ECE544NA Final Project: Malicious Code Execution Detection Using Bottleneck Stacked Autoencoder

Man Ki Yoon
Real-Time Systems Integration Lab., Dept. of Computer Science
Email: mkyoon@illinois.edu

Abstract—**In this project, Bottleneck Stacked Autoencoder, a deep learning method, is used to model the normal program (application) behavior and to detect abnormal behavior in terms of the system call usage. The method is evaluated by applying the proposed approach to an example application in which malicious code are embedded. The evaluation results show its effectiveness in terms of the detection accuracy and false positive rates, which are even comparable to Gaussian Mixture Model.**

I. Introduction

Real-time embedded systems are increasingly attracting attackers looking to compromise their safety and security. Protecting such systems from the attacks is challenging due to the ever-growing complexity of modern real-time embedded systems/applications; the additional complexity, in turn, exposes more security flaws [11]. Thus, instead of attempting to prevent every possible security breach, detection of intrusion, such as malicious code execution, by monitoring the application's *behavior* has drawn a great deal of attention due to its ability to detect novel attacks; an anomaly, *i.e.,* deviation, from the expected/normal behavior is considered malicious [3], [4]. Real-time embedded applications are a good fit for this type of security mechanism due to the regularity in their execution behavior; the set of what constitutes legitimate behavior is limited by design and also due to the (typically) small input set space.

Behavior-based intrusion detection systems (IDS) rely on specific behavioral *signals* such as network traffic [6], [12], control flow [1], system calls [5], [13], timing [14], *etc.* Among these, the system call has been of primary interest for use in behavior-based IDSes. The main reason is that many malicious activities involve accesses to system resources, such as file, I/O devices, *etc.*, and they use system calls to carry out privileged operations. Intrusion detection is performed based on the assumption that an attack would show anomalies in system call usage during its execution.

Thus, this project addresses an intrusion detection method for real-time embedded applications using system calls, specifically in the form of the *distribution of system call frequencies* [1]. It is a vector of non-negative integers, where each entry represents *the number of occurrences of a particular system call type* for an execution. Figure 1 shows an example distribution of system call frequencies. The underlying idea is that normal executions would follow an expected pattern of system call usage – this assumption holds for real-time systems that exhibit very little variations during execution. Another, related, assumption is that malicious activities will exhibit a different pattern of system call usage. For example, malware that leaks out a sensitive information would use some network-related system calls (*e.g.,* `socket`, `connect`, `write`, *etc.*) thus changing the frequencies of these calls. Hence, given a distribution obtained at runtime, a legitimacy test is carried out to check for the likelihood of the execution being a part of the expected executions. In order for an attacker to deceive such a test, he/she would need to not only know what the legitimate system call distribution looks like, but also be able to carry out the intended attack within the limited set of calls – both of these are very difficult to realize.

The application we monitor, however, may exhibit multiple *execution contexts* due to, say, different execution modes and/or inputs. In such cases, the system call frequency distributions (SCFDs) observed in different execution contexts could show considerable differences. For example, when uploading data to a server, the number of `read` and `write` calls used for larger data sets would be clearly different from the number of such calls for smaller data sets. Hence, representing such situations by a single behavioral context can lead to inaccuracies in the model(s) due to the smoothing out of irregularities. Thus, one can use, for example, k-means [10] or Gaussian Mixture Model (GMM) to find *distinct* execution contexts from a set of SCFDs. By using such methods, we can estimate the probability density of the SCFDs that represent the normal/legitimate behavior of the application

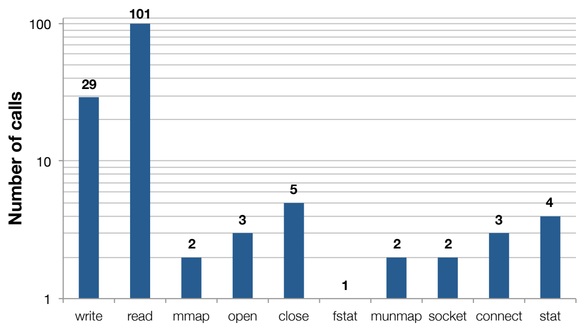


Fig. 1. A system call frequency distributions obtained from an example application. Each represents how many times it is called/used by the application during one execution.

[1]The term '*distribution*' here is nothing to do with *probability distribution*.

under monitoring.

