One-Shot Malware Outbreak Detection Using Spatio-Temporal Isomorphic Dynamic Features

Sean Park, Iqbal Gondal, Joarder Kamruzzaman, Leo Zhang

TREND MICRO | research

Published by

Trend Micro Research

Written by

Sean Park

Trend Micro Research

Iqbal Gondal

Federation University Australia

Joarder Kamruzzaman

Federation University Australia

Leo Zhang

Trend Micro Research

Stock image used under license from Shutterstock.com

Contents

03

Abstract

04

I. Introduction

06

II. Related Works

07

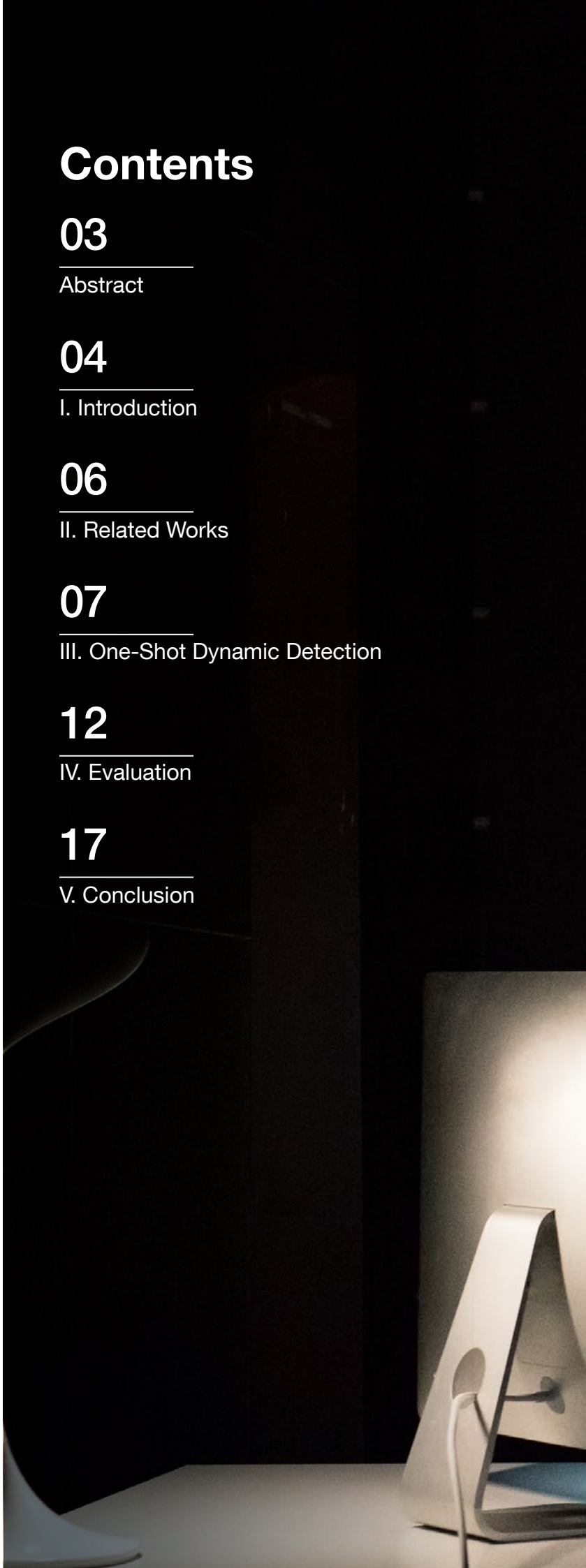
III. One-Shot Dynamic Detection

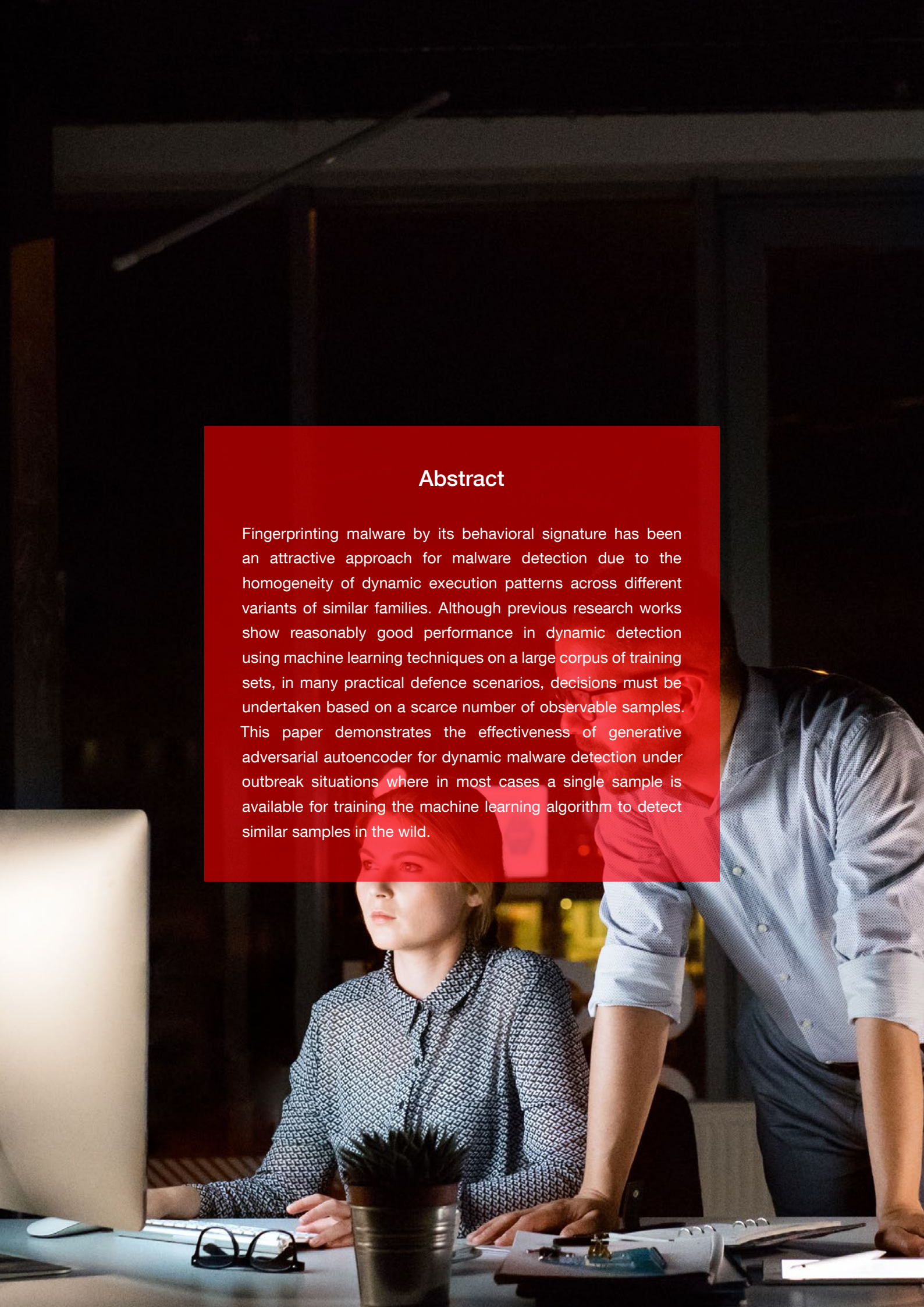
12

IV. Evaluation

17

V. Conclusion





Abstract

Fingerprinting malware by its behavioral signature has been an attractive approach for malware detection due to the homogeneity of dynamic execution patterns across different variants of similar families. Although previous research works show reasonably good performance in dynamic detection using machine learning techniques on a large corpus of training sets, in many practical defence scenarios, decisions must be undertaken based on a scarce number of observable samples. This paper demonstrates the effectiveness of generative adversarial autoencoder for dynamic malware detection under outbreak situations where in most cases a single sample is available for training the machine learning algorithm to detect similar samples in the wild.

I. Introduction

Every piece of malware has its own purpose, whether it is a banking trojan, ransomware variant, an information stealer, or advanced persistent threat. Therefore, it is also named by its behavior. All top-level malware categories contain multiple families, each of which has many variants. Despite the complex hierarchy of malware samples, the invariancy of execution offers a great opportunity to detect not only a wide range of malware family variants but also multiple families that share similar behavior patterns. This paper proposes a detection method that leverages this common nature of malware.

Since the early 21st century, many individuals and organizations have been breached and have suffered financial losses due to ransomware.[1] Although ransomware has a large number of different families, its execution fundamentally involves a behaviorally similar pattern where it downloads a custom encryption key from a remote command and control (C&C) server, enumerates the file system for the targeted files in the victim's machine, encrypts the target files using the key, displays the ransom note, and finally extorts the victim for payment. Likewise, various banking trojans that have impacted the world over the past decade, for example, Zeus, Dyre, and Trickbot, have kept their core behavior unchanged although they have gone through numerous binary structure changes to bypass static detection. After many years, the fact is that banking trojans have not changed their routine. They still hijack web traffic, inject a custom Java script within a target online banking page, and initiate or modify a covert transaction to underground channels. Even though there are minor variations in the behaviors of the variants of a family or a group of families, the substantial body of the execution remains relatively invariant. As a result, there appears a phenomenon (see Figure 1) commonly seen among different variants and, quite often, among cross-family malware samples that are behaviorally isomorphic in temporal behavior space. The method proposed in this paper will capitalize on the presence of this pattern exhibited by behaviorally similar malware samples.

