2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)

Automatic Detection of NoSQL Injection Using Supervised Learning

Md Rafid Ul Islam*, Md. Saiful Islam†, Zakaria Ahmed‡, Anindya Iqbal§, and Rifat Shahriyar¶
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology
Dhaka, Bangladesh
Email:*rfd.009@gmail.com, †saifulislam@cse.buet.ac.bd, ‡zakaria.buet13@gmail.com,
§anindya@cse.buet.ac.bd, ¶rifat@cse.buet.ac.bd,

***Abstract*—With the advancement in big data, NoSQL databases are enjoying ever-growing popularity. The increasing use of this technology in large applications also brings security concerns to the fore. Historically, SQL injection has been one of the major security threats over the years. Recent studies reveal that NoSQL databases also have become vulnerable to injections. However, NoSQL security is yet to receive the attention it deserves from the industry or academia. In this work, we develop a tool for detecting NoSQL injections using supervised learning. To the best of our knowledge, our developed training dataset on NoSQL injection is the first of its kind. We manually design important features and apply various supervised learning algorithms. Our tool has achieved 0.93 F_2-score as established by 10-fold cross-validation. We also apply our tool to a NoSQL injection generating tool, NoSQLMap and find that our tool outperforms Sqreen, the only available NoSQL injection detection tool, by 36.25% in terms of detection rate. The proposed technique is also shown to be database-agnostic achieving similar performance with injection on MongoDB and CouchDB databases.**

***Index Terms*—NoSQL, Injection, Database Security, MongoDB, CouchDB, Automatic Detection, Machine Learning, Supervised Learning**

I. Introduction

NoSQL (Not only SQL) is an alternative to traditional SQL. NoSQL databases give us the ability to work with large sets of distributed data with greater efficiency. Applications requiring high performance and scalability can be effectively developed using NoSQL databases. NoSQL is able to process a very large amount of data and distribute them across computing clusters faster than SQL databases [1]. Along with providing high scalability and high performance, NoSQL is designed to deal with large volumes of rapidly changing structured or unstructured data, a flexible data model for big data, object-oriented programming, and the like. Because of these benefits, these databases are getting popular for large-scale cloud and web applications. Google, Facebook, Adobe, eBay, Cisco, etc. are using NoSQL databases for their web applications[1].

SQL injection attack is one of the oldest and most fatal security threats. Even today, many large organizations are frequently falling prey to SQL injection into their traditional SQL databases. Keizer [2] mentions that hackers stole more than 450,000 login credentials from Yahoo by exploiting an SQL injection vulnerability in their servers in 2012. Seals [3] shows that SQL injection attack was used to steal the personal details of 156,959 customers from British Telecommunications company TalkTalk's servers in 2015. Hacker0 et al. [4] show that it is possible to steal bitcoin with SQLi. Hence, this is still a major security threat.

Researchers found that NoSQL databases also face the risk of being affected by injection attacks [5]. NoSQL injection vulnerability is reported by Diaspora [6] in its social community framework in 2010. Sullivan [7] demonstrates JavaScript query based code injection attacks for MongoDB. As NoSQL databases are getting more and more popular, vulnerability issue is also becoming a major concern. An article published in 2015 shows that about 40,000 web apps that use MongoDB databases are vulnerable to injection attacks[2]. According to OWASP top 10 security ranking of 2017, injection is the topmost security threat for applications and NoSQL injections are among them[3] [4].

Since NoSQL injection is a relatively new type of threat, there has not been adequate work addressing this problem. Existing works mostly discuss the types of attacks that are applicable to NoSQL. For example, Ron et al. [8] and Hou et al. [9] show some ways to generate injection queries. These works only discuss the types and severe effects of NoSQL injection and present some mitigation techniques that can be applied in the development phase of a system. Based on the literature, these injections can be classified into four types - PHP array injection, NoSQL OR injection, Javascript based injection, piggybacked queries.

Basic protection against injection would be input sanitization. However, it does not save applications from all types of injections. Since the injection completes the query string by balancing the start and end of each string in the query, this type of injection works even if PHP string sanitization is applied on the query. For example, *OR-injection* also works after sanitization since it also balances out the string quotations. So, to protect applications from the risk of NoSQL injection,

[1]https://www.mongodb.com/who-uses-mongodb

[2]https://www.securityweek.com/thousands-mongodb-databases-found-exposed-internet

[3]https://www.owasp.org/index.php/Top_10-2017_Top_10

[4]https://www.owasp.org/index.php/Top_10-2017_A1-Injection

978-1-7281-2607-4/19/$31.00 ©2019 IEEE
DOI 10.1109/COMPSAC.2019.00113

it is important that an automated NoSQL injection detection tool is developed.

Eassa et al. [10] attempt automatic detection of NoSQL injection using a syntactic parser. Joseph et al. [11] design a tool using non-deterministic finite automata. They test their tools against a very small number of injections. Joseph et al. only mention two examples in their study which cover only trivial javascript injections. Eassa et al. can only detect PHP array injections. Diglossia [12] detects injection based on processing the part of the query containing user input. This approach involves converting user inputs to shadow characters and then detecting injections applying a dual parser which is based on the shadow characters. These studies are not well described and we failed to implement these models. The tools and datasets are also not available. So, the evidence and outcomes of these studies are not enough to convince us that the necessity for developing a reliable automatic NoSQL injection detection tool is addressed.

Sqreen [13] provides several publicly available tools for security monitoring and protection monitoring including support for NoSQL injection detection. However, Sqreen severely fails to detect Javascript based injection and piggybacked queries. Another weakness of this tool is it takes almost 10-20 seconds to detect an injection. Incorporating this tool may lead to slowing down the server extremely. So, an acceptable automatic detection tool is yet to be designed which is very important for dealing with attacks once a system is already deployed.

