APPROVAL SHEET

Title of Thesis: Semantic Interpretation of Structured Log Files

Name of Candidate: Piyush Nimbalkar
MS Computer Science, 2015

Thesis and Abstract Approved: _____

Dr. Anupam Joshi
Professor
Department of Computer Science
and Electrical Engineering

Date Approved: _____

Curriculum Vitae

Name: Piyush Nimbalkar

Degree and Date to be Conferred: Master of Science, Summer 2015

Secondary Education:

- Sir Parashurambhau College, Pune, Maharashtra

Collegiate Institutions Attended:

- University of Maryland Baltimore County, MS Computer Science 2015
- Pune Institute of Computer Technology (PICT), Bachelor of Engineering 2011

Major: Computer Science

Professional Positions Held:

- Software Development Engineer Intern, Amazon, Inc. (06/03/2014 – 08/22/2014)
- Application Developer, ThoughtWorks, Inc. (08/03/2011 – 07/19/2013)

ABSTRACT

Title of Thesis: Semantic Interpretation of Structured Log Files

Piyush Nimbalkar, MS Computer Science, 2015

Thesis Directed by: Dr. Anupam Joshi, Professor
Department of Computer Science
and Electrical Engineering

Log files comprise a record of different events happening in various applications, operating systems and even in network devices. Originally they were used to record information for diagnostic and debugging purposes. Nowadays, logs are also used to track events which can be used in auditing and forensics in case of malicious activities or systems attacks. Various softwares like intrusion detection systems, webservers, anti-virus and anti-malware systems, firewalls and network devices generate logs with useful information, that can be used to protect against such system attacks. Analyzing log files can help in proactively avoiding attacks against the systems. While there are existing tools that do a good job when the format of log files is known, the challenge lies in cases where log files are from unknown devices and of unknown formats.

We propose a framework that takes any log file and automatically gives out a semantic interpretation as a set of RDF Linked Data triples. The framework splits a log file into columns using regular expression-based or dictionary-based classifiers. Leveraging and modifying our existing work on inferring the semantics of tables, we identify every column from a log file and map it to concepts either from a general purpose KB like DBpedia or domain specific ontologies such as IDS. We also identify relationships between various columns in such log files. Converting large and verbose log files into such semantic representations will help in better search, integration and rich reasoning over the data.

SEMANTIC INTERPRETATION OF STRUCTURED LOG FILES

by

Piyush Nimbalkar

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Master of Science
2015

Dedicated

to the memory of my father, Mohan Nimbalkar

&

to my mother, Alka Nimbalkar, without whose support I would not have made it to my Masters.

ACKNOWLEDGMENTS

First and foremost, I would like to acknowledge my advisor Dr. Anupam Joshi who has been a great mentor throughout my thesis. I am grateful for his much needed guidance and patience. I would also like to thank Dr. Varish Mulwad for helping and guiding me not just during my thesis, but also during my entire Masters. This thesis would also not have been possible without the constant help, regular feedback from all my colleagues working in the Ebiquity lab at University of Maryland, Baltimore County. Last but not the least, a would like to thank my family and all my friends who stood by me in both good and bad times during the past two years.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

The advancement of software has led to creation of huge amount of data. One such kind of data that is generated behind almost every system today, is logs. Every single thing we do on a software system results in triggering various pieces of software, and in turn creating several lines of logs. Log data is no longer just a tool for diagnostics and debugging software. It is increasingly used in cybersecurity, forensics, and for auditing purposes, due to the amount of vital information it holds. Recognizing the importance of logs, enterprises have started spending a lot of money to generate and store huge amount of log daily. Also, special emphasis is given to ensure creation of auditing information, so that it becomes easy to track user activities.

Log files give vital information about systems and the activities that are occurring. As such they present a valuable insight into the state of the system. Looking at the log file, not only do we understand the event, but can also trace its origin by audit trails and forensics. For instance, database logs can help trace the modifications on data, while web server log files speak a lot about the resources accessed from the web. On one hand, operating system logs help us figure out what is happening at the system level, on the other, firewall logs

help record malicious activities at the network level. Thus, log files form a vital tool in the cybersecurity. Using the information from log files we can pro-actively defend our systems against potential malicious activities and attacks.

Log file analysis is useful not only in defensive roles, but can also prove useful in the offensive approach to cybersecurity. Although offensive security techniques have ethical, legal, technical and practical considerations, it can be certainly useful in mitigating cyber threats [1]. There can be intrusive malware to track and observe the activities of targeted suspicious actors. This malware could be an intelligent log analyzer, trying to decode information from logs collected via various sources. Such a knowledge can help us secure our systems against zero day attacks.

1.1 Motivation

Huge amount of data generated in the form of logs is descriptive in nature because that was the primary purpose of logs. Now-a-days, people have started giving emphasis on structured log data. This eases the understanding and management of the logs. Today, a large amount of log files have a structure, which include the textual descriptions at the end. The structured log files also convey more information with less verbosity. Log formats are also being standardized, so that every vendor follows a format for certain set of tools or softwares. For instance, such recommendations provided in “Guide to Computer Security Log Management”, published by the National Institute of Standards and Technology (NIST) [2], indicate the growing importance of standardizing and analyzing log files.

It is important to know the structure of the log files to extract more and precise information. The difficulties in finding the exact structure of a log file are described by Kimball

and Merz [3]. They include multiple file formats, incomplete, inconsistent and irrelevant data, and dispersed information. There are even log files which have multi-line log entries.

There have been various attempts to tackle these problems with log files [4]. There are tools that parse specific log files and extract information from them [5]. However, this is not a scalable approach as it requires us to know the file format of a log file before analyzing it. Also, there are text processing techniques which extract information from log files as they would do from a normal chunk of text [6]. This information is used by Intrusion Detection and Prevention systems to detect malicious behavior. However, if we can leverage the structure of log files, we can clearly understand events in the log files and get more information from them. Most log files have distinct columns that do not seem to be related to each other and have a similar set of values. For instance, in *apache_access* log file, we have *content length* which is a numeric column and a *request path* which are file paths. If we know the meaning of the columns in the log file, we can easily say that the request on the *request path* gave a response of size equal to the *content length*. Often, there are semantic relations between various columns of log files. These may be documented somewhere but it is difficult for an autonomous system to understand the relations as a human would.

Apart from text processing systems, there are enterprise tools like Splunk [1], Sumologic [2], etc. These tools primarily focus on log management and analytics. Splunk does detect various fields in the log file but it does not always separate it out in proper columns. This happens particularly when it does not know the structure or source of the log file before hand. Thus, such enterprise tools have serious limitations in predicting the structure of unknown log files and further do not deal with finding semantic connections between

[1] http://www.splunk.com/
[2] https://www.sumologic.com/

various columns of the log file.