In this project, instead, a neural network-based approach is investigated. To be more specific, the Bottleneck Stacked Autoencoder [2], [8] is used to learn the system call frequency distributions of normal, legitimate executions and then to perform an anomaly detection. It is a deep network consisting of multiple Restricted Boltzmann Machines (RBMs), stacked one by one vertically, and thus enables us to learn high-order features of the inputs. As will be detailed in the next section, it can be used to learn the clusters of (normal) SCFDs by properly configuring the hidden layers. The experimental results of the proposed approach, based on an example application and various attack scenarios, show that the Bottleneck Stacked Autoencoder can effectively detect abnormal execution behavior.

A. Assumptions

The following assumptions are made without loss of generality: *(i)* The threat model is any malicious activities that use a collection of system calls. If it does not use any (e.g., tainting a data on memory), the activity at least has to affect executions afterward so that the future system call distribution may change. The proposed detection method cannot detect attacks that never alter system call usage. *(ii)* A malicious code can be secretly embedded in the application, either by remote attacks or during upgrades. The malicious code activates itself at some point after the system initialization. *(iii)* Every distinctive execution contexts are present when being profiled. This can be justified by the fact that most real-time embedded applications have a limited set of execution modes and input data are within fairly narrow ranges.

II. Malicious Code Execution Detection Using Bottleneck Stacked Autoencoder

This section explains how to model, train, and use Bottleneck Stacked Autoencoder for the detection of abnormal, malicious system call frequency distribution of the application under monitoring.

A. Definitions

Let $\mathcal{S} = \{s_1, s_2, \ldots, s_D\}$ be the set of all system calls provided by an operating system, where s_d represents the system call of type d.[2] During the n^{th} execution of an application, it calls a multiset σ^n of $\mathcal{S}$. Let us denote the n^{th} *system call frequency distribution* (or just *system call distribution*) as $\vec{x}^{(n)} = [m(\sigma^n, s_1), m(\sigma^n, s_2), \ldots, m(\sigma^n, s_D)]^T$, where $m(\sigma^n, s_d)$ is the multiplicity of the system call of type d in σ^n. Hereafter, $m(\sigma^n, s_d)$ is simplified as x_d^n. Thus,

$$\vec{x}^{(n)} = [x_1^n, x_2^n, \ldots, x_D^n]^T. \quad (1)$$

[2]The number of system call types, *i.e.,* D, is quite large in general. For example, in Linux 3.2 for x64, there are 312 system call types. However, an application normally uses a small set of system calls. Furthermore, the dimensionality can be significantly reduced by ignoring system call types that never vary.

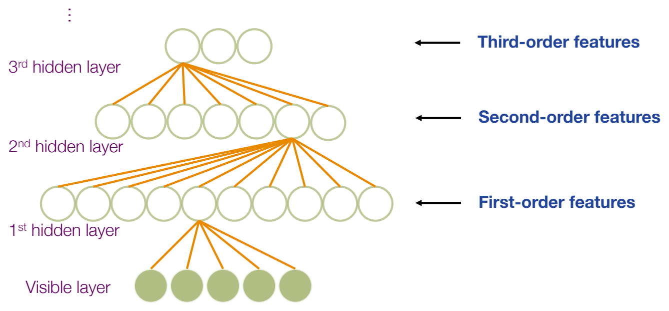


Fig. 2. Bottleneck Stacked Autoencoder.

Next, we define a *training set*, *i.e.,* the execution profiles of a sanitized system, as a set of N system call frequency distributions collected from N executions, and is denoted by

$$\mathbf{X} = [\vec{x}^{(1)}, \vec{x}^{(2)}, \ldots, \vec{x}^{(N)}]^T. \quad (2)$$

Note that in our anomaly detection problem, the training set purely consists of normal data, and thus the learning problem is semi-supervised.

B. Bottleneck Stacked Autoencoder with Two Hidden Layers

As briefly explained in the previous section, a stacked autoencoder enables learning of high-order feature representation of data (inputs) by stacking several RBMs (see Figure 2); the first hidden layer learns the first-order features in the raw input, and then the second layer learns the second-order features by grouping some of the first-order features, and so on. Thus, the multiple hidden layers makes it possible to learn some hierarchical grouping of the input.

