International Journal for Research in Engineering Application & Management (IJREAM)
ISSN : 2454-9150 Vol-05, Issue-09, Dec 2019

Study of Frame Work for Real Time Network Feature Construction Module, Architecture and Implementation

[1]Nabonarayan Jha, [2]Anand Singh,

[1]Lecturer, Department of Mathematics, Patan Multiple Campus, Patan Dhoka, Lalitpur, Tribhuvan University, Kathmandu, Nepal.

Department of Computer Engineering, Kantipur City College, Kathamndu, Nepal.

[1]nabonarayan123@gmail.com, [2]profdranand@gmail.com

Abstract: This paper presents the technical details of the system implementation. The whole system is implemented by using mixture of C++, Java and MATLAB language as we show fit for certain tasks. For instance, we used C++ under Linux to implement the feature extraction process, since this tasks is time critical and is designed to work on-line and in real-time. Conversely, some parts of the feature evaluation process were implemented in MATLAB and Java under Windows since those tasks were not time critical ones and also because MATLAB and Java languages offer a high number of libraries that facilitate the implementation process.

Keywords-- MATLAB, JAVA, ICMP, **TCP, DDoS**

I. INTRODUCTION

The overall feature extraction and evaluation process is depicted in Figure 1. As seen in the figure, there are three types of input data that the system requires as follows: the tuning combinations, the actual data, and the labels of the data. The system is design to read from already saved TCP dump files, and can be easily extended to sniff packets directly from the network. In order to analyze the data, the system requires it to be labeled. In the best case scenario this can be provided upfront through a labeling file, or can be approximated at the runtime. Thus, the system is designed to be independent of the data labels, which makes the third input of the system optional. The first processing unit that the data goes through is the *Framework for Real Time Network Feature Construction* module.

This module is design to be a highly customizable, stand-alone application that based on the feature classification schema, produces a total of 671 feature for each individual packet that it. Once the features are produced they are sent to the *Statistical Profiler Module,* where profile creation, protocol and attack filtering happens. Since for attack filtering the system needs to know the exact label of the data, in parallel with the feature extraction process, there is a packet labeling process which consists of a *Detection Module,* and an *Alert Aggregator* that works together for attack label extraction. These two modules will be bypassed in the event when the labels are already provided through an external labeling file.

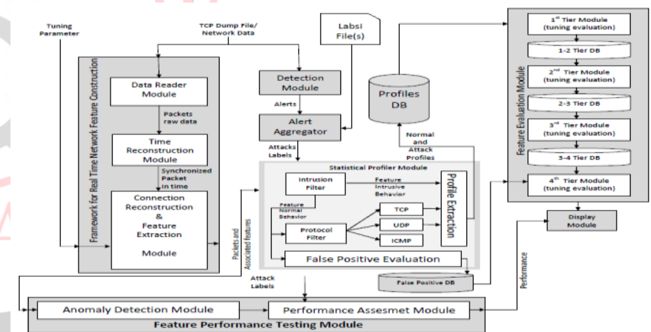

Figure 1: The overall view of the implementation block diagram.

The profile produced by the *Statistical Profiler Module* are temporarily saved in a custom-made database, which allows the next processing level to use them once all the profiles are computed. Furthermore, a second functionally of the *Statistical Profiler Module* is to estimate the potential number of false positives that each feature may produce under certain tunings. Once this task is done, the results are saved in the *False Positives DB.* The last processing step is performed by the *Feature Evaluation Module.* This module implements the core of our proposed evaluation. It has as input both the feature profiles as well as the feature false positive estimation, and it produces the final results.

In order to verify the experimental results, the packets and the associated features that are produced by the *Connection Reconstruction & Feature Extraction Module* are also sent to a *Feature Performance Testing Module* that empirically

 DOI : 10.35291/2454-9150.2019.0535 © 2019, IJREAM All Rights Reserved.

evaluates the performance of each individual feature and sends the results to the Display Module.

II. Detection Module

For labeling purposes we use an off-the-shelf product for the intrusion detection when no labels are provides for the given dataset. For the purpose of this work we choose to work with Snort, an open source intrusion detection system. Snort system uses a series of preprocessor modules and also rules that are customized to identify certain types of attacks. For instance, for scanning attacks, we use the sports can preprocessor, while for DoS and DDoS attacks we use the DoS and DDoS rules from Snort. We acknowledge that due to the nature of network data, there isn't any ideal solution that will detect all types of attacks with zero false positive and misclassified instances. That is why the final alerts that an IDS produces may not be correct all the time. However, despite these drawbacks this labeling procedure remains the sole option for unlabeled datasets. The output of Snort is saved in a comma separated file. The file contains for each individual alert the following information: alert time-stamp, message, protocol, source and destination IP, and source and destination ports. The output is dumped into a comma separated file by using the following line in the snorstconf file: output alert csv: alert.csv timestamp, msg, proto, src, srcport, dst, dstport.