1.2 Contribution

To solve these problems we have built a framework that takes a log file as input and gives out its semantic interpretation as Linked Open Data expressed in RDF (Resource Data Framework) [7]. Our framework works for any random log file that has structured columns. We split the log file into identifiable columns and then predict classes for them. Using these columns we generate a list of candidate relations between those columns whose classes we have predicted, which are given out into a RDF file. We have extended the IDSOntology [3] to help identify the relationships between various column classes. Moreover, other ontologies can also be used. We test the system against log files with different structure and unknown sources. We also test the system against a dataset of randomly generated synthetic log files.

[3] http://ebiquity.umbc.edu/ontologies/cybersecurity/ids/v2.3/IDSOntology.owl

Chapter 2

OVERVIEW AND RELATED WORK

2.1 Overview

Log file analysis has been an interesting area in the cybersecurity domain. By analyzing log files we can help prevent our system from malicious activities. Predicting the structure and the semantic meaning for a log file with unknown source or format definitely possess an advantage. Every year, large number of tools and devices are added to the current software world. A security specialist or system administrator would not want to manually analyze every new log file and then feed it into their system.

The proposed framework works well with structured log files, where we can separate them into distinct columns and then predict the possible relations between those. The part of predicting the semantic relations between various columns in the log file is our novel contribution to the world of log file analysis. Finding relations definitely benefits in understanding semantics of the log files. Having internal relations can help make the log files less verbose, by preventing the need to explain every detail. As the size increases, the cost to maintain the huge amount of log data increases. That is why, we find log files

with concise columns. We see columns like IP addresses, timestamps, email addresses, urls etc. Most of the columns in the log files have inherent connections to each other. These are not visible by looking at the log file and are documented somewhere else. For example let us consider, *sendmail* log file, which is a tool for email routing. Table 2.1 shows a sample sendmail log file with selected columns. Looking at the log file we see timestamps, random server names and email addresses. If an automated system can figure out the relationship between these fields then we can say that user with email address in the third column is sending an email to the address in fourth column via server mentioned in the second column at timestamp in the first.

Table 2.1. Sample sendmail log file with selected columns

Mar 23 03:12:21	mail_server_1	from=<user1@sender.com>	rcpt=<shanon@umbc.edu>
Mar 23 03:12:21	mail_server_2	from=<user2@sender.com>	rcpt=<george@umbc.edu>
Mar 23 03:12:21	mail_server_1	from=<user3@sender.com>	rcpt=<david@umbc.edu>
Mar 23 03:12:21	mail_server_2	from=<user4@sender.com>	rcpt=<chris@umbc.edu>

The framework is also crucial in Security Information and Event Management (SIEM) systems, to detect anomalous behavior in the system. If the SIEM, is able to know the structure of any random log file in the infrastructure, it can autonomously detect threats. We can defend out systems against unknown attacks before hand. For instance, an apache access log files has columns like IP address, resource path and user-agent, out of the many. The IP

address is the address of the machine requesting the resource accessed by the resource path. User-agent is the software which can request the resource for the user of the IP address. This can be anything from a browser to a command line tool. Knowing about the log file, now we can have relations like 'Resource_Path *requestedUsing* User-agent' or 'IP_Address *requestedResource* Resource_Path'. With this knowledge SIEMs can detect threats like an IP address is accessing unauthorized resources or if it is accessing using suspicious user-agents. This becomes easy on knowing the semantic information about the log file.

The world of Semantic Web has huge amount of data and is still growing. There are knowledge bases or ontologies which already have many classes and properties that are found in the columns of log files. If this data is extended to support different kinds of data found in log files, we will have a huge collection of semantic log file data. This can be used to process and smartly analyze generic log files.

2.2 Background

The details of how the framework is implemented will be discussed in the Chapter 3. Before that we will briefly introduce some concepts and terminologies used in this thesis for better understanding.

2.2.1 Linked Data

Before knowing more about Linked Data, we will briefly talk about Semantic Web. Semantic Web is a way of linking or correlating data between entities that allows for rich interrelations of the data available across the world on Web. Most of the data we see on the Web is in the forms of HTML pages which are linked with each other through

hyperlinks. Though the pages are linked, there is no linking between the data within the Web pages. This makes it difficult for computer systems to make sense of the plain Web pages. Semantic Web technologies have encouraged people to create open data stores on the Web, add their own vocabularies and rules to handle this data.

To create a semantic web of data, it is important to make the huge amount of data available in a standardized format. Apart from the data, we also need to maintain the relationships among the data. The collection of this data which is related with each other is known as Linked Data. DBPedia [1] is one such example of a large dataset of Linked Data.

2.2.2 Resource Description Framework (RDF)

The Resource Description Framework (RDF) is a set of specifications, designed to model the Linked Data discussed in the previous section. It acts as a metadata model. Using certain syntax, rules and serialization formats, it helps in representing knowledge bases in a generic way which can be used by semantic tools and applications. RDF data internally forms a directed multi-graph. RDF expressions resemble the class diagrams or entity-relationship models. They are usually expressed in the form of *triples*. For instance, to represent the fact "London is the capital of United Kingdom", the RDF *triple* will have: the subject as 'London', the predicate would be 'is capital of' and the object as 'United Kingdom'.

[1] http://dbpedia.org/

2.2.3 Web Ontology Language (OWL)

OWL, the Web Ontology Language is designed by the W3C Web Ontology Working Group. It is set of formal language to represent the structure of knowledge bases also known as ontologies. Ontologies are a flexible way to describe structure of information from the Internet which accommodates heterogeneous data sources. OWL which is written in XML, has formal semantics and build upon the standards of Resource Description Framework (RDF).

2.2.4 Specialist Framework

The specialist framework was built by Puranik [8]. This is an approach to classify a given input column using a set of specialists. These specialists are basically experts to identify a particular class of columns. The framework runs the column through all the specialists and gives a ranked list for it. The framework was designed to classify tables found on the web and create knowledge repositories from them.

2.2.5 RDFlib

RDFLib [2] is a pure python package created to work with RDF and other semantic data formats. It provides a rich set of parsers and serializers for RDF/XML, N3, NTriples, N-Quads, Turtle, RDFa and Mircodata. Apart from the parsers, it provides a Graph interface which allows us to store the relational data in a graph, keeping the semantic structure of the data. RDFLib also provides SPARQL implementation. SPARQL (SPARQL Protocol and

[2]https://rdflib.readthedocs.org/

RDF Query Language) is a query language for RDF datastores.

RDFLib facilitates adding plugins for parsers, serializers, data stores and to handle query results. We are currently using the OWL ontology to detect the relationships between various classes. RDFLib supports the parsing of OWL ontologies using the OWL-RL package developed on top of it.

