Ontology-Driven Data Semantics Discovery for Cyber-Security

Marcello Balduccini, Sarah Kushner, and Jacquelin Speck

College of Computing and Informatics
Drexel University
{mbalduccini, sak388, jspeck}@drexel.edu

Abstract. We present an architecture for data semantics discovery capable of extracting semantically-rich content from human-readable files without prior specification of the file format. The architecture, based on work at the intersection of knowledge representation and machine learning, includes machine learning modules for automatic file format identification, tokenization, and entity identification. The process is driven by an ontology of domain-specific concepts. The ontology also provides an abstraction layer for querying the extracted data. We provide a general description of the architecture as well as details of the current implementation. Although the architecture can be applied in a variety of domains, we focus on cyber-forensics applications, aiming to allow one to parse data sources, such as log files, for which there are no readily-available parsing and analysis tools, and to aggregate and query data from multiple, diverse systems across large networks. The key contributions of our work are: the development of an architecture that constitutes a substantial step toward solving a highly-practical open problem; the creation of one of the first comprehensive ontologies of cyber assets; the development and demonstration of an innovative, non-trivial combination of declarative knowledge specification and machine learning.
Keywords: data semantics discovery; ontologies; machine learning; cyber-security.

1 Introduction

An *ad hoc* data source is a data source for which parsing and analysis tools are not readily available [6]. Even well-documented, established file formats can evolve over time or change with various configuration settings, effectively becoming ad hoc to users who have not followed the changes. Ad hoc data sources present unique challenges for information technologists, cyber-security analysts, and other professionals who must parse and interpret such data for diagnostic or forensics purposes.

We attempt to address the challenges associated with ad hoc file formats through development of an data semantics discovery architecture for extracting semantically-rich content from human-readable files without prior specification of the file format. The proposed system includes modules for automatic file format identification, tokenization, entity identification, and storage of extracted

records and entities. Using a process driven by an ontology of domain-specific concepts, these components interact to parse data from an input file by identifying file format, records and entities within them, and by associating the extracted content with concepts from the ontology. Once data is extracted and stored, the ontology also provides an essential abstraction layer for querying the extracted data, with queries that can span across multiple file systems, file formats and levels of abstraction. In the prototype implementation presented in this paper, the ontology is tailored to cyber-security applications.

Searching for signs of a cyber attack in log files is one practical use for the proposed architecture. Time constraints and lack of documentation can make it difficult to find or create parsers for every log file format encountered, and the magnitude of those challenges increases when dealing with large networks of independent file systems. Security analysts must not only be aware of every type of log available on every network node, but be able to correlate information from multiple sources and reveal important underlying relationships between them. As a motivating example, consider a scenario in which a cyber-security analyst is notified of a new kind of cyber attack following this pattern:

1. A malicious e-mail with an attachment is received somewhere on the network. The sender's e-mail address varies, but it always ends in a ".net" suffix.
2. The recipient of the e-mail opens the attachment, unaware that it is a virus.
3. The virus establishes a DNS (Domain Name Server) tunnel[1] towards a server with the domain name "cyberattacks.com"

In this scenario, an analyst wishes to investigate whether this attack occurred somewhere on his or her network. However, the network includes many nodes, each with their own unique configuration, services, and corresponding log files (see Figure 1). The information is stored using different formats depending on the

```
client 192.168.157.1#5544: query: maliciousserver.com IN AXFR +T
  (192.168.157.129)
```

```
11/21/2013 2:15:59 PM 0A30 PACKET 00000085FD4B0610 UDP Snd
  199.7.91.13 cf1b Q [0000 NOERROR] A (3)www(6)maliciousserver(3)com(0)
11/21/2013 2:16:01 PM 00D4 PACKET 00000085FD4B2320 UDP Snd
  193.0.14.129 aba7 Q [0000 NOERROR] AAAA (3)www(6)maliciousserver(3)com(0)
```

Fig. 1. Sample DNS query records: bind format (top) vs MS DNS format (bottom)

node's specific configuration and softwares used, and understanding the meaning of a log entry requires knowledge that is not explicitly stated in the entry itself (e.g., string "+T" in Figure 1). To make things worse, in realistic circumstances, the analyst often has incomplete knowledge about the attack. In our case, for

[1] http://beta.ivc.no/wiki/index.php/DNS_Tunneling.

instance, the full address of the malicious DNS server and the e-mail address from which the virus originates are both unknown. Although fictitious, this scenario captures many challenges analysts are faced with in actual situations. In particular, the large amounts of data and the disparate, hardly predictable ways in which it may have been encoded make manual browsing of the files unfeasible. Additionally, traditional text-based search, which is relied upon by most state-of-the-art cyber-forensics tools, is also not advisable, as it typically leads to many irrelevant results and forces analysts to a time-consuming and error-prone manual post-processing phase. For example, searching for strings or email addresses (e.g., using regular expressions) with a ".net" suffix across all of the files will likely return matches that have nothing to do with emails received by a mail server, such as records from authentication logs. Furthermore, use of string matching does not allow an analyst to specify additional constraints, such as checking whether other logs indicate that the recipient's computer may have initiated a DNS tunnel to a certain family of servers..

