ARL-TR-8815 ● SEP 2019

Unsupervised Learning of Library Routines to Predict Function

by Anne Logie and Michael S Lee

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TR-8815 ● SEP 2019

Unsupervised Learning of Library Routines to Predict Function

Anne Logie and Michael S Lee
Computational and Information Sciences Directorate,
CCDC Army Research Laboratory

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
September 2019	Technical Report	October 2018–August 2019

4. TITLE AND SUBTITLE
Unsupervised Learning of Library Routines to Predict Function

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Anne Logie and Michael S Lee

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
CCDC Army Research Laboratory
ATTN: FCDD-RLC-NB
Aberdeen Proving Ground, MD 21005

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-8815

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
ORCID ID(s): Michael S Lee, 0000-0002-0419-6069

14. ABSTRACT

Since malware is a constantly evolving threat, it requires significant expertise to detect, identify, and mitigate. We postulate that deep learning can be adapted to this problem domain to provide automated analysis of arbitrary binary code to aid cyber analysts in the identification of functional components. As a proof-of-concept, we trained a convolutional autoencoder to reproduce various fields of the disassembled binaries of standard Linux libraries. We then performed clustering on the bottleneck layer to identify possible clusters of similarity among the various routines. Our spot check of 100 routines suggests that deep learning may indeed be useful for routine classification. However, further network-topology refinement and a concerted ground-truth labelling effort will be required to yield a production-level analytical tool.

15. SUBJECT TERMS
deep learning, malware, cyber warfare, Linux, binaries, topic modeling, unsupervised learning, disassembly

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Anne Logie
Unclassified	Unclassified	Unclassified	UU	21	19b. TELEPHONE NUMBER (Include area code) (410) 278-6394

Contents

List of Figures

List of Tables

1. Introduction

Cyber-attacks are a continuously growing threat to computer systems and networks. Traditional malware-identification techniques such as byte hashes are quickly becoming ineffective, as attackers are able to obfuscate malware through methods such as polymorphism (You 2010). To combat this phenomenon, cyber analysts are required to manually detect, identify, and mitigate potential and current threats to the systems and networks they protect. As this job becomes more and more challenging, it will become essential to provide automated analytics tools to streamline their efforts (Le Quan et al. 2018).

In this work, we propose that unsupervised machine learning, specifically multilayered convolutional networks, can be used to help identify the functions of underlying binary code, thereby elucidating the purpose and potential actions of unknown programs found on a system. One of the critical bottlenecks in the development of any such deep-learning-based tool is the massive ground-truth labeling of known data—in this case, function-category assignments for disassembled routine binaries. Therefore, in this work we explore the use of convolutional autoencoders to assist in the labeling task as a form of unsupervised learning. State-of-the-art convolutional autoencoders extract clusters that match labelled data quite accurately for the simple Modified National Institute of Standards and Technology (MNIST) handwriting digits data set and moderately well for the more complex Fashion-MNIST data set (Ben-Yosef and Weinshall 2018; Edwards and Lee 2019).

Analyzing disassembled binaries is analogous to the established field of natural language processing. Language modeling, for example, requests the following, "given N previous tokens (characters/words), predict the next token." Topic modeling, closely related to this work, asks the question, "Given N tokens, what is the title or description of this set?" Supervised topic modeling is fairly common, but often boils down to the detection of a few keywords that hint to the topic. In the case of assembly language, this simplification is not as likely given that the same operation codes (opcodes), such as `MOV` and `ADD`, are found in all function types. Perhaps sequences (Kang et al. 2016), subsets of opcodes, and histograms of opcode usage (Shabtai et al. 2009; Shabtai et al. 2012) are more indicative.

