LIBCAISE: Learning Interpretable Behavioral Components from Assembly Instructions of Software Executables

Jeremiah Greer, Rashmi Jha, Anca Ralescu, Temesguen Messay-Kebede, and David Kapp

Abstract—Traditional approaches to understanding program behavior involve either classifying programs with supervised machine learning algorithms or manually reverse engineering software. While many powerful classifiers exist, the features used by the classifiers often lack interpretability in the context of the software's behavior. As software and malware production increases, creating human-readable understanding of program behavior becomes more imperative. We propose LIBCAISE, a novel approach to understanding software behavior by applying clustering and topic modeling algorithms to assemblies of binary files to learn interpretable behavioral components of programs from assembly instructions of software executables. We apply this approach to statically derived assembly codes of multiple binary files and discuss the results of this as well as potential ways to expand upon the work.

Index Terms—Feature Extraction, Natural Language Processing, Pattern Analysis, Unsupervised Learning.

1 Introduction

There is a great need to understand how programs behave to ensure the security of software and to create the interpretable building blocks of program behavior by which to expand our understanding of software. This paper presents an approach to create a system that will enable us to gain a more readable understanding of the intrinsic differences between programs. The approach was studied on many commonly used sorting and searching programs as these programs form some of the basic components of many commercially available software and provide an excellent platform to develop and test the proposed approaches. Our observations indicate that our approach successfully isolates key behavioral components and presents them in a readable way, and maintains this at scale, though it loses some ability to derive fine-grained components it located in the small-scale experiment. LIBCAISE is scalable and extensible towards understanding any software with the possibility to extend the analysis towards malware behavior.

The paper is structured as follows: Section 2 details the background for the subject area, Section 3 discusses the extensive work done in the field to try to develop an understanding of program behavior (primarily malware, as most of the work is focused in this area), Section 4 details the specifics of our approach, as well as the assumptions made, Section 5 showcases the results of the work, Section 6 analyzes the effectiveness of the approach and discusses its limitations, Section 7 proposes areas in which the work can be extended, and finally Section 8 explains the key ideas and results of the work.

2 Background

The problem area of understanding binary executables is a more recent discussion that is compounded by the rise in third-party software and device platforms, including embedded and mobile devices. Verifying and understanding the behavior of programs is imperative to gaining insights into how programs relate to one another and can assist in identifying malfunctioning, incorrect, or malicious programs.

2.1 Third-Party Software

Software production has been steadily increasing over time, and as such, it is often more cost effective for companies to purchase Commercial Off-The-Shelf (COTS) software to fulfill their needs [1]. Not only this, but the rise of mobile devices and their corresponding app stores has drastically increased the availability of third-party software products/executables over time [2]. As such, it has become increasingly important to verify software behavior in the context of its expected purpose.

Normal program verification involves downloading from a trusted source or verifying the md5 hash of the download with the software distributor. This ultimately shifts the responsibility of distributing trustworthy software to the distributor, but there are multiple instances of programs such as malware making their way onto various distribution platforms, such as the Google Play Store [3].

Often times this is the result of hidden functionality being inserted into a normal program which goes against its intended operational paradigm, such as hidden functionality to mine user data in relatively benign programs [4] or having a program mine cryptocurrency while in use [5]. The user (and sometimes the distributor) has no way of knowing what these programs are doing or whether they are operating within their expected behavior profiles. Thus,

- *J. Greer is an MS student in the Department of Electrical Engineering and Computer Science, University of Cincinnati, Cincinnati, OH, 45220. E-mail: jeremiahgreer013@gmail.com*
- *R. Jha and A. Ralescu are with University of Cincinnati*
- *T. Messay-Kebede and D. Kapp are with Air Force Research Laboratory*

Manuscript received February 25, 2019; revised March 1, 2019.

understanding a program's behavior and how it relates to other programs of a similar class is imperative to ensuring programs are behaving as intended.

2.2 Malware

As access to software increases, so too does the risk of individuals downloading or installing malware on their systems. Whether it is stealing and selling identification and password data, holding computers or files for ransom from their owners, taking control of a remote system, or simply causing the host software or hardware platform to malfunction, malware comes in a wide variety of forms, and modern software production methodologies have made creating malware even easier, as often times it is simply an alteration of some main source malware [3].

There are many ways to verify the program that is being downloaded is what was expected and has not been modified, such as checking the md5 checksum against the distributor's checksum; however, many of these methods do not guarantee the safety of the program. Additionally, many times programs may not be doing anything that is particularly malicious, but is instead simply unwanted by the user, as previously mentioned with user data or cryptocurrency mining. Therefore, identifying, classifying, and understanding program behaviors, be they malicious, unwanted, or benign, becomes imperative to ensuring the safety of the user, platform, and program.

Usually programs are downloaded from a source or website and can generally be regarded as trustworthy, such as a secure website or mobile app store; however, there have been instances of trusted software containing malware and being distributed to users, unknown to both the distributor and the users. In 2017, 2.27 million users downloaded a version of CCleaner with malware embedded within, as the assailant was able to modify the program at the distribution site and thus modify the program without the knowledge of CCleaner or the users which downloaded this trusted software [6]. In addition, there are numerous instances of malware being downloaded through platforms such as the Google Play Store or Apple App Store [3]. As these platforms host millions of apps from developers, it can be difficult to identify potentially malicious software [2].

2.3 Importance of Interpretability

Classifying program behavior, and in particular in the cybersecurity domain as malicious or benign, is an unsolved problem due to the complexity of programs and the ever-evolving state of malware. Extensive work has been done in this area with a multitude of datasets, including Microsoft's 2015 Kaggle dataset [7], which contains numerous examples of malware with the end task of classifying each program as belonging to one of nine malware families; however, many of these works do not improve our intrinsic understanding of these programs and what makes them different.

Identifying software characteristics is a well-studied problem, but as software and malware change over time, a complete solution does not and most likely cannot exist without also being able to adapt. This adaptation necessitates developing a better understanding of how programs behave and how they might relate to one anther in a way that human experts can understand. Even though there are many instances of accurate classifiers, much of the work done is primarily focused on creating new classifiers or comparing them against other models to gauge effectiveness, while very few if any of the works go into understanding the differences between software or the features which are derived by their models. While this paper does not utilize malware classification and is instead left as future work, it is beneficial to maintain this mindset for the purposes of understanding some of the broader implications of this work and what it brings to both the machine learning and software analysis domains.

3 Previous Work

Much work has been done on understanding and analyzing general program behavior and how malware's behavior is differentiated from it, though as will be shown, few works have tried to maintain interpretability throughout the entirety of the model pipeline, thus outlining one of the main benefits to this project's approach, particularly for human experts. There are other approaches to interpretable machine learning discussed below as well.

3.1 Understanding Program Behavior

There are numerous perspectives with which to understand program behavior, though many of them either lack interpretability or are unable to be generalized to many types of software. Ghosh *et al.* [8] utilized neural networks and Elman networks to recognize recurrent features in program execution traces, and while successful and generalizable, the features derived by these networks are often unknowable by humans, leading to their general ”black-box” sentiment. Sherwood *et al.* [9] utilized Basic Block Vectors to identify behaviors in large-scale software execution (billions of commands), and clustered their behavior based on this model, but again, interpreting what these behaviors or vectors mean in relation to the specific goals of the program is difficult.