Using our proposed architecture, the analyst can import log files from across the network into a unified knowledge base. The architecture includes modules capable of parsing all log files, regardless of configuration-dependent format variations. Finally, the analyst can ask queries that specify the *types* of information they wish to find, while the system automatically identifies the correct sources and content. This enables searching for signs of the cyber attack using high-level queries that capture the entire attack, rather than having to piece by hand the possible evidence of the individual stages.

The key contributions of our work are: the development of an architecture that constitutes a substantial step toward solving an open problem of high practical importance; the creation of one of the first comprehensive ontologies of concepts related to cyber assets; the development and demonstration of an innovative, non-trivial combination of declarative knowledge specification and machine learning techniques.

The remaining sections of this paper are organized as follows. Background on existing solutions for the problems addressed by the architecture are described in Section 2. Section 3 provides details of each component of the architecture. An experimental evaluation of performance of the machine learning techniques user by the architecture is presented in Section 4. Section 5 concludes the paper and discusses possible directions of future work.

2 Related Work

To the best of our knowledge, our data semantics discovery architecture is the first of its kind. It is worth pointing out that the problem being solved here is substantially different from Natural Language Processing (NLP) and from traditional Information Extraction (IE). The data sources considered here typically lack the grammatical structure considered by NLP and IE. Furthermore, differently from NLP and IE, the meaning of a record frequently depends on the file that contains it – e.g., line "03/08/2015 10.0.0.1" describe very different events

depending on whether it is found in a web server log file or in a DNS server log file. For the most part, earlier and ongoing research has studied sub-problems addressed by our architecture.

The problem of describing knowledge related to cyber-security scenarios is the object of various proposed specifications, such as STIX, CybOX, MAEC.[2] However, none of them provides a comprehensive and hierarchical description of the software and hardware components of a system, covering operating system objects and events.

Aggregating data from multiple file systems can help network administrators detect network problems or diagnose potential causes of earlier problems. Varying file formats and data schemas can complicate these tasks. Doan, et. al. present Learning Source Descriptions (LSD), a system for reconciling schemas from disparate data sources using machine learning [5]. LSD learns semantic mappings between multiple XML data sources, employing and extending established machine learning techniques. LSD incorporates user feedback to improve the accuracy of the mappings.

Splunk is a tool for aggregating massive heterogeneous datasets of log file text into a semistructured time series database [3]. It claims to accept logs in "any" format, and allows full text searches across various data sources via its own query language. The decision to aggregate data into a time series database was motivated in part by the fact that time stamps are one of the only common fields among many different types of log data, and contain essential information for many types of analysis (including cyber-forensics). Splunk exploits this time series organization during searches, operating on only the time slices that intersect the query target time. The Splunk query language supports a wide range of complex functionality, including data mining techniques such as clustering, anomaly detection, and prediction.

The presence of log file formats unfamiliar to network analysts often complicates their diagnosis of system failures and vulnerabilities. SherLog is a diagnostic tool capable of reverse-engineering log file formats. However, SherLog is limited to single file systems and only applies to log files produced by specific, known executable programs [15].

Tupni, another tool for reverse-engineering both protocol and file formats, expands beyond simple data types, extracting record types, record sequences, and input constraints [4]. However, Tupni requires both a sample file and an application capable of parsing the file as input. The tool therefore can not support ad hoc data sources, which have no readily available parsing tools. Splunk, an aggregation tool discussed above, supports ad hoc formats in the sense that users may configure arbitrary input types. However, it does not automatically learn how to parse these input types. While Splunk does not require users to specify a schema for the data to be indexed, users must specify fields and values to extract. It includes tools that guide the user through creating regular expressions to extract fields and values for each incoming time-delineated event.

[2] https://stix.mitre.org/.

Tokenization and entity extraction form ad hoc data sources is one of the core problems related to data semantic discovery. In-depth studies of the log analysis process have found that non-technical users increasingly need data from log files, but code development knowledge is a beneficial or even necessary prerequisite to log file understanding [1, 12]. A lack of documentation can also create difficulties for technical users, who often have to sort through program code in order to discover what information is logged. Technical and non-technical users would benefit from tools that can automatically extract, categorize, and assign semantic meaning to tokens from log files.

