The use of statistic complexity for security and performance analysis in autonomic component ensembles.

ARCHIL PRANGISHVILI, IRAKLY RODONAIA, OTAR SHONIA, TENGIZ BAKHTADZE
Faculty of Informatics
Georgian Technical University
77 Kostava Str.,Tbilisi
GEORGIA

*Abstract:-*The paper proposes a new technique for detecting malware threats in autonomic component ensembles. The technique is based on the statistic complexity metrics, which relate objects to random variables and (unlike other complexity measures considering objects as individual symbol strings) are ensemble based. This transforms the classic problem of assessing the complexity of an object into the realm of statistics. The proposed technique requires implementation of the process *X* (which generates 'healthy' flows containing no malware threats) and objects generated by the actual (possible infected) process *Y*. The component flows files are used as objects of the processes *X* and *Y*. The result of the proposed procedure gives us the distribution of probabilities of malware infection among autonomic components. The possibility to use the results obtained to perform quantitative probabilistic verification and analysis of ASEs using the probabilistic model checking tool PRISM is demonstrated.

Keywords: cloud computing, autonomic component, autonomic ensemble, complexity measure, statistic complexity, traffic flows, malware, response time, performance, service level agreement.

1 Introduction

The problem of anomaly detection in autonomic component ensembles was considered in [1,2], where the following problem was set. A singleton application currently runs on one of the VMs at an Datacenter. During the session the application experiences consistently high CPU load. This increase may be caused either by legitimate traffic overload or by coordinated DDOS attacks launched against the PaaS provider. The latter might be wrongly assumed to be legitimate requests and resources would be scaled up to handle them. This would result in an increase in the cost of running the application (because provider will be charged by these extra resources) as well as in violation of SLA (due to increased response times). Hence, it is necessary to distinguish between these two cases, the earlier this distinction is made, the higher is the degree of protection of the application from failure and poor performance.To provide this protection, the following security measures were suggested. The traffic flows through the VM_i had to be analyzed using ***statistic*** complexity metrics. During the session the constant monitoring of the metric (by the special probe implemented in the separate module), along with measure of CPU load and available memory size, was being executed. If the traffic satisfied some pre-formulated criteria (indicated that there exist serious DDOS attack threats) then the application rapidly migrated to some other VM_j.

The techhnique described in [1, 2] implemented Kolmogorov complexity metrics to reveal possible malware attacks and had to deal only with DDOS attacks. Despite its usefulness, Kolmogorov complexity does not capture the intuitive notion of complexity very well. For example, random strings without any regularities, say, strings that are constructed bitwise by repeated tosses of a fair coin, have very large Kolmogorov complexity. However, those strings are not "complex" from an intuitive point of view — those strings are completely random and do not carry any interesting structure at all. Many approaches have been suggested to define some complexity measure that is closer to the intuitive notion of complexity and overcomes the difficulties of Kolmogorov complexity. For example, Kolmogorov complexity is based on algorithmic information theory considering objects as individual symbol strings, whereas the measures *effective measure complexity* (EMC), *excess entropy*, *predictive information, etc.*, relate objects to random variables and are *ensemble* (that is, set of interrelated objects –symbol strings) based.

The Kolmogorov complexity measures *M* assigns a complexity value to each individual object

x' under consideration. Let's denote it as $C_M(x')$. It is assumed that x' corresponds to a string sequence of a certain length and its components assume values from a certain domain. In [3] *statistic complexity* that is not only different to all other complexity measures introduced so far, but also connects directly to statistics, specifically, to statistical inference, was introduced. More precisely, a complexity measure with the following properties is introduced. First, the measure is bivariate comparing two objects, corresponding to pattern generating processes, on the basis of the *normalized compression distance (NCD)*[4] with each other:

$$NCD(x,y)=\frac{C(xy)-\min\{C(x),C(y)\}}{\max\{C(x),C(y)\}},$$

where *C(x)* denotes the compression size of string x and *C(xy)* the compression size of the concatenated stings *x* and *y*.

Second, this measure provides the quantification of an error that could have encountered by comparing samples of finite size from the underlying processes. Hence, the statistic complexity provides a statistical quantification of the statement '*X* is similarly complex as *Y* '. This implies that a fundamental complexity measure needs to be bivariate, C(*X, Y*), instead of univariate comparing two processes *X* and *Y*.

Next, the desirable property of any complexity measure is : a complexity measure should quantify the uncertainty of the complexity value. As motivation for this property we just want to mention that there is a crucial difference between an observed object x' and its generating process *X*. If the complexity of *X* should be assessed, based on the observation x' only, this assessment may be erroneous. This error may stem from the limited (finite) size of observations. Also, the possibility of measurement errors would be another source of wrong assessment.