Bowring *et al.* [10] applied Markov models and an active learning approach to multiple executions of programs to identify behaviors among them, utilizing extracted execution statistics to create behavioral profiles for programs. While powerful, the execution statistics lose some amount of information compared to execution traces or the program's static assembly code. Also, depending on the classifier(s) used, any extrapolated features may be difficult to interpret. Not only this, but dynamic analysis is often slow and potentially dangerous to collect the necessary data. Mohaisen *et al.* [11] developed a patent to describe a generalized malware analysis system which reflects many of the studies done on analyzing programs and malware. Most tend to take on a similar architecture and are primarily focused on classifier performance rather than gaining insights into understanding program behavior.

There exist many behavioral systems specifically tailored to malware as well. Jacob *et al.* [12] introduces the general framework by which one would create an interpretable malware detection system and establishes a taxonomy to discuss malware, something which would prove very useful to follow. Andromaly [13], Droidmat [14], and Crowdroid [15] all introduce behavior-based malware detection systems for the Android operating system, though their primary concerns are all specifically related to the detection of malware as opposed to understanding its behavior or purpose, or how it might be different from non-malicious programs.

3.2 Malware Analysis and Classification

While this paper's work is focused on interpreting general software and not specifically malware, the application to the malware and cybersecurity domain is important enough to warrant mentioning the other approaches towards malware analysis and classification, particularly as many focus their attention in this area on malware vs non-malware rather than general program analysis.

Identifying key behavioral characteristics of software and malware is one of the main goals of this work, but specifically to do so in such a way that the approach is automatically applicable to new programs while still maintaining human understanding. There are many methods of representing the structure or behavior of programs in a formal way, such as Abstract Syntax Trees, Control Flow Graphs, Call Graphs, and Dependency Graphs. Many of these methods have been used to great effect in understanding information flow through a program; however, many of these methods are either intensive, do not work on binaries/assemblies, or are difficult to use in creating direct comparisons between two programs.

One major work which seeks to categorize specifically malware behavior is Malware Attribute Enumeration and Characterization (MAEC) [16]. Essentially an encyclopedia of higher-level malware attributes, this work tries to characterize major malware families based on a set of common attributes held by malware; however, this work is generally human curated, which can potentially lead to error, but more importantly will not work with large amounts of new programs, as new malware can be created and distributed more quickly than a human expert can sufficiently study it. If a system were able to automatically extract these high-level features, then this structure would be very useful in discussing and classifying malware.

There are many works which seek to identify and classify malware based on their key features. Rieck *et al.* [17] extracted generalized behavior features (such as opening an IRC connection) of malware as it runs in a sandboxed environment and clustered them based on these components using an SVM with a bag-of-words model. Their approach allows for an interpretable understanding of their features and enables a feature ranking on the extracted dynamic components, something not often seen in the other approaches. SVMs could reasonably be extended to multi-class classification, though the features and their rankings would become more unwieldy as the number of classes grows. Since we hope to approach a general software domain where there could be any number of classes or behaviors, this seemed infeasible for our purposes.

Lindorfer *et al.* [18] developed a system to detect environment-sensitive malware, as some malware has developed the ability to recognize when it is being run in a sandboxed environment and thus behave differently, avoiding detection or study. They do this by analyzing programs run in multiple sandboxes multiple times and detecting differences between them using an Information Theory-based approach with Jaccard distances. While this gives an idea as to a program's environmental behavior, it fails to indicate anything intrinsic about its behavior or purpose.

Santos *et al.* [19] utilized a combined static-dynamic approach to detecting malware, combining opcode frequency and an execution trace as data. They apply a variety of models to this data and comparing their performances. As the main purpose of the work was to show that a combined static-dynamic approach performs better than either approach alone, it provides no greater understanding into differentiating malware behavior. Sun *et al.* [20] introduced a patent for a system which automatically generates a malware signature based on what malware it is classified as, but detail is not given in terms of what differentiates the malware or malware classes. Bailey *et al.* [21] sought to identify malware by using a file compression of dynamically generated system state logs and cluster the files based on this metric; however, again, analyzing the entire log would prove infeasible for a human expert, and the compression metric leaves limited interpretability in terms of the similarity between two files beyond the score itself.

Most major works seem primarily concerned with model performance, with relatively little work done in trying to create or assist in creating new insights into understanding program behavior. Enabling these insights requires maintaining interpretability throughout the process, thereby limiting model selection to those which act with interpretable features.

3.3 Interpretable Machine Learning

As this paper's goal is to develop a system which is able to extract interpretable behavior components from software, much focus was placed in identifying potential interpretable machine learning models. Doshi-Velez *et al.* [22] discussed the specifics of what it means for a machine learning model to be interpretable and establishes a taxonomy and set of human metrics by which one would be able to better assess the model and results. Vellido *et al.* [23] discuss the multitude of ways in which human interpretability of machine learning models can be achieved. Dimensionality reduction techniques, such as Principal Component Analysis, offer the simplest way to achieve interpretability of a machine learning model. Many problems exist in high-dimensional spaces, and as such, being able to reduce the dimensionality or extract a small subset of features ensures that whatever insights can be gained from the model are interpretable for human understanding. We employ a unique dimensionality reduction technique in the form of embedding clusters to reduce the total vocabulary of assembly instructions, but there are other parts of the process which cannot rely on dimensionality reduction alone, as doing so reduces the amount of data available to the model and may thus reduce the accuracy of the model or significance of any insights.

Hainmueller *et al.* [24] utilizes a different approach using Kernel Regularized Least Squares, which while it allows for interpretable models which do not depend on linearity or additivity, its interpretability may start to break down in higher-dimensional spaces, and given that our data is software, reducing its dimensionality will distort the data and our insights.

To ensure interpretability, we wish to operate on the textual data of assembly commands, as a human expert can analyze these commands and have a reasonable understanding of the underlying behavior of the program. Chen *et al.* [25] creates a system called Infogan which applies an Information Theoretic approach to Generative Adversarial Networks to create a system with interpretable learned features and applies this approach to image data. Infogan's random variables can be varied along their bounds to discern the effects they have on the dataset, and were found to identify and control things such as digit type, rotation, and width in an unsupervised manner. It would be interesting to see this approach adapted to text data and see what features can be learned; however, this approach limits the number of features one could learn and interpret from the model to the number of random variables, and it remains to be seen if adding more variables may damage the interpretability of each of the variables.

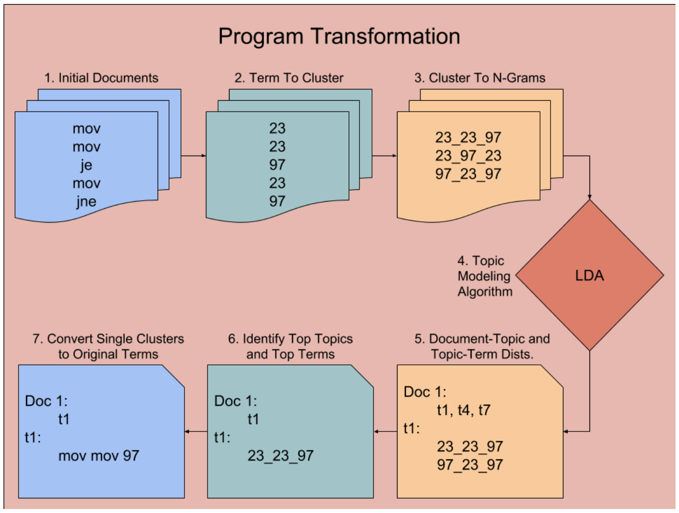


Fig. 1. Flowchart detailing the flow and transformation of information across the entire process. Documents are transformed and analyzed using LDA, and then the transformations are undone to allow for fine-grain interpretability.

