Cloud Service Detection Under Secured Networks

Helen Kulka
helen.kulka@mail.mcgill.ca

Bettina Kemme
kemme@cs.mcgill.ca

Mona Elsaadawy
mona.elsaadawy@mail.mcgill.ca

April 2021

1 Abstract

Network Security Protocols are ubiquitous in modern cloud applications. For large applications that are distributed among many clients (e.g. HTTP endpoint) and their resources (e.g. MySQL database), monitoring an application's status may result in a complicated call graph. In order to visualize these dynamic applications, a generic learning model has been built to identify relevant components. This model relies on network traces to predict active resources, however, application data within these traces is often encrypted. To this end, we are interested in measuring the effect of this encryption on our model's performance. We discovered that introducing secured traces to our model reduced model accuracy by 16%. Most mis-classifications occurred between Non-Secured and Secured Postgres traces.

2 Introduction

The proposed cloud management system, DyMonD, aims to clarify the structure and dependencies of a cloud application at any given moment. As more domains move to the cloud, the more complicated application call graphs become. A component may communicate only with internal components, but an increasing number of application components begin communicating with other distributed systems. For example, the checkout flow of a website may require access to a user database, a payment processor API, a Google Analytics endpoint, and an internal order confirmation endpoint. Monitoring the communication between these services, rather than monitoring the individual components themselves, is a crucial aspect of running a cloud application.

The increase in distributed network communication has been mirrored with an increase in data leaks and security attacks. To combat these attacks, internet security has been a main focus for many companies. For example, many browsers prevent users from entering sites that do not provide secure connections. The most common security protocol is Transport Layer Security, a cryptographic security method. In the context of internet browsers, TLS encryption is enabled through an HTTPS connection. However, TLS can be enabled in many different settings.

In the context of this paper, we investigate the use of TLS between Java clients and relational database systems. In a data-driven age, databases are an integral aspect of almost every cloud application. Similarly, both cloud operators and cloud users have an increased concern of their data falling into the wrong hands. In order to keep data secured and prevent attacks by third-parties, TLS connections are often used when connecting to databases.

Regarding the implementation of DyMonD, we must consider these encrypted connections. As the goal of DyMonD is to dynamically visualize the dependencies within a distributed applica-

tion, the engine for such a generic tool must be able to predict these dependencies. DyMonD is driven by a deep learning model that takes in network traces of a service, and uses those traces to predict which service is being referenced. In our case, these resources are databases.

For unsecured networks, the network traces used in our learning model concern the application data and data source directly. However, under secured networks this data is encrypted. Inferring the corresponding resource may be difficult. The goal of this paper is to deduce the degree to which encryption affects database prediction.

Before training our model and analyzing its performance, we needed to collect encrypted network traces. This task comprised a large share of the project since the configuration of client and server encryption is highly dependent on the database.

3 Background

The local system follows a basic client-server architecture. Our client is a Java application, which connects to our server, a database. Encryption must be configured on both the client and server in order to enable a TLS connection. Once a connection is established, the client, running 50 threads, sends a combination of `READ`, `INSERT`, and `UPDATE` queries to the databases. The network packets sent between the client and the database are recorded using tcpdump, a network sniffing tool. Before integrating these packets into the model's testing and training datasets, they are processed using one-hot encoding.

3.1 Client

A total of four Java applications are used to simulate real world usage. Three of these clients were previously developed in the Distributed Information Systems Lab (DISL), corresponding to the first three clients listed below. For these clients, 50 concurrent threads are run each making 100 requests.

1. **Netflix**: Video streaming service.
   Retrieve information about a particular movie, create new payments and accounts, and update a movie's ratings.