1.1 Challenges

Our task is full of challenges. Each routine may be a composition of a rather large tree of dependencies: a typical function calls other functions, which, in turn, calls other functions, and so on. Analyzing static code only focuses on the central

processing unit (CPU) operations within the routine and its function calls, but not the code associated with the functions it calls. This fact reduces the precision of both the automated and manual labeling procedures. In addition, the code of a binary has major traits beyond its function including the following:

1) Authorship: who actually wrote it, which could be one or more people;
2) Ancestry: where this code was derived from;
3) Programming language: most likely C or C++, but could be any other of a number of popular languages;
4) Compiler: different compilers produce different binary instructions for the same source code; and
5) Templates: routines within a particular library/package may share common features.

With these challenges in mind, we also had to choose a machine-learning framework that could manage the high nonlinear dimensionality of this problem. Convolutional autoencoders, along with generative adversarial networks (GANs), are typically used to denoise and improve the resolution of data such as images and audio. Their bottleneck layer, also known as the "latent space", often provides a low-dimensional semantic-rich vector that can be introduced into any modern dimension reduction or clustering algorithm. Here, we cluster the latent space to a specified number of groups and investigate whether these clusters match up with our intuition about routine functions while realizing that other traits such as authorship, and so on, could also be critical components of the latent space.

1.2 Previous Machine Learning Work in the Cyber Domain

Most machine-learning efforts for malware detection in the literature involve an initial step of manually defining features. A few recent works take advantage of advances in deep learning, namely convolutional neural networks (CNNs) that can learn features automatically (McLaughlin et al. 2017; Kolosnjaji et al. 2016; Huang and Kao 2018; Cao et al. 2018). This idea is parallel to the use of CNNs and recurrent neural networks (NNs) to encode, analyze, and predict natural language text (Young et al. 2018). Regarding supervised learning, Shin et al. (2015) used NNs to recognize the start/stop indices of functional blocks in binaries. Lee (2018) demonstrated that a CNN could classify opcode streams for five program classes. Raff et al. (2018) demonstrated classification of malware versus goodware for 2 million programs using gated CNNs. Gibert (2016) used image-based CNNs to classify nine virus families from a 2014 Microsoft Kaggle Competition

(Ronen et al. 2018). CNNs are also used to identify authorship of natural language text (Hitschler et al. 2017; Shrestha et al. 2017). Extension of the same authorship-attribution methods to assembly language should be possible.

2. Methods

2.1 Preprocessing Data Sets

The raw data set in this work was derived from the standard libraries found in the `/usr/lib64` directory of the CentOS 7 Linux distribution. Preprocessing was required to prepare the raw data set for five separate experiments. The raw data set contained multiple lib64 assembly routines that were disassembled and parsed into experiment-specific input formats in two preprocessing phases.

The first phase used objdump, a command-line program that displays information from object files, to disassemble lib64 object files and generate a disassembly report (Fig. 1). This disassembly report includes multiple lines that collectively comprise machine-code instructions compiled from a higher-level programming language. Each line contains information about a single machine instruction and can be decomposed further into five separate columns. The first column contains the code's memory address. The second column contains the executable machine instruction in hex (a hexadecimal number). The next two columns consist of the machine instructions translated into assembly language: the third column representing the opcode assembly language mnemonic and the fourth column representing the operands. An opcode portion of the machine instruction specifies what operation the CPU must perform on the operands, or arguments. The fifth column contains comments.

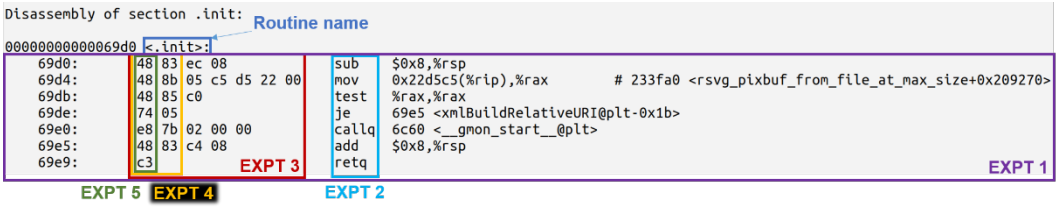

Fig. 1 First few lines of a typical disassembly report and outlines of regions considered as input for various experiments