Based on these considerations, the statistic complexity measure, suggested in [3], is defined by the following procedure:

1. Estimate the empirical distribution function $\hat{F}_{XX}$ of the normalized compression distance from n_1, $S_{X,X}^{n_1}=\{x_i=NCD(x',x'')\,|\,x',x''\prec X\}_{i=1}^{n_1}$, from objects x' and x'' of size m generated by process X (here '$\prec$' means 'is generated by *X*')
2. Estimate the empirical distribution function $\hat{F}_{XY}$ of the normalized compression distance from n_2, $S_{X,Y}^{n_2}=\{y_i=NCD(x',y')\,|\,x'\prec X,y'\prec Y\}_{i=1}^{n_2}$ from objects x' and y' of size m generated by two different processes X and Y
3. Determine $T=\sup_x\left|\hat{F}_{X,X}(x)-\hat{F}_{X,Y}(x)\right|$ and $p=Prob(T\le t)$
4. Define $C_S(S_{X,X}^{n_1},S_{X,Y}^{n_2}\,|\,X,Y,m,n_1,n_2):=p$ as *statistic complexity*

This procedure corresponds to a two-sided, two-sample Kolmogorov- Smirnov (KS) test based on the normalized compression distance [4] obtaining distances among observed objects.

The statistic complexity corresponds to the p-value of the underlying null hypotheses, $H_0:F_{XX}=F_{XY}$, and, hence, assumes values in [0,1]. The null hypothesis is a statement about the null distribution of the test statistic $T=\sup_x\left|\hat{F}_{X,X}(x)-\hat{F}_{X,Y}(x)\right|$, and because the distribution functions are based on the normalized compression distances among objects x' and x'', drawn from the processes *X* and *Y*, this leads to a statement about the distribution of normalized compression distances. Hence, verbally, H_0 can be phrased as "on average, the compression distance of objects from *X* to objects from *Y* equals the compression distance of objects only taken from *X*". If the alternative hypothesis, $H_1:F_{XX}\neq F_{XY}$ is true, this equality does no longer hold implying differences in the underlying processes *X* and *Y*, leading to differences in the NCDs

Applied to the problem of finding malware threats in the flows between autonomic components CP_i [1, 2], the above procedure will look as follows. For *each autonomic component* (AC) of the *autonomic-component ensembles* (ACEs) the processes *X* and *Y* are considered as the processes generating objects represented in the form of strings. The strings, in turn, represent traffic flows through these autonomic components. The specific ways of how flows are transformed into strings are considered later in the paper. The process *X* ('training process') is the process generating flows in the conditions when there are no malware threats. So, objects (strings) generated by the process *X* are 'healthy' (they do not contain any patterns of malware). These strings have to be generated preliminary (before actual workload on an autonomic components ensemble). Some fraction of objects (string) have to be generated for situation

with unusual (but not malicious) behavior. For randomly taken pairs x' and x'' (the amount of such pairs is n_1) of the generated strings the metric NCD(x' , x'') is calculated . The size of samples n_1 has to be sufficient to account for various possible situations and conditions that may occur in the specific autonomic ensemble under consideration. Then the empirical distribution function $\hat{F}_{XX}$ is being built and stored to the specific place .

When the ensemble starts actual operation (receives workload), the process Y ('production process') generates objects (strings) y', which represent actual current traffic between ensemble's components. Some of these objects may contain malware patterns. The sample of the size n_2 of objects x' (generated preliminary by the 'training process' X) and objects y' is being created and the metric NCD (x' , y') is calculated for each pair. Then the empirical distribution function $\hat{F}_{XY}$ is being built. Now , by applying the steps 3 and 4 of the above procedure, the values of the *statistic complexity* for *each autonomic component* can be computed.

However, it is well known that the *p*-value is *not* the probability that the null hypothesis is true, nor is it the probability that the alternative hypothesis is false. To calculate the probability that the null-hypothesis is true, given some data we have collected, we need to use Bayes' formula. Cohen [9] shows how the posterior probability of the null-hypothesis, given a *statistically significant* result (the data), can be calculated based on a formula that is a poor man's Bayesian updating function. Instead of creating distributions around parameters, his approach simply uses the *p*-value of a test (which is related to the observed data), the power of the study, and the prior probability the null-hypothesis is true, to calculate the posterior probability H_0 is true, given the observed data. Before we look at the formula, some definitions:

$P(H_0)$ is the prior probability (P) the null hypothesis (H_0) is true.

$P(H_1)$ is the probability (P) the alternative hypothesis (H_1) is true. Since we'll be considering only a single alternative hypothesis here, either the null hypothesis or the alternative hypothesis is true, and thus $P(H_1) = 1- P(H_0)$. We will use $1-P(H_0)$ in the formula below.

