Detection of Prevalent Malware Families with Deep Learning

Jack W. Stokes*, Christian Seifert†, Jerry Li‡ and Nizar Hejazi§
*Microsoft Research, Redmond, Washington, 98052, USA, Email: jstokes@microsoft.com
†Microsoft Corporation, Redmond, Washington, 98052, USA, Email: chriseif@microsoft.com
‡Snap Inc., Seattle, WA 98121, USA, Email: jerry.li@snap.com
§Uber Corporation, Seattle, WA 98101, USA, Email: nhejazi@uber.com

***Abstract*—Attackers evolve their malware over time in order to evade detection, and the rate of change varies from family to family depending on the amount of resources these groups devote to their "product". This rapid change forces anti-malware companies to also direct much human and automated effort towards combatting these threats. These companies track thousands of distinct malware families and their variants, but the most prevalent families are often particularly problematic. While some companies employ many analysts to investigate and create new signatures for these highly prevalent families, we take a different approach and propose a new deep learning system to learn a semantic feature embedding which better discriminates the files within each of these families. Identifying files which are close in a metric space is the key aspect of malware clustering systems. The *DeepSim* system employs a Siamese Neural Network (SNN), which has previously shown promising results in other domains, to learn this embedding for the cosine distance in the feature space. The error rate for K-Nearest Neighbor classification using *DeepSim's* SNN with two hidden layers is 0.011% compared to 0.42% for a Jaccard Index-based baseline which has been used by several previously proposed systems to identify similar malware files.**

***Index Terms*—Malware Detection, Siamese Neural Network**

I. Introduction

Detecting unknown, new variants of malware remains a significant problem facing anti-malware companies today. While these companies routinely track thousands of individual malware families and their variants, the most prevalent families are particular challenging. In this paper, we present a system called *DeepSim* to automatically identify similar, malware files and demonstrate that it can significantly improve the detection results compared to a baseline system utilizing the Jaccard Index which has been used in similar, previously proposed systems [1]–[3]. While being applicable to any polymorphic malware family, we show that *DeepSim* is particularly effective in combatting highly prevalent malware families.

A key component of some malware detection systems is determining similar files in a high-dimensional input space. For example, instance-based malware classifiers such as the K-Nearest Neighbor (KNN) classifier [4], [5] rely on the similarity score or distance between two files. The K-Nearest Neighbor classifier is the optimal classifier given an infinite amount of training data [6]. Malware clustering [1]–[3], [7] which identifies groups of malware files also relies on computing a similarity score between files. A number of malware detection systems use the Jaccard Index as the similarity score [1]–[3]. The problem with the Jaccard Index is that all features extracted from a file have equal weight. However, certain key features such as a particular URL or registry key pattern may be key to detecting individual files from highly prevalent families. What is needed to better detect similar malware is a method for learning these key features and giving them higher weight.

DeepSim employs a novel method of learning an embedding vector, based on deep learning, which detects similar malware files in the feature space using the cosine distance. Several researchers have explored using deep learning for malware classification [8]–[10]. Neural network architectures trained with millions of files are producing some of the best results for malware classification published in the literature [8]. Inspired by these results, we seek to find a neural network-based architecture to detect similar malware files. Bromley et al. [11] proposed a deep learning algorithm called the Siamese Neural Network (SNN) for learning similar pairs in the context of signature verification. We use a Siamese Neural Network to learn the feature embeddings in *DeepSim*. The SNN addresses the high-level goal of training a model which learns to give different weights to features based on their importance.

We have implemented *DeepSim* and tested it on a large collection of highly prevalent malware families. Implementing a K-Nearest Neighbor classifier using the SNN with two hidden layers yields an error rate of 0.011% compared to 0.42% for a KNN classifier which uses the Jaccard index to identify similar files. A summary of the main contributions of our work includes:

- We propose *DeepSim*, a deep learning architecture for identifying similar malware files. DeepSim learns weights corresponding to different features such that important features found in similar files, from the same family, contribute positively to the cosine similarity score and features which distinguish malware files from benign files contribute negatively to the cosine similarity score.
- We implement and evaluate *DeepSim* on a collection of highly prevalent families and demonstrate that it significantly outperforms a similar baseline system using the Jaccard Index which has also been used in several previously proposed malware detection and clustering systems.

II. DeepSim System Overview

We designed the *DeepSim* to detect highly prevalent malware families, and the overview for the system is shown in Figure 1. The left-hand side describes the components required for training *DeepSim's* underlying model while the right-hand side depicts the steps followed when evaluating a set of unknown files in order to automatically predict if they belong to one of the highly prevalent malware families. We next provide details of some of the individual components in our system.

