2017 IEEE 9th International Conference on Cloud Computing Technology and Science

Continuous location validation of cloud service components

Philipp Stephanow, Mohammad Moein, Christian Banse
Fraunhofer Institute for Applied and Integrated Security (AISEC), Munich, Germany
{firstname.lastname}@aisec.fraunhofer.de

***Abstract*—Continuously, i.e. automatically and repeatedly checking at what geographical locations cloud service components are hosted aims at validating that the cloud service satisfies regulatory and other compliance requirements. Yet continuous validation is challenging since it requires location techniques to adapt to network changes over time. In this paper, we present adaptive location classification, an approach to continuously validate the location of cloud service components. Our approach combines supervised and unsupervised learning techniques and is capable of adapting to network changes over time. We demonstrate the feasibility of our approach by presenting experimental results where we continuously validate the locations of cloud service components hosted at 14 different locations of the AWS Global Infrastructure.**

***Keywords*-cloud services; geographical location; classification**

I. Introduction

Cloud service components such as networks, compute resources, and storage are usually virtualised and migrating these virtual components from one geographical location to another is a standard feature most cloud service providers such as Amazon Web Service (AWS) and Google Cloud Platform offer to their customers. Examples include migrating IaaS, i.e. virtual machines, as well as PaaS applications such as Relational Database Services (e.g., AWS RDS) from one geographical location to another. Yet altering the location of a cloud service component may violate regulatory or compliance requirements the cloud service claims to satisfy. Therefore, it is essential to continuously, i.e. automatically and repeatedly check whether the location of a cloud service component is *valid*, that is, if it is in agreement with its expected location.

Consider, for example, the control *UP-02 Jurisdiction and data storage, processing and backup locations* of the Cloud Computing Compliance Controls Catalogue (C5) issued by the German Federal Office for Information Security (BSI) [1] requiring that *[..] Data of the cloud customer shall only be processed, stored and backed up outside the contractually agreed locations only with the prior express written consent of the cloud customer.".* In our terminology, *contractually agreed location* of UP-02 are *valid* locations.

While not containing controls explicitly addressing the geographical location of a cloud service's components, another example is the control *AIS-01 Application & Interface Security Application Security* of the CSA Cloud Control Matrix [2] requiring that *"Applications and programming interfaces (APIs) shall [..] adhere to applicable legal, statutory, or regulatory compliance obligations.".* The European Union (EU) Data Protection Directive contains a set of regulations where the location of cloud services' components can become important because it restricts personal information from flowing from EU member states to any other country whose laws do not have an *adequate level of protection* [3]. Similarly, the Australian Privacy Principles (APP) [4] do not permit personal information flowing to a foreign country unless those country's laws are *substantially similar* to APP.

Recent research has proposed several techniques to determine the location of cloud service components (e.g., [5][6][7][8]) and various IP geolocation techniques exist aiming to determine *unknown* locations of Internet hosts (e.g., [9][10][11][12][13]). Yet one challenge – although recognized by several approaches, e.g. [8][11] – remains unaddressed: Network delay as well as topology information used by existing location techniques are subject to changes over time [14][15]. Thus it is vital for location techniques to adapt to these changes as their results otherwise may be rendered inaccurate.

In this paper, we propose an approach to continuously validate the location of cloud service components. To that end, we introduce a process called *adaptive location classification (ALC)* which treats potential geographical locations of a cloud service component as classes. The ALC process continuously collects, predicts and updates a *classifier*, i.e. a supervised learning algorithm which uses properties of a cloud service component to determine its class. We evaluate our approach within an experiment where the ALC process is used to continuously validate locations of cloud service components hosted at 14 different locations on the AWS Global Infrastructure.

The contributions of this paper are as follows:

- A description of an adaptive location classification process to support continuous location validation of cloud service components,
- a method to explicitly control tolerance of prediction errors leading to incorrectly invalidating a component's location (false negatives), as well as
- experimental results of applying the ALC process to continuously validate the locations of cloud service components hosted at 14 different locations on AWS.

The remainder of this paper is structured as follows: The next section describes the ALC process, including the

2330-2186/17 $31.00 © 2017 IEEE
DOI 10.1109/CloudCom.2017.29

IEEE computer society

description of main process parameters' configuration and the data structure used to represent the features supplied to the classifier. Thereafter, Section III demonstrates the application of the ALC process, outlining central components of the implementation of the ALC process, consisting of describing cloud service components under validation, and presenting experimental results of applying the ALC process to AWS resources. Finally, we discuss related work in Section IV and conclude this paper in Section V.

II. Adaptive location classification

This section introduces adaptive location classification, a process which continuously collects, predicts and updates a classifier. A central question to begin with is what properties of a cloud service component should be used to predict its location? Assuming that the component is provided by some remote host, using network delay measured on different layers of the TCP/IP protocol suite and topology information, i.e. the path taken to a specific component, are obvious choices to derive features.