The second phase involved parsing and preparing each disassembled routine report into their corresponding experimental input format. Each experiment required different information from the lines of the disassembly report. The first experiment (E1) included the full line, while the second experiment (E2) included only the assembly language mnemonic opcode. The third experiment (E3) included the full

executable machine instructions, where the hexadecimal was converted to bytes. The fourth experiment (E4) included the first two bytes of the executable machine instruction, and the fifth experiment (E5) included only the first byte of the executable machine instruction. Figure 1 visually outlines the specific experimental inputs. The parsing process extracted this information from each line of the disassembled routine report and converted the information gathered for each disassembled routine into an input format the NN could digest. The information gathered consisted of a list of symbols. The symbol value was based on ASCII codes in E1 and E2, and bytes in E3–E5. Inputs to the autoencoder were always a list of bytes (`uint8`). Experiment-specific sample length requirements reduced the number of usable routines for each experiment. Specifically, this reduction process eliminated data samples (routines) less than the length requirement and trimmed data samples greater than the length requirement. Small routines were discarded since they are likely doing trivial operations like wrapping another routine. Because the fixed-length data samples for each experiment covered a different combination of features and columns, experiments that gathered more information per line had a smaller average routine line count than experiments that gathered less information per line. Finally, we used a Keras utility to one-hot encode data samples based on the number of unique symbols in the experiment. Experimental data sets did not exceed 200,000 disassembled routines. Table 1 summarizes experiment specification details.

Table 1 Input data for each experiment

Name	Input format	No. of routines	No. of symbols per sample	Average line count	No. of unique symbols
E1	All ASCII characters from the disassembly routine report	140541	1024	281	126
E2	ASCII characters from the routine's opcode mnemonics separated by spaces	116368	128	318	123
E3	All instruction bytes	106795	128	361	256
E4	First two instruction bytes	66377	128	552	256
E5	First instruction byte	36356	128	931	256

2.2 Autoencoder Model Topology

Next, we trained a convolutional autoencoder NN in Keras (Chollet 2015) with a backend of TensorFlow (Abadi et al. 2016) to reproduce the input data sets and generate a latent space for later clustering and visualization. The model topology contained mirroring encoder and decoder networks as seen in Fig. 2, sharing similarity to our previous model used for images (Edwards and Lee 2019). Each experiment's model had six convolutional layers with respective 30, 50, 70, 90, 110, and 130 filters that were transposed reflections of each other. The topology also included one intermediate dense layer with a dimension of 20. Each convolution layer used a kernel size of 4 and stride operation of 2, represented as 4/2 (downsample) and 4×2 (upsample via strided-transposed convolution) in Fig. 2. The encoder and decoder layers used an x-Gaussian activation function, $f(x) = xe^{-x^2}$, as introduced in the Edwards and Lee paper (2019), and the dense layer used a linear activation function to maximize the range of the latent space. The last layer for each model had a channel count equal to the number of desired symbols to generate a categorical one hot vector (kernel size = 1, stride = 1 and softmax activation function). Each model was compiled with the categorical cross entropy loss function and the Adam optimizer (Kingma and Ba 2014).