4 Procedure

Our goal was to develop a system which would assist in generating new knowledge and insights about software and malware behavior. To that end, we needed a system which would be interpretable to human experts while still offering utility for comparison. With these criteria in mind, we developed LIBCAISE, a system for learning interpretable behavioral components through a novel approach to software analysis by utilizing a variety of Natural Language Processing (NLP) techniques together to create a system which satisfies all of these requirements.

Little work has been done on analyzing software from a natural language perspective, but the methods used in these approaches often lend themselves to being more interpretable than other machine learning algorithms, as NLP's goal is often to gain insights on human language and its use and meaning as opposed to model performance. NLP approaches also have useful measures for taking the order of terms into account, as human sentences rely on order to derive meaning, with changes in order causing changes in meaning, something which programs mimic as well, though differently with the use of jump statements and function boundaries. Therefore, this paper extends some of these techniques toward program analysis to better derive meaning from the data and model.

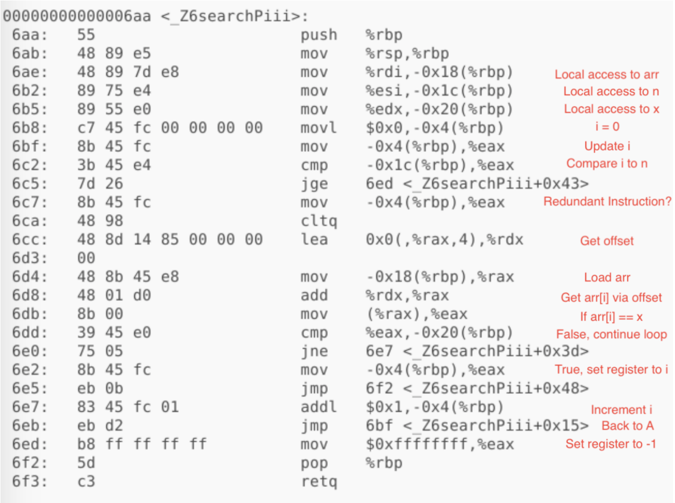
The overview of this approach is shown in Figure 1. The process is outlined in detail as follows:

1) The process takes in a binary executable and utilizes objdump [26] to extract the assembly instructions. Each document is a series of assembly commands with their arguments stripped from them.
2) Using Word2Vec, embeddings are learned for each of the commands. Using hierarchical clustering, these embeddings are then clustered together, with the threshold pre-determined by a human expert (though a more extensive study of assembly command usage would yield better clusters). Once found, these are saved for later uses.
3) Once clustered, the documents are transformed such that all assembly commands are replaced by their respective cluster ID's. This means that each document is now a sequence of cluster ID's rather than assembly commands.
4) After the commands are converted, the documents are transformed again by taking N-grams of these cluster ID's. A larger N corresponds to a more interpretable component, with the downside that the vocabulary is drastically increased.
5) Now that the documents have been transformed to a sequence of N-grams, Latent Dirichlet Allocation (LDA) is then applied to this transformed corpus, and the appropriate Document-Topic and Topic-Term distributions are learned.
6) Utilizing these distributions, the top T terms for each topic can be extracted, with these terms corresponding to the behavior that the topic captures. Similarly, the top D topics for each document can be extracted, resulting in a higher-level comparison between documents in terms of what topics they share.
7) Using these results, one can compare the behavior between two programs and determine to what degree they share certain behaviors. In addition, one can identify what those shared behaviors actually are and what sort of result or purpose they may have.

The system described fulfills the requirements of extracting interpretable behavioral components of software in an unsupervised manner. The details of each of the system's components is discussed below.

4.1 Data and Assembly Instructions

Before moving towards a specific model, we first wished to see what differences occur between programs with different purposes, such as sorting programs versus searching programs. These two types of programs are two fundamental



```
00000000000006aa <_Z6searchPiii>:
 6aa:   55                      push   %rbp
 6ab:   48 89 e5                mov    %rsp,%rbp
 6ae:   48 89 7d e8             mov    %rdi,-0x18(%rbp)     Local access to arr
 6b2:   89 75 e4                mov    %esi,-0x1c(%rbp)     Local access to n
 6b5:   89 55 e0                mov    %edx,-0x20(%rbp)     Local access to x
 6b8:   c7 45 fc 00 00 00 00    movl   $0x0,-0x4(%rbp)      i = 0
 6bf:   8b 45 fc                mov    -0x4(%rbp),%eax      Update i
 6c2:   3b 45 e4                cmp    -0x1c(%rbp),%eax     Compare i to n
 6c5:   7d 26                   jge    6ed <_Z6searchPiii+0x43>
 6c7:   8b 45 fc                mov    -0x4(%rbp),%eax      Redundant Instruction?
 6ca:   48 98                   cltq
 6cc:   48 8d 14 85 00 00 00    lea    0x0(,%rax,4),%rdx    Get offset
 6d3:   00
 6d4:   48 8b 45 e8             mov    -0x18(%rbp),%rax     Load arr
 6d8:   48 01 d0                add    %rdx,%rax            Get arr[i] via offset
 6db:   8b 00                   mov    (%rax),%eax          If arr[i] == x
 6dd:   39 45 e0                cmp    %eax,-0x20(%rbp)     False, continue loop
 6e0:   75 05                   jne    6e7 <_Z6searchPiii+0x3d>
 6e2:   8b 45 fc                mov    -0x4(%rbp),%eax      True, set register to i
 6e5:   eb 0b                   jmp    6f2 <_Z6searchPiii+0x48>
 6e7:   83 45 fc 01             addl   $0x1,-0x4(%rbp)      Increment i
 6eb:   eb d2                   jmp    6bf <_Z6searchPiii+0x15>  Back to A
 6ed:   b8 ff ff ff ff          mov    $0xffffffff,%eax     Set register to -1
 6f2:   5d                      pop    %rbp
 6f3:   c3                      retq
```

Fig. 2. Annotated version of Linear Search's search function in assembly. Similarities were found in sorting functions with the primary difference being sections of data modification and multiple for-loops.

concepts of many programs and were identified as appropriate vehicles to develop and understand this approach. Initial analyses were performed on a small dataset collected from the Geeks4Geeks Website [27], where C++ samples of major sorting and searching algorithms were downloaded and compiled onto a Ubuntu 64-bit system. A larger dataset was later utilized using ByteWeight's dataset [28].

