YELLOWTIGER: Exploring the Future of Binary File Analysis

Vulnerability, Exploitation, and Mitigation Team,
Laboratory for Telecommunication Sciences

YELLOWTIGER (YETI) is a tool built to aid in understanding executable files. Focused primarily on finding useful relationships between binaries, the data that YETI generates has applications across a variety of domains, such as identifying and understanding the purpose of malicious files, enhancing vulnerability analysis, and predicting authorship or provenance of the file.

One key area that YETI gives emphasis to is expanding the types of features that can be extracted from binary files. Strings, sequences of bytes, and fields extracted from the headers present in binary files are widely used because they are quick to access and easy to compare; YETI adds to these possibilities by allowing extraction of features that require deeper understanding of the binary, such as the structure of the code. An understanding of the code can expose the core logic of the binary's design and purpose. YETI enables researchers to explore the idea that features derived from a binary's code add value when trying to solve problems in this space, which was not easily possible before it was developed. While the additional processing required to generate these features adds challenges in scalability, results indicate that this approach may both be feasible computationally and add accuracy and flexibility when building analytics and machine learning models for binary files.

[Photo credit: iStock.com/undefined undefined]

How does YETI extract code features?

To gain a deeper understanding of binary files, YETI has leveraged the open-source reverse-engineering tool Ghidra (developed and maintained by NSA's Computer and Analytic Sciences Research Group). Ghidra is able to both disassemble and decompile the machine code of a binary so that the functionality can be analyzed and understood. In addition, Ghidra also provides a rich scripting application programming interface (API) and the ability to run scripts "headlessly," meaning that Ghidra can run a script on a file or set of files and extract data of interest for later analysis. By integrating Ghidra, YETI is able to gain access to a rich array of new features that are now being explored more deeply.

Some analytics built in YETI are focused on features already identified by Ghidra, for example immediate values used in the disassembly, cross-references of functions, or x86/64 instructions. Other analytics can use these features as building blocks for more complex analysis, opening doors to possibilities that are limited only by the imagination of the researcher creating them.

Details of YETI's architecture

YETI is built on top of a variety of technologies. The user portal for YETI is a web application developed in Plotly Dash, which is a useful web framework for researchers because it abstracts many of the design and interface details, supports the interactive charts that many of YETI's analytics leverage, and allows development in Python. Elasticsearch is used to store extracted data. Each component of YETI, for example the Elasticsearch instances, the web application, and the analytic workers, is run inside a Docker container. Celery is used for scheduling tasks such as running an analytic and delivering the resulting data to the component that consumes it. Kubernetes is used to orchestrate the various components needed to run YETI; it provides many benefits such as load distribution, recovery from some kinds of failures, and the ability to scale up or down resources to complete analytic tasks.

To enable development of analytics, YETI also has JupyterLab integrated into its architecture as a first-class citizen, meaning it can access Elasticsearch to retrieve and store data, schedule Celery tasks to run workers and collect data, and other features needed to prototype and test a new analytic. This allows researchers to experiment with different possibilities and build a capability they feel provides value before they build it into YETI as part of the user interface, which requires more of a time investment and is more cumbersome to test. See figure 1 for a simplified diagram of the YETI architecture.

Challenges of scaling YETI's analytics

Scalability is still a largely unexplored question in this space. Tools exist with similar architectures (e.g., built on top of Kubernetes, Docker, and Elasticsearch, doing analysis of binary files) that have reportedly been successfully scaled to handle millions of files. YETI tends to focus on analytics that process and analyze disassembly or decompilation in various ways, which is quite computationally intensive. So far, YETI has been used on much smaller populations of files (in the thousands). Similarly, in the machine learning that YETI applies, there are challenges with applying the algorithms over the quantities of data generated by YETI's analytics. There are a number of strategies which may help in stretching capabilities further, such as using free compute cycles to pre-compute and store results, optimizing data storage formats for the type of analytic the data is being used for, and filtering the data into smaller subsets based on some pre-analysis to reduce the number of files an analytic is run over.

Use cases for YETI

YETI has been used to develop a wide variety of analytics to date, ranging from workers that extract features of code and produce charts based on them, to machine learning capabilities that consume features generated by analytics and attempt to find relationships between files or groups of files. Typically when a new feature is being explored, the data of interest is extracted from a set of files and exploratory data analysis is conducted to determine meaningful patterns and inform decisions on how to display it, or how to apply machine learning algorithms to it.

