Informer: Irregular Traffic Detection for Containerized Microservices RPC in the Real World

Jiyu Chen
University of California, Davis
jiych@ucdavis.edu

Heqing Huang
ByteDance Inc.
huangheqing@bytedance.com

Hao Chen
University of California, Davis
chen@ucdavis.edu

ABSTRACT

Containerized microservices have been widely deployed in industry. Meanwhile, security issues also arise. Many security enhancement mechanisms for containerized microservices require predefined rules and policies. However, it is challenging when it comes to thousands of microservices and a massive amount of real-time unstructured data. Hence, automatic policy generation becomes indispensable. In this paper, we focus on the automatic solution for the security problem: irregular traffic detection for RPCs.

We propose Informer, which is a two-phase machine learning framework to track the traffic of each RPC and report anomalous points automatically. Firstly, we identify RPC chain patterns by density-based clustering techniques and build a graph for each critical pattern. Next, we solve the irregular RPC traffic detection problem as a prediction problem for time-series of attributed graphs by leveraging spatial-temporal graph convolution networks. Since the framework builds multiple models and makes individual predictions for each RPC chain pattern, it can be efficiently updated upon legitimate changes in any of the graphs.

In evaluations, we applied Informer to a dataset containing more than 7 billion lines of raw RPC logs sampled from an large Kubernetes system for two weeks. We provide two case studies of detected real-world threats. As a result, our framework found fine-grained RPC chain patterns and accurately captured the anomalies in a dynamic and complicated microservice production scenario, which demonstrates the effectiveness of Informer.

KEYWORDS

containers, microservices, GCN, RPC, anomaly detection

ACM Reference Format:
Jiyu Chen, Heqing Huang, and Hao Chen. 2019. Informer: Irregular Traffic Detection for Containerized Microservices RPC in the Real World. In *SEC '19: ACM/IEEE Symposium on Edge Computing, November 7–9, 2019, Arlington, VA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3318216.3363375

1 INTRODUCTION

The containerized microservice architecture, which makes each module of the application loosely-coupled, easy to maintain, and more elastic for dynamic service volumes, is prevalently applied by companies to provide all kinds of internal applications and public services. In 2014, Google released Kubernetes [8], aiming for automated container deployment, scaling, and management, which has become the standard container orchestration platform in the industry. Meanwhile, security issues inside the architecture are frequently exposed by the community, and security strategies are attracting more concerns. For example, many security enhancement mechanisms require rules and policies to control access, resources, and behavior of each container. However, in large enterprises that maintain various applications, there can be thousands of containers. It is infeasible to make policies for each container manually. Therefore, policy automation becomes an indispensable direction in security solutions for such complex systems.

In this paper, we are interested in the following problem: since microservices are deployed in different containers and machines, they need to communicate by remote procedure calls (RPCs) to provide the complete functionality. Once some of the containers were compromised, or malicious users were abusing the public APIs we provided, there could be unusual changes in RPC traffic. Our research question is, can we make irregular RPC traffic detection automatically in a simple yet effective framework?

To that end, we aim to design a machine learning framework for irregular RPC traffic detection by predicting future traffic, and detect anomalous traffic activities based on the predictions. We propose to represent RPC traffic at each time point as a directed, weighted, and attributed graph where each node stands for an RPC. The attributes of each node include RPC traffic, which is how many times the RPC is called in a fixed time period. The edges and weights represent the dependency among these RPCs, such as the order of the calls. With such representations, we can obtain a time-series of graphs after observing the RPC logs for a while. Hence, our task becomes making predictions given a time-series of graphs.

Machine learning techniques have shown promising performance in various security-related tasks, such as malware detection and intrusion detection. However, traditional machine learning models can hardly handle graphs and time-series simultaneously. To address this problem, graph convolution networks (GCNs) have attracted much attention in recent year. Compared with traditional graph analysis techniques, GCNs are advanced in their capability of extracting spatial information from the complex graph-structured data. Moreover, they can also be combined with other deep learning modules such as recurrent units to extract temporal information, which becomes spatial-temporal graph convolution networks.