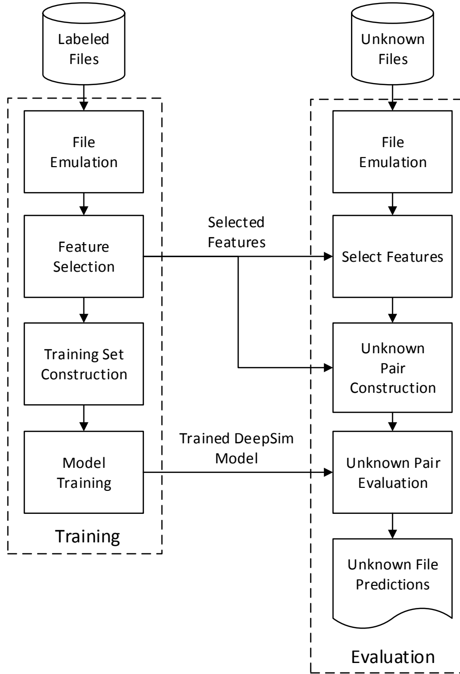

Fig. 1: *DeepSim* high-level system overview. The left-hand side outlines the steps for training, and the right-hand side illustrates the evaluation process.

File Emulation. *DeepSim's* first component for both model training and unknown file evaluation is lightweight file emulation which is performed using a modified version of a production anti-malware engine. This anti-malware engine extracts raw, high-level data during emulation and generates the logs which are consumed by the downstream processing. *DeepSim* utilizes dynamic analysis to extract the raw data from the labeled files for training the model and from the unknown files for predicting the family.

High-Level Feature Description. *DeepSim* employs two types of data extracted from the files including unpacked file strings and API calls with their associated parameters. It is important to note that the dataset that was provided to the researchers only included these two sets of data for analysis; it was not possible to include additional features for these files. Malware is often packed or encrypted. As it is emulated by the anti-malware engine, the malware unpacks or decrypts itself and often writes null-terminated objects to the emulator's memory. Typically, these null-terminated objects are the strings which have been recovered during unpacking or decryption, and these strings sometimes provide a good indication of whether or not the file is malicious.

In addition to the unpacked file strings, a second set of high-level features are constructed from a sequence of API calls and their parameter values. The API stream is composed of function calls from different sources including the user mode operating system and the kernel mode operating system. For example, there are a number of Windows APIs which can be used to read a registry key's value including the user mode functions RegQueryValue() and RegQueryValueEx() and the RtlQueryRegistryValues() function from kernel mode. Functions which perform the same logical operation are mapped to a single API event. In this example, calls to RegQueryValue(), RegQueryValueEx() and RtlQueryRegistryValues() are all mapped to the same API event ID (EventID). In addition, important API parameter values, such as the key name or the key value in our example, are also captured by the emulator. Using this data, the second feature set is constructed from a sequence of API call events and their parameter values. To handle the case where several API calls are mapped to the same event but have different parameters, only the two most important parameters shared by the different API calls are considered.

Low-Level Feature Encoding. Each malware sample may generate thousands of raw unpacked file strings or API call events and their parameters. Since we have designed *DeepSim* to detect *polymorphic* malware in addition to non-polymorphic malware, we cannot encode the potential features directly as sparse binary features. In other words, if each variant in a family drops a second, temporary file with a partially random name or contacts a command and control (C&C) server with a partially random URL, we do not want to represent the file name or the URL explicitly. To handle this case, we instead encode the raw unpacked file strings and the API calls and their parameters as a collection of N-Grams of characters. In particular, we use trigrams (i.e. N-Grams where N = 3) of characters for all values.

One limitation of a Jaccard Index-based similarity system is that it cannot distinguish between multiple types of features in the same set (e.g. EventID, Parameter Value1, Parameter Value 2). Also, short values such at the EventID (e.g. 98) have less influence on the Jaccard Index than longer features including the parameter values (e.g. registry key name). To improve the performance of the Jaccard Index baseline system, we overcome these limitations by expanding the EventID to the full API name and encoding the entire API name as a string using character level trigrams. Thus, representing the API name using their trigrams allows the API names to contribute more significantly to the file pair's Jaccard Index. We use this trigram representation of the API name for all models to fairly compare the results of the SNN model with the Jaccard Index-based model.

The SNN model, on the other hand, does not suffer from these limitations. Therefore in practice for *DeepSim*, we would encode the EventID or the API name as a single categorical feature because the two deep neural networks can learn to assign larger weights to the most important API calls for pairs of similar files. Thus, we could encode the entire call separately as (EventID, Parameter 1 N-Grams, Parameter 2 N-Grams), and we believe this would improve the SNN model's performance since it learns a specific representation for each combination of EventID and the N-Grams of the parameter values.

Feature Selection. There are hundreds of thousands of potential N-Gram features in the raw data generated during dynamic analysis, and it is computationally prohibitive to train the SNN model using all of them. Thus, we next perform per class, feature selection which yields the most discriminative features for our system using the mutual information criteria [12]. In order to process a production-level input data stream, we implemented all of the *DeepSim's* functions which are required to preprocess the data in Microsoft's Cosmos MapReduce system.

