A Prototype Implementation of an Anomaly Detection System for NoSQL Databases

Ron Wright (wright53)
University of Illinois
at Urbana-Champaign

Eric Badger (badger1)
University of Illinois
at Urbana-Champaign

Jian Tian (jtian4)
University of Illinois
at Urbana-Champaign

***Abstract*—We have designed a prototype plugin for implementing a system that detects anomalies in NoSQL systems, specifically MongoDB and Redis. With our prototype implementation, we have designed an anomaly detection system and a machine learning algorithm system to classify anomalous workloads in real time. We have created nine unique YCSB workloads to evaluate the performance of our prototype. The results showed that our framework with kNN algorithm worked perfectly on both MongoDB and Redis systems with high detection accuracy. Currently, the plugin works on a single system, but the plan is to evolve the implementation into a distributed intrusion detection system (IDS) that works in the cloud. We plan to sell our idea to enterprises who use or want to use NoSQL databases and are concerned with data security.**

I. Introduction

It is known that system downtime has cost companies a total of $26.5 billion in lost revenue in 2011. [1] A large proportion of companies are often unprepared for outages that can occur frequently and for an extended period of time. Companies must cope with the fact that downtimes can be regular with causes unbeknownst to them. For example, according to VeriSign, distributed denial-of-service (DDoS) attacks are often overlooked as a possible cause of downtime. [2] This is especially a problem in NoSQL systems such as MongoDB, where possible misuse of data can lead to executions of excessively large numbers of queries.

In the case of cloud computing, service level objectives (SLOs) offer constraints on resource usage and can help in determining anomalous system behavior. However, the administrator may not have the resource or the motivation to tap into the cloud, which means he or she has one less method for detecting abnormal workloads.

We propose an anomaly detection scheme to overcome the problem of using data analytics for anomaly detection in NoSQL databases. We assess the accuracy and performance of two NoSQL databases, namely MongoDB [3] and Redis [4]. MongoDB is an open-source NoSQL database that utilizes JSON and BSON objects to represent information. The MongoDB suite supports sharding, which allows the database to scale out. Redis is an open-source NoSQL database that utilizes keys and values to represent information. Similar to MongoDB, Redis supports partitioning of data (i.e., sharding) for scaling out the database. We created a simple anomaly detector for MongoDB and Redis that works directly with the databases themselves for gathering application metrics such as number of insertions, deletions, and updates, as well as system metrics such as CPU and memory utilization.

Other anomaly detection tools are general purpose and cannot differentiate between small number of queries and large number of queries. Our tool, on the other hand, takes this into consideration for more accurate anomaly detection. We used two different algorithms for anomaly detection: k-nearest-neighbor (kNN) and support vector machine (SVM).

Our paper is organized as follows. Section II describes related work assessing the applications and techniques of pre-existing anomaly detection systems. Sections III and IV describe our goals and techniques for our project, as well as the advantages and disadvantages of our techniques. Section V lists and explains the algorithms used for our prototype implementation, and Section VI outlines its implementation. Section VII details our experiments and evaluation results, Section VIII describes our entrepreneurial business plan explaining how and why

our system will be a viable product for enterprises concerned with NoSQL security, and Section IX concludes the paper.

II. Related Work

A. Distributed System Applications

The authors of [5] introduce a distributed intrusion detection system (IDS), a type of anomaly detection system, that uses a topological interest-based model and does not require the presence of analysis hierarchy for data analysis. Interests are specifications of data that an agent (or host) is interested in, but may not necessarily be available to the agent because of locality restrictions or the agent is not intended to observe the data. One disadvantage of this approach is that it in itself is not that secure, since it is vulnerable to interest spoofing and distributed denial-of-service (DDoS) attacks.

The authors of [6], [7], and [8] describe distributed anomaly-based IDSes. The first paper introduces a system for mobile ad-hoc networks (MANETs) that uses the C4.5 learning algorithm and involves a tree hierarchy for gathering differences in normal operation and actual operation, or anomaly indices, from nodes to cluster heads representing node clusters to a central manager node. One disadvantage of this approach is that the manager node (and the cluster heads) can become a centralized bottleneck and a single point of failure.

Next, in the second system, the authors of [7] describe a distributed anomaly detection model similar to what is described in [6] that uses supervised support vector machine (SVM) learning and involves multiple machines finding local outliers and sending them to a master site that finds global outliers within the local outliers. One disadvantage of this approach is that the master site consists of only one host that can become a centralized bottleneck in rare cases and a single point of failure.

Finally, the authors of [8] describe an anomaly-based IDS for MySQL, which parses each incoming query and uses string length, string character distribution, string prefix and suffix matching, and string structure inference and learning to assign anomaly scores to each query. Such an approach can be extended to NoSQL databases and distributed systems using pre-existing approaches for anomaly detection.

The blog in [9] details how to implement logging for MySQL to detect errors. The papers [10] and [11] depict signature-based attack detection systems for web applications. The first is for PHP-based web applications, while the second is a paper from UIUC that depicts a new mechanism called CANDID.

B. Anomaly Detection Techniques in Distributed Systems

The authors of [12] proposed an anomaly detection scheme consisting of rich statistical performance models for predicting system performance, a control policy simulator that runs simulations based on the performance model, and model management techniques to correct the models appropriately via machine learning. This is a preliminary approach, as the authors present only the percentage of slow workloads as a metric for anomaly detection. Our approach, on the other hand, measures system and application metrics.