Various geolocation techniques aiming at determining *unknown* locations of Internet hosts also use delay measurements and topology information (e.g. [9][10][11][12][13]). These techniques share an important assumption: They require that there are trusted *landmarks*, i.e geographical locations which are known and from which network delay and topology information are measured. Our approach also follows this assumption of trusted landmarks.

Since we are aiming to use classifiers to continuously validate the location of cloud service components, we have to take network changes over time into account as otherwise these changes can render our location predictions inaccurate. Such time-dependent learning problems are referred to as learning under *concept drift* [16]. Note that not every deviation from previous observations necessarily indicates a concept drift, some may result from momentary or short-termed anomalies. Distinguishing anomalies from permanent network changes is crucial as we otherwise update our classifier with flawed data. Figure 1 shows the steps of the adaptive location classification (ALC) process which are detailed hereafter.

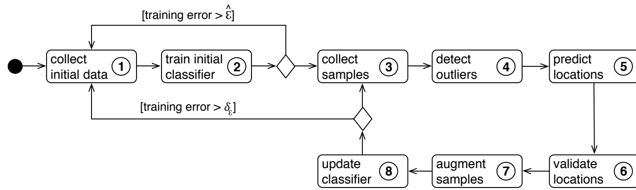

Figure 1: Adaptive location classification (ALC) process

A. Step 1: Collect initial data

The ALC process starts with the initial data collection which is used in the following step to train the initial classifier. To that end, we have to decide on which of the TCP/IP layers to measure network latency and which topology information to collect. Further, we have to define a feature vector, i.e. a data structure representing our delay measurements and topology information which are used as training as well as test input supplied to the classifier during the next steps.

On the *Internet layer*, we use the public IP address of the target host to ping it, i.e. measure the time delta between sending ICMP Echo Request and receiving ICMP Echo Reply packets. Also, we measure the time delta between sending ICMP Echo Request packets and receiving ICMP Time Exceeded packets for every hop until the target host is reached. Note that a single delay measurement is obtained by executing multiple successive measurements at a time, e.g. 20. Thus each data point is actually a distribution which we describe by its max^{IP}, min^{IP}, average (avg^{IP}), standard deviation (sd^{IP}), and $last^{IP}$ packet's delay. Further, in order to characterize the route taken to a target, we count the number of known and unknown intermediate routers.

On the *Transport layer*, we measure the time delta between sending a SYN and receiving a SYN-ACK TCP segment from the target host on a particular port. Analogous to measuring delay on the Internet layer, a single measuring point consists of multiple successive probes whose distribution is described by max^{TCP}, min^{TCP}, and average (avg^{TCP}).

Figure 2 shows how delay measurements on the Internet and Transport layer as well as topology information are transformed into a ten-dimensional feature vector: It includes the average over any descriptive statistic obtained from the delay measurements on the Internet layer, the known and unknown hop count, and the descriptive statistics of the delay measurements of the Transport layer.

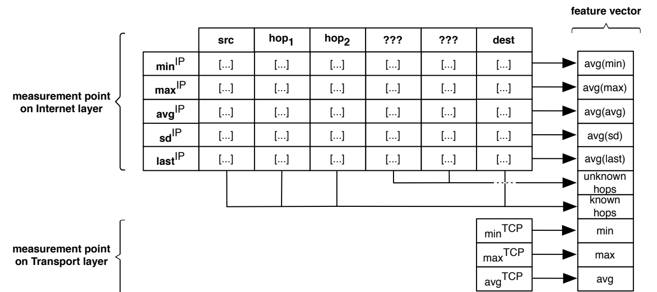

Figure 2: Feature vector derived from measurements on Internet and Transport layer

B. Step 2: Train initial classifier

In this step, we use the initially collected data to train the initial classifier. To that end, we have to select a supervised learning algorithm, potential candidates are the k-nearest neighbors algorithm (k-NN), support vector machine (SVM), decision tree, or random forest.

A central question at this point is how many data points are required to train the initial classifier? The answer depends on the required performance of the classifier. We choose the *training error* ε, that is, the proportion of overall incorrectly classified locations observed when using the trained model to classify locations of training set to describe the performance of the classifier. This allows us model our *required* performance as a constraint $\hat{\varepsilon}$ on the the observed training error ε.

Figure 1 indicates that we assume that an initial classifier with a training error larger than $\hat{\varepsilon}$ is incapable of meaningful predictions and therefore we restart the ALC process from scratch, i.e. reset it to Step 1. In this case, manual adjustments to increase the performance of the initial classifier are necessary which may include increasing the number of initial delay and topology measurements per location as well as choosing different parameter used for cross-validation, e.g. increasing 3-fold to 5-fold cross-validation.