2. **McGill**: Course Registration System
   Create courses, and update a course's information

3. **Venues**: Host and Organize Events
   Retrieve information regarding an event, create new organizations, and update a person's information

The fourth client is an open source benchmarking tool maintained by developers at Yahoo!:

4. **Yahoo! Cloud Serving Benchmark** (YCSB): Database Workload Framework
   This service uses predefined workloads to flood databases. YCSB requires that your database contains one table, `usertable` with 10 fields. Custom configs allow you to change the number of operations performed, the number of records, the number of threads, etc.

3.2 Server

Three database systems were used. All four clients connected to one instance of each of the following: **Postgres**, **DB2**, and **MySQL**. Since these databases are the server-side, i.e. the resources of our architecture, these are the components that are predicted by our deep learning model. For example, given a network trace between our Netflix client and Postgres server, our model should predict Postgres out of {Postgres, DB2, and MySQL}.

Both non-secure and secure network traces are required for the above three databases. The DyMonD dataset already contains non-secure traces for all three databases and all four clients. Some secured traces were captured as well. The following diagram shows the remaining **secure** traces to capture:

Client	Databases
YCSB	SDB2, SPSQL, SMYSQL
Netflix, Venues, McGill	SDB2

Fig. 1: *Client-Server Traces Remaining*

3.3 Encryption

As mentioned previously, enabling a TLS connection requires both client and server configuration. The client and server exchange a series of security parameters prior to a data transfer in what's known as the **Handshake protocol**. In a typical TCP interaction, as shown in Figure 2, application data begins directly after the `SYN`, `SYN-ACK`, `ACK` sequence [1].

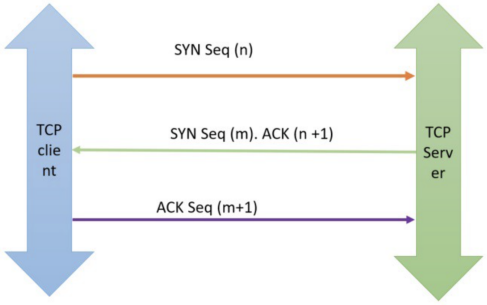

Fig. 2: *TCP Connection [2]*

The TLS handshake begins after this sequence for a secure connection, and only after its completion will application data begin to transfer.

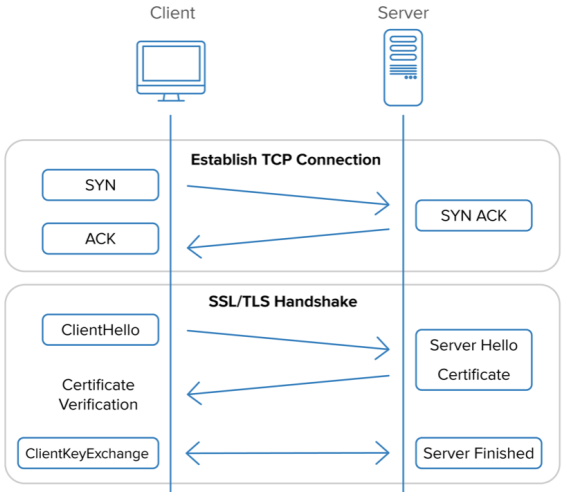

Fig. 3: *TLS Connection*

The TLS connection can be broken into the following major components [3]:

1. **Client Hello**: Client initiates contact with TLS version number, a session ID, available ciphers, and a *random value*

2. **Server Hello**: Server responds with TLS version number, a session ID, and available ciphers. If these are compatible, the connection continues to the next step. The server also responds with a *random value*, which will be used for encryption later.

3. **Certificate Verification** The server sends its certificate to the client, which contains the server's public key. The client uses this key to authenticate the server.

4. **Client Key Exchange** The server computes a *premaster secret* using both random values. This secret is encrypted by the public key from the server's certificate before being transmitted to the server. Since the server's private key matches the public key, the server will be able to decrypt the data.

5. **Server Finished** Now that the client and server have confirmed their identities, the following application data can be encrypted and decrypted. The secure connection has been formed.

