SAND2019-3604C

Named Entity Comparison Algorithms in Enterprise Decision Systems

James Obert*, Phil Campbell
Sandia National Labs
Albuquerque, NM, USA
jobert@sandia.gov

Abstract— **In enterprise decision systems it is necessary to gather detailed computing node data on all hosts via Configuration Modules (CMs). In a fully interoperable enterprise, each deployed CM collects node configuration and event data which is formatted compliant to a standard format such as the Common Platform Enumeration Model (CPE) standard. In practice, however, enterprise segment administrators generally do not adhere to configuration and event entity data naming standards such as CPE. Non-adherence to a common naming standard results in network nodes containing entity names with similar semantic meanings but with differing names. To properly manage the enterprise, a centralized enterprise decision system must somehow correlate disparate names with similar meaning. With the probability that node administrators will intuitively commonly name fragments of entity names based on the name meaning, this paper investigates various matching algorithms, and explores the use of hybrid algorithms to efficiently match entity names.**

Keywords—Entity Recognition; Intelligent Enterprise Management.

This research was funded by the Department of Defense as part of the Defense Information Services Agency's (DISA) Global Information Grid (GIG) interoperability initiative.

I. Introduction

In named entity matching where often an exact string match is not possible, approximate matching methods must be utilized. String *similarity measures* used to match named entities (NE) include sequential character based methods such as Levenstein [5] edit distance. The Levenstien distance metric represents the minimum number of insertions, deletions or substitutions needed to transform one string into another string. Many variants of Levenstien exist that improve the usefulness of the metric including measuring the cost of reordering or transposing characters in a string. The *bag-of-words* (BOW) methods is another approach named entity is represented as a set of features usually in the form of words or character ngrams. The least complex methods count the number of elements in common while other methods in this class represent each NE as a vector and consider parameters that are not included in the NE character sets themselves. Such methods take into the account the frequency of the NE occurrences within a document or dataset to identify relationships between NEs. Using a BOW approach is helpful when there may exist a relationship between the compared NEs with relation to the document or dataset which they reside within. For example, if two NEs occur in combination with similar frequency, that may indicate they possess equivalent semantic meaning. One such method that is popularly used is called the cosine similarity [1] method where each NE of interest in a dataset is represented as a separate feature in a vector. The value of each feature is the number of times the feature occurs in the dataset. The cosine similarity is a measure of the cosine angle between the inner product space between the feature vectors. Cosine similarity is generally used in positive space where unit vectors (similar vectors) are parallel and dissimilar vectors are orthogonal.

NE matching using similarity measures are effective at matching long multi-character NEs that differ only by a few characters while BOW methods are better suited for NE matching problems where there exists meaningful relationships between the NEs within a dataset. In the case of enterprise decision systems, often times the dataset does not contain meaningful relationships between NEs. In such cases the only method for matching coreferent NEs is to use similarity measures.

In this paper, various NE similarity measurement algorithms are utilized to quantify the variation between NEs within an enterprise management system. Matching accuracy and computing complexity of the various algorithms are illustrated, and an efficient hybrid solution is discussed. Finally, various naming scheme are explored in various use cases as means for solving enterprise entity naming issues.

II. BACKGROUND

A. *Character and Token Based Methods*

As stated previously, NEs in an enterprise decision system do not necessarily relate to one another explicitly within the enterprise management dataset. In such cases it is necessary to use string similarity measures to compare the NE character patterns for similarities in hope that system administrators have included similar ngrams in their naming schemes. Character-based methods compare strings in thier entirety. The most common character-based techniques are Levenstein distance [5], Damerau distance [18], Needleman–Wunsch[19], Smith–Waterman[20], Monge–Elkan[21], and Jaro[22]. The distance methods Levenstein and Damerau find the minimum number of operations consisting of insertions, deletions or substitutions of single characters or transpositions of two adjacent characters required to change one string to the other. The other methods inspect character sequence alignments using optimized schemes to derive similarity measures, and are generally utilized for comparing long character sequences in such applications as protein and DNA sequence analysis.

B. *Kernel Methods*

Kernel-based supervised machine learning classification algorithms such as support vector machines can be used to compare strings. Discriminative similarity kernel-based learning attempts to capture the distinction between different strings by considering both positive and negative samples (samples not closely matching) when performing training, and often can provide the most accurate results[23]. One promising kernel method proposed by Leslie in [24] uses a class of kernels that derive features from strings which are superior discriminating between character sequences as compared to generative training scenarios where only positive samples (samples closely matching) are considered.

In many cases NE matching datasets will contain a few positive training samples (character sequences we wish to match) along with many negative training samples. Often when insufficient positive training samples are present, classifier performance will be less accurate (poorly defined decision boundary) which requires the expansion of the training dataset. When adding additional data, it must be manually labeling which results in inefficient training scenarios. Classes of kernels that can process unlabeled data such as the profile kernel [25] and the mismatch neighborhood kernel [26] can much more efficiently classify unlabeled character sequences than the traditional supervised methods. The use of such methods can come at a greater computational cost and should be considered. In the sections that follow, we will discuss the use of a class of mismatch neighborhood kernel called sequence neighborhood kernel (SNK) in matching NE strings and illustrate its performance characteristics.