C. Step 3: Collect new samples

In this step, new measurements of cloud service components' presumed locations are conducted which are used in the next steps to predict and validate their locations. To that end, network latency and topology information are measured in the same way as for initial data collection. Thereafter, obtained measurements are transformed according to the feature vector definition shown in Figure 1.

D. Step 4: Detect outliers

This step aims at detecting outliers present in the newly collected samples of Step 3. Detecting outliers in the last samples is necessary since we aim at adding new measurements to the data collection at each iteration in order to update the classifier (see Step 8, Section II-H). To that end, various unsupervised outlier detection algorithms are available, e.g. based on k-NN graph or one-class SVM.

E. Step 5: Predict locations

In this step, we supply the last collected delay and topology measurements to the classifier which predicts the corresponding locations.

F. Step 6: Validate locations

In this step, the predicted locations of the cloud service components are compared with the expected locations to decide whether a component's location is valid or invalid. Recall that during training of the initial classifier, we observe the training error ε, i.e. the overall error the classifier makes when predicting a location based on the training set. This implies that we cannot exclude the possibility that a classifier's prediction is erroneous, resulting in *incorrect invalidations* (false negatives) of cloud service components' locations. Consequently, we may *not* want to invalidate a location based on a single prediction.

Note that false positive classifications, i.e. *incorrectly validating* a location are not considered here. Such errors would imply that the cloud service provider is actively cheating on the measurement because, in this case, although the expected location matches the predicted one, the measured data point actually stems from a different location.

In order to minimize the probability to incorrectly invalidate a cloud service component's location, we can consider a sequence of predictions for a location l within a particular time interval. As an example, consider having observed a sequence of predictions for location l where the first two agree with the expected component's location $(+)$ while the last prediction indicates disagreement $(-)$, i.e. $\langle +, +, - \rangle$. So does this sequence indicate that location l is valid or not? A basic solution is to vote, i.e. rely on highest frequency and thus, in this example, consider the location l valid. It is obvious that this naive approach entails many drawbacks, for example, it requires to arbitrarily define an uneven number of successive predictions to be considered when voting.

We propose to explicitly control the probability of incorrectly invalidating a location through introducing the invalidation window w_l^- which is derived using the observed training error ε. To that end, we assume that ε is independent, that is, any prediction indicating an invalid location has the probability ε to be incorrect. We define the invalidation error $\upsilon_l^- \in [0, 1]$ which describes the probability that the number of $|w_l^-|$ successive predictions for location l indicating disagreement are incorrect:

$$\upsilon_l^- = \varepsilon_1 \times \varepsilon_2 \times \cdots \times \varepsilon_{|w_l^-|} = \prod_{i=1}^{|w_l^-|} \varepsilon_i = \varepsilon^{|w_l^-|}. \quad (1)$$

The key idea at this point is to use υ_l^- as a process parameter: We configure υ_l^- to define the maximum probability of making an error when invalidating a location l which we are willing to tolerate. Following this idea, we rewrite the equation 1 as follows:

$$|w_l^-| \geq \frac{\log(\upsilon_l^-)}{\log(\varepsilon)}. \quad (2)$$

We can make use of Inequation 2 to compute the minimum number of successive predictions which are required to be observed in order to invalidate a location l for a predefined υ_l^- and ε. Naturally, if we are willing to accept an invalidation error equal or greater than the training error, i.e. $\upsilon_l^- \geq \varepsilon$, then the invalidation window size is $|w_l^-| = 1$.

As an example, consider that we are willing to tolerate a maximum probability of making an error when invalidating location l of 0.0001%, i.e. $\upsilon_l^- = 0.000001$. Furthermore, let's assume that the observed training error of the classifier is 2%, i.e. $\varepsilon = 0.02$. According to Inequation 2, the minimum number of successive predictions to observe indicating disagreement in this example is

$$|w_l^-| \geq \frac{\log(0.000001)}{\log(0.02)} \approx 3.53.$$

Therefore, in the above example, the minimum number of successive predictions indicating disagreement which are needed to decide that a location l is invalid – i.e. not at its presumed location – is four.

Note that there is a subtle detail to this approach: The observed training error ε refers to all incorrectly classified locations observed when applying the classifier to the training set while the invalidation error υ_l^- is defined for an individual location l. From the perspective of the classifier's performance, the formulation of Inequation 2 can thus be understood as a worst case expectation where misclassified samples of a single location l may be responsible for the entire observed training error ε.

Using the invalidation window approach implies that its size has to always be smaller or equal to the number of samples collected in Step 3 (Section II-C) for each target URL of a cloud service component, that is, we need to collect at least as many new data points which are necessary to make predictions completing the invalidation window defined for a location l. Further, the invalidation window bears the risk of misinterpreting a prediction error for what is actually a component hosted at an invalid location. Recall our above example of having observed the sequence $\langle +, +, - \rangle$: The last prediction disagreeing with the expected location may not be caused by an error of the classifier but may actually result from an invalid location. To counter this misinterpretation of predictions, two parameters are crucial: First, the time between collecting new samples (i.e., successive executions of Step 3) has to be selected suitably short such that relocating the cloud service component while the ALC process executes is rather unlikely. Second, the smaller we define the error of incorrectly invalidating a cloud resource's location (υ_l^-), the more successive predictions indicating disagreement are needed to invalidate a location. Therefore, the desired invalidation error υ_l^- should be configured carefully.