$P(T|H_0)$ is the probability (P) of the data T, which was obtained by the KS procedure

$$T = \sup_x \left| \hat{F}_{X,X}(x) - \hat{F}_{X,Y}(x) \right|$$

given that the null hypothesis (H_0) is true. In Cohen's approach, this is the *p*-value of a study.

$P(D|-H_0)$ is the probability of the data (a significant result), given that H_0 is *not* true, or when the $P(T|-H_0)$ is the probability of the data (a significant alternative hypothesis *is* true. This is the statistical power of a study.

$P(H_0|T)$ is the probability of the null-hypothesis, given the data. This is our posterior belief in the null-hypothesis, after the data has been collected. According to Cohen [9], it's what we really want to know. People often mistake the p-value as the probability the null-hypothesis is true.

$$P(H_0|T) = \frac{P(T|H_0)P(H_0)}{P(T|H_0)P(H_0) + P(T|-H_0)(1-P(H_0))}$$

In the numerator, we calculate the probability that we observed a significant *p*-value when the null hypothesis is true, and divide it by the total probability of finding a significant *p*-value when either the null-hypothesis is true or the alternative hypothesis is true. The formula shows that the lower the *p*-value in the numerator, and the higher the power, the lower the probability of the null-hypothesis, given the significant result you have observed.

Assuming H_0 and H_1 are a-priori equally likely, the formula simplifies to:

$$P(H_0|D) = \frac{0.99}{0.99 + Type\ 2\ error\ rate}$$

(provided that the level of significance α is 0.01 (less than one in a hundred chance of being wrong); here we have chosen this low threshold value (usually it is 0.05) in order to emphasize the importance of guaranteed revealing of malware threats)

Therefore, the obtained numerical value of the statistic complexity can be interpreted in the following sense: in the current conditions the flows of packets through the given autonomic component cannot be regarded as complex flows (with the probability equal to $1-P(H_0|D$). That is, the flows may contain some patterns (indicating the possible presence of some malware threats) with the probability $P_{infect} = 1-P(H_0|D)$. In our approach we assume that the probability $P_{infect} \geq 0.6$

It should be pointed out that in production conditions (when the ensemble is under actual workload) the sample size n_2 cannot be determined in advance. This size depends on actual working

conditions: traffic intensity, frequency of creation of objects (strings), actual hardware indices (CPU load, available memory, etc.). As a rule, the number n_2 is less than the number n_1. This fact can somewhat decrease the precision of the metric, but it requires additional technical consideration. In general, the statistic complexity has the very desirable property that the power reaches asymptotically 1 when $n_1 \to \infty$ and $n_2 \to \infty$. This means, for infinite many observations the error of the test to falsely accept the null hypotheses when in fact the alternative is true becomes zero. Formally, this property can be stated as $p \to 0$ for $n_1 \to \infty$ and $n_2 \to \infty$.

Finally, note that despite the fact that statistic complexity is a statistical test, it borrows part of its strength from the NCD and, respectively, Kolmogorov complexity on which this is based on. Hence, it unites various properties from very different concepts.

2 Application of statistic complexity to autonomic components ensembles.

In the proposed approach to anomaly detection in autonomic component ensembles, an attempt to deal with wide range of malware threats has been made (unlike the techhniques described above and in [1, 2], which had to deal only with DDOS attacks).

In autonomic cloud computing datacenters can be considered as autonomic-component ensembles (ACEs) and be represented by constructions of SCEL (Software Component Ensemble Language), a kernel language for programming autonomic computing systems [1, 5, 6]). Each (virtual) machine is running one instance of the Cloud Platform called Cloud Platform instance (CP_i).Each CP_i is considered to be a service component. Multiple CPs communicate over the Internet (IP ptotocol), thus forming a cloud and within this cloud one or more service component ensembles. The notions of autonomic components (ACs) and autonomic-component ensembles (ACEs) [5,6] have been put forward as a means to structure a system into well understood, independent and distributed building blocks that interact in specified ways.

The process part of a component (Fig.1) is split into an *autonomic manager* controlling execution of a *managed element.* The autonomic manager monitors the state of the component, as well as the execution context, and identifies relevant changes that may affect the achievement of its goals or the fulfillment of its requirements. It also plans adaptations in order to meet the new functional or non-functional requirements, executes them, and monitors that its goals are achieved, possibly without any interruption**.** A managed element can be seen as an empty “executor" which retrieves from the knowledge repository the process implementing a required functionality *id* and bounds it to a process variable *Z*, sends the retrieved process for execution and waits until it terminates. Also actual parameters for the process to be executed can be stored as knowledge items and retrieved by the executor (or by the process itself) when needed.

