Probabilistic Real-Time Intrusion Detection System for Docker Containers

Siddharth Srinivasan, Akshay Kumar(✉), Manik Mahajan, Dinkar Sitaram, and Sanchika Gupta

PES University, Bangalore, India
akshayma@gmail.com

Abstract. The use of containers has become mainstream and ubiquitous in cloud environments. A container is a way to abstract processes and file systems into a single unit separate from the kernel. They provide a lightweight virtual environment that groups and isolates a set of processes and resources such as memory, CPU, disk, etc., from the host and any other containers. Docker is an example of container-based technologies for application containers. However, there are security issues that affect the widespread and confident usage of container platform. This paper proposes a model for a real-time intrusion detection system (IDS) that can be used to detect malicious applications running in Docker containers. Our IDS uses n-grams of system calls and the probability of occurrence of this n-gram is then calculated. Further the trace is processed using Maximum Likelihood Estimator (MLE) and Simple Good Turing (SGT) to provide a better estimation of unseen values of system call sequences. UNM dataset has been used to validate the approach and a comparison of the results obtained using MLE and SGT has been done. We got an accuracy ranging from 87–97% for different UNM datasets.

Keywords: Intrusion detection · Containers · Cloud computing

1 Introduction

Virtualization can be implemented in two types, virtual machines (conventional) and containers (new). The conventional way to emulate a system and its resources is through virtual machines running on top of the hypervisor, which runs on top of the host OS. We are working with containers, which are processes running on top of the host OS rather than the hypervisor. There are certain problems with the conventional virtual machine virtualization: There is an increased overhead of running a fully installed guest operating system. There is a significant overhead from calls to the hypervisor from a guest OS. Virtual machines show an inability to freely allocate resources to processes. Top cloud companies like Google (Google Drive), Amazon (AWS) and Microsoft (Azure) are instead using containers, to run their processes in servers as an IaaS. Containers provide a mechanism to isolate a set of processes and resources (Processor, Memory, disk, etc.) from the host and any other containers. This isolation guarantees that any process inside the container cannot see other processes or resources outside the container. Containers are currently being used either as an

S. M. Thampi et al. (Eds.): SSCC 2018, CCIS 969, pp. 336–347, 2019.
https://doi.org/10.1007/978-981-13-5826-5_26

operating system or as an application packaging mechanism. OS containers are virtual environments that share the kernel of the host operating system but provide user space isolation. Application containers are designed to package and run a single service. Most of the companies (like Amazon, Google and Microsoft) use Docker containers to run their applications isolated from the host kernel in the server. Rather than virtualizing the hardware (which requires full virtualized operating system images for each guest), Docker containers virtualize the OS itself, sharing the host OS kernel and its resources with both the host and other containers. Summarily, Docker containers just abstract the operating system kernel rather than the entire device. Docker containers share kernel resources through features like namespaces, chroot, and cgroups. These allow containers to isolate processes, fully manage resources, and ensure appropriate security.

Docker containers are proving to be highly low-weight and hence, fast in its execution and well performing. However, their security has been the key issue, raised in all Docker virtualization conferences. Docker containers are vulnerable, when it comes to attacks like container breakout and Denial-of-Service (DoS). Docker containers are now ubiquitous and a predominant solution when it comes to virtualization in Linux servers, and hence security analysis through intrusion-detection is vital and crucial to ensure safe working of applications. Since the Docker community is always working on rectifying and documenting these vulnerabilities around the clock, this area provided a good scope for us to figure out ways to detect the possible occurrence of vulnerabilities, rather than the vulnerabilities per se. This detection is done through deviations from patterns observed on the official, tested and hence secure Docker images (A container is a running instance of an image. All instructions to get the container up-and-running are in the images). First the IDS is run for safe datasets and its behavior is recorded. Later, by running the IDS on malicious datasets, we'll be able to intrude into the containers and create the anomalies we detected, thus proving ourselves to be correct that those intrusions create problems which are detected. We have tested our approach on UNM datasets [9] to compare the performance. It's a holistic way of reviewing how a container can be affected and that it's not truly safe and secure.