4 Implementation

The above outline of the TLS handshake shows us some of the files required on both the client and server side to enable a TLS connection. While the general process is the same for all databases, implementation details vary widely between them. Implementing the encryption protocol can be thought of as a secondary step to building a client-server connection, which we will walk through below.

4.1 Connection

Clients use the JDBC API to establish connection to the databases, allowing a single application to access different databases using the same SQL statements. The general connection class is managed by the `java.sql` package. The `DriverManager` class within this package is responsible for creating the database connection and expects a database URL in the following format:

```
jdbc:<TYPE>://<HOST>:<PORT>/<NAME>
```

This URL is passed as an argument to the `DriverManager.getConnection()` function, along with the database username, database password, and SSL configs. As part of its initialization, the `DriverManager` will attempt to load available JDBC Drivers. Each database has its own JDBC driver, which must be included as an external dependency and can be loaded at run-time.

```
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <version>42.2.5.jre7</version>
</dependency>
```

Fig. 4: *External Dependency*

```
Class.forName("org.postgresql.Driver");
```

Fig. 5: *Loading the JDBC Driver*

4.2 Encryption

4.2.1 Server Encryption

On the server-side, enabling security can be separated into two tiers:

1. **Updating configuration values**: Config files that specify general database settings will contain default TLS settings that will need to be updated.

2. **Creating server certs and keys**: These files must be generated using proper credentials and protocols. They must exist in accessible directories.

MySQL defaults to TLS connections, so these steps were only required for Postgres and DB2.

DB2

DB2 is managed by IBM and runs inside of a Docker container. The IBM Global Security Kit (GSKit) installed with DB2 is used to generate keys [4]. DB2 requires server certificates to be stored in a *key database* - a specific folder for storing certificates. Using the GSKit commands, one can create a key database and server certificate. A third file will be created which stores the username and password to the key database. The server certificate must be moved to the key database as well as extracted to its own file, which will be later moved to the client. Once the certificate exists in the key database, the following configuration values must be updated [5]:

`SSL_SVR_KEYDB`: point to keydb file

`SSL_SVR_STASH`: point to the stash file containing keydb username and password

`SSL_SVR_LABEL`: the label of the key certificate

`SSL_SVCENAME`: the name of a new service in the /etc/services file that listens on a different port than the original tcp port.

`DB2COMM`: must be set to SSL

Postgres

The Postgres instance ran on a server managed by Postgres.app, an open sourced Postgres installation package with basic UI. Unlike DB2, Postgres installation does not include a custom certificate generation tool. Keytool, a Java command line tool, covers the same functionality. First, create a *keystore* with a common name equal to the host you will be accessing the database from. The keystore holds a similar function as the keydb in the previous step. A key will automatically be generated within the keystore that will be used to create a certificate in following steps. Next, create a Certificate

Signing Request (CSR), a file that contains the information (e.g. common name, organization, country) the Certificate Authority (CA) will use to create our certificate. Using both the key created in our keystore and our CSR, we create a signed certificate. The key and certificate must be stored as `server.key` and `server.crt` respectively within the server's Postgres data directory. Lastly, SSL must be set to "on" in the configuration file [6].

4.2.2 Blockers

4.2.3 Client Encryption

Since the JDBC drivers are unique to each database, client side TLS configurations varied between DB2 and Postgres. No additional configuration was required for MySQL, which handles the SSL authentication natively. For DB2 Connections, SSL is established with a URL parameter. The driver accesses proper certification by pointing to the keystore we created in the previous step in System properties, which can be seen in Figure 6.

```
String urlDB2 = "jdbc:db2://HOST:PORT/DBNAME:sslConnection=true;";
System.setProperty("javax.net.ssl.trustStore", "/PATH/TO/psql.keystore");
System.setProperty("javax.net.ssl.trustStorePassword", "KEYSTOREPASS");
```

