Big Data Based Security Analytics for Protecting Virtualized Infrastructures in Cloud Computing

Thu Yein Win, *Member, IEEE*, Huaglory Tianfield, and Quentin Mair, *Member, IEEE*

Abstract—Virtualized infrastructure in cloud computing has become an attractive target for cyberattackers to launch advanced attacks. This paper proposes a novel big data based security analytics approach to detecting advanced attacks in virtualized infrastructures. Network logs as well as user application logs collected periodically from the guest virtual machines (VMs) are stored in the Hadoop Distributed File System (HDFS). Then, extraction of attack features is performed through graph-based event correlation and MapReduce parser based identification of potential attack paths. Next, determination of attack presence is performed through two-step machine learning, namely logistic regression is applied to calculate attack's conditional probabilities with respect to the attributes, and belief propagation is applied to calculate the belief in existence of an attack based on them. Experiments are conducted to evaluate the proposed approach using well-known malware as well as in comparison with existing security techniques for virtualized infrastructure. The results show that our proposed approach is effective in detecting attacks with minimal performance overhead.

Index Terms—Virtualized infrastructure, virtualization security, cloud security, malware detection, rootkit detection, security analytics, event correlation, logistic regression, belief propagation

1 Introduction

A virtualized infrastructure consists of virtual machines (VMs) that rely upon the software-defined multi-instance resources of the hosting hardware. The virtual machine monitor, also called hypervisor, sustains, regulates and manages the software-defined multi-instance architecture. The ability to pool different computing resources as well as enable on-demand resource scaling has led to the widespread deployment of virtualized infrastructures as an important provisioning to cloud computing services.

This has made virtualized infrastructures become an attractive target for cyberattackers to launch attacks for illegal access. Exploiting the software vulnerabilities within the hypervisor source code, sophisticated attacks such as Virtualized Environment Neglected Operations Manipulation (*VENOM*) [1] have been performed which allow an attacker to break out of a guest VM and access the underlying hypervisor. In addition, attacks such as Heartbleed [2] and Shellshock [3] which exploit the vulnerabilities within the operating system can also be used against the virtualized infrastructure to obtain login details of the guest VMs and perform attacks ranging from privilege escalation to Distributed Denial of Service (DDoS).

- *T.Y. Win is with the Faculty of Business, Computing & Applied Sciences, University of Gloucestershire, Cheltenham GL50 2RH, United Kingdom. E-mail: twin@glos.ac.uk.*
- *H. Tianfield and Q. Mair are with the Department of Computer, Communications and Interactive Systems, Glasgow Caledonian University, Glasgow G4 0BA, United Kingdom. E-mail: {h.tianfield, q.mair}@gcu.ac.uk.*

Manuscript received 26 May 2016; revised 30 Apr. 2017; accepted 10 June 2017. Date of publication 0 . 0000; date of current version 0 . 0000.
(Corresponding author: Thu Yein Win.)
Recommended for acceptance by S. Cui.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TBDATA.2017.2715335

Existing security approaches to protecting virtualized infrastructures generally include two types, namely malware detection and security analytics. Malware detection usually involves two steps, first, monitoring hooks are placed at different points within the virtualized infrastructure, then a regularly-updated attack signature database is used to determine attack presence. While this allows for a real-time detection of attacks, the use of a dedicated signature database makes it vulnerable to zero-day attacks for which it has no attack signatures.

Security analytics applies analytics on the various logs which are obtained at different points within the network to determine attack presence. By leveraging the huge amounts of logs generated by various security systems (e.g., intrusion detection systems (IDS), security information and event management (SIEM), etc.), applying big data analytics will be able to detect attacks which are not discovered through signature- or rule-based detection methods. While security analytics removes the need for signature database by using event correlation to detect previously undiscovered attacks, this is often not carried out in real-time and current implementations are intrinsically nonscalable.

To overcome these limitations, in this paper we propose a novel big data based security analytics (BDSA) approach to protecting virtualized infrastructures against advanced attacks. By making use of the network logs as well as the user application logs collected from the guest VMs which are stored in a Hadoop Distributed File System (HDFS), our BDSA approach first extracts attack features through graph-based event correlation, a MapReduce parser based identification of potential attack paths and then ascertains attack presence through two-step machine learning, namely logistic regression and belief propagation.

The remainder of the paper is arranged as follows. Section 2 presents a review upon the existing security approaches. Section 3 proposes our big data based security analytics (BDSA) approach. Experimental evaluations are presented in Section 4, while Section 5 discusses our BDSA approach in contrast with the related work. Section 6 draws the conclusion.

2 Literature Review

2.1 Malware Detection in Virtualised Infrastructure

Malware refers to any executable which is designed to compromise the integrity of the system on which it is run. There are two prominent approaches to malware detection in cloud computing, namely in-VM and outside-VM interworking approach and hypervisor-assisted malware detection.

2.1.1 In-VM and Outside-VM Interworking Approach to Malware Detection

In-VM and outside-VM interworking detection consists of an in-VM agent running within the guest VM, and a remote scrutiny server monitoring the VM's behaviour. When a potential malware execution is detected the in-VM agent sends the suspicious executable to the scrutiny server, which then uses the signature database to verify malware presence or otherwise and then informs the in-VM agent of the results.

CloudAV, a cloud-based malware detection system featuring multiple antivirus engines, employs in-VM and outside-VM interworking approach to protect the guest VMs against attacks [4]. Apparently the effectiveness of this scheme depends on the frequency at which the virus signatures are updated by the antivirus vendors.

The in-VM and outside-VM interworking approach is also used by *CuckooDroid*, to detect mobile malware presence on Android devices [5]. It consists of an in-device agent which scans executables on the device and sends any suspicious executable to a remote scrunity server which runs a hybrid of anomaly-based and signature-based malware detectors. The scheme first extracts malware features by using static as well as dynamic analysis on malware apps. The obtained features are then used to train a one-class Support Vector Machine (SVM) classifier for anomaly-based detection. Implemented on an emulated Android platform, *CuckooDroid* achieved a detection accuracy of 98.84 percent.

2.1.2 Hypervisor-Assisted Malware Detection

Hypervisor-assisted malware detection, on the other hand, uses the underlying hypervisor to detect malware within the guest VMs.

A hypervisor-assisted malware detection scheme is designed in [6] to detect botnet activity within the guest VMs. The scheme installs a network sniffer on the hypervisor to monitor external traffic as well as inter-VM traffic. Implemented on Xen, it is able to detect the presence of the Zeus botnet on the guest VMs.

A hypervisor-assisted detection scheme is proposed in [7] using guest application and network flow characteristics. This scheme first uses *LibVMI* to extract key process features from the processes running within VMs and then uses *tcpdump* together with the *CoralReef* network packet analysis tool from Center for Applied Internet Data Analysis (CAIDA) to extract network flow features. The obtained features are then used to train one-class SVM classifiers to detect malware presence within guest VMs. Implemented on *KVM*, the scheme is able to detect well-known Distributed Denial of Service and botnet attacks such as Low Orbit Ion Cannon (LOIC) and Zeus.

The hypervisor-assisted detection is also used in AccessMiner [8]. Implemented as a custom hypervisor, AccessMiner monitors normal user behavior within the system and creates access activity models which are used for anomaly-based malware detection. To ensure that the underlying hardware is protected, it intercepts the guest system call requests and uses a policy checker module to determine if it should access the system resource.

2.2 Security Analytics

Security Analytics refers to the application of analytics in the context of cybersecurity [9]. Based on a variety of data collected from different points within an enterprise network, security analytics aims to detect previously undiscovered threats by use of analytic techniques.

Common techniques of security analytics include clustering and graph-based event correlation.

2.2.1 Clustering for Security Analytics

Clustering organises data items in an unlabeled dataset into groups based on their feature similarities [10]. For security analytics, clustering finds a pattern which generalises the characteristics of data items, ensuring that it is well generalized to detect unknown attacks. Examples of cluster-based classifiers include K-means clustering and k-nearest neighbors, which are used in both intrusion detection and malware detection.

Clustering is used for security analytics for industrial control systems [11] in an networked critical infrastructure (*NCI*) environment. First, data outputs from various network sensors are arranged as vectors and K-means clustering is applied to group the vectors into clusters. The MapReduce model is then applied to the grouped clusters to find groupings of possible attack behaviour, thus allowing the detection to be carried out efficiently.

In [12] an "attack pyramid"- based scheme is proposed to detect advanced persistent threats (APTs) in a large enterprise network environment. Based on *threat tree* modeling, different planes (namely hardware, user, network, application) to which an attack may be launched are placed hierarchically with the end goal placed at the top. First, outputs from all available sensors in the network (e.g., network logs, execution traces, etc) are put into contexts. Then, in terms of the contexts various suspicious activities detected at each attack plane are correlated in a MapReduce model, which takes in all the sensor outputs and generates an event set describing potential APTs. Finally, an alert system determines attack presence by calculating the confidence levels of each correlated event.

Security Intelligence techNology for Blocking APT (SINBAPT) [13] uses big data processing such as HDFS and MapReduce together to detect the presence of APTs in an enterprise network environment. Used for anomaly-based detection, the scheme collects log data from different sources (e.g., Netflow, application logs, etc) and applies a

MapReduce model for feature extraction. Once organized into clusters, the data is then used to determine attack presence according to pre-defined rules.

2.2.2 Graph-Based Event Correlation

While clustering determines attack presence through grouping common attack characteristics, it is limited in establishing an accurate correlation which may exist between events. This makes it difficult to accurately identify the sequences of events leading to the presence of an attack within the network, as well as the entry point of the attack.

Graph-based event correlation overcomes this limitation by representing the events from the logs obtained as sequences in a graph. Given a collection of logs from different points within the network (e.g., firewall logs, web server logs, etc.), these events are correlated in a graph with the event features (e.g., timestamp, source and destination IP, etc.) represented as vertices and their correlations as edges. This enables the accurate identification of the entry point which an attack enters, as well as the sequences of events which the attack undertakes.