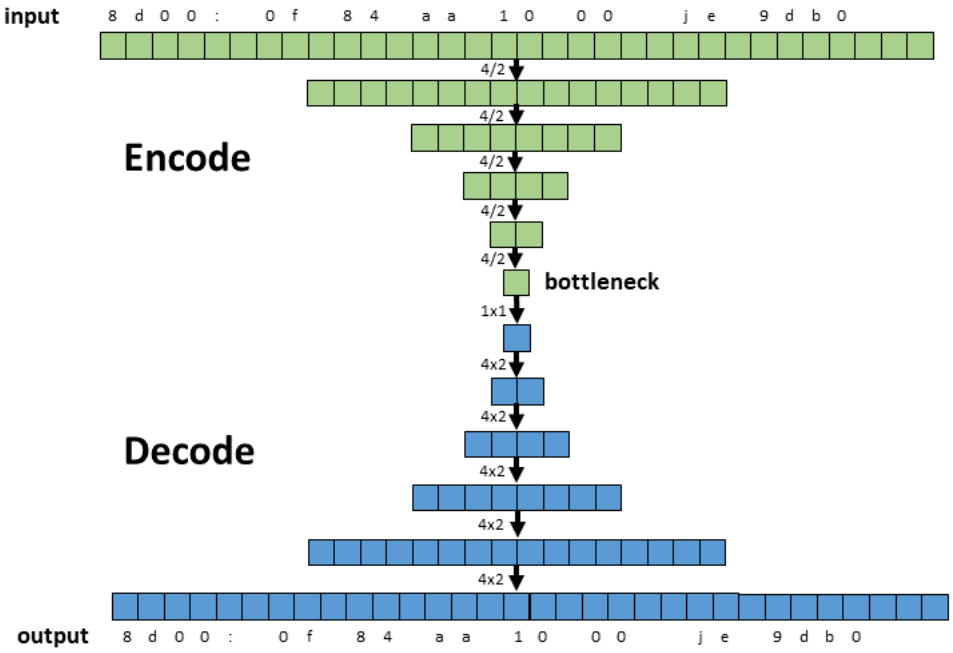

Fig. 2 Schematic of convolutional autoencoder topology used in this work

2.3 Clustering and Semisupervised Evaluation

The latent space of autoencoders provides a sophisticated input for other unsupervised learning methods like clustering and dimension reduction. Even though the purpose of the autoencoder is to reproduce input data, it tends to learn a useful low-dimensional latent space in its bottleneck layer. It is often argued that similar input data should project into similar locations in the latent space. We clustered the latent space with a Gaussian mixture model (GMM) and specified the formation of 12 clusters. GMMs (McLachlan and Basford 1988) generate multidimensional ellipsoids that are more adaptive than spherically constrained k-means clusters. Then we visualized these clusters in a 2-D space using the powerful t-Distributed Stochastic Neighbor Embedding (t-SNE) dimensionality reduction algorithm (van der Maaten and Hinton 2008).

Next, we performed a second type of analysis by actually labeling some of the data by hand. Specifically, we conjectured that there are 12 functional routine categories (Table 2) that a deep-learning clustering algorithm might detect including basic operations, data, memory, file, process, system, network, graphics, math, encryption, compression, and other algorithms. We selected these classes based on our intuition and Microsoft's website for run-time routines by category (Microsoft 2019). Then, we hand-labeled 100 randomly selected routines from our data set with one of these categories.

We compared our hand-labeled assembly routines to their GMM-predicted cluster indices to determine whether our belief of how the assembly routines should cluster aligned with how they actually clustered. We used the Hungarian algorithm to solve the linear sum assignment problem to optimally match the assigned clusters to the predicted clusters with the `scipy.optimize` function, `linear_sum_assignment`. Then, we determined how well the 100 randomly selected assembly routines aligned between manual and predicted labels. Most of the experiments had fewer than 100 samples because some of the samples were eliminated during preprocessing for being too small.

Table 2 Routine categories manually assigned in this work

Index	Class name	Description
1	Basic operations	Type casting, definitions, and other simple operations
2	Data	Sets with one or more variables
3	Memory	Large-scale manipulation of memory including clearing, allocation, and transformation
4	File	Access–Input/Output for files, storage, external storage devices
5	Process	Sharing/communication between processes/applications
6	System	Any system calls not part of other categories (e.g., setting time; manipulating system variables or more sophisticated system functions)
7	Network	Operations that involve two or more computers
8	Graphics	Pixels, video, lines, fonts, transforms of the graphical memory vs. changing a single graphical user interface (GUI) variable
9	Math	Mathematical functions
10	Encryption	Cryptographic functions
11	Compression	Data-compression functions
12	Algorithm	Any algorithm that does not fit neatly into existing categories