To understand the differences between programs, we first closely examined one sorting and one searching program to see what differences might exist between their assemblies. Figure 2 shows an annotated version of Linear Search's search function, and when compared with a sorting program such as Selection Sort (which effectively contains a modified version of a linear search as it sorts the program), the primary differences found were multiple for-loops and sections for data modification, something which most, if not all, sorting programs should have, but no realistic searching programs should have, as efficient searching programs should only have one loop, and should not be modifying the data. This formed the basis of our argument: Programs can be thought of as being made up of a set of underlying components, and these components are in some way shared across all programs. The order and frequency in which these components occur can enable behavioral comparisons between programs, and understanding the components themselves allows for fine-grain understanding of behavior. The system presented in this work only incorporates the commands themselves so that basic functionality of the software could be understood. Incorporating arguments would allow for a better understanding of command targets (such as specific registers or functions), but would first require abstracting these in a way similar to Symbolic Execution so that program-to-program comparisons can be maintained.

4.2 Word2Vec Embeddings and Clustering

Assembly commands also deserve their own form of abstraction, primarily in the context in which they are used. Using sorting as an example, changing a single greater-than comparator to a less-than comparator does not change the fact that the program is sorting, all it changes is the sorting order from increasing to decreasing. In order to allow for greater robustness in program types, our system must be able to take the context in which a specific command is used into account. One of the most effective methods of this is Word2Vec [29], which is able to identify terms used in similar contexts and can even be extrapolated to higher-level relationships (such as man-¿woman being similar to king-¿queen). Using the embeddings learned from Word2Vec, we gain a numerical representation of each command's context, with more similar commands sharing similar contexts.

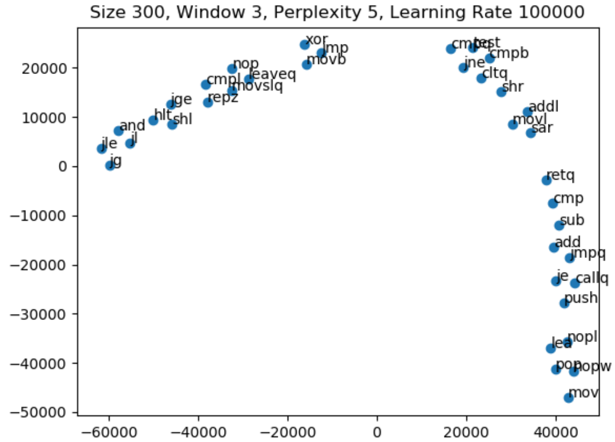


Fig. 3. T-SNE plot of small-scale data Word2Vec results.

The small-scale results of this are shown in the T-SNE plot in Figure 3. The findings shown in the plot correspond relatively well to human understanding, as most of the jump comparators are close to each other (jle, jg, jge); however, je is actually quite far from the rest of the comparators. We believe this is the result of the small data sample (16 documents) skewing the results. Once these representations were found, we applied hierarchical clustering to them and used our best understanding of command similarity to define the cutting point. Though this approach was adequate to confirm the validity of our system, a more formal study of assembly language usage would be preferred to validate the contexts in which commands are used and how they relate to one another. The results of this are shown in Figure 4. Both of these processes are repeated for the larger (about 850 documents) dataset, though the results of those embeddings and clusters are not shown in this paper as the images are too cluttered to be meaningfully presented.

4.3 N-Grams

After converting the assembly commands to their corresponding cluster ID's, a way to have components with interpretable behavior was desired. To that end, individual commands carry little meaning beyond their intended purpose, so we needed some alteration of them. This brings us to N-Grams, which are sequences of terms (in this case cluster IDs) adjacent to each other (e.g. the 2-grams of "mov addl mov pop" are "mov addl", "addl mov", and "mov pop"). Smaller N-grams are less interpretable than

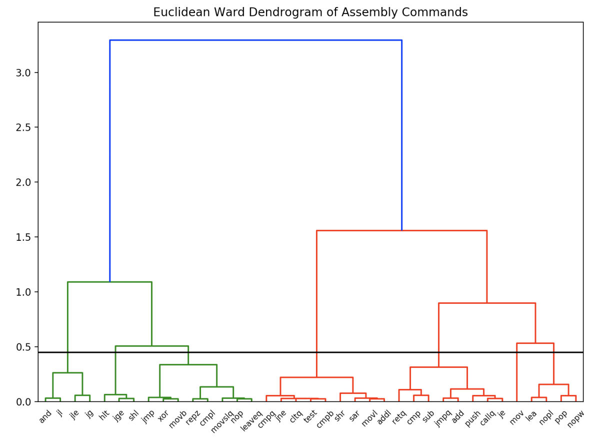


Fig. 4. Hierarchical Clustering results of the small dataset.

large N-grams, and so a sizeable N must be chosen to maintain interpretability, which for our purposes means N = 8. Unfortunately, this has the side effect of drastically increasing vocabulary, so some caution must be taken.

4.4 LDA and hLDA

As we defined programs as being made up sets of behavioral components, and those behaviors are largely unknown to us, we needed a system which would allow us to extract those behaviors in an unsupervised manner. This is actually a well-studied area in languages with NLP approaches in the area called Topic Modeling. Blei *et al.* [30] developed a method for unsupervised topic modeling called Latent Dirichlet Allocation, and would later expand upon it by developing Hierarchical Latent Dirichlet Allocation (hLDA) [31]. LDA essentially considers each document to be made up of some weighting of latent topics, and each topic is made up of some weighting of terms, and finds the distribution on these weightings which can be used for inference. hLDA is a hierarchical variant which adds the idea that topics are actually structured in a hierarchy. This approach is more akin to software than flat topics; however, there are not many performant hLDA libraries available for larger datasets. This lead to a focus on LDA for the time being. LDA has more performant versions available, such as Microsoft's LightLDA [32], which we utilize in our analyses on the larger dataset.

5 Results

The following are the results of applying our approach to both small-scale and large-scale data, with small-scale referring to the 16 sorting and searching program samples from GeeksForGeeks (with an end-goal to differentiate sorting and searching programs), while the large-scale also includes the data files from ByteWeight, which consist of a variety of binutils programs compiled using different compilers and different compiler arguments.

5.1 Small-Scale Results

Below are the results of small-scale experiments with LDA and hLDA, both of which perform quite well.

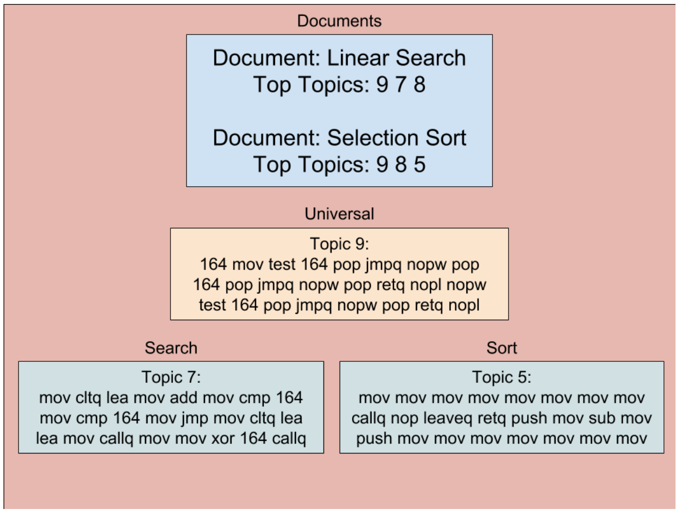

Documents
Document: Linear Search
Top Topics: 9 7 8
Document: Selection Sort
Top Topics: 9 8 5
Universal
Topic 9:
164 mov test 164 pop jmpq nopw pop
164 pop jmpq nopw pop retq nopl nopw
test 164 pop jmpq nopw pop retq nopl
Search
Topic 7:
mov cltq lea mov add mov cmp 164
mov cmp 164 mov jmp mov cltq lea
lea mov callq mov mov xor 164 callq
Sort
Topic 5:
mov mov mov mov mov mov mov mov
callq nop leaveq retq push mov sub mov
push mov mov mov mov mov mov mov

Fig. 5. Side-by-side comparison of Linear Search and Selection Sort. Selection sort can be thought to contain the entirety of linear search (as a slightly modified linear search is used in the selection sort algorithm), so these two programs will be the most similar while still having different purposes.