G. Step 7: Augment samples

This step adds collected samples of the latest execution of Step 3 which are not marked as outliers to the existing data set. Further, in case a location l has been invalidated in the previous step, then all data points used for predictions of the invalidation window leading to the invalidation of l are discarded, i.e. will *not* be used to update the classifier.

Note that the increasing size of the data set may become too large for some classifiers, e.g. for SVM. We address this issue by controlling the upper bound of the training set size: If the upper bound is reached, then we apply a *sliding window*, i.e., each time newly collected samples are added to the data set for retraining (see next step), the same amount of oldest data points in the training set are discarded.

H. Step 8: Update classifier

Using the augmented data set, the current classifier is updated, that is, a new classifier is trained. Analogous to Step 2 (see Section II-B), we compute the training error ε_{t+1} which updates the last observed training error at ε_t. ε_{t+1} is then compared to the performance constraint δ_ε which is the maximum training error observed *after updating* that we are willing to tolerate. If ε_{t+1} satisfies the constraint, we use Inequation 2 to update invalidation window size $|w_l^-|$.

The above paragraph implies that the invalidation window size can always be adapted as needed. This begs the question how to initially configure the maximum size of w_l^-. As an example, let's assume that we collect ten new samples for each location (Step 3) at each iteration of the ALC process which limits the invalidation window size of each location $|w_l^-|$ to a maximum size of ten. With a desired invalidation error of 0.003%, i.e. $\upsilon_l^- = 0.00003$, the training error observed after having retrained the classifier using the augmented data set (Step 8) may increase to a maximum of (through rewriting Equation 1)

$$\delta_\varepsilon = \upsilon_l^{\left(\frac{1}{|w_l^-|}\right)} = 0.00003^{\left(\frac{1}{10}\right)} \approx 0.3529.$$

Put differently: An invalidation window size larger than ten implies that the observed training error of the classifier *after updating* may exceed 35.29%. Thus, given a maximum training error observed after updating δ_ε, we can derive the maximum invalidation window size. Further, if we observe a training error ε during Step 8 at any iteration of the ALC process which is larger than δ_ε, then we will terminate and restart the ALC process accordingly from scratch (i.e., reset the process to Step 1).

Note that the initial constraint on the training error $\hat{\varepsilon}$ (Step 2) to define required performance of the initial classifier as well as the performance constraint δ_ε applied after each update of the classifier (Step 8) are both configured manually.

Having completed Step 8, the ALC process, again, moves to Step 3, i.e. collecting new samples. Before executing the measurement, the process may wait for a defined interval. The ALC process keeps on repeating Step 3 through 8 until interrupted.

III. Application

In this section, we experimentally evaluate our approach to continuously validate locations of cloud service components. We begin with outlining main parts of the implementation of the ALC process described in the previous section. Thereafter, we describe the cloud service components under validation as well as the data set used to evaluate the ALC process (Section III-B). Finally, we present the experimental results of applying the ALC process (Section III-C).

A. Implementation of the ALC process

This section outlines the main components of the implementation of the ALC process.

1) Step 1 and Step 3: Conducting delay & topology measurements: Delay measurements and topology information are collected using *MTR*[1], a common network diagnostic tool used to determine whether a host can be reached over an IP network and what path packets to that host took. Further, *Nping*[2] which offers the possibility to measure response times on the TCP Layer. Data points are stored following the feature vector schema shown in Figure 2.

2) Step 2, Step 5 and Step 8: Training, prediction and updating: We use a classifier based on *Linear Support Vector Classification (LinearSVC)*[3] with 5-fold cross-validation to train the initial classifier (Step 2) and to update, i.e. retrain the classifier (Step 8). This classifier takes as input the measurements provided by Step 1 and 3 where each class label is given by the target location for which the measurements have been conducted.

Finally, in Step 5, we predict class labels of newly collected data points provided by Step 3. We use *scikit-learn*[4] and the LinearSVC which we trained beforehand to predict the location of the cloud service components.

3) Step 4: Outlier detection: Step 4 aims at detecting outliers in newly collected samples, that is, data points provided by Step 3 for each target location. To that end, we also use *scikit-learn* to conduct unsupervised outlier detection using *OneClassSVM*[5].

4) Step 6 and Step 7: Validate locations and augment data set: Our prototype uses Python to implement the invalidation window approach described in Section II-F. Further, data points are persisted and managed using MongoDB.