After further analyzing the real-world RPC data, we found that the main challenge of modeling RPC graphs by GCNs is the large number of unique RPCs existing in the system. It would be time-consuming, error-prone, and hard-to-update if we built a unified

SEC '19, November 7–9, 2019, Arlington, VA, USA

ACM ISBN 978-1-4503-6733-2/19/11…$15.00
https://doi.org/10.1145/3318216.3363375

model to track all the RPCs at the same time. However, one fact is that not all RPCs are related to each other. Usually, one RPC is only dependent on a small group of RPCs in an RPC chain to finish a target functionality. Hence it would be a good idea to build independent models for different groups of RPCs.

Taking into account the aforementioned considerations, we propose Informer, a two-phase irregular RPC traffic detection framework. In the first phase, we define the distance between two RPC chains and apply the density-based clustering technique DBSCAN to find different RPC chain patterns. In the second phase, we build a spatial-temporal graph convolution network DCRNN for each critical RPC chain pattern, instead of a unified model. Then we make predictions and anomaly detection based on the observations of the RPC traffic in previous timesteps. In evaluations, we demonstrate the effectiveness of Informer by applying it to a large dataset sampled from raw RPC logs in a real-world Kubernetes production system, which faces billions of daily active users. The results show that our framework is capable of finding fine-grained RPC chain patterns and accurately predicting the behavior of all RPCs.

2 BACKGROUND

Containerized microservices. The microservice is an architecture that decouples an application into multiple individule services. Each microservice works independently for a small functionality of the application and locates in a different container. A container is a standard unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another [4]. The containerized microservice architecture benefits from its advantage of high maintainability and has become the mainstream strategy of application deployments. Currently, the most widely used system for automating deployment, scaling, and management of containerized applications is Kubernetes [8]. Though having much success, the containerized microservice architecture also has its fallbacks. One main limitation is that the complexity of the entire system increases significantly with the number of microservices.

Graph Convolution Networks. Graph convolution networks are designed to learn features from complex graph-structured data. Graphs are non-euclidean, which means regular image convolution layers cannot be directly applied to graphs. In general, there are two ways to define graph convolutions. The first is the spectral graph convolution, which leverages the spectral graph theory [7]. The second is the spatial graph convolution, which samples the neighbors of each node and aggregates their features for each filter. Since spectral convolutions only support undirected graphs, we only consider spatial convolutions. The diffusion convolution [1] is one of the spatial graph convolutions designed to train on directed graphs. The diffusion graph convolution layer is defined as:

$$\mathbf{H} = \sum_{k=0}^{K} f(\Theta^{(k)}(\mathbf{D}^{-1}\mathbf{W})^{k}\mathbf{X})$$

where $f(\cdot)$ is the activation function, $\mathbf{D} = diag(s_i)$ is the diagonal matrix which contains the sum s_i of each row, $\mathbf{A}$ is the attribute matrix, and $\Theta^{(k)}$ is the parameter of the k_{th} filter.

Graph convolution networks are deep neural networks with graph convolution layers. Classic deep learning architectures for images and texts can be combined with GCNs, such as graph autoencoders [2, 13], graph attention networks [11, 15], spatial-temporal graph convolution networks [9, 14].

3 METHODOLOGY

3.1 Data representation

Before discussing how we process and represent data, we provide definitions to prevent confusion in the context:

- **RPC** RPCs or remote procedure calls are made between two methods in different containers to provide a functionality collaboratively. Generally, we make each container locate on a (logically) different machine. Note that there can be multiple methods inside the same container. We can consider either fine-grained RPCs made between two methods or coarse-grained RPCs made between two containers, depends on our requirements and computation resources. Moreover, the same container can be duplicated and deployed on multiple machines to provide concurrency, so we can also consider even more fine-grained RPCs made between two pairs of (method, container, machine).
- **RPC traffic** The traffic of an RPC is the number of times the RPC is called during a fixed period of time.
- **RPC log** The system will log each RPC, which is the raw RPC log. Fields of each log include the source method/container, destination method/container, and the timestamp. An RPC log also contains a field of chain ID identifying which instance of functionality that the RPC belongs to.
- **RPC chain** A functionality usually requires a set of RPCs. These RPCs can form a chain of calling dependencies, which we refer as an RPC chain. By gathering all the RPC logs with the same chain ID and ordering them by time, we can obtain an *RPC chain instance*. The RPC chain instances can vary for the same functionality, depends on real-time conditions. In the context of this paper, each model will be built upon an *RPC chain pattern* which contains all RPCs that are possibly required by the functionality.
- **RPC graphs** A static RPC graph $\mathcal{G}_{\text{static}} =< \mathbf{V}, \mathbf{E}, \mathbf{W} >$ is a graph build from a set of related RPCs, where $\mathbf{V}$ is the node set with each node representing an RPC and $\mathbf{E}$ is the edge set with each edge representing an RPC dependency. A temporal RPC graph $\mathcal{G}_t =< \mathcal{G}_{\text{static}}, \mathbf{X}_t >$ is a static RPC graph with an attribute matrix $\mathbf{X}_t$ at timestep t.

