Identifying Ubiquitious Third-Party Libraries in Compiled Executables Using Annotated and Translated Disassembled Code with Supervised Machine Learning

Jedediah Haile
Idaho National Laboratory
Idaho Falls, USA
jed.haile@inl.gov

Sage Havens
Idaho National Laboratory
Idaho Falls, USA
sage.havens@inl.gov

***Abstract*—The size and complexity of the software ecosystem is a major challenge for vendors, asset owners and cybersecurity professionals who need to understand the security posture of these systems. Annotated and Translated Disassembled Code is a graph based datastore designed to organize firmware and software analysis data across builds, packages and systems, providing a highly scalable platform enabling automated binary software analysis tasks including corpora construction and storage for machine learning. This paper describes an approach for the identification of ubiquitous third-party libraries in firmware and software using Annotated and Translated Disassembled Code and supervised machine learning. Annotated and Translated Disassembled Code provide matched libraries, function names and addresses of previously unidentified code in software as it is being automatically analyzed. This data can be ingested by other software analysis tools to improve accuracy and save time. Defenders can add the identified libraries to their vulnerability searches and add effective detection and mitigation into their operating environment.**

Keywords—Bayes method, classification algorithms, clustering methods, databases, graph theory, internet, k-nearest neighbor search, machine learning, matrices, neural network, reverse engineering, supervised learning, supply chain management, support vector machines, vector

I. Introduction

Modern open source and proprietary software rarely consist of applications built from scratch, the developers instead reuse third-party libraries to speed up development time and minimize effort by removing the need to rebuild existing software. When a developer uses devices or firmware from a vendor or open source, one might think they are secure, but this isn't always the case. According to a 2012 Forrester survey, out of the 336 software companies surveyed, nearly all work with third-party libraries and less than 50% of them test third-party libraries for security [15], [23]. A 2018 survey from CrowdStrike covering 1,300 IT professionals, found 66% have experienced a software supply chain attack, and 79% believe software supply chain attacks will be the largest cyber threat to their organization within the next three years [10]. The current state of automated reverse engineering does not have adequate security solutions due to a lack of software and firmware auditing capabilities. Identifying common patterns or vulnerabilities between a firmware is difficult due to differences in the underlying processor architecture and compilation methods [2]. Prior attempts to use machine learning to automate common reverse engineering tasks have not been accurate enough to merit production use, and often require source code or the software to be compiled in a specific way. These requirements cannot be practically met.

This paper describes a novel approach for the identification of ubiquitous third-party libraries using Annotated and Translated Disassembled Code (@DisCo) and supervised machine learning. Our research developed a cross architecture feature set which can be extracted from compiled software using the angr disassembler [19], [24], [25]. These engineered feature sets are then used with supervised machine learning to correctly identify the name and origin of third-party functions in firmware

II. Use Cases

A. Augmented Reverse Engineering

Static reverse engineering software at the assembly level is a tedious task where any additional information potentially decreases the time and cost of the outcome. In most cases where a reverse engineer's time is required, the binary has been stripped of all function names. If this same binary contains statically linked libraries, then third-party functions are indistinguishable from program specific functions. The ability to correctly and timely identify these third-party functions within disassemblers like IDA Pro, Binary Ninja and Ghidra, is extremely valuable. The application of @DisCo and a trained model to this unknown binary, will output the address of the unknown function and the matched function name of the identifiable third-party libraries which can then be ingested by one's preferred disassembler for more sophisticated reversing.

B. Supply Chain Modification

One of the greatest threats facing computer systems today is supply chain attacks. As cyber security enhances, it becomes

Department of Energy–Cybersecurity Energy Systems & Emergency Response–Cybersecurity of Energy Delivery Systems (DOE-CESER-CEDS) funded Firmware Indicator Translation (FIT) Project

harder for attackers to gain a foothold, and the effort of attack changes focus to upstream from the product [3]. For the sake of this paper, an attacker then maliciously affects a third-party library or application the product trusts and uses. A well-known case of a supply chain attack is NotPetya, where a nation state hijacked Linkos Group's computer update server by exploiting a vulnerability in the content management system. This resulted in a hidden back door which propagated to thousands of computers, and was eventually used to introduced NotPetya, the malicious software which shut down a fifth of the world's shipping capacity [4]. Another example is XcodeGhost, when a backdoored version of Xcode, the IDE used for Apple application development, was posted as a download across popular Chinese forums [2]. This backdoored version of Xcode compiled in an information gathering command and control server within the mobile applications. At the time of discovery, over 4,000 apps within the App Store were infected with this malicious software [2].

