Model-based cluster analysis for identifying suspicious activity sequences in software

Hemank Lamba Thomas J. Glazier Javier Cámara
Bradley Schmerl David Garlan Jürgen Pfeffer

Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
{hlamba,tjglazier,jcmoreno,schmerl,garlan,jpffefer}@cs.cmu.edu

ABSTRACT

Large software systems have to contend with a significant number of users who interact with different components of the system in various ways. The sequences of components that are used as part of an interaction define sets of behaviors that users have with the system. These can be large in number. Among these users, it is possible that there are some who exhibit anomalous behaviors – for example, they may have found back doors into the system and are doing something malicious. These anomalous behaviors can be hard to distinguish from normal behavior because of the number of interactions a system may have, or because traces may deviate only slightly from normal behavior. In this paper we describe a model-based approach to cluster sequences of user behaviors within a system and to find suspicious, or anomalous, sequences. We exploit the underlying software architecture of a system to define these sequences. We further show that our approach is better at detecting suspicious activities than other approaches, specifically those that use unigrams and bigrams for anomaly detection. We show this on a simulation of a large scale system based on Amazon Web application style architecture.

1. INTRODUCTION

In the contemporary corporate setting, having malicious users interact with software is potentially disastrous and can cost organizations millions of dollars. For instance, Cummings et al. [9] studied 80 cases of fraud in the financial services sector. They found that in most cases, malicious users interacted with the parts of the systems for which they had permission to access, based on their roles in the companies. However, such users were behaving *anomalously* compared to how they had behaved before, or how others in similar roles behaved. The impact of such cases was as high as 28 million dollars. Similar results were found in other sectors – government [22], information technology, telecommunications [23], and critical infrastructure sectors [21]. Suspicious behavior can arise from the existence of compromised user accounts, rogue users, or by less knowledgeable users who break things unwittingly.

Many challenges exist in identifying suspicious behavior since the number of suspicious activities is a tiny fraction of the number of normal activities. The scale of modern systems, including the number of components they comprise, the variety of functions that they perform, and the diverse and large number of users that they are required to interact with, make manual inspection infeasible.

Many approaches have been proposed to find malicious users in review websites [16] and social media data [17, 30]. Most of these approaches attempt to identify suspicious users based on the distribution of the products/activities they perform [16, 17], based on the inter-arrival time distribution [10], or based on group attacks [3, 19, 30, 33]. All of these approaches assume that a user interacts with each component of a system independently, i.e., the interaction pattern with a single component (or the system as a whole) determines whether a user is suspicious.

However, software systems are actually composed of many components, and a particular user interaction is actually a sequence of interactions with *different* components as defined by its software architecture [32]. For example, in a simple web application, assuming a 3-tier architecture, a user's request will first interact with a web server, then an application server, and finally a database. Suspicious users can be detected by identifying atypical interaction sequences with these components. For example, a user interaction that communicates with a database, without first interacting with the web server, could signal that the user has a back door into the database and is exfiltrating data. Therefore, it is essential to consider sequences of user activities over the given software architecture for detecting suspicious users. Sequences can also be determined by examining log data. In this scenario, password access (login followed by a password change), file transfer (login followed by a file transfer) can be seen as a suspicious sequence of activities.

As modern software systems are large, with many users performing activities on the system, it is essential for security analysts to focus and prioritize their inspections on suspicious users to determine whether their activities are dangerous to the organization or not. Thus, a good algorithm for detecting suspicious user behavior should provide a suspiciousness score for each user, allowing security experts to prioritize their inspection activites.

Informally, our problem can be stated as follows:

PROBLEM 1 (INFORMAL). *Given a set of users interacting with various components in a software system, compute a suspiciousness score for each user.*

In this paper, we describe an approach that first abstracts out the activities carried out by all users of the system to produce sequences of activities We then present a principled way of constructing a Bayesian model for generating the user sequences. Finally, a statistic based on the likelihood-based metric (which measures how much a user deviates from other users) is used to determine the suspiciousness score of a user.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

Our primary contributions are as follows:

Modeling sequences: Our approach takes into account the temporal sequencing of activities carried out by users. It does not assume that the activities performed by users are independent of each other.

Complex System Representation: Our approach can represent complex software systems at a software architecture level of abstraction, enabling scalability and interpretability.

Accuracy: We show that on a simulated dataset of user activities with different percentages of anomalous users, our method was able to detect suspicious users with significant precision and recall, and perform better than other commonly used methods based on activity count alone.

Scoring Function: Our proposed method provides a suspiciousness score for each user sequence, thus allowing security experts to prioritize their inspections.

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 motivates the problem and provides examples of threat scenarios that our approach aims to handle. We describe our approach in Section 4. We describe our evaluation in Section 5 and conclude by discussing the results and future work in Section 6.

2. BACKGROUND AND RELATED WORK

Over the past few years a lot of work has been done in the field of finding user-related anomalies. Most of that work is centered on finding fake reviews [1, 17, 37], finding fake followers on social media [30], finding bots on social media [10], and finding suspicious reactions to retweet threads [14]. We can classify the approaches proposed in this work in the following general categories:

Graph-based approaches: Graph-based approaches assume that the data can be represented in a graph format. For instance, product review data (e.g., at Amazon) can be represented as a bipartite graph where nodes are users and products, and an edge is present between a particular user and a particular product if that user has reviewed/rated that product. Most graph-based approaches for finding fraud work on simple graphs (ignoring the node, edge labels). The most common technique used is singular value decomposition (SVD), where similar nodes in the graph can be clustered together based on the products that they rated [20, 27, 30]. Techniques like belief propagation [1] and Markov random fields [26] have also been applied to discover anomalous subgraphs. Additional features such as edge-labels [31] and neighborhood information [37] have also been used to discover fraud in graphs.