B. Cloud service components under validation and data set

Figure 3 shows the AWS global infrastructure: Using AWS terminology, there are 16 geographical *regions* (orange circles) each having multiple *availability zones* (indicated by the number in the orange circles) as well as planned regions (green circles). These regions include [17]: Oregon, Northern California, California, Northern Virginia, Ohio, Central Canada, Sao Paulo, Ireland, Frankfurt, London, Singapore, Sydney, Tokyo, Seoul, Mumbai Beijing, and AWS Gov Cloud.

Within our experiment, we deployed 14 Amazon EC2 instances in total, each hosted in a different of the above region, excluding Beijing and AWS Gov Cloud. Each instance had the following configurations: Ubuntu Server 16.04 LTS with 1 vCPU, 0.5 GB main storage and 8 GB of persistent storage. Further, each instance was associated with a publicly reachable IP address and a security group

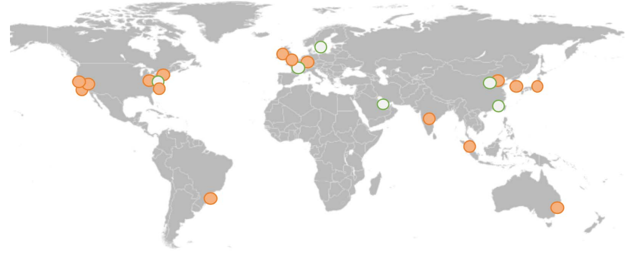

Figure 3: AWS Global Infrastructure (Figure based on [17])

was configured to allow for traffic via ICMP, via TCP on port 22 and via SSH from and to the landmark. Lastly, the landmark, i.e. the point from which delay and topology measurements were conducted was also hosted on AWS, at region Frankfurt.

For the 14 EC2 instances described above, we collect a total of 139699 delay measurements. The collection is split up into two periods, the first starts at 17th December 2016, 12:04:40 (UTC) and ends at December 23rd 2016, 10:29:52 (UTC) while the second period starts at 25th December 2016, 12:20:27 (UTC) and ends at 3rd January 2017, 10:34:46 (UTC). Each single data point contains the information described by the feature vector schema shown in Figure 2.

C. Experiment and evaluation

Using the data set described in the previous section, hereafter we delineate the experimental application of the ALC process and evaluate the results.

1) Process configuration: In order to initialize the ALC process within our experiment, we define a upper bound for the misclassification error observed during training of $\hat{\varepsilon} = 3.27\%$. This corresponds to initially collecting 13979 records or using the first ≈10% of the data set to train the initial classifier, that is, Step 1 & 2 of the ALC process (see Figure 1).

Further, for each of the 14 locations we define the identical invalidation error $\upsilon_l^- = 0.001\%$ which delineates the maximum error we are willing to tolerate when invalidating a location. Assuming that we are willing to tolerate a maximum training error observed after updating the classifier of $\delta_\varepsilon = 35\%$ (Step 8), then adapting Inequation 2 as follows allows us to obtain the maximum invalidation windows size:

$$|\tilde{w_l}^-| \leq \frac{\log(\upsilon_l^-)}{\log(\delta_\varepsilon)} \leq \frac{\log(0.00001)}{\log(0.35)} \leq 10.97.$$

Thus the maximum invalidation windows is ten which corresponds to maximum training error observed after updating $\delta_\varepsilon = 0.00001^{\left(\frac{1}{10}\right)} \approx 31.62$.

2) Process execution: Applying the results of the above paragraph to our experiment, the remaining 90% or 125720 of data points of the data set are split up into 898 successive batches where each batch contains 10 probes per location,

[1] https://linux.die.net/man/8/mtr
[2] https://nmap.org/nping/
[3] http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html
[4] http://scikit-learn.org/
[5] http://scikit-learn.org/stable/modules/generated/sklearn.svm.OneClassSVM.html

that is, 140 probes per batch. Starting with the oldest one, each batch *simulates* the execution of *Step 3: Collect new samples* of the ALC process through supplying 140 newly collected samples to *Step 4: Detect outliers*. Once outlier detection has completed, location of all non-outliers is predicted (Step 5) and the predictions are compared with the expected location (Step 6) to compute the proportion of the correctly classified samples per batch, i.e. the test accuracy per batch. Figure 4 shows how the test accuracy per batch evolves over time. Having completed Step 6, the initial data set is augmented with non-outliers (Step 7) and the classifier is updated, that is, retrained (Step 8). Since there are 898 batches each of supplying 140 newly collected probes, the Step 3 to 8 of the ALC process are iterated over 898 times which implies that the classifier is updated 898 times during our experiment. After each update of the classifier, we observe the training error ε which the classifier produces when applied to the respective training set at that time. Figure 5 shows how the training error evolves over time.