The intrinsic behavior of a sample is best defined by its dynamic execution log. As with many previous works, this paper will leverage the API call sequence as a feature in our model while maintaining its spatio-temporal characteristics. The API call logs are obtained via a custom sandbox[2] that captures a selected set of user-mode API calls along with their parameters. Dynamic detection has several challenges of its own. A multithreaded operating environment can result in the API call events shuffled in threaded code blocks. A sample can exhibit several different behaviors depending on how it is executed in a sandbox. Anti-analysis and nondeterministic programs can lead the sample to different execution paths. Resilience

over mutation of dynamic execution is a long-term challenge as well. All of these perturbations in the event log interfere with accurate dynamic detection. In this paper, the global feature that captures local characteristics is deemed optimal for dynamic detection models.

From a practical defense point of view, time-to-detection is a critical factor when there is a new malware outbreak and several thousand variants of its kind could soon follow. The task involves building a quick knowledge base out of a handful of samples that are available from the outbreak, in such a manner that the detection is resilient to minor changes in malware behavior. This is very different to the traditional problem setting where a large corpus of training samples are available and used to predict the maliciousness of test samples. Due to the availability of only a limited number of training samples, the problem in this paper is posed as similarity-based detection instead of traditional binary classification.

The order of the API call sequence within a block is the key to the semantics of the behavior. Unfortunately, traditional machine learning models such as Support Vector Machine (SVM), Random Forest, and Gradient Boosting generally use n-gram,[3] which is computed from the original sample in order to come up with a fixed-size feature vector, thus taking away the important context information present in the global scope. This paper explores a deep learning model that fully utilizes the order of API call sequence information. Given that similarity-based detection is desired, a generative deep learning model is used and evaluated against several traditional machine learning approaches.

This paper presents a novel method that detects malware outbreaks using dynamic execution features when very limited training samples are available. The proposed model, generative adversarial autoencoder, automatically extracts perturbation-resilient and context-aware features from the raw API call sequences to detect malware outbreaks proactively in practical threat response environments.