Now we introduce how we build RPC graphs. The procedure for generating static RPC graphs is in algorithm 1. The RPC set $\mathbf{C}$ stores RPCs in the form of (src, dst). We directly assign the RPC set $\mathbf{C}$ to the node set $\mathbf{V}$, which means nodes in the graph are RPC source and destination pairs. There is an edge between two nodes when they share the same source or destination, specifically: when A and B are dependent (one's destination is another's source), there is a directed edge from A to B (or B to A) with weight 1.0; when A and B share same source or destination, there is both an edge from A to B and from B to A with weight 0.5 (or any empirical value depends on the real situations).

To build the temporal RPC graph, we go through the raw RPC logs, compute the traffic (and other attributes that we are interested

Algorithm 1: Generate RPC graph from a given RPC set.

```
Input: An RPC set $S$
Result: The static RPC graph G, The adjacency matrix $A$
$V \leftarrow S$;
$E \leftarrow$ empty_set();
$A \leftarrow$ empty_matrix(shape = V.len() × V.len());
for $i \leftarrow 0$ to $V$.len() do
    for $j \leftarrow i$ to $V$.len() do
        if $V[i]$.src == $V[j]$.src or $V[i]$.dst == $V[j]$.dst then
            $E$.add([($V[i], V[j]$), ($V[j], V[i]$)]);
            $A[i, j] \leftarrow 0.5$;
            $A[j, i] \leftarrow 0.5$;
        end
        if $V[i]$.src == $V[j]$.dst then
            $E$.add(($V[j], V[i]$));
            $A[j, i] \leftarrow 1$;
        end
        if $V[i]$.dst == $V[j]$.src then
            $E$.add(($V[i], V[j]$));
            $A[i, j] \leftarrow 1$;
        end
    end
end
$G \leftarrow (V, E)$;
return $G, A$;
```

in) for each time period. Intuitively, RPC traffic should be calculated from RPC chain instances but not unordered raw RPC logs. However, in real-world, since the number of new logs per second is way too large for us to extract complete RPC chain instances, we can hardly obtain a complete chain. Instead, we can only compute from the raw data where chains are mixed.

3.2 RPC chain pattern mining

In real-world, a microservice system can have thousands of different microservices distributed in even more containers. It is unfavorable to build a unified model for the entire RPC set. The main reason is that it is hard to maintain the large model when new RPCs are coming in, and old RPCs are being deprecated very frequently. It costs expensive resources to retrain the model. The second reason is that sometimes we only want to track a small subset of RPCs.

Instead of the unified model, we build an independent model for each RPC chain pattern, since RPCs made inside a chain pattern are highly related, while RPCs made among chain patterns are not, which is illustrated in Figure 1. Thus, the first phase of our framework is identifying all RPC chain patterns from a set of RPC chain instances.

Formally, suppose $\mathbf{S}$ is the set of all RPCs, and $\mathbf{C} = \{\mathbf{C}_1, ..., \mathbf{C}_n | \mathbf{C}_i \subseteq \mathbf{C}\}$ denotes a set of RPC chain instances we observed for a long enough time period. Our task is to find a set $\{\mathbf{S}_1, ..., \mathbf{S}_k | \mathbf{S}_i \subseteq \mathbf{S}\}$, where each element $\mathbf{S}_i$ is an RPC chain pattern that contains all the RPCs that are dependant and related to a same functionality.