Training and Test Set Construction. Before training *DeepSim's* model, we must first construct a training set consisting of the selected N-Gram features from pairs of malware files which are known to be similar as well as those from benign files which are dissimilar. We determine the similar malware file pairs for this training set based on several criteria. First, in order to correctly train the *DeepSim* model, we must carefully choose the similar file pairs. Randomly selecting two files whose families match does not work well in practice [13]. The problem is that there are many different variants of some of these families. To solve this problem, we utilize the malware file's detection signature. An anti-malware engine utilizes specific signatures to determine if an unknown file is malicious or benign. Each signature is often very specific and has a unique identifier (SignatureID). Therefore the first step in determining similar pairs for training is to group pairs of malware files detected with identical SignatureIDs. While most malicious files that are detected with the same SignatureID belong to the same malware family, this is not always the case. Thus, we also require that candidate file pairs must be labeled as belonging to the same malware family.

For our data, we are only able to construct pairs of malware files based on the SignatureID and the malware family. Benign files all belong to one class and are not assigned a SignatureID since they are not malicious. As a result, we cannot construct similar pairs for benign files. To overcome this limitation in our data, we also construct a dissimilar pair constructed by randomly selecting a unique malware file and benign file to form the pair.

The format of this training set is shown in Table I. The *Training Set ID* is constructed from the concatenation of the SHA1 file hashes for Malware 1 (M_1), and either Malware 2 (M_2) or a Benign File (B) (i.e. $SHA1_{M1}_SHA1_{M2,B}$). This ID allows us to identify which files were used to construct the training instance. The next field in the training set provides the label where 1 indicates that the two files are similar (M1,M2) and -1 indicates that they are dissimilar (M1,B). The third field provides the selected N-Gram features from the primary malware file M_1. The N-Grams from the matching malware file M_2 or the randomly selected benign file (B) are provided in the final field.

Field
Training Set ID
Label (similar | dissimilar)
Malware 1 N-Gram Features
Similar Malware 2 or Benign File N-Gram Features

TABLE I: Training and test set instance format.

For the hold out test set used for evaluating all models later in Section III, we need to ensure that the file pairs in the training and test sets are unique. To do so, we first randomly select a pair of malware files for the training and test sets followed by a malware file and benign file pair. We next select a second similar malware file pair. If either of the files in the second pair match one of the files in the first pair, we add this malware pair to the training set. If it is not in the training set, we add it to the test set. Similarly, we do this for the a second dissimilar malware and benign pair. We continue to do this procedure of randomly selecting malware pairs and alternatively adding them to training or test sets until each is complete.

Model Training and Evaluation. After the training set has been constructed, the Siamese Neural Network model is trained. Figure 2 depicts the SNN model we propose for *DeepSim* which is based on the architecture described in Huang et al. [14]. The weights are adapted during training based on the cosine distance between the embedding of the known malware files, M_1 on the left-hand side and the malicious or benign files on the right-hand. The combined set of similar and dissimilar files is denoted as $F \in \{M_2, B\}$. We use backpropagation with stochastic gradient descent (SGD) and the Adam optimizer to train the model parameters.

When testing files, known malware files are input to the left-hand side, and known malware or benign files are then evaluated using the right-hand side. Thus during testing, the model's output represents the cosine distance between the embedding of the known malware files for the left-hand side and the embedding of the malicious or benign file on the right-hand side.

Unknown Pair Construction and File Evaluation. The format for the evaluation set is provided in Table II. Similar to the *Training Set ID*, the *Evaluation Set ID* includes the SHA1 file hash of the known malware file and an unknown file (i.e. $SHA1_{M1}_SHA1_U$) which allows us to determine which malware file is similar to the unknown file. The other two fields include the N-Grams from the known malware file and the unknown file.

To evaluate unknown files, we first include the selected N-Gram features from all of the known variants of our highly prevalent families in the training set. These features correspond

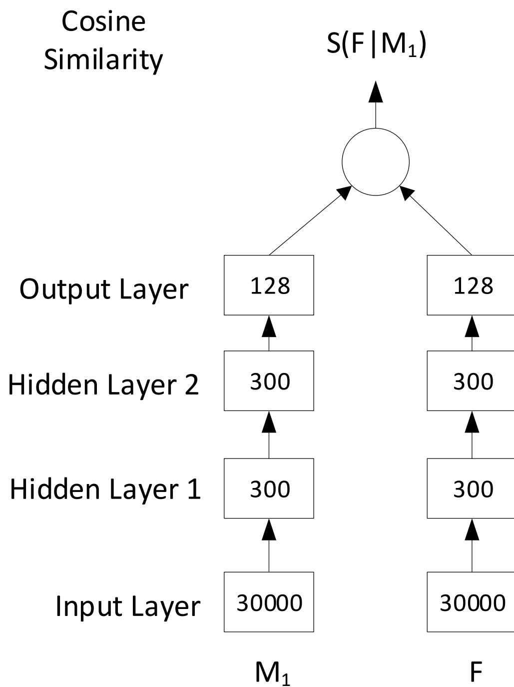

Fig. 2: Proposed *DeepSim* model. Two deep neural networks are trained where the cosine similarity score is used to learn the parameters for both DNNs. During traing, the left and right-hand branches are trained with known similar and dissimilar pairs. During evaluation, unknown file are input to the right-hand DNN and compared to the known malware in the left-hand DNN. The output is then computed as the cosine similarity between the output of the two DNNs.

to the left-hand side of the *DeepSim* model in Figure 2. We then include the selected N-Gram features from all of the unknown files that arrive for processing within a particular time period (e.g. day).