Recent success of supervised learning in automatic fraud and malware detection motivated us to explore this direction. Guruswamy [14] explains in his article on Forbes why machine learning models are better than rule-based systems. While rule based detection approaches suffer from possible attacks beyond the coverage of the rules, machine learning based approaches are likely to train themselves with properties of injections that are not visible while formulating the rules. Hence, they can detect new types of injection when attacked. Since no benchmark dataset is available for NoSQL injection queries, we first generate a dataset of benign and malicious MongoDB queries with extensive study of available relevant resources. The literature is far from being enriched. So, we are able to find a very small dataset to train our model which is not sufficient. We manually generate a large number of benign and injection queries. Then we manually augment our dataset by applying cross-overs (combining parts of two queries) and mutations (tweaking one element of a query) over the existing dataset. We validate the generated queries by developing a simple, vulnerable website which works on top of a MongoDB database. While replicating the experiments for CouchDB database, we follow the same procedure. Finally, Our dataset contains 1004 MongoDB (including 203 injections) and 350 CouchDB (including 50 injections) queries. We model the detection problem as a binary classification (benign query and injection query) problem and use this dataset to train popular supervised learning methods i.e., decision tree (ID3) [15], random forest [16], AdaBoost [17], neural network [18], [19], support vector machine (SVM) [20], k nearest neighbor (IBk) [21], and XGBoost [22]. We evaluate their performance using 10-fold cross-validation. Based on the experimental results on MongoDB, we have found that a model trained with a neural network provides the highest mean recall (92.94%) along with 91.87% mean accuracy, 93.55% mean precision, and 0.9343 mean F_β ($\beta = 2$) score.

For CouchDB dataset, despite dataset size is smaller by 65.12%, F_β score degrades by 3.85% only on average; which indicates that our approach is database-agnostic and can easily be extended to other NoSQL databases as well.

Note that in the context of threat detection, recall measures the percentage of the vulnerable components correctly predicted as such and is widely considered an effective criterion for recommending a model. F_β ($\beta = 2$) score combines the recall and precision measures and imposes lower weight on precision.

We also generate a separate test dataset containing injections only using a NoSQL injection generation tool named NoSQLMap[5]. The set of queries are independent of our training dataset and direct output of NoSQLMap without any kind of manual processing. We find that our tool can detect 36.25% more injections than Sqreen.

In summary, the specific contributions of our work are:

- We generate a dataset of 1354 NoSQL queries (including around 75% benign and 25% injection) and validate the dataset by practically testing on a local server. To the best of our knowledge, this is the first labeled dataset of this kind.
- We design 19 features for classifying benign and injection queries. These features seem to be highly effective as our tool performs quite good despite being trained on a small number of samples.
- We demonstrate that popular supervised learning models can effectively solve the NoSQL detection problem.
- We provide a publicly available tool and describe how to integrate our tool with an existing mechanism. To the best of our knowledge, our tool performs better than any publicly available tool of this kind. To encourage reproduction, we release the dataset and tool[6].

The remainder of the paper is organized as follows. Section II introduces NoSQL injection, its threat, and different types. We discuss studies in relevant fields in Section III. In Section IV, we describe our approach towards the design of an automated tool for detecting injections. In Section V, we evaluate the performance of our tool. In Section VI, we present how our tool can be integrated with a web application. We show the comparison of our tool with existing ones in Section VII. Finally, Section VIII concludes the paper.

II. Preliminaries

In this section, we demonstrate the risks of NoSQL databases and how NoSQL injection attacks are executed.

[5]https://github.com/codingo/NoSQLMap

[6]https://github.com/anonymous1363101/nosql-injection-detection

Later, we introduce different types of NoSQL injections we intend to detect.

A. NoSQL Injection

NoSQL database is a schema-free database that supports easy replication, simple API, and high consistency. This type of database provides higher performance and speed and consumes fewer resources. The most common data models in NoSQL databases are column-based, document-based, key-value mapping-based, graph-based, and multi-model. NoSQL databases, such as MongoDB, CouchDB, etc, are yet to be robust against security attacks. Malicious users can exploit these security vulnerabilities to execute privilege escalation attacks to get access to other user accounts of same or higher privilege levels. When NoSQL is first introduced, it is thought to be free of injections, unlike the traditional SQL databases. But later the works by Hou et al. [23], Okman et al. [24], and Ron et al. [8] show that NoSQL databases are also vulnerable to some injections similar to SQL injections.

In 2015, three students of University of Saarland, Germany showed that about 40,000 MongoDB databases on the internet are vulnerable[7]. They claimed to be able to get read and write access to thousands of databases containing sensitive customer data from web shops without any special hacking tools. They reported the existence of many MongoDB web servers that remain vulnerable to injection attacks.

OWASP[8] and an IBM study [8] have also shown that NoSQL databases are vulnerable to injection attacks, although they do not use traditional SQL syntax.

Consider the following script for a login form where user inputs username and password.

```
$collection->find(array(
“username” => $_GET[‘username’],
“password” => $_GET[‘password’]
));
```

When a user provides the username and password, it sends an http request. For example, if username is **admin** and password is **12345678**, the corresponding http request is

login.php?username=admin&password=12345678

The script matches the username and password and returns true if both are correct. Now, an attacker can alter the query by passing an array as input like this,

login.php?username[$ne]=null&password[$ne]=null

This creates the following MongoDB query,

```
$collection->find(array(
“username” => array(“$ne” => null),
“password” => array(“$ne” => null)
));
```

This query eventually exposes all the entries where username and password are not *null*. Thus an attacker is able to get unauthorized information from MongoDB.

An attacker may also append an additional query with the original one by manipulating input. For example, when username is **G. R. R. Martin** the query is,

db.doc.find({ username: ‘***G. R. R. Martin***’ })

Now, if an attacker put **G. R. R. Martin’});db.dropDatabase(); db.insert({username: ‘dummy’, password: ‘dummy** as username, the following query will be executed:

db.doc.find({ username: ‘**G. R. R. Martin’**}); **db.dropDatabase(); db.insert(**{**username: ‘dummy’, password: ‘dummy**’})

MongoDB treats this query as three independent queries instead of one and runs all of them. Here, the second query deletes the database completely which is disastrous.