Time-based approaches: A lot of work has been done in detecting anomalous patterns in multivariate time series data [8, 25, 28, 35]. Beutel et al. proposed an algorithm to find fraudulent temporal patterns in graphs [3]. Costa et al. found that second-order temporal features, such as inter-arrival time, could be used to detect bots on social media websites [10]. Temporal patterns of reviews have also been used to detect unusual (and hence potentially anomalous) time periods of activity [36, 15].

Sequence Anomaly Detection: Sequence anomaly detection involves detecting sequences that are anomalous with respect to some definition of normal behavior. The key challenges in sequence anomaly detection are that sequences might not be of equal length, and therefore any distance based approach cannot be applied. Transformation techniques such as box-modeling have been proposed, where each instance is assigned to a box, and features over these boxes are compared [7]. Teng et al. proposed a modeling approach to generate sequential rules from sequences [34]. This approach requires training data, which is often not feasible for large systems, as manual annotation of sequences is challenging due to the volume and the number of variations. Similarly, another approach proposed by Forrest et al. [11] uses a Hidden Markov Model (HMM) technique to detect anomalous program traces in operating system call data. The authors train a HMM using training sequences, and then the number of mistakes made by inputting the test sequence into the trained HMM is the anomaly score for the given test sequence. The work by Forrest et al. is closest to our approach; however the algorithm requires training data which is hard to obtain. A popular technique that has been used in anomaly detection is clustering points, and marking points not belonging to any cluster as suspicious outliers. Clustering sequences requires computing distances between all pairs of input. However computing distances between sequences requires formulating a definition of distance – which needs to be robust to variable length and misaligned sequences. A popular way of computing distance is counting the number of frequent patterns (n-grams) occurring in both sequences [12].

Graph-based approaches are not robust enough to find suspicious paths (sequences) in graphs, and hence cannot be applied in our context. In our approach, we assume that the dataset given to us is agnostic of time i.e., for generating sequences. We do not consider the time at which a particular activity was done, but only what activity was performed, making temporal approaches ill-suited to our context. Sequence based approaches are closer to what we want to do. However, these approaches have only been used in the context of bioinformatics, and have not been applied to software architecture. Some of the mentioned techniques require training data to differentiate between normal and abnormal behavior, which is not feasible for all types of data. When applied to bioinformatics, the sequence based approaches assume that the sequences are of fixed lengths and aligned. To adapt the approach to our context, we need to be able to deal with variable lengths (users may interact with different numbers of components) and are not necessarily aligned (sequences may start at different points in the architecture). Our approach can work with unaligned, variable length sequences generated by users interacting with a system, and represented at an architectural level of abstraction. Additionally, our approach can create different features based on the abstraction level with which it extracts features from each activity, and also provides a suspiciousness score for each sequence.

3. MOTIVATING EXAMPLE AND THREAT SCENARIO

Modern software applications are commonly deployed on flexible infrastructures like public and private clouds. Elasticity, where parts of the applications can be duplicated on an arbitrary number of cloud devices, challenges traditional graph-based security methods because of the frequent changes in the instances deployed, meaning that if care is not taken they will be treated as different nodes in a graph. However, these different deployments technically serve the same architectural purpose. For example, a large-scale web system will elastically add and remove VMs to handle variability in web traffic, but each VM performs the same function – namely, a web server.

Because of this architectural interchangability, a given user's path of interaction with individual computing instances is generally predictable, despite frequent changes in the individual instances. For example, a typical user request path might go from a web server to an application server to a database, as determined by the logic of the application. However, a subsequent request would follow the same path in the architecture, but would likely interact with different instances of web servers, application servers, and database servers. Using the architecture of the system as an abstraction and examining users' traces through that architecture allows for an effective analysis despite the dynamic nature of the system.

Dynamic cloud infrastructures are commonplace in industrial IT systems, some of which have suffered high-profile data breaches. A frequent pattern in these breaches is when an attacker gains a 'foothold' inside a network by compromising a single server. From this server the