II. Related Works

Youngjoon Ki et al. handcrafted API call sequences using Longest Common Subsequences (LCS) and Multiple Sequence Alignment (MSA) for detection,[4] which are vulnerable to obfuscation techniques such as bogus API call insertion. Davide Canali et al. studied a systematic way of extracting common signatures of a given dataset.[5] Manuel Egele et al. explained various controlled environments to extract system events required for dynamic malware detection.[6] Several approaches take advantage of various machine learning techniques experimenting with Random Forest, k-Nearest Neighbor, Naïve Bayes, J48 Decision Tree, Support Vector Machine, and Multilayer Perceptron Neural Network.[7, 8, 9, 10, 11] They essentially use API call sequences as a feature and use a large corpus of training samples for classification. Grégoire Jacob et al. explained the efficacy of behavioral signature-based detection over its static counterparts.[12] Adaptability and resilience are two important properties of the assessment. Dong-Jie Wu et al. introduced the use of clustering on Android dynamic detection, utilizing k-means and EM (Expectation Maximization) algorithm.[13] Faraz Ahmed et al. implemented a binary classification model using a spatio-temporal feature set selected by information gain.[14] Wenyi Huang and Jack Stokes created the first deep-learning-based approach using Multilayer Perceptron.[15] Bojan Kolosnjaji et al. investigated with deep learning and tried to solve a binary classification problem using convolutional neural network and recurrent neural network.[16] Sean Park et al. discussed malware outbreak detection using static features.[17]

The majority of the previously mentioned approaches tried to solve a binary classification problem over a relatively large training set as opposed to setting the outbreak situation as the problem, which is the focus of this paper. The proposed approach described in this paper attempts to identify similar samples after training with a single malicious sample of each kind.

III. One-Shot Dynamic Detection

This section discusses the features, model, and metric of the proposed method. The proposed method takes advantage of the generative power of adversarial neural network against spatio-temporal features extracted from API call events, which shows remarkable generalization of a single sample of its kind used in the training.

A. Features

Figure 1 illustrates the API call events of the variants of a malware family, which are identified by the model proposed in this paper. Although there are several variations in the family name, the visual representation of the API call events for this family remains very much isomorphic.

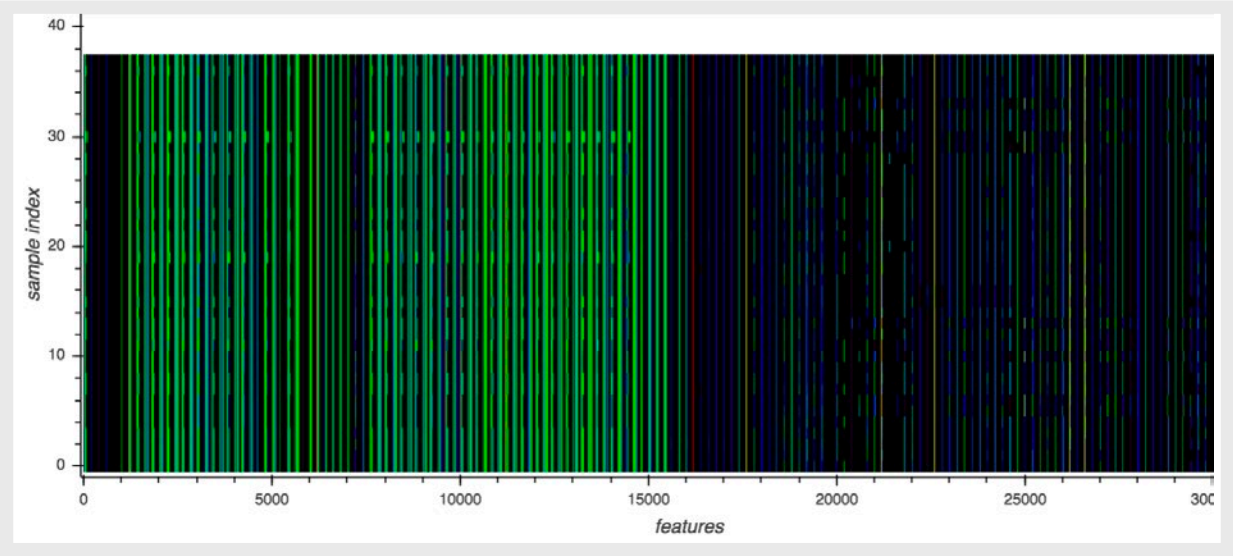


Figure 1. Visualization of the API call events of variants of malware family

Note: Each row represents a per-sample feature, which is a sequence of API call events made by a malware sample. Each normalized API call event is rendered as a pixel with a unique color assigned via a lookup table. The X-axis is the feature while the Y-axis is the sample number. This cluster contains 38 samples (HO_WINPLYER.MSMIU18, 15 samples; OSX_Agent.PFL, 3 samples; OSX_Generic.PFL, 1 sample; OSX_SearchPage.PFM, 18 samples; OSX_WINPLYER.RSMSMIU18, 1 sample).

The steps to construct a feature are as follows (an example is shown in Figure 2).

1. Map API call name to a unique ID.
2. Assign a unique ID for each character found in the API call arguments.
3. Pad each API call event with zeros to a predefined fixed size.

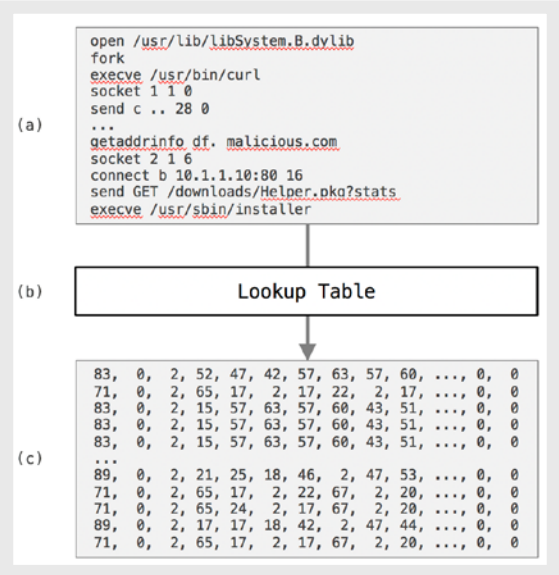


Figure 2. An example of dynamic execution log transformation

Note: Shown in the figure are (a) a dynamic execution log, (b) symbol-to-ID lookup table constructed by creating a map of symbols from all training samples, and (c) a two-dimensional (2D) array constructed from (a) with padded zeros at the end of each API call event.