Items containing processes or parameters can be thought of as *awareness* data. Autonomic managers can add/remove/replace these data from the knowledge repositories thus implementing the adaptation logic and therefore changing the managed element's behavior. The autonomic manager can also add a new service or even remove an existing one.

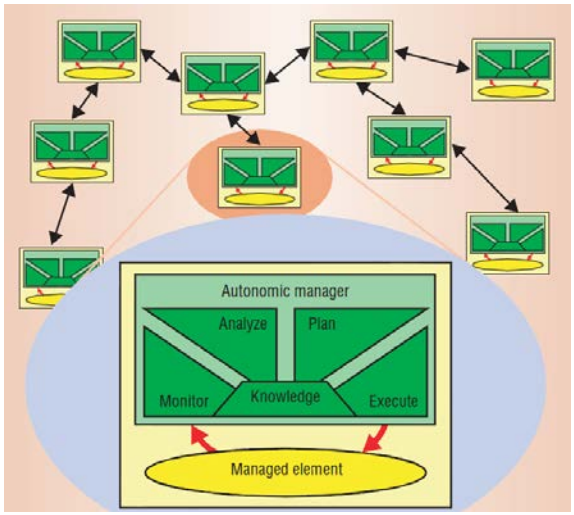

Fig.1 Functional description of a component

In our approach the notions of *netflows*, their *informational-theoretical metrics* and components’ *autonomic manager* are essentially leveraged. A network *flow* can be defined in many ways. In a general sense, a flow is a series of packets with some attribute(s) in common. Each packet that is forwarded within a router or switch is examined for a set of IP packet attributes. These attributes are the IP packet identity or fingerprint of the packet and determine if the packet is unique or similar to other packets. All packets with the same source/destination IP address, source/destination ports, protocol interface, and class of service are grouped into a flow and then packets and bytes are labeled. This methodology of fingerprinting or determining a flow is scalable because a large amount of network information is condensed into a database of netflow information called the netflow cache.

A *netflow-enabled device (netflow exporter*: router or switch) (see the Fig.2) sends to the *netflow collector* single flow as soon as the relative connection expires.This can happen when 1) when TCP connection reaches the end of the byte stream (FIN flag or RST flag) are set; 2) when a flow is idle for a specific timeout; 3) if a connection exceeds long live terms (30 minutes by default). Packets captured by the netflow collector are stored to a *flow storage* . In our approach the duration of each flow's formation time is unknown in advance and actually is defined by relevant collectors on the basis of the selected connection expiration time criteria.

Flows accumulated at the flow storage, are then subdivided into *component flows*. That is, flows which have the component's IP address as a destination address are grouped and sent to the corresponding component (more exactly, to the *autonomic manager* of a component - these flows are marked with blue arrows on the Fig.2).

After receiving their destined flows, the

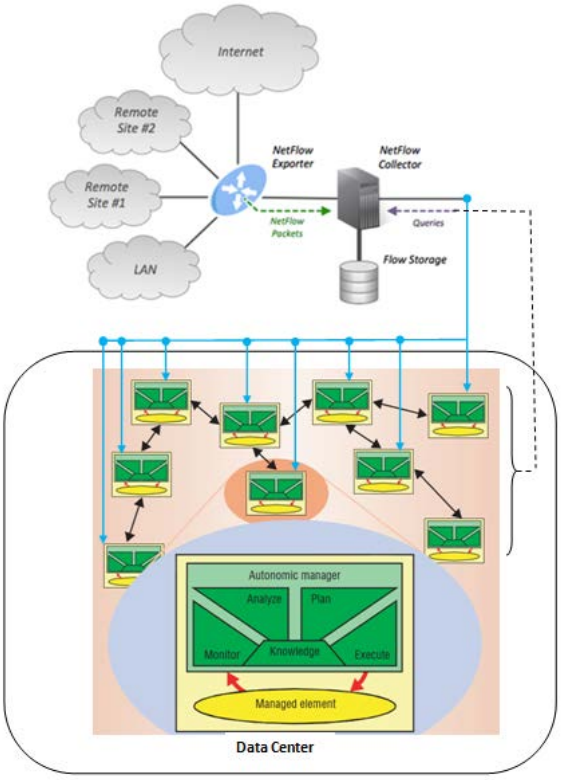

Fig.2 Interaction between netflow devices and autonomic components

component's autonomic manager can start the processing in order to reveal the abnormal behavior of flows in accordance with the following technique.

Application for collecting and processing NetFlow statistics are defined below (Figure 3):

Once the collector populates the raw file, the file is passed on to the second component in the system, which is called an aggregator. The aggregator receives the file from the collector and processes it using predefined information from the database. The data thus processed (aggregated) is stored in the database.