Graphs-based event correlation is used in *BotCloud*, a botnet detection system for large enterprise environments [14]. Based on the Netflow data which describe the various network traffic flows between clients, the scheme represents the network flow between clients in the form of a dependency graph. The graph is then input into a MapReduce model to identify network IP associations using *PageRank* algorithm.

Graphs-based event correlation is presented in the security framework designed to detect attacks within critical infrastructures [15]. The scheme collects events from differ-ent sources within the network, and generates a temporal graph model to derive different event correlations for threat detection.

Relationships between files are represented as a graph to detect malware presence [16]. The scheme first collects from the clients the file lists which describe their mutual relationships, and determines if there are potentially malicious relationships. The file associations are then used to generate an undirected weighted file relationship graph, and based on the graph a belief propagation classifier is trained. On the dataset from the Comodo Cloud Security Center, the scheme achieved a detection accuracy of 95.81 percent.

2.3 Limitations of Existing Approaches

Existing approaches to detecting attack presence are limited in terms of their ability to detect threats in real-time as well as to scale across multiple hosts.

One of the limitations of existing security approaches stems from the use of a dedicated signature database for threat detection. This applies to approaches that feature a regularly updated attack signature database for threat detection. Typically in the in-VM and outside-VM interworking approach, an in-VM agent detects and passes any suspicious file to the remote scrunity server, which uses the signature database to determine if it is a malware. The dependence on a regularly-updated signature database makes it limited in detecting zero-day attacks. While *BareCloud* [17] and *CloudAV* [4] attempt to get around this limitation by using multiple antivirus engines for threat detection, they are still limited in detecting previously undiscovered attacks due to the *post factum* data in updating the signature database. This is further exacerbated by an increased number of false positives reported by the different antivirus engines.

Security analytics removes the need for signature database by correlating events from the collected logs, but they still suffer from the *post factum* data in training for threat detection. Typically *BotCloud* [14] and *Nazca* [18] collect data over long periods of time (usually over a 24 hour period) and apply analytics for threat detection. While a long period of time allows for a rich collection of data, that entails a tendency in detecting threats which have already taken place over a breadth of time within the network. This makes it difficult, if not impossible, to focus on immediate events and take immediate actions against a compromised point within the network.

Another limitation of existing security approaches is the centralized execution process. For instance, *SINBAPT* [13] runs on a single host, collecting data from various points within the network and analysing them as a single centralized process. While centralized execution process is feasible in network environments in which there is a single centralized server responsible for monitoring all network components, it is infeasible for large network environments in which multiple guest VMs are hosted on different hosts and attack presence has to be communicated to other hosts in near real-time.

3 Proposed Approach

3.1 Overall Framework

The basic idea of our proposed approach is to detect in real-time any malware and rookit attacks via a holistic and efficient use of all possible information obtained from the virtualized infrastructure, e.g., various network and user application logs. Our proposed approach is a big data problem for the following characteristics of the network and user application logs collected from a virtualized infrastructure:

- *Volume:* Depending on the number of guest VMs and the size of the network, the amount of the network and user application logs to be collected can range from approximately 500 MB to 1 GB an hour;
- *Velocity:* The network and user application logs are collected in real-time, in order to detect the presence of malware and rootkit attacks, accordingly the collected data containing its behavior needs to be processed as soon as possible;
- *Veracity:* Due to the "low and slow" approach that malware and rootkit take in hiding their presence within the guest VMs, data analysis has to rely upon event correlation and advanced analytics.

The design principles, which are integral in the development of our BDSA approach to protecting virtualized infrastructures, can be elaborated as follows.

- Design Principle # 1 - Unsupervised classification: The attack detection system should be able to classify potential attack presence based on the data collected from the virtualized infrastructure over time.
- Design Principle # 2 - Holistic prediction: The attack detection system should be able to identify potential attacks by correlating events on the data collected from multiple sources in the virtualized infrastructure.

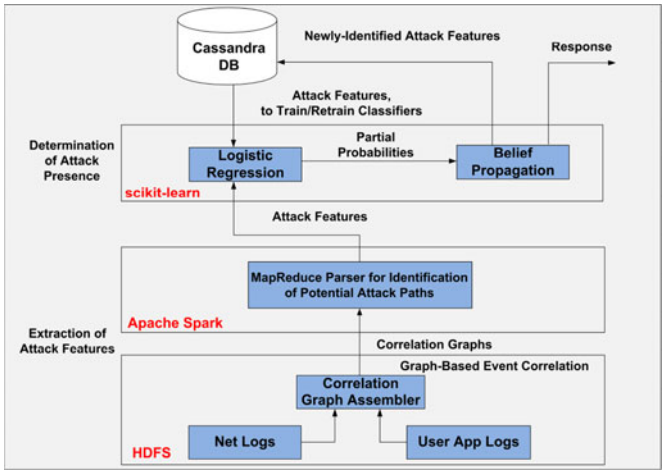


Fig. 1. Conceptual framework of the proposed big data based security analytics (BDSA) approach.

- Design Principle # 3 - Real-time: The attack detection system should be able to ascertain attack presence as immediately as possible so as for the appropriate countermeasures to be taken immediately.
- Design Principle # 4 - Efficiency: The attack detection system should be able to detect attack presence at a high computational efficiency, i.e., with as little performance overhead as possible.
- Design Principle # 5 - Deployability: The attack detection system should be readily deployable in production environment with minimal change required to common production environments.

Fig. 1 illustrates the overall conceptual framework of our proposed big data based security analytics approach, with the different components highlighted in blue. Our BDSA approach consists of two main phases, namely

- Extraction of attack features through graph-based event correlation and MapReduce parser based identification of potential attack paths, and
- Determination of attack presence via two-step machine learning, namely logistic regression and belief propagation.

Prior to the online detection of attacks, there is actually a system initialization, in which offline training of the logistic regression classifiers is carried out, that is, the stored features are loaded from the *Cassandra* database to train the logistic regression classifiers. Specifically, well-known malicious as well as benign port numbers are loaded to train a logistic regression classifier to determine if the incoming/outgoing connections are indicative of an attack presence. Likewise, well-known malware and legitimate applications together with their associated ports are loaded to train a logistic regression classifier to determine if the behavior of an application running within the guest VM is indicative of an attack presence. These trained logistic regression classifiers are ready for online use, upon the extraction of new attack features, to determine if the potential attack paths are indicative of attack presence.

In the Extraction of Attack Features phase, first, it carries out Graph-Based Event Correlation. Periodically collected from the guest VMs, network and user application logs are stored in the *HDFS*. By assembling the information contained in these two logs, the Correlation Graph Assembler (*CGA*) forms correlation graphs.

Then, it carries out the Identification of Potential Attack Paths. A MapReduce model is used to parse the correlation graphs and identify the potential attack paths i.e., the most frequently occurring graph paths in terms of the guest VMs' IP addresses. This is based on the observation that a compromised guest VM tends to generate more traffic flows as it tries to establish communication with an attacker.

In the Determination of Attack Presence phase, two-step machine learning is employed, namely logistic regression and belief propagation are used. While the former is used to calculate attack's conditional probabilities with respect to individual attributes, the latter is used to calculate the belief of an attack presence given these conditional attributes.

From the potential attack paths, the monitored features are sorted out and passed into their logistic regression classifiers to calculate attack's conditional probabilities with respect to individual attributes. The conditional probabilities with respect to individual attributes are passed into belief propagation to calculate the belief of attack presence.

Once attack presence is ascertained, the administrator is alarmed of the attack. Furthermore, the *Cassandra* database is updated with the newly-identified attack features versus the class ascertained (i.e., attack or benign), which are then used to retrain the logistic regression classifiers.

3.2 Extraction of Attack Features

3.2.1 Graph-Based Event Correlation

The IP addresses of the guest VMs are used to obtain the memory process lists on the VMs as well as the ports to which the processes are listening.

TShark is used to obtain the network logs containing the traffic flows of the guest VMs. Specifically it collects the source and the destination IP addresses along with their respective port numbers. It also undertakes the remote execution of the `netstat` command to obtain the guest VMs' memory process lists.

The network logs contain connection entries describing the guest VMs' internal as well as external network connections, namely the source and destination IP addresses (i.e., IP_{source} and $IP_{destination}$) as well as the port numbers (i.e., $Port_{source}$ and $Port_{destination}$) used. Each entry in the network logs is of the format as below:

$$net\_log := \langle IP_{source}, Port_{source}, IP_{destination}, Port_{destination} \rangle. \quad (1)$$

The user application logs, on the other hand, contain process entries detailing the applications running within the guest VMs and the port numbers on which the applications are listening for connections. Each entry in the application logs is of the format as below,

$$app\_log := \langle IP_{guest}, App_{guest}, UserID_{App}, Port_{App} \rangle, \quad (2)$$

where IP_{guest} refers to the guest VM's IP address, App_{guest} refers to the application running on the VM, $UserID_{App}$ refers to the user ID to which the user application is

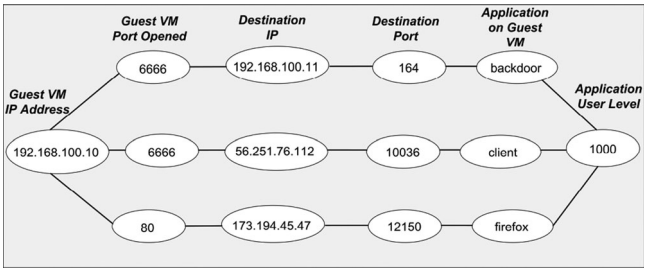


Fig. 2. Correlation graph assembled from network and user logs.