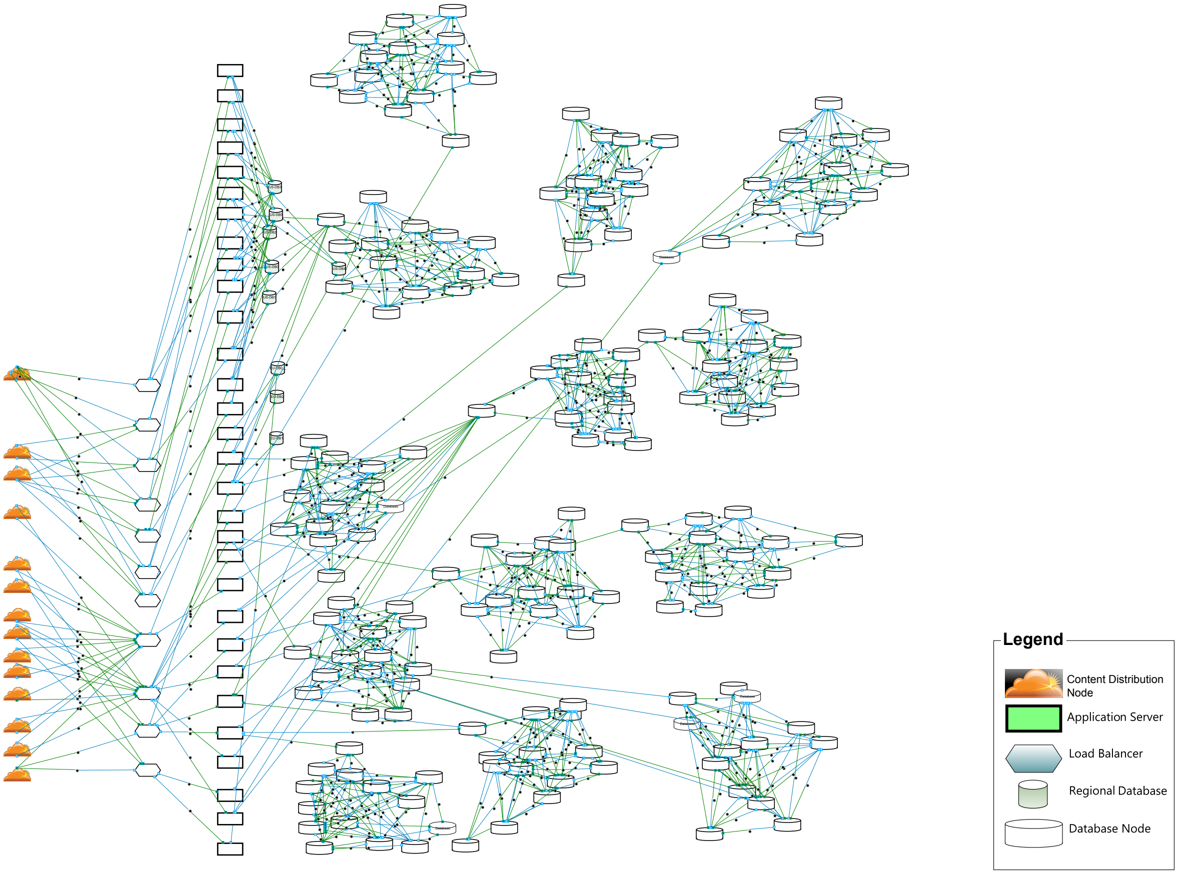


Figure 1: Example architecture of Amazon Web application.

attacker will move through the network looking for vulnerable systems of interest and, when ready, exfiltrate data by hiding it within normal network traffic. In this attack scenario, there may be two distinct architectural traces. For example, first there might be a linear trace, during which the attacker is finding vulnerable components of interest and, second, a star trace in which the compromised component is contacting other components to retrieve data.

Figure 1 shows an example Amazon web application in which a user first interacts with a content delivery network (CDN) to retrieve static content, a load balancer to distribute their request to an appropriate application server, the application server, and finally a database. This architectural pattern includes the common components of a standard modern web architecture, and at 240 nodes is representative of a significant enterprise web system that would exceed the capabilities for all but the most significant web properties (e.g. Netflix, Google, Yahoo, etc.).

In Figure 2, we illustrate how Bayesian modeling of the data might be helpful to discover suspicious users. Consider an organization, where three primary roles exist (Figures 2a–2c). The software system of the organization contains various components that are accessed by its users. Each software component is represented as a node in the graph, but our approach can be adapted to various levels of abstraction (e.g. geographic location of each deployed component). We hypothesize that each individual within a role in the organization will create traces that are reflective of that job role and will be similar to each other. However, an attacker or malicious user will not follow an existing pattern. Therefore, those sequences of events will be easily identifiable, even amongst the normal and varying behavior of legitimate users. Figure 2d shows an example of a suspicious trace which is visibly distinct from the normal behavior.

4. APPROACH

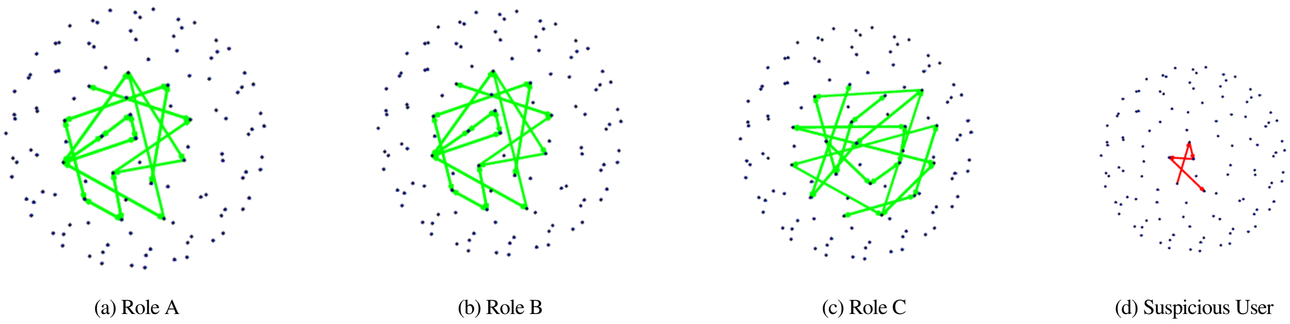
Sequences of user interactions with a software system can be a rich source of information about suspicious user behaviors. We want to provide a metric of suspicious behaviors that security experts (or downstream tools) can use to further examine whether a user account is compromised/malicious. We first explain the set of symbols used in our approach in Table 1.

Our problem, therefore, can be captured as:

PROBLEM 2 (FORMAL). *Given a set of users $\mathcal{U}$ operating on a software system that has a given architecture, with each user generating a trace $\mathcal{A}_u$, and an abstraction function $\mathcal{F}$, find and score suspicious users based on their respective execution sequences.*

Our solution to this problem requires three elements, as shown in Figure 3:

- **Behavior collection:** Instrumenting the system as it runs, to determine traces of user interactions as they occur and determining how to abstract the sequence information into an analyzable form.
- **Data modeling:** Clustering these traces and giving each sequence an affinity score within the cluster with which it shares the most in common.
- **Scoring anomalies:** Identifying outliers as sequences that have a high probability of being in their own cluster.