In this section, several analytics that have been explored within YETI are discussed, along with results. YETI was used to curate the data in these experiments and to extract features explored in these analyses. Some of the most promising results from the work are described below. Better understanding of the value of function prologue data through

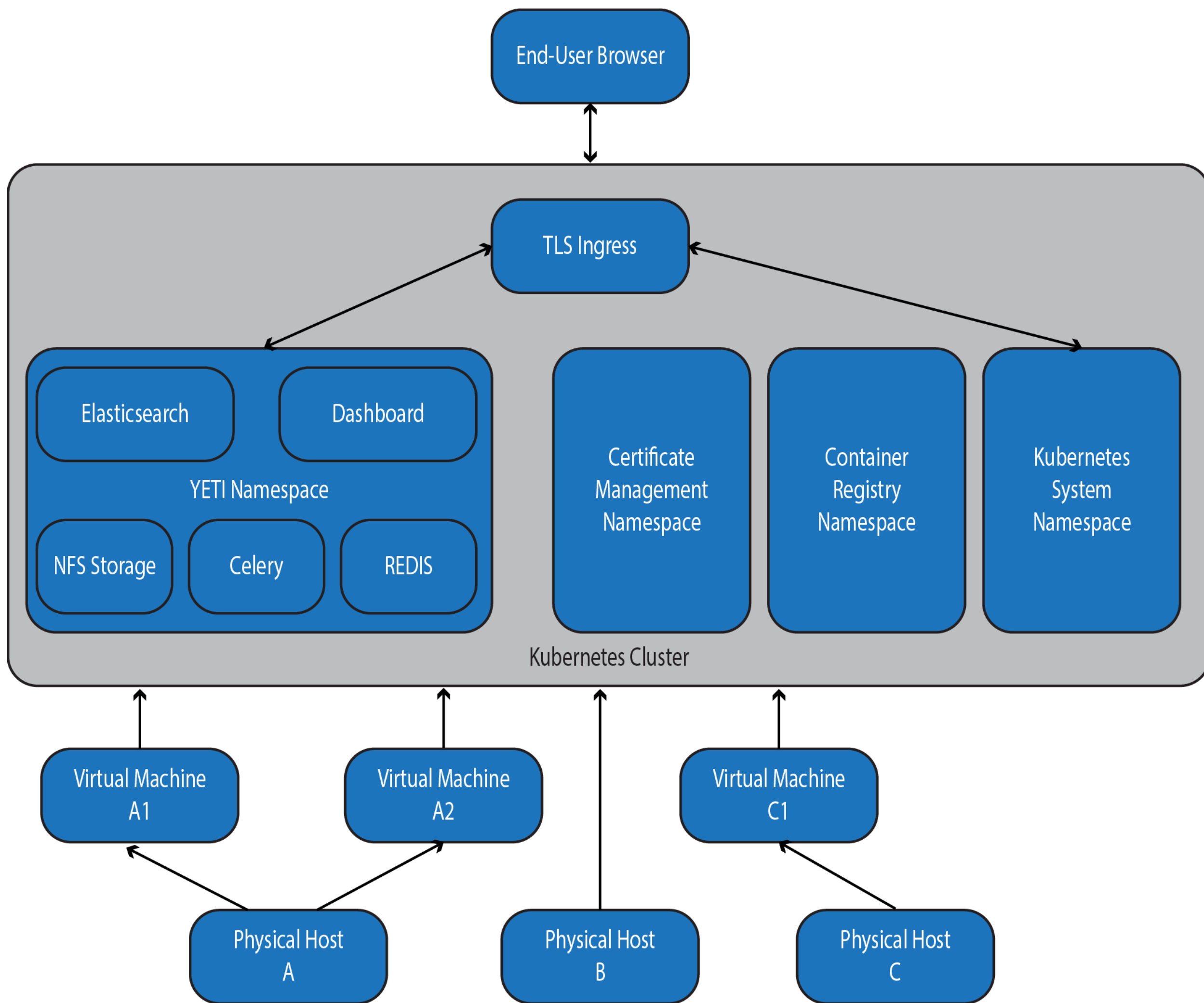


FIGURE 1. This diagram shows a simplified version of the YETI architecture.

exploration in YETI is an area of active research within the team.

Function prologues

Binaries contain compiled code, and code tends to be organized into functions, or pieces of code that perform specific tasks that can be re-used, or "called," as needed. When a program is compiled, the compiler tends to have certain patterns of assembly instructions that are used at the beginning and end of each function; these patterns handle setting up the stack and registers with the expected data and are generally specific to both the compiler and the architecture that the program is being compiled for. This makes them an interesting feature to use in assessing if two files are similar to one another; similar function prologues may mean that the code was compiled with the same compiler and options, and for the same architecture. To test this theory, several summer interns working with the YETI team explored using function prologues to help categorize files using machine learning.