We propose to apply clustering techniques, under the observation that RPC chain instances of the same functionality have similar RPCs. Since we do not know the total number of different chain patterns in advance, we leverage the density-based clustering method DBSCAN [5]. For clustering, we define the distance metric between two sets $\mathbf{A}$ and $\mathbf{B}$ by the *overlap coefficient* [12]:

$$d(\mathbf{A}, \mathbf{B}) = 1 - \frac{|\mathbf{A} \cap \mathbf{B}|}{\min\{|\mathbf{A}|, |\mathbf{B}|\}}$$

The procedure for RPC chain clustering is shown in algorithm 2. Note that two different RPC chain patterns $\mathbf{S}_i$ and $\mathbf{S}_j$ may contain same RPCs, which means there exist some chain instances $\mathbf{C}_l, \mathbf{C}_i \subseteq \mathbf{S}_i, \mathbf{C}_j \subseteq \mathbf{S}_j$ such that $d(\mathbf{C}_i, \mathbf{C}_l) = d(\mathbf{C}_j, \mathbf{C}_l) = 0$, so that two chain patterns might be combined into one cluster. To eliminate the influence of shared RPCs, we put all the $\mathbf{C}_i$ into a noise point set $\mathbf{R}$ which satisfy: $\exists j \neq i$, such that $d(\mathbf{C}_i \subsetneq \mathbf{C}_j$. The rest $\mathbf{C}_i$ follow the original clustering algorithm. Finally, each of the RPC chain pattern is the union of all the RPC chain instances in each cluster.

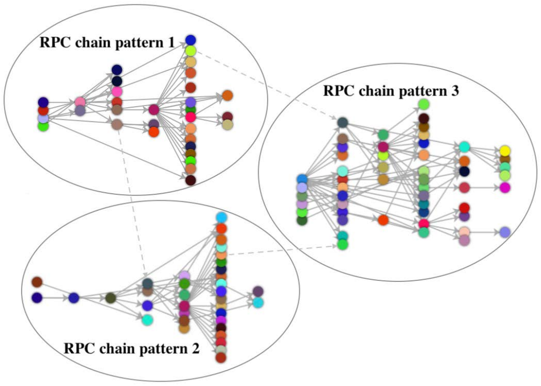


Figure 1: Illustration of RPC chain patterns. Each circle is an RPC chain pattern. Solid arrows are intra-cluster dependencies, and dotted arrows are inter-cluster dependencies.

Algorithm 2: RPC chain clustering

```
Input: An RPC chain instance set $C$, min points in a cluster
        $min\_pts$, epsilon $eps$, distance function $d$
Result: The RPC chain pattern set $S$
$R \leftarrow$ empty_set();
for $i \leftarrow 0$ to $C$.len() do
    for $j \leftarrow i + 1$ to $C$.len() do
        if $d(C[i], C[j]) == 0$ then
            $R$.add(min_len($C[i], C[j]$));
        end
    end
end
$C \leftarrow C$.substract($R$);
$clusters \leftarrow$ DBSCAN($C, min\_pts, eps, d$);
for $cluster$ in $clusters$ do
    $S$.add(union(cluster))
end
return $S$;
```

After we obtained the RPC chain patterns set $\mathbf{S}$, we can either perform irregular traffic detection for all RPC chain patterns or only select the RPC chain patterns that contain RPCs of our interests (e.g., RPC that creates a new user). This phase can significantly decrease the overhead when we need to update the models since each model works independently.

3.3 Irregular RPC traffic detection

Now we address the irregular RPC traffic detection problem for a selected RPC chain pattern. Assume that we have a stable RPC chain pattern, which means no RPC will be modified during a long enough time period. Essentially, we have a time-series of attribute matrices with the same static graph, and we want to predict the attributes of next (or next several) timestep based on previous observations.

Formally, we have a static graph $\mathcal{G} =< \mathbf{V}, \mathbf{E}, \mathbf{W} >$ of the RPC chain pattern, where $\mathbf{V}$ is the node set, $\mathbf{E}$ is the edge set and $\mathbf{W}$ is the weighted adjacency matrix. At each timestep t, the attributes of each node is represented as an matrix $\mathbf{X}_t \in \mathbf{R}^{n \times m}$ where $n = |\mathbf{V}|$ is the number of nodes and m is the number of attributes. When we only consider the traffic, $m = 1$. Then the irregular RPC traffic detection problem becomes given a time-series $\mathbf{X}$: $[\mathbf{X}_{t-s}, ..., \mathbf{X}_{t-1}]$, make prediction of the next k time steps $[\tilde{\mathbf{X}}_t, ..., \tilde{\mathbf{X}}_{t+k-1}]$, and learn a function $f(\mathbf{X}, \tilde{\mathbf{X}}) : \mathbf{R}^{n\times m} \times \mathbf{R}^{n\times m} \longrightarrow \mathbf{R}$ which take as input the predictions and the real observations, and output the anomaly classifications of the observations.