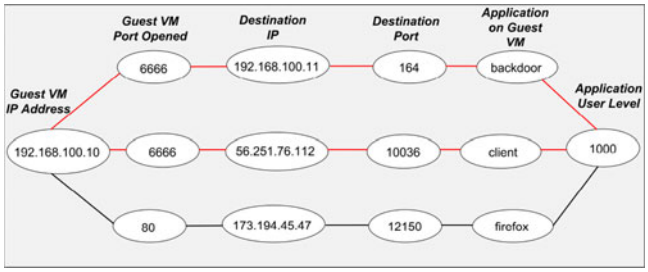


Fig. 3. Potential attack paths in the correlation graph as flagged up by MapReduce parser.

running, and $Port_{App}$ refers to the port opened by the application.

Once obtained, the log entries are used to form a correlation graph based on the following observations. The first observation is that a compromised guest VM tends to communicate more frequently with other guest VMs, resulting in an increase in the network traffic containing its IP address. The second observation is that the communication means of the malware running on the compromised VM is through its execution on it and listening for external connections. In the light of these two observations a correlation graph is formed which best describes the guest VMs' behavior by assembling the information obtained of the network and user application logs.

Before it proceeds to form the correlation graph, first only those entries with the guest VMs' IP addresses either as the source or the destination are filtered out of the network logs. This eliminates the routine traffic flows which periodically check the status of the host high performance cluster (HPC) by applications such as *Apache Hadoop*.

The filtered network log entries are then assembled with the user application logs according to the guest VMs' IPs and the port numbers which are opened by the user applications. A path is grown with the monitored features as vertices and their correlations as edges of the form $\langle IP_{source}, Port_{source}, IP_{destination}, Port_{destination}, App_{guest}, Port_{App}, UserID_{App}\rangle$. As a result a correlation graph is formed an example of which is shown in Fig. 2.

The formed correlation graph, consisting of multiple paths is then stored in the HDFS on the HPC node as a new entry, called correlated log, of the format as below:

$$correlated\_log := \langle IP_{source}, Port_{source}, IP_{destination}, Port_{destination}, App_{guest}, UserID_{App}\rangle. \quad (3)$$

3.2.2 MapReduce Parser for the Identification of Potential Attack Paths

Identification of potential attack paths is carried out by parsing the correlation graph with a MapReduce model. MapReduce is a distributed programming model which consists of two processes namely *Map* and *Reduce*.

In the *Map* process, key-value pairs of the form (k_i, v_i) are sorted from the correlation graph, where k_i denotes the monitored traffic flow while v_i is the count of occurrence of the traffic flow in the graph. Taking the correlation graph in Fig. 2 as an example, the *Map* process represents each path in the graph as a key k_i and its occurrence as a value v_i as shown in Snippet 1.

Snippet 1. (path, count) pairs of correlation graph sorted in *Map* process

(<192.168.100.10, 6666, 192.168.100.11, 164, backdoor, 1000>, 1)

(<192.168.100.10, 6666, 58.251.76.112, 10036, client, 1000>, 1)

(<192.168.100.10, 80, 173.194.45.47, 12150, firefox, 1000>, 1)

In the *Reduce* process, the key-value pairs obtained during the *Map* process are unified. With the same example the *Reduce* process analyzes the intermediate key-value pairs generated from the *Map* process, and unifies all those key-value pairs, aggregating their occurrence counts, if their source IPs as well as source ports are the same regardless of the other elements on the path. This generates a set of new- variant key-value pairs (k_i', v_i'), where k_i' represents the unified path for a distinctive source IP and port, while v_i' is the total occurrence counts within the graph. For the example correlation graph above, the (unified path, aggregated counts) pairs are shown in Snippet 2.

Snippet 2. (unified path, aggregated counts) pairs from *Reduce* process

(<192.168.100.10, 6666, .., .., .., .>, 2)

(<192.168.100.10, 80, 173.194.45.47, 12150, firefox, 1000>, 1)

Flagged up by the *MapReduce* parser's output, any graph paths with an occurrence count greater than one are potential attack paths and are thus picked up and passed onto the determination of attack presence phase. For the example correlation graph above, the potential attack paths are identified (marked in red) as shown in Fig. 3.

3.3 Determination of Attack Presence

The potential attack paths identified from the correlation graph as flagged up by the MapReduce parser can be readily retrieved into the different attack features. We refer to the stripping process as attack feature Sorter out of attack paths. For the determination of attack presence two-step machine learning is used, namely logistic regression and belief propagation.

Logistic regression provides a quick means of ascertaining whether a given test data projects to one of the two predefined classes, as well as supporting the quick training of a

classifier given a training set, $(X \sim Y)$, which denotes a series of features versus classes. This makes it suitable for calculating attack's conditional probabilities with respect to (wrt) individual attributes. Furthermore, whenever an attack presence has been ascertained, the logistic regression classifiers can be quickly retrained in real-time using the newly-identified attack features for future attack detection.

Belief propagation takes into account the conditional probabilities in order to calculate the belief of attack presence within the virtualized environment. This allows for a holistic approach to attack detection, ensuring that the calculated belief accurately reflects the probability contributions from the individual attributes.

The determination of attack presence consists of two phases, i.e.,

- Training and retraining of logistic regression classifiers
- Attack classification using belief propagation

Conditional probabilities with respect to the attributes are calculated based on the features observed from the logs using the trained logistic regression classifiers. Using any of the obtained conditional probabilities with respect to individual attributes alone is not enough to obtain a complete perspective of the attack probability. Therefore, observations of all attributes should be taken advantage of to ascertain attack presence. Belief propagation is used to calculate the belief of an attack by taking into consideration attack's conditional probabilities with respect to all the attributes.

3.3.1 Training and Retraining of Logistic Regression Classifiers

Used in binary classification problems, logistic regression provides a quick means of training a classifier which is used to determine if a particular test data projects to one of the two pre-defined classes.

Logistic regression operates as follows. Let C be a set of two pre-defined classes, i.e., $\{c, \bar{c}\}$ or $\{0, 1\}$ (e.g., $\{attack, benign\}$). Suppose there are n independent features and a feature data series is of the form $\chi = [\chi_1, \chi_2, \ldots, \chi_n]^T$. Let $X = \{\chi^{(1)}, \chi^{(2)}, ..\chi^{(N)}\}$ be the series of N obtained data of feature data series χ, where χ_j denotes the j^{th} sampled data of χ. Let $Y = \{y^{(1)}, y^{(2)}, \ldots, y^{(N)}\}$ be the corresponding class set specifying which one of the two pre-defined classes c and $\bar{c}$ each feature χ projects to.

Four attributes are defined to characterize a potential attack, namely incoming network connections ($in\_connect$), outgoing network connections($out\_connect$), unknown binary executions ($unknown\_exect$) and opened ports ($port\_change$). While the monitored features refer to the sensor data out of the computer system being monitored, attributes are defined to characterize the situation where an attack may present. The first two attributes are used to determine attack presence based on their source and destination port numbers, while the latter two attributes are used to determine attack presence based on the applications running within the guest VMs as well as the ports opened by the applications.

In order to determine the presence of the attack with respect to the attributes, logistic regression classifiers are trained for analyzing the source and destination ports as well as the applications and the ports which are opened in the guest VMs.

Logistic regression calculates the probability P of attack at which a feature χ projects to one of the two pre-defined classes using the *logit* function as below:

$$P(y = c|\chi) = \frac{1}{1 + e^{-\xi}}, \tag{4}$$

where $\omega_0, \omega_1, \omega_2, \ldots, \omega_n$ are the weighting coefficients, and

$$\xi = \omega_0 + \omega_1\chi_1 + \omega_2\chi_2 + \cdots + \omega_n\chi_n. \tag{5}$$

In the context of our BDSA approach, we set two logistic regression classifiers LR_{app} and LR_{port} using Eq. (4). Once trained, beforehand in a batch, and retrained with newly-identified attack features, the conditional probabilities with respect to individual attributes are calculated using the respective logit functions.

To train a logistic classifier for port analysis we have gathered a set of 300 port numbers used by different malware applications as well as another set of 300 ports used by legitimate applications (e.g., `SSH`) to be the training data set. While the malware port numbers are obtained from *SANS* [19], the port numbers used by legitimate applications are obtained from Internet Assigned Numbers Authority (IANA) [20].

In order to train the logistic regression classifier, the obtained ports are first categorized into two groups, namely $sys\text{-}port$ containing the legitimate port numbers, and $malware\ port$ containing the malware port numbers. Each of the port categories is then encoded with a numerical value, with $sys\ port$ assigned a value of 1 while $malware\ port$ a value of 2, so that they can be represented as feature vectors χ_{port} during the training of the logistic regression classifier.

The port numbers are treated as numerical values, in order to cater for the port numbers which do not deviate significantly from those in the training set and thus belong to the same port category. For example the `nginx` web server listens for connections on port 80 when deployed in a guest VM without any web server running on it prior to its deployment. When deployed on a guest VM which already has another web server (e.g., `Apache`) running on it, however, it needs to update its default port to another value (e.g., 82) since the `Apache` web server also listens for con- nections on port 80 to avoid causing access conflicts. Therefore, by treating the port numbers as numerical values in the feature set, it allows minor port changes such as this to be classified as legitimate port numbers without misclassification. During the experiments, we find that the trained port logistic regression classifier is able to identify a port number as belonging to the same port category if it does not deviate from the training set port beyond 2 (i.e., port 82 is classified as a legitimate port due to its close proximity to port 80).

Table 1 shows the port numbers together with their encoded port category values and their classifications, with 0 representing a legitimate port and 1 representing a malware port.