The paper is divided into 5 sections. Sections 2, 3 and 4 talks about Literature Survey, Proposed Approach and Evaluation respectively. Section 5 concludes the paper followed by References.

2 Literature Survey

Sequence-based system call anomaly detection has been around for a long time. Abed et al. [3] used Bags of System Calls (BoSC) frequency-based approach for tracing system calls rather than sequence-based. They were able to detect anomalies in container behavior and analyze them. If the number of deviations of frequencies of occurrences of system calls in a test container exceeds those of the official ones' safe sequences, then an anomaly is detected. It claimed to have better and faster performance than the conventional method. However, we felt that frequencies would be less accurate. Sequences would offer a more concrete whitelist of system calls. This is because, a system call could occur in any order having the same frequency. If that call,

say, deletes a file before another call reads that deleted file, the frequency of occurrences of the system calls would not change, but because of the sequence we can capture the anomalous behavior. Hence we felt that sequences would give a detailed view of the traces and better detect potential and possible anomalies.

[5] has talked about using N-Grams for language modelling. Given the history of words seen, they predicted the next word using the Markov Model of N-Grams. They computed and then smoothed the probabilities of two-word sequences (bigrams) as bigrams can represent all the historical words in that epoch. Using all such probabilities, they were able to get a machine complete and even produce an English sentence. Naseer et al. [8] proposed a classifier for arbitrarily long sequences of system calls using a naïve Bayes classification approach. The class-conditional probabilities of long sequences of system calls are mapped to an Observable Markov Chain Model for classifying incoming sequences efficiently. The technique was benchmarked against leading classification techniques and was found that to perform well against techniques like naive Bayes multinomial, Support Vector Machine (SVM) and Logistic Regression. It also yields a better compromise on detection rate to accuracy. But the problem with their approach was that they do not take care of system call parameters.

[4, 7] leverage system call arguments, contextual information and domain level knowledge to produce clusters for each individual system call. These clusters are then used to rewrite process sequences of system calls obtained from kernel logs. These new sequences are then fed to a naive Bayes classifier that builds class conditional probabilities from Markov modeling of system call sequences. The results were then tested on the 1999 DARPA dataset and was found to show significant performance improvements in terms of false positive rate, while maintaining a high detection rate when compared with other classifiers. The problem with this paper was that identifying the best subset of system calls to cluster was done manually, which is inefficient and the classification of the clusters of system calls was found to take a long time.

Stolfo et al. [6] published a paper, "PAYL: Anomalous Payload-based Network Intrusion Detection" with respect to network payload analysis. The training data is a profile consisting of the relative frequencies of the payloads traced. They classify each and every payload as 1-gram. The relative frequency of each 1-gram is the number of occurrences of the 1-gram divided by the total number of 1-gram. Their standard deviation is then computed port wise. This completes the training data. This computation process is repeated for each testing set traced. The Mahalanobis distance is computed between the training set and each testing set. If the distance exceeds a certain threshold, they raise an (anomaly) alert. This approach proposes intrusion detection for traditional systems using network payloads.

Our objective is to develop an Intrusion Detection System for applications running on Docker Containers i.e. given an application running in a container, our system should be able to determine whether that application is malicious or not. We plan on developing a Host Based Intrusion Detection system (HIDS) to monitor applications running on a single machine. The proposed HIDS can perform the following: (a) Monitor the application running within the Docker container in real time (b) Uses a system call-based approach for detection of anomalies and hence report intrusions when they occur.

3 Proposed Approach

This paper proposes an n-gram approach for intrusion detection using system calls to detect malicious applications in container environment. Unlike the frequency-based approach proposed by Abed et al. [3], every sequence of system call is maintained as an n-gram instead to account for the proportion of system call occurrences, keeping in mind the order in which system calls occur as well. The approach taken can achieve higher accuracy in detecting attacks such as Trojan attacks, DoS, DDoS and SQL Injection that occur in applications running in Docker containers.