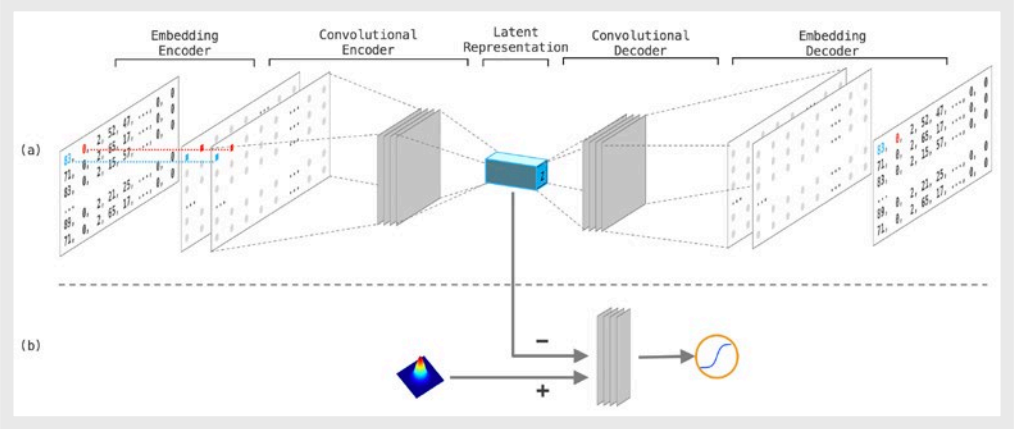


Figure 3. Generative adversarial autoencoder with 2D API call event feature and embedding

Note: (a) Autoencoder with each value in 2D feature mapped to an embedding. Note that the value in blue gets mapped to an embedding of size two at its corresponding position marked with a dotted blue line. The embedding value is chosen from the embedding lookup table for each symbol at the input feature. The same applies to the value in red. (b) Discriminator with positive samples from Gaussian normal distribution and negative samples from the latent representation, z, obtained from the input feature.

A dynamic execution log consists of a sequence of API call events, each of which is broken down into an API identifier and its corresponding API arguments. A sample dynamic execution log is shown in Figure 2(a). The symbol lookup table shown in Figure 2(b) is constructed by creating a map of all possible call argument characters. This symbol lookup is performed on a per-character instead of on a per-word basis.

This design decision was made to allow an arbitrary number of arguments and values. Figure 2(c) shows the final feature transformed from the dynamic execution log, which will be fed to the deep learning model.

B. Model

Adversarial autoencoder[18] forms the core of the model, which consists of an autoencoder and generator-discriminator pair. In addition, DCGAN[19] is used as an autoencoder in order to cope with various perturbations that occur in API call events. The proposed model is jointly trained with stochastic gradient descent by minimizing the reconstruction loss on spatio-temporal input **x** over latent representation **z** in Equation (1) and by performing min-max adversarial game in Equation (2). Consensus optimization[20] is also applied on top of Adam optimization[21] to aid stable GAN[22] training. Batch normalization[23] is used within the GAN generator to help generate stable latent representations.

$$E_{x}[E_{q(z|x)}[-\log p(x|z)]] \quad (1)$$

$$\min_{G}\max_{D} \; E_{x\sim p_{data}}[\log D(x)] + E_{z\sim p(z)}[\log(1-D(G(z))] \quad (2)$$

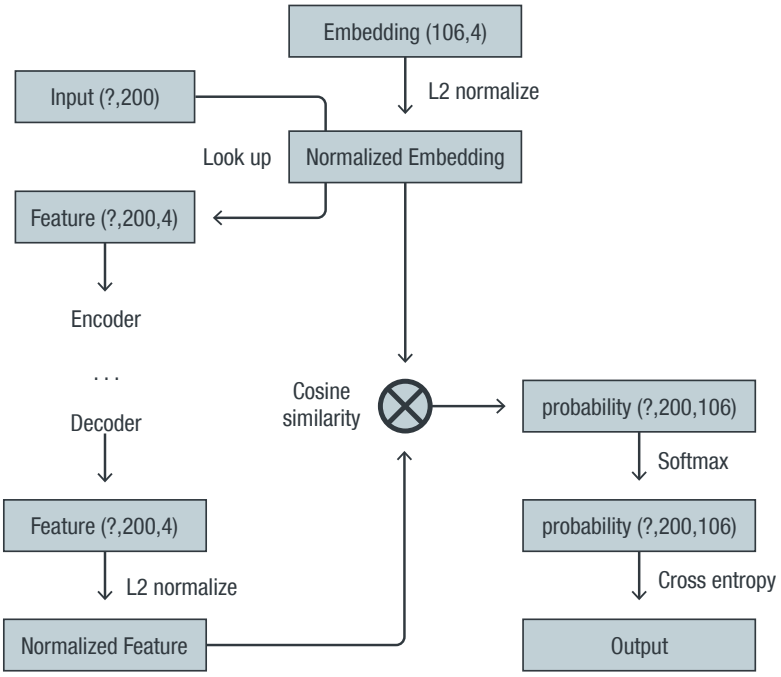


Figure 4. Autoencoder loss calculation diagram

However, several key adjustments are made on top of the base adversarial autoencoder model to fit the two-dimensional symbol inputs into the model and to generate consistent latent representation over variable length inputs.

The input feature is a two-dimensional array with a fixed width. As shown in Figure 3(a), each symbol at the input is mapped to an embedding vector using an embedding lookup table, which transforms the 2D symbol inputs to the three-dimensional (3D) array required by the convolutional encoder. The autoencoder cost calculation process is shown in Figure 4 for each API call event where the feature length is limited to 200, the embedding vector size is 4, and the symbol lookup table size is 106. The variable length API call event is denoted as "?". The reconstructed symbol probability is calculated via cosine similarity obtained as an inner product of L2 normalized reconstructed input and L2 normalized embedding.[24] *Output* of Figure 4 contains the cross entropy between the one hot-encoded input symbol and the softmax'ed reconstructed symbol probability. The autoencoder cost is calculated by taking the mean of the sum of the negative log likelihood value at each symbol position in the input.

Furthermore, variable length inputs pose a challenge in producing consistent latent representations. Although the convolution operation can be performed against arbitrary length inputs, the size of the latent representation, z, can vary depending on the input length. This is not a desirable property when a fixed size **z** is needed for similarity comparison during detection. It is required to perform pooling on the variable size convolutional filters produced by variable length inputs. While global max pooling works well when the features are extracted for classification problems, it performs poorly for autoencoders because each **z** bit gets biased to a relatively larger value, saturates to a distinct bit range, generates poor reconstruction, and therefore negatively impacting the accurate clustering of similar samples. Our experiments show that global average pooling performs better for autoencoders. Notably, Gaussian normal distribution that drives adversarial training is nicely imposed on z bits with global average pooling (see Figure 5).