This hierarchical grouping fits well to the anomaly detection problem considered in this project. That is, in general, an application shows widely varying system call distributions due to multiple execution modes and a wide range of possible inputs. In such scenarios, finding a single multivariate Gaussian distribution (*i.e.,* a single cluster/centroid) for the whole set can result in inaccurate models because it would include even many non-legitimate points that belong to none of the execution contexts. Thus, it is more desirable to consider that observations are generated from a set of *distinct* distributions, each of which corresponds to one or more execution contexts, and each execution makes a small variation from each of the distribution it is generated from. This is a valid model for real-time embedded systems since the code in such system tends to be fairly limited in what it can do – hence such analyses is quite powerful in detecting variations and hence, catching intrusions.

In the problem being considered here, the network is modeled with two hidden layers (*i.e.,* two RBMs) as shown in Figure 3; this is sufficient to learn the clusters of normal system call frequency distributions, in which, again, each data point (input token) is a vector of integers. More specifically, the first hidden layer, $\vec{h}_1$, consists of Gaussian units, and the second hidden layer, $\vec{h}_2$, consists of binary units. With this model, each of the first hidden layer unit corresponds to a

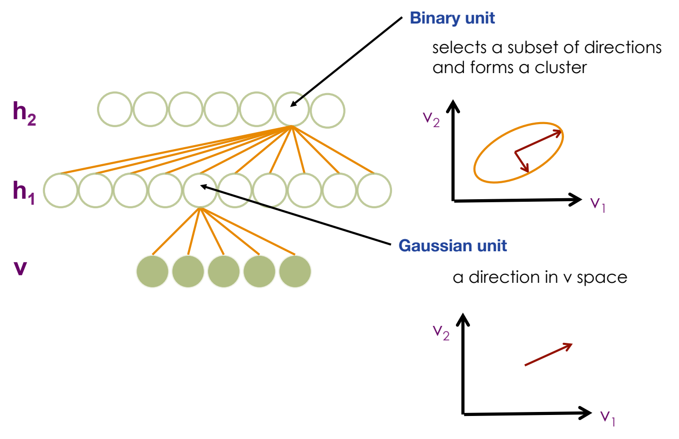


Fig. 3. Bottleneck Stacked Autoencoder with Gaussian-Binary hidden layers.

direction in the v-space, *i.e.,* the input space, and then each of the second hidden layer units selects some of the directions to form a particular cluster.

The training of the network is done layer-wise; we first learn the first RBM with the training set $\mathbf{X}$ as the visible nodes and then use the hidden layer of the first RBM as the input (visible) nodes of the second RBM. Here, the first RBM, in which both visible and hidden units are Gaussian, is trained by the stochastic Contrastive Divergence (CD) algorithm [7], [9] as covered in the class. The learning of the second RBM, in which the hidden units are binary, is also done by the CD algorithm but is slightly different from the case of Gaussian hidden units. First of all, since we have trained the first RBM, $\hat{h}_1^{(n)}$ for $n = 1, \ldots, N$ can be calculated by the weights, $\mathbf{w}_1$, and the biases, $\vec{c}_1$ and $\vec{b}_1$ (see Figure 4). Then, each $h_{2,k}^{(n)}$ for the n^{th} training token is activated (that is, having 1) with the probability

$$Pr\left(h_{2,k}^{(n)} = 1 \middle| \hat{h}_1^{(n)}\right) = \sigma\left(\vec{w}_{2,k}\hat{h}_1^{(n)} + c_{2,k}\right), \tag{3}$$

where $\sigma(a) = 1/(1+\exp(-a))$ is the sigmoid function. Once we have sampled $\hat{h}_2^{(n)}$, we then reconstruct the first hidden layer by

$$\tilde{h}_1^{(n)} = E\left[\hat{h}_1^{(n)} | \mathbf{w}_2, \vec{b}_2\right] = \mathbf{w}_2^T \hat{h}_2^{(n)} + \vec{b}_2.$$

Then, the weights and biases are updated as follows (similar to the case of Gaussian-Gaussian units of the first RBM):

$$\mathbf{w}_2 = \mathbf{w}_2 + \eta\left(\hat{h}_1^{(n)^T}\hat{h}_2^{(n)} - \tilde{h}_1^{(n)^T} Pr\left(\vec{h}_2^{(n)} = 1 \middle| \tilde{h}_1^{(n)}\right)\right) \tag{4}$$

$$\vec{b}_2 = \vec{b}_2 + \eta\left(\hat{h}_1^{(n)} - \tilde{h}_1^{(n)}\right) \tag{5}$$

$$\vec{c}_2 = \vec{c}_2 + \eta\left(\hat{h}_2^{(n)} - Pr\left(\vec{h}_2^{(n)} = 1 \middle| \tilde{h}_1^{(n)}\right)\right), \tag{6}$$

where η is the learning rate, and $Pr\left(\vec{h}_2^{(n)} = 1 \middle| \tilde{h}_1^{(n)}\right)$ is a vector of $Pr\left(h_{2,k}^{(n)} = 1 \middle| \tilde{h}_1^{(n)}\right)$ for $k = 1, \ldots, K$, which can be calculated by (3) (Notice $\tilde{h}_1^{(n)}$, not $\hat{h}_1^{(n)}$). The network is trained with these update rules, iterating over the entire training set for multiple times (*e.g.,* 100 times).