Using the representation as shown in Table 1, a training set $(X_{port} \sim Y_{port})$ are created. X_{port} consists of a series of feature vectors χ_{port} each of which is of the form $\chi_{port} = T$ $[\chi_{port\_number}, \chi_{port\_category\_value}]$, and Y_{port} contains the

TABLE 1
Examples of Training Set: Ports versus Classes

Port number (χ_1)	Port category	Port category value (χ_2)	Class (y)
22	sys_port	1	0
80	sys_port	1	0
8,080	sys_port	1	0
6,666	malware_port	2	1
1,090	malware_port	2	1
7,777	malware_port	2	1

TABLE 2
Training Set: Applications versus Classes

Application category	Application category value (χ_1)	User ID	User ID value (χ_2)	Port number (χ_3)	Class (y)
web_app	1	1,000	1	80	0
web_app	1	0	0	81	0
sys_util	2	0	0	5,353	0
unknown	3	0	0	164	1
unknown	3	1,000	1	7,777	1

corresponding class which each χ_{port} projects to. Using the first entry in Table 1 as an example, its feature vector χ_{sys_port} is represented as $[22, 1]^T$ while its corresponding class vector y_{sys_port} is represented as 0.

The obtained training set $X_{port} = \{\chi^{(1)}, \chi^{(2)}, \ldots, \chi^{(i)}\}$ together with its corresponding class vector $Y_{port} = \{y^{(1)}, y^{(2)}, \ldots, y^{(i)}\}$ are then used to train a logistic regression classifier using *scikit-learn* which uses Eq. (6) to train the classifier. The training set $(X_{port} \sim Y_{port})$ are stored in the *Cassandra* distributed database

$$P(Attack|\chi_{port}) = \frac{1}{1 + e^{-\xi_{port}}}, \tag{6}$$

where,

$$\xi_{port} = \omega_0 + \omega_1\chi_1 + \omega_2\chi_2 + \cdots + \omega_n \cdot \chi_n \tag{7}$$

Similarly, to train a logistic regression classifier for application analysis we have identified benign internet-interfacing user applications (e.g., `firefox` for web browsing, `nginx` for web server) as well as those applications that are frequently used by malware and botnet programs (e.g., *netcat*). The identified applications are categorized into three categories: *web_app*, *sys_util*, and *unknown* depending on their usages. Each of the application category is then encoded with a numerical value with *web_app* assigned a value of 1, *sys_util* a value of 2, and *unknown* a value of 3. Similarly each of the user ID is encoded with a numerical value, with user ID 0 (i.e., `root` user) assigned a value of 0 and user ID 1000 (i.e., non-`root` user) assigned a value of 1. Table 2 shows the application categories and the user IDs together with their respective category values and classifications, with 0 representing a legitimate application and 1 representing a possible malware application.

Once the features are encoded with numerical values, a training data set $(X_{app} \sim Y_{app})$ then is formed , X_{app} the series of feature data series $\chi_{app} = [\chi_{app\_category\_value}, \chi_{user\_id\_value}, \chi_{port}]^T$ together with Y_{app} containing the corresponding class which each χ_{app} projects to. Using the first entry in Table 2 as an example, its feature vector χ_{web_app} is represented as $[1, 1, 80]^T$ and its corresponding class vector y_{web_app} is represented as 0.

The training set $X_{app} = \{\chi^{(1)}, \chi^{(2)}, \ldots, \chi^{(i)}\}$ together with its corresponding class vector $Y_{app} = \{y^{(1)}, y^{(2)}, \ldots, y^{(i)}\}$ are then used to train a logistic regression classifier using *scikit-learn* which uses Eq. (8) to train the classifier. The training set $(X_{app} \sim Y_{app})$ are stored in the *Cassandra* distributed database as separate column tables

$$P(Attack|\chi_{app}) = \frac{1}{1+e^{-\xi_{app}}}, \tag{8}$$

where,

$$\xi_{app} = \omega_0 + \omega_1\chi_1 + \omega_2\chi_2 + \cdots + \omega_n\chi_n. \tag{9}$$

It should be noted that for each logistic regression classifier, it should have its own weighting coefficients, corresponding the respective feature vector. During the training of the logisitc regression classifers for our proposed BDSA approach, the `scikit-learn` machine learning package uses the *Coordinate Descent* [21] algorithm to automatically calculate the weights $\omega_0, \omega_1, \omega_2, .., \omega_j$ for a given training set $(X \sim Y)$.

Unknown ports and application are ascertained using the trained logistic classifiers. The column tables of the respective features in the Cassandra database are then updated and used to retrain the logistic classifiers for future classifi-cation. The trained port and application logistic regression classifiers are used to calculate the conditional probabilities which are input into belief propagation, as inputs their respective feature vectors.

3.3.2 Attack Classification Using Belief Propagation

Presence of attack is determined by analyzing four attributes, namely incoming network connections (*in_connect*), outgoing network connections (*out_connect*), unknown binary executions (*unknown_exect*) and opened ports (*port_change*). This is based on the observation that the presence of an attack tends to result in changes in these attributes, as the infected guest VM attempts to establish external connections with the remote attacker. With each attribute represented by a node, they form a Bayesian network as illustrated in Fig. 4a.

Used in graphical models such as Bayesian networks and Markov Random Fields (*MRF*), belief propagation is used calculate the probability distribution (i.e., belief) of a target node's state using *message passing* [22]. Given a node v in a Bayesian network, the belief $BEL(v)$ of its state is calculated using the marginal probabilities from its neighbouring nodes. Belief propagation takes into account the neighbouring nodes'

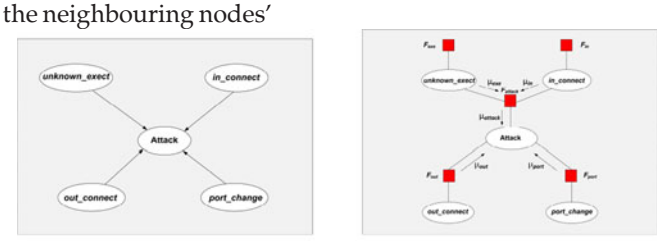


Fig. 4. (a) Bayesian network of attributes, (b) Bayesian network with factor graphs.

individual influence in calculating the belief of v's state, and is therefore used in our BDSA approach for determining attack presence.

Given feature vectors (i.e., $\chi^{in\_connect}$, $\chi^{out\_connect}$, χ^{port_change}, $\chi^{unknown\_exect}$) which are of the following form,

$$\begin{aligned}
\chi^{in_connect} &= [\chi_{port_number}, \chi_{port_category_value}]^T \\
\chi^{out_connect} &= [\chi_{port_number}, \chi_{port_category_value}]^T \\
\chi^{port_change} &= [\chi_{app_category_value}, \chi_{user_id_value}, \chi_{port}]^T \\
\chi^{unknown_exect} &= [\chi_{app_category_value}, \chi_{user_id_value}, \chi_{port}]^T,
\end{aligned} \tag{10}$$

the port and application logistic regression classifiers (i.e., LR_{port} and LR_{app}, respectively) which are trained using `scikit-learn` produce as outputs their respective conditional probabilities which calculate the probabilities of each feature belonging to each of the two pre-defined classes (i.e., *Attack* and *Benign*) which are of the form as below:

$$\begin{aligned}
P^{port_change}(Attack|\chi^{port_change}) &= \Big[P^{port_change}_{Attack}, \\
&\qquad P^{port_change}_{Benign}\Big] \\
P^{unknown_exect}(Attack|\chi^{unknown_exect}) &= \Big[P^{unknown_exect}_{Attack}, \\
&\qquad P^{unknown_exect}_{Benign}\Big] \\
P^{in_connect}(Attack|\chi^{in_connect}_{port}) &= \Big[P^{in_connect}_{Attack}, \\
&\qquad P^{in_connect}_{Benign}\Big] \\
P^{out_connect}(Attack|\chi^{out_connect}_{port}) &= \Big[P^{out_connect}_{Attack}, \\
&\qquad P^{out_connect}_{Benign}\Big].
\end{aligned} \tag{11}$$

P_{Attack} represents the conditional probability with respect to an attribute being indicative of an attack, and P_{Benign} being the conditional probability with respect to an attribute being indicative of a benign. Intuitively if a given attribute projects to an attack, then its attack probability (i.e., P_{Attack}) would be much higher than its benign probability (i.e., P_{Benign}) and the reverse would be true if it were benign.

The training set for $\chi^{in\_connect}$ and $\chi^{out\_connect}$ corresponds to the entries as shown in Table 1, and the training set for χ^{port_change} and $\chi^{unknown\_exect}$ corresponds to the entries as shown in Table 2.

While the trained port and application logistic regression classifiers provide the conditional probabilities with respect to individual attributes, each of them on its own is not able to provide a complete picture of attacks within the virtualized environment. Therefore, belief propagation is applied to calculate the belief in the presence of attack given these conditional probabilities.

To apply belief propagation, the monitored features are first represented as nodes in a Bayesian network as shown in Fig. 4a. The Bayesian network provides a representation of the relationship between different features in determining attack presence. Each node consists of a Conditional Probability Table(*CPT*) containing the marginal probabilities of each possible state (i.e., attack or benign) with respect to the attribute. The initialized CPTs of each of the nodes in the Bayesian network are shown in Tables 3a, 3b, 3c, 3d, and 3e. The initialised values in the CPTs of each node act as placeholders to ensure consistency prior to the execution of our BDSA approach.

TABLE 3
CPTs

(a) *unknown_exect*

Attack	Benign
0.5	0.5

(b) *in_connect*

Attack	Benign
0.5	0.5

(c) *out_connect*

Attack	Benign
0.5	0.5

(d) *port_change*

Attack	Benign
0.5	0.5

(e) *Attack*

Attack	Benign
0.5	0.5

During the execution of the approach, however, the CPTs of the monitored features are updated with the respective attack and benign probabilities (i.e., P_{Attack} and P_{Benign}) which are calculated by the trained port and application logistic regression classifiers.