III. Alert Aggregator

The aim of the alert aggregator module is to reduce the number of alerts generated by the *Detection Module*. It is designed to implement the following tasks:

- Filter the intrusions that are not relevant by removing those intrusions that are out of the scope of this study. For instance, intrusions that are on different protocols than the ones considered in our experimental results (i.e., TCP, UDP, and ICMP)
- Merge multiple alerts that share the same attributes (e.g., attackers, victim, type of intrusion) into a single intrusion. For instance, merging the same alert that appears at two or more distinct times.
- Merge multiple alerts that share the same attacker and multiple victims into a single alert. Examples of such scenarios can be horizontal scanning, DDoS attacks that can be farther reported as a single alert.

The implementation of this module is done using Java, and it consists of a single class that parses the output of the output of the detection engine and returns a more compact set of alerts that later on are used by *Statistical Profile Module*. Note that both, the *Detection Modula* and the *Alert Aggregator Module* are bypassed in the case when the labels of the dataset are provided.

IV. Statistical Profiler Module

The next processing step is the attack and normal profile generation. The task is accomplished by the *Statistical Profiler Module*. Figure 2 depicts the underlying block diagram of this module. This module uses the provided attack labels to filter out the intrusive and normal behavior of each individual feature, and stores the corresponding statistical data into uniquely identifiable profiles. Each profile keeps track of the mean μ and standard deviation σ statistics of a particular feature during the normal or intrusive stages. It is known that the features tend to have different values for different protocols. For instance, the size of the ICMP packets is expected to be smaller than the size of TCP packets. Thus, instead of creating a single profile for the normal behavior of a feature, our system creates individual normal profiles for each protocol that applies to the current feature.

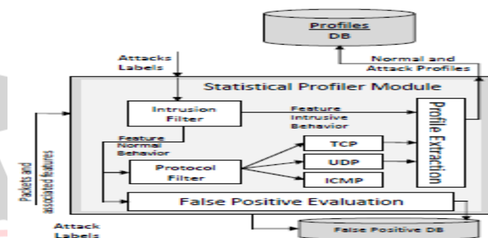

Figure 2: The overall view of the *Statistical Profiler Module* block diagram.

Let f_i represents the ith feature to be analyzed $\xi_j$denote the jth encountered intrusion, and $\tau_k$represent the kth tuning factor selected. Consequently, two types of profiles are defined as follows:

- Normal profile: A normal profile is uniquely identified by$< f_i, \tau_k >$tuple, and characterizes a particular featuref_iExtracted using $\tau_k$tuning during the normal network operation. The profile keeps track of the $\mu(f_i, \tau_k)$ and $\sigma(f_i, \tau_k)$ Statistical representing the normal mean and standard deviation during the previously specified scenario.
- Intrusive profile: An intrusive profile is uniquely identified by $< f_i, \xi_j; \tau_k >$tuple, and characterizes a particular feature$f_i$extracted using $\tau_k$tuning while under the $\xi_j$attack. Similarly, the profile keeps track of the $\mu(f_i, \xi_j; \tau_k)$ and $\sigma(f_i, \xi_j; \tau_k)$ Statistics representing the mean and standard deviation of the intrusive behavior.

All these statistical profiles are stored in the *DB Profiles* database, and are constantly updated. Note that the $\tau_k$factor depends on the type of the current featuref_i.

The module also evaluates the false positives that each features produces. The *False Positive Evaluation* sub module is responsible for this task, and the detail algorithms that it implements are described. Once the evaluation is done, the false positive predictions $(\text{i.e.}, FP(f_i))$are saved

 DOI : 10.35291/2454-9150.2019.0535 © 2019, IJREAM All Rights Reserved.

ISSN : 2454-9150 Vol-05, Issue-09, Dec 2019

into *False PositiveDB*. The database keeps for each feature $f_i$the corresponding $FP(f_i)$value. The process of extracting profiles and false positive prediction is repeated once for each TCP dump file and tuning combinations, until all the possible combinations are exhausted.

V. Feature Performance Testing Module

The purpose of this module is to empirically evaluate the performance of each individual feature in the detection process given a set of attacks and corresponding tunings. The module is depicted in Figure 3 and implements two functions. The first one is an anomaly detection module that mines the data for possible instructions, and the second function performs the assessment on the alerts that the anomaly detection module produces. Once that is done, it reports the final performance to the Display Module. We choose to work with a very simplistic threshold-based anomaly detection algorithm that uses a lower and upper control limit to define the boundaries of the normal values. The two thresholds are computed during the training phase and are set on both sides of the normal population mean at five types of equations. During the detection phase, an anomaly is signaled for each individual point that exceeds the two boundaries.

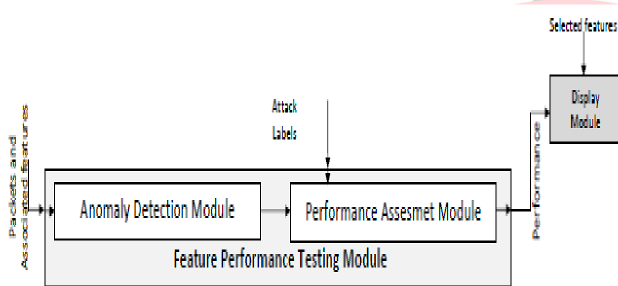

Figure 3: The overall view of the block diagrams for the *Feature Performance Testing Module* and *Display Module*.