Depending on the number of known variants of the highly prevalent families and the incoming rate of unknown files, it may be necessary to further prefilter the number of file pairs to be considered. One method to do this is to employ the MinHash algorithm [15] to reduce the number of pairs of files used during training or the number of file pairs included during evaluation. A locality sensitive hash algorithm is used for a similar task in [1]. The MinHash algorithm is approximately O(n) and identifies only a small number of samples which need to be compared to each unknown file being evaluated.

Field
Evaluation Set ID
Known Malware N-Gram Features
Unknown File N-Gram Features

TABLE II: Evaluation set instance format.

Once the set of unknown file pairs has been constructed, they can be evaluated with the trained with the Siamese Neural Network. If the similarity score exceeds a prescribed threshold, the file is automatically determined to belong to the same family as the known malware file in the evaluation pair.

Another method to detect if an unknown file is malicious is to replace the cosine() distance with an optional K-Nearest Neighbor (KNN) classifier and assign the unknown file to the voted majority malware family or benign class of the K known files with the highest similarity scores. We evaluate the performance of including an additional KNN classifier later in Section III and find that assigning the label of the single closest file ($K = 1$) performs very well. Therefore we only need to find the single most similar file for our system.

Implementation. In order to process a production-level input data stream, we implemented all of the *DeepSim* functional blocks which are required to preprocess the data for training and testing in Microsoft's Cosmos MapReduce system. These functional blocks include the feature selection and training set construction for training as well as the evaluation functions which select the features to create the unknown pair dataset. Once the datasets are constructed, the *DeepSim* model is trained and results for the evaluation or test set are evaluated on a single computer. In practice, evaluating the prediction scores from the trained *DeepSim* model for the set of unknown files and the K-Nearest Neighbor classifier would also be performed in the MapReduce platform.

III. Evaluation

In this section, we conduct several experiments to evaluate the performance of the *DeepSim* system. We first describe the setup and hyperparameters used in the experiments. Next, we compare the performance of *DeepSim* with that of a baseline Jaccard Index-based system for the task of file similarity — the Jaccard Index has been used in several previously proposed malware detection systems [1]–[3]. Finally, we investigate the performance of a K-Nearest Neighbor classifier based on features from the SNN and the Jaccard Index computed from the set of highly prevalent malware files.

DataSet. Analysts from Microsoft corporation provided researchers with the raw data logs extracted during dynamic analysis of 190,546 Windows portable execution (PE) files from eight highly prevalent malware families and 208,061 logs from benign files. These files were scanned in April 2019 by the company's production file scanning infrastructure over a period of several weeks using their production anti-malware engine.

Following the procedure described in Section II, we first create a training set with 596,478 pairs of either two similar malware files (296,871 pairs) or a malware file and a benign file (299,607 pairs). Each row in the training set is unique. A similar malware pair is chosen to be distinct and a dissimilar pair is formed from the combination of a randomly selected malware file and benign file. For testing, we construct a separate holdout dataset consisting of 311,787 distinct pairs which include 119,562 unique similar pairs and 192,225

	Training Set			Test Set		
Family	File Count	Dissimilar Pair Count	Similar Pair Count	File Count	Dissimilar Pair Count	Similar Pair Count
Ardunk	15793	36592	37194	8272	14828	25545
Clean	130787	na	na	77274	na	na
Dinwod	15472	35014	37127	7281	12979	25809
Nabucur	12097	26895	37987	6475	11364	29524
Picsys	16578	40267	37350	9072	16547	22610
Sfone	18299	47796	37167	10296	19255	16002
Sivis	15907	36704	37330	7959	14221	25572
Soltern	12164	26389	38031	6403	11311	30435
Tescrypt	18220	47214	37421	10258	19057	16728
Total	255317	296871	299607	143290	119562	192225

TABLE III: DataSet counts.

Family	SNN	JI
Ardunk	0.0000%	0.0000%
Clean	0.0060%	0.3200%
Dinwod	0.0000%	0.0000%
Nabucur	0.0310%	3.3300%
Picsys	0.0000%	0.0000%
Sfone	0.0000%	0.0000%
Sivis	0.0500%	0.0000%
Soltern	0.0000%	0.0000%
Tescrypt	0.0100%	0.1400%
All (avg)	0.0110%	0.4200%

TABLE IV: Performance measurements for KNN (K=1) for different highly prevalent malware families.

unique dissimilar pairs. The detailed breakdown per family is shown in Table III.