DEC0DE is a tool for recovering information from mobile phones with unknown storage formats, to aid in criminal investigations [13]. The tool compares small blocks of unparsed data to a library of known hashes in order to reveal information of interest, then parses the remaining data with adapted NLP techniques. Fisher, et. al. introduced LearnPADS, an end-to-end system for generating data processing tools directly from ad hoc data [6–8]. It employs a multi-phase algorithm for inferring the structure of ad hoc data sources and generating templates in the PADS data description language. The data itself is then used to generate a semistructured query engine, format converters, statistical analyzers, and visualization routines, without human intervention. The system has similar goals to our work, but does not include a method for storing and retrieving previously-recognized formats, which would prevent repeating the structure-inference process every time a particular ad hoc structure is encountered. Furthermore, unlike our work, the LearnPADS system is not capable of inferring higher-level relationships from the available data in order to establish links between information from multiple files, possibly across multiple file systems (e.g., to allow a user to ask "show me all incoming traffic from source IP 10.0.0.10"). Another drawback is the PADS language itself, which requires users to provide a priori knowledge of the data formats present in the data set to be analyzed. This means that LearnPADS can support ad hoc file formats, but not ad hoc entity strings as our proposed architecture can.

After learning to parse and extract tokens from an ad hoc file format, it is necessary to assign meaning to the extracted entities. Splunk relies on the user to specify the semantic meaning of extracted entities that it does not already recognize. Seaview uses fine-grained type inference to generate log file visualizations based on the semantic meanings (e.g., "Student ID" as opposed to "Integer" or "String") of extracted tokens [9]. Seaview infers semantic relationships between fields in log files, but does not represent record types or file types as our architecture does.

FlashExtract is a newer framework for extracting data from ad hoc documents using examples [10]. However, extraction is performed on a per-file basis, requiring users to highlight examples in every individual input file instead of generating parsing templates for previously-seen formats. FlashExtract also does not utilize an ontology to define semantic relationships between data entities.

3 The Data Semantics Discovery Architecture

Figure 2 shows the proposed data semantics discovery and the relationships between its components.

The system includes a software modules for File Format Identification, Template Generation, Parsing, Data Storage, and Querying. The user provides an ontology of domain-specific concepts and their relationships, which captures the concepts and the levels of abstractions that the user expects to later use for querying. All of the classes that one would expect for such a domain-specific ontology are specified, including concepts file types, record types, entity types, but also processes, files, system queues, etc. In addition to these typical components, concepts descriptions in the ontology also include, whenever available, a specification of training data in the form of collections of labeled samples.

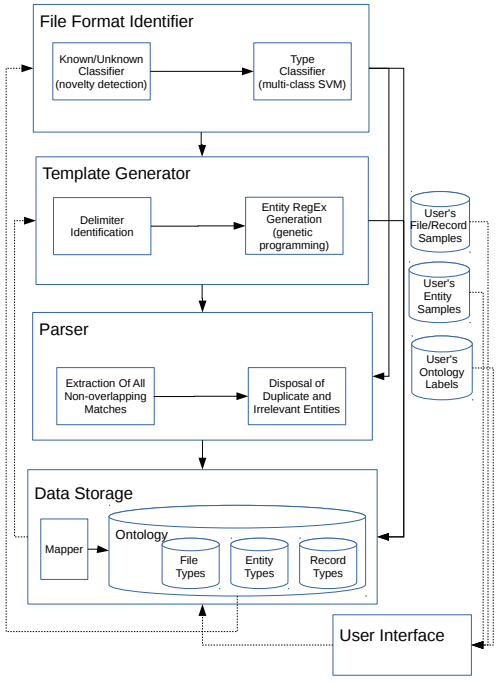

Fig. 2. Proposed architecture

An overview of each component is provided below. However, any architectural component can be modified or replaced to better suit other applications.

In order to simplify the process, our work relies on a few assumptions. The Template Generator and Parsing components assume that alphanumeric characters are never used as delimiters between tokens, and only non-alphanumeric characters may serve as delimiters. However, the Template Generator lifts this assumption, as non-alphanumeric characters are often part of data entities (e.g., "." as part of an IP address token). While the Template Generator does not assume structure to be homogeneous throughout the entire input file, it assumes that the file contains at least one group of lines that can be parsed using each set of delimiters. The process for identifying delimiters is described in greater detail in later. Tokens from the same column in any given line group are assumed to represent the same entity type (e.g., if the token before the first delimiter on a line represents a timestamp, all lines that are formatted the same way contain a timestamp in that position). While all entities appearing on a given line of text

are considered to be part of the same record, their order is not considered for interpretation of the record.