2.3 Related Work

Nascimento *et al.* [9] tried using ontologies to analyze security logs. They used the ModSecurity logs to create an ontology model to test the usage of ontologies in log analysis. They tested the system to prove that it was easier to interpret and find co-relations of events by modeling logs as ontologies. It was found that use of ontologies helped in classification of terms, inferences and relationships. It also proved useful in filters for searches using SPARQL on the ontology.

Joshi *et al.* [10] describe an automatic framework that generates and publishes a RDF linked data representation of cybersecurity concepts and vulnerability descriptions extracted from various vulnerability databases. This cybersecurity linked data collection was intended for vulnerability identification and to support mitigation efforts. The prime idea was to use the unstructured cybersecurity related data as linked data and leverage reasoning of security concepts, which can help detect and prevent zero day attacks.

Mulwad *et al.* [11] describe a framework to detect and extract information about attacks and vulnerabilities from Web text. They used Wikitology, a knowledge base derived from Wikipedia, to extract concepts related to vulnerabilities. On mapping these concepts to related concepts in DBPedia, they generated machine understandable relations. The de-

scribed framework was meant to extract information from any Web texts like chat rooms or social media feeds and is useful in detecting existing attacks as well as potential new attacks.

Splunk [12] is a log analysis and management tool developed to ease searching and diagnosing problems. It analyzes structured and unstructured log files and tries to identify the fields from the log files. It works well with the log files whose structure it already knows. For unknown sources of log files, it fails to identify the columns, but tries to spit out the set of fields from the log files. Splunk is an enterprise tool meant not just for log file analysis, but for also for searching, management, storage and visualization. It can help people to identify security issues in a faster and more affordable way.

Chapter 3

SYSTEM DESIGN

3.1 Architecture Overview

The architecture of the system comprises of several modules that process a given input log file, and an RDF (Resource Description Framework) file describing columns and relations is given as the output. The modular nature of the architecture allows us to add more specialists to the existing framework. Figure 3.1 shows the block diagram of the system architecture. It begins with *Tabulate* module, where the input log file is parsed to detect its structure. Once the structure is detected the structured log file is separated into columns. These columns are then passed to the *Decode* stage. In the *Decode* module the columns are checked against various classifiers to find a match. After checking against different classifiers, the columns are given a score against every classifier in the system. The scores form a ranked list of probable classes for the columns in the log file. This ranked list is used in the *Relationship Generator* to identify the relations between various columns in the log file. Once the columns are identified and the relationships have been detected, the *Generate RDF* module will produce a set of RDF triples to represent the inferred semantics. The four major modules have been described in detail in rest of the

section.

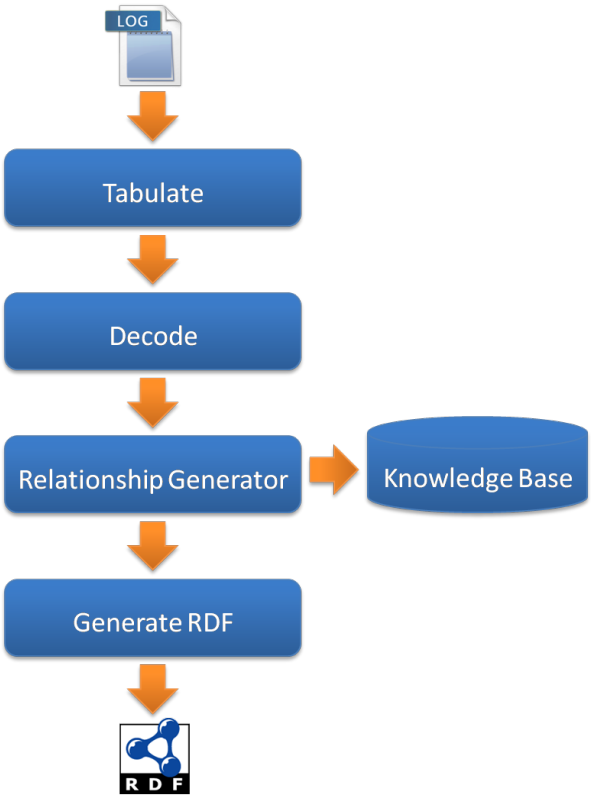

FIG. 3.1. System Architecture

3.1.1 Tabulate

The *Tabulate* module takes the structured log file as input and gives out a table with possible set of columns to the *Decode* module for classification. The input log file can have any structure and the module will try to detect columns in it. The log file undergoes multiple iterations for the following reasons:

- To split the log file based on delimiters

- To detect consistent number of columns throughout the log file
- To detect and separate sub-columns

In the first iteration of the *Tabulate* module, we split the log files using generic delimiters like space, square braces, round braces, single and double quotes. For braces and quotes we consider only those that are in pairs. Every row is thus separated in varied number of columns that may or may not be consistent.

Although most of the lines in a structured log file have fixed number of columns, it is not necessary that after separating on the above delimiters we will get a table with same number of columns. Also, some log entries can be outliers due to consistency in the way the logs are generated. But in most of the cases, there is a textual description at the end of the log entry which does not have a fixed number of words. If the description is not enclosed in a brace or quote, then it will be divided into several different columns. Considering this fact, we dynamically decide the number of columns that are to be extracted. For this, we put a threshold on the percentage of rows that will be considered in the system. We select rows starting with the ones that have the highest number of columns. We then select the next set of rows that have the second highest number of columns. We continue this process till we reach the threshold number of rows. At this moment we record the number of columns of the current set of rows. This is the number of columns that we choose to output from the system. All the rows having columns less than the selected number of columns are discarded. We keep the threshold pretty high (above 70 %), so as to collect more number of rows. We also tried keeping the rows that have columns less than the selected number of columns and utilize them in the *Decode* module. It was observed if a large percent of log entries have consistent columns, then the remaining smaller percent usually forms abnormal entries. Thus they create noise in the *Decode* module when trying to detect the

class of the column.

In the third iteration, we loop through the columns that are already separated in the previous iterations. In this iteration, we check the columns to detect sub-columns. We use the same set of delimiters to split the columns further. If all the elements in a particular columns are separable, then we split the column into multiple sub-columns. Before separation, we run that complete column through the *Decode* module to check if the whole column matches any of the known classes. If there is a match found, then there is no need of splitting the column, as we can use the column as a whole.

3.1.2 Decode

The *Decode* module does the work of assigning classes to the columns given by the *Tabulate* module. For the classification of columns we created an extension of the specialists approach implemented by Puranik [8]. Puranik defines a specialist as "an entity that has expertise in the domain for which it is a specialist". The mentioned specialists system was created for generic table data found on the Internet like SSN, phone numbers, addresses, etc. We extend this system to classify specific fields that are usually found in log files viz., timestamps, IP addresses, URLs, etc.