After the marginal probabilities are represented as CPTs, the belief (BEL_{Attack}) of the *Attack* node's state is then calculated using *message-passing*. This involves passing the marginal probabilities with respect to individual attributes (i.e., $port\_change$, $unknown\_exect$, $in\_connect$, and $out\_connect$) into the *Attack* node in the identified Bayesian network. Their attack probabilities (i.e., $[P_{Attack}]$) are then aggregated in the *Attack* node before calculating BEL_{Attack} as below:

$$\begin{aligned}
BEL_{Attack} = {} & P^{port_change}(Attack|\chi^{port_change}) \\
& \times P^{unknown_exect}(Attack|\chi^{unknown_exect}) \\
& \times P^{in_connect}(Attack|\chi^{in_connect}_{port}) \\
& \times P^{out_connect}(Attack|\chi^{out_connect}_{port}).
\end{aligned} \tag{12}$$

However one of the limitations of this approach is the size of the CPT table for the *Attack* node. Given the number of nodes involved, the *Attack* node has to maintain 32 entries ($2^5 = 32$) containing the conditional probability distributions of each of the four nodes within the Bayesian network as well as its own attack probabilities obtained through applying Eq. (12). In addition it also makes it difficult to update the CPT entries within the *Attack* node efficiently to reflect the updated CPT values of the individual nodes, as they are updated during the execution of our BDSA approach. The individual CPTs of the attributes as well as the joint conditional probabilities between them are therefore represented as a *factor graph* [23].

Used in factor graphs to represent the structure of a factorization, factor graphs in a Bayesian network encode the individual as well as joint Conditional Probability Tables (CPTs) among the nodes in the Bayesian network [23]. Given belief propagation's message-passing formula to cal culating marginal probabilities, representing the CPTs as factor graphs allows the changes in the local CPTs to be

TABLE 4
Joint CPT Between *Unknown_Exect*, *in_Connect*, and *Attack*

$node_{unknown\_exect}$	$node_{in\_connect}$	$node_{Attack}$
Attack	Attack	Attack
Attack	Benign	Attack
Benign	Attack	Attack
Benign	Benign	Attack
Attack	Attack	Benign
Attack	Benign	Benign
Benign	Attack	Benign
Benign	Benign	Benign

tracked more efficiently during the execution of the algorithm.

In the updated Bayesian network identified in Fig. 4b, the factor graphs are illustrated by red square boxes together. Factor graphs F_{exe}, F_{in}, F_{out}, and F_{port} represent the CPTs of the individual attributes, F_{attack} represent the joint CPT between *unknown_exect*, *in_connect*, and *Attack* which is of the following form as shown in Table 4.

The use of factor graphs reduces the number of entries in the joint conditional probability distributions for the $Attack$ node (denoted by F_{attack}) since it only needs to track 8 entries representing the conditional probabilities between 3 nodes (i.e., *unknown_exect*, *in_connect*, and *Attack*) as shown in Table 4. This makes it easier to update its conditional probability values during the execution of our BDSA approach.

The process of belief propagation can further be explained, with an example of the potential attack paths as shown in Fig. 2.

Given a potential attack path as shown in Snippet 3, the logistic regression classifiers for $unknown\_exect$ as well as $in\_connect$ determine if the application and the incoming connection, respectively, are indicative of a malware attack presence. First, the application details (i.e., *backdoor*, *unknown*, 1000, 164) and the incoming connection details (i.e., 192.168.100.10, 6666) are extracted from the attack path. They are then represented as feature vectors and put as test data to their respective classifiers. Similarly, for the logistic regression classifiers for $out\_connect$ and $change\_port$, the outgoing connection (i.e., 192.168.100.11, 164) and application port details (164, $malware\_port$) are extracted and then put as test data to their respective logistic regression classifiers.

Snippet 3. Example of potential attack path

(< 192.168.100.10, 6666, 192.168.100.11, 164, backdoor, 1000 >, 1)

Using the updated Bayesian network with factor graphs, the message-passing approach of belief propagation works as follows. Upon receiving the marginal probabilities from their respective logistic regression classifiers, each of the nodes in the updated Bayesian network calculates a message μ which represents the probability of their respective attributes being an attack ($Attack$) or benign ($Benign$). The messages μ are passed into the $Attack$ node to calculate belief BEL_{Attack} of its state and is calculated as below, where each Ω corresponds to each of the nodes in the Bayesian network (i.e., $\Omega_1 = unknown\_exect$, $\Omega_2 = in\_connect$, $\Omega_3 = out\_connect$, and $\Omega_4 = port\_change$)

$$\begin{aligned}
\mu_{exe\rightarrow Attack}(Attack) &= \sum_{exe\in\Omega_1} F(exe, attack)\\
\mu_{in\rightarrow Attack}(Attack) &= \sum_{in\in\Omega_2} F(in, attack)\\
\mu_{out\rightarrow Attack}(Attack) &= \sum_{out\in\Omega_3} F(out, attack)\\
\mu_{port\rightarrow Attack}(Attack) &= \sum_{port\in\Omega_4} F(port, attack).
\end{aligned} \tag{13}$$

While the messages from the $out\_connect$ and $port\_change$ (i.e., μ_{out} and μ_{port}, respectively) directly go into the $Attack$ node for calculation, the messages from $unknown\_exect$ and $in\_connect$ (i.e., μ_{exe} and μ_{in}, respectively) are passed into factor node F_{attack} to calculate their joint conditional probability distribution which reflects their mutual relationship. This is done by multiplying each entry in F_{attack} with μ_{exe} and μ_{in}, before summing over all possible states (i.e., Attack and Benign) of the *unknown_exect* and *in_connect* nodes. This results in the generation of a message $\mu_{F\,attack}$ which is calculated as below, and passed into the *Attack* node

$$\mu_{attack} = \sum_{unknown\_exect}\sum_{in\_connect} F_{attack}(unknown\_exect, in\_connect) \times \mu_{exe} \times \mu_{in}. \tag{14}$$

Using the probability values calculated by the $out\_connect$ and $change\_port$ nodes(i.e., μ_{out} and μ_{port} respectively), the belief BEL_{Attack} is calculated as below:

$$BEL_{Attack} = \mu_{port} \times \mu_{F_{attack}} \times \mu_{out}. \tag{15}$$

Algorithm 1 provides the pesucode of the belief propagation algorithm which is used in our BDSA approach.

Algorithm 1. Belief Propagation for BDSA

Input: P^{port_change}, $P^{unknown\_exect}$, $P^{in\_connect}$, and $P^{out\_connect}$

1: **Initialize:** Create the Bayesian network of attack features using factor graphs as shown in Fig. 4b
2: Set the factor graphs F_{exe}, F_{in}, F_{out}, and F_{port} with the placeholder CPTs as shown in Tables 3a $\sim$ 3e.
3: **while** True **do**
4: Update the factor graphs F_{exe}, F_{in}, F_{out}, and F_{port} with the respective conditional probabilities P_{Attack} and P_{Benign}.
5: Calculate $\mu_{exe\rightarrow Attack}$, $\mu_{in\rightarrow Attack}$, $\mu_{out\rightarrow Attack}$, and $\mu_{port\rightarrow Attack}$
6: For $unknown\_exect$ and $in\_connect$, calculate F_{attack} using Eq. (14).
7: Calculate BEL_{Attack} using Eq. (15).
8: **if** $BEL_{Attack} < lower\_belief$ **then**
9: Alarm "attack presence".
10: Update the tables in *Cassandra* DB with newly-identified attack features.
11: End do
12: End

At the initialization phase, the prior probabilities with respect to individual attributes are assigned values based on the initial observations obtained offline. During the lifetime of the execution of the BDSA approach, the

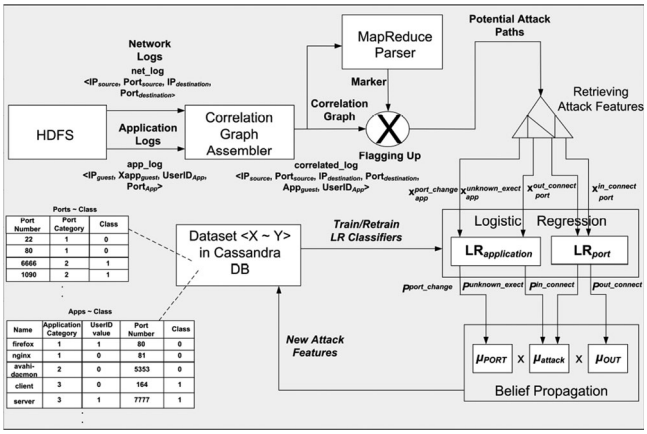


Fig. 5. Information flows in our BDSA approach.

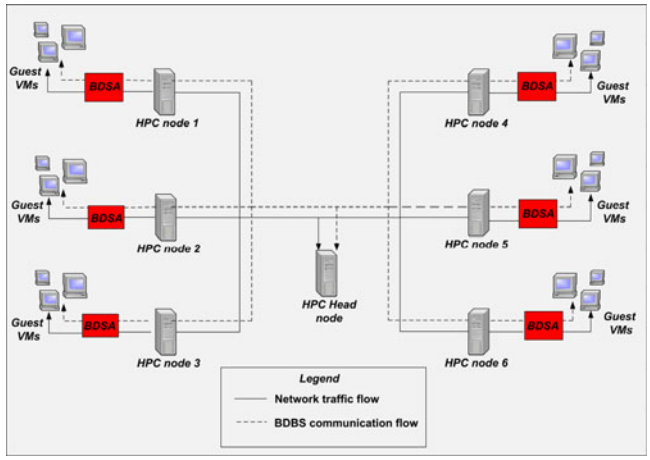


Fig. 6. Testbed system topology.

probabilities contained in the factor nodes are updated based on the updated logistic classifiers values.

3.4 Overall Algorithm of BDSA

Our BDSA approach can be formulated in pseudocodes as shown in Algorithm 2. The overall information flows of our BDSA approach can be illustrated as in Fig. 5.