A Ghidra script was developed to extract the first few instructions from each function in a binary. Because the length of a prologue is unpredictable, part of the analysis involved experimenting with various lengths and trying to find the ideal sets of instructions to consider as prologues. The amount of input was then reduced by looking across functions

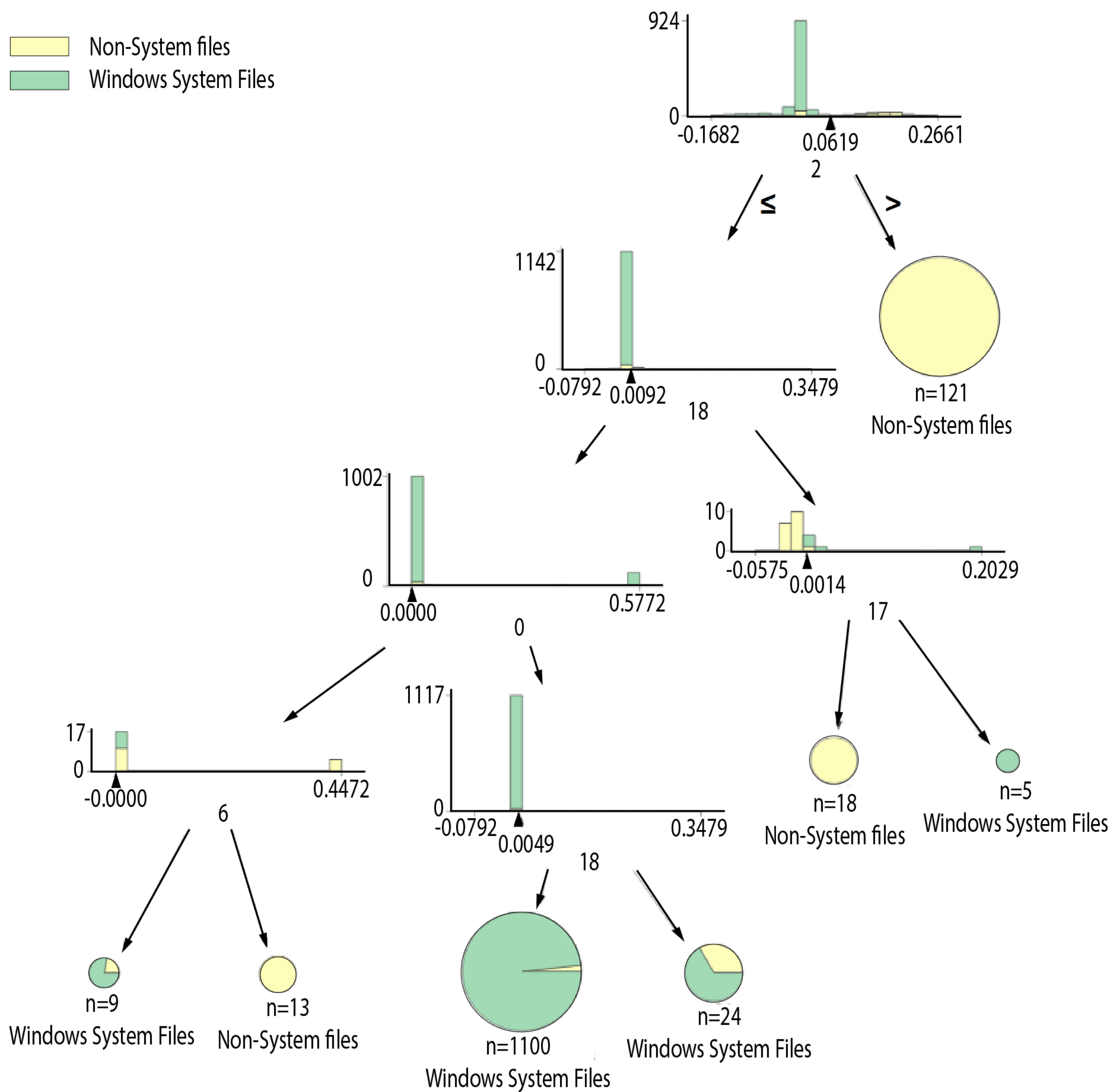


FIGURE 2. This decision tree shows classification decisions for a collection of files.

and files to determine the most common sets of prologue instructions. For this work, a body of 2,500 binaries produced 1.5 million prologue features; singular value decomposition was used to reduce this to 20 features.

Using this data, a random forest model was trained and used for two-class classification between system and non-system binary files. It was able to detect 99.6 percent of system files correctly, and 76 percent of non-system files.