The authors of [13] proposed an anomaly detection scheme based on the Hidden semi-Markov Model (HsMM) for detecting malicious anomalies in HTTP requests at the application layer. The sequence of HTTP requests made by the user is converted into a sequence of HsMM states, and then the forward-backward algorithm is used to correct the behavior model with the observed behavior characteristics. Finally, the inferred model is used to judge whether any future user behavior is normal or anomalous. Our approach, on the other hand, utilizes machine learning algorithms on statistical data.

The authors of [14] introduced an entropy-based anomaly testing tool for large exascale systems known as EbAT, which analyzes metric distributions rather than metric thresholds. It encompasses wavelet analysis and visual spike detection to detect anomalies. Metric categories include OS metrics, application metrics, and platform metrics. Examples of OS metrics include CPU and memory usage. This approach is based on signature detection and human judgment, but our approach uses machine learning algorithms with simple, clear-cut differentiation of normal and anomalous workload characteristics.

The authors of [15] proposed an anomaly detection scheme that uses Tukey and Relative Entropy

Statistics for detecting anomalies in workload behavior at different levels of abstractions in large data centers. This set of approaches, namely point thresholds and windowing, employs system metrics such as CPU and memory usage. In point thresholds, a set of observations are placed into quartile intervals and an observation is deemed anomalous if it ever falls out of its quartile interval. In windowing approaches, hypothesis testing and the multinomial goodness-of-fit test are used to detect anomalies in observations. Our approach is similar, except kNN and SVM classifiers are used instead of more advanced statistical techniques.

III. Goals

One commonality seen in the analysis presented in [16] is that neither MongoDB nor Cassandra provides adequate auditing. This presents an opportunity for major improvement. To improve the auditing and alleviate issues related to both attacks and anomalous behavior, it is our goal to implement an accurate and efficient anomaly detection for NoSQL databases (though it could be extended to other databases). To make our product as effective as possible, we have implemented it in a modular fashion such that it is easily extendable to other databases.

To meet our goal of being accurate and efficient, we have implemented and compared two different machine learning techniques: k-nearest neighbor and support vector machine. In addition, we use metrics that are collected by the databases themselves, so as to get rid of the need for any outisde monitoring tools.

A future goal, that is outside of the scope of this prototype, is to turn our anomaly-based IDS into an Intrusion Protection System (IPS), so that it can not only detect attacks and anomalies, but can actually dynamically react to these anomalous behaviors.

IV. Techniques

In our system, we have implemented an anomaly-based detection scheme that leverages machine learning as its heuristic. For our anomaly-based detection scheme, we are using the k-nearest neighbor algorithm and support vector machines to detect anomalies in the database. We will describe these two in detail in Section V. To get the metrics to do the detection on, we rely on the databases themselves. Querying the metrics from the database adds a non-zero amount of strain on the database itself, but we believe that it is negligible and can be ignored, since we are only making at most 2 additional queries/second.

Originally, we were looking to do just attack detection on single queries themselves that exploited vulnerabilities of the database. We created a Naïve Bayes machine learning implementation for anomaly detection at the resolution of actual queries. However, after quite some time, we realized that there was not enough readily available data to train and test the system on. We do not have the expertise nor the manpower to craft enough malicious and benign queries to be able to accurately train a machine learning-based system. Because of this, we turned to signature-based detection on the queries. With MongoDB as our target, we found out that signature-based detection was a futile task. This is because the vulnerabilities that we found in MongoDB all had to do with manipulating the BSON object that was sent to MongoDB. The problem with this is that it means that the malicious query will not be logged exactly as it was passed to MongoDB, and therefore, we can not use it to train. We firmly believe that our initial anomaly-based scheme to detect malicious queries works and is extendable, but we do not have the means to correctly train or test it.

In these regards, we switched our focus from the queries to the databases. Instead of detecting malicious queries by examining their contents, we monitored the system statistics of the databases to see if there were unusual changes on the workloads. In this way, we are able to avoid the issue with the lack of training data, since we can configure the system to generate training sample for both normal and abnormal workflows. Then, we can apply machine learning techniques based on the training samples to figure out the scenarios when anomalous workflows occur in the system.

A. Anomaly-based Detection vs. Signature-based Detection

Anomaly-based detection works by detecting attacks and anomalous behavior based on rules or heuristics. In our case, the heuristic would be

our machine learning algorithms. On the other side, signature-based detection recognizes attacks by matching a known signature or patterns of events. Anomaly-based detection schemes have the ability to detect nefarious behavior in near real-time and the ability to detect attacks that the system has not seen before. Assuming that the system can handle the rate of events coming into the system, the anomaly-based detection scheme will be able to detect attacks in real-time. However, there will be a rate at which the system cannot handle the incoming events, because the computation time of the detection will be greater than the time between queries. This means that in anomaly-based detection schemes, it is important to minimize the computation time of the algorithm. Another option is to batch the heuristics that the anomaly-based detection scheme works on. That way, the computation time won't scale at the same rate as the rate of events. Signature-based detection schemes will also run into a point where they cannot work in real-time, but they will be able to serve a much higher rate of events in real-time, because the computation time of the signature-based detection is usually much lower.