The architecture first identifies the format of the input using supervised machine learning. The result dictates whether the system attempts to retrieve the corresponding parsing template, or create one if no such template exists. The Parser extracts tokens to match per-entity regular expressions in the template, then disposes of duplicate or irrelevant entries. Finally, the extracted tokens are classified by entity type (e.g., "Date-Time," "IPV4," etc.), and mapped to the ontology as complete records. If it encounters information that does not correspond to existing ontology classes, the system is capable of adding new classes to the ontology. This feature is, however, beyond the scope of the present paper and will be discussed separately.

3.1 Ontology

In the proposed architecture, an ontology provides domain knowledge about cyber assets, associates type labels and samples, and stores the data extracted from the files. The domain-specific knowledge contained in the ontology is of the type that is found in textbooks or manuals. This information is specified at development time, and we expect that it is sufficiently general to be sufficient for most scenarios and applications, but obviously it can also be easily extended at run-time. The two top-level classes of the ontology are events and objects, described in more details next.

Events: this class is used to describe host-level events. The data semantics discovery architecture views log files as collections of records of events, with each log file potentially including multiple types of records. The sub-classes of events include:

- *Hardware Events*: events that occurred at the hardware level. This category contains sub-classes for events such as overheating, physical disk damage, peripherals connected/disconnected, etc.
- *OS Events*: events relevant to the kernel, communications layer, software processes, and user actions. Each is described by a different sub-classes, and further divided as appropriate.

All event records identified by the system will be (direct or indirect) sub-classes of the above. The latter class encompasses the largest set of sub-classes, and it is likely that most ad hoc log formats encountered by users will describe events from that category.

Objects: this class represents basic data entities, such as email addresses and network addresses, as well as physical and software objects. Intuitively, events result from actions performed by objects and/or on objects. The ontology includes all objects related to events the user wishes to monitor through log file records. Sub-classes capturing specific object types, including:

- *Hardware Objects*: physical components of a computer, such as a keyboard or a video card.

– *OS Objects*: software objects for which the OS is directly responsible, such as processes, threads, memory; also, objects that exist within, or are created by, applications.

Sub-classes of OS Objects include DHCP tables, which are maintained by the DHCP service, and email messages, which are handled by the email daemon. Files and directories are also OS Objects, and the various types of log files are further sub-classes of files.

The links among the various objects and between objects and events are expressed using a few general *properties*. For example, properties allow the system to memorize in which log file a record was found. Some properties apply to whole classes of the ontology, while others are specific to instances of classes containing the information extracted from log files.

The most important property in the first category is `trainingSamples`. This property is applicable to any class and specifies a path the file(s) containing samples of that class to be used for classifier training. Samples are used as training data for extracting and identifying information from the data sources (see below). Properties that are applied to specific instances include:

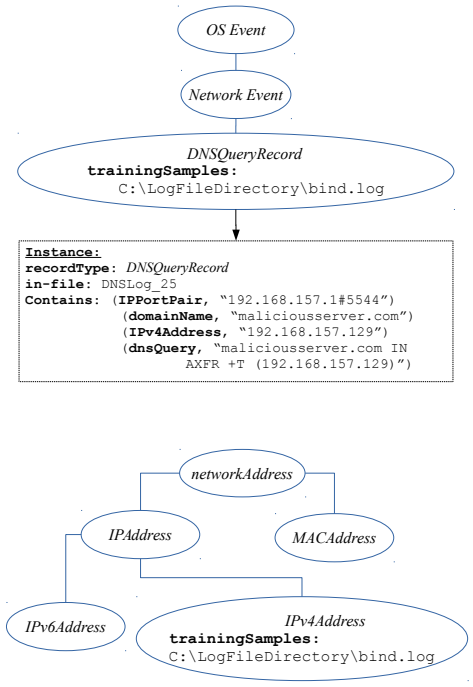

Fig. 3. Storage of a DNS Query Record

– `in-file`: applicable to any event record (see *Events* class), this property allows to specify in which log file the record was.
– `contains`: this property is applicable to event records as well, and is used to specify which data entities were found in that record. For example, many record types contain a date-time entity.
– `text`: this property is applicable to any data entity. It is used to specify the text that was identified as describing that data entity. For example, the value of the `text` property for an *IPv4Address* data entity could be "10.0.0.1."