Table 1. Example of bigram probabilities (after smoothing)

Bigram sequence	Probability
{alarm, sigprocmask}	0.5
{alarm, sigsuspend}	0.5
{brk, brk}	0.25
{brk, sigprocmask}	0.25
{chdir, open}	1.0

Table 2. Example of trigram probabilities (after smoothing)

Trigram sequence	Probability
{alarm, sigprocmask, select}	0.5
{alarm, sigsuspend, sigreturn}	0.5
{brk, brk, open}	0.25
{brk, fstat, mmap}	0.5
{brk, open, stat}	0.25

The setup for our Intrusion Detection system for a Docker container is as shown in Fig. 1. A common mount point is made between the container and the host system using a shared folder. The Web service running inside the container is traced using the **strace** utility using all the process identifiers associated with the service, and the trace of system calls that are obtained in real-time is passed to the IDS. This merely provides a mechanism for the IDS to read the sequence of system calls generated within the container in real-time.

Every sequence is passed to the IDS where it generates n-grams of system calls and keeps calculating the probabilities of occurrences of these n-grams. These calculated probabilities are used to accumulate the overall relative n-gram probabilities for that session of monitoring the container (Ref Tables 1 and 2).

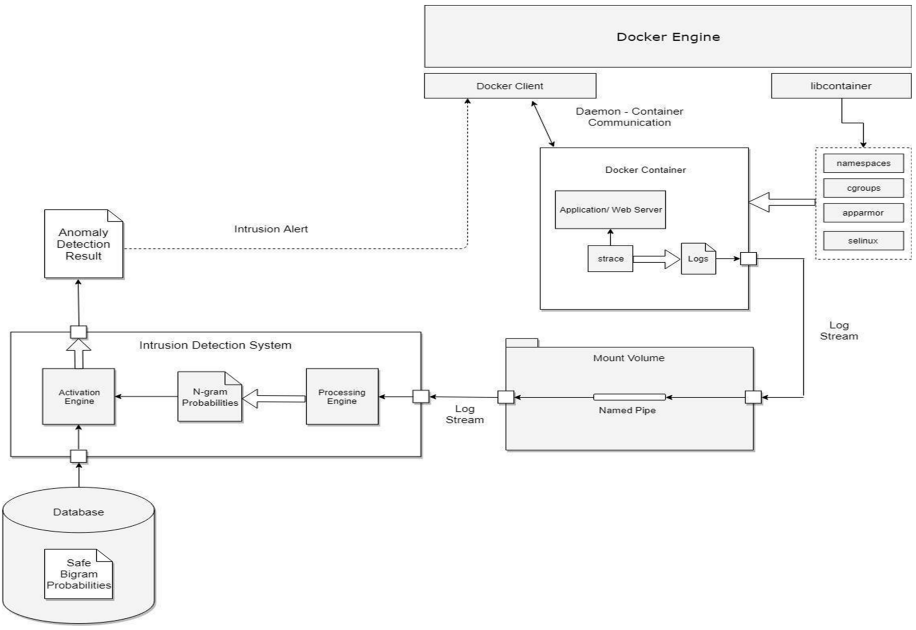

Fig. 1. Structure of proposed real-time intrusion detection system

The HIDS works in two modes: Normal and Detection. In normal mode, **strace** traces the application and the bigram probabilities of these traced sequences of system calls are stored in a database. These sequences are termed as 'safe sequences'. In detection mode, **sqlmap** is used to inject malicious workloads periodically as and when **strace** is running in the normal mode. N-gram probabilities are then calculated for these sequences termed as 'unsafe' sequences.

The detection of an intrusion is based on a thresholding mechanism. The IDS keeps checking each n-gram probability with the probabilities stored in the database. If the n-gram doesn't exist in the database, or if the difference between the probabilities of the n-gram observed and stored is beyond a certain threshold, then flag the behavior as possibly anomalous. If the number of flags reaches a certain threshold, then the IDS claims the activity to be malicious and gives the option to either continue running the container or to completely shut down the operation. The threshold values are obtained by observing the probabilities of n-grams on both normal and detection modes and noting down the highest difference in probability between both modes.

This approach solves the major problem faced in [4, 7], since sequences of system calls which are anomalous as denoted by their n-gram representation is automatically flagged, in contrast with the manual clustering of system call subsets, and the system call sequences constituting to an intrusion can be much more efficiently computed.