Sqreen [13] shows that it is very easy to attack a MongoDB database using injection and change the content of the database if no security measure is taken by the developer[9]. A Node.js application with JSON data format is also vulnerable if no security mechanism is applied.

B. Types of NoSQL Injections

Here, we introduce 4 types of injections applicable to NoSQL. Although all types of injections are possible for MongoDB, we find that only 2 types are applicable to CouchDB.

1) PHP Array Injection: The Table I shows a scenario where user input in a login form is exploited to execute an injection attack. PHP array injections inject PHP codes into an application so that the query conditions are modified. When the server executes this modified query, the attacker gains information that is not supposed to be retrieved by the original query.

2) NoSQL OR Injection: Unlike SQL queries, JSON structure makes ‘OR injections’ hard in MongoDB and CouchDB,

TABLE I
PHP ARRAY INJECTION EXAMPLE

Database Type	Query	Injection
MongoDB	db.logins.find({ username: { **$ne: 1** }, password:{ **$ne: 1** } })	{ $ne: 1 }
CouchDB	POST /users/_find HTTP/1.1 Accept: application/json Content-Type: application/json Host: localhost:5984 { “selector”: { “username”: { **“$ne”: null** } } }	{ “$ne”: null }

[7]https://www.securityweek.com/thousands-mongodb-databases-found-exposed-internet

[8]https://www.owasp.org/index.php/Testing_for_NoSQL_injection

[9]https://blog.sqreen.io/mongodb-will-not-prevent-nosql-injections-in-your-node-js-app/

TABLE II
OR INJECTION EXAMPLE

Database Type	Query	Injection
MongoDB	db.doc.find({ username: 'tolkien', **$or:[** {}, { **'a': 'a',password:** ''} **], $comment: 'successful MongoDBinjection**'})	', $or:[{}, { 'a': 'a',password: ''}], $comment: 'successful MongoDBinjection
CouchDB	POST /users/_find HTTP/1.1 Accept: application/json Content-Type: application/json Host: localhost:5984 { "selector": { "username": "vchaulk0**", "$or":** [{ "password": "12345**" }, { "password":** { **"$ne": "null"** } }] } }	", "$or": [{ " }, { "password": { "$ne": "null" } }]

TABLE III
JAVASCRIPT INJECTION

Query	Injection
db.stores.mapReduce (function() { for (var i = 0; i ¡ this.items.length; i++) { emit(this.name, this.items[i].**a);** } },**function(kv)** { **return 1;** }, { **out: 'x'** }**); db.injection.insert** ({**success:1**}**); return 1;db.stores.mapReduce (function()** { { **emit(1,1**); } }, function(name, sum) { return Array.sum(sum); }, { out: 'totals' });"	a); } },function(kv) { return 1; }, { out: 'x' }); db.injection.insert ({success:1}); return 1;db.stores.mapReduce (function() { { emit(1,1

but still, it is possible to bypass security procedures by injecting an always true condition (for example, an empty string) using 'OR' keyword.

In Table II, we find that an empty expression is attached to the input using an OR condition. An empty expression is always true and consequently, it makes the password check ineffective.

3) JavaScript Based Injection: An attacker can forcefully return true or put a condition inside *$where* operator that always results as true. For example, if there is a condition like '' == '', it will always give true value. As MongoDB allows execution of JavaScript codes in order to perform complex queries, it is also possible to inject malicious commands by manipulating JavaScript functions. This type of injection is not possible in CouchDB. An example is shown in Table III.

4) Piggybacked Queries: In MongoDB, attackers can exploit assumptions in the interpretation of escape sequences and special characters (such as termination characters like carriage return [CR], line feed [LF], closing braces, and semicolons) to end a query and insert additional harmful queries like *db.dropDatabase()* to be executed by the database, which can lead to disastrous effects like deleting all users from the database. CouchDB does not have this type of injections. Table IV shows an injection of this type.

III. RELATED WORKS

Database and web security are among the most threatening areas in information security. Albeit many works have been performed on SQL injection, it is still one of the major vulnerabilities of a database. Some of the notable SQL injection attack incidents are presented in Table V.

SOFIA [25] is a programming-language and source-code independent tool which also can be used with various attack generation tools. While most approaches detect user inputs

TABLE IV
PIGGY-BACKED QUERY

Query	Injection
db.doc.find({ username: 'G. R. R. Martin'}**); db.dropDatabase(); db.insert({username: 'dummy ', password: 'dummy** '})	'}); db.dropDatabase(); db.insert({username: 'dummy ', password: 'dummy

TABLE V
SQL INJECTION DAMAGE STATISTICS

Organization	Damage	Year
Yahoo	450,000 plain text passwords stolen	2012
LinkedIn	6.5 million hashed passwords hacked	2012
Bitcoin	Hacker shows bitcoin can be stolen with SQLi	2016
Arizona voter database	Data of 200,000 voters stolen	2016
Qatar National Bank	Sensitive financial information leaked	2016
Hetzner South Africa	Over 40,000 customer details including bank accounts leaked	2017

in SQL statements and compare them with safe user inputs to detect injection, this method parses SQL statements and creates parse trees, which are fed to a clustering algorithm. Then the tree edit distance is used to measure the distance among the parse trees. The challenge of this approach is to learn to characterize benign SQL statements so that it can accurately identify one from an injection attack statement.

Most of the automatic detection studies use parsing based analysis [26] [27]. In recent years, supervised learning has becoming popular for automatic fraud or malware detection with high accuracy. Impression fraud detection by Haider et al. [28], Android malware detection by Amos et al. [29], JavaScript malware detection by Wang et al. [30], malicious web content detection by Hou et al. [31], web application vulnerability prediction by Shar et al. [32], predicting cross-site scripting (XSS) security vulnerabilities by Gupta et al. [33], and predicting vulnerable software components via text mining by Scandariato et al. [34] are some of the examples.