(a) Role A

(b) Role B

(c) Role C

(d) Suspicious User

Figure 2: User sequences overlaid on nodes in the architecture.

Symbol	Description
$\mathcal{U}$	Set of Users
$\mathcal{A}$	Set of nodes in the software architecture
$\mathcal{A}_u$	Set of nodes traced in the architecture by user u
$\mathcal{F}$	$\mathcal{F}$: Abstraction function, which converts each activity to a set of features
S_u	A sequence pertaining to a user u, where each sequence is represented as a series of $\mathcal{F}(A_u)$, where $F(a)=a$ is also possible.
len_u	Length of sequence of user u
K	Number of clusters

Table 1: Table of Symbols and their Descriptions used in this paper.

4.1 Behavior collection

For this part of our approach, we first describe the software system as a graph. We do this by using its component and connector (C&C) software architecture [32]. A C&C view describes the run-time configuration of the software in terms of nodes that perform computation or store data (i.e., components), and edges that represent communication between components (i.e., connectors). This can originate from the design documentation for the system, or be derived from the architectural patterns used to develop the system (e.g., Amazon's description of how to construct cloud-based applications, or through common idioms such as N-tiered architectures).

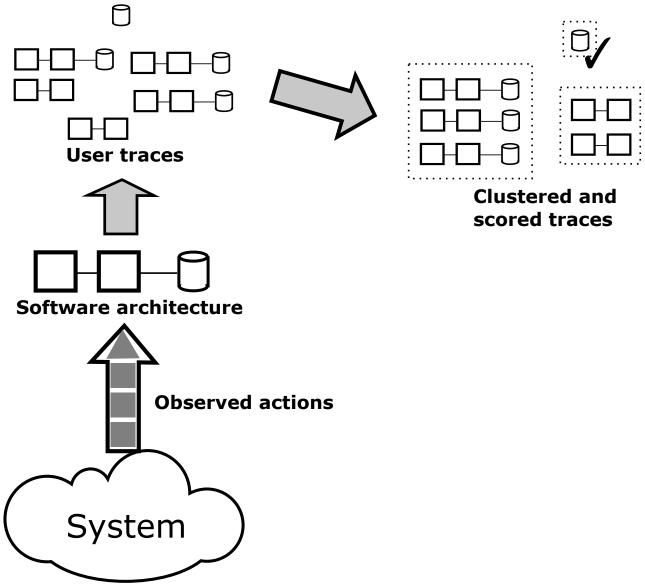


Figure 3: Overview of our approach.

Secondly, we monitor the execution of the software system to extract users' activities while they are interacting with the system, to produce interaction traces or sequences. A sequence is the order in which users interact with components along communication paths. The sequences can be abstracted out by defining an abstraction function $\mathcal{F}$ over the connected components. For example each component along the communication path in our example can be represented by the type of server (CDN, database, etc.), or by the location where it is placed (Europe, US, Japan, etc.), or a combination of both. Abstraction reduces noise that might have crept into the system due to multiple nodes of the same type, and allows us to leverage various pieces of contextual information (e.g. geographic location of deployment) to enhance the effectiveness of the technique. Different definitions of abstraction functions can find different types of suspicious behavior. These sequences can be employed to characterize paths over the underlying software architecture graph. For example, if a user accesses component 1 (e.g., a web page) which then interacts with component 2 (e.g., a server) which in turn accesses data component 3, and next component 1 again, followed by component 3, we can represent this user as the path $1 \rightarrow 2 \rightarrow 3 \rightarrow 1 \rightarrow 3$, or if we define abstraction function to be the type of component, it can be defined as Web Server $\rightarrow$ Server $\rightarrow$ Data. Figure 4 shows an example of extracting this information. We can get this information either by accessing log files, or in previous work we have been able to do this as the system executes [5, 6].

4.2 Data Modeling

For the task of finding suspicious sequences, we use a model-based approach. Specifically, the approach assumes that each user sequence belongs to a cluster, and sequences not belonging to any cluster can be marked as suspicious. As previously explained, standard clustering approaches cannot be used in this context, since defining distances between sequences is non-trivial. Therefore, we use a model-based approach, specifically the software architecture, to characterize each of the sequences and consequently how similar or how different the sequences are from each other. It has been shown elsewhere that model-based approaches for clustering sequences are better for highly dynamic and large scale data [4].

In our model-based approach, we assume that each user has a role within the organization, and we further assume that the activities a user carries out during a session is dependent on that role. For example, a data analyst in a corporate marketing department will interact with specific software systems and databases relevant to their job duties. However, another user, an accountant in finance, will interact with different software systems and databases. In each of these roles the users can select various actions from a set of possible actions relevant and allowed for that job function. This could be represented as a generative model in the following way:

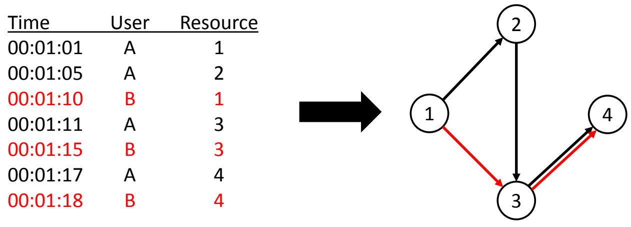
1. A user implicitly chooses what cluster she belongs to. This assignment may depend on her role, or the task that she wants to perform.
2. Given the cluster, then her behavior, in the form of a sequence of activities, is generated from the model by using an activity distribution related to that cluster.