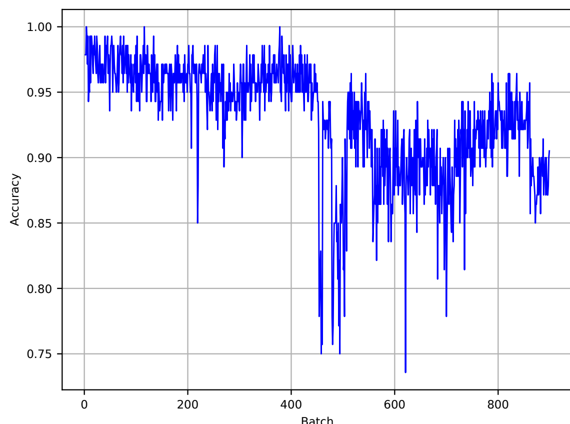

Figure 4: Evolving test accuracy per batch over time

Using LinearSVC implies that the required storage and compute resources increase quickly with increasing size of training set[6]. Therefore, we cannot increase the size of the training set arbitrarily but define an upper bound, in our case 30000 data points. Once the ALC process reaches this upper bound through iteratively augmenting the data set, we apply the sliding window described in Section II-G, that is, for each newly collected batch of 140 samples (minus those filtered by outlier detection), we remove the same amount of the oldest data points in the training set.

3) Evaluation statistics: For each batch, we observe the correctly classified locations per batch, i.e. the *test accuracy per batch*, and the correctly classified locations during training, i.e. *training accuracy*. Table I shows these statistics, thus summarizing the outcome of our experiment by including mean ($\bar{x}$), median ($\hat{x}$) standard deviation sd, min, and max. Furthermore, Figure 4 shows how the test accuracy of each experiment evolves over time, that is, with increasing number of batches consumed by the ALC process. Note that we assume all 14 instances to reside at their claimed location at the time of the experiment, that is, the observed deviations from expected locations result from errors the classifier makes.

The experimental results are as follows (Table I): The mean test accuracy per batch is 92.96% which translates to a mean misclassification rate of 7.04% per batch. At batch number 621, the accuracy drops below 75%, with a minimum at 73.57%. Further, the mean training accuracy is 92.13% with a minimum of 90.07%.

Furthermore, when inspecting the evolution of the training error shown in Figure 5, we can observe a sharp increase at batch 455. Figure 6 illustrates how the invalidation windows size compensates for this increase in training error: After consuming batch number 455, it adapts the window size from $|w_l^-| = 4$ to $|w_l^-| = 5$. It is important to note that while the invalidation window size ranged from 3 to 5 during the course of the experiment, the invalidation window was *never* satisfied for any location for any batch. Put differently: Since we are assuming that all 14 instances reside at their claimed location during the experiment, we can conclude that none of the cloud service components' locations was incorrectly invalidated. This is, after all, the expected – and desired – result of this experiment.

Table I: Results of continuous location validation of 14 AWS EC2 instances using 10% of total data set as initial training data and 898 successive batches with each 140 newly collected samples

Parameter	$\bar{x}$ (%)	$\hat{x}$ (%)	sd (%)	max (%)	min (%)
Test accuracy per batch	92.96	94.28	4.35	100	73.57
Training accuracy	94.13	95.41	2.47	97.91	90.07

IV. Related Work

This section presents related work and explains how the ALC process complements existing location techniques.

A. Locating cloud service components and data

Eskandari et al. [8] conduct delay measurements of around 500 websites and use polynomial regression to train a prediction model. Labels required for this supervised learning approach are obtained from geoIP databases while they test their model using three randomly picked destinations. Fotouhi et al. [6] create a set of public landmarks from which geolocation measurements are performed. They use *constraint-based geolocation (CBG)*, a method that was originally introduced by Gueye et al. [10], to locate Internet