Instead of working with raw N-gram probabilities, we processed the trace and operated on the Maximum Likelihood Estimators (MLE) [13] and Simple Good Turing (SGT) instead.

3.1 Maximum Likelihood Estimator

For example, to compute a particular bigram probability of system call y given system call x, we'll compute the count of bigram C(xy) and normalize sum of all bigrams that share the first system call x

$$\mathrm{P}(\mathrm{Wn}/\mathrm{Wn}-1) = \frac{C(Wn-1Wn)}{C(Wn-1)} \tag{1}$$

Similarly, using MLE for calculating n-gram probabilities

$$\mathrm{P}(\mathrm{Wn}/\mathrm{W1}\ldots\mathrm{Wn}-1) = \frac{C(W1\ldots Wn)}{C(W1\ldots Wn-1)} \tag{2}$$

The main advantage of this approach is simplicity, which is useful especially when we wish to isolate the working of the IDS itself, i.e., run the IDS separately in a container. In this case we may make use of a 'bridge' network within which we can allow two Docker containers to securely transfer information from the container providing the service to the container monitoring the other one. The major disadvantage of this approach however is that the processing and memory usage increases as compared to running the IDS on the host system.

3.2 Simple Good Turing

Simple Good-Turing [14] builds on the intuition that we can estimate the probability of events that don't occur based on the number of N-grams that only occur once. According to the rules of Good Turing smoothing, we define what is known as "frequencies of frequencies" for a given count. In other words, if an N-gram occurs 'r' times, we denote Nr as the number of times the count 'r' occurs in the entire frequency distribution of the dataset. The probability of observing any class which has occurred 'r' times in the future is equal to

$$\mathrm{Pr} = \frac{(r+1)\times Nr+1}{Nr} \tag{3}$$

There is one major disadvantage with Good-Turing Smoothing, which is that for large occurring counts however the corresponding frequency of frequency values would tend to Zero even if they exist. To solve this problem, a variant of Good-Turing was introduced, known as Linear Good-Turing (LGT) Smoothing. They fix the issue dealing with frequency of frequency Nr for large 'r' by averaging the large Nr with neighboring values that are zero in occurrence. Hence another estimator is introduced. If q, r and t represent the consecutive indices, then another term Zr is introduced such that,

$$\mathrm{Zr} = \frac{Nr}{0.5\times(t-q)} \tag{4}$$

And once that is done the resulting values are smoothed using log linear regression to minimize the variance that was otherwise large using only Good Turing smoothing. The proposal made for SGT (Simple Good-Turing) Smoothing was that the normal Good-Turing estimates are to be used if the values are considerably different from the LGT (Linear Good-Turing) estimates, and once the saturation point is reached, to use the LGT estimates instead.

4 Evaluation

4.1 Environmental Setup

To test our approach, we ran a containerized web application which is known to be vulnerable. In our case we were running the **dvwa** application in a single container, which is used by security professionals to test their skills in a legal environment. The host system is based on Ubuntu 16.04, and the host executes a python script containing the logic for our Intrusion Detection System. We used MongoDB for storing the bigram probabilities for safe sequences of system calls. A new feature is added while building the IDS. Initially the Activation Engine accepted a small fraction as an extra input which denoted the threshold of the comparison between the currently calculated N-gram probabilities at the given time instant as well as the N-gram Probabilities stored on the Database denoting safe sequence. This only means that this threshold value had to be passed manually and was generally determined by observing the minute deviations that occur with N-gram probabilities between two or more sequences of system calls that denoted the normal mode of operation of the container. Instead of passing it manually, we now make use of a Multiple Linear Regression Model which takes the two different N-gram Probabilities as input and returns the updated threshold probability which must be used for the sake of detecting intrusions. Since the threshold prediction model makes use of a linear regression model, the assumption that we make during the normal mode (training phase) is that the N-gram probabilities (both the current and the safe ones) are linearly dependent on the new threshold value.

The approach proposed was run on different computing systems of different specifications, and it was noted that although the number of n-grams decreases with an increase in the value of ‘n’, there exists a noticeable lag when running the system for large values of ‘n’. This effect can only be attributed to the additional processing required to compute the n-grams present in the traces for a large value of ‘n’. For modern systems, the best value of n is found to be revolving at three and four respectively.