To better understand the classification process, a second experiment was performed on the same data using decision trees instead of a random forest. It had similar performance, with 99 percent of system files correctly detected, and 72 percent of non-system files. This was done in order to gain insight into how classification decisions were being made by the model. Figure 2 shows an example of one of the visualizations generated.

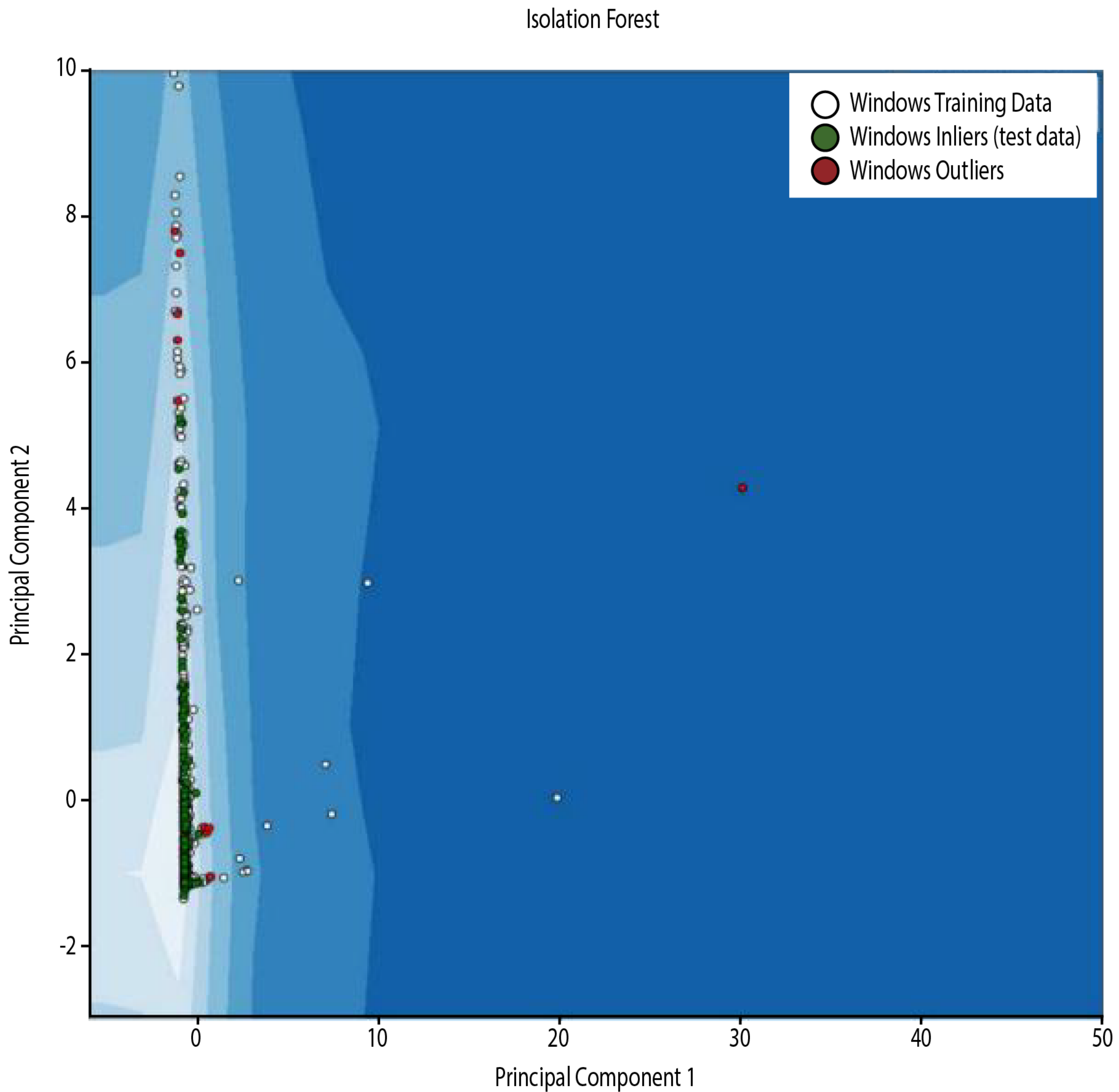


FIGURE 3. These isolation forest results show the Windows system test set for x86 Windows prologues.

In this decision tree, at each node in the graph, in cases where there is a mix of system and non-system files, the distribution of the data is shown. The black carrot under the distribution indicates where the algorithm chose to divide the data. The number from 0-19 below the distribution is the dimension of the data that was used to make this determination. This illustration of the process is useful in gaining an understanding of how the classification process progresses.

Another separate experiment to help measure the analytic value of function prologue data was training and testing an isolation forest, which is an unsupervised learning algorithm that detects anomalies by isolating outliers in the data. The model was trained on Windows files, and figure 3 and 4 compare the results of testing it against additional Windows system files, and non-system files.