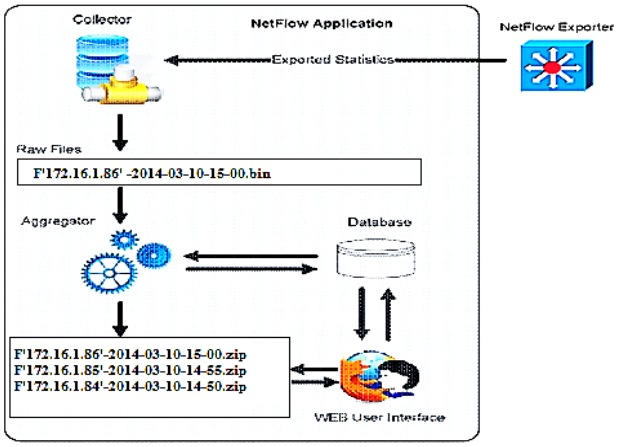

Fig. 3: Components of the NetFlow system for analysis of the statistics

The user interface is a web application that enables us to obtain information on the status of the network, based on the data aggregated in the database. If it is necessary to get more detailed information about a specific communication, the user may open the relevant raw file via the web and filter it according to the desired criteria. The location of the device collecting NetFlow statistics depends on the architecture of the network itself. The amount of NetFlow information exported by network devices is directly dependent on the amount of traffic passing through that device (exporter). Experience has shown that the amount of NetFlow traffic does not exceed 1% of the total amount of traffic through the network, so the "distance" between the server (collector) and the network device exporting the data (exporter) is not relevant. The accessibility and the security of the server are the more important parameters

In the proposed approach the different files with the particular titles (relevant to the concrete SCP_i's IP addresses) to store component flows are used. For example, for the component flow to the SCP_i with IP address 172.16.1.86, occurred on 2014/03/16 at 15:00, the files with titles F'171.16.1.86'-2014-03-10-15-00.bin and the F'171.16.1.86'-2014-03-10-15-00.zip will be created.

If we look at known threats in data networks from point of unwanted traffic, we can separate the following groups [7]:

1. Denial of service attacks.
2. Port scans and remote vulnerability searching and virus spread.
3. P2P files exchange networks.
4. Email spam and web popup.
5. Open resources misuse (open DNS, open mail relay, open proxy, Trojan horse, etc.)

In our approach we observe the traffic flow attributes provided by "The Network Intrusion Dataset" of the provided by the "Information Exploration Shootout (IES)" project. The dataset consists of four files, the first is believed to be free of attacks and the rest contain attacks that were simulated and stored, each file containing instances of a single different attack behavior [8, 11]:

- Source/destination IP address and port number

To measure changes in IP address and port number space we observe a value of Shannon entropy related to these attributes (entropy is used to capture the degree of dispersal or concentration of the distributions for traffic attributes). Entropy values are calculated for separate component flows files (obtained by using the utility *nfdump,)* . Different AMs (Autonomic Manager) use various time periods length (see connection expiration time criteria above). The following network variables are used for each component flows files: entropy of source IP address, entropy of destination IP address, entropy of destination port number, entropy of source port number . Duration attributes of each component flow time are different and depend on the traffic conditions and selected connection expiration time criteria

- Number of bytes and packets

These values are: bytes received by a host, bytes sent by a host, packets received by a host, packets sent by a host. Again, duration attributes of each component flow files are different

- TCP flags

The attribute TCP_FLAG - a difference between number of SYN packets sent and RST and FIN packets received - is measured in the proposed approach. In normal conditions, in long time observation we should get the mean value of TCP_FLAG near zero. Intrusive actions like system scanning, DoS attacks, may cause the temporal distortion of the mean value of TCP_FLAG

- Duration of the connection

During various types of attacks, this value will be affected and so an anomaly may be detected. For example, worm infection will generate a large number of connections with similar duration. We simply use the value of connections' duration attribute contained in the given component flow file.

- Communication Patterns

Fan-in is the number of nodes that originate data exchange with the current CP_i, while *Fan-out* is the number of hosts to which CP_i initiates conversations. The above patterns are invariant during most time of normal system activity or change in a predictive way. But while attack appears they will change significantly.

As one can see, the component flows files contain the same volume of information (they contain the same amount of attributes of the same size). Hence, we can assume that the size *m* of a component flow file represents the object (in terms of the statistic complexity procedure) of size *m.* In general, component flows files are regarded as objects $x', x'', x''', \ldots\ldots\ldots$ generated by the process *X* ('training process') and $y', y'', y''' \ldots\ldots\ldots\ldots$, objects generated by the process *Y* ('production process').