To see how the information from data sources is encoded by the architecture, consider the sample DNS query record from Figure 1 (top), a special case of a DNS record that a user might search for in the example from Section 1.

Examples of key ontology elements pertinent to the identification and storage of this record are illustrated in Figure 3. Property `trainingSamples` of classes *DNSQueryRecord* and *IPv4Address* provides the location of the training samples for the extraction algorithms. Parsing of the record, performed using the algorithms described later, creates a new instance of the *DNSQueryRecord* class. This class is categorized as a *Network Event*, which is a sub-class of *OSEvent*. A reference to the file in which the record was found is memorized by the instance's `in-file` property. Finally, the components of the record identified by the parsing algorithms are stored as instances of appropriate data entity classes and linked to the *DNSQueryRecord* instance via its `contains` property. For illustrative purposes, here we visualize them as pairs of entity types and values:

- (*IPPortPair*, "192.168.157.1#5544")
- (*domainName*, "maliciousserver.com")
- (*IPv4Address*, "192.168.157.129")
- (*dnsQuery*, "maliciousserver.com IN AXFR +T (192.168.157.129)")

3.2 File Type Identification

Machine learning approaches for automatic classification can be divided into two broad categories: supervised and unsupervised. Supervised methods both compare unlabeled input samples to a set of labeled training samples. Unsupervised approaches require no training data, instead labelling unknown input samples by searching for hidden structures in the data set. A supervised approach best suits our goal of categorizing formats according to labels from an ontology, although exploration of semi-supervised approaches that require fewer labeled input samples is a goal for future work (see Section 5). The File Format Identification component of our architecture identifies file types using a two-stage approach:

- Determining if file type is known: Has the system previously seen files of the same type as the given input file? More generally, does this file contain entities that the system is capable of identifying?
- Identifying the file type: If the given input file is of a known type, which known type is it?

In order to train both classifiers, the architecture extracts training data from the samples provided by the ontology, as discussed above.

The first stage of File Format Identification determines whether the input file is of a "known" or "unknown" (i.e., not previously seen) type using a One-Class Support Vector Machine (SVM) classifier for Novelty Detection. The One-Class SVM is an adaptation of the traditional pairwise SVM, which determines whether or not observed data points come from the same distribution by classifying them as "in-distribution" or "outliers" [11]. Given a set of initial observations

from the same distribution, each described by p features, the classifier learns a contour enclosing the distribution in p-dimensional space. If new observations lay within the contour, they are considered to come from the same population as the initial observation. If they lay outside the contour, they are considered "outliers" belonging to some other distribution. We train a One-Class SVM to recognize all files for which the system has data samples as "known" (i.e., "in-distribution"). The second stage applies to input files classified as "known" during the first stage, and determines which known file type the input corresponds to. Training data for this stage includes labels for each known file format, which coincide with the corresponding class names from the ontology. For the second stage of File Format Identification, we combine several "traditional" pairwise SVM classifiers to create a multi-class classifier [14].

From a technical perspective, both classification stages of the file type identification process use, as features for the learning algorithms, n-grams of space-delimited tokens. In early evaluations, a combination of tri-grams and 4-grams produced the best performance with over 99% accuracy for the second stage of File Format Identification. To reduce dependency on the appearance of specific string values, we perform pre-processing to replace characters with generic *character type labels*, i.e. numeric characters are replaced by character "N" and alphabetic characters are replaced by "A." Punctuation characters are left as-is because they are often important features of specific entities (e.g., "." in the IP address "192.168.1.1").[3]

As an example of file type identification in the prototype implementation of this system, consider a file being analyzed for the motivating use case introduced in Section 1. An excerpt from the file is shown in Figure 1 (bottom).

After extracting features, we use novelty detection to determine whether the file is of a recognized type. The file falls within the distribution of recognized samples, and is classified as being of a known type. The second classification stage compares the file to each class of known files, and identifies it as a MS DNS log.

3.3 Template Generation and Parsing

When it encounters an unrecognized file format, the architecture uses structural cues from the file to generate a parsing template. For the prototype implementation presented here, Template Generation is a two-stage approach, consisting of Delimiter identification, which identifies groups of lines that are parsed similarly, and then identifies the delimiters in each line group, and Regular Expression Generation, which forms regular expressions for the entities in each line group after separating tokens in each line group using the delimiters identified. Both steps are detailed next. The output of this process is a template for parsing the input file. The template contains a set of regular expressions for matching each entity contained in the file.

[3] The evaluation of the effects of the replacement by character type labels and a comparison with other possible approaches are in progress and will be discussed in a separate article.