Signature-based detection schemes work by detecting events that it has already flagged as malicious via signatures or patterns. Therefore, the signature-based approach will only be able to detect attacks that it already knows and will not be able to extrapolate to new malicious actions or ones that are slightly different than the signature that it has seen before. This renders signature-based detection schemes completely useless for zero-day attacks, which are attacks that have never been used before and thus have never been seen before. Anomaly-based detection schemes, however, allow the system to detect attacks that it has not explicitly seen before. By using a set of rules or heuristics, the anomaly-based detection can detect strange behavior in the system even if the system has not explicitly seen that behavior before. For example, if there were two attacks that compromised the same vulnerability of a system (such as a dropall tables attack), but did so in a slightly different way, the signature-based detection would not be able to detect the attack, while the anomaly-based detection would have a fighting chance of detecting anomalous behavior in the system.

Intrusion detection systems are measured mostly on their detection ability. It is possible for them to label normal events as anomalous or label an attack as a normal set of events. These two phenomena are known respectively as false positives and false negatives. Signature-based detection schemes will not run into the false positive problem because it knows exactly what the signature of the attack looks like and will only classify events that match as attacks. Because of the strict matching classifications, though, signature-based detection schemes can also have a very high percentage of false negatives, where they are not able to correctly classify attacks as attacks. Anomaly-based detection schemes are more aggressive. Since they work on heuristics or rules, they are able to classify more sets of events as attacks. Depending on the threshold of the anomaly-based scheme, there will usually be either a high percentage of false positives or of false negatives. Setting the threshold too low for classifying a set of events as an attack will lead to a lot of false negatives, while setting it too high will lead to a lot of false positives. Neither of these is ideal and so tuning the anomaly-based detection scheme is very important.

The last major difference between signature-based and anomaly-based detection schemes is the complexity. Building a signature-based detection scheme will be fairly easy because it only involves pattern-matching sets of events against known attacks. As stated above, this will also lead to shorter computation time. However, it requires constantly updating the repository of known attacks against which to match the event sets, which is an incurred cost over the lifetime of the system. Anomaly-based detection schemes, on the other hand, are much more complex. They run algorithms that are difficult to understand and to implement and increase the size and complexity of the code. In most cases, this means that there will be an increase in the number of bugs in the system. Unlike signature-based detection schemes, though, there will be little upkeep with the anomaly-based detection. There will be an upfront cost of tuning the system to a reasonable number of false positives and false negatives, but after that the system can, for the most part, run without any intervention.

V. ALGORITHMS

In current anomaly-based IDS applications, machine learning technique has been an indispensable tool to detect intrusions through large amount of intrusion detection data [17]. In our implementation to detect anomalous workflows, we have applied different types of classification methods to accurately capture the unusual pattern of query statistics and hardware usage during the workflows. In the following we will introduce two algorithms that we have implemented in our IDS: k-nearest neighbor (kNN) and support vector machine (SVM).

A. k-nearest neighbor

kNN is a simple, non-parametric, and lazy-learning classification method in pattern recognition. Basically, given a set of training data with labels, an object is classified by a majority vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors (where k is a positive integer). The kNN algorithm does not require a training phase. Instead, during the testing phase, it requires the calculation of distances from the querying object to all the other objects in the training set.

In our workflow detection application, the kNN algorithm was performed on a series of feature vectors, where the features included query-specific statistics such as number of inserting, updating, querying, etc., and hardware and network statistics such as CPU and memory usage, number of bytes in and out of the network, and so on. In addition, we obtained the workflow statistics at each second and formed a per-second feature vector. Each vector expanded to a multi-dimensional space and each dimension contained distinct range of values.

Notice that a drawback of the kNN algorithm is that it is sensitive to outliers. Therefore, before the testing phase begins we need to smooth our training data to eliminate the noise that could potentially affect the classification result. In this regard, we performed a moving average smoothing (box smoothing) to the training data with replacing each data value with the average of neighboring values. To avoid shifting the data, it is best to average the same number of values before and after where the average is being calculated. The smoothing equation for a given vector $\mathbf{x}$ is shown in Eq. 1:

$$\overline{\mathbf{x}}_i = \frac{1}{2M+1} \sum_{j=-M}^{M} \mathbf{x}[i+j] \tag{1}$$

Furthermore, our object vectors have multi-dimensions and each of the dimensions has different range of values. For example, the dimension "query" may have numerical values from a few hundreds to over ten thousand, while the dimension "insert" for a normal workflow is significantly lower than "query". Therefore, we must perform a feature scaling to standardize the range of independent variables or features of data. For the simplicity of future distance calculation, we applied the method of scaling to unit length, i.e., scaling the components of a feature vector such that the complete vector has length one. This is done by dividing each component by the Euclidean length of the vector (Eq. 2):

$$\mathbf{x}' = \frac{\mathbf{x}}{||\mathbf{x}||} \tag{2}$$

After this pre-processing of the data, we can apply the kNN algorithm to the testing object by calculating the Euclidean distances between the object to all the other vector objects in the training set. The selection of k value depends on the data, so we performed sensitivity studies by varying k values and chose the one that produced the optimal result. Notice that it is still a naive approach to decide the k value; more complicated heuristic techniques have been reported for dealing with this issue (such as hyperparameter optimization). However, we have seen that for our case setting a k value by iterative testing should be good enough to obtain accurate output.