A scalable method for storing program binaries in a database and the ability to recognize vulnerable or changed third-party libraries over time, aids in identifying system modification, early detection, and supply chain management. @DisCo hashes and disassembles each binary it ingests, allowing for a specific and simple way to see changes in a binary over time. @DisCo analysis identifies ubiquitous libraries to enable the defenders to add these libraries to their vulnerability searches and work to detect when the vulnerabilities are exploited or include less vulnerable libraries or functions into their operating environment.

C. Software Assurance

Older legacy computer systems often do not take cyber security into consideration, as computer crime was rare during their development and many developers failed to anticipate a future where the internet would be pervasive. While many of these systems have retired, the term *legacy system* still refers to any outdated system still in use [1]. When it comes to securing a legacy system, the first step is to know what the legacy system's software contains. Often this is impossible as the original developers are no longer available, the system came from a third-party vendor, or the documentation is lacking. A straightforward method for identifying third-party libraries within the software is an invaluable starting point. If the legacy system is without updates to necessary components, there will inevitably be vulnerabilities which must be either mitigated or hardened against. Timely identification of vulnerabilities within critical legacy systems is key.

III. @DISCO

A. Graph Database

Current tools for binary analysis and reverse engineering have no consistency, no way to capture, preserve, and share information, and are not scalable solutions. @DisCo is an answer to this problem. @DisCo captures the hashes of each binary and then begins disassembling, and breaking the binary down into functions, blocks, and feature sets. It stores data about each of these components in an OrientDB graph database [20]. @DisCo then finds each edge, and edge type between these components, finally storing them to create a complete and queryable control flow graph. This graph database can then be used to capture additional information about a binary, preserve the state of the binary at specific times, and is easily shared.

B. Breaking Down a Binary

The initial disassembly of the binary is done by the open source tool, angr [19], [24], [25]. angr's graph analysis plugin, CFGFast, creates a knowledge base containing control flow graphs, call graphs, and other valuable information.

To begin, @DisCo creates a base library vertex storing the hash, architecture, base path and version of the binary. Next, it stores every function identified by angr. Each function vertex will consist of the library record id (rid) it belongs to, the function address, function name. Each function vertex is linked to its library by an edge. As each function is processed, the function's blocks will be iteratively processed. The block vertex contains the library rid and function rid it belongs to, the starting address, and label or name for the block and a representation of the block's instructions. This same block vertex also contains the block's instructions in PyVex [25], angr's low-level intermediate language. Each block contains call edges to any functions called from the block, feature edges to feature vertices contain features related to the block, and various types of jump edges which indicate how control flow may move beyond the block.

Fig. 1 contains the source code of a simplistic hello world program. For this example, this program is compiled using GNU Compiler Collection (GCC) with no extra flags. This binary is processed by @DisCo and entered into the OrientDB database. Fig. 2 shows the CFG created by a simple query within OrientDB to display the function *main*. Fig. 2 begins with the base vertex for the binary labeled with record identifier #170:0, and then #229:0 is the function object for *main*. The main function consists of a singular block, #164:0, which has a feature set. This same block makes one call to *printf*, #217:0. This call consists of a singular block and feature set.

```
#include "stdio.h"

int main(){
        printf("hello world");
        return 0;
}
```