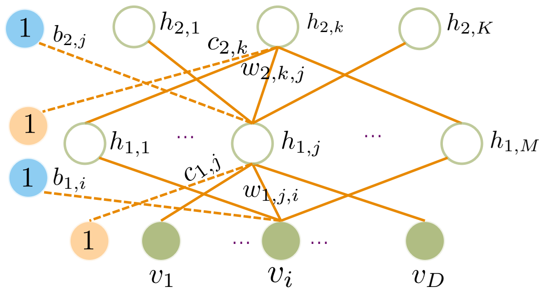


Fig. 4. Notations of visible and hidden nodes, weights, and biases in bottleneck stacked autoencoder with two RBMs.

Now, suppose we have learned the weights $(\mathbf{w}_1, \mathbf{w}_2)$ and biases $(\vec{b}_1, \vec{b}_2, \vec{c}_1, \vec{c}_2)$ using the training set. Let us denote the parameters as π. Then, given a test token $\vec{x}$, that is, a system call frequency distribution observed from the application under monitoring in the run-time, its legitimacy test (whether or not it is abnormal given the training set) is done by calculating $Pr(\vec{v}|\pi)$ where $\vec{v} = \vec{x}$. Intuitively, it should be high enough for the test tokens that are similar (or close) to what we have seen during the training (provided that the training was properly done). In other words, we should be able to reconstruct the given token using the parameters π. To find $Pr(\vec{v}|\pi)$, we first feed forward the given test token $\vec{x}$ into the network by setting $\vec{v} = \vec{x}$. Then, the first hidden layer units become having

$$\hat{h}_1 = E\left[\vec{h}_1 | \pi, \vec{v}\right] = \mathbf{w}_1\vec{v} + \vec{c}_1 \tag{7}$$

since the units are Gaussian. Similarly,

$$\hat{h}_2 = E\left[\vec{h}_2 | \pi, \hat{h}_1\right] = \sigma\left(\mathbf{w}_2\hat{h}_1 + \vec{c}_2\right) \tag{8}$$

since the second hidden layer's units are binary. Then, we try to reconstruct the first hidden layer and then the visible layer. First,

$$\tilde{h}_1 = E\left[\vec{h}_1 | \pi, \hat{h}_2\right] = \mathbf{w}_2^T\hat{h}_2 + \vec{b}_2 \tag{9}$$

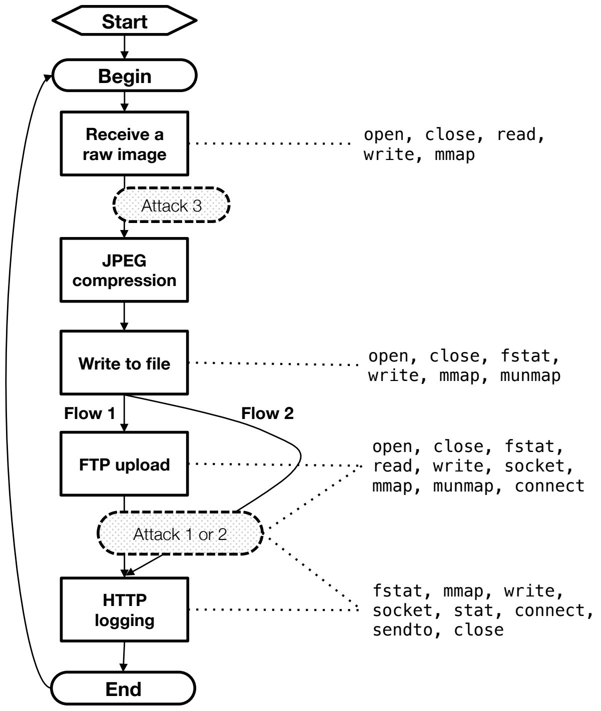
Then,

$$\tilde{v} = E\left[\vec{v} | \pi, \tilde{h}_1\right] = \mathbf{w}_1^T\tilde{h}_1 + \vec{b}_1 \tag{10}$$