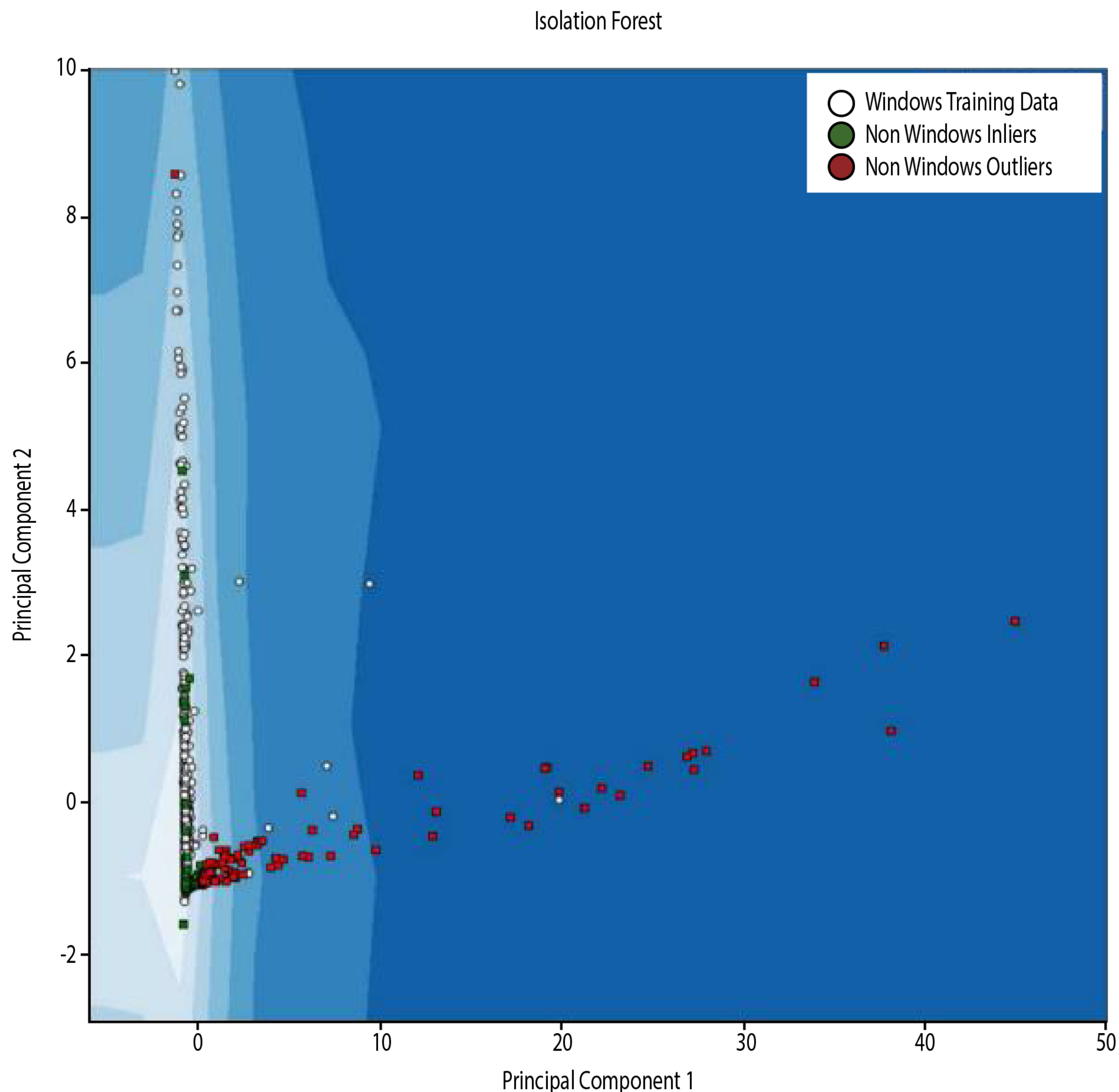


FIGURE 4. These isolation forest results show x86 non-system test set prologues.

To interpret figures 3 and 4, white points are the training set files; green points are the files detected as inliers, and red points are files detected as outliers by the algorithm. Shades of blue indicate the varying outlier scores assigned, with darker blue being a higher outlier score.

Comparing figures 3 and 4, more non-system files were detected as outliers (52 percent) versus Windows system files detected as outliers (5 percent). Because the percent detected as outliers varies depending on the contamination parameter used with the model, the point of interest is the difference between the percentages for each class of files. We can infer from this large difference that there are many features of the non-system files that set them apart from the Windows system files.