Fig. 1. Source code of hello word program

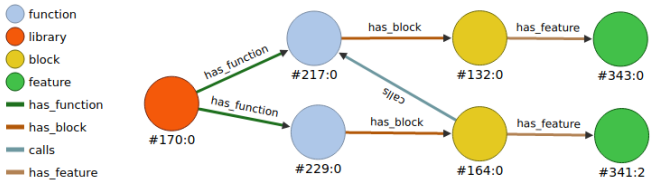

Fig. 2. CFG of hello word program

C. Feature Sets

Each block object stores the low-level intermediate language of the instructions. This language is an abstraction from the assembly level and the high-level languages, such as source code. The use of a low-level language is important as it differs from assembly by not being architecture dependent. This aspect

allows for function identification to be compilation and architecture independent. Fig. 3 is the x86 assembly of the main program from Fig. 1. Fig. 4 displays PyVex's [25] low-level interpretation of the same block of code. It is this low-level language that the feature set is extracted from. PyVex is fundamentally built from statements which are made up of expressions. These statements and expressions can be tokenized into 37 distinct token types. @DisCo tokenizes each block's PyVex interpretation and counts how many times each distinct token is used. Fig. 5 demonstrates a subset of the feature set of the block instructions shown in Fig. 4. This feature set has omitted the tokens which do not appear within the block for brevity.

```
main:
nop     edx, edi
push    ebp
mov     ebp, esp
lea     edi,[0x0040115d] {"hello world"}
mov     eax, 0x0
call    printf
mov     eax, 0x0
pop     ebp
retn
```

Fig. 3. X86 disassembly of hello world program

```
---- IMark(0x401149, 4, 0) ----
---- IMark(0x40114d, 1, 0) ----
t0 = GET:I64(offset=56)
t1 = GET:I64(offset=48)
t2 = Sub64(t1,0x00000008)
PUT(offset=48) = t2"
STle(t2) = t
---- IMark(0x40114e, 3, 0) ----
PUT(offset=56) = t2
---- IMark(0x401151, 7, 0) ----
PUT(offset=72) = 0x00402004
---- IMark(0x401158, 5, 0) ----
PUT(offset=16) = 0x00000000
PUT(offset=184) = 0x0040115d
---- IMark(0x40115d, 5, 0) ----
t3 = Sub64(t2,0x000000008)
PUT(offset=48) = t3
STle(t3) = 0x00401162
t4 = Sub64(t3,0x00000080)
==== AbiHint(0xt4, 128, 0x00401050) ====
```

Fig. 4. PyVex interpretation of main block

```
Ist_imark       6
Ist_abihint     1
Ist_rdtmp       9
Ist_wrtmp       5
Ist_store       2
Ist_put         6
Iex_get         2
Iex_const       7
Iex_binop       3
```

Fig. 5. Feature set of main block

IV. Supervised Machine Learning

@DisCo can produce different types of corpora for machine learning tasks which are valuable in supply chain management such as identification of software components and libraries included in firmware, change detection and forensics. We focus here on some common machine learning techniques which can be useful when analyzing software.

The data set used in this section was created by choosing seven programs which each use a small set of external libraries: libc, libpthread, libdl, libkdump and ld-linux. These programs were statically compiled using GCC 7.4.0 and Clang 6.0.0 with four levels of optimization flags on Ubuntu 18.04 Linux system. This procedure resulted in 56 binaries, eight for each program.

@DisCo was then used to analyze the 56 binaries along with the dynamic versions of the five shared libraries, for a total of 61 binaries loaded into @DisCo's graph database. Since each of the 56 binaries use the same five shared libraries and have been statically linked, each contains copies of all the exported functions of the five libraries, meaning 57 copies of each shared library function will be represented in @DisCo. There is some duplication of functions across these five libraries, resulting in as many as 61 representations of some functions. For the purpose of the following experiments we selected all functions consisting of three or more blocks, resulting in a corpus of 59,543 function representations.

A. Clustering

The function features discussed in Feature Sets can be examined by the use of clustering algorithms to gain insight into how well the feature sets may work for classification and regression machine learning. Ideally the function features can be made to cluster into distinct clusters representing each function.

We examined clustering using t-distributed Stochastic Neighbor Embedding (TSNE) [16], Agglomerative Clustering with ward and average linkage [14], DBScan [17], and K-Means [8]. We applied scikit-learn's StandardScaler [11] to normalize the data, followed by Principle Component Analysis (PCA) [13]. Fig. 6 shows a scatter plot of 50 randomly sampled classes from the corpora, consisting of 2970 functions. The function representations cluster naturally, the scatter plot does not appear to show nearly 3000 points since many of them stack upon one another. Fig 7 shows the results of agglomerative clustering with ward linkage on the same data. In this plot each data point of the functions is represented by a number from 1 to 50, indicating which function class the point belongs to. Fig. 8 shows a TSNE plot of all functions which shows there is a definite structure to the function representations.