We should also mention that although kNN doesn't require a training phase, the testing phase is really expensive due to the distance calculation for each pair of objects. Suppose we have n training samples, each of them is a m-length vector, then for each testing object the computational time is $O(nm)$. Furthermore, if we have n' testing samples, the total cost for the anomaly detection will be $O(nmn')$, which can be really high for very high-dimensional vectors. However, in our case since each vector only contains fixed number of dimensions ranging from 20 to 30, kNN can still be

performed quite fast for the classification of testing objects.

B. Support vector machine

The SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. Unlike kNN, which is a non-linear classifier, SVM should be aware of the separability of the data so that it can apply different kernel functions for the classification. For linearly separable data, we can select two hyperplanes in a way that they separate the data and there are no points between them, and then try to maximize their distance. However, before implementation of the SVM model, we should first determine if our workflow data are linearly separable so that we can choose proper model representation.

We started with Perceptron, which is a linear binary classifier that makes its predictions based on a linear predictor function combining a set of weights with the feature vector. It first assigns a weight vector for each of the classes ('normal" and "anomaly" in our case), and then iteratively updates the weight vectors by looping over the training set when the predicted label is different from the true label until convergence. However, after testing with our workflow scenarios we have found that the two classes can not be separated by a hyperplane, which means our data is not linearly separable.

For non-linear data, SVM can apply a kernel trick to maximum-margin hyperplanes, i.e., map the original non-linear data to a space so that they become linearly separable. There are some popular kernel functions widely used for the transformation, and in our study, we applied the most popular Gaussian radial basis function (RBF), for which the corresponding feature space is a Hilbert space of infinite dimensions. The radial basis function is shown in Eq. 3:

$$k(x_i, x_j) = \exp\left(-\frac{||x_i - x_j||^2}{2\sigma^2}\right) \qquad (3)$$

Compared to kNN, the most of the computational cost of SVM is in the training phase during which the model needs to learn and build a classifier based on the training samples. However, the testing phase is really fast since it just needs to apply a trained classified to the testing object to obtain the predicted class label.

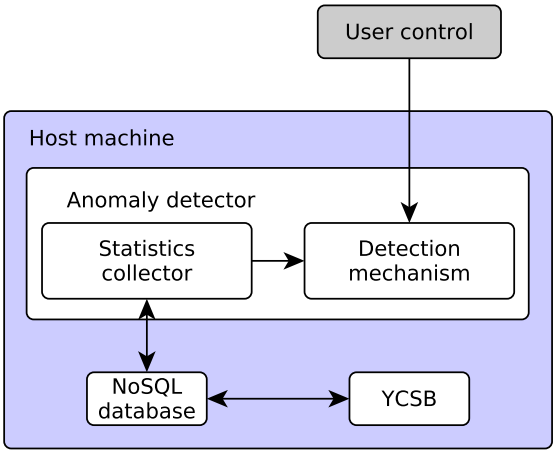

Fig. 1. Architecture of our prototype implementation.

VI. Implementation

A. MongoDB

To evaluate our prototype tool, we used a single machine running MongoDB 2.2.3 and several Python scripts that records the workload statistics from Yahoo! Cloud Serving Benchmark (YCSB) [18] as well as the detection methods (both kNN and SVM algorithms). YCSB is a popular Java open-source specification and program suite developed at Yahoo! to compare relative performance of various NoSQL databases. Basically the goal of the YCSB project is to develop a framework and common set of workloads for evaluating the performance of different "key-value" and "cloud" serving stores. Its workloads are used in various comparative studies of NoSQL databases. Therefore, we can create a set of workloads to examine the system aspects. Figure 1 shows the setup and architecture of our prototype tool.

For the evaluation, we execute different workload scenarios from a mongo client ("mongo"), which are executed on the mongo daemon ("mongod"). All of the scenario statistics are collected by the IDS via the MongoDB system server status output. The complete list of parameters from the server status output can be found in [19]. The IDS consists of a proactive server status collector and a detection

component written in Python. The proactive server status collects the server statistics every second, and converts the new queries into a format recognizable by the classification framework as discussed in section V, which classifies the per-second workflow as being normal or anomalous. In addition, to make the IDS work with MongoDB, we used a Python plugin known as PyMongo [20].

B. Redis

Redis is an open source, BSD licensed, advanced key-value cache and store. We applied the same YCSB benchmark tool on Redis database to evaluate different workloads. For the implementation of obtaining system status from Redis, we used the Python interface to the Redis key-value store (redis 2.10.3). Then we can get information and statistics about the server in a format that is simple to parse by computers and easy to read by humans using the INFO command as explained in [21]. Therefore, we can obtain similar metrics of system status as for MongoDB so that we have two NoSQL databases to test on.

In our implementation, the entire detection procedure is divided into three phases: a normal training phase, an anomalous training phase and a testing phase. During the two training phases, we select the database to work on and the classification method used for detection, and then specify the time period (in second) we would like to read in the per-second training data. Then during the testing phase, the training samples will first be pro-processed for smoothing and scaling, and our detection mechanism will perform classification on each of the per-second testing record and obtain the predicted class as either normal or anomalous.