Every column is passed through the specialists in the system. All the specialists give their score to the column based on how well the column matches the class of the specialist. We normalize this score to form a ranked list to predict the class of the system. If there is no match to any of the specialists then the columns is classified as 'NA' (No Annotation). The normalized scores can also be considered as the confidence with which the specialists classifies a particular column into its respective class.

Specialists can be of different types like regular expression based specialists, dictionary based specialists and classifier based specialists. Currently we are using only regular expression based and dictionary based specialists. But we can further extend it to use classifier based specialists as well. Following is the list of specialists present in the current system:

1. **Timestamp Specialist**

   The Timestamp specialist is a regular expression based specialist, which handles a lot of timestamp formats which are used in various systems for logging. This could also be implemented as a classifier based specialist using the Date specialist implemented by Puranik [8].

2. **IP Address Specialist**

   The IP Address specialist is a regular expression based specialist. It checks for valid IP address formats and also makes sure that they are in a valid range.

3. **Port Number Specialist**

   The Port number specialist is a regular expression based specialist. It checks for valid port numbers, that is, those that are in the range of 0 to 65535.

4. **URL Specialist**

   The URL specialist is a regular expression based specialist, which looks for various URL formats viz., *HTTP*, *HTTPS*, *FTP*, *FTPS*. It also detects URL having basic authentication.

5. **Filepath Specialist**

   The Filepath specialist is also a regular expression based specialist. It looks for filepaths irrespective of the underlying system.

6. **Email Specialist**

   The Email specialist is a regular expression based specialist, that looks for valid email formats in the given column. This follows the Internet standards to detect a valid email.

7. **HTTP Status Code Specialist**

   The HTTP Status Code specialist is dictionary based specialist. It looks for the HTTP status code values provided by the W3 standards. These HTTP status codes are integer values.

8. **HTTP Method Specialist**

   The HTTP Method specialist is a dictionary based specialist. It looks for the known HTTP verbs / methods like *GET*, *PUT*, *POST*, *DELETE*, etc.

9. **HTTP Protocol Version Specialist**

   The HTTP Protocol Version specialist is a dictionary based specialist, which looks for the known HTTP protocol versions. The HTTP protocol versions are in a string format.

10. **Log Level Specialist**

    The Log level specialist is a dictionary based specialist, which looks for the generally recommended log levels keywords. The log levels are in string format and contains log levels like *Info*, *Debug*, *Error*, *Warn*, etc.

We can easily extend the system to add more specialist to detect other fields in the log files. These specialists can be regular expression based or dictionary based specialists. For instance, if a system administrator decides to have a specialist for server names, then they can have a dictionary based specialist. They will keep a dictionary with all the server

names. In future, if our system comes across a log file with server name as the column, it will detect and classify the column precisely.

3.1.3 Relationship Generator

The *Decode* module gives out a ranked list of class labels for every column in the table, which is further used to predict the relations between various pairs of columns. We consider the topmost ranked prediction as the class label for that particular column. Considering a pair of columns at a time, we search the *Knowledge Base* to find relationship between them. Every column is checked against all other predicted columns in the table. For now, we do not consider the columns that are not annotated by the *Decode* module.

In the current experiments we have used the IDS Ontology[1] as our knowledge base. This OWL ontology which was created at University of Maryland, Baltimore County, was developed for Intrusions Detection Systems. We extended this ontology to add certain classes and relations that were found in log files that we collected.

We use RDFLib, a python library to parse the ontology so that it becomes easy to search it. Using the SPARQL plugin in the RDFLib, we search for classes that act as domain and/or range for properties. For instance, when we are given a pair of columns, we find all the properties where one class is domain and the other is range or vice versa. This way we extensively search for all combinations of columns in the ontology.

We give multiple relationships if there are multiple finds for a particular pair of columns. In case of log files, we do not have the instances of columns in knowledge base. If there were instances present, then using an approach similar to that developed by

[1] http://ebiquity.umbc.edu/ontologies/cybersecurity/ids/v2.3/IDSOntology.owl

Mulwad [13], we could have ranked the relations.

3.1.4 Generate RDF

The *Relationship Generator* module gives a list of triples: subject, object and the relation. We serialize these triples using the RDFLib library to create an RDF file. This RDF file with triples can be used by systems like SIEMs or Intrusion Detection Systems to feed into their system. Using this information these systems will now know the possible columns and relations in any given log file and will not need human intervention to interpret the log files.

Chapter 4

EXPERIMENTS AND RESULTS

In this chapter we discuss the datasets used and the experimental setup to test the column separation, column annotations and relation predictions. We also evaluate the framework as we explain the results from the various experiments performed on the datasets.

4.1 Data Setup

We use different sets of log files to test the framework. Primarily, we use two datasets viz., original log files and synthetically generated log files. Table 4.1 gives a general statistic about the log files in each dataset, total columns, average columns and relations per table in all the log files of the dataset.

The Original dataset contains actual log files as obtained from various tools and services running in an operating system. This includes *Linux* system services like syslog, kern, auth, etc. We also have log files from commonly used services like apache2, mongodb, sendmail, printers, etc.

Table 4.1. Number of log files, total columns present, average columns and relations per table in each dataset

Dataset	*Tables*	*Columns*	*Average Columns/ Table*	*Average Relations/ Table*
Original	11	78	7	3
Synthetic	20	150	8	5

The Synthetic dataset has a set of artificially generated log files using some known columns. These columns are generally seen in log files like those present in the Original dataset. The synthetic log files were generated to test the framework against log files with unknown source and format.

To create these synthetic log files, we randomized the whole process. The number of columns in every log file was randomly chosen. Later, we arbitrarily selected the type of columns from the manually prepared superset of columns. While generating the columns we probabilistically added different column values to the selected columns. This noise was added to simulate unpredictable nature of log files. Also, all the values added in the columns for a particular type were generated at random to ensure variation in length and other properties of the column.

4.2 Tabulation

In this experiment we tested the precision with which the log files are converted into tables with proper set of columns. The *Tabulate* module of the framework, splits the log file to convert it into a table. This is a simple test where we have log files with separable columns. We feed these log files to the *Tabulate* module and get an output table with certain set of columns. We try and match all the output columns to the expected sample. Thus, we not only check the number of columns separated we also check the exact boundaries of separation.