B. Classification

To investigate the possibility of using @Disco function representations to identify the components in software, we explored several different methods of training a classifier to recognize individual functions. We separated the corpora into train and test sets as 80/20 split by random selection.

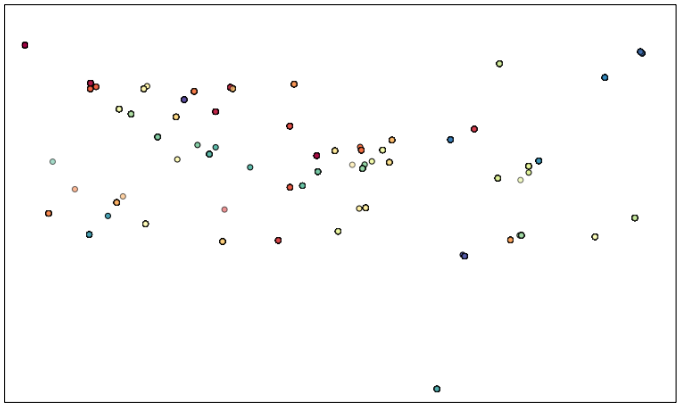

Fig. 6. Random sample of 50 classes with 2970 total samples

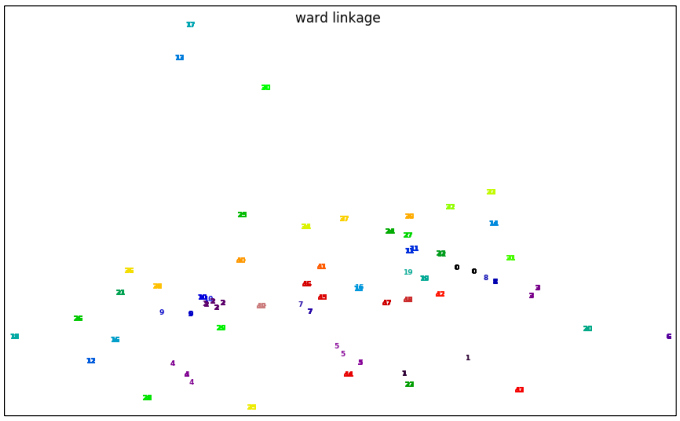

Fig. 7. Agglomerative clustering with ward linkage

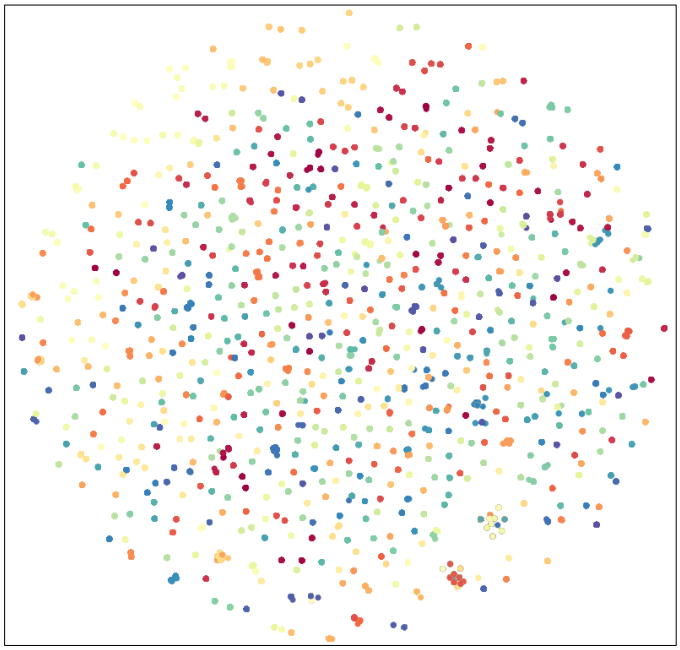

Fig. 8. TSNE Plot of all functions

1) Scikit-learn: First, we compared eight classifiers from scikit-learn using their default parameters. The classifiers were K-Neighbors [9], Linear Support Vector Machine (SVM) [7], Radial Basis Function (RBF) SVM [7], Decision Trees [22], Random Forest [5], Neural Net, AdaBoost [21], and Naïve Bayes [18]. We randomly selected 11,482 functions representing 200 functions for training data, and each classifier was scored using 5-fold cross validated F1. We noticed the F1 scores varying by a large amount across different random selections of data, when we tried this test with the full dataset about half the classifiers would fail to complete. Tuning of each classifier's parameters and batching of data would be required. Table I shows each classifier's F1 score.

TABLE I. *Convolutional Classifier Output*

Classifier	Cross Value F1
K-Neighbors	0.94 (+/- 0.01)
Linear SVM	0.72 (+/- 0.02)