The source code for this study is located at: https://bitbucket.org/ebadger1/cs525.

VII. Evaluation

A. Experimental Setup

All experiments were run on a Dell XPS 8700 desktop with an Intel® Core™ i7-4790 CPU @ 3.60GHz and 8GB of RAM. The desktop is running Ubuntu 12.04.

B. YCSB Workloads

Our training and testing sets were created using nine unique YCSB workloads. Each of these workloads was different and could be used to model the workload of a certain application or a certain time of day. For example, if workloads are significantly different at 6pm than they are at 6am, then the result should be two different workloads to characterize them. Our system trains under the assumption that the normal workloads of a system are fairly consistent. There can be many of these normal workloads that would characterize each as normal. In addition, there can be multiple anomalous workloads that the IDS trains on. To make the IDS effective, one needs to train it correctly so that it can detect what is correct and what is incorrect. Since there is no dependency here, the sysadmin can decide what it deems as normal. So, if the sysadmin would like the system to detect a DDoS attack, then it can train the system to understand that a large number of users is anomalous. Furthermore, a common breach of a database would lead to a dropall tables or a table dump. If one trains the system on such workflows, it will be able to detect the rapid decrease in records or the large amount of network output.

The nine workloads that we have chosen characterize a wide variety of workload types. The YCSB benchmarking tool allows experimenters to choose read, update, scan, read-modify-write, and insert proportions for the workloads. We use all of these except for insert. In our evaluation we figured out that insert does not work correctly in our case since it always tries to write the same records. This means that we can insert for training, but once we insert as a test it will error out. Because of this, we did not include insert measurements in our evaluation, but we believe that our results can be generalized to include this additional operation. The rest of the proportions are defined as such:

- **Insert**: Insert a new record.
- **Update**: Update a record by replacing the value of one field.
- **Read**: Read a record, either one randomly chosen field or all fields.
- **Scan**: Scan records in order, starting at a randomly chosen record key. The number of records to scan is randomly chosen.[22]

The proportions of all of the workloads are defined in Table I.

C. Metrics

MongoDB and Redis both log metrics of their performance, which we used for our detection. However, both of the databases log different metrics in different ways.

For MongoDB, we used the ServerStatus command, which gave us a plethora of information. All of the MongoDB metrics and their details are available at [23]. The metrics that we used in our tool are:

- **Operation Counters**: query, insert, update, delete, getmore, command
- **Connections**: current
- **Network**: bytesIn, bytesOut, numRequests
- **Objects**: objects

For Redis, we used the INFO command. Like the ServerStatus comamand, this gave us access to a lot of important Redis metrics. All of the Redis metrics and their details are available at [24]. The metrics that we used in our tool are:

- **Network**: total_net_input_bytes, total_net_output_bytes
- **Commands**: total_commands_processed
- **Clients**: connected_clients
- **CPU**: used_cpu_sys, used_cpu_user
- **Keys**: db0keys, keyspace_hits, keyspace_misses, SortedSetBasedCmds, KeyBasedCmds, HashBasedCmds, SetTypeCmds, GetTypeCmds

D. Training

Before any workloads were run, the MongoDB and Redis databases were filled with 1,000,000 records. After the database was filled, each of the seven experiments were run. In each experiment, the anomaly detection tool was trained for 60 seconds on each of the normal workloads and for 20 seconds on each of the anomalous workloads. Because there are more anomalous workloads than normal workloads, we decided to train on the normal workloads for longer to balance out the number of normal and anomalous points. This was repeated for both MongoDB and Redis.

E. Testing

After the system was successfully trained on all of the normal and anomalous workloads, it was tested on the same workloads to see if it was able to label each of them correctly even in the face of statistical noise and other issues affecting latency and responsiveness. Each of the workloads was tested for 10 seconds and was given a pass or fail value. A pass value was given if the workload was correctly identified 100% over the 10 second period and a fail value was given otherwise.

We have evaluated the testing performance using both kNN and SVM algorithms. We found that while kNN worked well, SVM failed to separate the normal and anomalous workloads. The possible reason is that kNN is a typical non-linear, non-parametric classifier, which means it does not require an explicit model to fit the structure of the sampling data. On the contrary, non-linear SVM needs a kernel function which maps the original data to the space that they are linearly separable. Although we used Gaussian radial basis function (as well as other kernel functions) in our SVM implementation, it is not guaranteed that these functions will work for our complicated and high-dimensional workload objects. Therefore, we will only use kNN to illustrate and present the performance of our IDS. Table II and Table III show the detection accuracy for each of the experiments on for MongoDB and Redis respectively, using the kNN framework.

F. Results

The results of the experiments are shown in Table II and Table III. Using these tables, it is clear that the anomaly tool performed fairly well across the board for classifying the different workloads as normal or anomalous. MongoDB and Redis acted very similarly, even though the metrics available for each database different.

In experiments 1 and 2, the tool was 100% accurate for both MongoDB and Redis. With only one normal workload for both of these experiments, it was very easy to classify that workload as normal and everything else around it as anomalous. Experiments 3, 4, and 7 also finished with 100% accuracy on both databases.