In spite of the growing popularity of NoSQL databases, there are few recent works pointing out its security issues. Leavitt et al. [35] discusses issues such as limitations, advantages, concerns, and doubts regarding NoSQL databases. Ron et al. [8] lists some possible types of code injection in NoSQL databases. Swathy Joseph and Jevitha Kp [11] discuss an automata-based approach to prevent NoSQL injections. Eassa et al. [10] proposes a tool called 'NoSQL Racket' which can detect only 'PHP array injections' by comparing code static code analysis and runtime code analysis. Hong et al. [36] proposes a parse tree based injection detection mechanism for NoSQL databases. Okman et al. [24] reviews two of the most popular NoSQL databases (Cassandra and MongoDB) and outlines their main security features and problems. Some existing works only address injection detection in queries from trusted client applications. For example, Diglossia [12] converts user input to shadow characters and employs a methodology to separate user input from the query. Then, based on user input

CCS13论文的不足之处

TABLE VI
SOURCES USED FOR COLLECTING NOSQL BENIGN QUERIES

https://docs.mongodb.com/manual/,
https://www.tutorialspoint.com/mongodb/,
https://www.journaldev.com/6221/mongodb-findandmodify-example,
http://php.net/manual/en/mongocollection.findandmodify.php ,
https://specify.io/how-tos/find-documents-in-mongodb-using-the-mongo-shell,
http://no-fucking-idea.com/blog/2012/04/01/using-map-reduce-with-mongodb/ ,
http://thejackalofjavascript.com/mapreduce-in-mongodb/ ,
http://www.querymongo.com/ ,
https://stackoverflow.com/questions/30435073/mysql-to-mongodb-query-conversion-issue ,
https://stackoverflow.com/questions/27915598/how-to-convert-group-by-having-query-from-mysql-to-mongodb-in-phalcon ,
https://stackoverflow.com/questions/42692413/sql-query-convert-to-mongodb ,
http://docs.couchdb.org/en/2.1.1/

TABLE VII
SOURCES USED FOR COLLECTING NOSQL INJECTION QUERIES

https://www.idontplaydarts.com/2010/07/mongodb-is-vulnerable-to-sql-injection-in-php-at-least/,
https://zanon.io/posts/nosql-injection-in-mongodb,
http://blogs.adobe.com/asset/files/2011/04/NoSQL-But-Even-Less-Security.pdf,
http://www.syhunt.com/?n=Articles.NoSQLInjection,
http://docs.mongodb.org/manual/faq/developers/#how-does-mongodb-address-sql-or-query-injection,
http://php.net/manual/en/mongocollection.find.php,
http://blog.websecurify.com/2014/08/hacking-nodejs-and-mongodb.html,
http://blog.websecurify.com/2014/08/attacks-nodejs-and-mongodb-part-to.html,
https://security.stackexchange.com/questions/83231/mongodb-nosql-injection-in-python-code,
https://www.infoq.com/articles/nosql-injections-analysis,
https://www.owasp.org/index.php/Testing_for_NoSQL_injection

analysis, they detect injections. Hou et al. [23] also proposes a strategy to prevent injections, which also depends on the proper control of client application.

IV. DESIGN OF DETECTION MODEL

In this study, we have used feature based supervised learning classifiers to detect injections. To train the classifiers, we had to generate our own dataset since there is no labeled dataset on NoSQL injection available. The methodology for the development of the proposed tool is shown in Figure 1.

A. Training Dataset Generation

Since the literature is far from being rich, we first generate a dataset of benign and malicious MongoDB queries with extensive study of available relevant resources. So, we are able to find a very small dataset to train our model which is not sufficient. The benign queries are collected from MongoDB and CouchDB official sites and some other links (Table VI). And, the injection data is collected from popular security sites, blogs, and studying the state of the art works (Table VII). Then, we manually generate a large number of benign and injection queries by augmenting the dataset by applying cross-overs (combining parts of two queries) and mutations (tweaking one element of a query) over the existing dataset. We validate the generated queries by developing a simple, vulnerable website which works on top of a MongoDB database. While replicating the experiments for CouchDB database, we follow the same procedure. Finally, Our dataset contains 1004 MongoDB (including 203 injections) and 350 CouchDB (including 50 injections) queries.

In our dataset, we have included all 4 types mentioned in Section II-B, i.e. PHP array injection, NoSQL OR injection, JavaScript-based injection, and piggybacked queries. We have tried to create a balanced dataset, but the number of 'OR injection' is less than the other injections in our dataset. The reason is the absence of many variations in 'OR injection' like we have found in other types of injections. In Table VIII, we present some examples and the distribution of different types of injections in our training dataset. The details of these four types of injections have already been discussed in Section II.

We have generated a labeled dataset for both MongoDB and CouchDB. MongoDB dataset contains more samples because it is being used by many web applications and thus its benign and injection examples more available. Table IX contains the summary of our dataset.

B. Feature Design

Selecting appropriate features are very important for any feature-based supervised learning classifier. We have designed 19 features to start with. Then we have selected the 10 highest ranked features based on information gain and correlation. It is to be noted that any individual feature will not dictate the decision of detection, rather in combination with other features they yield a weighted outcome of the classifier that determines our prediction. The designed features and the intuitions behind choosing these are briefly explained below.

- Contains Empty String: A lot of NoSQL injections use an empty string to create a condition that evaluates to true for most database entries.
- Contains Injection Payload: A payload file contains some common substrings or signatures of NoSQL injections. We have taken the original NoSQL injection payload file from cr0hn's GitHub repository[10]. Then we have added a few more of injection payloads found on the web.
- Contains Not Equal: *$ne* keyword is present in most of the PHP array injections.
- Contains Comparison: *find()*, *find.sort()*, *$eq*, *$gt*, *$gte*, *$lt*, *$lte*, *$ne*, *$in*, and *$nin* are used to select the relevant entries that a user needs. These keywords are found in most NoSQL injections, as well as benign queries.
- Contains Logical Operator: *$or*, *$and*, *$not*, and *$nor* keywords are found to be used to create always true

[10]https://github.com/cr0hn/nosqlinjection_wordlists/blob/master/mongodb_nosqli.txt