3. Results

3.1 Visualization of Clusters

Clustering results (Fig. 3) were based on the different portions of the disassembled report included as input in the five different experiments (E1–E5). Each experiment iterated and parsed the disassembled routine report by line, but extracted a different portion of each line. Recall that the portions included by the experiments: E1) full line, E2) mnemonic opcode, E3) full executable machine instruction in bytes, E4) first two bytes of the executable machine instruction, and E5) first byte of the executable machine instruction. E1 and E3 included more information per report line than the other experiments and their t-SNE presented the least visibly discrete clusters. E2, E4, and E5 included less information per report line and focused more on clustering opcode information. This fine-grained focus placed on opcode-only information appeared to improve clustering and visualization. Of all experiments, E5, with only one byte per instruction line considered, displayed the most visually distinctive clusters.

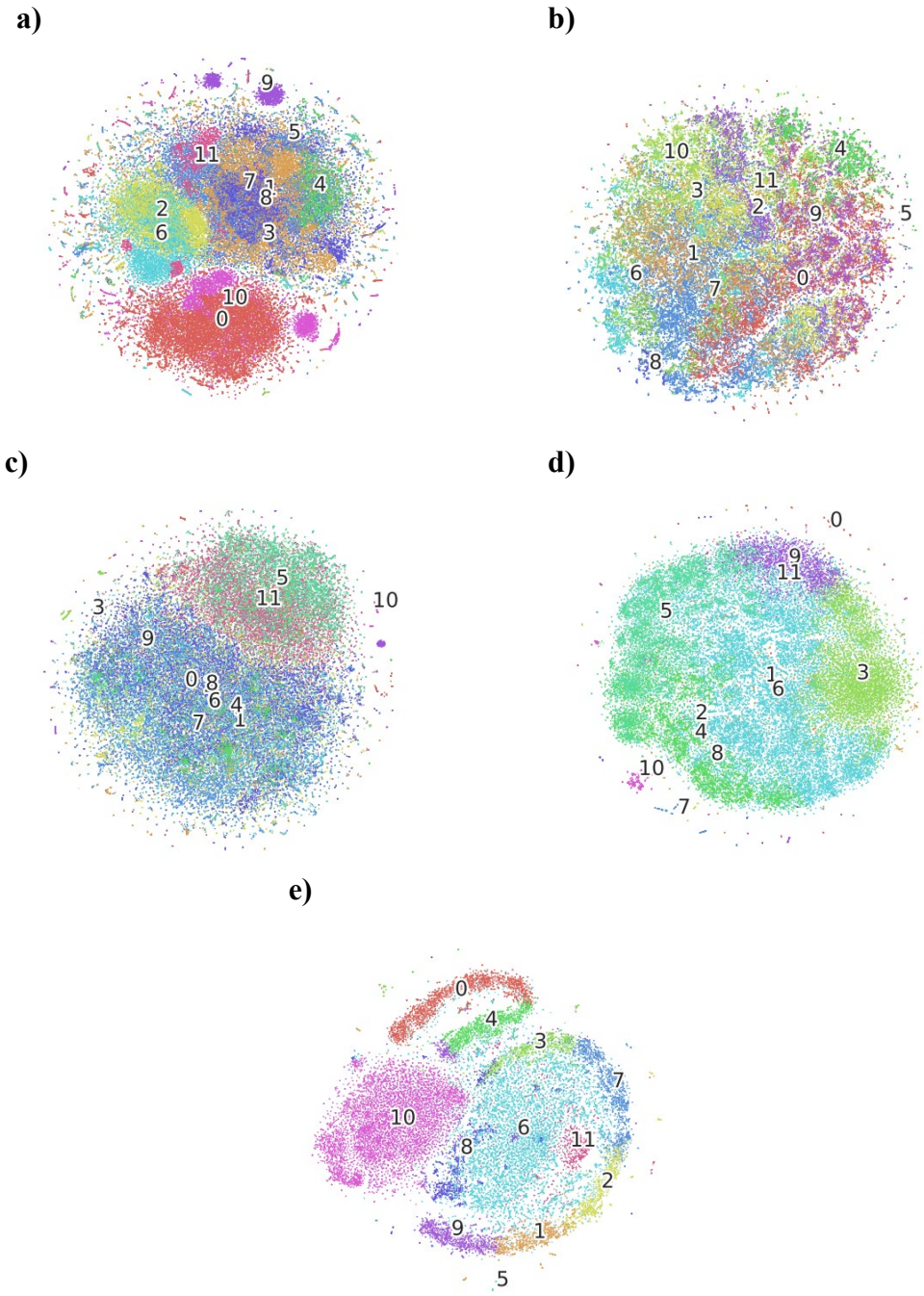

Fig. 3 2-D *t*-SNE visualization of GMM clusters of the autoencoder latent space for a) E1: full report, b) E2: opcode mnemonic only, c) E3: full binary, d) E4: first two instruction bytes, and e) E5: first instruction byte

3.2 Matching Clusters to Labels via Linear Sum Assignment

The 100 hand-labeled assembly routines were compared to their corresponding cluster assignment to determine to what extent the routines were clustering based on function. The linear-sum-assignment algorithm attempted to match cluster

index and hand-labeled index to optimize a cost value/score as seen in Table 3. A perfect score would equal the sample size multiplied by negative one. The worst possible score would be roughly the sample size divided by the number of clusters. Table 3 lists the number of data samples included in this analysis and each experiment's optimal cost value. The agreement between data sets in each experiment performed better than random, but the evidence suggested only weak correlations. E4 and E5 appeared to generate clusters that most closely correlated with our handcrafted function labels.

Table 3 Linear-sum-assignment results

Experiment	No. of samples	Optimal cost value	Effective accuracy
E1	100	–23	23%
E2	86	–20	23%
E3	80	–23	29%
E4	46	–16	35%
E5	30	–11	37%

4. Discussion