4.2 Validation of Approach

To test our approach, we made use of UNM datasets [9] which are widely used for the sole purpose of validating different approaches for anomaly detection. These datasets contain system calls generated for different kinds of programs, and different intrusions such as Trojan attacks, buffer overflows etc. Some of the normal traces provided are “synthetic”, referring to traces collected by running a prepared script, and some traces

are “live”, referring to traces collected during normal usage of a production computer system. For our convenience, we have grouped the list of system calls generated in each trace by the PIDs of the processes that generate them before creating the required n-grams and corresponding probabilities. For the sendmail dataset, we preprocessed mainly the daemon and the log traces since in event of an intrusion, it is known that most of the changes are observed in these traces. Tables 3 and 4 depicts the number of safe and intrusive system calls, bigrams and trigrams considered for each dataset for testing our model respectively.

A major significance in our analysis is that while testing our approach with the normal traces the accuracy has very low variance and ranges as decimal differences between 99 and 100%. The major variation is observed only by testing the model with intrusive data.

Table 3. Number of safe files evaluated for each UNM dataset

UNM dataset name	Number of system calls	Number of bigrams	Number of trigrams
Synthetic sendmail	1800582	1800232	1799882
Sendmail Cert	1576086	1575792	1575498
Syn_lpr	2398	2389	2380
Inetd	541	538	535
Stide	2013755	2013739	2013723

Table 4. Number of intrusive files evaluated for each UNM dataset

UNM dataset name	Number of system calls	Number of bigrams	Number of trigrams
Synthetic sendmail	1119	1116	1113
Sendmail Cert	6504	6481	6458
Syn_lpr	164232	163231	162230
Inetd	8371	8340	8309
Stide	205935	205830	205725

4.3 Results

The performance of the proposed Intrusion Detection System for Docker containers is expressed by a confusion matrix which contains two rows & two columns and reports the number of false positives, false negatives, true positives, and true negatives respectively. The confusion matrix further allows more detailed analysis by estimating the accuracy, sensitivity and specificity values. The above measures were calculated for IDS using both MLE and SGT estimators in order to determine the performance.

Figure 2 and Table 5 depicts the scatterplot and performance metrics of using MLE over UNM dataset our system. As can be seen from the results obtained our system boasts of high values of sensitivity for all the datasets tested in the range of 96–100%, and the False Positive Rate is very low, ranging from 0–14%, which means that the rate

of false alarms occurring is less. Regarding detecting an intrusion, it was observed that our system performed the best with both the Synthetic sendmail and the STIDE datasets and this could be attributed to the large size of traces found in these datasets.

Compared to the accuracy values calculated when using MLE technique for N-gram occurrences, the corresponding SGT values of accuracy (Ref Table 6 and Fig. 3) were comparatively less. This is because most datasets provided only a single trace or two for safe sequences for system calls as well as a single intrusive trace as well. Simple Good Turing smoothing works best for a vast number of traces available for normal mode of operations so that it can make a better estimate as to whether an incoming trace is safe or malicious. STIDE was the only dataset that provided a huge number of safe traces of system call sequences .

