithms. Siam journal on computing. 1, 146–160 (1972).
35. Cohen, J.: A coefficient of agreement for nominal scales. Educational and psychological measurement. 20, 37–46 (1960).