TABLE 1
Final results of small-scale LDA model. The top topics are determined based on the learned document-topic distribution. Focus on specific topics is contained in Figure 5. Hyperparameters for these results were the following: Number of topics = 13, $\alpha = 1e - 10, \beta = 0.1$

LDA Table Results

Program	Top Topics		
$BinarySearch$	9	7	12
$InterpolationSearch$	9	2	7
$LinearSearch$	9	7	8
$Rec.LinearSearch$	9	11	12
$JumpSearch$	9	11	19
$ExponentialSearch$	9	7	12
$BubbleSort$	9	8	5
$Rec.BubbleSort$	9	8	5
$MergeSort$	6	9	0
$Rec.MergeSort$	6	9	0
$QuickSort$	5	9	11
$Rec.QuickSort$	9	5	8
$InsertionSort$	9	4	8
$Rec.InsertionSort$	9	4	11
$SelectionSort$	9	8	5
$Rec.HeapSort$	9	1	11

5.1.1 LDA Results

Figure 5 shows a comparison between the Linear Search and Selection Sort programs with n-grams of size 8. The three topics displayed below were the most significant topics. Topic 9 was held as the most heavily weighted topic for most of the programs, and was the second most significant for three of the sixteen. Topics 7 and 5 were found to be the key differentiators between sorting and searching programs, with topic 7 being found in the majority of searching programs and no sorting programs, and topic 5 being found in many sorting programs and no searching programs. The full small-scale results are shown in Table 1. Note the fact that topic 5 contains indications of heavy data modification (through the large number of mov statements), while topic 7 makes note of array comparisons in a loop.

TABLE 2
Final results of small-scale hLDA model. Each document is a leaf node in a hierarchy of topics, so those at the same leaf node are most similar. Hyperparameters for these results were the following: Number of levels = 3, number of samples = 500, $\alpha = 10, \gamma = 1, \beta = 0.1$

hLDA Table Results

Program	Deepest Topic in Hierarchy
$BinarySearch$	8
$InterpolationSearch$	8
$LinearSearch$	8
$Rec.LinearSearch$	8
$JumpSearch$	8
$ExponentialSearch$	8
$MergeSort$	4
$Rec.MergeSort$	4
$BubbleSort$	10
$Rec.BubbleSort$	10
$QuickSort$	10
$Rec.QuickSort$	10
$SelectionSort$	10
$InsertionSort$	6
$Rec.InsertionSort$	6
$Rec.HeapSort$	6

TABLE 3
Final results of large-scale LDA model. The top topics are determined based on the learned document-topic distribution. Hyperparameters for these results were the following: Number of topics = 100, $\alpha = 0.001, \beta = 0.001$. The - symbol means that no additional topics were formed as part of its weighting (due to their weights being too small).

LDA Table Results

Program	Top Topics				
$BinarySearch$	91	86	62	78	42
$InterpolationSearch$	91	88	86	61	90
$LinearSearch$	91	86	33	78	−
$Rec.LinearSearch$	91	61	86	89	20
$JumpSearch$	91	78	86	89	27
$ExponentialSearch$	91	61	86	78	3
$BubbleSort$	91	86	51	3	−
$Rec.BubbleSort$	91	86	51	90	80
$MergeSort$	91	86	61	74	20
$Rec.MergeSort$	91	86	61	74	20
$QuickSort$	91	61	86	62	−
$Rec.QuickSort$	91	61	86	62	24
$InsertionSort$	91	86	−	−	−
$Rec.InsertionSort$	91	86	61	−	−
$SelectionSort$	91	86	61	80	90
$Rec.HeapSort$	91	86	61	20	56

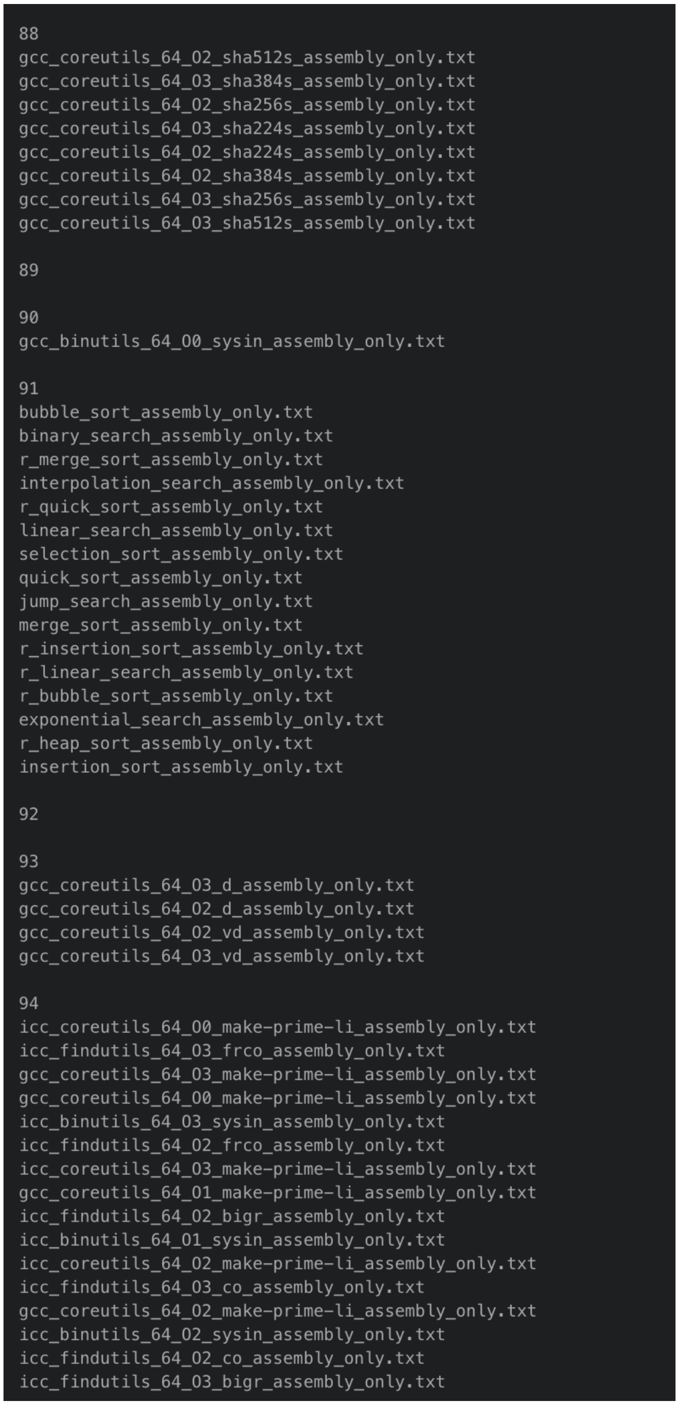
5.1.2 hLDA Results

Table 2 shows the final topics corresponding to each of the programs from hLDA. One can observe that all search programs were successfully isolated from all sort programs. In addition, various recursive versions of programs were matched with their non-recursive versions. These results are very promising and clearly indicate tremendous potential for this approach.