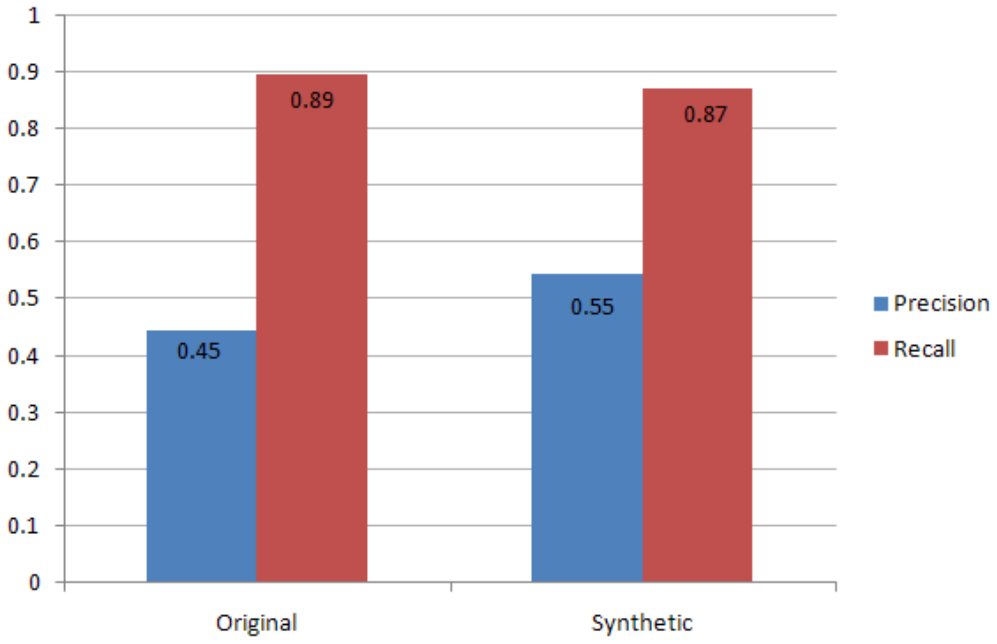

FIG. 4.1. Precision and recall for column separation

As discussed in previous chapter, we separate the log files using standard delimiters. Due to this, columns with multiple words but no surrounding characters, get separated as multiple columns in our framework. It is observed, that in most of the cases such descriptive

columns with multiple words are found at the tail of the log entry. Thus, most of the initial columns are easily separated. Due to the said fact about log files, our framework can give out more columns than the expected number of columns. We get a high recall as we do not expect those columns to be separated. While the recall is high, we observe a low precision. Due to the extraneous columns, which are not there in the original log files, we get a low precision compared to the recall. In Figure 4.1 we see that for both the datasets, we get a high recall and low precision.

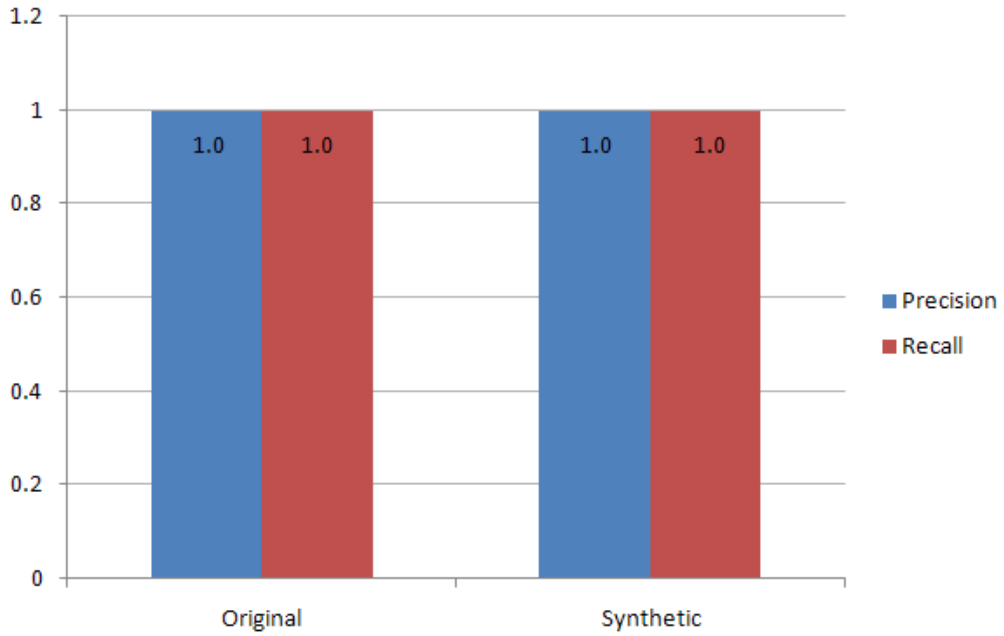

FIG. 4.2. Precision and recall for column separation without trailing textual data

We repeat this experiment with the same log files, but now we strip out the trailing textual part. In real life, we can find tools which can clean the log files and in this case strip the unwanted text. Figure 4.2 shows the precision and recall with the modified log files. Now, we see that both precision and recall are high for the datasets. The high precision in the later case indicates that most of the log files have structured data at the beginning of the

log entries and is followed by a more verbose message field.

Although the recall of both the datasets is almost similar, we see that it is slightly better in the Original dataset (Figure 4.1). Some of the log files like *apache_access* have the textual part embedded in quotes. This makes it easier for the *Tabulate* module to separate the column as a whole. While in the Synthetic dataset, we have appended the text without enclosing quotes. Hence, the recall is better in the Original dataset compared to the Synthetic dataset.

4.3 Column Annotation

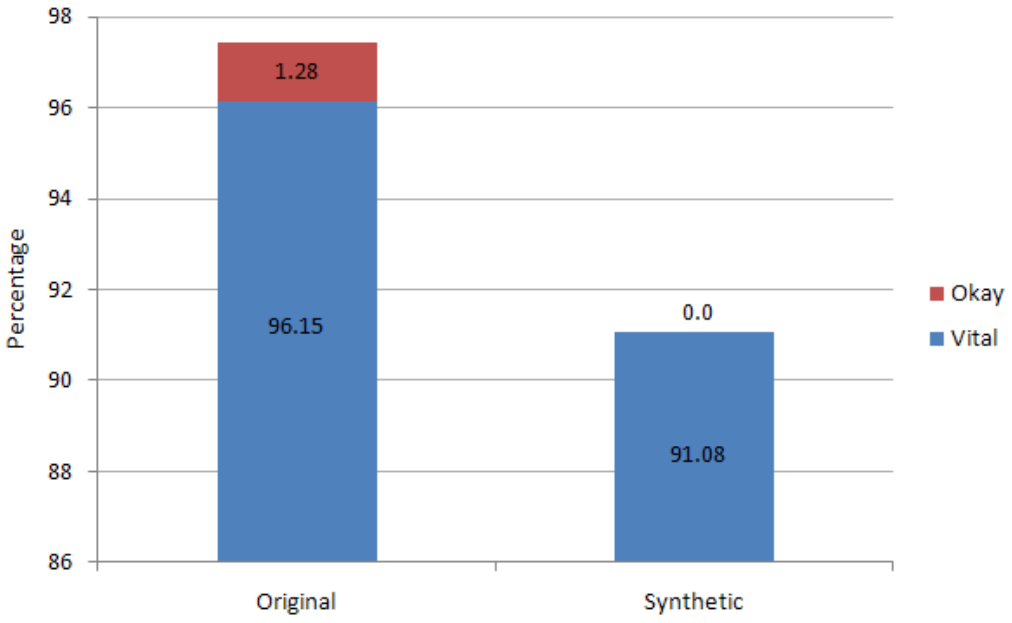

FIG. 4.3. Percentage of *vital* and *okay* classes at rank 1

In the *Decode* module we annotate the columns present in the table generated by the *Tabulate* module. We gave columns from the log files to manual annotators. The