Fig. 6: *JDBC SDB2 Connection*

For Postgres connections, the parameter `ssl` must be set to true. In addition, the client key and certificate must be passed as arguments within the database URL. The parameters `sslcert` and `sslkey` must point to the key and certificate created for the server in the previous step, as illustrated in Figure 7:

```
props.setProperty("ssl","true");
props.setProperty("sslcert","/PATH/TO/server.crt");
props.setProperty("sslkey","/PATH/TO/server.key");
Connection conn = DriverManager.getConnection(urlPSQL, props);
```

Fig. 7: *JDBC SPSQL Connection*

5 Evaluation

5.1 Dataset

The network traces were collected with tcpdump and stored as packet captures (PCAPs). Each packet contains network traces corresponding to the SQL queries made from the Java clients to our databases. For each client, we ran the following four workloads, where each number is the percent of the corresponding SQL operations performed:

READ	UPDATE	INSERT
50	50	0
95	0	5
95	5	0
100	0	0

Fig. 8: *Operations by Workload*

Each client completed these workloads across 50 concurrent threads. Each client had access to the following number of database records:

Netflix	McGill	Venues	YCSB
50	100	500	100-5000

Fig. 9: *Database Records by Client*

We also note the variety in TLS versions across both databases and clients. Newer versions of TLS introduce more secure options for clients and new acceptable cipher-suites (encryption methods), however both the client and server must be compatible with the same TLS version.

	YCSB	Netflix, McGill, Venues
DB2	1.1	1.1
Postgres	1.3	1.2
MySQL	1.3	1.2

Fig. 10: *TLS Versions by Client*

The three figures above show features of our system that produce more realistic data. (1) Using different TLS versions across clients and databases, (2) changing the number of database records between clients, and (3) running four workloads on each client, introduced variability into our data that corresponds more realistically to complex cloud application network traffic.

5.2 Deep Learning Model

The deep learning model used is a combination of a Convolutional Neural Network (CNN) and a Recurrent Neural Network (RNN). The model outputs a service type by detecting the service-specific patterns within the input packet. Since CNNs are commonly used to classify images, we conceptualize a packet data as an array of pixels, where each pixel represents a byte of data. A kernel action extracts location invariant patterns from these images in order to produce feature maps. In the combined CNN-RNN model, these feature maps are first extracted through the convolutional layer. The output of the CNN is then used as refined sequential data input to the RNN [7]. Optimizing an RNN with Long Short Term Memory (LSTM) has been shown to achieve better classification for different service types since LSTMs aid in training of time-series data. The final model included two LSTM layers (BiLSTM). The first layer is applied on the input sequence (forward layer), while the reverse form of the input sequence is fed into the second LSTM layer (backward layer). BiLSTMs improve the model prediction by including a secondary layer of learning dependencies in the sequential input [8].

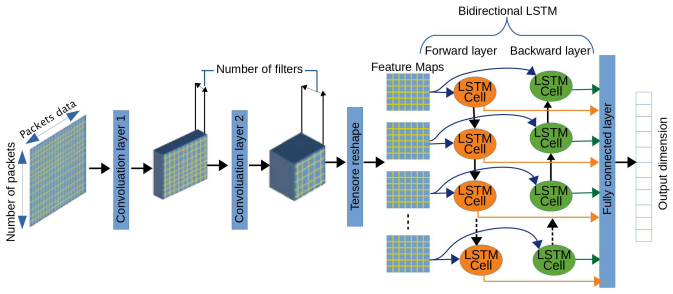

Fig. 11: *CNN + Bi-LSTM[7]*

5.2.1 Model Tuning

Finding optimal hyper-parameters on such a complex model is a computation-heavy and time-consuming task that may benefit from optimization. The method used for tuning the hyper-parameters is the Sequential Model-Based Optimization (SMBO). This Bayesian optimization technique stores the results from previous trials to prune the remaining set of hyper-parameters to tests for later iterations. Other algorithms that employ brute-force strategies are poor candidates for finding optimal hyper-parameters efficiently. The resulting hyper-parameters were then validated using 3-fold cross-validation.