5.2 Large-Scale Results

This section discusses the results of the large-scale LDA experiments.

Figure 6 shows a small subset of the large-scale results, namely the documents being sorted into bins based on what each document's highest-weighted topic was. As can be seen, all sorting and searching programs are placed in the



```
88
gcc_coreutils_64_02_sha512s_assembly_only.txt
gcc_coreutils_64_03_sha384s_assembly_only.txt
gcc_coreutils_64_02_sha256s_assembly_only.txt
gcc_coreutils_64_03_sha224s_assembly_only.txt
gcc_coreutils_64_02_sha224s_assembly_only.txt
gcc_coreutils_64_02_sha384s_assembly_only.txt
gcc_coreutils_64_03_sha256s_assembly_only.txt
gcc_coreutils_64_03_sha512s_assembly_only.txt

89

90
gcc_binutils_64_00_sysin_assembly_only.txt

91
bubble_sort_assembly_only.txt
binary_search_assembly_only.txt
r_merge_sort_assembly_only.txt
interpolation_search_assembly_only.txt
r_quick_sort_assembly_only.txt
linear_search_assembly_only.txt
selection_sort_assembly_only.txt
quick_sort_assembly_only.txt
jump_search_assembly_only.txt
merge_sort_assembly_only.txt
r_insertion_sort_assembly_only.txt
r_linear_search_assembly_only.txt
r_bubble_sort_assembly_only.txt
exponential_search_assembly_only.txt
r_heap_sort_assembly_only.txt
insertion_sort_assembly_only.txt

92

93
gcc_coreutils_64_03_d_assembly_only.txt
gcc_coreutils_64_02_d_assembly_only.txt
gcc_coreutils_64_02_vd_assembly_only.txt
gcc_coreutils_64_03_vd_assembly_only.txt

94
icc_coreutils_64_00_make-prime-li_assembly_only.txt
icc_findutils_64_03_frco_assembly_only.txt
gcc_coreutils_64_03_make-prime-li_assembly_only.txt
gcc_coreutils_64_00_make-prime-li_assembly_only.txt
icc_binutils_64_03_sysin_assembly_only.txt
icc_findutils_64_02_frco_assembly_only.txt
icc_coreutils_64_03_make-prime-li_assembly_only.txt
gcc_coreutils_64_01_make-prime-li_assembly_only.txt
icc_findutils_64_02_bigr_assembly_only.txt
icc_binutils_64_01_sysin_assembly_only.txt
icc_coreutils_64_02_make-prime-li_assembly_only.txt
icc_findutils_64_03_co_assembly_only.txt
gcc_coreutils_64_02_make-prime-li_assembly_only.txt
icc_binutils_64_02_sysin_assembly_only.txt
icc_findutils_64_02_co_assembly_only.txt
icc_findutils_64_03_bigr_assembly_only.txt
```

Fig. 6. Documents placed into bins corresponding to their highest-weighted topic. All sorting and searching programs are placed into the same bin, with no other programs being placed within it.

same bin, with no other programs being placed with them. Additionally, examination of nearby bins reveals other successes, such as isolation of multiple types of hash functions, or different compiled versions of the same program (make-prime-li).

Table 3 details the results of the large-scale test for the original sorting and searching programs. The main difference of this test is to see what topics would be found when there are a wider variety of programs, and how the programs would relate to one another when scaling up. Results show programs pairing with their recursive versions somewhat less frequently, and there do not exist any major differentiators between the sorting and searching programs. That being said, all of the small-scale programs (sorting and searching programs of similar style and the same origin, compiler, and compiler arguments) were found to have the same most significant topic (topic 91). While not as strong as the small-scale results, they are still quite promising.

6 Discussion

The details of our experimental findings are discussed in this section, including the trade-offs between LDA and hLDA, and the differences between the small-scale and large-scale results.

6.1 LDA and hLDA

Results from the small-scale LDA and hLDA experiments were both incredibly promising, as both were able to identify topics which separated programs of different behaviors, fulfilling the goal of being able to learn interpretable behavior components from the assemblies of various programs. These components represent the key differences in behavior that would identify programs as being a sorting or searching program (or neither). Not only that, but this system enables one to understand to what degree a program is of a certain type or what subtype it may belong to based on other programs of the same type, such as identifying a program as closer to binary search or linear search; however, it should be noted that these approaches do not come without their issues. Identifying the number of topics in LDA requires either hand-tuning or the use of a hyperparameter searching algorithm such as grid search to optimize perplexity. Alpha and beta also require tuning, but the number of topics has the strongest effect on model results. Finding the hyperparameters for hLDA would require something similar except more parameters need to be found. These problems are not unique to LDA and hLDA however, as almost all machine learning algorithms require some form of hyperparameter tuning [33]. Most systems utilizing machine learning to identify these components would require some form of hyperparameter search.

Determining the number of topics is a difficult problem for LDA as it may change depending on the number of documents and their contents. One could perform a hyperparameter grid search, using perplexity as the basis for determining the optimal number of topics; however, perplexity may not be an adequate measure. Change *et al.* [34] found that perplexity did not correlate with human judgment in the topics found very well. While there may be other methods for determining the adequacy of a number of topics, it may be more beneficial to use the Hierarchical Dirichlet Process [35], as it generates similar results compared to LDA, but the number of topics is no longer a hyperparameter and is instead determined by the model. Applying HDP instead of LDA may be more beneficial for determining the number of topics; however, there are still other hyperparameters which need to be tuned, and HDP may not be as performant as LDA.

It is clear that hLDA (Table 2) performs better as a classifier for sorting and searching programs compared to LDA (Table 1), as it is able to create a complete and clear separation between sorting and searching programs with additional insights for recursive non-recursive pairs. In addition, hLDA's innate hierarchical structuring of components is more appropriate to our idea of behavioral components having a hierarchical relationship; that being said, LDA's result structure is more directly interpretable in terms of the components that make up a given program, and gives a clearer picture in terms of understanding which components make up a given document and to what weighting they have. In addition, hLDA is much less scalable than LDA, as it requires iterating through the length of every document multiple times. We found that it runs approximately 10-15x longer than LDA with the trade-off of achieving better results, though the results for a given hyperparameter set were more inconsistent with hLDA than LDA, which is most likely the result of hLDA making use of additional random variables in its model, decreasing its consistency in results. With these ideas in mind, we decided to maintain LDA for the large-scale experiment until such a time as a more performant hLDA library is made available or better hardware is found, and leave this as an area to study in the future.

6.2 Large-Scale LDA

As performance was poor using Python for LDA from GuidedLDA [36], we searched for more efficient libraries for LDA and found Microsoft's LightLDA program, written in C++ to be multi-threaded and able to be distributed to multiple machines. We thought this to be the most appropriate package to use moving forward into any larger data sets.