While our experiments and results only scratch the surface, they provide some insights for future exploration. First, we find that the best input feature to characterize routines appears to the first byte of each instruction. This makes intuitive sense, in that it limits analysis to the opcodes, which others have found to be the most informative (Lee 2018). Second, beyond opcode mnemonics, the first byte in x86 instructions often groups multiple similar opcodes.

Another observation we made is that there appears to be a correlation between the aesthetics of the cluster visualization and the clustering accuracy relative to our manually labeled ground truth. This suggests that internal consistency between two unsupervised metrics (manifold reduction and clustering) is predictive of external consistency with human-intuited groupings.

Regarding the "low" 37% accuracy we find, it should be noted that we would not expect 100% accuracy compared with our function label given the other characteristics of authorship, programming language, compiler, and so on. Also, it is fully expected that a supervised learning approach with hand-curated labels would provide a significant accuracy boost in the same way that going from

unsupervised to supervised learning improves accuracy from 60% to 90% in Fashion-MNIST (Ben-Yosef and Weinshall 2018).

In x86 binaries, the opcode bytes are nontrivial to parse because they vary in length. Specifically, our experiments do not account for the variable byte length of the opcodes. Experiments E4 and E5 extracted the same number of bytes only from the beginning of each instruction. Further experiments should investigate extracting the correct number of bytes associated with each opcode. This may perform better than examining only the first one or two bytes of each instruction.

Finally, one should consider some other caveats of this work:

1) All of the libraries in this study were "shipped" with the Linux operating system; thus, the tasks they perform may be narrow compared with the universe of tasks that executables run.

2) All of the libraries in this study are open source and nonproprietary with a majority having noncommercial authorship. Compared with malware, the programming style could be very different.

3) Some real-life malware may have a completely different "supply chain" in terms of the libraries and functions it uses; some may not even call external libraries.

4) Some aspects of Linux binaries are fundamentally different from Windows binaries, where different types of malware are more prevalent.