As a final note on these results, while it may not seem substantial to detect only 52 percent of non-system files as outliers, consider that information was lost during principal component analysis (PCA), and also that this is focused on a single feature; in future work it would be worthwhile to explore performance

with additional features included.

Index of coincidence

The YETI team has been performing exploratory data analysis by calculating an index of coincidence (IC) score over the bytes of code in an executable file. Coincidence counting is a technique which places two texts side-by-side and counts the number of times that identical letters appear in the same position in both texts. This count is taken either as the ratio of the total, or normalized by dividing by the expected count for a random source model [1]. The formula for calculating the IC is shown in equation 1. Because in code, like in natural language text, we expect certain values to appear more often than others, the IC is higher than it would be for random values. Furthermore, code generated by different compilers may be able to be characterized by the IC calculation, which would be useful in measuring how similar two binaries are to one another.

$$IC = c\sum_{i=1}^{c}\frac{n_i}{N}\frac{n_i - 1}{N - 1} \qquad (1)$$

To calculate the IC, we wrote a Ghidra script that iterated over the functions in a program and performed the calculations over the bytes in the body of the function. Files that did not contain any functions were removed from the dataset. Each file had one non-negative number representing the IC.

In the experiments measuring IC on code and testing it as a feature, we encountered several challenges. One was the size of the available datasets. To accurately characterize code from different compilers, we theorize that a much larger quantity of data may be needed. In some cases, for specific sets of files, it may be difficult or impossible to find a large enough quantity of available data.

We also observed that better results were achieved with IC when instead of calculating over the entire body of the function, we identified the bytes that were most meaningful to the purpose or label of the file and filtered out bytes that were less impactful for the purpose of the file. We explored several methods to identify these bytes, including:

- Trying to remove the prologue and epilogue bytes from the calculations, and
- Using entropy and cross-entropy calculations on different sections of the function to try to identify changes that might indicate where the substance of the functions was.

Finally, we found that experiments training models on IC calculations had poor classification results when used individually, but when combined with function prologue data, the IC was consistently within the top 10 percent of most important features of the classifiers. This demonstrates that while by itself IC is not distinguishing, at least for the amount of data available, at later points of the classification process, IC may become better able to divide the data into effective groupings.

Entropy and cross-entropy

In a similar vein to the IC, another analytic we developed involved calculating entropy over the distribution of the raw bytes of a file. Entropy calculations contain information about the distribution and counts of bytes that make up a file; see equation 2 for the formula used to calculate it in YETI's experiments. The same strategy used with the IC calculations proved most effective, namely extracting the most relevant bytes from the functions, and eliminating files without functions from the dataset.

$$H(X) = \sum_{i=1}^{c} -p(x_i)log_2 p(x_i) \qquad (2)$$

Entropy has been successfully used in previous research on malware detection, which was one reason we chose it for further exploration [2, 3].

Cross-entropy is a similar metric that serves to allow comparison between two probability distributions, with a higher value for the cross-entropy calculation indicating that the two distributions are further apart from one another [4]. To explore this feature, we initially calculated each file's cross-entropy with every other file in the dataset (see equation 3). Using this data, for some categories where files had sufficiently similar scores, we identified a representative file so that cross-entropy calculations did not have to be stored for *every* file.

$$H(p, q) = \sum_{i=1} -p(x_i)log_2 q(x_i) \qquad (3)$$

Results showed that cross-entropy was able to differentiate between system and non-system files with around 90 percent accuracy in each case.