As shown in Figure 6, global average pooling is performed on the last convolutional layer of the encoder by producing the z bits from each filter's mean value (blue part of Figure 6). Then z is unpooled by setting bit values to the decoder filters at the argmax index locations saved from the encoder filters (red part of Figure 6).

C. Metric

The latent representations obtained through adversarial autoencoder contain real numbers that can be directly compared with each other by calculating the mean squared error (MSE).

If binary bits are desired for efficient comparison, z needs to be binarized using the bitwise mean value of the training samples, which splits the bit distributions into halves and subsequently helps to better distinguish between samples. Adversarial training driven by Gaussian normal distribution significantly

helps in spreading the bits evenly. Hamming distance is then used to compute the distance for the given two latent vectors. However, our experiment indicates that MSE metric over real valued z performs slightly better than hamming distance over binary z bits.

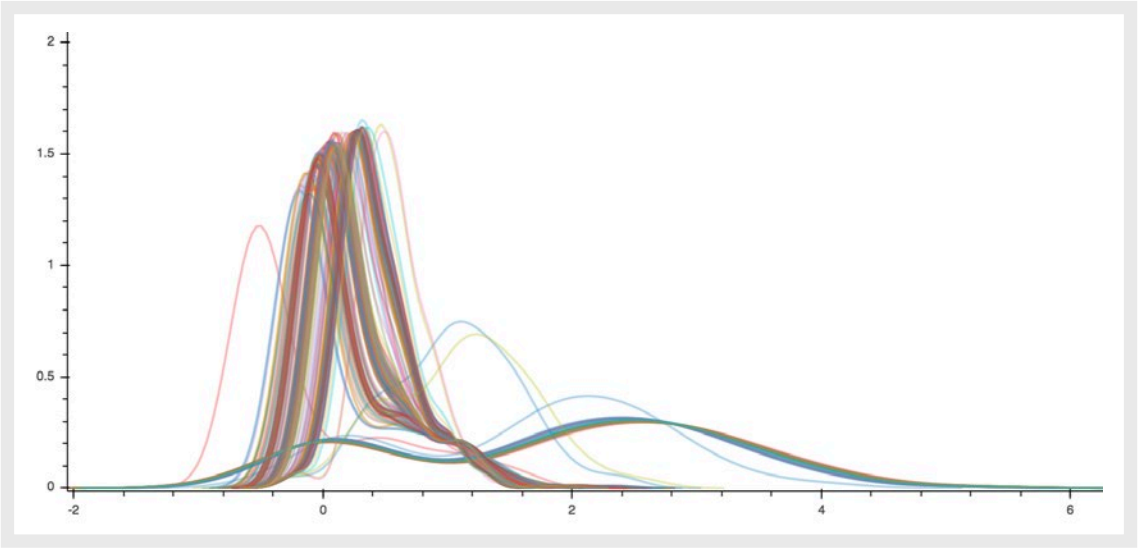


Figure 5. Distribution of each bit in z

Note: Each bit has Gaussian normal distribution as instructed by adversarial training, which helps distinguish between different samples. Global average pooling allows this adversarial property as well as the sound reconstruction required for autoencoder.

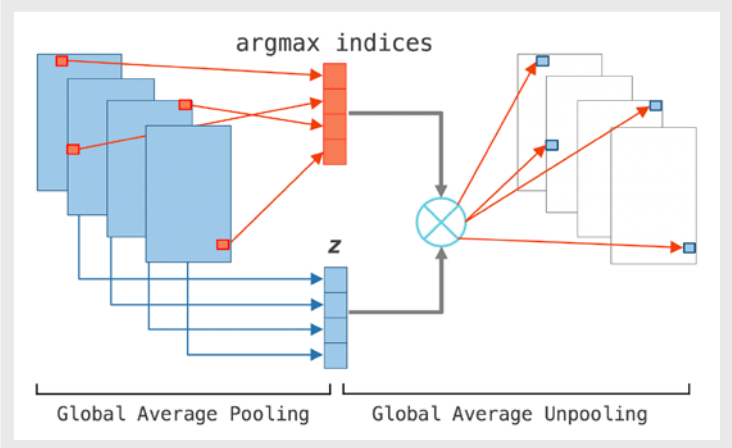


Figure 6. Global average pooling, latent representation (z) generation, and global average unpooling

Note: The argmax of each encoder filter is marked in red. The mean value of each encoder filter is marked in blue.

IV. Evaluation

A. Baselines and Dataset

Gradient Boosting, Support Vector Machine, and Random Forest models were chosen as the baselines that the proposed model is evaluated against. The two-dimensional variable length input feature vector used for the proposed model is collapsed to a one-dimensional array in order to produce an n-gram feature[25] for the baseline models. Note that clustering models were not included in the baseline since they need a decent number of training samples to work, which is different from the problem setting put forward in this research.

Dynamic execution logs of 2,855 in-the-wild OS X malware samples collected from a proprietary commercial sandbox and 7,541 benign OS X samples were used for evaluation. A snapshot of family distribution for a portion of 2,855 in-the-wild malware samples is shown in Figure 7. Dynamic execution logs were analyzed and labeled by human experts, with focus on the similarity of API call event sequences. Of the samples, 353 unique API call sequence patterns out of 2,855 malicious samples were identified and were used for training. The 7,541 benign samples were split into mutually exclusive training and testing sets for all baseline models while benign samples were not included for the training of the proposed model in order to test its outbreak detection capability.