As mentioned above, the input token $\vec{v} = \vec{x}$ should be close to the $\tilde{v}$, reconstructed from the hidden nodes by using the parameters π learned from the training set, if $\vec{x}$ is normal. Since the visible units are Gaussian, $Pr(\vec{v}|\pi) \approx Pr(\vec{v}|\pi, \tilde{h}_1)$ can be defined by a Gaussian distribution centered at $\tilde{v}$. That is,

$$Pr(\vec{v}|\pi) \approx \mathcal{N}\left(\vec{v}; \mu = \tilde{v}, \Sigma = I\right), \tag{11}$$

where $\Sigma = I$ is due to the conditional independence between the visible nodes given $\tilde{h}_1$. This can represent how well the input token $\vec{v} = \vec{x}$ is reconstructed by the hidden units. Finally, the legitimacy test of a given test token can be performed by checking if (11) is lower than a pre-defined threshold, θ. If is is, we consider that the application behavior corresponding to $\vec{x}$ is malicious.



Start
Begin
Receive a raw image
open, close, read, write, mmap
Attack 3
JPEG compression
Write to file
open, close, fstat, write, mmap, munmap
Flow 1
Flow 2
FTP upload
open, close, fstat, read, write, socket, mmap, munmap, connect
Attack 1 or 2
HTTP logging
fstat, mmap, write, socket, stat, connect, sendto, close
End

Fig. 5. The execution flow of the target application and the system call types used at each stage.

III. Evaluation

In this section, the intrusion detection method described in the previous section is evaluated with an example application.

A. Target Application Model

I implemented a target application that works as follows (Figure 5). The application runs periodically, and during each execution instance, it *(a)* captures a raw image from a webcam, *(b)* compresses it to a JPEG file, *(c)* uploads the file to an FTP server and finally *(d)* writes a log via a HTTP post. The distributions of the system call frequencies exhibited by the application is mainly affected by the stages after the JPEG compression. While the amount of memory required by a raw image is always fixed (*e.g.,* 2.6 MB for 1280 x 720 resolution), a JPEG image size can vary (27 KB – 97 KB) because of compression. This results in a variance in the number of `read` and `write` system calls. In order to include further variations in the use of system calls, a branch before the FTP upload stage is added. The system could randomly skip the FTP upload stage with a probability of 0.5. This affects the number of occurrences of some network and file-related system calls during actual execution. Hence, the application has two legitimate flows that are denoted by "Flow 1" and "Flow 2" as depicted in Figure 5.

B. Attack Examples

The following attack scenarios are introduced into the application:

1) Attack 1: This attack code uploads the JPEG image that the application has just compressed from the received raw image to a separate FTP server. This attack uses the same functions used by the legitimate FTP uploads and requires some file and network-related system calls.
2) Attack 2: In this scenario, the attack code steals user authentication information that is used to connect to the FTP server and posts it to a separate HTTP server. This attack invokes the same HTTP logging calls used by the legitimate execution instance (network-related syscalls).
3) Attack 3: This attack modifies the array that contains the raw image received from the webcam. The attack erases the array by calling `memset`. This attack does not require any system calls.

Attack 1 and Attack 2 are inserted on the both flows as marked in Figure 5. Attack 3 is executed before the JPEG compression stage. Note, again, that Flow 2 is launched with a probability of 0.5. If enabled, the attack code executes at the marked place.

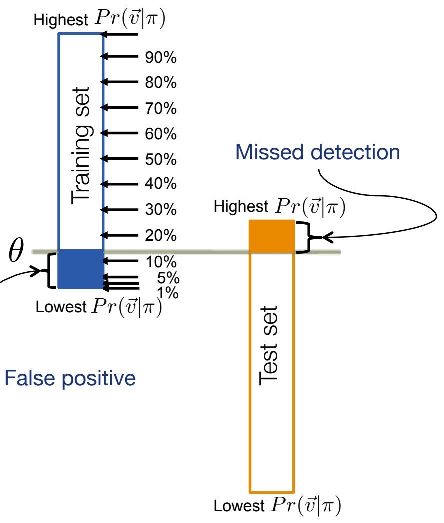


Fig. 6. Threshold, false positive, and missed detection.