[6]http://scikit-learn.org/stable/modules/svm.html\#complexity

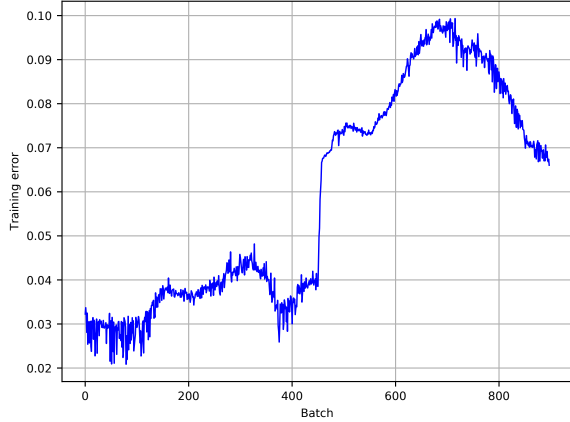

Figure 5: Evolving training error ε observed per batch over time

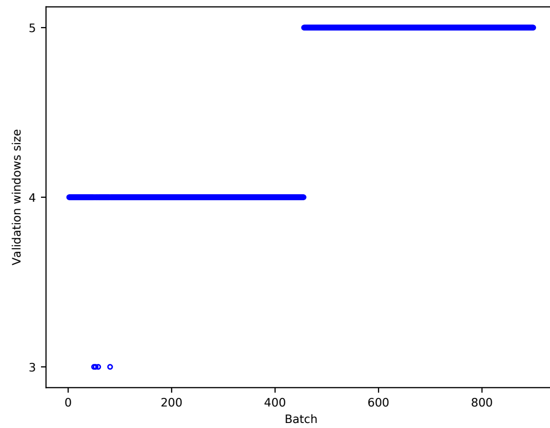

Figure 6: Adaption of invalidation window size per batch over time

hosts, which is deployed on EC2 instances to estimate the location of physical hosts within Google data centers. Ries et al. [18] build on virtual coordinate systems (VCS) which are used to predict network latencies to determine the geographic location of virtual machines without having to conduct extensive measurements.

Gondree and Peterson [19][7] introduce a generic framework to actively validate geographic locations of data in the cloud, using latency-based techniques. To that end, they propose *contraint-based data geolocation (CBDG)* which combines latency-based geolocation techniques with a probabilistic proof of data possession. Fu et al. [20] extend this approach by improving the scheme presented in [7] through using Trusted Platform Modules (TPM). Albeshri et al. [21] present a similar approach called *GeoProof*, a protocol that combines proof of storage protocols with distance-bounding protocols. Proof of storage protocols allow to verify integrity of stored data without completely downloading it while distance-bounding protocols is an authentication protocol between a verifier and a prover, serving to establish the claimed identity and physical location of the prover. In a related line of work, Jaiswal and Kumar [5] propose an application level scheme which combines download times of files and network delay of IP packets to locate the data center where these files are stored.

B. IP geolocation techniques

Apart from locating cloud service components as well as data stored in the cloud, there is a set of general techniques which were developed to determine the geographic location of Internet hosts. One of the early works on this subject is presented by Padmanabhan et al. [9]. They propose three distinct techniques to locate Internet hosts: (1) *Geotrack* which infers the location using DNS entries and names of nearby network nodes, (2) *GeoPing* which uses network delays to estimate the coordinates of an Internet host, and (3) *Geocluster* uses Autonomous System (AS) information, that is, their prefixes to extract geographical clusters (inter-domain routing information which is derived from Border Gateway Protocol (BGP)). Gueye et al. [10] introduce *constraint-based geolocation (CBG)* which uses a special variant of multilateration to infer the area in which an Internet host is located. Katz-Bassett et al. [11] propose *topology-based geolocation (TBG)* which initially makes a conservative estimate of possible Internet locations using maximum transmission speed of IP packets. This first estimate is then refined by considering latencies between routers which are on the path from the landmark to the Internet host. Eriksson et al. [12] take a different approach by framing IP geolocation as a machine-learning classification problem. They use a Naive Bayes estimation method to classify IP geolocations. Lastly, Arif et al. [13] present *GeoWeight* which extends CBG by also taking into consideration that for a measured latency to an Internet host, some distance are more probably then others. They use this insight to formalize an additional constraint and show that under certain circumstances, their approach outperforms CBG.

C. Identification of gaps and contribution

None of the above approaches proposes a method how to *continuously* check the location of cloud service components or hosts. Therefore, our approach complements current research efforts in two major ways: First, existing technique to validate previously *known* locations of cloud service components and data (e.g., [8][12]) can adapt our process model to allow for continuous validation. Second, geolocation techniques to determine *unknown* locations of hosts, components or data (e.g., [6][7]) can be used to establish trusted labels mapping URLs or IPs to locations which are then used by the classifier of the ALC process.

V. CONCLUSION

In this paper, we introduced adaptive location classification, a process allowing to continuously validate the location of cloud service components. We demonstrated the feasibility of our approach through continuously validating the locations of components hosted at 14 different locations of the AWS Global Infrastructure.

One fundamental limitation of our approach is that it may only determine the location of a proxy server but not the actual location of cloud service components. This issue has also been raised by previous approaches, e.g. [5][9][18][22]. Yet even when neglecting this limitation, the conclusions drawn about the location of a cloud service's components are confined to locating physical server on which some cloud service component is running. A natural extension to locating components is to determine the geographical location of *data* stored by the cloud service, usually referred to as *data sovereignty*. This requires simultaneously establishing the geographical location of the server on a network and proving that the data is actually stored at that location [7][19]. Still, this approach does not cover locating any other copies of the data stored on a different server. Without having full control of the network, tracking any such copies is a different (and hard) problem [19].

As part of future work, we will investigate the performance of other classifiers, e.g. based on random forests, to improve the test accuracy further. Also, we will apply more sophisticated methods to detect concepts drifts to choose the point in time when to retrain the classifier more efficiently.