RBF SVM	0.51 (+/- 0.05)
Decision Trees	0.94 (+/- 0.01)
Random Forest	0.94 (+/- 0.01)
Neural Net	0.94 (+/- 0.01)
AdaBoost	0.04 (+/- 0.02)
Naïve Bayes	0.94 (+/- 0.01)

The second experiment was to tune parameters for Support Vector Machines. We chose to use GridSearchCV and support vector classifier (SVC) from sci-kit learn [11]. GridSearchCV runs several trial runs with a classifier, with varying parameters. When the search is complete, the best combination of parameters is selected. The best parameters were RBF kernel, regularization parameter C=1000.0 and kernel coefficient gamma=0.0001. With these parameters SVM yielded a precision score of 0.90, recall 0.92, accuracy and F1 of 0.90 across the full data set, demonstrating that choosing good parameters is a critical aspect. Fig. 9 outlines the SVM classification boundaries.

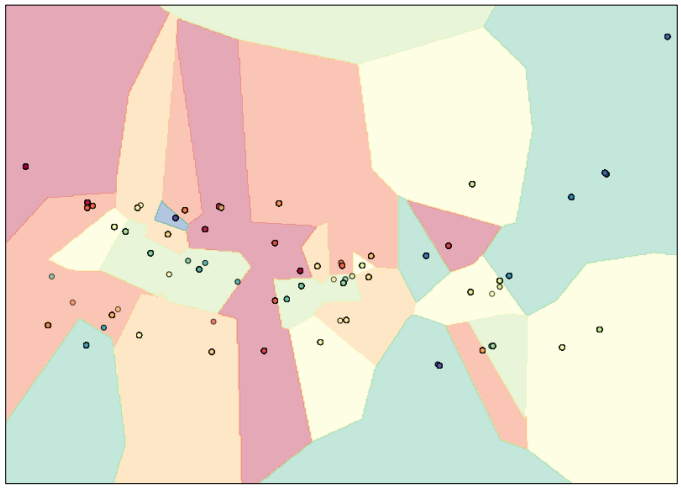

Fig. 9. SVM classification boundaries

2) PyTorch: Looking for better performance we implemented a neural network classifier in PyTorch [6]. The sizes of the encoding linear layers were 834x640, 640x512, 512x256, each layer connected by batch normalization, a rectified linear unit activation function (ReLU) [3] and 20%

dropout, then a final classifier linear layer of size 256x834 with Softmax [12] to choose the final class. All applied sequentially. We used the Adam optimizer with a learning rate of 0.001 and used cross entropy loss. Loss was backpropagated after each batch. This process is demonstrated in Fig. 10.