C. Evaluation

To obtain the training set, the application was executed 2000 times without any attack code activation. The target application used 20 types of system calls. Among them 15 types had non-zero variance in the training set (Figure 5 shows some of the types). Thus, the feature set is the 15-dimension system call frequency distributions. Then, the training set (and the test sets) is z-normalized using the means and the standard deviations of the training set. Three test sets (300 tokens each) were obtained by enabling each of the three attack types. Thus, each test set purely consists of abnormal tokens. The evaluation in the following is done by testing how many of the abnormal tokens could be detected. Each of the results is the average of 20 runs (because of the randomness in the Contrastive Divergence algorithm).

The bottleneck stacked autoencoder with two hidden layers is trained as explained in the previous section. Then, the probability of each training token was calculated by (11). This is to find different settings of the threshold θ. That is, by

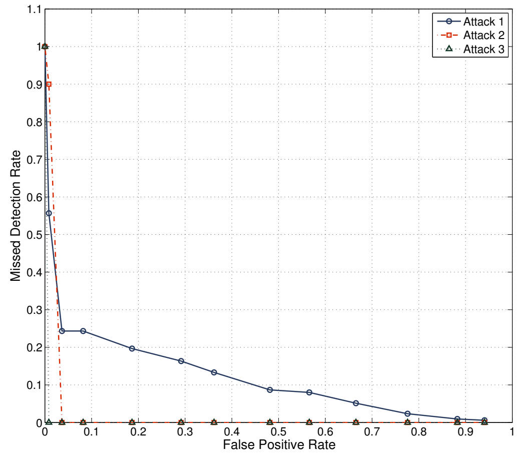


Fig. 7. Detection Error Tradeoff (DET) graph for 50 and 5 units in the first and the second layers, respectively.

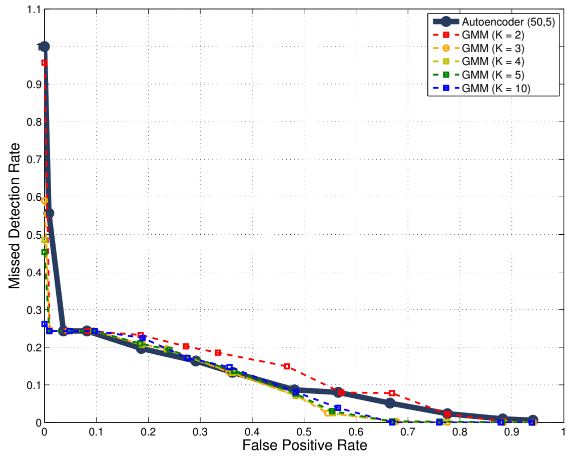


Fig. 8. Bottleneck Stacked Autoencoder vs. Gaussian Mixture Model.

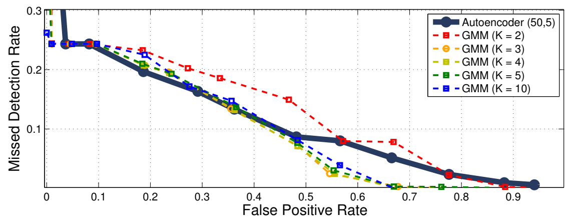


Fig. 9. Magnified view of Figure 8.

setting $\theta = k^{th}$ percentile of $\{Pr(\vec{v}^{(n)}|\pi)|n = 1,\ldots,N\}$, we can control the false positive rate and then calculate the missed detection rate according to the chosen false positive rate. As shown in Figure 6, total 13 different threshold values were used. This includes the minimum, the maximum, and k^{th} percentile for $k = 1, 5, 10, 20, \ldots, 90$ of $Pr(\vec{v}^{(n)}|\pi)$ in the training set. Intuitively, as the threshold increases, the false positive rate also increases while the missed detection rate decreases. Ideally, the parameter set π should be trained properly so that the probabilities of the training set (the left box in the figure, which purely consists of normal data) and those of the test set (the right box, which purely consists of abnormal data) can be separated by a proper value of θ as clearly as possible.

Figure 7 shows an example result for the three attack scenarios tested by a network of 50 and 5 hidden units in the first and the second hidden layers, respectively. The plot is the Detection Error Tradeoff (DET) graph, which shows how well we could detect the abnormal behavior for a fixed false positive rate or vice versa. Each marker corresponds to each threshold setting. The threshold increases left to right. A good anomaly detector makes DET curves to be as close as possible to the origin (no false alarm and no missed detection). As can be seen from the result, our bottleneck stacked autoencoder method detected most of the attacks made by the scenario 2 and 3 without compromising the false positive rate. This was because the changes in system call frequency distribution due to their malicious activities were visible enough:

- Attack 2 (HTTP post): Because of the additional HTTP request by the attack code, `socket`, `connect`, `close` and `stat` were called more in both Flows 1 and 2; this resulted in very low probability of the test tokens, thus flagging malicious activities.