ACKNOWLEDGMENT

This work was partly funded by the Bavarian Ministry of Economic Affairs and Media, Energy and Technology, within the project *Bayern-Cloud* as well as by the EU H2020 project *EU-SEC*, Grant No. 731845.

REFERENCES

[1] The German Federal Office for Information Security (BSI)), "Cloud Computing Compliance Controls Catalogue (C5)," https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/CloudComputing/ComplianceControlsCatalogue/ComplianceControlsCatalogue.pdf?__blob=publicationFile&v=4, 2016.

[2] Cloud Security Alliance (CSA), "Cloud Control Matrix: Seurity Controls Framework for Cloud Providers & Consumers," https://cloudsecurityalliance.org/research/ccm/, 2015.

[3] "Directive 95/46/EC of the European Parliament of the Council (Data Protection Directive)," http://eur-lex.europa.eu/legal-content/en/TXT/?uri=CELEX:31995L0046, 1995.

[4] "Australian Privacy Principles (APP)," https://www.oaic.gov.au/individuals/privacy-fact-sheets/general/privacy-fact-sheet-17-australian-privacy-principles, January 2014.

[5] C. Jaiswal and V. Kumar, "IGOD: identification of geolocation of cloud datacenters," *Journal of Information Security and Applications*, vol. 27, pp. 85–102, 2016.

[6] M. Fotouhi, A. Anand, and R. Hasan, "PLAG: Practical Landmark Allocation for Cloud Geolocation," in *8th International Conference on Cloud Computing (CLOUD)*. IEEE, 2015, pp. 1103–1106.

[7] M. Gondree and Z. N. Peterson, "Geolocation of data in the cloud," in *Proceedings of the third ACM conference on Data and application security and privacy*. ACM, 2013, pp. 25–36.

[8] M. Eskandari, A. S. De Oliveira, and B. Crispo, "VLOC: An Approach to Verify the Physical Location of a Virtual Machine In Cloud," in *6th International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2014, pp. 86–94.

[9] V. N. Padmanabhan and L. Subramanian, "An investigation of geographic mapping techniques for Internet hosts," in *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4. ACM, 2001, pp. 173–185.

[10] B. Gueye, A. Ziviani, M. Crovella, and S. Fdida, "Constraint-based geolocation of internet hosts," *IEEE/ACM Transactions On Networking*, vol. 14, no. 6, pp. 1219–1232, 2006.

[11] E. Katz Bassett, J. P. John, A. Krishnamurthy, D. Wetherall, T. Anderson, and Y. Chawathe, "Towards IP geolocation using delay and topology measurements," in *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*. ACM, 2006, pp. 71–84.

[12] B. Eriksson, P. Barford, J. Sommers, and R. Nowak, "A learning-based approach for IP geolocation," in *International Conference on Passive and Active Network Measurement*. Springer, 2010, pp. 171–180.

[13] M. J. Arif, S. Karunasekera, and S. Kulkarni, "GeoWeight: internet host geolocation based on a probability model for latency measurements," in *Proceedings of the Thirty-Third Australasian Conferenc on Computer Science-Volume 102*. Australian Computer Society, Inc., 2010, pp. 89–98.

[14] R. V. Oliveira, B. Zhang, and L. Zhang, "Observing the evolution of Internet AS topology," in *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 4. ACM, 2007, pp. 313–324.

[15] R. Pastor Satorras, A. Vázquez, and A. Vespignani, "Dynamical and correlation properties of the Internet," *Physical review letters*, vol. 87, no. 25, p. 258701, 2001.

[16] I. Zliobaite, "Learning under concept drift: an overview," *arXiv preprint arXiv:1010.4784*, 2010.

[17] "AWS Global Infrastructure," https://aws.amazon.com/about-aws/global-infrastructure/?nc1=h_ls, [Online; retrieved October 30, 2017].

[18] T. Ries, V. Fusenig, C. Vilbois, and T. Engel, "Verification of data location in cloud networking," in *4th IEEE International Conference on Utility and Cloud Computing (UCC)*. IEEE, 2011, pp. 439–444.

[19] Z. N. Peterson, M. Gondree, and R. Beverly, "A Position Paper on Data Sovereignty: The Importance of Geolocating Data in the Cloud," in *HotCloud*, 2011.

[20] D. L. Fu, X. G. Peng, and Y. L. Yang, "Trusted validation for geolocation of cloud data," *The Computer Journal*, 2014.

[21] A. Albeshri, C. Boyd, and J. G. Nieto, "Geoproof: proofs of geographic location for cloud computing environment," in *32nd International Conference on Distributed Computing Systems Workshops*. IEEE, 2012, pp. 506–514.

[22] P. Gill, Y. Ganjali, B. Wong, and D. Lie, "Dude, where's that IP?: circumventing measurement-based IP geolocation," in *Proceedings of the 19th USENIX conference on Security*. USENIX Association, 2010, pp. 16–16.