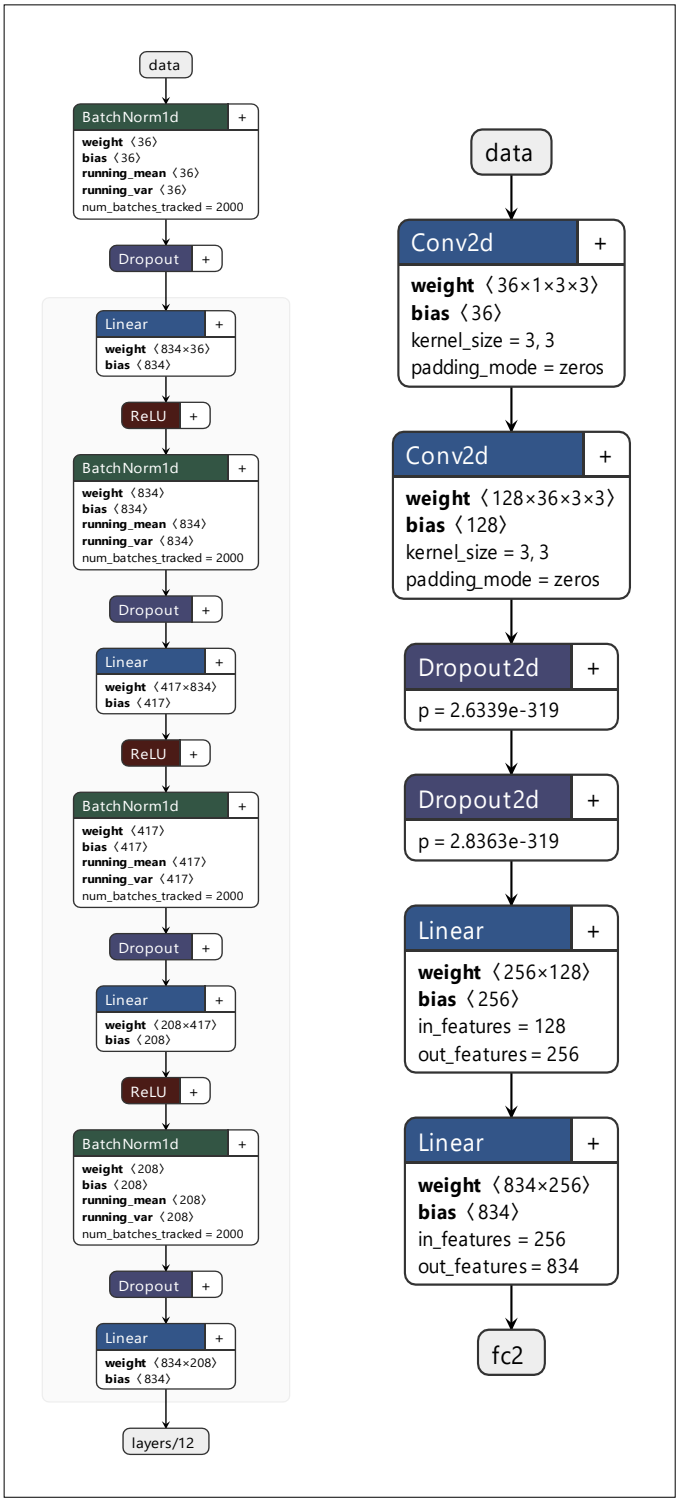

Fig. 10. Neural Network and CNN Models

The network was trained on a Nvidia Titan RTX with a batch size of 40,000 for 2,000 epochs. Training took an average of one minute and 53 seconds. The final results of training were precision 0.93, recall 0.95, F1 0.94 and accuracy 0.95.

Finally, we implemented a convolutional neural network consisting of two convolution networks with kernel size 3, a ReLU activation function, then a maxpool to coalesce the convolutions. The pooled convolutions go into a fully connected network of size 128x256, a dropout of 0.25, a fully connected network of size 256x834, a ReLu and a dropout of 0.5. Softmax used to produce the final classification.

The convolutional network was trained in the same fashion as the neural network. Training time averaged 4 minutes 50 seconds. Fig. 11 shows the function output and the results were precision 0.93, recall 0.94, F1 0.93 and accuracy of 0.94.