5. Conclusion

In this work, we grouped library routines based on their binaries and disassembled reports using deep-learning autoencoders and a standard unsupervised clustering algorithm. Of the many combinations of features considered, we found the first byte of each instruction line to be the most informative for the purposes of manifold reduction, clustering, and correspondence to known function. To push the field of automated function analysis forward, future work should consider 1) further improving the ability of autoencoders to cluster and organize data and 2) supervised training against a larger set of routines with hand-labeled functions.

6. References

Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M, et al. TensorFlow: a system for large-scale machine learning. In: Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16); 2016 Nov 2–4; Savannah, GA. Washington, DC; Berkeley (CA): USENIX Association; c2016. p. 265–283.

Ben-Yosef M, Weinshall D. Gaussian mixture generative adversarial networks for diverse datasets, and the unsupervised clustering of images; 2018 Aug 30; arXiv preprint arXiv:1808.10356.

Cao D, Zhang X, Ning Z, Zhao J, Xue F, Yang Y. An efficient malicious code detection system based on convolutional neural networks. In: CSAI '18. Proceedings of the ACM 2018 2nd International Conference on Computer Science and Artificial Intelligence; 2018 Dec 8–10; Shenzhen, China. New York (NY): Association for Computing Machinery; c2018. p. 86–89.

Chollet F. Keras; 2015. [accessed 2019 Sep 24] https://github.com/fchollet/keras.

Edwards SN, Lee MS. Using convolutional neural network autoencoders to understand unlabeled data. In: Pham T, editor. SPIE Defense + Commercial Sensing; Proceedings Vol. 11006, Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications; 2019 Apr 14–18; Baltimore, MD. Bellingham (WA): SPIE-International Society for Optics and Photonics; c2019.

Gibert LD. Convolutional neural networks for malware classification. [master's thesis]. [Barcelona (Catalonia, Spain)]: Universitat Politècnica de Catalunya; 2016.

Hitschler J, van den Berg E, Rehbein I. Authorship attribution with convolutional neural networks and POS-eliding. In: Proceedings of the Workshop on Stylistic Variation; 2017 Sep; Copenhagen, Denmark. Stroudsburg (PA): Association for Computational Linguistics. c2017. p 53–58.

Huang TT, Kao HY. R2-D2: color-inspired convolutional neural network (CNN)-based android malware detections. In: 2018 IEEE International Conference on Big Data; 2018 Dec 10–13; Seattle, WA. Washington (DC): IEEE Computer Society; c2018.

Kang BJ, Yerima SY, Sezer S, McLaughlin K. N-gram opcode analysis for android malware detection. Int J Cyber Situation Aware. 2016;1(1):231–254. arXiv preprint arXiv:1612.01445.

Kingma DP, Ba J. Adam: a method for stochastic optimization. In: ICLR 2015. Proceedings of the 3rd International Conference for Learning Representations; 2015 May 7–9; San Diego, CA. arXiv preprint arXiv:1412.6980; 2014.

Kolosnjaji B, Webster G, Zarras A, Eckert C. Deep learning for classification of malware system call sequences. In: Kang BH, Bai Q, editors. AI 2016: Advances in Artificial Intelligence. Proceedings of 29th Australasian Joint Conference on Artificial Intelligence; 2016 Dec 5–8; Hobart, TAS, Australia. Cham, Switzerland: Springer International Publishing AG; c2016. p. 137–149.

Le Q, Boydell O, Namee BM, Scanlon M. Deep learning at the shallow end: malware classification for non-domain experts. In: DFRWS USA 2018. 18th Annual Digital Forensics Research Workshop Conference; 2018 July 15–18; Providence, RI. Digital Investigation 26. 2018:S118–S126.

Lee Michael S. Convolutional neural networks for functional classification of opcode sequences. In: Blowers M, Hall RD, Dasari VR, editors. SPIE Defense + Security; Proceedings Vol. 10652, Disruptive Technologies in Information Sciences; 2018 Apr 15–19; Orlando, FL. Bellingham (WA): SPIE-International Society for Optics and Photonics; c2018.