- Attack 3 (Image impairment): This attack does not use any system calls; it just changes the values in the data. However, this affected executions that follow, especially ones that depends on the data; the raw image and the JPEG compression. The `memset` filled the array with 0's and resulted in 15 KB of black image. Such image sizes were not typical during normal executions. Hence, calls to `read` and `write` were much less frequent as compared to the normal executions (where these calls were used often to write the larger images to files or upload on FTP servers).

However, the performance of our detection method for attack type 1 was low compared to the two attack scenarios. The attack executed on Flow 1 was easily caught because it changes some network related system calls (as in the HTTP post attack), which is enough to make the distribution of the system call frequencies to fall outside the legitimate regions. On the other hand, if the attack is launched on Flow 2, it was not easy to catch it since the attack uses the same functions that are invoked by legitimate code. Thus, it looks like the application is following Flow 1 where the FTP upload is actually legitimate. This resulted in some missed detections. Nevertheless, in terms of the absolute performance, the proposed approach seems effective in detecting malicious codes; as an example, it detected around 75% of the attack instances (of type 1) while making around 5% of false alarms. In the rest of the evaluation, only the result of attack 1 will be used since the other two's result do not look interesting.

Next, the proposed approach is compared with the Gaussian Mixture Model, which is one of the common methods for unsupervised/semi-supervised anomaly detection. The GMM

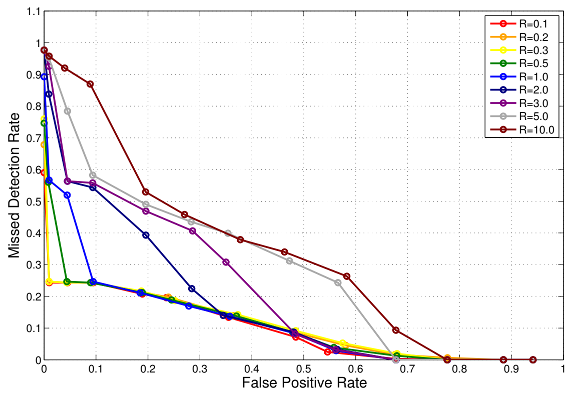


Fig. 10. GMM ($K = 3$) with different regularization numbers.

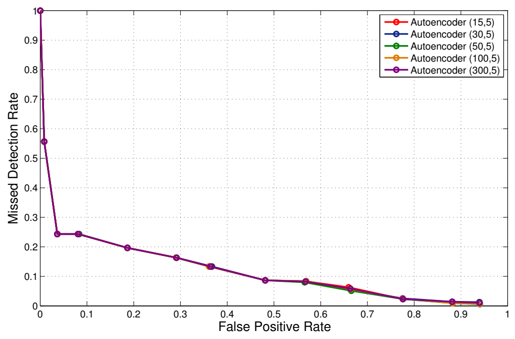


Fig. 11. First layer: [15,30,50,100,300] units, Second layer: 5 units.

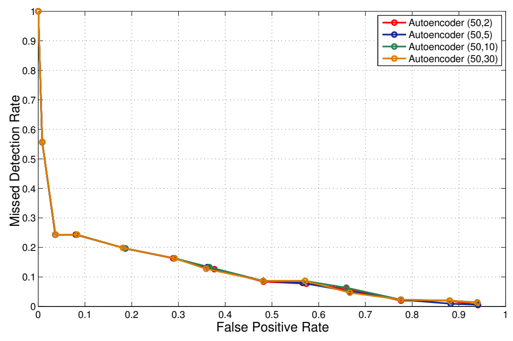


Fig. 12. First layer: 50 units, Second layer: [2,5,10,30] units.

training with the given training set had a problem of singularity of the covariance matrix during the EM process, thus a regularization number $r = 0.1$ was used to add rI to the covariances matrices. Figure 8 shows the results of GMM with different number of components, $k = 2, 3, 4, 5, 10$ and that of the proposed method. As can be seen from the DET curves of GMM, the performance do not change significantly with varying number of components (except for $K = 2$). Apparently, 2 components could not capture well the clusters of normal system call frequency distributions. The result indicates that 3 components are enough to capture the distinctive execution contexts and adding more component do not necessarily improve the accuracy in this particular problem. Now, if we compare the stacked autoencoder approach with GMMs (magnified in Figure 9), we can see that their performances do not significantly differ: for the low false positive rates (≤ 0.3), it is even slightly better than GMM. This similarity between our autoencoder model and the GMM comes from the former's hidden layer structure – Gaussian units in the first hidden layer and binary units in the second hidden layer - as explained in the previous section. As a side result (Figure 10), the GMM performed better with smaller regularization number as expected. With r smaller than 0.1, however, the singularity of the covariance matrices could not be resolved.