Experimental Setup. The SNN was implemented and trained using the PyTorch deep learning framework. The deep learning results were computed on an NVidia P100 GPU (graphics processing unit). The Jaccard Index computations were also implemented in Python. For learning, the minibatch size is set to 256, and the learning rate is set to 0.01. The network architecture parameters are shown in Figure 2.

File Similarity. We compare *DeepSim*, which employs an SNN to learn a file embedding in the feature space to a Jaccard Index-based baseline system [1], [2]. The Jaccard Index (JI) for sets A and B is:

$$JI(A,B) = \frac{|A \cap B|}{|A \cup B|}. \tag{1}$$

The elements of the sets correspond to the N-Gram encoding described in Section II for the corresponding feature type (i.e. strings, API events and their parameters).

We begin by comparing the Jaccard Index for both similar and dissimilar files as our baseline in Figure 3. In general, the Jaccard Index is high for similar files and small for dissimilar ones as we expect. However, the figure indicates that the Jaccard Index for a reasonably large number of similar files is less than 0.9. The Jaccard Index for dissimilar files also has a small peak near the value 0.65.

In Figure 4, we next compare *DeepSim's* SNN similarity score for both similar and dissimilar files. Since the range of the SNN scores is $[-1, 1]$ while the Jaccard Index varies from $[0, 1]$, we transform the SNN score to a new value from $[0, 1]$ so that all plots can be compared fairly. In contrast to the Jaccard Index similarity scores, the SNN similarity behaves as expected where the model learns to emphasize the weights from similar files to produce a cosine similarity score very close to 1.0 and the weights from dissimilar files to produce a cosine similarity value which is well separated from the similar files. This behavior makes it much easier to set a value for the SNN threshold which automatically predicts that two files are indeed similar.

During these tests, we found that computing the Jaccard Index is extremely slow. To deal with this limitation and have an Jaccard Index experiment finish within 2-3 days, we had to implement the MinHash algorithm described in Section II where we varied the MinHash filtering threshold. We conducted three different tests including i) training set pairs = 50,000 and test set pairs = 10,000 with a threshold value of 0.5, ii) training set pairs = 10,000 and test set pairs = 10,000 with a threshold value of 0.8, and iii) training set pairs = 100,000 and test set pairs = 10,000 with a threshold value of 0.8. The results reported in Figure 3 and Table V are from test i) with an error rate of 0.42%. The corresponding error rates for tests ii) and iii) are significantly worse at 3.12% and 8.47%, respectively. Using the GPU, we were easily able to evaluate the results for the SNN for all 143,278 items in the test set within 2 days. Thus, the SNN is not only much more accurate, but it is also much faster to evaluate compared to the Jaccard Index.

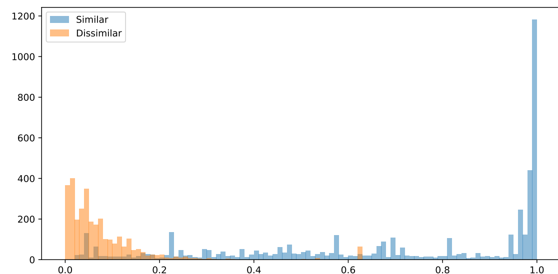

Fig. 3: Jaccard Index similarity score distribution for the similar and dissimilar files.

Family Classification While Figures 3 through 4 demonstrate that SNN produces a much improved similarity score com-

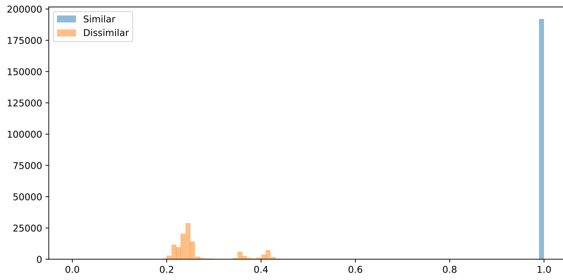

Fig. 4: SNN similarity score distribution for the similar and dissimilar files.

pared to the Jaccard Index, it is important to understand if this leads to improved detection rates. To investigate, we next compare the K-Nearest Neighbor classification results, with $K = 1$, for these two similarity models. We utilized the Facebook AI Similarity Search library [16].

Evaluating a KNN classifier requires us to revisit our test set. As described previously, we form similar pairs of malware files whose SignatureID and family match and dissimilar pairs where the second file is known to be benign. The *DeepSim* model is then used to compare an unknown file to the set of known malware files (i.e. the left-hand side of Figure 2) to determine if it is similar to any of these previously detected malware files. However to evaluate the output of the *DeepSim* model for a KNN classifier, we must instead compare an unknown file to a set of known malware and *benign* files. For our test set, this is the set of files denoted as F on the right-hand side of Figure 2. In essence, we have swapped the set of known test files from the left-hand side of Figure 2 to the right-hand side. When evaluating an unknown file using the KNN classifier, the distances from the unknown file to the known set of malware and benign files are computed, and the label is determined from the majority vote of the K closest files. It should be noted that in the following test, the test files are all malicious and we are trying to determine the malware file's family .