We propose to apply spatial-temporal graph convolution networks to simultaneously learn spatial features of the graph and temporal features of the time-series. Spatial-temporal graph convolution networks are GCNs that combine with temporal units such as the Gated Recurrent Unit (GRU) [3] to learn from graph time-series. In our work, we leverage the Diffusion Convolution Recurrent Neural Network (DCRNN)[9] to model our graphs. The DCRNN leverages bidirectional diffusion convolutions to take into account both upstream and downstream neighbors of each node. The bidirectional diffusion convolution is defined as:

$$\mathbf{\Theta} \star_{\mathcal{G}} \mathbf{X} = \sum_{k=0}^{K} (\theta_1^{(k)} (\mathbf{D_W}^{-1}\mathbf{W})^k + \theta_2^{(k)} (\mathbf{D_{W^\top}}^{-1}\mathbf{W}^\top)^k)\mathbf{X}$$

where $\mathbf{\Theta} = [\theta_1 \theta_2]$ is the filter parameters, $\mathbf{X}$ is the attribute matrix, K is the number of diffusion steps, $\mathbf{W}$ is the adjacent matrix, $\mathbf{D_W}$ is the diagonal matrix of the sum of each of the rows in $\mathbf{W}$.

Combining the diffusion convolution layer with the GRU, we get the DCGRU, which is defined as follows:

$$\mathbf{r}^{(t)} = \sigma(\mathbf{\Theta}_r \star_{\mathcal{G}} [\mathbf{X}^{(t)}, \mathbf{H}^{(t-1)}] + \mathbf{b}_r)$$

$$\mathbf{u}^{(t)} = \sigma(\mathbf{\Theta}_u \star_{\mathcal{G}} [\mathbf{X}^{(t)}, \mathbf{H}^{(t-1)}] + \mathbf{b}_u)$$

$$\mathbf{C}^{(t)} = \tanh(\mathbf{\Theta}_C \star_{\mathcal{G}} [\mathbf{X}^{(t)}, (\mathbf{r}^{(t)} \odot \mathbf{H}^{(t-1)})] + \mathbf{b}_C)$$

$$\mathbf{H}^{(t)} = \mathbf{u}^{(t)} \odot \mathbf{H}^{(t-1)} + (1 - \mathbf{u}^{(t)}) \odot \mathbf{C}^{(t)}$$

where $\mathbf{\Theta}$ are filter parameters, $\mathbf{X}^{(t)}$ and $\mathbf{H}^{(t)}$ is the input and the output of time step t.

On top of the DCGRU layers, the DCRNN refers to the famous seq2seq model which leverages an encoder-decoder architecture [10] to predict the attributes for each RPC simultaneously.

After we obtained the predictions of the RPC traffic from the DCRNN, we can perform anomaly detection. The most straightforward way would be manually setting thresholds on the prediction loss. On the other hand, it would be better if we automated the setting of thresholds under the assumption that the noises between the observations and the real underlying patterns, which are approximated by the models, satisfy the normal distribution. We can set the threshold using the testing data as follows:

1. Compute the mean μ and the standard deviation σ of the test errors from the predictions;
2. The errors satisfy the empirical rule, so we set the upper/lower thresholds for the predictions of timestep t as $(\tilde{\mathbf{X}}_t + \mathcal{M} \pm 3 * \Sigma)$, where $\mathcal{M}, \Sigma \in \mathbf{R}^{n\times m}$ are the mean value matrix and the standard deviation matrix for each entry in $\mathbf{E}_t = \mathbf{X}_t - \tilde{\mathbf{X}}_t$.