Delimiter Identification. Many log file formats are "homogenous," meaning that all lines contain the same fields and use the same delimiter. Examples of homogenous log file formats include Comma-Separated Values (CSV) files and Tab-Separated Values files. For these log files, identifying the delimiter is relatively straightforward. However, some log files include a variety of different line formats, with varying delimiters, field types, and even numbers of fields on each line. To account for these files, Template Generation begins by grouping lines in the input file that are formatted the same way. The remaining steps in the Template Generation procedure are applied separately for each line group.

We have explored several methods for identifying line groups, largely based on heuristics:

- Clustering by whitespace: lines with the same number of whitespace characters are clustered together.
- Clustering by all punctuation characters: similar to the first proposed method, but considering the weighted counts of all punctuation characters.
- Clustering by alphabetic and numeric characters: considering weighted counts of all types of characters.

In preliminary experiments with the prototype implementation, the first method produced accurate templates for the files relevant to the scenarios of interest.

Delimiters are identified for each line group. We first identify candidate delimiters by counting the number of appearances of each punctuation character on each line in the given line group, and counting the number of characters between appearances on each line. The delimiter for the line group is the candidate that meets the following criteria:

- Has the minimum standard deviation in its per-line count.
- Has the largest standard deviation in the character distance between its appearances (only applies if multiple characters meet the first criterion).

The first criterion is motivated by the assumption that all lines within the same group contain the same number of fields. If this is the case, a delimiter character should appear the same number of times on each line. However, we allow for some variation in the count in case the character also appears within a nested entity. The second criterion only applies if multiple characters meet the first criterion, and is motivated by the experimental observation that few entity types have fixed character lengths.

Regular Expression Generation. Once data entities in a given line group have been identified, by the process of elimination after identifying delimiters, the Template Generator learns Regular Expressions representing each column of tokens (i.e., tokens before the first delimiter in a line group, tokens between the first and second delimiters, etc.) extracted from the input file. To generate regular expressions that match each column of tokens from a given line group, the prototype implementation uses a Genetic Programming algorithm similar to the one described in [2]. Extracted tokens are first pre-processed to replace alphanumeric characters with "A" or "N." The pre-processed tokens are used as

inputs to the Genetic Programming algorithm, from which we obtain a regular expression that best fits all of the tokens from the column.

The templates generated by this process contain lists of regular expressions for each line group. This substantially simplifies the parsing process, allowing to reduce it to the task of extracting strings using regular expressions.

A particularly challenging case for Template Generation and Parsing is that of user-configurable file formats, such as that of Apache web server logs. The softwares that generate these logs provide users practically complete freedom in the specification of which of the available data entities should appear in the logs, and in which order. The Template Generation and Parsing component accommodates these types of files by adopting two strategies:

- Allowing multiple templates for each file type: When multiple templates exist, the Parser attempts to extract entities using all of them. The template with the largest number of recognized entities extracted is considered to be "correct" for our purposes.
- Generating a new template if too few recognized entities are extracted by the Parser: When very few of the extracted tokens represent recognized entity types, the system generates a new template and stores it with the existing template(s) for the input file format. We apply a threshold for percentage of recognized entity types to determine whether a new template should be created.

3.4 Ontological Mapping

After parsing, the extracted tokens, records, and the files themselves are mapped to the ontology, as follows.

The mapping algorithm begins with Entity Identification, in which tokens extracted during the parsing process are mapped to data entities from the ontology. The process is similar to that of File Type Identification: using the classes and samples from the ontology as training data, first we determine if the input token is a known entity type, then determine which type it is. We use SVMs for this classification problem as well, with features extracted by applying the same replacement by character type labels described earlier and by creating a count vector of the character types (letters, numbers, and punctuation) for each token.

Once the extracted entities have been identified, combinations of entities are stored as records as in the example from Section 3.1. The Mapper module must determine which type of event record object to create (see Section 3.1 for an overview of event types from the built-in ontology).

As before, training data for the record classification task is obtained from the ontology, which specifies training samples and whose class names are used as labels. The output of the File Format Identification module and the results of the Entity Identification process described above are combined to form a feature vector for supervised classification using SVMs.

The first element of the feature vector contains the file format label for the input file, or the special label "UNKNOWN" if the file format was not recognized.

The remaining elements contain the count of every known entity type found during Entity Type Identification, including the number of unrecognized entities. For the kind of data considered here, the prototype implementation has shown good results with this feature representation, which ignores the order of entities' appearance in each record.

Finally, a supervised learning algorithm similar to the algorithm described in Section 3.2 is used to classify the record as one of the known types from the ontology.