The following parameters were tested on both datasets. The underlined values correspond to the selected optimal value. When the value differs between non-secured and secured datasets, the secured dataset is indicated by red and the non-secured dataset is indicated by green.

Hyperparameter	Values
Kernel Init	normal, uniform, glorot uniform
Kernel Size 1	3, 5, 7
Kernel Size 2	3, 5, 7
Filter	32, 64
BiLSTM layer units	64, 128, 256
Output size	64, 128, 256
Regularizer	0.0,0.03,0.01
Activation	relu, softmax,tanh
Optimization typel	adam, rmsprop
Batch Size	32, 64, 100
Epochs	50, 100, 200
BiLSTM Dropout Rate	0.0, 0.2, 0.3, 0.4
Output layer Dropout Rate	0.0, 0.2, 0.3,0.4

Fig. 12: *Chosen Hyperparameters, Red SSL, Green NoSSL*

5.2.2 Learning Environment

The model training was performed in a Google Co-laboratory Research Environment (Colab). Colab is a Python interpreter hosted in the cloud with access to all relevant Python libraries and computing resources for free. We configured the notebooks to use Tensorflow with GPU in order to efficiently run the models.

5.3 Results

	DB2	MySQL	PSQL
DB2	189	0	11
MySQL	0	200	0
PSQL	0	0	225

Fig. 13: *Non-SSL Confusion Matrix*

	DB2	MySQL	PSQL	SDB2	SMySQL	SPSQL	Total
DB2	200	0	0	0	0	0	200
MySQL	0	200	0	0	0	0	200
PSQL	0	0	25	0	0	200	225
SDB2	0	0	0	300	0	0	300
SMySQL	0	0	0	0	300	0	300
SPSQL	0	0	46	0	0	254	300

Fig. 14: *SSL Confusion Matrix*

In this section, we evaluate the performance of our model across two datasets. First, we ran the model on the non-secured traces. Next we concatenated the non-secured traces with the secured traces. The relevant performance metrics corresponding to each model are compared to measure the effect of the secured traces. The main performance metrics used were accuracy and F1-Score.

Accuracy and F1 are concerned with the frequency of the following types of classifications: true positives, true negatives, false positives, and false negatives, or TP, TN, FP, and FN respectively. Each trace is classified as one of these four types, so the total number of classifications is (TP + TN + FP + FN). For each trace of service type s, a true positive of that trace is the correct classification of that service as type s during testing. True negatives are all services $s' \neq s$ that are correctly not classified as s. False positives are then all flows of type $s' \neq s$ that are incorrectly classified as s, and false negatives are all flows of type s that are incorrectly labeled as one of $s' \neq s$.

Accuracy measures the total number of true positives and true negatives out of all classifications, a ratio which will tend towards 1 as the model gets better at classification. The F1 Score encompasses two ratios associated with false positives and false negatives. Precision, which accounts for false positives, tells us how many positive identifications were actually correct. Recall, which accounts for false negatives, tells us how actual positives might have been missed. F1 gives more holistic information about the model.

We found the non-secured decreased model accuracy by 16% and decreased the F1 score by 18%. We present the respective F1 scores and accuracy below:

	NoSSL	SSL
F1 Score	0.99	0.82
Accuracy	0.99	0.83
Precision	0.99	0.84
Recall	0.99	0.83

Fig. 15: *Performance by Dataset*

The confusion matrices are presented for the non-secured and secured datasets above, in Figures 13 and 14. The rows correspond to the true values for each trace, and the columns correspond to the predicted values. All numbers on the diagonal are correctly predicted values (true positives). Any value off a diagonal has been incorrectly classified as the column value with a true value of the row value. This can be thought of as the false positives for the column value, and the false negatives for the row value.