Experiments 5, 6, 8, and 9, however, had errors of some sort. Upon further review, we were able to de-

Workload	Proportions			
	Read	Update	Scan	Read-Modify-Write
A	0.5	0.5	0	0
B	0.95	0.05	0	0
C	0	0	0	0
F	0.85	0.1	0.01	0.04
G	0.5	0.2	0.2	0.1
H	0	0.05	0	0.95
I	0.4	0.2	0.05	0.2
J	0.25	0.25	0.25	0.25
K	0.999	0	0.001	0

TABLE I
WORKLOAD PROPORTIONS USED IN EACH WORKLOAD

	Training Workloads		Testing Workloads								
Experiment #	Normal	Anomaly	A	B	C	F	G	H	I	J	K
1	A	B, C, F, G, H, I, J, K	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass
2	B	A, C, F, G, H, I, J, K	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass
3	A, B	C, F, G, H, I, J, K	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass
4	C, F	A, B, G, H, I, J, K	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass
5	G, H	A, B, C, F, I, J, K	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Fail	Pass
6	I, J	A, B, C, F, G, H, K	Pass	Pass	Pass	Pass	Fail	Pass	Pass	Fail	Pass
7	A, B, C	F, G, H, I, J, K	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass
8	F, G, H	A, B, C, I, J, K	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Fail	Pass
9	I, J, K	A, B, C, F, G, H	Pass	Pass	Pass	Pass	Fail	Pass	Pass	Fail	Pass

TABLE II
MONGODB EXPERIMENTAL RESULTS

	Training Workloads		Testing Workloads								
Experiment #	Normal	Anomaly	A	B	C	F	G	H	I	J	K
1	A	B, C, F, G, H, I, J, K	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass
2	B	A, C, F, G, H, I, J, K	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass
3	A, B	C, F, G, H, I, J, K	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass
4	C, F	A, B, G, H, I, J, K	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass
5	G, H	A, B, C, F, I, J, K	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Fail	Pass
6	I, J	A, B, C, F, G, H, K	Pass	Pass	Pass	Pass	Fail	Pass	Pass	Fail	Pass
7	A, B, C	F, G, H, I, J, K	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Pass
8	F, G, H	A, B, C, I, J, K	Pass	Pass	Pass	Pass	Pass	Pass	Pass	Fail	Pass
9	I, J, K	A, B, C, F, G, H	Pass	Pass	Pass	Pass	Fail	Pass	Pass	Pass	Pass

TABLE III
REDIS EXPERIMENTAL RESULTS

termine that the reason that our tool was incorrectly classifying the workloads was because workload G and workload J were surprisingly similar. Taking a closer look, we can see that in Table 2 and Table 3 workload G and workload J are almost identical. The rest of the metrics tell the same story for MongoDB as well as Redis. The reason behind this is the scan command in YCSB. It takes a very long time to finish compared to all of the other commands because it scans a random number of records, which would average to be very large since the number of keys in each database is 1,000,000. Therefore, since both J and G had a relatively high scan percentage (0.2 and 0.25 respectively), they were dominated by scans. Thus, the anomaly tool keeps waffling back and forth between normal and anomalous since it is right on the boundary between the two workloads. With additional metrics, this error can be weeded out, but we do not have access to those metrics at this time.

The graphs for the rest of the metrics are available in our previously-mentioned BitBucket, but are left out of the paper for the sake of brevity.

VIII. Business Plan

While our prototype implementation of the anomaly detector for MongoDB and Redis is written in the form of Python scripts that communicate with the NoSQL daemons, the distributed anomaly detector will be implemented as a proprietary C++ application that runs alongside the NoSQL databases. Keeping the anomaly detection code separate from the open-source implementation is important when it comes to complying with open-source licenses, while keeping the anomaly detection application proprietary allows us to protect our IP and create revenue off of our product.

The anomaly detection application appeals to companies that have NoSQL databases already in production but are not fully secured as well as companies that are secured but want to prevent misuse of their database system. We are targeting enterprises that are concerned with improving the security of their NoSQL databases. Futhermore, we are targeting enterprises that are committed to protecting their customers against attacks. Keeping consumer data protected against attacks and keeping the customers happy is important, notably for companies in cases where data breaches lead to class-action lawsuits against the company, resulting in lost revenue.

We are still working on the design of a working distributed anomaly detection prototype and plan to expand the anomaly detection to intrusion detection as NoSQL databases mature. We can market this prototype to companies currently using NoSQL databases, especially ones that are not already up to par with modern security. By offering this plugin, we will be able to make money not only from the plugin itself, but also from consulting on the current security of their system as well as how to seamlessly add our system to theirs and keep it up to date.

We have found dynamic SQL injection detection approaches for MySQL, but so far, we have not found any anomaly-based distributed IDS that works with NoSQL databases. As a result, this would introduce another frontier for MongoDB and Redis and more generally NoSQL security and anomaly detection.

IX. Conclusion and Future Work

We have described a prototype implementation of an IDS that runs alongside MongoDB and Redis. Our current implementation is able to perform anomaly detection on anomalous workloads through a workflow analysis. In addition, we have implemented two non-linear classification methods, k-nearest neighbor and support vector machine, for the anomaly detection. We created 9 unique YCSB workloads to test the effectiveness of our IDS. Our experiment showed that kNN algorithm worked perfectly on both MongoDB and Redis systems with high detection accuracy, while SVM could not separate the normal and anomalous workloads.