As in any model-based approach, it is assumed that data from different users are generated independently. Our current approach assumes that a user is equally likely to belong to any cluster and does not use any type of additional knowledge, such as the users role in the organization. Furthermore, the input to the clustering algorithm is the number of clusters (K) as found by the BIC algorithm and the sequence data. A potential enhancement to our method in the evaluation of real world datasets, is to get such information from the dataset itself (e.g., the number of valid roles and which users are assigned to role). In our algorithm, each cluster is associated with an n^{th}-order Markov chain.[1] The probability of an observed sequence $x_0,x_1,x_2...x_d$ belonging to a particular cluster c_k is given by:

$$p(x|c_k)=p(x_0,c_k).\prod_{i=1}^{i=d}p(x_i|x_{i-1},c_k)$$

where $p(x_0,c_k)$ is the probability of x_0 being the starting state of sequences in cluster c_k, and $p(x_i|x_{i-1})$ is the transition probability of moving from state $i-1$ to state i in the model specified by cluster c_k.

Each cluster can be represented by a transition matrix that captures the probability of moving from one state to another in a sequence belonging to some cluster. These transition matrices are represented by $\theta=(\theta_1,\theta_2,\theta_3....\theta_k)$. Algorithm 1 starts by initializing all the transition matrices randomly, and also initializing sequences into clusters randomly. Then, it keeps on alternating with E-step and M-Step until it reaches convergence. E-Step assumes that transition matrices are fixed, and assigns each sequence to a cluster that maximizes the probability of observing that sequence in the given data. M-Step assumes that cluster assignments are fixed, and updates the transition matrix that can best explain the cluster assignments. For experimental purposes, we have set the convergence condition as one in which there is little change (10^{-4}) in the log-likelihood of the data.



Time	User	Resource
00:01:01	A	1
00:01:05	A	2
00:01:10	B	1
00:01:11	A	3
00:01:15	B	3
00:01:17	A	4
00:01:18	B	4

Figure 4: Traces of two users from log files modeled as paths on a graph. Nodes represent resources or components (e.g. a server) and edges represent the transition of the users interaction from one software component to another.

Algorithm 1 Model-based sequence clustering algorithm

```
Input: Number of clusters(K) and sequence data
Output: Cluster assignments and scores
Initialize cluster assignments $z_1,z_2,...z_n$
Initialize transition probability matrices $\theta_1,\theta_2,...\theta_k$
while Not Converged do
    E-Step: Assign each sequence to most likely cluster
    for $i=0$ to $n$ do
        $z_i$= $\text{argmax}_{z_i} p(x|c_i)$
    end for
    M-Step:Update transition matrices $\theta_1,\theta_2,...\theta_k$
    for $z=1$ to $k$ do
        for all sequences in cluster $z$ do
            Count number of transition edges from $a$ to $b$ in the sequence.
            Update transition matrix $\theta_z$
        end for
    end for
end while
```

However, the model also requires as input the number of clusters. For large datasets, figuring out the number of clusters can be a problem because sequence clustering is sensitive to this parameter. A different number of clusters could potentially produce very different results, which might not be consistent with what we are trying to capture, or explain the data well. To come up with an appropriate number of clusters, we employ the Bayesian information criterion [29] as described in Algorithm 2.

Algorithm 2 BIC Algorithm

```
Input: Candidate range of number of clusters $(k_min,k_max)$
Input: Number of iterations $iters$
Output: Number of clusters
for $k=k_min$ to $k_max$ do
    for $i=0$ to $iters$ do
        Cluster using Algorithm 1.
        Compute log-likelihood $l(D;\theta)$for partition obtained using Algorithm 1 .
        Calculate BIC value = $l(D;\theta)-\frac{k\log(n)}{2}$
    end for
    Choose highest BIC value for this cluster $k$
end for
Return $k$ corresponding to highest BIC values among all clusters
```

4.3 Scoring

We need to assign to each sequence a suspiciousness score. This is the likelihood of the occurrence of a particular sequence, given the model of the system as represented by a set of sequences accepted to be normal behavior. Thus, the suspiciousness score is a statistic based on the likelihood of the data. Low likelihood would mean that the given model cannot explain this particular sequence, thus establishing that sequence as an outlier. Therefore, the suspiciousness score is inversely proportional to the likelihood of the model. However, the score also takes into account the length of the sequence because a longer the trace has

[1] The order of the Markov chain can be increased based on scalability and the history of states to include.

moreterms, which might lead to a higher negative log-likelihood. Therefore we define the suspiciousness score of a given sequence as follows:

$$susp_u = \frac{-\log Likelihood(u)}{len_u}$$

5. EXPERIMENTS

We conducted experiments to answer the following questions:

- **Q1. Effectiveness**: How well does the approach perform in terms of precision and recall?
- **Q2. Signal to Noise Ratio**: How does the proposed approach scale when decreasing signal to noise ratio?
- **Q3. Effect of architecture size**: How does the architecture's size affect the accuracy of the proposed approach?
- **Q4: Effect of abstraction function**: How do the results change given different abstraction functions?

We compared our approach with two naive baseline methods: unigram and bigram. Each baseline approach is based on the distribution of the activities in a sequence. Unigram is based on the distribution of activities while bigram, is based on distribution of pair of activities. In our approach, we used used GFADD [24], which computes an outlier score based on the density of the set of points in the given feature space, to compute outlying sequences. This approach is similar to the approaches that take into account activity distribution of users [2, 16].

In this section, we first present out dataset generation setup, and then we consider the four questions noted earlier.