4 EVALUATION

4.1 Experiment configurations

4.1.1 Dataset preparation. In section 3, building the framework requires two data sets: a set of RPC chain instances, and a series of attribute matrices. In our experiments, we uniformly sampled 10^4 RPC chain IDs within 24 hours, which are then used for finding the RPC chain instances. In the experiment, after we clustered these chain instances into chain patterns, we selected an RPC chain pattern with 51 RPCs that are related to user services.

The attribute matrices are generated from logs which are uniformly and real-timely sampled raw RPC logs from a real-world Kubernetes system. Due to the massive data traffic, we only sampled a small portion of raw logs. Specifically, we generated a data point for a time interval of $\gamma = 20$ (minutes), with around 7-million lines of sampled raw RPC logs per interval.

We continuously sampled for two weeks, leading to a dataset with $\frac{60}{20} \times 24 \times 7 \times 2 = 1008$ data points. We set 80% of the dataset to be the training set, and the rest to be the validation/test set. Since the magnitude of the traffic varies from 0 to 10^5, we took logarithms of the RPC traffic in the training process to reduce data fluctuations and took exponentiations in evaluations.

4.1.2 Models. We have two models in the framework:

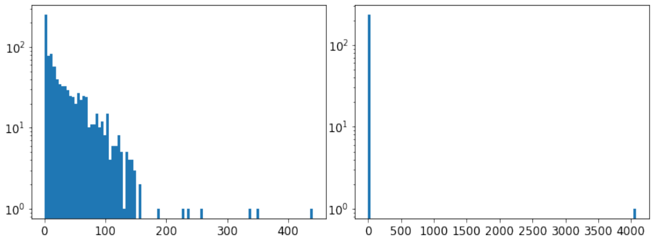
DBSCAN: As described in algorithm 2, we apply the DBSCAN clustering algorithm to obtain chain patterns. The parameters of the DBSCAN are as follows: the minimum number of points inside a cluster *min_pts*=1, the radius for neighbor searching *eps*=0.05.

DCRNN: The DCRNN model has 2 layers of DCGRU with bidirectional diffusion convolution. Each DCGRU has 64 RNN units. The maximum diffusion step K=2, and the model will predict attribute matrices in 5 future timesteps. Some other training parameters are listed as follows: using the Adam optimizer, learning rate=0.01, learning rate decay ratio=0.1, max epoch=100 with early stopping. The detailed parameters can be found in [9].

4.1.3 Environment. The experiments were run in Python3.7 + Tensorflow 1.13, on an Intel Xeon E5-2630v4 CPU, and an NVIDIA TESLA V100 GPU.

4.2 RPC chain mining

We compare clustering with a simple strategy: building a large graph containing the union of all the RPCs inside the 10^4 RPC chain instances by algorithm 1, and then find RPC chain patterns by looking for connected components inside the large graph. In the end, each connected component is an RPC chain pattern.



(a) DBSCAN **(b) Connected components**

Figure 2: Histogram of the number of unique RPCs inside a RPC chain pattern obtained by two methods.

Figure 2 is the histogram showing the number of unique RPCs inside each RPC chain pattern. Since many RPCs work individually and independently, we can see that most of the RPC chain patterns obtained by both methods contain a single RPC. Besides, from Figure 2a we can see that all the chain patterns obtained by DBSCAN clustering have tens to hundreds of unique RPCs. Meanwhile, in Figure 2b there is a dominant chain pattern with more than 4000 RPCs, and the rest chain patterns are all tiny RPC patterns.

This is reasonable since many RPC chain patterns contain a same subset of RPCs, there definitely will be a dominant connected component containing most of the RPCs inside the graph, leading to the situation of building a large unified model which we want to circumvent. Instead, by applying the clustering strategy, we can find more fine-grained RPC chain patterns with smaller scales and make our model more light-weighted and flexible.

4.3 Irregular RPC traffic detection

Table 1 shows the performance of the trained model for our selected RPC chain pattern. We quantify the model's prediction performance from three different metrics:

- Mean Absolute Error MAE $= \frac{1}{n}\sum_{i=1}^{n}|\tilde{x}_i - x_i|$
- Mean Absolute Percentage Error MAPE $= \frac{1}{n}\sum_{i=1}^{n}|\frac{\tilde{x}_i - x_i}{x_i}|$
- Root Mean Square Error RMSE $= \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\tilde{x}_i - x_i)^2}$

We can see that the model can make pretty well prediction on the future five steps, while the first prediction has the best performance.