annotations from the manual annotators form the ground truth for this module. The manual annotators mark each class as *vital*, *okay* or *incorrect*, for every column in the table. This is analogous to the strategy adopted by Venetis *et al*. [14]. For instance, an annotator could mark, the *URL* class as *vital*, *FilePath* as *okay* and *HTTPMethod* as *incorrect*, for the column *URL*. Each column may have multiple *vital*, *okay* or *incorrect* labels. This task of manual annotation was given to three Graduate students in Computer Science at the University of Maryland, Baltimore County.

The human annotators were given ten classes for every column. If they could not match the column to any of the known ten classes, they would mark it as *No Annotation* (N.A.). Even in our framework we marked N.A. for the columns that were not recognizable by the system. We considered the column annotation to be accurate if both the human annotator and the framework gave *No Annotation*.

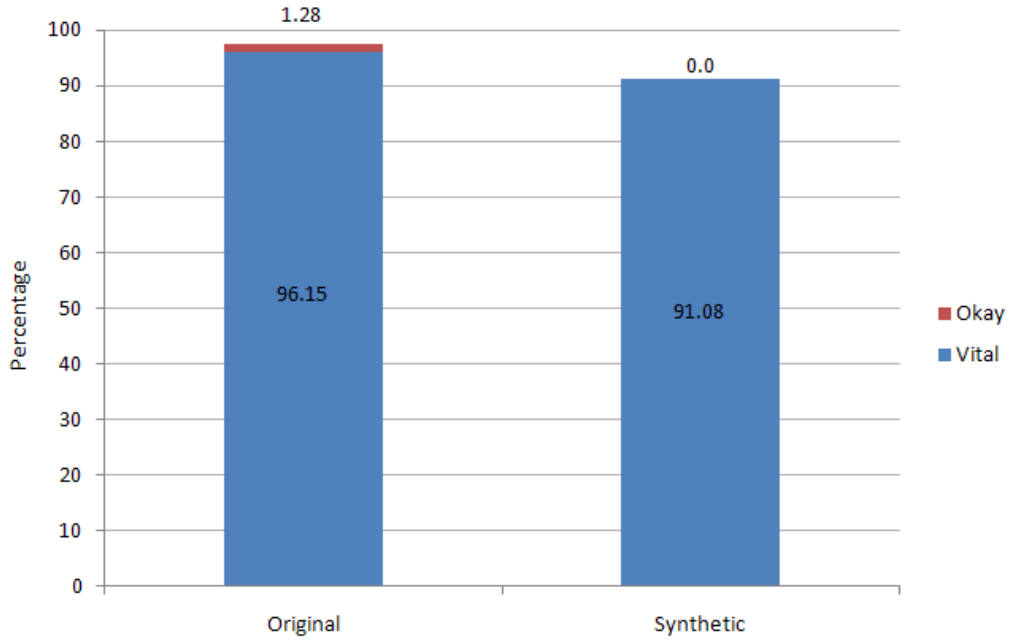

FIG. 4.4. Percentage of *vital* and *okay* classes at rank 1

In the *Decode* module, we get a ranked list of classes for every column in the table. Figure 4.3, shows the percentage distribution of *vital* and *okay* classes found at rank 1. This is an aggregate of all the columns in the datasets. In the Original dataset, 96.15% of the top ranked class labels are marked as *vital* and 1.28% are marked as *okay*. While in the Synthetic dataset, 91.08% of the top ranked class labels are marked as *vital*. There are no top ranked class labels marked as *okay*. Thus, the distribution is highly inclined towards *vital* for columns in both the datasets.

Figure 4.4 shows the zoomed out version of Figure 4.3. It is observed that there is not much difference in the percentage of correctly predicted classes for both datasets. Overall we see a good percentage of relevant class labels, i.e., either *vital* or *okay*, are found at rank 1. Thus, the system has a good accuracy for column prediction as most of the predicted class labels are found at the topmost rank.

Figure 4.5 shows the distribution of relevant class labels at different ranks in the system. As concluded from Figure 4.3, we can see that most of the relevant class labels are found at rank 1. Very few relevant class labels are found at ranks 2 and 3. It is observed that none of the lower ranks from 4 have the relevant class labels. Due to the almost structured nature of the log file, we get a lot of rows with appropriate column values, leading to proper annotation of the column.

Also, the relevant class labels for both the datasets are comparable. We see that percentage of relevant class labels at rank 1 are slightly less for Synthetic dataset. This is because columns like *URL* are sometimes also classified as *FilePath*. This causes the *URL* class label to be at rank 2. There are more such columns in the Synthetic dataset due to our probabilistic approach of log file generation.