Since our data set does not allow us to form pairs of similar benign files, we cannot measure the false positive and true negative rates. If a file is benign, we cannot compute the score for similar benign files from the data. For example, Chrome.exe is not similar to AcroRd32.exe even though both files are benign. If a file is a non-matching file, we cannot compute this because we have generated the KNN results from a combination of the matching and non-matching pairs.

Table V summarizes several performance metrics for malware family classification. The table indicates that the SNN outperforms or is equivalent to the JI for most of the families. Overall, the SNN has an error rate of 0.011% compared to 0.420% for the JI.

The previous results suggest that the individual malware families are well separated in the latent vector space where the latent vector is the final embedding which is output by each branch of the SNN in Figure 2. In Figure 5, we use the t-sne method [17] to project these vectors into a two-dimensional space. This figure confirms that these classes are indeed well separated.

Family	SNN	JI
Ardunk	0.0000%	0.0000%
Clean	0.0060%	0.3200%
Dinwod	0.0000%	0.0000%
Nabucur	0.0310%	3.3300%
Picsys	0.0000%	0.0000%
Sfone	0.0000%	0.0000%
Sivis	0.0500%	0.0000%
Soltern	0.0000%	0.0000%
Tescrypt	0.0100%	0.1400%
All (avg)	0.0110%	0.4200%

TABLE V: Performance measurements for KNN (K=1) for different highly prevalent malware families.

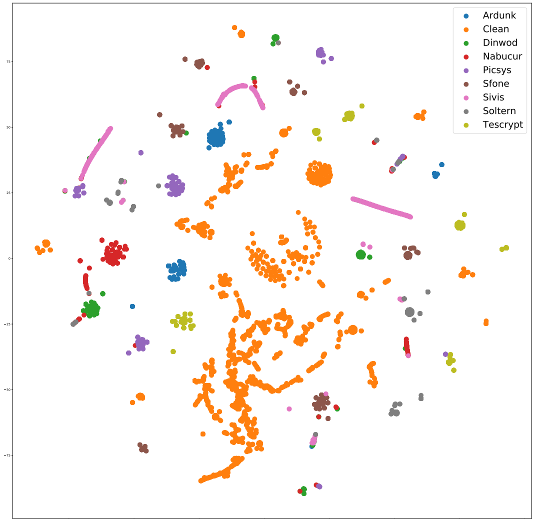

Fig. 5: Visualization of the separability of the latent vectors of the malware classes using the t-sne method.

IV. Evasion

Since *DeepSim* utilizes dynamic analysis for extracting its underlying features, it is susceptible to previously proposed methods that attackers use to detect and thwart these types of analysis systems. A nice summary of the different types of attacks employed against dynamic malware classification systems can be found in [18] and [3].

To avoid detection, malware sometimes employs cloaking [19], [20] which occurs when it does not perform any malicious action if it detects that it may be executing in an emulated or virtualized environment typically used by dynamic analysis systems. To overcome these attacks, researchers often propose systems which execute an unknown file in multiple analysis systems and search for differences among the outputs

which might indicate that malware is using cloaking [3], [21], [22].

Malware may also try to delay the execution of its malicious code hoping that the emulation engine eventually halts after some amount of time [20], [21]. To prevent this evasion technique, dynamic analysis systems may alter the system time. This in turns leads to malware checking the time from the internet [3]. If malware is not able to connect to the internet to check the time, it may choose to halt all malicious activity.

Recently, other researchers in adversarial learning have begun to explore attacking deep learning system by crafting examples which are misclassified by a deep neural network [23]. For malware analysis [24], a white-box attack requires access to the DNN's model parameters which means successfully breaching an anti-malware company's corporate network if the DNN analyzes unknown files in a backend service. Thus, successfully employing this attack on a backend service would be very challenging. If a DNN were to be run locally on the client computer, attackers may be able to reverse engineer the model and successfully run a variant of this attack on *DeepSim's* SNN architecture. To combat this attack, it might be possible to run the SNN in an SGX enclave.

V. Related Work

Similarity and Clustering. *DeepSim* is most closely related to systems which consider the discovery of similar files. A few of these systems include the following research efforts.

Lee and Mody [7] proposed a system to cluster files based on their behavior using k-means clustering. The distance metric used in their system is the Euclidean distance based on features derived from dynamic analysis of a file.

A scalable malware clustering system is presented by Bayer et al. in [1]. In this work, the authors extend Anubis for dynamic analysis of malware and create behavior profiles from these execution traces. A locality sensitive hashing scheme is used to significantly reduce the number of pairs which must be considered. Malware files are grouped by hierarchical clustering where the Jaccard Index is used as the similarity metric. In [13], Li et al. further investigate the challenges of evaluating different malware clustering algorithms by trying to cluster malware using algorithms proposed for detecting plagiarism.