As it was described, the proposed procedure requires implementation of the 'training process' *X* (which generates 'healthy' flows containing no malware threats) before starting real 'production' (real-time) process *Y*. In order to decrease overheads, this process is executed just once with as large value of the sample size n_1 as it is possible. The obtained results (the empirical distribution function $\hat{F}_{XY}$) is stored to each CP_i which can run applications subsequently. When applications are executed on the CPs, the objects $y', y'', y''' \ldots\ldots\ldots\ldots$, (corresponding component flows files) are created and the empirical distribution functions $\hat{F}_{XY}$ are calculated on each CP_i . Then, according to the steps 3 an4 of the procedure, the value of statistic complexity for each autonomic component is calculated.

The result of the proposed procedure gives us the distribution of probabilities of malware infection among autonomic components of the datacenter.

As it was said above the probability $P_{infect} \geq 0.6$ can be practically regarded as a serious malware threat. In this condition the immediate migration of the application from the VM (where the application is being run currently) to another VM (which is to be selected by using the ensemble's components autonomic managers' knowledge base and issuing the special SCEL statement **qry**) is required.The corresponding procedure of SCEL language is described below.

The traffic flows through the node (CPi) has is being analyzed using the statistic complexity metrics. If the probability P_{infect} , associated with the statistic complexity, becomes equal or more than 0.6. the application has to migrate from the CPi

where it was running to another CPi (which may belong to the same ensemble or other ensemble). A new CPi must be found according to some requirements: probability P_{infect} and CPU load must be rather low, integrated hardware index (which includes such indicators as processor speed, available memory, available disk space, number of cores, etc) must correspond to the application resource requirements (they are published in the interface of the CP_i where the application is running). If the required CP_i is found, the application has to migrate there as soon as possible and stop its running on the "old" CP_i. The process formally can be described in SCEL statements. We assume that, other than id, the interfaces and provide the attributes "ComplexityLevel", "CPULoad" and "Memory" stores a context information, updated by the underlying infrastructure (usually, from the firewalls, gateways or special probes) and are

The CP_i where the application is running is the SCEL component: I[K,Π,AM[ME]]. The autonomic manager AM is defined as follows:

AM ≜ $P_{ComplexityMonitor}$ [$P_{CPULoad}$ [$P_{migrateCP}$]]
$P_{ComplexityMonitor}$ ≜ |**ry**("*ComplexityLevel*", "high") @ self.
get("*ComplexityHigh*", false) @self.
put(("*ComplexityHigh*", true) @self.
qru("*ComplexityLevel*","low")@self.
get("*ComplexityHigh*",true)@self.
put("*ComplexityHigh*",false)@self.$P_{ComplexityMonitor}$
P_{CPULoa} ≜ **qry**("*CPUloadLevel*","low")@ self.
get("*CPULow*", false) @self.
put(("*CPULow*", true) @self.
qru("*CPUloadLevel*","high")@self.
get("*CPULow*",true)@self.
put("*CPULow*",false)@self. $P_{CPULoad}$
$P_{MigrateCP\,i}$ ≜ |**ry**("*Cloud service*", ?X)@ self
get("*Cloud service_args*", ?*sessionId*, ?*memoryValue*, ?*CPUValue*) @self.
/* retrieving from the knowledge repository the process
/*implementing a required functionality id and
/*bounding it to a process variable X */
/* searching an item *c* among components belonging
/*to the ensemble identified by predicate Ω */
***qry**("*CPiId*", ?c) @Ω.
/* storing actual parameters of the process to be
/*executed /*in the found component c : moving from
/*"old" VM to /*newlu found VM */
put("*Cloud service*", ?*sessionId*, ?*memoryValue*, ?*CPUValue*)@c
get("*Cloud service*", "*sessionId*", "*terminated*") @self.
/* removing the process from the knowledge
/*repository of 'old' CPi */
get("*Cloud service*", "*sessionId*", X) @self.nil
/* eliminating the process in 'old' CPi */

Here the predicate Ω is determined as follows:

Ω (I) =(I ComplexityLevel="High")
∧ (I.CPULoad <75) ∧ (I .Memory>=500))

and is used for group-oriented communication in the action qry("CPiId", ?c) @. This predicate defines the ensemble of components which publish in their interfaces attributes "ComplexityLevel", "CPULoad" and" Memory"along with relevant values. We assume that these attributes are provided by the interface of each component and obtain dynamically updated values from corresponding probes (sensors) as a result of constant monitoring (sensing) of the computing environment. We assume also that the attribute "ComplexityLevel" gives an indication in the range [0:1] of the statistic complexity level of data flow through the ensemble, the attribute "CPULoad" – in the range [0:100], the attribute "Memory" – in the range [0:1000]. In this context the meaning of the predicate Ω is as follows: find a component CPi (or components) where the "ComplexityLevel" is high (that is, a value of the stastistic complexity that refers to the $P_{infect} \geq 0.6$), "CPULoad" is less than 75 and available memory index "Memory" is more than 500.