	precision	recall	f1-score	support
GI_fxstatat64	1.00	1.00	1.00	14
_IO_adjust_column	1.00	1.00	1.00	16
_IO_adjust_wcolumn	1.00	1.00	1.00	15
_IO_cleanup	1.00	1.00	1.00	10
_IO_default_doallocate	1.00	1.00	1.00	19
_IO_default_finish	1.00	1.00	1.00	15
_IO_default_pbackfail	1.00	1.00	1.00	7
_IO_default_seekpos	1.00	1.00	1.00	11
_IO_default_setbuf	1.00	1.00	1.00	12
_IO_default_uflow	1.00	1.00	1.00	17
_IO_default_xsgetn	0.95	1.00	0.97	19
_IO_default_xsputn	1.00	1.00	1.00	15
_IO_do_write	1.00	1.00	1.00	14
_IO_doallocbuf	1.00	1.00	1.00	12
_IO_enable_locks	1.00	1.00	1.00	20
_IO_file_attach	1.00	1.00	1.00	18
_IO_file_close_it	1.00	1.00	1.00	15
_IO_file_doallocate	1.00	1.00	1.00	13
_IO_file_fopen	0.91	1.00	0.95	10
_IO_file_open	1.00	1.00	1.00	11
_IO_file_read	1.00	1.00	1.00	11
_IO_file_seekoff_maybe_mmap	1.00	1.00	1.00	11

Fig. 11. Convolutional Classifier Output

C. *Analysis*

After reviewing these results, we determined several factors were limiting further improvement of the machine learning results. The first problem is one of labeling. All the classification approaches were fully supervised, the accuracy of the labels is reflected in the training results. When some of the misclassifications were examined, we found that some functions had different labels but identical feature sets. Further analysis found 18,857 such conflicts. In the case of the functions *register_printf_function* and *register_printf_specifier* the assembler is the same, as seen in Fig. 12. In other cases, the assembler may be different, but the instruction counts are the same. It is possible to automate the process of resolving some of these differences, but further research would be required.

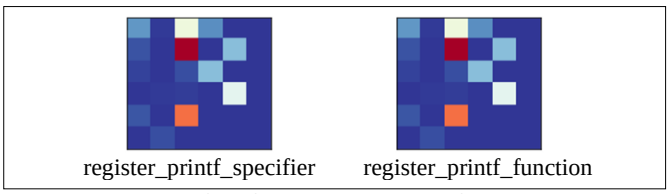

Fig. 12. Visualization of two functions with the same feature vector

This initial work uses recent advances in graph learning by developing a graph representation of each function's control flow graph, with each block's contents represented by the vex embedding of each function graph by using adjacency matrices for the edges concatenated with a learned embedding of the block features. Other approaches to this problem could consist of:

- A richer feature set made from a different LLIL.
- Create embeddings of assembler language to improve the representations of the disassembled code.
- Use an LLIL which has been developed to work well in transformer and recurrent network architectures.

V. Current and Future Research

The Firmware Indicator Translation (FIT) project is ongoing to finish the release of @DisCo and other tools. Ubiquitous libraries found on FIT partner provided firmware will be translated into Structured Threat Information eXpression (STIX) to enable detection and potential remediation if vulnerabilities exist. Results from these analyses in structured threats provide better contextual indicators for malware, sharable and actionable threat intelligence which enriches a test corpus another DOE project - Geo Threat Observables (GTO). Grid Modernization Laboratory Consortium (GMLC) projects focused on cyber security will take FIT's machine learning concepts further in the Firmware Command and Control and Deep Learning Malware projects, moving beyond static binary analysis. Internal INL research for Reverse Engineering at Scale (RE@Scale) will be advanced with new compiler and linked library capabilities. Currently FIT tools are being used for forensics and supply chain analysis projects internally at INL and will be available soon on INL's GitHub page.

Acknowledgment

We would like to acknowledge our leadership support at Idaho National Laboratory for the novel RE@Scale concept and the key INL researchers, Jared Verba and Gordon Rueff, for their contributions. We gained knowledge and insight from asset owners in Southern California Edison and Detroit Edison. FIT also worked with technology and industry partners New Context, Eaton, Hitachi, and Siemens. A considerable thank you is due to the FIT team, Rita Foster, Bryce McClurg, Zachary Priest, Bryan Beckman, and Justin Cox. Our final appreciation goes to DOE's leadership and technical direction to apply concepts for better cyber analytics to GMLC protections ensuring these concepts are spread broader in the research community focused on protecting our nation's electric grid.

References

[1] "Assessing Security Risk in Legacy Systems," 14 Decemeber 2006. [Online]. Available: https://www.us-cert.gov/bsi/articles/best-practices/legacy-systems/assessing-security-risk-in-legacy-systems. [Accessed 18 January 2020].