Overall, our model performed well in classifying both non-SSL and SSL services. For the non-SSL packets, the only source of error came from the mis-classification of 11 DB2 packets as PSQL, which only accounts for 6% of the total DB2 traces. Neither MySQL or PSQL had any false negatives or false positives. We note that the row sums for each service correspond to the total number of packets encoded with that service. The classes are evenly distributed, with the secured services slightly over represented with an average of 208 entries for non-secured packets and 300 entries for secured packets.

For both non-SSL and SSL services, the only source of error came from one relatively low-performing service. For the SSL packets, the

only mis-classifications occurred between secured and non-secured Postgres services. By the PSQL row in the confusion matrix, we can see that more than 88% of the PSQL traces were classified as secured PSQL traces. On the other hand, 15% of secured PSQL traces were classified as non-secured PSQL. However, we notice that these errors are contained to the Postgres services. All 4 other services achieved an accuracy of 100%.

5.4 Conclusion and Future Work

The work covered in this paper concerns the ability of the DyMonD model to dynamically predict services used by cloud components. On large cloud applications, most connections between components will be encrypted. Our results showed a decrease in accuracy of 16% from 99% to 83% when including encrypted traces. However we saw that this reduction in accuracy was confined to the mis-classification of secured and non-secured traces for the *same* service. In the larger context of DyMonD, both the secured and non-secured services will likely be encoded as the same value. For example, one component may access the Postgres database with a TCP connection, while another component may access the database with a TLS connection, however the service itself is the same for both clients.

The most relevant direction for future work on the topic of encrypted trace collection would require further reference to the operators of large distributed cloud systems. Using new methods of encryption may introduce new variability to the dataset (e.g. encrypting database entries from the client while connecting over TCP). However, unless alternative methods are a standard practice among real cloud applications, replicating them locally will not benefit the model. Integrating more databases, specifically those hosted on the cloud (e.g. AWS, Google Cloud) would test the model's ability to recognize new services. In order to introduce more variability into the data, future work could involve building clients in other languages that are managed by cloud servers (e.g. a Javascript endpoint hosted by Heroku). Both directions would integrate realistic cloud application network traces that would supplement the initial findings of this paper.

References

[1] *SSL/TLS in Detail*. Microsoft TechNet. October, 2009
https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc785811(v=ws.10)?redirectedfrom=MSDN

[2] *What is tcp three way handshake? What is SYN, ACK packets?*. Medium, CSPS Protocols. February 1, 2019
https://medium.com/@cspsprotocols247/what-is-tcp-three-way-handshake-what-is-syn-ack-packets-dc1b122a0c06

[3] *Establishing a SSL/TLS Session*. Okta Developer. https://developer.okta.com/books/api-security/tls/how/

[4] *IBM Global Security Kit (GSkit)*. IBM Informix 12.10. IBM Corporation 2019.
https://www.ibm.com/docs/en/informix-servers/12.10/12.10?topic=protocol-global-security-kit-gskit

[5] *Creating a key database file with a self-signed certificate*. IBM Security Directory Server Version 6.4.0.
https://www.ibm.com/docs/en/sdse/6.4.0?topic=database-creating-key-file-self-signed-certificate

[6] *Postgres 17.9: Secure TCP/IP Connections with SSL*. PostgreSQL 9.5.25 Documentation, Chapter 17. Server Setup and Operation.
https://www.postgresql.org/docs/9.5/ssl-tcp.html

[7] Mona Elsaadawy. *DynAppDis: Dynamic Application Topology Discovery and Service Detection Framework for Cloud Data Centers*
https://www.cs.mcgill.ca/ kemme/disl/index.html

[8] M. Lopez-Martin, B. Carro, A. Sanchez-Esguevillas and J. Lloret, *Network Traffic Classifier With Convolutional and Recurrent Neural Networks for Internet of Things*, in IEEE Access, vol. 5, pp. 18042-18050, 2017, doi: 10.1109/ACCESS.2017.2747560.
https://ieeexplore.ieee.org/document/8026581