As described before, our prototype implementation is written in Python, but we plan to port our final product to C++ to improve performance and protect our IP. Additionally, our prototype is basic and can only run on a single machine. In the future we plan to write our new implementation in C++ and write it so that it scales to a large, distributed NoSQL system setup on Google Cloud services. This will not only include MongoDB and Redis, but also other NoSQL databases so that our detection framework can serve as a general tool for

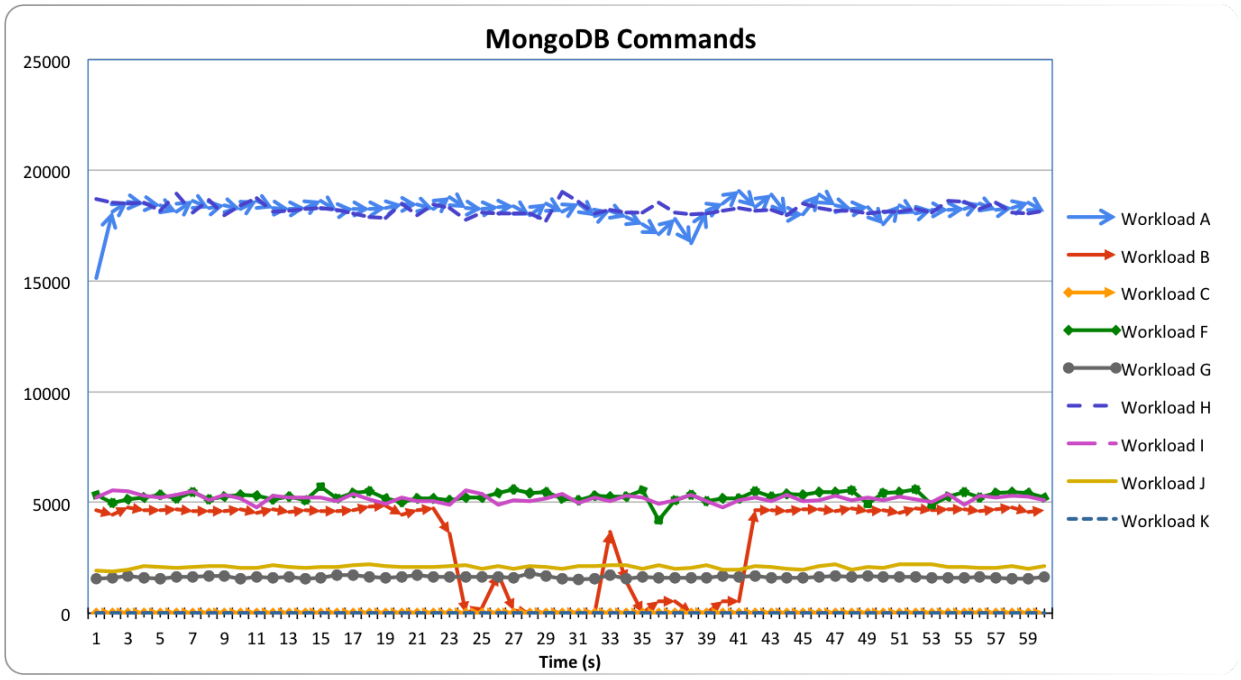

Fig. 2. MongoDB commands per second during training

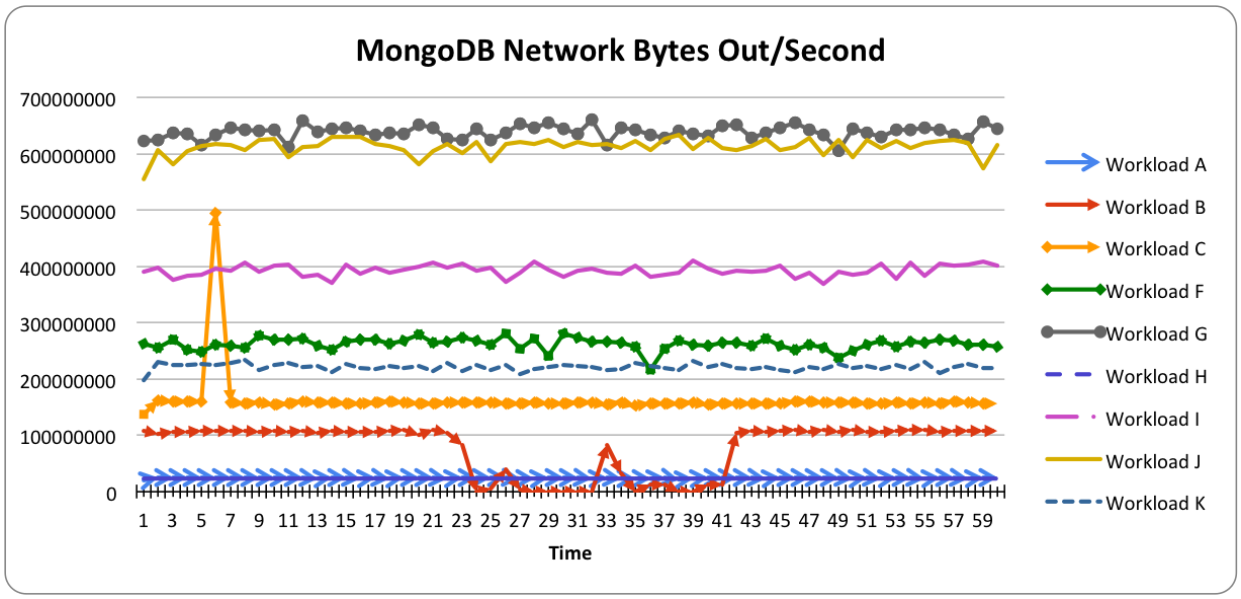

Fig. 3. MongoDB network output per second during training

NoSQL systems. Moreover, currently we only have one non-linear classification algorithm (kNN) to work, while SVM still has trouble dealing with such high-dimensional and complex data frame. In future work we will also try to improve the SVM model with more representative kernel functions that can fit our application, and try to implement other non-linear classification framework such as multi-layer perceptron (MLP, a feedforward artificial neural network model). Finally, we will also work on the development of our IDS to turn it into an Intrusion Protection System (IPS) to dynamically react and detect the anomalous behaviors in the system.

REFERENCES

[1] "IT downtime costs $26.5 billion in lost revenue." [Online]. Available: http://www.informationweek.com/storage/disaster-recovery/it-downtime-costs-265-billion-in-lost-re/229625441

[2] "DDoS and downtime: Considerations for risk management," May 2012. [Online]. Available: http://www.verisigninc.com/assets/whitepaper-ddos-risk-management.pdf

[3] "MongoDB." [Online]. Available: https://www.mongodb.org/

[4] "Redis." [Online]. Available: http://redis.io/

[5] R. Gopalakrishna, "A framework for distributed intrusion detection using interest- driven cooperating agents," CERIAS, Purdue University, Tech. Rep. [Online]. Available: https://www.cerias.purdue.edu/papers/archive/2001-44.pdf

[6] J. Cabrera, C. Gutierrez, and R. Mehra, "Infrastructures and algorithms for distributed anomaly-based intrusion detection in mobile ad-hoc networks," in *Military Communications Conference, 2005. MILCOM 2005. IEEE*, Oct 2005, pp. 1831–1837 Vol. 3.

[7] K. Das, K. Bhaduri, and P. Votava, "Distributed anomaly detection using 1-class SVM for vertically partitioned data," *Stat. Anal. Data Min.*, vol. 4, no. 4, pp. 393–406, Aug. 2011. [Online]. Available: http://dx.doi.org/10.1002/sam.10125

[8] F. Valeur, D. Mutz, and G. Vigna, "A learning-based approach to the detection of SQL attacks," in *Proceedings of the Second International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 123–140. [Online]. Available: http://dx.doi.org/10.1007/11506881_8