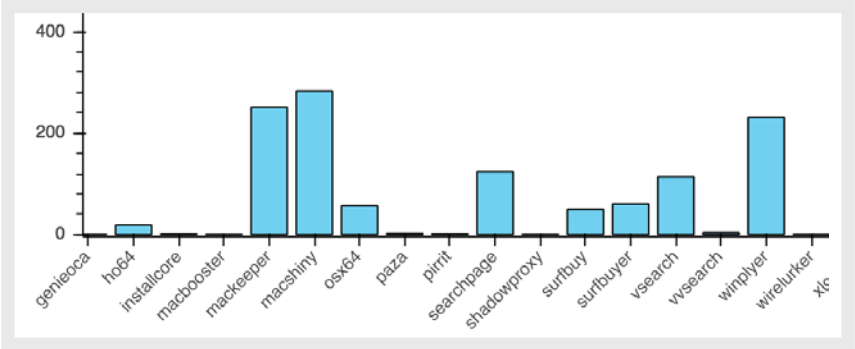


Figure 7. A snapshot of the family distribution for a portion of the 2,855 in-the-wild malware samples used in the evaluation

	Experiment 1	
	Malicious	**Benign**
Train	353	**3,770**
Test	2,855	**3,771**

Table 1. Sample splits

B. Results

To simulate an outbreak, we assigned a unique label for each of the 353 unique malicious training samples and evaluated how accurately the label matches with the prediction. Therefore, the TP (True Positive) ratio in Table 2 shows the label-wise matches of all malicious testing samples against malicious training samples.

Baseline models did not produce a good TP ratio against malicious samples, scoring less than 50% detection rate, while near zero FP (False Positive) was recorded. In short, it demonstrates that the supervised baseline models trained with a very small number of malicious training samples are not suitable for the problem setting of outbreak detection. On the other hand, *aae-mse*, the proposed generative adversarial autoencoder using mean squared error (MSE) as autoencoder loss, showed 99.1% true positives along with 0.1% false positives, with MSE decision threshold set to 0.000025. This threshold influences how much the model generalizes the detection.

Model	TP	FP
gradient-boosting-1gram	0.194	0.026
gradient-boosting-2gram	0.108	0.027
svm-1gram	0.056	0.000
svm-2gram	0.056	0.000
randomforest-1gram	0.460	0.000
randomforest-2gram	0.385	0.000
aae-mse (proposed model)	**0.991**	**0.001**

Table 2. Evaluation results

Threshold	TP	FP
0.000500	1.000	0.306
0.000100	1.000	0.057
0.000075	1.000	0.033
0.000050	1.000	0.010
0.000025	**0.991**	**0.001**
0.000010	0.860	0.000

Table 3. Effect of threshold

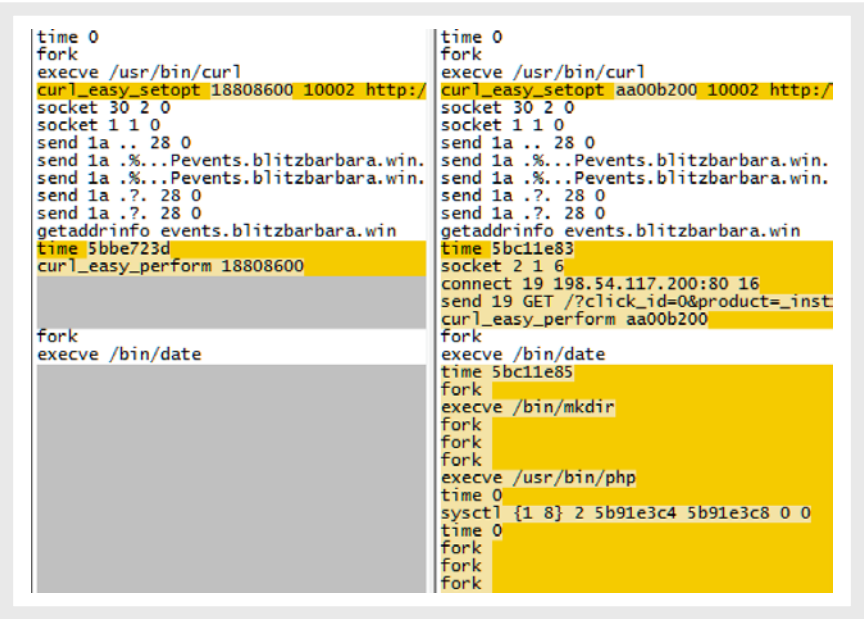


Figure 8. An example of API call snippets for two similar samples detected by the proposed model

Note: White lines indicate identical events. Yellow lines indicate non-identical events due to a difference in the event. Gray lines indicate absent events relative to the other event.

The higher the threshold is set, the more reliable the detection is whereas the model is more likely to miss the mutated samples. The effect of various threshold values is shown in Table 3. The threshold in bold shows the performance described in Table 2. One can draw the inference with a sufficiently low threshold for accurate detection while running a separate inference with a higher threshold to grab potentially malicious samples for further analysis.

In summary, the analysis showed that the proposed model is capable of reliably generalizing the sample variations while keeping a sufficiently fair distance from previously unseen benign samples. In contrast, traditional classifiers are much less likely to work for detection under the malware outbreak situations.

C. Analysis

In this section, we analyze several key aspects of how the latent representation produced by generative adversarial autoencoder is correlated to the semantics of API call events.

```
send d ... 24 0                          send d ... 24 0
send d ... 24 0                          send d ... 24 0
getaddrinfo erbnx.magnetlotus.win        getaddrinfo kafgp.protectcanyon.win
getaddrinfo erbnx.magnetlotus.win        getaddrinfo kafgp.protectcanyon.win
time 5bbd9ebf                            time 5bc72a1d
socket 2 1 6                             socket 2 1 6
connect d 23.219.92.154:80 16            connect d 23.219.92.144:80 16
send d GET /sdl/mmStub.tar.gz?ts=153915359 send d GET /sdl/mmStub.tar.gz?ts=1539779099
curl_easy_perform cf005e00               curl_easy_perform 7900be00
fork                                     fork
execve /usr/bin/bsdtar                   execve /usr/bin/bsdtar
dlopen /usr/lib/libSystem.B.dylib 2      dlopen /usr/lib/libSystem.B.dylib 2
time 5bbd9ec2                            time 5bc72a1d
time 0                                   time 0
chmod ./mm-install-macos.app/Contents/_Code chmod ./mm-install-macos.app/Contents/_Code
chmod ./mm-install-macos.app/Contents/Resou chmod ./mm-install-macos.app/Contents/Resou
chmod ./mm-install-macos.app/Contents/MacOS chmod ./mm-install-macos.app/Contents/MacOS
chmod ./mm-install-macos.app/Contents/ 4075 chmod ./mm-install-macos.app/Contents/ 4075
chmod ./mm-install-macos.app/ 40755      chmod ./mm-install-macos.app/ 40755
fork                                     fork
execve /bin/chmod                        execve /bin/chmod
dlopen /usr/lib/libSystem.B.dylib 2      dlopen /usr/lib/libSystem.B.dylib 2
fork                                     fork
```

Figure 9. An example of API call snippets of two similar samples detected by the proposed model

Note: Both samples are distinguishable by the access to different remote network nodes.