At a high-level, the large-scale results were quite good, as all of the sorting and searching programs were found to have the same highest-weighted topic. As indicated in Figure 6, no other programs were binned with them. In addition, other similar programs were found to be mapped together, such as different versions of hashing functions (sha512s, sha256s, etc.) or the same program with different arguments or compilers (make-prime-li). This can give potentially immediate insights into a program's point of origin in relation to other programs and its behavior. This also confirms the system is stable under different compilers and compiler options, which would prove very beneficial for maintaining robustness in applications related to malware analysis.

At a finer granularity, the system does not perform as well on the small-scale experiment. Most of the searching and sorting programs have a high level of interconnectivity in terms of shared top topics, and some of the recursive versions are matched with their non-recursive forms (such as quick sort and merge sort), but there does not exist any single identifiable topic for sorting or searching which might be a key differentiator between them. This is most likely due to the heavy weighting of program types, as the majority of programs are from the ByteWeight data set, and many of the programs are duplicates with different compilers or compiler arguments, thereby shifting many of the topics found to be more closely related to them than a small subset of sorting and searching programs. Increasing the number of topics could potentially improve this, but there is a bound to the effectiveness of increasing the number of flat topics, and as we discuss later, using a hyperparameter search to optimize perplexity may not lead to the correct topics.

Despite there being no key differentiators at the large-scale, this system still provides the basis by which one could, in an unsupervised manner, find interpretable behavioral components of programs and compare programs based on these shared or unshared components. Being able to extract these components in an unsupervised way is the key to creating new insights, as much of the underlying behavioral differences between programs remain unknown to us in an understandable way, despite much previous work being done in studying their behavioral differences in attempts to classify them. This system also provides a strong foundation on which many subsystems can be added to improve performance or gain greater insight into program behavior.

7 Future Work

There are a multitude of directions towards which this work could be extended, with several of them being discussed below, along with any potential roadblocks they may have.

In its current state, our system largely focuses on behavior as separate from context, but combining this with a contextual analysis system would greatly enhance understanding of behavior. For one, incorporating the arguments of the instructions would give greater clarity in terms of the actual operations and machine state being altered by the program itself. Beyond the instructions themselves, incorporating the environment and machine state into the model (such as the physical environment around the machine, the type of machine, or the machine's operating system) would assist in generating a system-wide view of behavior and how its state is modified by the underlying behavioral components. This would improve understanding of software and its behavior and would be particularly beneficial for isolating malware and determining its severity in a given context.

Our current approach focuses on statically derived assembly commands taken from the objdumps of binary executables; however, this does not give a complete picture of the software and has multiple shortcomings, such as ignoring function boundaries and ignoring jump locations. If a dynamic trace were collected through execution of the programs such that each program were represented by the assembly commands used in the order of their execution throughout the entire runtime, one would thereby gain insights into the function boundaries and jump statements. Unfortunately, collecting these dynamic traces takes time, as each program must be run multiple times with different arguments, as not all commands are run in every execution. As concluded in [19], a combined static-dynamic approach would most likely be best.

This paper details an initial proof of concept on a small-scale sort versus search data set, with additional testing on a large-scale data set. While this study did not focus on malware, its applicability to the malware analysis domain is clear. The difficulty with studying malware stems from safely acquiring copies of malware and using them for study. The most widely available data set is Microsoft's Kaggle data set [7], which can be used to try to classify malware into one of nine families. This would be the most immediately useful data set to study, though due to its size, performance may become an issue if one does not use an efficient library. The issue with this data set is its lack of benign program samples, which would be necessary if trying to find differentiators between malware and non-malware programs and their corresponding behaviors. One could add the ByteWeight data set to it, but more varied programs and more examples of commercial software would assist in making the approach generalizable.

8 Conclusion

Understanding a software's behavior, both intrinsically and in relation to other software, is an important stepping stone towards effective program behavioral analysis and can give greater insights into understanding how programs relate to one another. We detail the importance of this understanding and of having interpretable behavioral components and presented our system LIBCAISE, a novel approach to identifying behavioral components in software by applying NLP techniques to the domain of software analysis. While there are various tradeoffs to our approach, it succinctly captures multi-level interpretable software behavior and lays important groundwork on which a variety of directions may extend from.

Acknowledgments

The authors would like to thank the University of Cincinnati (UC), Air Force Research Laboratory (AFRL), and Defense Associated Graduate Student Innovators (DAGSI) for providing the resources necessary to complete this work and for the collaboration with members of AFRL. This work was funded in part by DAGSI award number RY12-UC-18-4. Code and data available on GitHub [1]

References

[1] V. K. Agrawal, V. K. Agrawal, and A. R. Taylor, "Trends in commercial-off-the-shelf vs. proprietary applications," *Journal of International Technology and Information Management*, vol. 25, no. 4, p. 2, 2016.

[2] "App stores: number of apps in leading app stores 2018." [Online]. Available: https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/

[3] D. Maier, T. Müller, and M. Protsenko, "Divide-and-conquer: Why android malware cannot be stopped," in *2014 Ninth International Conference on Availability, Reliability and Security*. IEEE, 2014, pp. 30–39.

[1] https://github.com/jgreer013/libcaise

[4] A. Bosu, F. Liu, D. D. Yao, and G. Wang, "Collusive data leak and more: Large-scale threat analysis of inter-app communications," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. ACM, 2017, pp. 71–85.

[5] D. Palmer, "8 illicit crypto-mining windows apps removed from microsoft store," Feb 2019. [Online]. Available: https://www.coindesk.com/8-illicit-crypto-mining-windows-apps-removed-from-microsoft-store

[6] P. Arntz, "Infected ccleaner downloads from official servers," Sep. 2017. [Online]. Available: https://blog.malwarebytes.com/security-world/2017/09/infected-ccleaner-downloads-from-official-servers/

[7] R. Ronen, M. Radu, C. Feuerstein, E. Yom-Tov, and M. Ahmadi, "Microsoft malware classification challenge," *CoRR*, vol. abs/1802.10135, 2018. [Online]. Available: http://arxiv.org/abs/1802.10135

[8] A. K. Ghosh, A. Schwartzbard, and M. Schatz, "Learning program behavior profiles for intrusion detection," in *Proceedings of the Workshop on Intrusion Detection and Network Monitoring*. Berkeley, CA, USA: USENIX Association, 1999, pp. 51–62. [Online]. Available: http://dl.acm.org/citation.cfm?id=647593.728880

[9] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. 5, pp. 45–57, Oct. 2002. [Online]. Available: http://doi.acm.org/10.1145/635508.605403

[10] J. F. Bowring, J. M. Rehg, and M. J. Harrold, "Active learning for automatic classification of software behavior," in *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '04. New York, NY, USA: ACM, 2004, pp. 195–205. [Online]. Available: http://doi.acm.org/10.1145/1007512.1007539

[11] A. Mohaisen, O. Alrawi, and M. Larson, "Systems and methods for behavior-based automated malware analysis and classification," U.S. Patent US20 140 244 733, 2014. [Online]. Available: https://patents.google.com/patent/US20150244733

[12] G. Jacob, H. Debar, and E. Filiol, "Behavioral detection of malware: from a survey towards an established taxonomy," *Journal in computer Virology*, vol. 4, no. 3, pp. 251–266, 2008.