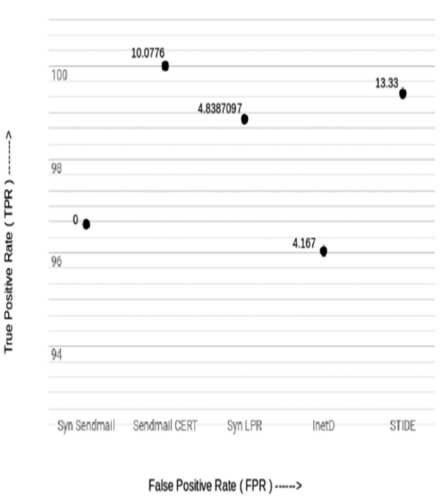

Fig. 2. Scatterplot for TPR vs FPR for MLE

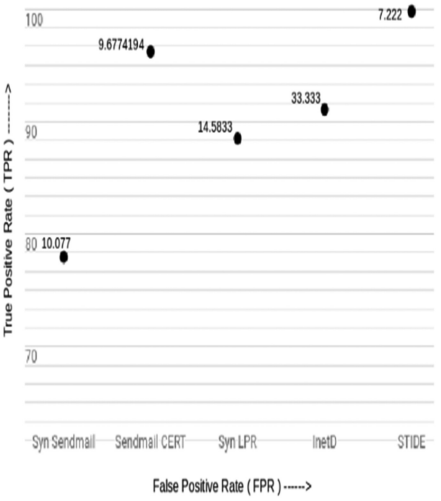

Fig. 3. Scatterplot for TPR vs FPR for SGT

Table 5. Performance metrics of IDS using MLE over UNM dataset

UNM dataset name	True Positive (TP)	True Negative (TN)	False Positive (FP)	False Negative (FN)	Accuracy %	Sensitivity	Specificity
Synthetic sendmail	57	123	0	2	98.9	96.61	100
Sendmail cert	53	118	6	0	96.61	100	95.15
Syn_lpr	87	46	2	1	97.79	98.86	95.83
Inetd	97	3	0	4	96.15	96.03	100
Stide	1012	156	24	6	97.49	99.41	86.67

Table 6. Performance metrics of IDS using SGT over UNM dataset

UNM dataset name	True Positive (TP)	True Negative (TN)	False Positive (FP)	False Negative (FN)	Accuracy %	Sensitivity	Specificity
Synthetic sendmail	46	116	7	13	89.01	77.96	89.92
Sendmail cert	51	112	12	2	92.09	96.22	90.32
Syn_lpr	77	41	7	10	87.40	88.50	85.41
Inetd	92	2	1	9	90.38	91.08	66.67
Stide	1016	167	13	2	98.74	99.80	92.77

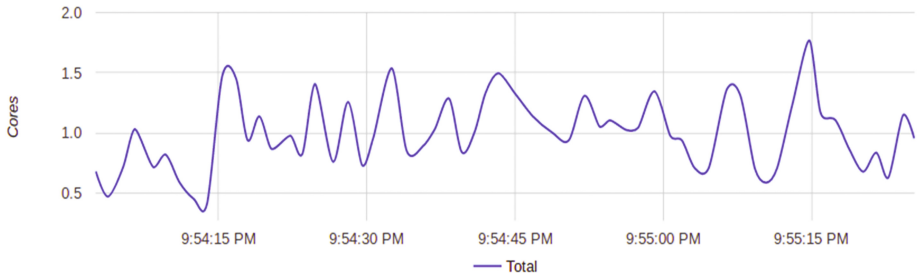

Fig. 4. CPU Usage of running IDS in host system

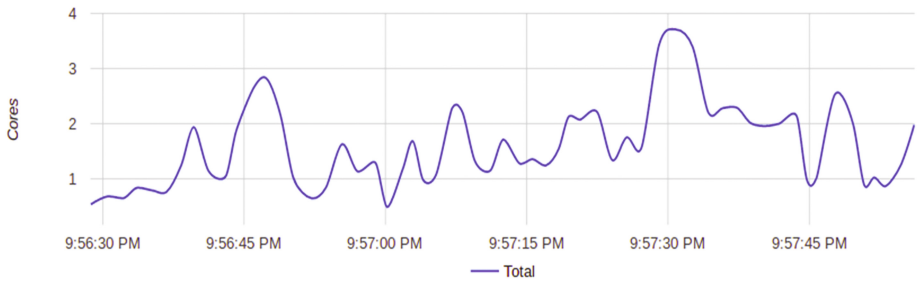

Fig. 5. CPU Usage of running IDS in container

5 Conclusion

Drawing from the results obtained above for our proposed approach, it is evident that using the MLE estimator for denoting the occurrences is recommended only when small number of traces are available for the training mode, and SGT is preferred otherwise. We also observed the CPU utilization while running the IDS in a separate container as compared to running it in the host system itself and found out the CPU utilization is more in the former case than in the latter (Figs. 4 and 5).