5.1 Dataset

We model the architecture used for evaluating our approach after the reference architecture employed for Amazon Web application hosting.[2] An example of an instance that we employed for our experiments is shown in Figure 1. We incorporate two geographical areas (U.S. and Europe), each of which includes:

- A *content delivery network* (e.g., Amazon CloudFront) that delivers dynamic content to users and optimizes performance via smart request routing.
- A set of *load balancers* that distribute incoming application traffic among multiple Web application servers.
- A set of *application servers* to process requests that may be deployed, for example, on Amazon EC2 instances.
- *Columnar Databases* that host application data specific to a particular region.

In addition to these geographical zones, the architecture also includes high-availability *distributed databases* that host additional application data and are shared by the different geographical areas (represented as node clusters on the right-hand side of Figure 1). This architecture style is a classic N-tier Cloud system, of the kind that is used by a number of large organizations to store their data and provide scaleable access to many clients.

Unfortunately, we could not have access to a real, large-scale system. Therefore, validation of our approach began with the generation of a large scale software architecture, where we could simulate realistic behavior of a large number of users. The first step was to codify the kinds of components and connectors, and rules governing their correct configuration, into an Acme style [13] that could be used to govern the generation of a correct architecture. To generate the architecture of an Amazon Web application, we translated this style to Alloy [18], a language based on first-order logic that allows modeling structures (known as *signatures*) and the relations between them as constraints. In particular, we use Alloy to formally specify the set of topological constraints of the architecture, and then use the Alloy analyzer tool to automatically generate models that satisfy those constraints. Listing 1 shows the encoding of the basic signatures for part of the Amazon Web architecture that includes the declaration of content delivery network components, load balancers, and some of the topological constraints that determine how components of these types should be connected among them.

```
abstract sig comp { conns : set comp } // Abstract component
//No component must be connected to itself
fact { all n:comp | not n in n.conns }
//Content Delivery Network, Load Balancer, and Application Server
sig CDN, LB, AS extends comp { }
// All components must be reachable at least from one CDN
fact { some c:CDN | all n:comp−CDN | n in c.∗conns }
//CDNs must be connected only to LBs
fact { all c:CDN.conns | c in LB }
//Each CDN must be connected at least to some LB
fact { all c:CDN | some l:LB | l in c.conns }
// LBs must be connected only to AS
fact { all l:LB.conns | l in AS }
...
```

Listing 1: Alloy architecture specification excerpt.

Since Alloy does not scale to generate models satisfying the constraints with an arbitrary number of instances of the different signatures, we divided our Alloy specification into two parts that describe different sub-architectures: (i) geographical area, including content delivery networks, load balancers, application servers, and databases, and (ii) distributed databases. Next, we ran the Alloy analyzer several times on (i) and (ii) to obtain different instances of the sub-architectures, that we then merged in order to obtain an architecture much larger than what we would be able to construct by directly trying to generate the overall architecture from a monolithic Alloy specification.

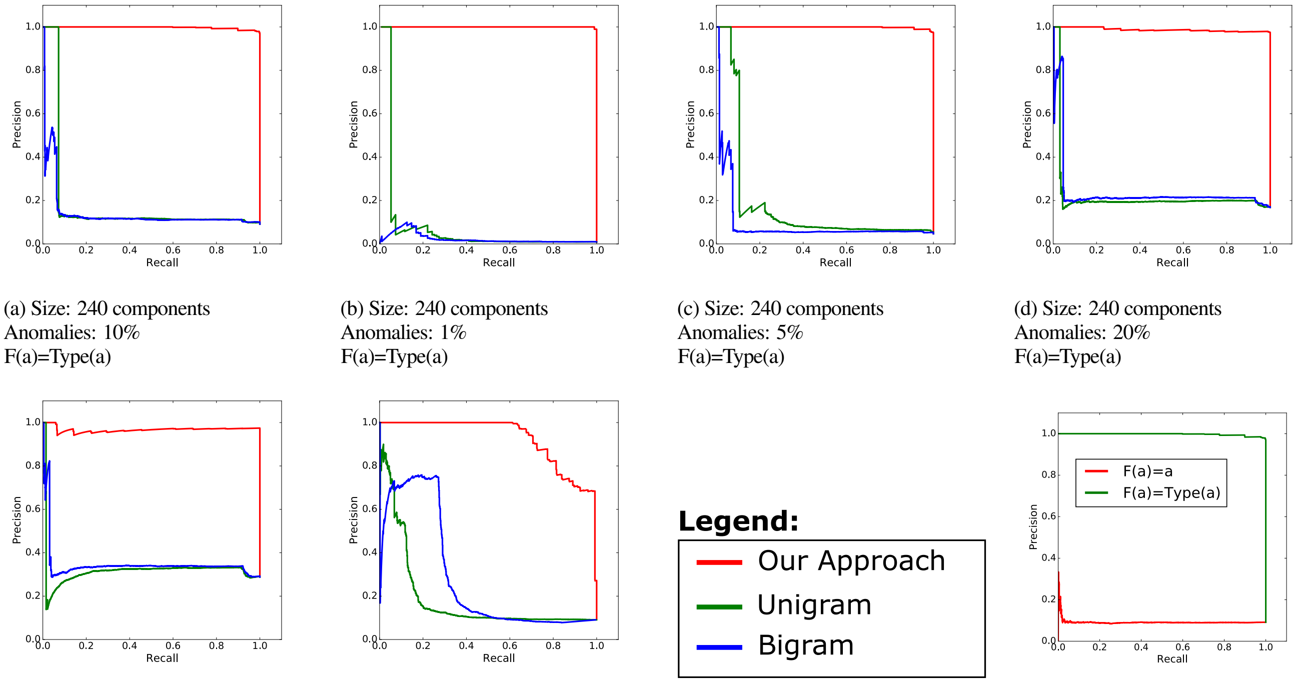
This generated architecture is then input to a simulator that creates user traces by following valid connections within the software architecture. Transitions from one node within the architecture to another are chosen randomly among the valid connections. These transitions are performed after a simulated "think time" that is chosen randomly between 1 and 30 seconds. At each node the trace has a randomly chosen probability, between 1% and 25%, of "returning" (i.e., being the end point of the user trace). The path is then reversed back through the system again with another simulated “think time”.