Now, one interesting question could be how the number of hidden units would affect the performance. Intuitively, we should expect to see a better performance with more hidden units because of the increased expressiveness. However, this was not the case in the considered problem as Figure 11 and 12 show. There were no significant improvement with more units in the first layer or second layer. This is possibly because our problem is already simple enough and thus could be modeled well with a small number of hidden units. The impact of hidden unit count could be more visible if the method was applied to a more complex problem.

Lastly, Figure 13 shows the results of the stacked autoencoder with *binary* units in the first hidden layer.[3] If we compare it with Figure 11, the original model (Gaussian-Binary hidden layers) subtantially outperforms this binary-binary model. This is because the Gaussian hidden layer is much more expressive than the binary layer. The binary-binary hidden layers cannot capture/allow valid noise in the input and thus often determine even some valid variations as abnormal. This, as the DET curves show, results in very poor performance. In some cases, it was even worse than random guess ($y = x$ line).

[3]The training of the first RBM in this model is now same with that of the second RBM in the original model.

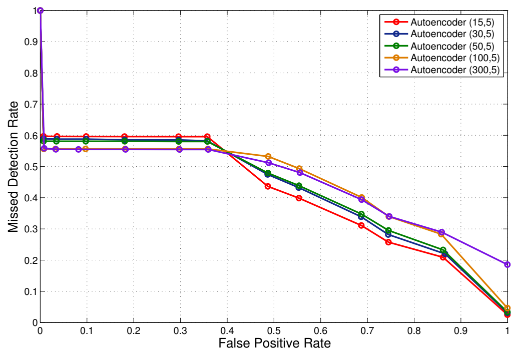


Fig. 13. The result of binary-binary hidden layers. First layer: [15,30,50,100,300] units, Second layer: 5 units.

IV. Conclusion

In this project, the bottleneck stacked autoencoder, a deep learning method, was applied to an anomaly (malicious code execution) detection problem. The evaluation results based on an example application showed its effectiveness in terms of the accuracy which is comparable to Gaussian Mixture Model. An interesting direction would be an application of the proposed approach to other types of program behaviors, for example, memory usage pattern,[4] which would be much more high-dimensional and complex enough to show the impacts of various configurations of the network on the performance.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):4:1–4:40, Nov. 2009.

[2] Y. Bengio, P. Lamblin, D. Popovici, H. Larochelle, U. D. Montral, and M. Qubec. Greedy layer-wise training of deep networks. In *Advances in Neural Information Processing Systems*, pages 153–160, 2007.

[3] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, July 2009.

[4] D. E. Denning. An intrusion-detection model. *IEEE Trans. Softw. Eng.*, 13(2):222–232, Feb. 1987.

[5] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 120–128, 1996.

[6] M. Handley, V. Paxson, and C. Kreibich. Network intrusion detection: evasion, traffic normalization, and end-to-end protocol semantics. In *Proceedings of the 10th conference on USENIX Security Symposium - Volume 10*, 2001.

[7] G. Hinton. *A Practical Guide to Training Restricted Boltzmann Machines*. 2012.

[8] G. Hinton and R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(45786):504–507, 2006.

[9] G. E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Comput.*, 14(8):1771–1800, Aug. 2002.

[10] S. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, Mar. 1982.

[11] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady. Security in embedded systems: Design challenges. *ACM Trans. Embed. Comput. Syst.*, 3(3):461–491, Aug. 2004.

[12] R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 305–316, 2010.

[13] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusion using system calls: alternative data models. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1999.

[14] M.-K. Yoon, S. Mohan, J. Choi, J.-E. Kim, and L. Sha. SecureCore: A multicore-based intrusion detection architecture for real-time embedded systems. In *IEEE Real-Time Embedded Technology and Applications Symposium*, pages 21–31, 2013.

[4] A snapshot of the memory usage would look like a heat map whose size can be easily over several hundreds of thousand real values.