3.5 Querying

Once the information has been extracted and stored, the analyst can leverage the hierarchical organization of the ontology to ask queries that span across multiple files and are independent of how the information was originally encoded. In the case of the motivating example from Section 1, our architecture enables the analyst to check for instances of the cyber attack of interest by posing the following query, encoded here in a simplified pseudo-language to simplify the presentation:

```
SELECT R1, R2, R3 WHERE
R1 is a mailRecord,
  R1.contains (emailAddress, .net),
  R1.contains (DateTime D1),
R2 is a dnsQueryRecord,
  R2.contains (DateTime D2),
  D2 > D1,
  R2.contains (domainName, *cyberattacks.com),
  R2.contains (networkAddress, victimPC),
R3 is a dnsQueryRecord,
  R3.contains (DateTime D3),
  D3 > D2,
  R3.contains (domainName, *cyberattacks.com),
  R3.contains (networkAddress, victimPC)
```

The first four lines of the query identify receipt of an e-mail from a ".net" e-mail address and the remaining lines identify two subsequent DNS queries to the malicious server, both occurring on the same network node. The query also requires that the email arrival precedes the DNS queries (conditions $D2 > D1$ and $D3 > D2$). The first line of the query specifies that the corresponding records must be returned, although of course it would be easy to also return, for example, the date times and address of the victim, or the complete sender e-mail address.

Such a query can be easily expressed in a state-of-the-art query language such as SPARQL; for example the following shows how the first 4 lines are translated:

```
SELECT ?r1 ?r2 ?r3 WHERE {
    ?r1 rdf:type dsd:mailRecord .
    ?r1 dsd:contains ?e1 .
    ?e1 rdf:type dsd:EmailAddress .
```

```
        ?e1 dsd:text ?addr .
        FILTER (REGEX(str(?addr), ".net")) .
        ?r1 dsd:contains ?d1 .
        ?d1 rdf:type dsd:DateTime .
        ...
    }
```

Queries can even be built automatically from a higher-level specification, which for example could be part of a library of known cyber attacks.

It is important to stress the practical advantage of the design of the architecture for high-level query answering. In the case of the present example, analyst can identify which network node opened the DNS tunnel regardless of how the DNS queries were actually logged by the server(s). In fact, depending on a server's configuration, the information in the log files might identify network nodes by their IPv4 addresses, their IPv6 addresses, or even their MAC addresses. However, because class *networkAddress* is a super-class for *IPv4Address*, *IPv6Address*, and *MACAddress* (see Figure 3), the use of the *networkAddress* class in the query encompasses all three network address types. This additional level of abstraction allow analysts to disregard irrelevant low-level details as needed.

4 Supervised Learning Evaluation

Successful identification and storage of known data types depends on the effectiveness of supervised learning as described in Section 3. In this section, we report on an empirical evaluation of the learning components of our architecture to enable comparison with future approaches. We evaluate the performance of the supervised learning modules for file format, entity, and record classification with ten cross-fold validations. The data used for this evaluation consist of 2,022 text files from 29 file classes, which contain a combined 12,622 distinct record samples from 22 record classes. The data for entity classification consist of 291 entity samples from 11 classes. The number of entity samples is small compared to the number of record and file samples because we have accounted for duplicates removed during the feature pre-processing step by replacement by character type labels described earlier. Recall that this pre-processing replaces specific alphanumeric characters with character type representations, reducing the number of unique samples required. For all classification problems, the number of samples was distributed as close to uniformly as possible across all classes.

Common performance metrics for supervised learning include *precision*, the fraction of retrieved instances that are relevant, *recall*, the fraction of relevant instances that are correctly retrieved, and *f-measure*, the harmonic mean of precision and recall. Each metrics ranges from 0 to 1, with 1 being the best possible score. The average metrics over all ten cross-validation folds are shown in Table 1. Although the performance is, overall, satisfactory, a discussion of possible ways to improve it is provided in the next section. SVMs were chosen for all three classification problems because they outperformed several other classifier types in preliminary evaluations, but further evaluations of various classifier and feature combinations will be included in future work (see Section 5).

	Precision	Recall	F-Measure
File Format Identification	0.9791	0.9808	0.9799
Record Type Identification	0.835	0.8438	0.8394
Entity Type Identification	0.8279	0.7819	0.8042

Table 1. Supervised learning performance

5 Conclusions and Future Work

We have presented a data semantics discovery architecture capable of parsing and interpreting data from multiple ad hoc data sources, and of correlating information from multiple sources regardless of the format and level of abstraction at which the information was originally encoded. The extraction process is effectively driven by an ontology of domain-specific concepts, which provides samples and labels for the underlying algorithms. The same architecture is also used for answering queries about the extracted data.