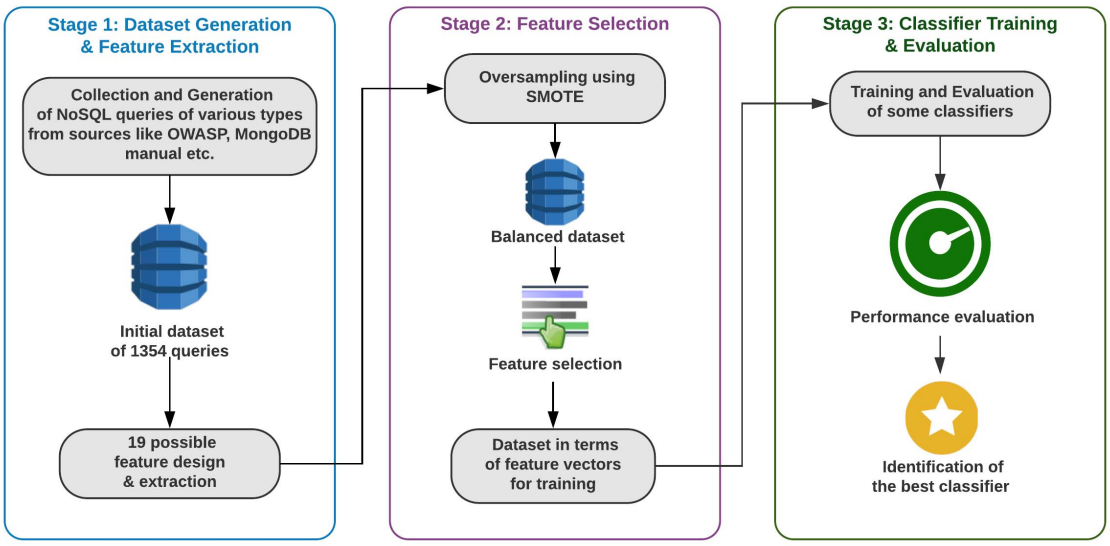

Fig. 1. Conceptual model of our solution strategy

TABLE VIII
MONGODB AND COUCHDB INJECTION QUERIES SUMMARY

Injection Type	No of Samples	Example
PHP array injection	134	db.logins.find({ username: { **$ne: 1** }, password: { **$ne: 1** } })
OR Injection	20	db.users.find({ username: "dummy**", $or: [** {}, { **password: ""** }]})
JavaScript based Injection	43	db.users.find({username: 'admin**', $where: function(){return 1}})//**', password: 'abcd'})
Piggybacked queries	56	db.users.find({username: ''}**);db.users.drop();** **db.users.insert({username:'rafid', password: 'rafid**'})

TABLE IX
MONGODB AND COUCHDB DATASET STATISTICS

Database	Query Type	No of Samples in Dataset
MongoDB	Benign	801
	Injection	203
CouchDB	Benign	300
	Injection	50

statements.

- Contains Evaluation Query Operation: Most JavaScript and some PHP injections contain *$where*, *$mod*, *$regex*, and *$text* command.
- Presence of *;return*, *return true*, and *return 1*: Diglossia [12] shows that some injections are possible using *return* command. But most JavaScript benign queries contain *return*, too. We have found three variants that are most of the times considered malicious. These are - *;return*, *return true*, and *return 1*.
- New Query: If a new query starts after another where should have been only one query, it is a type of injection called 'Piggy Backing'.
- Always True Expression: *//*, */.*/* etc indicate 'any' in regular expression. When they are used as injection the query becomes true for every entry in the database. Thus, privacy breach may happen.
- Contains Element Query Operations: MongoDB element query operations e.g., *$exisits* and *$type* are strategically used in injection attacks.

- Null comparison: Most NoSQL injections contain null. If *null* is present inside the query, then it may be comparing something with *null* which always yields true in NoSQL (e.g. MongoDB) syntax.

- Targets Table: *createTable()* and *showTable()* commands can respectively create a new table or show the current table. Both of which can be used as malicious commands to create an access point or to get confidential data.

- Alters Collection: *createCollection()* and *drop()* commands affect the database directly. These are usually not allowed to be performed through user input or input from a Rest API.

- Drop Database: *dropDatabase()* command deletes the entire database and its entries.

- Update query: *$update* and *$save* commands can change the data entries.

- Remove query: An attacker can use the *$remove* command to remove important data from a database.

- Limit keyword: *Limit* keyword is used to restrict access to all data entries. But attackers can exploit it to get access to more data than they have access to.

- Infinite Loop: *while(true)* will send the server to execute an infinite loop which may be used to commit a denial of service attack.

- Contains ;}//: Diglossia [10] showed that *;}//* can be used tactically for stronger Javascript attacks. Hence, if a query contains *;}//*, it has a higher possibility of being an injection.

C. Feature Selection

We use WEKA's ClassifierSubsetEval [37] with J48(decision tree) [38], IBK(k nearest neighbor) [21] classifiers, and greedy step-wise search with backward elimination to select and rank 10 out of initially designed 19 features based on information gain and correlation (Table X) separately. This is done by combining both our dataset of MongoDB and CouchDB.

We select these 10 features to improve the performance of our classifiers and find that reducing feature dimension significantly improves our model in terms of accuracy, precision, recall, and F_β ($\beta = 2$) score.

V. Evaluation And Results

In this section, we discuss the evaluation methods and present the performance measures of different classifiers.

TABLE X
Feature Ranking by Information Gain and Correlation

Rank	By Information Gain	By Correlation
1	Contains Comparison	Contains Comparison
2	New Query	New Query
3	Contains Empty String	Contains Empty String
4	Contains Not Equal	Contains Not Equal
5	Contains Payload	Contains Payload
6	Presence of Return	Always True Expression
7	Always True Expression	Presence of Return
8	Evaluation Query Operation	Evaluation Query Function
9	Contains Logical Operator	Element Query Operation
10	Element Query Operation	Contains Logical Operator

A. Evaluation Methodology

We design the detection problem as a binary classification (where the two classes are Benign and Injection) using 10 selected features mentioned in Table X. We use supervised learning classifiers such as - decision tree based ID3 algorithm [15], artificial neural network [18], [19] with back-propagation, random forest [16], AdaBoost [17], k nearest neighbor (IBk) [21], support vector machines (SVM) [20], and XGBoost [22]. We investigate the performance of the classifiers using 10-fold cross-validation. In 10-fold cross-validation, the dataset is randomly partitioned into 10 equal folds. Then one of the folds is selected as the validation set and the remaining 9 folds are selected to train the classifier. We repeat it 10 times to use each of the folds as the validation set exactly once. The final estimation is the average of the 10 results from the folds. We also test our model with a separate test dataset where injections are generated using the NoSQL injection generation tool named NoSQLMap[11] (both MongoDB and CouchDB). This tool is not used while generating our original dataset.