Independently of the service component on which the cloud service is being executed ("old ' CPi or newly found "receiver of migrated service" CPi) the SCEL statements which describe the process P_s executed by the managed element ME are as follows:

P_s ≜ **get** ("*Cloud service*", ?*sessionId*, ?*memoryValue*, ?*CPUValue*)@self.
get("*CPUload*", ?L) @self.
/* L is a current CPUload of the component
get("*memory*", ?M) @self.
/* M is a current allocated memory
put("*CPUload*", (L+ *CPUValue*))@self.
put("*memory*", (M- *memoryValue*))@self.
P_s [X(*sessionId*, *memoryValue*, *CPUValue*)]
/* the new process (additionally to the already running process P_s), having actual parameters *sessionId*, *memoryValue*, *CPUValue,* starts */

The run-time Java implementation of the SCEL formal code (expressed in jResp environment) has been developed. The main classes of jResp, corresponding to the above SCEL code, are as follows: *ServiceComponent, CloudService, ServiceCaller, RequestHandler, OfferAgent* . The scenario, described above, is realized by means of jResp classes *Scenario* and *Main* (the structure of datacenter):

It should be pointed out that detection of malware threats and consequent migration are being executed in real-time scale and thus minimize damage from possible malware threats. This also contributes to maintaining the required SLA.

The time of migration must be taken into account when determining the response time. In general, streams of requests generated by each client (application) may be decomposed into a number of different VMs. In case of more than one VM serving the i^{th} client, requests are assigned probabilistically. The response time of a VM (placed on server *j*) is computed according to the Pollaczek-Khinchin formula (M/G/1 queueing system) :

$$\overline{R} = \frac{\overline{x}}{1-\rho} + \frac{\overline{x}\rho}{1-\rho}\frac{C_v^2 - 1}{2}$$

where $\overline{x}$ - average service time for a client (application), σ - standard deviation for average service time, $C_v^2 = \frac{\sigma^2}{\overline{x}}$ - coefficient of variation of the service time , λ- arrival rate of a client (application), μ – service rate of a client, $\rho = \lambda\overline{x} = \lambda / \eta < 1$ - server utilization

A VM unit is defined as the basic unit of virtual resource, which is associated with a set of physical resources such as CPU time, main memory, storage space, electricity etc. In real cloud systems, any virtual resource a customer can apply should be a multiple of the VM unit.

Migrating a VM between servers causes a downtime in the client's application. Duration of the downtime is related to the migration technique used in the datacenter. The downtime also is the function of the link speed and VM memory size.

Let's assume that an application *i* had to migrate n_i times during its execution cycle. We introduce the following notations:

n_i - amount of migration of the *i*-th application during its execution cycle;

m_k - the number (index) of VM (CP) on which the application runs in *k*-th migration period;

SC_{ip} - probability of migration (equal to P_{infect}, computed in the procedure of determining statistic complexity described above) obtained for the *i*-th application running on the *p*-th VM in the given time period

$\overline{R}_{ij}$ - - response time for the application i running on the *j*-th VM in the given period

Then the formula (1) must be updated by adding the term representing the expected downtime of the VM_{ij}:

$$\begin{cases} \overline{R}_{im_1} & if\ \ n_i = 1 \\ \sum_{k=1}^{n_i} (SC_{im_k} * (\overline{R}_{im_k} + DT_{im_k}(LinkSpeed))) & otherwise \end{cases}$$

As numerous simulation experiments executed with the use of the simulation systems CloudSim and OPNET Modeler show, the obtained estimation of response times is much closer to the actual response times (observed in real operational conditions) and thereby contributes to maintaining the required SLA

3. Quantitative verification based on the statistic complexity estimate.

Model checking represents a formal technique for verifying whether a system satisfies its specification.The technique involves building a mathematically-based model of the system behaviour, and checking that system properties specified formally in a temporal logic hold within this model.The result is based on an exhaustive analysis of the state space of the considered model - a characteristic that sets model checking apart from complementary techniques such as testing and simulation.

Quantitative verification techniques [10] are implemented within PRISM , a probabilistic model checker, which provides direct support for discrete-time Markov chains (DTMCs), Markov decision processes (MDPs) and continuous- time Markov chains (CTMCs). PRISM puts particular emphasis on *quantitative* properties. For example, PCTL (and CSL) allow expression of logical statements such as "the probability of eventual system failure is less than p", denoted $P_{<p}$ [F fail]. In PRISM, it is more typical to simply ask "what is the probability of eventual system failure?", expressed as $P=?$ [F fail]. The property specification language also allows numerical values such as these to be combined in arithmetic expressions, allowing more complex measures to be expressed.