Additionally, two different patterns of anomalies were randomly injected into the dataset, at rates varying between 1% and 40%, depending on the experimental run. The first type of anomaly had a "star" pattern where the traces interacted with a set of nodes, each interaction originating at the same node. The second is a linear trace through a randomly determined set of nodes. These patterns are representative of the different types of attack scenarios discussed in Section 3. The output of the simulations was a set of traces, which included both normal and suspicious behaviors.

5.2 Results

Effectiveness. We apply the approach to the simulated dataset with the injected anomalies. Keeping the architecture size at 240 nodes, and the percentage of injected anomalies at 10%, we compared our approach with the other baseline approaches. To measure the effectiveness, we use standard precision and recall metrics. We show the results in Figure 5a. As you can see, our approach was able to rank all suspicious users perfectly (ideal line being y=1 till x=1) and also outperforms the baseline unigram and bigram approaches.

[2] https://aws.amazon.com/architecture/



(a) Size: 240 components
Anomalies: 10%
F(a)=Type(a)

(b) Size: 240 components
Anomalies: 1%
F(a)=Type(a)

(c) Size: 240 components
Anomalies: 5%
F(a)=Type(a)

(d) Size: 240 components
Anomalies: 20%
F(a)=Type(a)

(e) Size: 240 components
Anomalies: 40%
F(a)=Type(a)

(f) Size: 100 components
Anomalies: 10%
F(a)=Type(a)

(g) Size: 240 components
Anomalies: 10%
F(a)=Type(a) & F(a)=a

Figure 5: Precision/Recall graphs for the experiments.

Signal to Noise Ratio. To further evaluate the effectiveness of our approach, we varied the percentage of anomalies in the data while keeping the size of the architecture constant. We plot this in Figures 5b–5e. We discovered that our approach still performed consistently well with up to 40% of injected anomalies.

Effect of Architecture Size. Further, we evaluated our approach on different architecture sizes. Keeping the percentage of injected suspicious sequences as 10%, we changed the architecture size and applied our approach, using sizes of 100 and 240 total nodes. The results are presented in Figure 5f and again show that our performance is robust to the size of architecture. In fact, it appears that the baseline approaches perform worse for larger size systems (c.f., Figure 5a), perhaps because of the limited amount of contextual information that they use, which our approach exploits.

Effect of Abstraction Function. The abstraction function can have a significant impact on the performance of the the proposed approach. This is shown in Figure 5g. As illustrated, the accuracy with abstraction function $\mathcal{F}(a) = a$ is much lower than when we abstract each machine to the type of machine $\mathcal{F}(a) =$Type(a). In this experiment we looked at the least abstract and most abstract function for our scenario. This demonstrates that our approach is flexible with respect to the abstraction function, and in fact finding the right abstraction function will be key to the practical application of our approach.

6. CONCLUSIONS

In this paper we described an approach to find and score suspicious execution sequences generated by a user due to their interactions with software mapped to an underlying software architecture. Our method exploits contextual information about the technical system, specifically the software architecture, to provide an appropriate abstraction and a model-based clustering technique clusters a users sequence through this model together to determine how likley they are to have occurred. The lower the score, the more suspicious they are.

Model-based approach: The proposed model-based approach exploits contextual knowledge behind sequence generation over software architectures, and provides a robust framework for detecting suspicious sequences.

Scoring function: We proposed a likelihood-based statistic to score each user sequence on the basis of its suspiciousness.

Accurate and Scalable: We showed that when our approach was tested on simulated data, it performed better than the given baselines that used unigrams and bigrams distribution. Our approach also scaled well to different architecture sizes and different percentage of injected anomalies, although it is sensitive to the specific abstraction function used when mapping to an architecture.

For future work, we would like to do further analysis of our algorithm to understand better the scope and limitations. For example, though 240 node system represents significant sized enterprise systems, they do not represent extremely large scale systems like Netflix or Google search. Furthermore, we have only showed two ends of the spectrum on using the abstraction function: either treat each node as a unique node in the graph or group them all together using their type. Partitioning the node-space using different properties could prove interesting.

Another area of future work is to test our approach on real data. In our experiments, because of a lack of availability of real data, we relied on expertise of the kinds of traces and anomalies that are typical for such systems to guide our simulation. Despite this, we hope that real data would confirm our results. We would also like to experiment on different styles of systems.

Acknowledgments

This work is supported in part by the National Security Agency. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the National Security Agency or the U.S. government.

7. REFERENCES

[1] Akoglu, L., Chandy, R., and Faloutsos, C. Opinion fraud detection in online and by network effects. *ICWSM* (2013).

[2] Akoglu, L., McGlohon, M., and Faloutsos, C. Oddball: Spotting anomalies in weighted graphs. In *PAKDD* (2010), pp. 410–421.

[3] Beutel, A., et al. Copycatch: stopping group attacks by spotting lockstep behavior in social networks. In *WWW* (2013).

[4] Cadez, I., Heckerman, D., Meek, C., Smyth, P., and White, S. Visualization of navigation patterns on a web site using model-based clustering. In *KDD* (2000).