Our training dataset is imbalanced. For MongoDB dataset of 1004 queries, the ratio of benign to malignant queries is $3.95 : 1$. And, for CouchDB dataset of 350 queries this ratio of benign to malignant is $6 : 1$. Hence, we use oversampling with SMOTE (synthetic minority oversampling technique) [39] to improve the ratio of benign and malignant queries to $1.13 : 1$ (for MongoDB) and $1.1 : 1$ (for CouchDB). We tune SMOTE parameters such as SMOTE percentage to 250% and Number of Neighbors to 2 for the MongoDB dataset and SMOTE percentage to 450% for CouchDB dataset. Table XII shows the performance measures after applying oversampling.

We experiment with the 7 classifiers tuning their hyper-parameters to obtain better trained models. Based on the consistency of performance metrics on training and validation sets, we can claim that our model is not overfitted. The parameter values given in Table XI are found to be optimal for each classifier.

[11]https://github.com/codingo/NoSQLMap

TABLE XI
PARAMETERS USED FOR SEVEN CLASSIFIERS

Classifier	Parameter Name	Value
Decision Tree(ID3)	No Parameters	Null
Random Forest	Size Per Bag	100
	Number of Iterations	200
	Number of Trees	200
AdaBoost	Classifier Used	J48
	Number of Iterations	1000
	Use Resampling	True
	Percentage of Weight Mass to base	100
Neural Network	Learning Rate	0.05
	Maximum Epochs	2000
	Number of Hidden Layers	4
	Number of Nodes in Hidden Layer	10, 10, 6, 10
SVM	Type of SVM	C-SVM, C = 1
	Kernel Function	$e^{-\gamma\|u-v\|^2}$
	Class Weights	{1, 1}
k Nearest Neighbor	Number of Neighbors	5
XGBoost	Maximum Depth	2
	Objective Function	binary logistic

We ensure a wide coverage of different types of classifiers that are commonly used for similar problems.

B. Results

The results of our selected feature set (Table X) with respect to evaluation measures such as accuracy, precision, recall, and F_β score for 7 classifiers are given in Table XII. These results are obtained using the oversampled dataset. Before applying oversampling, the precision has been 89.06% (for MongoDB), 94.68% (for CouchDB) and recall has been 76.65% (for MongoDB) and 64.4% (for CouchDB), respectively. After oversampling, we see that recall has increased for both datasets with only a slight decrease in precision in CouchDB.

We observe from Table XII that even though precision is higher in other classifiers, recall and F_2 score is higher in neural network. As even one injection query execution can compromise the whole system, we think recall is the most important measure in this case. That is why we choose F_β ($\beta = 2$) score with less weight on precision to be our deciding performance measure. With the highest F_2 score, neural network is selected as our representative model for detecting NoSQL injections for both MongoDB and CouchDB.

We also calculate the confidence interval of classification error for neural network from our 10-fold cross-validation results. We use the *Wilson Score Interval* [40] to calculate the confidence interval denoted by I to assess the reliability of the result.

$$I = error \pm const \times \sqrt{\frac{\text{error} \times (1 - \text{error})}{N}} \qquad (1)$$

TABLE XII
PERFORMANCE MEASURES OF 10-FOLD CROSS-VALIDATION OF DIFFERENT CLASSIFIERS

Dataset	Classifier	Accuracy	Precision	Recall	F_2 Score
MongoDB	Decision Tree (ID3)	91.6642%	93.4370%	92.6929%	0.932872
	Random Forest	91.8772%	93.5465%	92.9375%	0.932460
	AdaBoost (boosting with J48)	91.7880%	93.4722%	92.8735%	0.933518
	Neural Network	**91.8772%**	**93.5537%**	**92.9392%**	**0.934302**
	SVM	89.4552%	91.0479%	91.5189%	0.91
	k Nearest Neighbor	91.6196%	93.3030%	92.7668%	0.931952
	XGBoost	89.5101%	90.79104%	87.9179%	0.884429
CouchDB	Decision Tree (ID3)	88.3333%	90.7801%	85.3333%	0.896358
	Random Forest	88.5666%	90.8256%	85.8%	0.897641
	AdaBoost (boosting with J48)	88.6333%	90.8386%	85.9333%	0.898132
	Neural Network	**88.6667%**	**90.8451%**	**86.00%**	**0.898328**
	SVM	85.2%	84.6685%	85.9667%	0.849250
	k Nearest Neighbor	88.6667%	90.8451%	86.0%	.898328
	XGBoost	85.36%	85.00%	84.06%	.842463

TABLE XIII
CONFIDENCE INTERVAL OF CLASSIFICATION ERRORS OF THE CLASSIFIERS

Dataset	Classifier	Confidence Interval of Classification Error
MongoDB	Neural Network	[7.9974%, 8.0501%]
CouchDB	Neural Network	[11.2297%, 11.4368%]

In (1), $error$ implies the classification error, $const$ is the constant value that defines the likelihood and N is the number of observations used to evaluate the model. The constant value we have used is 1.96 for 95% likelihood.

Table XIII shows the result of the confidence interval of classification error of neural network for 95% likelihood. For neural network classifier on MongoDB dataset, the value of confidence interval implies that there is a 95% likelihood that the true classification error of the model is in the interval [7.9974%, 8.0501%] for unseen data.