For this purpose, the data from each dataset has been divided into 2 parts, one for training and one for testing. The training part consists of 80% of the normal data whereas the testing part consists of the rest 20% plus all the intrusions in the datasets. The detection results are computed for each individual tuning value and feature. The average of those individual runs is further reported. For statistical significance, the best and worst cases are excluded. The *Performance Assessment Module* receives the generated alerts from the *Detection Module* and compares them with the true attack labels that it has access to. The module computes four main evaluation functions as follows:

- **The number of detected Intrusions:** This value represents the number of actual attacks detected by the current feature during the testing phase.
- **The number of misclassified Intrusions:** This value represents the number of attacks that the current feature misclassifies.
- **True positive rate:** This value represents the percentage of correctly classified intrusions over the total number of intrusions that the current feature produces while in the testing phase.
- **False positive rate:** This value represents the percentage of normal data incorrectly classified as intrusion by the current feature while in the testing phase.

VI. Software Testing

The numerous building blocks of the system combined with the massive amount of data that it processes forced us to empirically test end evaluate the performance and reliability of each individual building block, as well as of the whole system. The whole system (as well as all its building blocks) was thoroughly checked for programming mistakes and software failure. To accomplish this goal, we use static and dynamic testing. Static testing is the forces of inspecting the source code without actually running it; whereas, dynamic testing involves running the code with a given set of test cases. The dynamic testing was done using *unit testing* methodology, where a set of unit tests were applied to all the subcomponents of each module. Finally, *black box testing* methodology was used to check the functionality of each main building block against its requirements. For instance, in the case of *Connection Reconstruction & Feature Extraction* module we used a small testing dataset consisting of several connections. We computed the expected result for each of the studied features and then we compared that against the output of the program.

VII. Conclusion

In this paper we presented the technical details of the implementation. In particular, the feature construction process was in detail explained, and also the feature evaluation process. The logistics around data labeling and empirical evaluations were outlined as well. The implementation is done by the use of three languages (i.e., C++, Java, and MATLAB) that complement each other through the functionalities that they provide. For instance, for an efficient memory usage we use C++ language under Linux to implement the feature extraction process. Conversely, whenever speed of memory was not an issue, for convenience and also libraries that are available we used Java MATLAB under Windows.

References

[1] A. Blum and R.L. Rivest, Training a 3-node neural network is np-complete, COLT '88: Proceedings of the first annual workshop on Computational learning theory (San Francisco, CA, USA), Morgan Kaufmann Publishers Inc., 1988, pp. 9{18.

 DOI : 10.35291/2454-9150.2019.0535 © 2019, IJREAM All Rights Reserved.

[10] A.L. Blum and P. Langley, Selection of relevant features and examples in machine learning, Artificial Intelligence, 97(1997), no. 1, 245{271.

[11] DARPA, Darpa intrusion detection and evaluation dataset 1999, http://www.ll.mit.edu, Website, Last accessed February 2006.

[3] J.L. Verdegay A. Sancho-Royo, Methods for the construction of membership functions, vol. 14, 1999, pp.1213{1230

Joachim Biskup and Ulrich Flegel, Transaction-based pseudonyms in audit data for privacy respectinfintrusion detection, Proceedings of Recent Advances in Intrusion Detection, 3rd International Symposium, (RAID 2000) (Toulouse, France) (H. Debar, L. M, and S. F. Wu, eds.), Lecture Notes in Computer Science, Springer-Verlag Heidelberg, October 2000, pp. 28{48.

[2] KDD 99, The fifth international conference on knowledge discovery and data mining http://kdd.ics.uci.edu/, Website, Last accessed October 2005.

[4] Magnus Almgren and Ulf Lindqvist, Application-integrated data collection for security monitoring, Proceedings of Recent Advances in Intrusion Detection, 4th International Symposium, (RAID) 2001) (Davis, CA, USA) (W, L. M Lee, and A. Wespi, eds.) Lecture Notes in Computer Science, Springer-Verlag Heidelberg, October 2001, pp. 22{36.

[6] M. Ben-Basaat, Handbook of Statistics 2: Classification, pattern recognition and reduction of dimensionally, ch. Use of Distance Measures, Information Measures and Error Bounds in Feature Evaluation, pp. 773{791, North Holland, 1982.

[5] R. Basu, R.K. Cunningham, S.E.Webster, and R.P. Lippmann, Detecting low-profile probes and novel denial-of-service attacks, Proceedings of the workshop on Information Assurance and Security, United States Military Academy (IEEE, ed.), June 2001, pp. 5{10.

[9] Tepdump public repository, http://www.tcpdump.org/,March 10, 2005, last access.

[7] V. Berk, G. Bakos, and R. Morris, Designing a framework for active worm detection on global networks, Proceedings of the IEEE International Workshop on Information Assurance (Darmstadt, Germany), March 2003, pp. 13{23).

 DOI : 10.35291/2454-9150.2019.0535 © 2019, IJREAM All Rights Reserved.