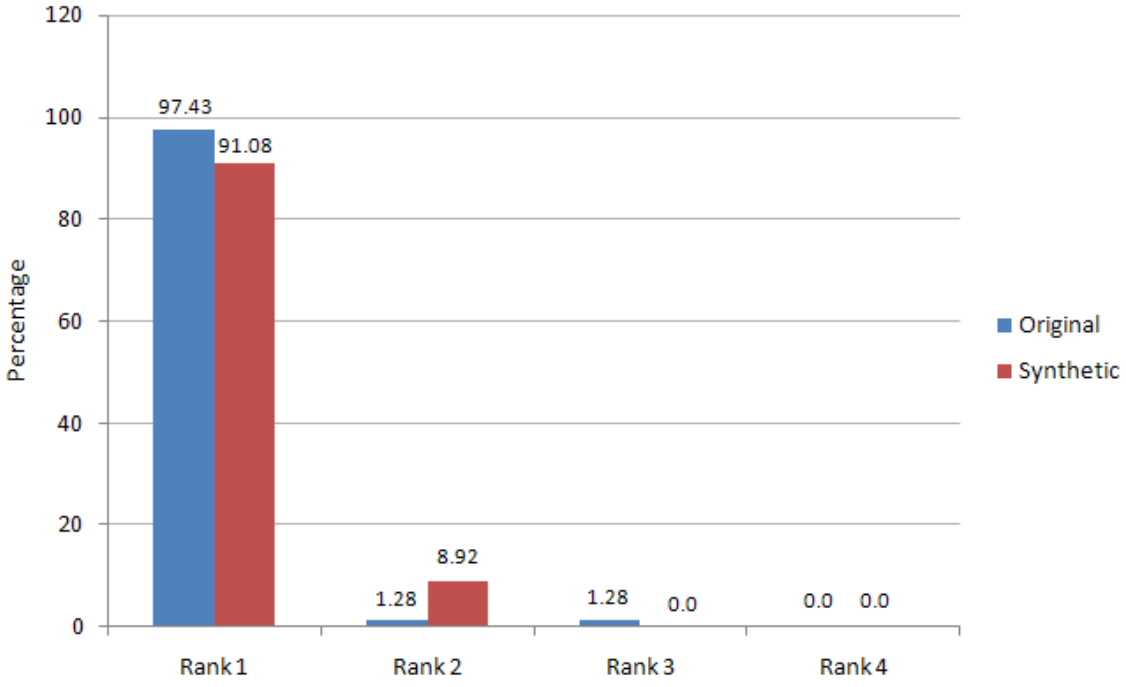

FIG. 4.5. Percentage of relevant classes at ranks 1 to 4

4.4 Relation Annotation

Similar to the *Decode* module we annotate the relations between various columns present in the table. These relations are predicted using the IDS Ontology. We gave tabulated log files to human annotators, and they were asked to annotate the relation for a set of column pairs in them. For every pair of columns they were given a set a relation labels. The annotators marked each relation label as *vital*, *okay* or *incorrect*, for every pair of columns given to them. This is again, similar to the strategy used by Venetis *et al.* [14]. The annotators were allowed to mark multiple labels as *vital*, *okay* or *incorrect*. They were given a choice to mark a relation as No Annotation (N.A.) if they did not find suitable relation from the given set of relation labels.

Figure 4.6 shows the precision and recall calculated for the relation prediction module. It is observed that the recall for both datasets is very high. The relations that are predicted by our framework come out to be accurate, and hence we get a very high recall. While the precision of the Original dataset is very high, that of the Synthetic dataset comes out quite less comparatively. Most of the relationships we have in the ontology are added by observing the actual log files present in the Original dataset. That is why, our framework's predictions match quite well with the actual relations. On the other hand, in Synthetic dataset, there are a lot of false positives. This means we are predicting relations that do not exist in the human annotated source. Hence, we see a dip in the precision of relation prediction for Synthetic dataset.

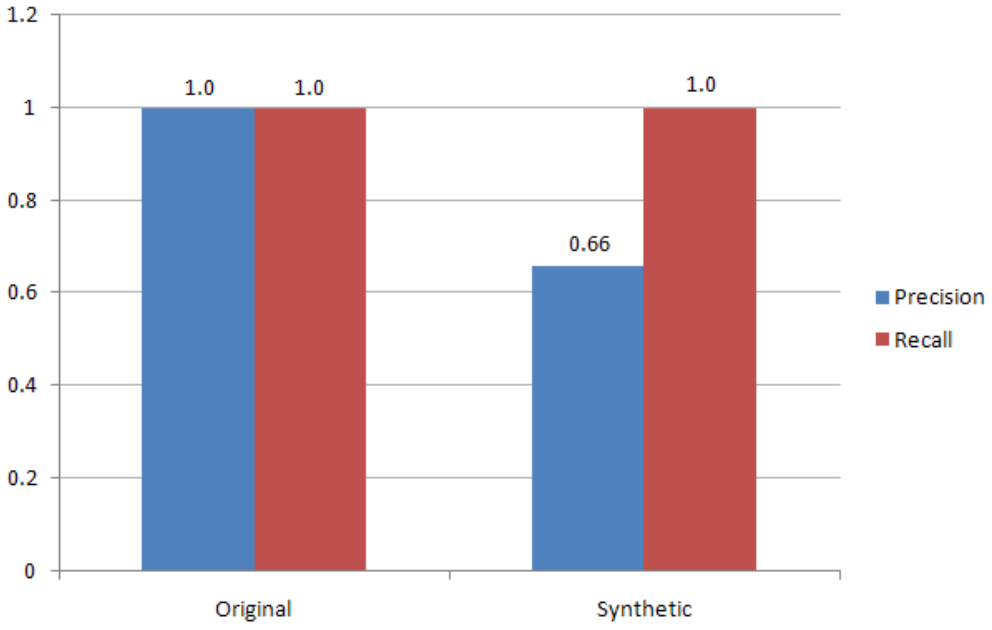

FIG. 4.6. Precision and recall for relation annotations

Figure 4.7 shows the percentage distribution of predicted relations that are marked as *vital* and *okay*. Almost all of the relations predicted are annotated as *vital* by the human

annotators. In the Original dataset all the predicted relations are annotated as *vital*. While in Synthetic dataset, a small percentage i.e. 6.67% of predicted relations are marked as *okay* and rest are *vital*. There are pairs of column like *EmailAddress* and *Timestamp* that have multiple relations like *sentAt* and *receivedAt* amongst them. This causes some of them to be marked as *okay* by human annotators. Such pairs occur more frequently in the Synthetic dataset and hence it has higher percentage of *okay* labels compared to the Original dataset. Figure 4.8 is a zoomed out version of Figure 4.7. It clearly shows that there is not much difference in the distribution of *vital* and *okay* labels for both datasets. Also, most of predicted relations are marked as *vital*.