Algorithm 2. Security Analytics in BDSA

1: **Initialize:** Obtain benign and malicious parameters of the attack features from *Cassandra* DB.
2: Train classifiers for monitored features using Logistic Regression.
3: **while** True **do**
4: Collect network and user application logs from guest VMs.
5: Filter network log entries using the guest VMs' IP addresses.
6: Form correlated_log.
7: Use correlated_log to form a correlation graph G.
8: Input G into MapReduce parser to identify potential attack paths $\{attack\_paths\}$, which is a sub-set of all graph paths as shown in Fig. 3.
9: **for** each $attack\_path$ in $\{attack\_paths\}$ **do**
10: $i \leftarrow 0$.
11: **for** each monitored feature $t_{feature}$ in $attack\_path$ **do**
12: Calculate P^{port_change}, $P^{unknown\_exect}$, $P^{in\_connect}$, and $P^{out\_connect}$
13: Pass P^{port_change}, $P^{unknown\_exect}$, $P^{in\_connect}$, and $P^{out\_connect}$ into Step. 4 of Algorithm 1.
14: End do
15: End do
16: End

The execution of the proposed BDSA approach begins by loading all well-known malicious as well as benign port numbers from the distributed Cassandra database. Both of these port types are then used to train a classifier using using logistic regression. This allows the proposed approach to determine on-the-fly the probability of an unknown port being malicious, before passing it to the belief propagation framework for final aggregation.

A trained logistic classifier is used to determine if any of the attributes are malicious or benign, before passing their respective probabilities to the belief propagation process for final probability aggregation. Belief propagation process takes attack's conditional probabilities with respect to individual attributes to calculate the belief of attack presence, taking into account each conditional probability values to ensure that the value obtained is not influenced only by any conditional probability alone.

4 Experimental Evaluation

4.1 Testbed Setup

The proposed big data based security analytics was implemented using Python. For experiments, BDSA is run on the HPC server nodes running Ubuntu 14.04 in the on-campus Virtualiszation Open Technology Research (*VOTER*) network. Fig. 6 illustrates the testbed setup in prototyping the proposed BDSA approach, while the software stack on each HPC node is illustrated in Fig. 7. Each of these servers consists of an Intel Xeon quadcore processor at 2.66 GHz along with 12 GBs of memory, with Linux kernel version 3.18.18 (64-bit) running on it. A virtualization environment is first set up using Kernel-based Virtual Machine (*KVM*) on each of the HPC nodes to enable multiple guest VMs to be run on them, as well as support their migration across the nodes. Apache Hadoop is then installed on the server nodes to support distributed log storage, and Apache Spark is installed to provide real-time data collection and MapReduce parsing. Cassandra columnar database system is installed on top of it to support distributed storage of identified

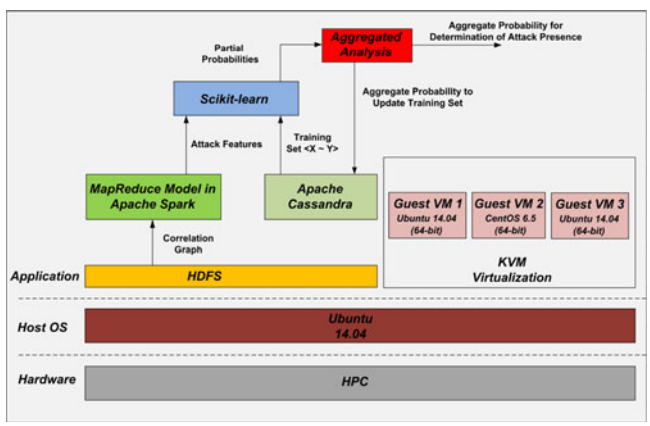


Fig. 7. Software stack on HPC node.

TABLE 5
Malware and Rootkits Tested

Malware/ Rootkit	Category	Execution space on guest VM	Characteristics
Reverse shellcode	Malware	User space	Establishing an external reverse shell connection
C & C botnet	Malware	User space	Creating a master-slave botnet connection
XingYiQuan	Rootkit	Kernel space	Executing in the guest VM's kernel and establishing an external connection
Azazel	Rootkit	Kernel space	Executing in the guest VM's kernel and establishing an external connection

malicious applications and ports, as well as the real-time retraining of the logistic regression classifiers. Finally the *scikit-learn* Python machine learning package is then installed on the nodes to enable the creation of logistic regression classifiers for the BDSA approach.

Based on our experiments, an attack path is considered malicious if the belief BEL_{Attack} calculated from belief propagation is below the threshold $lower\_belief = 0.2$.

The BDSA approach is evaluated by creating a guest VM running CentOS 6.5 as well as another two guest VMs running Ubuntu 14.04 (64-bit) on one of the aforementioned HPC server nodes. Aspects for evaluation include the ability to detect both userspace malware as well as kernel-level rootkit attacks and the time taken to detect the presence of attacks within the guest VMs.

4.2 Detection of Userspace Malware and Kernel-Level Rootkits

The ability of the BDSA approach to detect different malware attacks is evaluated by executing the two userspace malware programs as well as the two kernel-level rootkits on the guest VMs. The malware and rootkits are taken from PacketStorm [24] as shown in Table 5. They were selected due to the availability of their source code, which enables the severity of their attacks to be modified and tested against our BDSA approach.

4.2.1 Detection of Userspace Malware

Also known as *application-level* malware, userspace malware runs at the application-level of the guest operating system alongside other legitimate applications. The ability of our BDSA approach to detect userspace malware is evaluated by executing the aforementioned userspace malware on the guest VMs.

In order to run the userspace malware, a test scenario is set up, that is, one guest VM acts as an attacker while another guest acts as an attack victim. The attacker VM is then made to listen to different non-standard port numbers using `netcat`, and then runs the reverse shellcode on the victim VM. The same test scenario is used for creating a Command & Control (C & C) botnet, by running the server component of the botnet on the attacker VM and its client component on the guest VM. In both of the test scenarios, the userspace malware is executed *as is* with only the hard-coded destination IP addresses and the port numbers modified.

Both userspace malware programs are executed 5 times with up to three guest VMs. In all cases, our BDSA approach is able to detect them through monitoring the communication flows between them as well as the ports which are opened on the guest VMs.

4.2.2 Detection of Kernel-Level Rootkits

While userspace malware runs at the application-level alongside other legitimate applications, kernel-level rootkits run within the kernel of the operating system. Rootkit normally proceeds in two steps. First, rootkit makes attempts to gain privileged level (`root`) access into the operating system. Then, it installs itself into the operating system kernel as a Loadable Kernel Module (*LKM*). Because it is the privileged level at which they are executed, rootkits are difficult to be detected using traditional application-level malware detection approaches.

The ability of our BDSA approach to detect kernel-level rootkits is evaluated by executing the *XingYiQuan* and *Azazel* rootkits on the guest VMs. These rootkits take control of the guest VM by modifying the underlying system call table (`sys_call_table`) entries and establishing external network connections using the *Netfilter* kernel module, which thus makes it difficult for application-level firewalls to detect the communication flows. However, while *XingyiQuan* is not persistent across reboots *Azazel* is persistent in the guest VM's kernel across boots.

As in the case for user-level malware in order to run rootkits, a client-server test scenario is set up, that is, one guest VM acts as an attacker while another VM acts as an attack victim. The rootkits are then run on the client, with the attacker VM made to listen for connections using `netcat`.

Both rootkits are executed 5 times with up to three guest VMs. In all cases, our BDSA approach is able to detect them. By remotely executing the `netstat` command at the `root` evel, our BDSA approach is able to detect the applications as well as the ports which are being opened by the rootkits.

4.3 Measurement of the Average Detection Time

In order to measure the amount of time taken for our BDSA approach to detect attack presence in the guest VMs, the two userspace malware programs as well as the two kernel-level rootkits taken from PacketStorm [24] as shown in Table 5 are executed on the guest VMs. The tests are carried out in 3 cases, namely with 1 guest VM, with 2 guest VMs, and with 3 guest VMs, respectively.

In each test first the malware programs and rootkits are executed in their respective execution spaces on the guest VMs, with the BDSA approach running on the HPC host. The malware programs and rootkits are executed on the guest VMs, and the detection time for each attack execution is recorded accordingly. The detection time D of one test is the summation of the recorded times after the execution of the malware and rootkits, that is,

$$D = T_{malware1} + T_{malware2} + T_{rootkit1} + T_{rootkit2}.$$

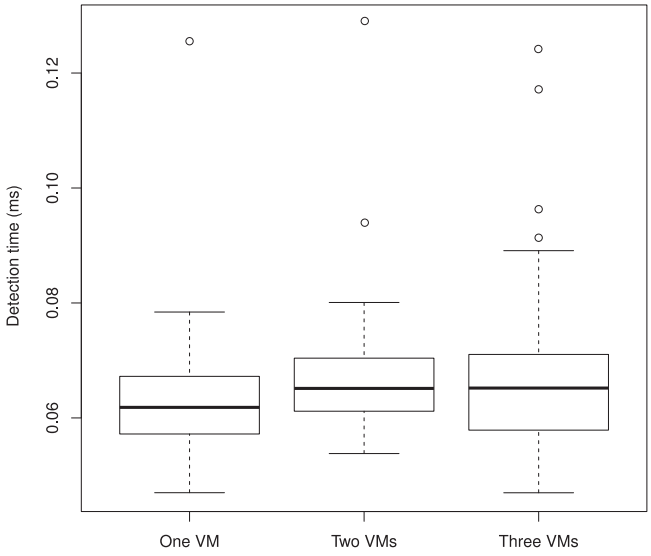


Fig. 8. Detection times of our BDSA.