[5] Casanova, P., Garlan, D., Schmerl, B., and Abreu, R. Diagnosing architectural run-time failures. In *In Proceedings of the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems* (2013).

[6] Casanova, P., Schmerl, B., Garlan, D., and Abreu, R. Architecture-based run-time fault diagnosis. In *In Proceedings of the 5th European Conference on Software Architecture* (2011).

[7] Chan, P. K., and Mahoney, M. V. Modeling multiple time series for anomaly detection. In *ICDM* (2005).

[8] Cheng, H., Tan, P., Potter, C., and Klooster, S. Detection and characterization of anomalies in multivariate time series. In *SDM* (2009), pp. 413–423.

[9] Cummings, A., Lewellen, T., McIntire, D., Moore, A., and Trzeciak, R. Insider threat study:illicit cyber activity involving fraud in the US financial services sector. *Special Report, CERT, Software Engineering Institute* (2012).

[10] Ferraz Costa, A., et al. Rsc: Mining and modeling temporal activity in social media. In *KDD* (2015).

[11] Forrest, S., W. C., and Pearlmutter, B. Detecting intrusions using system calls: Alternate data models. In *In Proceedings of the 1999 IEEE ISRP* (1999).

[12] Fu, Y., Sandhu, K., and Shih, M.-Y. Clustering of web users based on access patterns. *Web Usage Analysis and User Profiling* (2000), 21–38.

[13] Garlan, D., Monroe, R. T., and Wile, D. Acme: Architectural description of component-based systems. In *Foundations of Component-Based Systems*, G. T. Leavens and M. Sitaraman, Eds. Cambridge University Press, 2000, pp. 47–68.

[14] Giatsoglou, M., Chatzakou, D., Shah, N., Faloutsos, C., and Vakali, A. Retweeting activity on twitter: Signs of deception. In *PAKDD* (2015).

[15] Gunneman, S., Gunneman, N., and Faloutsos, C. Detecting anomalies in dynamic rating data: A robust probabilistic model for rating evolution. In *KDD* (2014).

[16] Hooi, B., Shah, N., Beutel, A., Günnemann, S., Akoglu, L., Kumar, M., Makhija, D., and Faloutsos, C. BIRDNEST: bayesian inference for ratings-fraud detection. In *Proceedings of the 2016 SIAM International Conference on Data Mining, Miami, Florida, USA, May 5-7, 2016* (2016), pp. 495–503.

[17] Hooi, B., Song, H. A., Beutel, A., Shah, N., Shin, K., and Faloutsos, C. Fraudar: Bounding graph fraud in the face of camouflage. In *KDD* (2016).

[18] Jackson, D. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[19] Jiang, M., et al. Catchsync: catching synchronized behavior in large directed graphs. In *KDD* (2014).

[20] Jiang, M., et al. Inferring strange behavior from connectivity pattern in social networks. In *PAKDD* (2014).

[21] Keeney, M., Capelli, D., Kowalski, E., Moore, A., Shimeall, T., and Rogers, S. Insider threat study: Computer sabotage in critical infrastructure sectors. In *CERT Program and Software Engineering Institute* (2005).

[22] Kowalski, E., Cappelli, D., and Moore, A. Insider threat study: Illicit cyber activity in the government sector. *Software Engineering Institute* (2008).

[23] Kowalski, E., Cappelli, D., and Moore, A. Insider threat study: Illicit cyber activity in the information technology and telecommunications sector. *Software Engineering Institute* (2008).

[24] Lee, J. Y., Kang, U., Koutra, D., and Faloutsos, C. Fast anomaly detection despite the duplicates. In *WWW Companion* (2013).

[25] Li, X., and Han, J. Mining approximate top-k subspace anomalies in multi-dimensional time-series data. In *VLDB* (2007), pp. 447–458.

[26] Pandit, S., et al. Netprobe: a fast and scalable system for fraud detection in online auction networks. In *WWW* (2007).

[27] Prakash, B., et al. Eigenspokes: Surprising patterns and community structure in large graphs. *PAKDD* (2010).

[28] Ramaswamy, S., Rastogi, R., and Shim, K. Efficient algorithms for mining outliers from large data sets. *SIGMOD 29* (2000), 427–438.

[29] Schwartz, G. E. Estimating the dimnesson of a model. *Annals of Statistics 6* (1978), 461–464.

[30] Shah, N., Beutel, A., Gallagher, B., and Faloutsos, C. Spotting suspicious link behavior with fbox: An adversarial perspective. In *ICDM* (2014).

[31] Shah, N., Beutel, A., Hooi, B., Akoglu, L., Gunnemann, S., Makhija, D., Kumar, M., and Faloutsos, C. Edgecentric: Anomaly detection in edge-attributed networks. *arXiv preprint arXiv:1510.05544* (2015).

[32] Shaw, M., and Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[33] Shin, K., Hooi, B., and Faloutsos, C. M-zoom: Fast dense-block detection in tensors with quality guarantees. In *ECML/PKDD* (2016).

[34] Teng, H., C. K., and Lu, S. Adaptive real-time anomaly detection using inductively generated sequential patterns. In *Proceedings of IEEE Computer Society Symposium on Research in Security and Privacy* (1990).

[35] Vahdatpour, A., and Sarrafzadeh, M. Unsupervised discovery of abnormal activity occurences in multi-dimensional time series with applications in wearable systems. In *SDM* (2010), pp. 641–625.

[36] Xie, S., Wang, G., Lin, S., and Yu, P. Review spam detection via temporal pattern discovery. In *KDD* (2012), pp. 823–831.

[37] Ye, J., and Akoglu, L. Discovering opinion spammer groups by network footprints. In *COSN* (2015).