VI. DEPLOYMENT STRATEGY

Our proposed tool will work as a server plugin. It works in two steps. First, a listener listens to the port where the server communicates with the NoSQL database and forwards it to the port our tool (classifier) listens on. For example, if the server code sends a query to MongoDB on port 100 and our tool listens on port 101, our listener which intercepts the data coming through port 100 will forward the query to port 101. Next, our tool filters and sends only the benign query back to MongoDB's port 100. Hence, every query would be filtered through our tool before running in a NoSQL database management system. The conceptual model of the proposed system is presented in Figure 2.

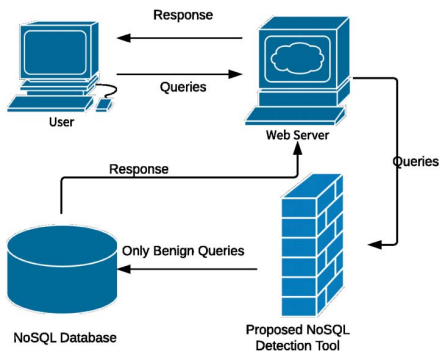

Fig. 2. Conceptual model of our proposed system

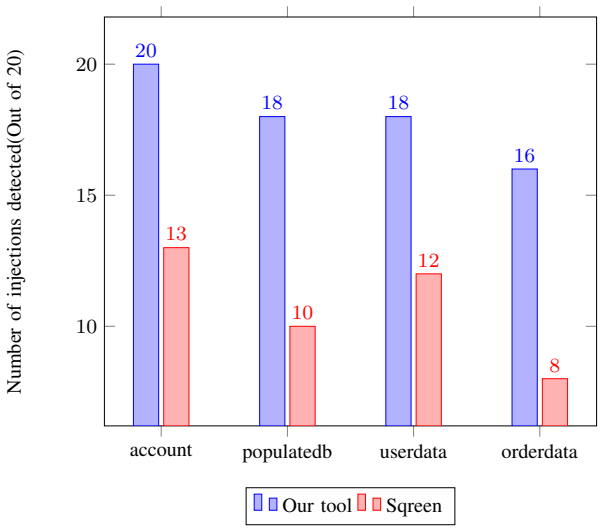

Fig. 3. Performance comparison between Sqreen and our tool

VII. Comparative Study

We have used an independent third-party tool, NoSQLMap[12] to generate injections for testing our tool against injection attacks outside our original dataset. NoSQLMap has three sample databases such as shops, customers, and appUserData. This tool also provides four vulnerable web applications such as account, populatedb, userdata, orderdata. We have generated 20 injections through NoSQLMap for each of these four web applications and tested the generated injections for Sqreen [13] and our tool.

Sqreen has detected 13, 10, 12, and 8 injections for each test set and our tool has detected 20, 18, 18, and 16 injections, respectively. The performance comparison between the two tools is shown in Figure 3. The detection rate of our method is 36.25% higher on average than Sqreen.

Sqreen's NoSQL injection detection support is only available for Ruby and Node.js based servers. On the contrary, our tool is platform independent. Sqreen supports only one NoSQL database, i.e., MongoDB. On the contrary, our scheme can be extended to work with other NoSQL databases with minor adjustments as we have already demonstrated it with CouchDB.

From a careful study, we have found that Sqreen can detect only a few types of injections and fails against some important ones. It can detect some PHP array injections and OR injections, however, is helpless against JavaScript-based injections(Table - III) and piggy-backed injections(Table - IV). Our scheme can detect all of these with moderate accuracy, as the designed supervised learning approach does not rely on any particular syntax structure.

We also have found some studies on detecting NoSQL injection attacks. The works of Eassa et al. [10] and Joseph et al. [11] are the most relevant ones. However, they have not released their tools and hence we could not compare with those.

VIII. Conclusion and Future Works

Despite having significant security risks, prevention of NoSQL injection is not getting the attention it deserves. In this work, we propose an automated system to detect any type of query which may lead to NoSQL injection attack and demonstrate the performance of the system for MongoDB and CouchDB databases. Our major contribution is the generation of the labeled dataset containing around 1350 NoSQL queries. We explore multiple machine learning methods with careful tuning of hyper-parameters for classification and present the performance of these methods in terms of accuracy, precision, recall, and F_β score. We can claim that our system can detect most of the injections based on extensive experiments. As detecting an injection is our major concern, we recommend neural network as the most effective method as it provides the highest recall and F_2 score. We also compare our study with the only available tool, Sqreen and observe that our system significantly outperforms it.

To the best of our knowledge, this is the first work to propose a methodology based on supervised learning which can detect NoSQL injection attacks. Our tool provides high accuracy and enables server injection vulnerability testing by professionals without disclosing the confidential application code of an enterprise. However, the automatic design of features from relevant literature can be an interesting research direction, but it is quite impossible as the relevant corpus is very small. A larger dataset is also more likely to improve the performance. We leave these issues as possible future works.

Acknowledgment

We want to express our gratitude to the authors of 'SOFIA: an automated security oracle for black-box testing of SQL injection vulnerabilities' for providing their dataset on benign SQL queries. We also want to thank Vladimir de Turckheim and other members of Sqreen team for providing us valuable instructions on using their tool and other information. And

[12]https://github.com/codingo/NoSQLMap

last but not least, we thank Samsung for supporting us by their research grant "Code Review Usability Measurement".

REFERENCES

[1] MongoDB, "Nosql databases explained," https://www.mongodb.com/nosql-explained.

[2] G. Keizer, "Yahoo fixes password-pilfering bug, explains who's at risk," 2012.

[3] T. Seals, "Sql injection possible vector for talktalk breach," Infosecurity Magazine, 10 2015, accessed: December 02, 2017.

[4] e. . hacker0 (25), oumar (57), "How i hacked hundreds of bitcoins! ama," Steemit, 8 2016, accessed: December 02, 2017.

[5] "Infoq," https://www.infoq.com/articles/nosql-injections-analysis, 1 2017.

[6] "Security lessons learned from the diaspora launch," http://www.kalzumeus.com/2010/09/22/security-lessons-learned-from-the-diaspora-launch.