After the execution of each of the malware or rootkit in each test an interval of wait for approximately two minutes is taken before the test is repeated for the next time. Given the different execution space in which the tested malware and rookits operate on the guest VM, this inter-test waiting helps to prevent the detection time from being affected by caching, typically employed by the guest OS to store frequently triggered instructions to facilitate faster execution.

The tests are repeated 10 times consecutively, with an interval of wait between the two tests. The 10 times of the consecutively repeated tests are set as one round. For each case of 1, 2, and 3 VMs 5 rounds of tests are carried out. After each round of tests, a cease of attacks for approximately two minutes is taken to remove the collected logs from the HDFS; then, it resumes a new round. The detection times are averaged across the 5 rounds to eliminate potential inconsistency of measurements. That is, in the jth test for the ith case, the detection time D_{ij} is as below:

$$D_{ij} = \left(\sum_{k=1}^{5} D_{ij}^{k} \right) /5, \tag{16}$$

where k is the index for the rounds of tests (i.e., k = 1,..., 5); i is the index for the cases of VMs (i.e., i =1,2,3); and j is the index for the consecutively repeated tests (i.e., j=1,..,10), D_{ij} is the bundled time of detecting all the 4 malware programs and rootkits in Table 5 after being launched as a pack of attacks on to the guest VMs.

Therefore, in each case of the 1, 2 and 3 VMs there are 50 measurements of the detection time which are averaged across the 5 rounds of tests for consistency purpose. The resulting 10 detection times D_{ij} are then illustrated in boxplots as shown in Fig. 8.

As expected, there is only a slight increase in the detection time as the number of guest VMs increases. When tested with a single VM, the median detection time in the boxplot is approximately 0.06 ms which increases to 0.07 ms with the introduction of a second VM. The slight increase in median detection time is because the two guest VMs run different operating systems, with one running Ubuntu 14.04 and the other running CentOS 6.5. This results in a delay time in obtaining the guest process lists from the VMs, due to the difference in processing remote command executions (`netstat`) by the guest OSes. While both guest OSes are able to process the same remote command executions, the CentOS guest OS uses as its access control module the stricter *SeLinux* (Secure Linux) instead of the relatively more flexible *AppArmor* access control module used in the Ubuntu guest OS. This meant that the *SeLinux* conducts more rigorous checks on the remote command execution before allowing it to be executed on the guest VM, causing an increase in delay as a result.

In addition, the two outlier detection times in the case of two guest VMs stems from the tendency of the guest SSH server to reset itself periodically after a certain number of connections (1,000 in this case) as a built-in mechanism to prevent against Distributed Denial of Service attacks. With the introduction of a third guest VM running Ubuntu 14.04, the median detection time increased slightly up to 0.066 ms which is relatively consistent with previous case with two guest VMs. This is due to two of the guest OSes running the more flexible *AppArmor* access control module, which enables the remote commmand executions to be executed on the guest VMs and the results to be obtained quicker. However the number of outlier detection times also increased from two in the previous case to four, reflecting the guest SSH server running in the third guest VM to periodically reset itself to prevent against DDoS attacks.

4.4 Comparisons with Existing Security Approach

In order to evaluate comparatively the performance of our BDSA approach, we have also implemented the VMI-based *Livewire* virtualization security approach based on the work by Garfinkel et al. [25].

The reason of choice behind this is that *Livewire* is similar approach to threat detection in using external monitoring of guest VM behaviour. Specifically, *Livewire* periodically polls the guest VM behavior through executing remote commands such as `ps` as well as obtaining the hardware-level information to infer the guest VM's behavior. Due to the similarity in this regard to our BDSA approach in monitoring threat in guest VMs, *Livewire* is used for the comparative evaluation.

The comparative evaluation scenario is carried out as fol- lows. First a guest VM running Ubuntu 14.04 (64-bit) on both the *Livewire* host and the host on which our BDSA approach was deployed. With the polling period being set to 1 second for both approaches, the botnet code as well as the malware code from *PacketStorm* [24] is then executed on the guest VM. The tests are run for 5 times for consistency of measurements and the average detection times for both approaches are obtained, and plotted in Fig. 9.

At a first glance, *Livewire* is able to detect attack presence faster than our BDSA approach by approximately 0.04 ms. The reason behind this is due to the ability of *Livewire* to take advantage of the *principle of locality* for threat detection. Given that *Livewire* runs on the same host on which the guest VM is located, it is able to traverse the host physical memory faster. This is evidenced by the 28 outlier detection times in the *Livewire* boxplot, and can be attributed to the delays caused by the software interrupts issued by the KVM hypervisor.

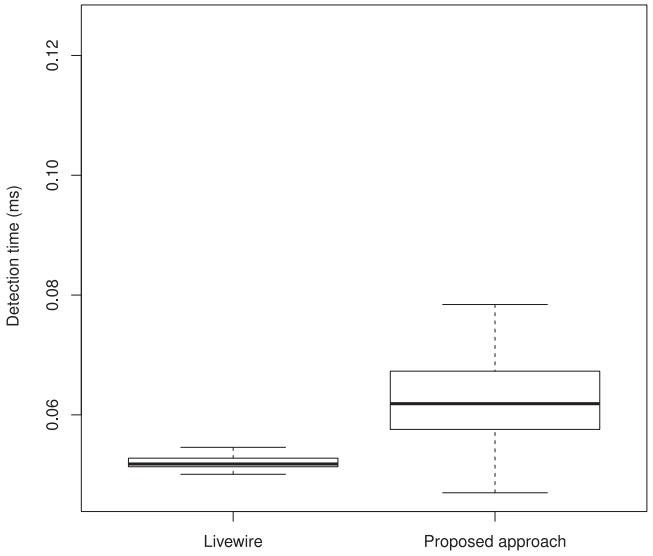


Fig. 9. Performance comparison between *Livewire* and our BDSA approach.

Given the remote process taken by our BDSA approach in detecting attack presence in guest VM, it requires establishing remote SSH connections to the guest VM to obtain its application logs. This is evidenced by the single outlier detection time of approximately 0.12 ms, which is attributed to the periodic resetting of the SSH connection from the guest VM.

In addition, the detection time in our BDSA approach also comprises the time taken by the master HPC head node as shown in Fig. 6 in distributing the *MapReduce* processing of the logs to the different HPC nodes over the network, and obtaining the correlation graphs from them for threat detection. As a result of these two factors our BDSA approach incurs a slightly higher detection time than the *Livewire* approach.

However, it is also observed that in this *best case* scenario as can be seen in the boxplot, our BDSA approach provides a faster detection time of approximately 0.02 ms against 0.04 ms of *Livewire*. In respect of scalability over the number of VMs, we have also shown that the overall detection times are fairly consistent with the increase in the number of guest VMs whereas *Livewire* has to be deployed duplicately in each VM. With all considerations put together, apparently our proposed BDSA approach provides an overall competitive performance even compared to *Livewire*.

5 Discussions in Contrast to Related Work

The use of data mining to mine attack patterns in network logs is used in *Beehive* security approach [26]. Basically, *Beehive* detects attacks through correlating logs obtained from different points within the enterprise network. First it collects logs from different points (e.g., web server logs, user logs, IDS logs, etc) within the enterprise network over a two-week period. The logs are then parsed using known network configuration details to extract 15 features for each host based on the IP addresses of websites accessed, details of the application used, violation of network policy, and changes in network flow characteristics caused by the accesses. Next, the extracted features are represented as a vector, with dimension reduction carried out by principal component analysis (PCA), and finally clustering is undertaken on the feature vectors to determine attack presence. *Beehive* is able to detect attacks from large amounts (approximately 1TB) of log data. However, it is limited in providing prompt threat quarantine and elimination due to its post-factum nature. Our BDSA approach has overcome this limitation by detecting attacks in real-time, including formation of correlation graph by assembling network and user application logs stored in HDFS, identification of potential attack paths using the MapReduce model in Apache Spark, and determination of attack presence by using belief propagation to calculate the belief of attack presence and to retrain classifiers.

Graph-based analysis is used in *BotCloud* [14] to detect botnet attacks. It involves two steps. First *Netflow* traffic logs obtained over a 48 hour period are represented as a network graph. Then, MapReduce model together with PageRank algorithm is used to identify subgraphs consisting of common source/destination traffic flows. *BotCloud* does not support real-time threat mitigation/quarantine due to its post-factum length of data. Our BDSA approach has over-come this limitation by obtaining traffic and application logs in real-time, allowing for detection of attacks in real-time and an immediate response to attack presence.

Big data analytics together with machine learning using *Netflow* traffic logs is used to detect Peer-to-Peer (P2P) botnet [27]. The approach involves three steps. First well-known botnet code is executed on the testbed and the network traffic over a 48 hour period is collected, and important packet features (e.g., time delay between packet transmissions, packet header length, etc) are extracted and stored in an Apache Hive database. With the extracted values, a MapReduce model is used to cluster common features. Finally a Random Forest classifier is trained on the clustered features using Apache Mahout. This detection scheme is limited in detecting sophisticated attacks which can adapt their communication behaviour to trick the system by mimicking normal communication flow. The logistic regression together with belief propagation in our BDSA approach is able to detect such attacks since attack presence can be dynamically determined based on changes in any of the attributes. Our BDSA approach is more insightful as it takes into account the changes both in the characteristics of the applications running within the guest VMs and in the network traffic flow.

Table 6 provides a summary of their features as well as their strengths and limitations.

6 Conclusion

In this paper, we have put forward a novel big data based security analytics approach to protecting virtualized infrastructures in cloud computing against advanced attacks. Our BDSA approach constitutes a three phase framework for detecting advanced attacks in real-time. First, the guest VMs's network logs as well as user application logs are periodically collected from the guest VMs and stored in the HDFS. Then, attack features are extracted through correlation graph and MapReduce parser. Finally, two-step machine learning is utilized to ascertain attack presence.