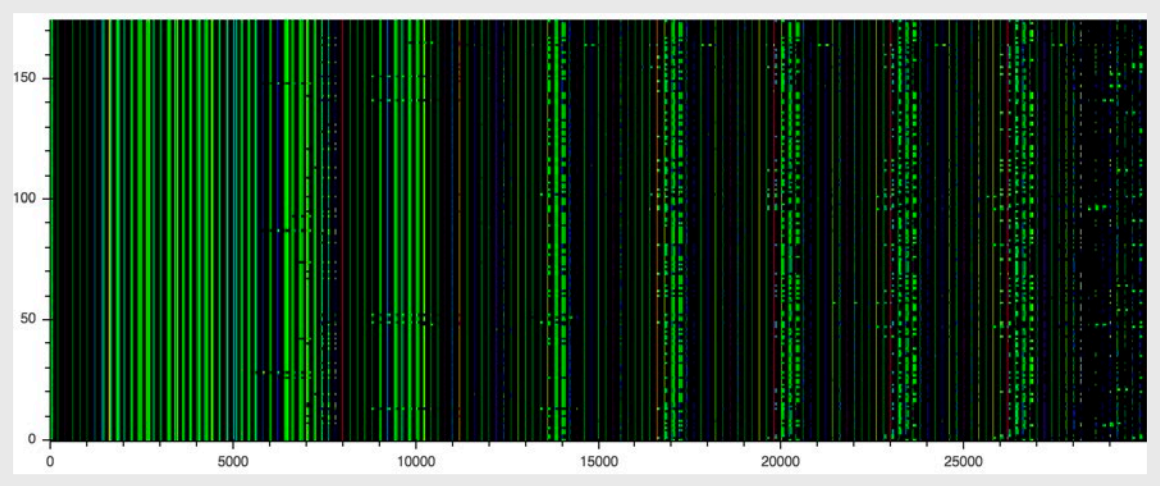


Figure 10. Visualization of Cluster 2 detected by *aae-mse* (nsamples=209)

Note: Cluster 2 consists of OSX_Bnodlero.PFL (2 samples), OSX_Bundlore.PFL (88 samples) OSX_Bundlore.PFM (109 samples), OSX_BundloreCA.PFL (1 samples), OSX_BundloreCA.PFM (4 samples), and OSX_SurfBuy.ESS2 (5 samples).

Figure 8 shows an example of two similar samples detected by the proposed model. The majority of the API call events remain identical (marked in white background). A sequence of API calls (marked in yellow background) appears, initiated by a successful network connection establishment in the sample on the right-hand side. Those API calls that are absent (marked in gray background) appear in the sample on the left-hand side. This demonstrates the robust detection capability of the model even when samples execute different code paths in different operational environments.

Figure 9 shows API call snippets of *OSX_Bundlore.ESS2* (on the left-hand side) and *OSX_Bundlore.PFL* (on the right-hand side), showing that *aae-mse* found these samples similar to each other. Although the URL and the IP address of the remote command and control server have changed, *aae-mse* successfully identified this variation.

The cluster detected by *aae-mse* shown in Figure 10 contains 209 different samples of six different variants of a family.

V. Conclusion

The proposed generative adversarial autoencoder using API call event as features can produce good latent representations with a small training set that can be used for distance-based similarity between samples. The training set restriction seriously tests the generalization capability of the model. The results show that the proposed model provides effective detection for gradually diverging mutation of malware species in terms of behavior. The model was also found to be effective in discovering multiple heterogeneous malware families that share similar dynamic execution events.

References

1. James Youngblood. (2016). “Ransomware.” In *Business Theft and Fraud, pp. 307–309*. Boca Raton, FL: CRC.

2. William Wright, David Schroh, Pascale Proulx, Alex Skaburskis, and Brian Cort. (2006). “The sandbox for analysis.” *Proceedings of the SIGCHI conference on Human Factors in computing systems - CHI 06*. Last accessed on 30 July 2019 at https://www.researchgate.net/publication/221515702_The_Sandbox_for_Analysis_-_Concepts_and_Methods.

3. Igor Santos, Yoseba K. Penya, Jaime Devesa, and Pablo Garcia Bringas. (2010). “N-Grams-Based File Signatures For Malware Detection.” *Proceedings of the 11th International Conference on Enterprise Information*. Last accessed on 30 July 2019 at https://www.researchgate.net/publication/220710220_N-grams-based_File_Signatures_for_Malware_Detection.

4. Youngjoon Ki, Eunjin Kim, and Huy Kang Kim. (2015). “A Novel Approach to Detect Malware Based on API Call Sequence Analysis.” *International Journal of Distributed Sensor Networks, vol. 11, no. 6, p. 659101*. Last accessed on 30 July 2019 at https://www.researchgate.net/publication/279155742_A_Novel_Approach_to_Detect_Malware_Based_on_API_Call_Sequence_Analysis.

5. Davide Canali, Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, Engin Kirda. (2012). “A quantitative study of accuracy in system call-based malware detection.” *Proceedings of the 2012 International Symposium on Software Testing and Analysis - ISSTA 2012*. Last accessed on 30 July 2019 at https://dl.acm.org/citation.cfm?id=2336768.

6. Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. (2012). “A survey on automated dynamic malware-analysis techniques and tools.” *ACM Computing Surveys, vol. 44, no. 2, pp. 1–42*. Last accessed on 30 July 2019 at https://sites.cs.ucsb.edu/~chris/research/doc/acmsurvey12_dynamic.pdf.