Perdisci et al. [25] cluster HTTP traffic to discover malware. The system merges coarse-grained clusters of statistical features with fine-grained clusters of structural features to automatically generate signatures. The fine-grained features are computed in part using the Jaccard Index.

Jang et al. propose a large-scale malware clustering system called BitShred [2]. BitShred employs feature hashing, bit vectors, and a MapReduce implementation on Hadoop to significantly speed up the computation and reduce the memory consumption compared to previously proposed systems on a single CPU. BitShred also uses the Jaccard Index as the similarity measure for its co-clustering algorithm.

In [26], Gregoire et al. propose a new method for detecting similar malware files based on static analysis. This system can be used as a prefilter to reduce the number of files submitted for more in-depth analysis by a factor ranging from three to five. Pearson's chi squared test is used as the similarity metric in Gregoire's system.

In DISARM [3], Lindorfer et al. compare the Jaccard Index between the behavior profiles generated by multiple instances of dynamic analysis of a single unknown file in multiple Anubis sandboxes to detect similar malware. Their results reveal previously unknown evasion behavior in the malware samples they analyzed.

Rafique and Caballero built a system called FIRMA to cluster malware [27] and generate signatures based on network traffic. FIRMA's clustering is based on both the application protocol and the transport protocol. The application protocol clustering is based in part on the Jaccard Index of the URL parameters.

Kong and Yan [28] extract the functional call graph by static analysis of malware files. They next learn a similarity distance metric based on the functional call graph and use this to learn an ensemble of classifiers to detect unknown malware.

To overcome the evasion techniques malware uses to detect if it is being emulated, Kirat et al. [22] proposed BareCloud which compares the execution of an unknown file on four different analysis systems including Bare-Metal which executes the malware on the native operating system and analyses the network traffic and changes on the disk drive, Ether which supports hypervisor-based malware detection, Anubis which uses emulation to detect malware, and the Cuckoo Sandbox which uses Virtualbox to provide virtualization-based detection. BareCloud then discovers differences in the outputs of these systems to detect evasion behavior by the file. To compare the outputs of the different analysis systems, the authors propose a new hierarchical similarity measure.

Stock et al. propose Kizzle [29] which uses the DBSCAN algorithm to cluster JavaScript malware. In particular, the authors use the system to identify several exploit kits.

Deep Learning. Although malware classification has been an active area of research starting in 1994 [30], we primarily focus on the earlier work employing neural network architectures [6]. Interestingly, Kephart et al. [31] were the first to discuss malware classification in general, and they proposed a neural network-based system.

Dahl et al. [8] were the first authors to investigate the use of deep learning in malware classification. In this paper, the authors proposed a deep, feed forward neural network with sigmoid activation functions and the softmax output function. Although their experiments did not indicate that adding more than one hidden layer improved the classification results, their analysis showed a reduction of 43.0% for a shallow neural network with a single hidden layer compared to logistic regression.

A one-sided perceptron was combined with a Restricted Boltzmann Machine (RBM) for malware classification by Benchea and Gavrilut [32].

Saxe and Berlin [10] trained a two-layer deep neural network based on features extracted during static analysis of a

file. Overall, this model shows very good promise for static malware classification. However, they did not provide results for a shallow network or architectures for more than two hidden layers, so it remains unclear whether or not deep learning helps with static malware classification.

Pascanu et al. [9] recently proposed using several different models based on both recurrent neural networks (RNNs) and echo state networks (ESNs). The basic architecture of echo state networks is similar to recurrent neural networks, except that the ESN model parameters are randomly initialized whereas they are typically learned using backpropagation in an RNN.

VI. Conclusions

When not successfully subverted by emulation and virtualization detection techniques, dynamic analysis has been shown to yield excellent results in detecting unknown malicious files. In this paper, we have proposed *DeepSim* — a dynamic analysis system for learning malware file similarity based on a Siamese Neural Network. We demonstrate the performance of *DeepSim* on highly prevalent families which require a large amount of support by analysts and automated analysis. We have shown that *DeepSim* offers significant improvement in the K-Nearest Neighbor classifier error rate compared to a similar system based on the Jaccard Index which has been previously proposed for several other systems. The results show that *DeepSim* reduces the KNN classification error rate on these highly prevalent families from 0.420% for the Jaccard Index to 0.011% for a SNN with two hidden layers. As such, we believe that *DeepSim* can be an effective tool for reducing the amount of analysts' time and automation costs spent combatting these highly prevalent malware families.

References

[1] U. Bayer, P. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering," in *Annual Network and Distributed System Security Symposium (NDSS)*, 2009.

[2] J. Jang, D. Brumley, and S. Venkataraman, "Bitshred: Feature hashing malware for scalable triage and semantic analysis," in *CCS*, 2011.

[3] M. Lindorfer, C. Kolbitsch, and P. M. Comparetti, "Detecting environment-sensitive malware," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2011.