McLachlan GJ, Basford KE. Mixture models: inference and applications to clustering. Statistics, Textbooks and Monographs, Vol. 84. New York (NY): M. Dekker; 1988.

McLaughlin N, Martinez Del Rincon J, Kang B-J, Yerima S, Miller P, Sezer S, Safaei Y, Trickel E, Zhao Z, et al. Deep android malware detection. In: CODASPY '17. Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy; 2017 Mar 22–24; Scottsdale, AZ. New York (NY): Association for Computing Machinery; c2017.

Microsoft. Universal C runtime routines by category. [accessed 2019 Sep 18] https://docs.microsoft.com/en-us/cpp/c-runtime-library/run-time-routines-by-category?view=vs-2019.

Raff E, Barker J, Sylvester J, Brandon R, Catanzaro B, Nicholas CK. Malware detection by eating a whole EXE. In: Workshops of the Thirty-Second AAAI Conference on Artificial Intelligence; 2018 Feb 2–3; New Orleans, LA. Menlo Park (CA): Association for the Advancement of Artificial Intelligence. c2018. p 268–276.

Ronen R, Radu M, Feuerstein C, Yom-Tov E, Ahmadi M. Microsoft malware classification challenge. 2018 Feb 22; arXiv preprint arXiv:1802.10135.

Shabtai A, Moskovitch R, Elovici Y, Glezer C. Detection of malicious code by applying machine learning classifiers on static features: a state-of-the-art survey. Inform Secur Tech Rep; 2009;14(1):16–29.

Shabtai A, Moskovitch R, Feher C, Dolev S, Elovici Y. Detecting unknown malicious code by applying classification techniques on OpCode patterns. Secur Inform. 2012;1(1).

Shin ECR, Song D, Moazzezi R. Recognizing functions in binaries with neural networks. In: Proceedings of the 24th USENIX Security Symposium; 2015 Aug 12–14. Washington, DC; Berkeley (CA): USENIX Association. c2015. p. 611–626.

Shrestha P, Sierra S, González F, Montes M, Rosso P, Solorio T. Convolutional neural networks for authorship attribution of short texts. In: Lapata M, Blunsom P, Koller A, editors. Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers; 2017 Apr; Valencia, Spain. Stroudsburg (PA): Association for Computational Linguistics; c2017. p. 669–674.

You I, Yim K. Malware obfuscation techniques: a brief survey. In: BWCCA 2010. Proceedings of 2010 International Conference on Broadband, Wireless Computing, Communication and Applications; 2010 Nov 4–6; Fukuoka, Japan. New York (NY): Institute of Electrical and Electronics Engineers (IEEE); c2010. p. 297–300.

Young T, Hazarika D, Poria S, Cambria E. Recent trends in deep learning based natural language processing [review article]. IEEE Comp Intel Mag. 2018;13(3):55–75.

van der Maaten L, Hinton G. Visualizing data using t-SNE. J Mach Learn Res. 2008;9:2579 –2605.

List of Symbols, Abbreviations, and Acronyms

2-D	two-dimensional
CNN	convolutional neural network
CPU	central processing unit
GAN	generative adversarial network
GMM	Gaussian mixture model
GPU	graphical processing unit
GUI	graphical user interface
MNIST	Modified National Institute of Standards and Technology
NN	neural network
opcode	operation code
OS	operating system
t-SNE	t-Distributed Stochastic Neighbor Embedding
x86	Intel-based x86 processors

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
1 (PDF)	CCDC ARL FCDD RLD CL TECH LIB
1 (PDF)	GOVT PRINTG OFC A MALHOTRA
10 (PDF)	CCDC ARL FCDD RLC B HENZ T PHAM FCDD RLC N B RIVERA FCDD RLC ND T BRAUN J CLARKE M DE LUCIA G SHEARER M WEISMAN FCDD RLC NT A SWAMI FCDD RLS SA L KAPLAN