Proceeding from the distribution of migration probabilities for all ACs (equal to P_{infrct} , computed for each AC and based on the procedure of determining statistic complexity described above), parameters of applications being performed by each AC (such as service times, response times (including migration times), etc.) which are being computed and stored in the repositories of corresponding autonomic managers of an AC) we can perform *quantitative* probabilistic analysis of security, availability and performance of ASEs.

The probability that the system is in a particular state of interest, either at a specific time instant (transient) or in the long-run (steady-state) can be expressed using the P and S operators, respectively. Consider, for example, an ACE whose state can be classified as either "operational" or " failed" and assume that *oper* is a Boolean variable whose value is true if system is operational. Alternatively, *oper* could be a more complex expression over state

variables expressing this fact. The following properties describe the availability of the system:

- $P=_?$ [$F^{[t;t]}$ oper] – "the instantaneous availability of the system, i.e. the probability that it is operational at time instant *t*";
- $S=_?$ [oper] – "the long-run availability of the system, i.e. the steady-state probability that it is operational".

In particular, the use of PRISM and data obtained by implementation of the above techniques allows us to define and obtain answers for the following types of questions:

- $P=_?$ [$F[^{0;600}]$ migrate $_{ACi}$] – "the probability that component AC_i will migrate within 10 minutes"
- $P=\{$"responseTime"$\}=_?$ [$RS_{ACi} > RS\_SLA$] – "the probability that response time of the component AC_i will be more than the response time required by SLA term"

4. Conclusions

In the paper we presented a new technique for detecting malware threats in autonomic component ensembles. The technique is based on the statistic complexity metrics. Unlike the Kolmogorov complexity, which is based on algorithmic information theory considering objects as individual symbol strings, the statistic complexity relate objects to random variables and are ensemble based. It is a bivariate measure that compares two objects, corresponding to pattern generating processes, on the basis of the normalized compression distance with each other. Besides, this measure provides the quantification of an error that could have been encountered by comparing samples of finite size from the underlying processes. The approach transforms the classic problem of assessing the complexity of an object into the realm of statistics. The statistic complexity is applied to the problem of detecting malware threats in autonomic component ensembles. The proposed procedure requires implementation of the 'training process' *X* (which generates 'healthy' flows containing no malware threats) and objects generated by the actual (possible infected) process *Y* ('production process'). The component flows files are used as objects of the processes *X* and *Y*. The result of the proposed procedure provides the distribution of probabilities of malware infection among autonomic components of the datacenter. The proposed procedure of detecting malware threats and consequent migration are being executed in real-time. This also contributes to maintaining the required SLA. The possibility to use the results obtained to perform quantitative probabilistic verification and analysis of ASEs using the probabilistic model checking tool PRISM is demonstrated.

REFERENCES:

[1]. A. Prangishvili, O. Shonia, I. Rodonaia, V.Rodonaia. Formal security modeling in autonomic cloud computing environment . WSEAS / NAUN International Conferences, Valencia, Spain, 2013

[2]. A.Prangishvili, O.Shonia, I.Rodonaia, M. Mousa. Formal verification in autonomic-component ensembles, WSEAS / NAUN International Conferences, Salerno, Italy, 2014

[3]. F. Emmert-Streib. Statistic Complexity: Combining Kolmogorov Complexity with an Ensemble Approach, Queen's University, Belfast, United Kingdom, 2010

[4]. Cilibrasi R, Vitanyi P. Clustering by compression. IEEE Transactions Information Theory 51: 1523–1545. 2005

[5]. ASCENS, P.: http://www.ascens-ist.eu/ (2010)

[6]. Rocco De Nicola, Michele Loreti, Rosario Pugliese, Francesco Tiezzi. "SCEL- a Language for Autonomic Computing". ASCENS project , Technical report, January 2013

[7]. Unwanted traffic identification problems. Martins Ekmanis. Department of Telecommunications, Riga Technical University, Azenesiela 12, LV-1048, Riga, Latvia

[8]. G. Kołaczek, K. Juszczyszyn. Attack pattern analysis framework for multiagent intrusion detection system. International Journal of Computational Intelligence Systems, Vol.1, No. 3 (August, 2008), 215 - 224

[9]. J. Cohen, The earth is round (p <.05). American Psychologist, 49, 997-1003), 1994

[10]. M. Kwiatkowska, G. Norman, D. Parker. PRISM: Probabilistic Model Checking for Performance and Reliability Analysis. Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, 2010

[11]. Information exploration shootout project. http://ivpr.cs. uml.edu/shootout/about.html. January 2007.