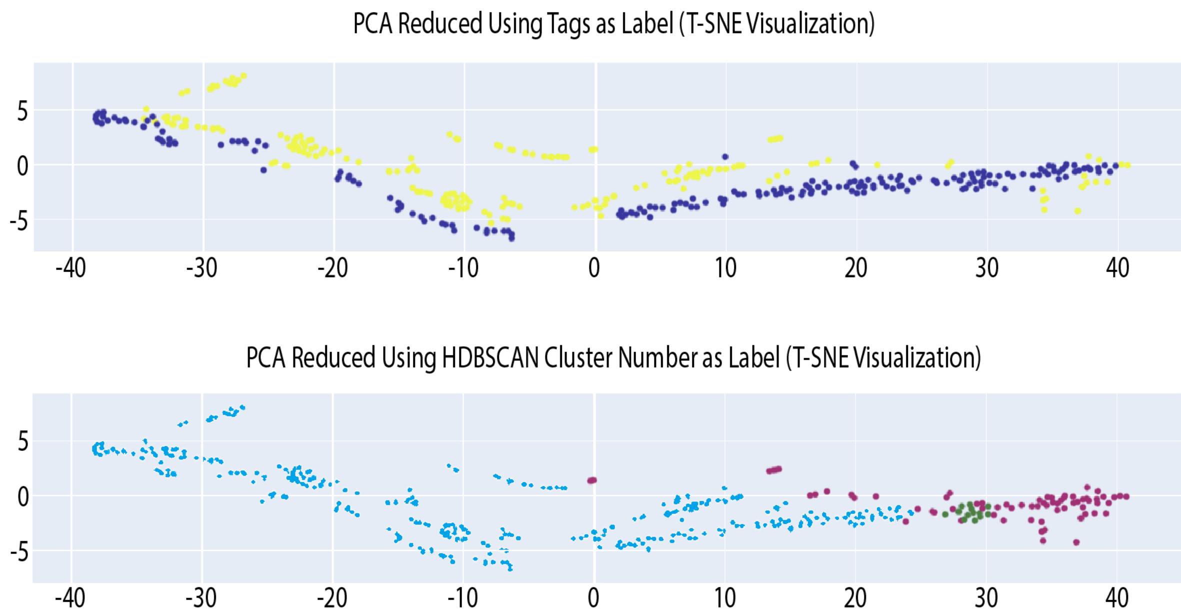


FIGURE 5. This attempt at clustering did not succeed at breaking data into desired clusters.

Clustering binaries

YETI has also been used for research into clustering binary files, building off of existing research on clustering malware [5]. The aim is to see which features contribute positively to clustering binary data. Once "good" clusters have been established, the next step is to look at the distribution of the features based on the clusters and determine what is considered normal for that cluster (i.e., that subset of binary files). Many of the features tested with the clustering algorithms, such as function cross-reference counts, are extracted using YETI to run Ghidra scripts on the binaries.

In figure 5, PCA is used to reduce the dimensionality of the data. The data includes Windows system files, and a relatively new dataset curated and provided by the NSA Research Directorate's Laboratory for Physical Sciences called Assemblage; features extracted from the data and used for clustering include standardized/one-hot encoded header data, function cross reference counts, and instruction occurrences. We then used hierarchical density-based spatial clustering of applications with noise (HDBSCAN) to cluster and assign labels to each file (including a -1 noisy label, which is the purple color in the lower image in figure 5) [6]. The top image in figure 5 shows the two-dimensional (2-D) projection of the data with the marker color being the known label (Windows or Assemblage). The bottom image shows the 2-D projection of the data with the marker color being the HDBSCAN cluster label. Overall, we can see that most of the points are labeled light blue, with a few points labeled green and a few others labeled the noisy purple.

In figure 6, PCA is again used to reduce the dimensionality of the data, but this time instruction occurrences are excluded as a feature, and HDBSCAN is used to cluster and assign labels to each file. Once again, the top image shows the 2-D projection of the data with the marker color being the known label (Windows or Assemblage). The bottom shows the 2-D projection of the data with the marker color being the HDBSCAN cluster label. In this case, there is better delineation between the two known file families. Most of the blue points in the top chart are clustered together in the bottom. Some of the yellow points are properly clustered together (the top darker pink cluster) while the rest are broken down into further clusters or marked as noisy.

There are many applications for successful clustering of binary files. Within YETI, one goal is to be able to automatically tag files with relevant labels based on clusters they fall into rather than relying on users