Table 1: Model performance on 5-step future predictions

Error t	MAE	MAPE	RMSE
1	0.24	0.0379	0.33
2	0.24	0.0393	0.35
3	0.24	0.0390	0.34
4	0.25	0.0400	0.35
5	0.25	0.0404	0.35

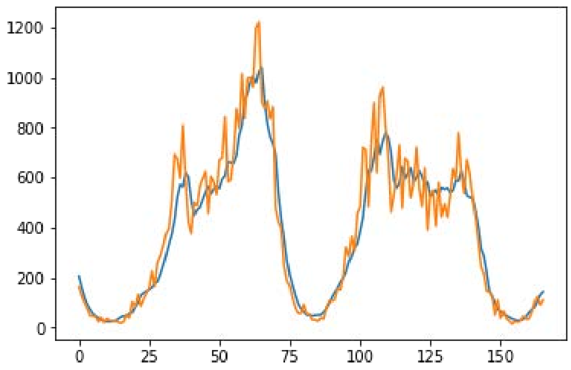


Figure 3: Traffic prediction of a randomly selected RPC. X-axis is the timeline, Y-axis is the traffic. The orange curve is the real observation, and the blue curve is the prediction.

Figure 3 shows the predictions of a randomly selected RPC of the coming two days. We can observe that though the traffic has a periodical changing trend, there exists no universal pattern. Nonetheless, we see that the model can well-capture the changing trend of the RPC traffic with a smooth prediction curve, despite the noises in the real-world data, indicating the model indeed is capable of making predictions based on the observations of the past timesteps.

4.4 Case study

We perform two case studies of real-world malicious traffic to demonstrate the effectiveness of Informer in anomaly detection.

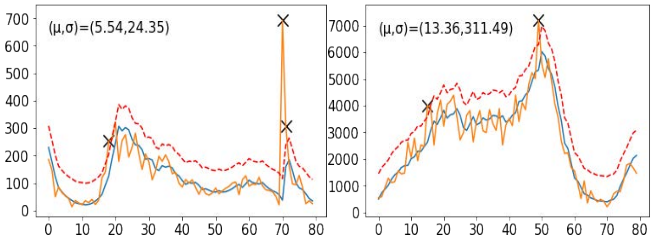
Case study 1: Batch resgiration. Batch registration of bot accounts is illegal behavior that is commonly found in real-world applications. These bot accounts will be used for other hacking services for the black market, from fake followers to scamming. Maintainers of the applications need to detect the bot accounts as sooner to the time they are registered as possible.

In this case, we focus on the RPC that is used to perform human-machine validation, which is a mandatory step for account registration. Each registration at least requires one (but not too many) human-machine validation RPC. When malicious users did batch registration, the traffic of this RPC would significantly increase.

Case study 2: Account cracking. Account cracking is another case where malicious users abuse the public API. Currently, most of the applications support retrieving forgotten accounts, which have been bound to a mobile number, by the Short Message Service (SMS) via mobile phone. Once the users type in the correct validation codes sent by the service, they will be validated as legal users.

In this case, we focus on the RPC that sends requests to the SMS server. If the malicious users want to crack accounts violently, they have to send a large number of requests in a short time.

Figure 4 shows the result of the case studies, where each upper threshold is computed based on the mean μ and standard deviation σ of the MAE (after exponentiations) as discussed in subsection 3.3. From Figure 4a we can see that there are three anomalous points at two significant increments of RPC traffic, the first one is at timestep 18, and the other two are at timestep 71 and 72. Similarly, from



(a) Batch registration **(b) Account cracking**

Figure 4: Case study: The orange curve is the real observation, the blue curve is the prediction, the red dotted curve is the upper threshold, and crosses are anoamlous points.

Figure 4b we can see that there are two anomalous points at timestep 15 and 50. We manually checked the raw RPC logs during these time periods and found that all these points are anomalous or at least some users made irregular behavior.

Discussion. From Figure 3 and Figure 4, we can see the real-world observations have much more noises than the predictions. It is common in real-world that the data are noisy, especially when sampling, which makes the curves fluctuate significantly. The noises in the real-world may cause false positives in the predictions. How to mitigate the influence of these noises will be an important future work when applying the model to large real-world systems.