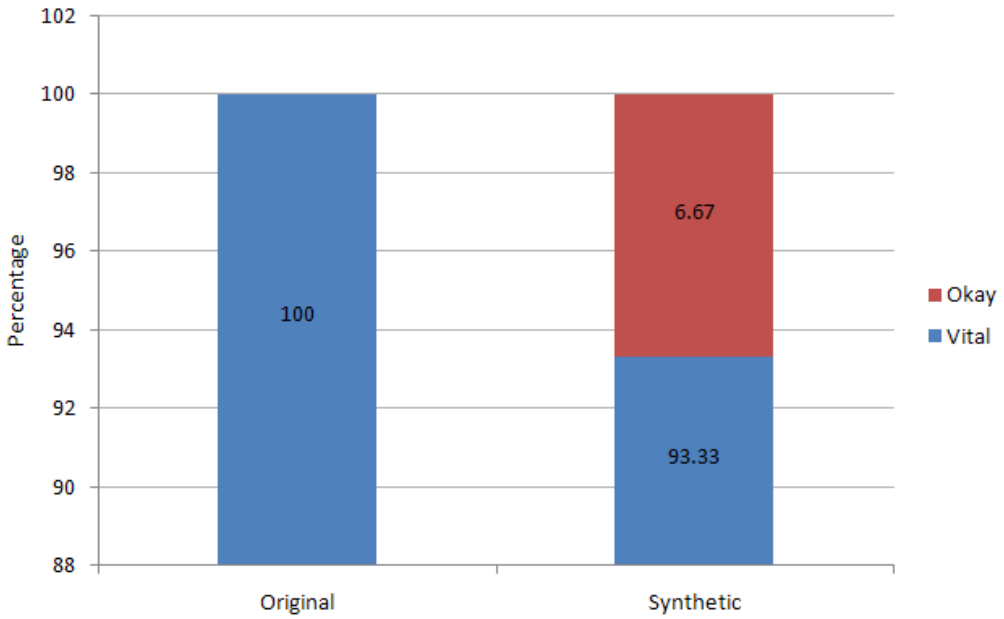

FIG. 4.7. Percentage of *vital* and *okay* relation labels

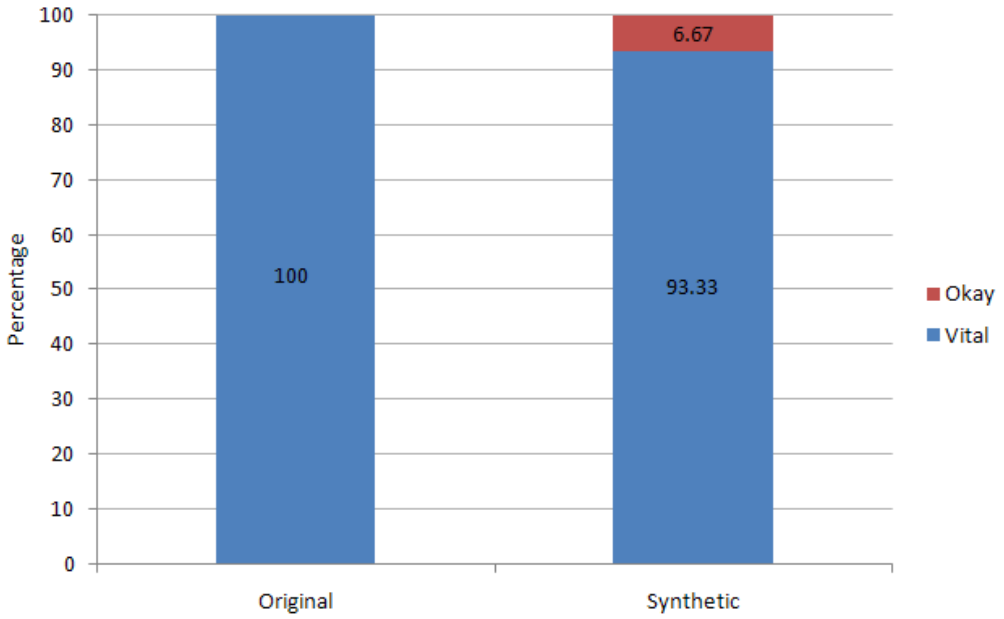

FIG. 4.8. Percentage of *vital* and *okay* relation labels

Chapter 5

CONCLUSION

In this thesis, we developed a framework which takes any log file with unknown source and format, and automatically gives its semantic interpretation in the form of RDF Linked Data. We were able to split the log files with a very good recall, which means we got most of the columns that we were expecting. Also, we effectively predicted the class labels of the columns in the log file. We can extend it to predict more classes of columns, by adding different specialists for every new column found in other log files. For the relation annotations, we were able to predict all possible relationships between the pairs of columns, with a high accuracy. The system performed well for actual log files as well as synthetically generated log files to simulate unknown source.

If we extend the ontology to have more classes and relations between the columns, then we can get a good semantic interpretation of any log file. This information can be used by tools to understand the log files more intelligently and help prevent our systems against unknown threats and attacks.

REFERENCES

[1] "The offensive approach to cyber security in government and private industry," http://resources.infosecinstitute.com/the-offensive-approach-to-cyber-security-in-government-and-private-industry, accessed: 2015-07-01.

[2] K. Kent and M. Souppaya, *Guide to Computer Security Log Management: Recommendations of the National Institute of Standards and Technology*. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, 2006.

[3] R. Kimball and R. Merz, *The Data Webhouse Toolkit*. New York: John Wiley and Sons, Inc, Wiley Computer Publishing, 2000.

[4] G. Giuseppini and M. Burnett, *Microsoft Log Parser Toolkit: A complete toolkit for Microsoft's undocumented log analysis tool*. Syngress, 2005.

[5] "Weblog expert," http://www.weblogexpert.com/info/ApacheLogs.htm, accessed: 2015-06-20.

[6] H. Saneifar, S. Bonniol, A. Laurent, P. Poncelet, and M. Roche, "Terminology extraction from log files," in *Database and Expert Systems Applications*. Springer, 2009, pp. 769–776.

[7] D. Brickley and R. V. Guha, "Rdf vocabulary description language 1.0: Rdf schema," 2004.

[8] N. Puranik, "A specialist approach for the classification of column data," Master's thesis, University of Maryland, Baltimore County, 2012.

[9] C. H. do Nascimento, R. E. Assad, B. F. Lóscio, and S. R. L. Meira, "Ontolog: A security log analyses tool using web semantic and ontology," *Web Application Security*, pp. 1–12.

[10] A. Joshi, R. Lal, and T. Finin, "Extracting cybersecurity related linked data from text," in *Semantic Computing (ICSC), 2013 IEEE Seventh International Conference on*. IEEE, 2013, pp. 252–259.

[11] V. Mulwad, W. Li, A. Joshi, T. Finin, and K. Viswanathan, "Extracting information about security vulnerabilities from web text," in *Web Intelligence and Intelligent Agent Technology (WI-IAT), 2011 IEEE/WIC/ACM International Conference on*, vol. 3. IEEE, 2011, pp. 257–260.

[12] "Splunk," http://www.splunk.com, accessed: 2015-07-01.

[13] V. V. Mulwad, "Tabel–a domain independent and extensible framework for inferring the semantics of tables," Ph.D. dissertation, University of Maryland, Baltimore County, 2015.

[14] P. Venetis, A. Halevy, J. Madhavan, M. Paşca, W. Shen, F. Wu, G. Miao, and C. Wu, "Recovering semantics of tables on the web," *Proceedings of the VLDB Endowment*, vol. 4, no. 9, pp. 528–538, 2011.

[15] J. Valdman, "Log file analysis," *Department of Computer Science and Engineering (FAV UWB)., Tech. Rep. DCSE/TR-2001-04*, 2001.

[16] J. H. Andrews, “Testing using log file analysis: tools, methods, and issues,” in *Automated Software Engineering, 1998. Proceedings. 13th IEEE International Conference on*. IEEE, 1998, pp. 157–166.

[17] T. Berners-Lee, J. Hendler, O. Lassila *et al.*, “The semantic web,” *Scientific american*, vol. 284, no. 5, pp. 28–37, 2001.