[13] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, ""andromaly": a behavioral malware detection framework for android devices," *Journal of Intelligent Information Systems*, vol. 38, no. 1, pp. 161–190, 2012.

[14] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "Droidmat: Android malware detection through manifest and api calls tracing," in *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*. IEEE, 2012, pp. 62–69.

[15] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, pp. 15–26.

[16] T. M. Corporation, "Malware attribute enumeration and characterization." [Online]. Available: http://maecproject.github.io/about-maec/

[17] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov, "Learning and classification of malware behavior," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2008, pp. 108–125.

[18] M. Lindorfer, C. Kolbitsch, and P. M. Comparetti, "Detecting environment-sensitive malware," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2011, pp. 338–357.

[19] I. Santos, J. Devesa, F. Brezo, J. Nieves, and P. G. Bringas, "Opem: A static-dynamic approach for machine-learning-based malware detection," in *International Joint Conference CISIS'12-ICEUTE 12-SOCO 12 Special Sessions*. Springer, 2013, pp. 271–280.

[20] N. Sun, P. Winkler, C. Chu, H. Jia, J. Geffner, T. Lee, J. Mody, and F. Swiderski, Patent, 2006. [Online]. Available: https://patents.google.com/patent/US9996693B2/en

[21] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario, "Automated classification and analysis of internet malware," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2007, pp. 178–197.

[22] F. Doshi-Velez and B. Kim, "Towards a rigorous science of interpretable machine learning," *arXiv preprint arXiv:1702.08608*, 2017.

[23] A. Vellido, J. D. Martín-Guerrero, and P. J. Lisboa, "Making machine learning models interpretable." in *ESANN*, vol. 12. Citeseer, 2012, pp. 163–172.

[24] J. Hainmueller and C. Hazlett, "Kernel regularized least squares: Reducing misspecification bias with a flexible and interpretable machine learning approach," *Political Analysis*, vol. 22, no. 2, pp. 143–168, 2014.

[25] X. Chen, Y. Duan, R. Houthooft, J. Schulman, I. Sutskever, and P. Abbeel, "Infogan: Interpretable representation learning by information maximizing generative adversarial nets," in *Advances in neural information processing systems*, 2016, pp. 2172–2180.

[26] "Objdump." [Online]. Available: http://sourceware.org/binutils/docs/binutils/objdump.html

[27] "Geeksforgeeks website." [Online]. Available: https://www.geeksforgeeks.org/

[28] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "{BYTEWEIGHT}: Learning to recognize functions in binary code," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 845–860.

[29] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[30] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[31] T. L. Griffiths, M. I. Jordan, J. B. Tenenbaum, and D. M. Blei, "Hierarchical topic models and the nested chinese restaurant process," in *Advances in neural information processing systems*, 2004, pp. 17–24.

[32] J. Yuan, F. Gao, Q. Ho, W. Dai, J. Wei, X. Zheng, E. P. Xing, T.-Y. Liu, and W.-Y. Ma, "Lightlda: Big topic models on modest computer clusters," in *Proceedings of the 24th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2015, pp. 1351–1361.

[33] M. Claesen and B. D. Moor, "Hyperparameter search in machine learning," *CoRR*, vol. abs/1502.02127, 2015. [Online]. Available: http://arxiv.org/abs/1502.02127

[34] J. Chang, S. Gerrish, C. Wang, J. L. Boyd-Graber, and D. M. Blei, "Reading tea leaves: How humans interpret topic models," in *Advances in neural information processing systems*, 2009, pp. 288–296.

[35] Y. W. Teh, M. I. Jordan, M. J. Beal, and D. M. Blei, "Sharing clusters among related groups: Hierarchical dirichlet processes," in *Advances in neural information processing systems*, 2005, pp. 1385–1392.

[36] V. Singh, "Guided lda," 2017. [Online]. Available: https://guidedlda.readthedocs.io/en/latest/

Jeremiah Greer received in B.S. degree in Computer Science from the University of Cincinnati, Cincinnati, Ohio, in 2018, and is currently pursuing his M.S. degree in Computer Science at the University of Cincinnati. He has accepted a position at Microsoft in Redmond for Fall 2019. His current research interests include natural language processing, neural networks, software and malware analysis, and artificial intelligence, particularly in game AI.

Rashmi Jha is an Associate Professor in Electrical Engineering and Computing Systems Department at the University of Cincinnati. She worked as a Process Integration Engineer for Advanced CMOS technologies at IBM Microelectronics, between 2006-2008. She finished her Ph.D. and M.S. in Electrical Engineering from North Carolina State University in 2006 and 2003, respectively, and B.Tech. in Electrical Engineering from IIT Kharagpur, India in 2000. She has been granted 12 US patents and has authored/co-authored several publications. She has been a recipient of Summer Faculty Fellowship award from AFOSR in 2017, CAREER Award from the National Science Foundation (NSF) in 2013, IBM Faculty Award in 2012, IBM Invention Achievement Award in 2007. She is the director of Microelectronics and Integrated-systems with Neuro-centric Devices (MIND) laboratory at the University of Cincinnati. Her current research interests lie in the areas of Artificial Intelligence, Cybersecurity, Neuromorphic SoC, Emerging Logic and Memory Devices, Hardware Security, and Neuroelectronics.

Anca Ralescu is a Full Professor in the EECS Department, University of Cincinnati, which she joined in 1983. She holds degrees in Mathematics from University of Bucharest, Romania (BS, 1972), and Indiana University, Bloomington (MA, 1980, PhD 1983). She currently heads the Machine Learning and Computational Intelligence Laboratory in the EECS Department, where with her students she is involved in machine learning research with applications to image understanding, text understanding, and cyber-security. Other research interests include brain-computer interface, and artificial intelligence. During 1991-1995 Dr. Ralescu was the Assistant Director of the Laboratory for International Fuzzy Engineering, Yokohama, Japan. She held visiting positions at various universities, including University of Bristol, UK, Tokyo Institute of Technology, University of Oviedo, Spain, Osaka University, and Paris-Tech ENST, France.

Temesguen Messay-Kebede is a former faculty in the Electrical and Computer Engineering at the University of Dayton. He is currently a Research Engineer for the Avionics Cyber Protection Team in the Avionics Vulnerability Mitigation Branch, Sensors Directorate, at Wright-Patterson Air Force Base in Dayton, Ohio. He studied Electrical and Computer Engineering at the University of Dayton and he obtained his Ph.D. in December 2014. His research areas include pattern recognition, machine learning, image processing and understanding, robotics and cyber-security.

David Kapp is the Avionics Cyber Protection Team Lead in the Avionics Vulnerability Mitigation Branch, Sensors Directorate, at Wright-Patterson Air Force Base in Dayton, Ohio. Dr. Kapp has spent the last eighteen (18) years performing research and development in novel software protection and anti-tamper solutions for the DoD. His passion is building biologically-inspired adaptable and resilient cyber protection systems. He holds a Ph.D. in Electrical Engineering from Virginia Tech, where he specialized in electromagnetic scattering from randomly rough surfaces.