Our architecture moves beyond parsing techniques requiring prior knowledge of file formats and is a step toward parsing data sources with completely arbitrary formats. Any component of the architecture can be adjusted or replaced to better suit a user's needs or to perform comparative studies of alternative techniques. The evaluation of the architecture in realistic conditions is under way.

From a practical point of view, our architecture improves upon existing search methods common in cyber-security tools by adding a layer of semantic understanding of the extracted data via an ontology, which allows a user to ask higher-level queries and at the same time tends to return more relevant results than the string-based search methods used by most cyber-security tools.

This paper presented the overall architecture and described its use. Next, we plan to study how the performance of the system (e.g., execution time, accuracy) is affected by the adoption of different techniques for the implementation of its various components. For example, different combinations of features or use of ensemble methods may improve classification performance over the metrics presented in Section 4. In turn, improving classification performance may benefit overall performance because the system naturally depends on accuracy of classification during insertion of data into the knowledge base.

The use of the ontology may help to compensate for misclassifications or ambiguities between related low-level data types. If, for example, an IPv4 address is misclassified as an IPv6 address, it will still be identified as network address and will be returned in response to queries for entities of that type. Exploration and quantitative evaluation of this idea are another subject for future work.

Finally, although we have discussed our architecture in a cyber-security context, we believe it to be applicable to a wide range of domains by simply providing an appropriate ontology and training samples. Verification of this claim will be another direction of future work.

Acknowledgment: The authors would like to thank Philip J. Yoon for useful discussions on the topic of ad hoc data sources.

References

1. S. Alspaugh, B. Chen, J. Lin, A. Ganapathi, M. Hearst, and R. Katz, "Analyzing log analysis: an empirical study of user log mining," in Conference on Large Installation System Administration (LISA), 2014.
2. A. Bartoli, G. Davanzo, A. De Lorenzo, M. Mauri, E. Medvet, and E. Sorio, "Automatic Synthesis of Regular Expressions from Examples with Genetic Programming," Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation, 2012.
3. L. Bitincka, A. Ganapathi, S. Sorkin, and S. Zhang, "Optimizing data analysis with a semistructured time series database," in Proceedings of the 2010 workshop on managing systems via log analysis and machine learning techniques (SLAML '10), 2010.
4. W. Cui, M. Peinado, K. Chen, H. J. Wang, and L. Irun-Briz, "Tupni: Automatic reverse engineering of input formats," in Proceedings of the 15th ACM Conference on Computer and Communications Security, ACM, 2008.
5. A. Doan, P. Domingos, and A. Y. Halevy, "Reconciling Schemas of Disparate Data Sources: A Machine-Learning Approach," ACM SIGMOD Record, vol. 30, no. 2, p. 509–520, 2001.
6. K. F. White, D. Walker, K. Q. Zhu, and Peter, "From dirt to shovels: fully automatic tool generation from ad hoc data," ACM SIGPLAN Notices, vol. 43, no. 1, p. 421–434, 2008.
7. K. Fisher, D. Walker, and K. Q. Zhu, "LearnPADS: automatic tool generation from ad hoc data," in Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, p. 1299–1302, 2008.
8. K. Fisher and D. Walker, "The PADS project: an overview," in Proceedings of the 14th International Conference on Database Theory, 2011, ACM, 2011.
9. S. Hangal, "Seaview: Using Fine-Grained Type Inference to Aid Log File Analysis." (2011).
10. V. Le and S. Gulwani, "FlashExtract: A framework for data extraction by examples." Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. ACM, 2014.
11. B. Scholkopf, et. al., "Estimating the support of a high-dimensional distribution," Neural Computation, vol. 13, no. 7, pp. 1443–1471, 2001.
12. W. Shang, M. Nagappan, A. E. Hassan, and Z. M. Jiang, "Understanding Log Lines Using Developmental Knowledge," in 2014 IEEE International Conference on Software Maintenance and Evolution, 2014.
13. R. J. Walls, E. G. Learned-Miller, and B. N. Levine, "Forensic Triage for Mobile Phones with DEC0DE," in USENIX Security Symposium, 2011.
14. Wu, Lin and Weng, "Probability estimates for multi-class classification by pairwise coupling," JMLR 5:975–1005, 2004.
15. D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, "SherLog: error diagnosis by connecting clues from run-time logs," ACM SIGARCH Computer Architecture News, vol. 38, no. 1, p. 143–154, 2010.