7. Ivan Firdausi, Charles Lim, Alva Erwin, and Anto Satriyo Nugroho. (2010). “Analysis of Machine learning Techniques Used in Behavior-Based Malware Detection.” *2010 Second International Conference on Advances in Computing, Control, and Telecommunication Technologies*. Last accessed on 30 July 2019 at http://asnugroho.net/papers/act2010-firdausi.pdf.

8. Brandon Amos, Hamilton Turner, and Jules White. (2013). “Applying machine learning classifiers to dynamic Android malware detection at scale.” *2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC)*. Last accessed on 30 July 2019 at http://bamos.github.io/data/papers/amos-iwcmc2013.pdf.

9. Parvez Faruki, Vijay Laxmi, M. S. Gaur, and P. Vinod. (2012). “Behavioural detection with API call-grams to identify malicious PE files.” *Proceedings of the First International Conference on Security of Internet of Things - SecurIT 12*. Last accessed on 30 July 2019 at https://dl.acm.org/citation.cfm?id=2490440.

10. Ronghua Tian, Rafiqul Islam, Lynn Batten, and Steve Versteeg. (2010). “Differentiating malware from cleanware using behavioural analysis.” *2010 5th International Conference on Malicious and Unwanted Software*. Last accessed on 30 July 2019 at https://ieeexplore.ieee.org/document/5665796.

11. Ping Wang and Yu-Zhih Wang. (2015). “Malware behavioural detection and vaccine development by using a support vector model classifier.” *Journal of Computer and System Sciences, vol. 81, no. 6, pp. 1012–1026*.

12. Gregoire Jacob, Hervé Debar, and Eric Filiol. (2008). “Behavioral detection of malware: from a survey towards an established taxonomy.” *Journal in Computer Virology, vol. 4, no. 3, pp. 251–266*.

13. Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. (2012). “DroidMat: Android Malware Detection through Manifest and API Calls Tracing.” *2012 Seventh Asia Joint Conference on Information Security*. Last accessed on 30 July 2019 at https://ieeexplore.ieee.org/abstract/document/6298136.

14. Faraz Ahmed, Haider Hameed, M. Zubair Shafiq, Muddassar Farooq. (2009). “Using spatio-temporal information in API calls with machine learning algorithms for malware detection.” *Proceedings of the 2nd ACM workshop on Security and artificial intelligence - AISec 09*.

15. Wenyi Huang and Jack W. Stokes. (2016). “MtNet: A Multi-Task Neural Network for Dynamic Malware Classification.” *Detection of Intrusions and Malware, and Vulnerability Assessment Lecture Notes in Computer Science, pp. 399–418*.

16. Bojan Kolosnjaji, Apostolis Zarras, George Webster, and Claudia Eckert. (2016). “Deep Learning for Classification of Malware System Call Sequences.” *AI 2016: Advances in Artificial Intelligence Lecture Notes in Computer Science, pp. 137–149*.

17. Sean Park, Iqbal Gondal, Joarder Kamruzzaman, and Jon Oliver. (2019). “Generative Malware Outbreak Detection.” *IEEE ICIT*. Last accessed on 30 July 2019 at https://documents.trendmicro.com/assets/white_papers/GenerativeMalwareOutbreakDetection.pdf?_ga=2.128148300.1365403863.1564474432-327212201.1564474432.

18. Alireza Makhzani, Jonathon Shlens, Navdeep Jaitly, Ian Goodfellow, and Brendan Frey. (2015). “Adversarial autoencoders.” *arXiv:1511.05644*. Last accessed on 30 July 2019 at https://arxiv.org/abs/1511.05644.

19. Alec Radford, Luke Metz, and Soumith Chintala. (2015). “Unsupervised representation learning with deep convolutional generative adversarial networks.” *arXiv:1511.06434*. Last accessed on 30 July 2019 at https://arxiv.org/abs/1511.06434.

20. Lars Mescheder, Sebastian Nowozin, and Andreas Geiger. (2017). “The numerics of gans.” *Advances in Neural Information Processing Systems, pp. 1825-1835*.

21. Diederik P. Kingma and Jimmy Ba. (2014). “Adam: A method for stochastic optimization.” *arXiv:1412.6980*. Last accessed on 30 July 2019 at https://arxiv.org/abs/1412.6980.

22. Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. (2014). “Generative adversarial nets.” *Advances in neural information processing systems, pp. 2672-2680*.

23. Sergey Ioffe and Christian Szegedy. (2015). “Batch normalization: Accelerating deep network training by reducing internal covariate shift.” *arXiv:1502.03167*. Last accessed on 30 July 2019 at https://arxiv.org/abs/1502.03167.

24. Yizhe Zhang, Dinghan Shen, Guoyin Wang, Zhe Gan, Ricardo Henao, and Lawrence Carin. (2017). “Deconvolutional paragraph representation learning.” *Advances in Neural Information Processing Systems, pp. 4169-4179, 2017*.

25. Igor Santos, Yoseba K. Penya, Jaime Devesa, and Pablo Garcia Bringas. (2009). “N-Grams-Based File Signatures For Malware Detection.” *Proceedings of the 11th International Conference on Enterprise Information*. Last accessed on 30 July 2019 at https://www.researchgate.net/publication/220710220_N-grams-based_File_Signatures_for_Malware_Detection.

26. Jonathan Masci, Ueli Meier, Dan Cireşan, and Jürgen Schmidhuber. (2011). “Stacked Convolutional Auto-Encoders for Hierarchical Feature Extraction.” *Lecture Notes in Computer Science Artificial Neural Networks and Machine Learning – ICANN 2011, pp. 52–59*. Last accessed on 30 July 2019 at http://people.idsia.ch/~ciresan/data/icann2011.pdf.

TREND MICRO™ RESEARCH

Trend Micro, a global leader in cybersecurity, helps to make the world safe for exchanging digital information.

Trend Micro Research is powered by experts who are passionate about discovering new threats, sharing key insights, and supporting efforts to stop cybercriminals. Our global team helps identify millions of threats daily, leads the industry in vulnerability disclosures, and publishes innovative research on new threats techniques. We continually work to anticipate new threats and deliver thought-provoking research.

www.trendmicro.com

TREND MICRO™ | research