A major advantage of running it in the host system is that we can run multiple Docker containers and a single IDS monitoring these containers, hence this approach is much more scalable, and is considered as part of future work. A major enhancement to our approach would be to make possible the detection of other attacks toward Docker containers, such as Container Breakout (which involves automatic escalation of user privileges without any permission request), Cross-site Request Forgery (CSRF), XSS injection, and detection of malware in the container as well. Future work also includes extending the Probabilistic n-gram mechanism for intrusion detection for an implementation of NIDS (Network-Based Intrusion Detection System), which means to analyze the payload of network packets in the same probabilistic manner, as well as mapping the implementation to a standard Machine Learning technique, especially a Bayesian Model or a Hidden Markov Model (HMM).

References

1. Gupta, S., Kumar, P.: System cum program-wide lightweight malicious program execution detection scheme for cloud. Inf. Secur. J. Global Perspect. **23**(3), 86–99 (2014)
2. Gupta, S., Kumar, P.: An immediate system call sequence-based approach for detecting malicious program executions in cloud environment. Wireless Pers. Commun. **81**(1), 405–425 (2015)
3. Abed, A.S., Clancy, C., Levy, D.S.: Intrusion detection system for applications using linux containers. In: Foresti, S. (ed.) STM 2015. LNCS, vol. 9331, pp. 123–135. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-24858-5_8
4. Koucham, O., Rachidi, T., Assem, N.: Host intrusion detection using system call argument-based clustering combined with Bayesian classification. In: 2015 SAI Intelligent Systems Conference (IntelliSys), London, pp. 1010–1016 (2015)
5. Jurafsky, D., Martin, J.H.: "Language Modeling with Ngrams" Speech and Language Processing, Chap. 4 (2016)
6. Wang, K., Stolfo, S.J.: Anomalous payload-based network intrusion detection. In: Jonsson, E., Valdes, A., Almgren, M. (eds.) RAID 2004. LNCS, vol. 3224, pp. 203–222. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30143-1_11
7. Rachidi, T., Koucham, O., Assem, N.: Combined data and execution flow host intrusion detection using machine learning. In: Bi, Y., Kapoor, S., Bhatia, R. (eds.) Intelligent Systems and Applications. SCI, vol. 650, pp. 427–450. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33386-1_21
8. Assem, N., Rachidi, T., El Graini, M.T.: Intrusion detection using Bayesian classifier for arbitrarily long system call sequences. IADIS Int. J. Comput. Sci. Inf. Syst. **9**, 71–81 (2014)
9. Computer Science Department, Farris Engineering Center. Computer Immune Systems Data Sets (1998) http://cs.unm.edu/~immsec/data/synth-sm.html. Accessed 21 Apr 2013
10. Chiba, Z., Abghour, N., Moussaid, K., El Omri, A., Rida, M.: A survey of intrusion detection systems for cloud computing environment. In: 2016 International Conference on Engineering & MIS (ICEMIS), Agadir, pp. 1–13 (2016)
11. Sukhanov, A.V., Kovalev, S.M., Stýskala, V.: Advanced temporal-difference learning for intrusion detection. IFAC-PapersOnLine **48**, 43–48 (2015). https://doi.org/10.1016/j.ifacol.2015.07.005. This work was supported by the Russian Foundation for Basic Research (Grants No. 13–07-00183 A, 13-08-12151 ofi_m_RZHD, 13-07-00226 A, 14-01-00259 A and 13-07-13109 ofi_m_RZHD) and partially supported by Grant of SGS No. SP2015/151, VŠB - Technical University of Ostrava, Czech Republic

12. Hubballi, N., Biswas, S., Nandi, S.: Sequencegram: n-gram modeling of system calls for program-based anomaly detection. In: 2011 Third International Conference on Communication Systems and Networks (COMSNETS 2011), Bangalore, pp. 1–10 (2011)
13. Jurafsky, D., Martin, J.H.: Speech and Language Processing. Copyright © 14. All rights reserved. Draft of September 1, 2014 (2014)
14. Gale, W.A., Sampson, G.: Good-turing frequency estimation without tears. J. Quant. Linguist. **2**(3), 217–237 (1995)