[2] "Protecting Our Customers from XcodeGhost," 2015 September 22. [Online]. Available: https://www.fireeye.com/blog/executive-perspective/2015/09/protecting_our_custo.html. [Accessed 18 January 2020].

[3] A. F. Agarap, "Deep Learning using Rectified Linear Units (ReLU)," *CoRR*, vol. abs/1803.08375, 2018.

[4] A. Greenberg, "The Untold Story of NotPetya, the Most Devastating Cyberattack in History," 7 December 2018. [Online]. Available: https://www.wired.com/story/notpetya-cyberattack-ukraine-russia-code-crashed-the-world/. [Accessed 18 January 2020].

[5] A. Liaw and M. Wiener, "Classification and regression by randomForest," *R news*, vol. 2, p. 18–22, 2002.

[6] A. Paszke, et al., "PyTorch: An Imperative Style, High-Performance Deep Learning Library," 3 12 2019.

[7] C. Cortes and V. Vapnik, "Suppor-Vector Networks," September 1995. [Online]. Available: https://doi.org/10.1007/BF00994018. [Accessed 24 January 2020].

[8] D. Arthur and S. Vassilvitskii, "K-means++: the advantages of careful seeding," in *In Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*, 2007.

[9] D. Coomans and D. L. Massart, "Alternative k-nearest neighbour rules in supervised pattern recognition," *Analytica Chimica Acta*, vol. 136, p. 15–27, 1982.

[10] D. Larson, "Global Survey Reveals Supply Chain as a Rising and Critical New Threat Vector," 1 April 2019. [Online]. Available: https://www.crowdstrike.com/blog/global-survey-reveals-supply-chain-as-a-rising-and-critical-new-threat-vector. [Accessed 18 January 2020].

[11] F. Pedregosa, et al., "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, p. 2825–2830, 2011.

[12] I. Goodfellow, Y. Bengio and A. Courville, " Softmax Units for Multinoulli Output Distributions," *In Deep Learning*, MIT Press, 2016.

[13] I. T. Jolliffe, "Principal Component Analysis and Factor Analysis," in *Principal Component Analysis*, Springer New York, 1986, p. 115–128.

[14] J. Ward, "Hierarchical Grouping to Optimize an Objective Function," *Journal of the American Statistical Association*, vol. 58, no. 301, pp. 236-244, 1963.

[15] L. McMinn and J. Butts, "A Firmware Verification Tool for Programmable Logic Controllers," in *ICCIP*, Heidelberg, 2012.

[16] L. van der Maaten and G. Hinton, "Visualizing High-Dimensional Data Using t-SNE," *Journal of Machine Learning Research*, pp. 9:2579-2605, 2008.

[17] M. Ester, K. Hans-Peter, J. Sander and X. Xiaowei, "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise," *Proceedings of the 2nd International Conference on Knowledge Discovery and Data Mining*, p. 226–231, 1996.

[18] N. Friedman, D. Geiger and M. Goldszmidt, "Bayesian Network Classifiers," *Mach. Learn.*, vol. 29, p. 131–163, 11 1997.

[19] N. Stephens, et al., "Driller: Augmenting Fuzzing Through Selective Symbolic Execution.," in *NDSS*, 2016.

[20] OrientDB, "OrientDB. Hybrid Document-Store and Graph NoSQL Database," 2017.

[21] R. E. Schapire, "Explaining AdaBoost," in *Empirical Inference*, Springer Berlin Heidelberg, 2013, p. 37–52.

[22] R. Quinlan J., *Induction of Edecision Etrees*, 1986.

[23] S. Raemaekers, A. Van Deursen and J. Visser, "An Analysis of Dependence on Third-Party Libraries in Open Source and Proprietary Systems," in *Sixth International Workshop of Software Quality and Maintainability*, Amsterdam, 2012.

[24] S. Yan, et al., "Sok:(state of) the art of war: Offensive techniques in binary analysis," *Security & Privacy. IEEE Computer Society*, p. 138–157, 2016.

[25] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel and G. Vigna, "Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware.," in *NDSS*, 2015.