TABLE 6
Comparison of Security Approaches

Approach	Working Characteristics	Strengths	Limitations	Improvements by our BDSA approach
Beehive [26]	Use of PCA and clustering for attack detection. Clustering-based correlation.	Identifying previously unknown attacks.	Post-factum threat detection.	Real-time threat detection.
BotCloud [14]	Use of PageRank algorithm for C & C botnet detection. Graph-based correlation.	Identifying botnets and their connections.	Limited monitoring scope (i.e., network logs).	Wide monitoring scope (i.e., network logs and application logs).
Large-scale botnet detection [27]	Use of common packet characteristics. Clustering-based correlation.	Detecting well known botnet attacks.	Limited against mimicry/ hidden attacks.	Detecting mimicry/ hidden attacks. Wide protection scope.
Our BDSA approach	External monitoring of guest VM behavior. Graph-based correlation. Use of logistic regression with belief propagation, i.e., classifiers at individual and aggregate levels, for determination of attack presence.	Detecting both botnets and in-VM malware. Real-time automated retraining of classifiers.	Occasional latency increase due to SSH server reset by guest VMs	Real-time threat detection. Detecting mimicry/ hidden attacks. Wide monitoring scope.

Logistic regression is applied to calculate attack's conditional probabilities with respect to individual attributes. Furthermore, belief propagation is applied to calculate the overall belief of an attack presence. From the second phase to the third, the extraction of attack features is further strengthened towards the determination of attack presence by the two-step machine learning.

The use of logistic regression enables the fast calculation of attack's conditional probabilities. More importantly, logistic regression also enables the retraining of the individual logistic regression classifiers using the new attack features as they are obtained from attack detection. The use of belief propagation calculates the aggregate belief of an attack presence by taking into account the conditional probabilities with respect to individual attributes, which thereby achieves a holistic view of the guest VM's behavior.

The effectiveness of our BDSA approach is evaluated by testing it against well-known malware and rookit attacks. In all cases, it has been shown that our BDSA approach is able to detect them while maintaining a consistent performance overhead with increasing number of guest VMs at an average detection time of approximately 0.06 ms. Tested against *Livewire*, our BDSA approach incurs less performance overhead in attack detection through monitoring the guest VM's behavior.

Our BDSA approach has taken advantage of the distributed processing of HDFS and real-time ability of MapReduce model in Spark to address the velocity and volume challenges in security analytics. To tackle the veracity issue posed in zero-day attacks, our BDSA approach addresses this challenge by enforcing the on-the-fly mechanism for the retraining of logistic regression classifiers.

REFERENCES

[1] D. Fisher, "'venom' flaw in virtualization software could lead to VM escapes, data theft," 2015. [Online]. Available: https://threatpost.com/venom-flaw-in-virtualization-software-could-lead-to-vm-escapes-data-theft/112772/, Accessed on: May 20, 2015.

[2] Z. Durumeric, et al., "The matter of heartbleed," in *Proc. Conf. Internet Meas. Conf.*, 2014, pp. 475–488.

[3] K. Cabaj, K. Grochowski, and P. Gawkowski, "Practical problems of internet threats analyses," in *Theory and Engineering of Complex Sys-tems and Dependability*. Berlin, Germany: Springer, 2015, pp. 87–96.

[4] J. Oberheide, E. Cooke, and F. Jahanian, "CloudAV: N-version antivirus in the network cloud," in *Proc. USENIX Secur. Symp.*, 2008, pp. 91–106.

[5] X. Wang, Y. Yang, and Y. Zeng, "Accurate mobile malware detection and classification in the cloud," *SpringerPlus*, vol. 4, no. 1, pp. 1–23, 2015.

[6] P. K. Chouhan, M. Hagan, G. McWilliams, and S. Sezer, "Network based malware detection within virtualised environments," in *Proc. Eur. Conf. Parallel Process.*, 2014, pp. 335–346.

[7] M. Watson, A. Marnerides, A. Mauthe, D. Hutchison, and N.-ul-H. Shirazi, "Malware detection in cloud computing infrastructures," *IEEE Trans. Depend. Secure Comput.*, vol. 13, no. 2, pp. 192 205, Mar./Apr. 2016.

[8] A. Fattori, A. Lanzi, D. Balzarotti, and E. Kirda, "Hypervisor-based malware protection with AccessMiner," *Comput. Secur.*, vol. 52, pp. 33–50, 2015.

[9] T. Mahmood and U. Afzal, "Security analytics: Big data analytics for cybersecurity: A review of trends, techniques and tools," in *Proc. 2nd Nat. Conf. Inf. Assurance*, 2013, pp. 129–134.

[10] C.-T. Lu, A. P. Boedihardjo, and P. Manalwar, "Exploiting efficient data mining techniques to enhance intrusion detection systems," in *Proc. IEEE Int. Conf. Inf. Reuse Integr.*, 2005, pp. 512–517.

[11] I. Kiss, B. Genge, P. Haller, and G. Sebestyen, "Data clustering-based anomaly detection in industrial control systems," in *Proc. IEEE Int. Conf. Intell. Comput. Commun. Process.*, 2014, pp. 275–281.

[12] P. Giura and W. Wang, "Using large scale distributed computing to unveil advanced persistent threats," *Sci. J.*, vol. 1, no. 3, pp. 93–105, 2012.

[13] H. Kim, I. Kim, and T.-M. Chung, "Abnormal behavior detection technique based on big data," in *Frontier and Innovation in Future Computing and Communications*. Berlin, Germany: Springer, 2014, pp. 553–563.

[14] J. Francois, S. Wang, W. Bronzi, R. State, and T. Engel, "BotCloud: Detecting botnets using MapReduce," in *Proc. IEEE Int. Workshop Inf. Forensics Secur.*, 2011, pp. 1–6.

[15] L. Aniello, et al., "Big data in critical infrastructures security monitoring: Challenges and opportunities," *arXiv:1405.0325*, 2014.

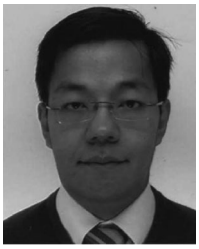
[16] L. Chen, T. Li, M. Abdulhayoglu, and Y. Ye, "Intelligent malware detection based on file relation graphs," in *Proc. IEEE Int. Conf. Semantic Comput.*, 2015, pp. 85–92.

[17] D. Kirat, G. Vigna, and C. Kruegel, "BareCloud: Bare-metal analysis-based evasive malware detection," in *Proc. 23rd USENIX Secur. Symp.*, 2014, pp. 287–301.

[18] L. Invernizzi, et al., "Nazca: Detecting malware distribution in large-scale networks," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2014, pp. 23–26.

[19] SANS, "Intrusion detection FAQ: What port numbers do well-known trojan horses use?" 2001. [Online]. Available: https://www.sans.org/security-resources/idfaq/oddports.php, Accessed on: Sep. 30, 2015.

IANA, "Service name and transport protocol port number registry," 2015. [Online]. Available: http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml, Accessed on: Sep. 30, 2015.
J. Friedman, T. Hastie, and R. Tibshirani, "Regularization paths for generalized linear models via coordinate descent," *J. Statist. Softw.*, vol. 33, no. 1, 2010, Art. no. 1.
J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Mateo, CA, USA: Morgan Kaufmann, 2014.
F. R. Kschischang, B. J. Frey, and H.-A. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 498–519, Feb. 2001.
PacketStorm, "Packetstorm security," 2013. [Online]. Available: http://tinyurl.com/qhygrsu, Accessed on: Oct. 29, 2014.
T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2003, pp. 191–206.
T.-F. Yen, et al., "Beehive: Large-scale log analysis for detecting suspicious activity in enterprise networks," in *Proc. 29th Annu. Comput. Secur. Appl. Conf.*, 2013, pp. 199–208.
K. Singh, S. C. Guntuku, A. Thakur, and C. Hota, "Big data analytics framework for peer-to-peer botnet detection using random forests," *Inf. Sci.*, vol. 278, pp. 488–497, 2014.



Thu Yein Win received his MSc degrees from the University of Bedfordshire and Asian Institute of Technology. He received his PhD degree in cloud and big data analytics security from Glasgow Caledonian University and is a lecturer in computing in the Faculty of Business, Computing & Applied Sciences, University of Gloucestershire, United Kingdom. He specializes in virtualization security and big data-based security analytics. His work examines the security issues in virtualization, with the aim of developing a security system which protects the virtualization environment against security attacks. His research interests encompass a wide range of topics in security including virtualization security, big data-based security analytics, cloud computing and information security as well as operating systems security. He is a professional member of the British Computing Society (BCS) and a member of the IEEE.

Huaglory Tianfield received the BEng (Hons.), MEng (research), and PhD Eng degrees all in electronic engineering. He is a professor of computing with Glasgow Caledonian University, Scotland, United Kingdom. since March 2001. He is extensively involved in professional activities. He is chair of IEEE Systems, Man, and Cybernetics Society Technical Committee on Cyber-Physical Cloud Systems, editor-in-chief of the *Multiagent and Grid Systems - An International Journal*, and associate editor of the *IEEE Transactions on Systems, Man, and Cybernetics: Systems*. He is director of Cloud & Data Lab. His research areas include cloud computing, cyber security, big data analytics, and Internet of Things. He has (co-)authored more than 180 research articles published in refereed journals and conferences, and is a frequent invited speaker at conferences and institutions all over the world.

Quentin Mair is a senior lecturer with Glasgow Caledonian University, Scotland, United Kingdom. He was a research assistant with the University of Stirling from 1986 to1990, where he developed software tools for the Espirit DESCARTES project (formal specification of real-time systems). Since 1991, he has been a member of academic staff in the Division of Computing, Glasgow Caledonian University, where his interests cover programming, operating systems, and distributed systems. Between 1997 and 2003, he contributed to the Framework 4 VISCOUNT and Framework 5 DIECoM. projects. His current interests include cloud and high-performance computing. He is a member of the IEEE.