[4] Y. A. W. Du and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *International Conference on Security and Privacy in Communication Networks (SecureComm)*, 2013.

[5] J. Hegedus, Y. Miche, A. Ilin, and A. Lendasse, "Methodology for behavioral-based malware analysis and detection using random projections and k-nearest neighbors classifiers," in *Proceeding of the International Conference on Computational Intelligence and Security (CIS)*, 2011.

[6] C. M. Bishop, *Pattern Recognition and Machine Learning*. Springer, 2007.

[7] T. Lee and J. Mody, "Behavioral classification," in *Annual Conf. of the European Institute for Computer Antivirus Research (EICAR)*, 2006.

[8] G. E. Dahl, J. W. Stokes, L. Deng, and D. Yu, "Large-scale malware classification using random projections and neural networks," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2013, pp. 3422–3426.

[9] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas, "Malware classification with recurrent networks," in *Proceeding of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2015, pp. 1916–1920.

[10] J. Saxe and K. Berlin, "Deep neural network based malware detection using two dimensional binary program features," *arXiv preprint arXiv:1508.03096v2*, 2015.

[11] J. Bromley, I. Guyon, Y. Lecun, E. Säckinger, and R. Shah, "Signature verification using a siamese time delay neural network." vol. 7, 01 1993, pp. 737–744.

[12] C. D. Manning, P. Raghavan, and H. Schutze, *An Introduction to Information Retrieval*. Cambridge University Press, 2009.

[13] P. Li, L. Liu, D. Gao, , and M. K. Reiter, "On challenges in evaluating malware clustering," in *Proceeding of the International Symposium on Recent Advances in Intrusion Detection Symposium (RAID)*. Springer, 2010, pp. 238–255.

[14] P.-S. Huang, X. He, J. Gao, L. Deng, A. Acero, and L. Heck, "Learning deep structured semantic models for web search using clickthrough data," in *International Conference on Information and Knowledge Management (CIKM)*, 2013.

[15] A. Rajaraman and J. D. Ullman, *Mining of Massive Datasets*, 2nd ed. Cambridge: Cambridge University Press, 2014.

[16] J. Johnson, M. Douze, and H. Jégou, "Billion-scale similarity search with gpus," *arXiv preprint arXiv:1702.08734*, 2017.

[17] L. van der Maaten and G. Hinton, "Visualizing data using t-sne," 2008.

[18] X. Chen, J. Andersen, Z. M. Mao, M. Bailey, and J. Nazario, "Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware," in *Proceedings of the Annual IEEE International Conference on Dependable Systems and Networks (SIGMETRICS)*, 2008.

[19] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi, "A fistful of redpills: How to automatically generate procedures to detect cpu emulators," in *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2009.

[20] T. Raffetseder, C. Kruegel, and E. Kirda, "Detecting system emulators," in *Proceedings of the Information Security Conference (ISC)*, 2007.

[21] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, , and G. Vigna, "Efficient detection of split personalities in malware," in *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2010.

[22] D. Kirat, G. Vigna, and C. Kruegel, "Barecloud: Bare-metal analysis-based evasive malware detection," in *Usenix Security Symposium*, 2014.

[23] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial systems," *IEEE European Symposium on Security and Privacy*, 2016.

[24] J. W. Stokes, D. Wang, M. Marinsescu, M. Marino, and B. Bussone, "Attack and defense of dynamic analysis-based, adversarial neural malware detection models," in *Proceeding of the IEEE Military Communications Conference (MILCOM)*, 2018.

[25] R. Perdisci, W. Lee, and N. Feamster, "Behavioral clustering of http-based malware and signature generation using malicious network traces," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.

[26] G. Jacob, P. M. Comparetti, M. Neugschwandtner, C. Kruegel, and G. Vigna, "A static, packer-agnostic filter to detect similar malware samples," in *Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, 2012.

[27] M. Z. Rafique and J. Caballero, "FIRMA: Malware Clustering and Network Signature Generation with Mixed Network Behaviors," in *Proceedings of the International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, October 2013.

[28] D. Kong and G. Yan, "Discriminant malware distance learning on structural information for automated malware classification," in *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2013.

[29] B. Stock, B. Livshits, and B. Zorn, "Kizzle: A signature compiler for exploit kits," in *Microsoft Research Tech Report MSR-TR-2015-12*, 2015.

[30] N. Idika and A. P. Mathur, "A survey of malware detection techniques," Purdue Univ., Tech. Rep., February 2007. [Online]. Available: http://www.eecs.umich.edu/techreports/cse/2007/CSE-TR-530-07.pdf

[31] J. O. Kephart, "A biologically inspired immune system for computers," in *Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*. MIT Press, 1994, pp. 130–139.

[32] R. Benchea and D. T. Gavrilut, "Combining restricted boltzmann machine and one side perceptron for malware detection," in *International Conference on Conceptual Structures (ICCS)*, 2014, pp. 93–103.