[9] Xavier. (2015, Nov) MySQL attacks self-detection. [Online]. Available: http://blog.rootshell.be/2012/11/01/mysql-attacks-self-detection/

[10] H. Shahriar, S. North, and W. Chen, "Early detection of SQL injection attacks."

[11] S. Bandhakavi, P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan, "Candid: Preventing SQL injection attacks using dynamic candidate evaluations," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 12–24. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315249

[12] P. Bodík, R. Griffith, C. Sutton, A. Fox, M. Jordan, and D. Patterson, "Statistical machine learning makes automatic control practical for internet datacenters," in *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, ser. HotCloud'09. Berkeley, CA, USA: USENIX Association, 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855533.1855545

[13] Y. Xie and S. zheng Yu, "A large-scale hidden semi-markov model for anomaly detection on user browsing behaviors," *IEEE/ACM Transactions on Networking*, vol. 17, no. 1, pp. 54–65, Feb 2009.

[14] C. Wang, V. Talwar, K. Schwan, and P. Ranganathan, "Online detection of utility cloud anomalies using metric distributions," in *2010 IEEE Network Operations and Management Symposium (NOMS)*, April 2010, pp. 96–103.

[15] C. Wang, K. Viswanathan, L. Choudur, V. Talwar, W. Satterfield, and K. Schwan, "Statistical techniques for online anomaly detection in data centers," in *2011 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, May 2011, pp. 385–392.

[16] A. Zahid, R. Masood, and M. Shibli, "Security of sharded NoSQL databases: A comparative analysis," in *2014 Conference on Information Assurance and Cyber Security (CIACS)*, June 2014, pp. 1–8.

[17] D. M. Farid, M. Z. Rahman, and C. M. Rahman, "Article: Adaptive intrusion detection based on boosting and nave bayesian classifier," *International Journal of Computer Applications*, vol. 24, no. 3, pp. 12–19, June 2011, full text available.

[18] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 143–154.

[19] "Server status output - mongodb manual." [Online]. Available: http://docs.mongodb.org/manual/reference/server-status/

[20] "pymongo 2.8 : Python Package Index." [Online]. Available: https://pypi.python.org/pypi/pymongo/2.8

[21] "Info-redis." [Online]. Available: http://redis.io/commands/INFO

[22] "Benchmarking cloud serving systems with ycsb." [Online]. Available: http://www.cs.duke.edu/courses/fall13/compsci590.4/838-CloudPapers/ycsb.pdf

[23] "Server status output." [Online]. Available: http://docs.mongodb.org/manual/reference/server-status/

[24] "Info." [Online]. Available: http://redis.io/commands/INFO