5 RELATED WORK

The irregular RPC traffic detection is similar to road traffic forecasting, where several spatial-temporal graph convolution networks have been explored. For example, STGCN [14] combines 1-D convolution layers with graph convolution layers; ASTGCN [6] adds attention mechanisms to STGCN to further capture the dynamic spatial-temporal information; and DCRNN [9], which we leverage in the Informer framework, makes use of the diffusion process and the GRU. Deep learning models have shown excellent performance in such problems, which motivated our work. On the other hand, the core difference between our scenario and road traffic is the way to build the graph and the scale of the data.

To the best of our knowledge, we are the first to study the problem of irregular RPC traffic detection and apply state-of-the-art machine learning techniques in containerized microservices scenarios. Our framework can handle large containerized microservice system with thousands of RPCs and real-time data which can hardly be handled by a unified GCN model.

6 CONCLUSION

In the past few years, more and more companies have been deploying their applications in a distributed and containerized manner. In this paper, we focus on the automatic solution for irregular RPC traffic detection in containerized microservice productions. For microservice architectures, due to its characteristics of frequent development iteration and massive data flows, it is better to build light-weighted and distributed models instead of a unified model for irregular traffic detection.

Under such considerations, we propose a two-phase machine learning framework named Informer. It firstly extracts RPC chain patterns by clustering and then builds a graph model for each RPC chain pattern, which simultaneously learns spatial and temporal features, to predict future RPC traffic. The framework is flexible and easy to deploy since the model of each chain pattern is independent, light-weighted, and easy-to-update upon changes in the system. We evaluated our framework on billions of lines of data sampled from a large Kubernetes system. From the results of two case studies, we demonstrate the strength and effectiveness of Informer in detecting real-world threats among microservices communications.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1801751.

This research was partially sponsored by the Combat Capabilities Development Command Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-13-2-0045 (ARL Cyber Security CRA). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Combat Capabilities Development Command Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

REFERENCES

[1] James Atwood and Don Towsley. 2015. Search-Convolutional Neural Networks. *CoRR* abs/1511.02136 (2015). arXiv:1511.02136 http://arxiv.org/abs/1511.02136

[2] Shaosheng Cao, Wei Lu, and Qiongkai Xu. 2016. Deep neural networks for learning graph representations. In *Thirtieth AAAI Conference on Artificial Intelligence.*

[3] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555* (2014).

[4] Docker Inc. [n.d.]. Docker: Enterprise Container Platform. https://www.docker.com/

[5] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. [n.d.]. A density-based algorithm for discovering clusters in large spatial databases with noise.

[6] Shengnan Guo, Youfang Lin, Ning Feng, Chao Song, and Huaiyu Wan. 2019. Attention Based Spatial-Temporal Graph Convolutional Networks for Traffic Flow Forecasting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 922–929.

[7] Mikael Henaff, Joan Bruna, and Yann LeCun. 2015. Deep convolutional networks on graph-structured data. *arXiv preprint arXiv:1506.05163* (2015).

[8] Kubernetes contributors. [n.d.]. Kubernetes: Production-Grade Container Orchestration. https://kubernetes.io/

[9] Yaguang Li, Rose Yu, Cyrus Shahabi, and Yan Liu. 2017. Diffusion convolutional recurrent neural network: Data-driven traffic forecasting. *arXiv preprint arXiv:1707.01926* (2017).

[10] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. *CoRR* abs/1409.3215 (2014). arXiv:1409.3215 http://arxiv.org/abs/1409.3215

[11] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[12] MK Vijaymeena and K Kavitha. 2016. A survey on similarity measures in text mining. (2016).

[13] Daixin Wang, Peng Cui, and Wenwu Zhu. 2016. Structural deep network embedding. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining.* ACM, 1225–1234.

[14] Bing Yu, Haoteng Yin, and Zhanxing Zhu. 2017. Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting. *arXiv preprint arXiv:1709.04875* (2017).

[15] Jiani Zhang, Xingjian Shi, Junyuan Xie, Hao Ma, Irwin King, and Dit-Yan Yeung. 2018. Gaan: Gated attention networks for learning on large and spatiotemporal graphs. *arXiv preprint arXiv:1803.07294* (2018).