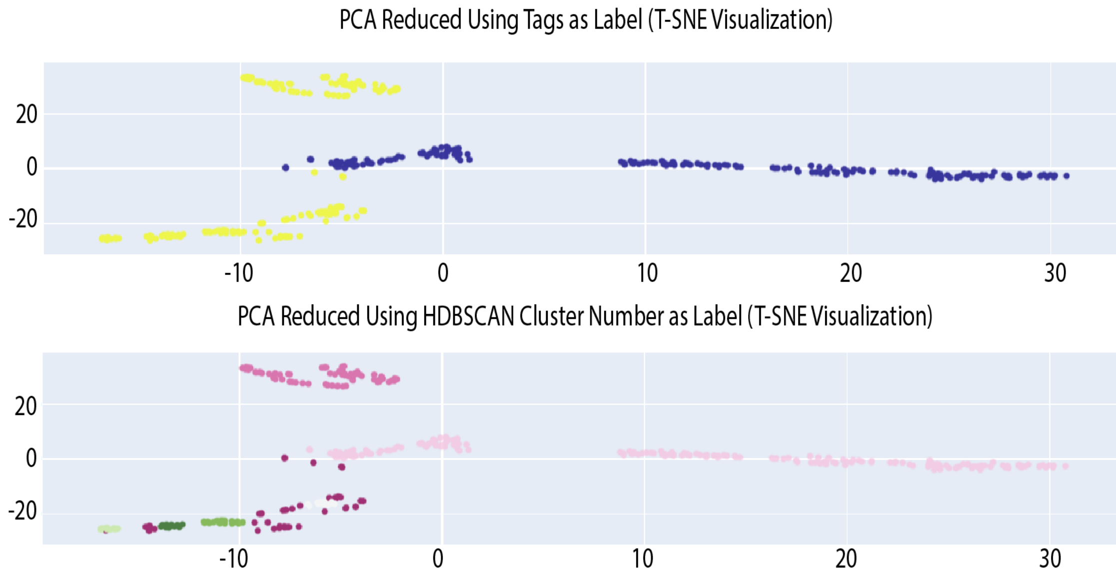
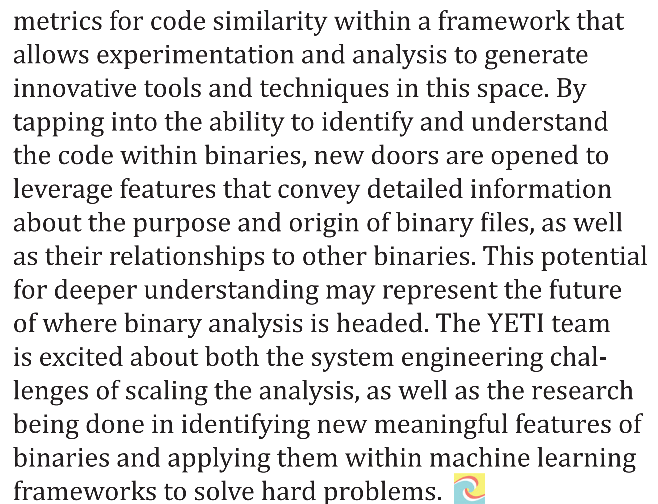


FIGURE 6. We achieved this more successful attempt at clustering by removing noisy features.

to correctly tag data upon upload. A more common use case is of course being able to identify suspicious files if they categorize into clusters with known malware, or to gain insights into attribution of a file back to its author [7].

Conclusion

YETI is a flexible and robust tool for research on scalable binary analysis to better understand binary similarity and quantify relationships between binaries in meaningful ways. There is a great deal of ongoing and planned work exploring the features and analytics described above, as well as developing new analytics for use in YETI. YETI houses a growing set of useful



metrics for code similarity within a framework that allows experimentation and analysis to generate innovative tools and techniques in this space. By tapping into the ability to identify and understand the code within binaries, new doors are opened to leverage features that convey detailed information about the purpose and origin of binary files, as well as their relationships to other binaries. This potential for deeper understanding may represent the future of where binary analysis is headed. The YETI team is excited about both the system engineering challenges of scaling the analysis, as well as the research being done in identifying new meaningful features of binaries and applying them within machine learning frameworks to solve hard problems.

References

[1] Friedman W. *The Index of Coincidence and its Applications in Cryptography.* Department of Ciphers Publ 22. Geneva (IL): Riverbank Laboratories; 1922.

[2] Gilbert D, Mateu C, Planes J, Vicens R. "Classification of malware by using structural entropy on convolutional neural networks." *Proceedings of the AAAI Conference on Artificial Intelligence.* 2018;32(1). Available at: https://doi.org/10.1609/aaai.v32i1.11409.

[3] Lyda R, Hamrock J. "Using entropy analysis to find encrypted and packed malware." I*EEE Security and Privacy.* 2007;5(2):40–45. doi: 10.1109/MSP.2007.48.

[4] Cybenko G, O'Leary DP, Rissanen J, editors. *The Mathematics of Information Coding, Extraction and Distribution.* New York (NY): Springer; 1999. ISBN: 978-0-387-98665-4.

[5] Kim S. "PE header analysis for malware detection." (2018). Master's Projects, 624, San Jose State University. Available at: https://doi.org/10.31979/etd.q3dd-gp9u and https://scholarworks.sjsu.edu/etd_projects/624.

[6] McInnes L, Healy J. "Accelerated hierarchical density based clustering." In: *2017 IEEE International Conference on Data Mining Workshops (ICDMW);* 2017; IEEE: pp 33–42. Available at: https://doi.org/10.1109/ICDMW.2017.12.

[7] Rosenblum N, Zhu X, Miller BP. (2011). "Who wrote this code? Identifying the authors of program binaries." In: C*omputer Security-ESORICS 2011: 16th European Symposium on Research in Computer Security;* 2011 Sep 12–14; Leuven, Belgium: pp. 172–189). Springer Berlin Heidelberg. Available at: https://doi.org/10.1007/978-3-642-23822-2_10.