[7] B. Sullivan, "Server-side javascript injection," Senior Security Researcher, Adobe Secure Software Engineering Team, 6 2011.

[8] A. Ron, A. Shulman-Peleg, and E. Bronshtein, "No sql, no injection? examining nosql security," *arXiv preprint arXiv:1506.04082*, 2015.

[9] B. Hou, Y. Shi, K. Qian, and L. Tao, "Towards analyzing mongodb nosql security and designing injection defense solution," in *Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing (HPSC), and IEEE International Conference on Intelligent Data and Security (IDS), 2017 IEEE 3rd International Conference on*. IEEE, 2017, pp. 90–95.

[10] A. M. Eassa, O. H. Al-Tarawneh, H. M. El-Bakry, and A. S. Salama, "Nosql racket: A testing tool for detecting nosql injection attacks in web applications," *International Journal of Advanced Computer Science and Applications*, vol. 8, no. 11, 2017. [Online]. Available: http://dx.doi.org/10.14569/IJACSA.2017.081178

[11] S. Joseph and K. Jevitha, "An automata based approach for the prevention of nosql injections," in *International Symposium on Security in Computing and Communication*. Springer, 2015, pp. 538–546.

[12] S. Son, K. S. McKinley, and V. Shmatikov, "Diglossia: detecting code injection attacks with precision and efficiency," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 1181–1192.

[13] Sqreen, "Web application and user protection," https://www.sqreen.io.

[14] K. Guruswamy, "Data science: Machine learning vs. rules based systems," *Forbes*, Dec 2015.

[15] C. Jin, L. De-Lin, and M. Fen-Xiang, "An improved id3 decision tree algorithm," in *Computer Science & Education, 2009. ICCSE'09. 4th International Conference on*. IEEE, 2009, pp. 127–130.

[16] T. K. Ho, "Random decision forests," in *Proceedings of the Third International Conference on*. IEEE, 1995.

[17] S. R. Freund Y., "A desicion-theoretic generalization of on-line learning and an application to boosting," in *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, 2005, pp. 23–37.

[18] J. J. Hopfield, "Artificial neural networks," *IEEE Circuits and Devices Magazine*, vol. 4, no. 5, pp. 3–10, 1988.

[19] Y. LeCun, "A theoretical framework for back-propagation," in *Artificial Neural Networks: concepts and theory*, P. Mehra and B. Wah, Eds. Los Alamitos, CA: IEEE Computer Society Press, 1992.

[20] N. Cristianini and J. Shawe-Taylor, "An introduction to support vector machines," 2000.

[21] D. Aha and D. Kibler, "Instance-based learning algorithms," *Machine Learning*, vol. 6, pp. 37–66, 1991.

[22] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," *CoRR*, vol. abs/1603.02754, 2016. [Online]. Available: http://arxiv.org/abs/1603.02754

[23] B. Hou, K. Qian, L. Li, Y. Shi, L. Tao, and J. Liu, "Mongodb nosql injection analysis and detection," in *Cyber Security and Cloud Computing (CSCloud), 2016 IEEE 3rd International Conference on*. IEEE, 2016, pp. 75–78.

[24] L. Okman, N. Gal-Oz, Y. Gonen, E. Gudes, and J. Abramov, "Security issues in nosql databases," in *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*. IEEE, 2011, pp. 541–547.

[25] D. A. L. C. B. Mariano Ceccato, Cu D. Nguyen, "Sofia: an automated security oracle for black-box testing of sql-injection vulnerabilities," in *ASE 2016 Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM New York, NY, USA, 2016, pp. 167–177.

[26] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks." in *USENIX Security Symposium*, vol. 98. San Antonio, TX, 1998, pp. 63–78.

[27] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," 2005.

[28] C. M. R. Haider, A. Iqbal, A. H. Rahman, and M. S. Rahman, "An ensemble learning based approach for impression fraud detection in mobile advertising," *Journal of Network and Computer Applications*, 2018.

[29] B. Amos, H. Turner, and J. White, "Applying machine learning classifiers to dynamic android malware detection at scale," in *2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC)*, 2013, pp. 1666–1671.

[30] J. Wang, Y. Xue, Y. Liu, and T. H. Tan, "Jsdc: A hybrid approach for javascript malware detection and classification," in *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*. ACM, 2015, pp. 109–120.

[31] Y.-T. Hou, Y. Chang, T. Chen, C.-S. Laih, and C.-M. Chen, "Malicious web content detection by machine learning," *Expert Systems with Applications*, vol. 37, no. 1, pp. 55 – 60, 2010.

[32] L. K. Shar, L. C. Briand, and H. B. K. Tan, "Web application vulnerability prediction using hybrid program analysis and machine learning," *IEEE Transactions on Dependable and Secure Computing*, vol. 12, no. 6, pp. 688–707, Nov 2015.

[33] M. K. Gupta, M. C. Govil, and G. Singh, "Predicting cross-site scripting (xss) security vulnerabilities in web applications," in *2015 12th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, July 2015, pp. 162–167.

[34] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 993–1006, Oct 2014.

[35] N. Leavitt, "Will nosql databases live up to their promise?" *Computer*, vol. 43, no. 2, 2010.

[36] H. Ma, T.-Y. Wu, M. Chen, R. Yang, and J.-S. Pan, "A parse tree-based nosql injection attacks detection mechanism," 2017.

[37] A. W. Moore and M. S. Lee, "Efficient algorithms for minimizing cross validation error," in *Eleventh International Conference on Machine Learning*. Morgan Kaufmann, 1994, pp. 190–198.

[38] J. Quinlan, *C4.5: Programs for Machine Learning*. Elsevier Science, 2014.

[39] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "Smote: synthetic minority over-sampling technique," *Journal of artificial intelligence research*, vol. 16, pp. 321–357, 2002.

[40] E. B. Wilson, "Probable inference, the law of succession, and statistical inference," *Journal of the American Statistical Association*, vol. 22, no. 158, pp. 209–212, 1927.