C. *Hashing Methods*

Several hashing algorithms exist that are often used in exact nearest neighbor and R-near neighbor search problems. When such methods are applied to high-dimension cases, the required computational resources become quite high. In an effort to optimize search efficiency there has been much effort dedicated to developing approximate nearest neighbor search algorithms. Where the exact nearest neighbor search is a proximity search optimization that finds the closest set of points to a given point which is guided by a dissimilarity function. An approximate nearest neighbor search is guided by a metric that considers points some specified distance from the given point with the thought that in most cases an approximate nearest neighbor is as good as a true nearest neighbor.

The most efficient hashing algorithms map the target item to be matched such that an approximate nearest neighbor search can be efficiently and accurately performed using a small subset of the dataset. In the hashing context, the target items are called hash codes. In this paper, we may also call it short/compact code interchangeably. The two methods for using hash codes to perform nearest neighbor search are a hash table lookup and Fast distance approximation. A hash table data structure consists of buckets which are each indexed by hash codes. Each reference item in a dataset is assigned a bucket. The hashing algorithm creates a hash table that maximizes the probability of collision of near items. The Fast distance approximation hashing method compares the searched for item with each reference item by fast computing the distance between the query and the hash code of the reference item. The reference items that have the smallest distances are the possible nearest neighbors. This is then followed by comparing possible nearest neighbor true distances computed using the original features and then attaining the K nearest neighbors or R-near neighbor.

One class of approximate nearest neighbor hashing algorithms is explored and discussed in the following sections and its performance and NE string matching accuracy is discussed in detail.

III. Enterprise Named Entity Similarity Methods

A. *Enterprise Management Domain*

The experimental enterprise management system used (shown in Figure 1) in our analysis primarily gathered detailed asset inventory data on network hosts via the McAfee Asset Configuration and Compliance Module (ACCM) and the McAfee Agent (MA). In the experimental enterprise management system all ACCM and MA managed network nodes were managed by different administrators who did not explicitly name and format collected asset data per the Common Platform Enumeration Model (CPE) standard 2.3 which specifies format and naming conventions per node attribute. Therefore the collection components did not consistently name assets (NEs) using CPE names as a guide, nor has stored asset data fully aligned with the CPE formatting standards. The algorithms described below explores the use of potential NE matching algorithms that given disparate ACCM or MA names can accurately and efficiently find a closely matching CPE name.

Using a representative sample of ACCM and MA collected asset data, the goals of this study were to:

- To describe the variance between the CPE naming and formatting standards and the ACCM and MA sample dataset.
- To measure the accuracy of selected algorithms in matching ACCM and MA names to CPE 2.3 names.
- To suggest the use of a hybrid matching algorithm in performing name matching.
- To explore the use of the SWID naming scheme in various use cases as means for solving naming issues.

The diagram shows the data analysis framework used in the analysis.

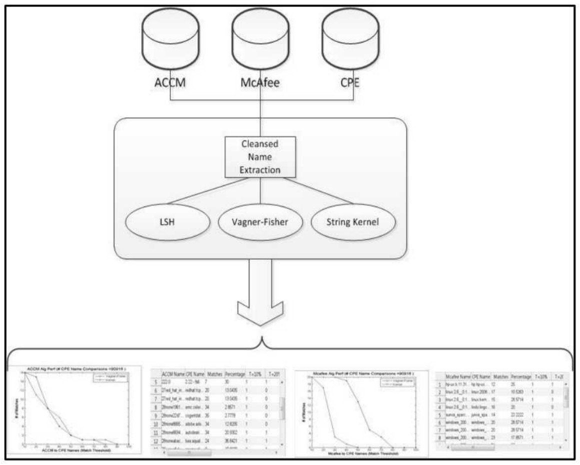


Figure 1: Data Fusion Framework

As will be illustrated in the sections that follow, various candidate string matching and hashing algorithms are utilized to quantify the variation. Matching accuracy and computing complexity of the various algorithms are illustrated, and an efficient hybrid solution is discussed. Finally, the SWID naming scheme is explored in various use cases as means for solving naming issues.

As stated previously, the ACCM and Mcafee MA datasets do not fully comply with the naming and formatting conventions of the CPE standard. For this reason a cleansing and name extraction component was developed in order to extract and format ACCM and McAfee MA datasets to closely match the CPE 2.3 naming format. The ACCM by default adds ASCII control characters as delimiters in addition to the accepted CPE colon delimiters. During name extraction, these non-CPE delimiters are removed from the ACCM dataset such that string matching can be performed using the matching algorithms: Vagner-Fisher and a String Kernel variant. As the MA data fields are delimited with colons without the addition of ASCII control characters, cleansing of the MA dataset was not necessary. As shown in the Figure 1, the Locality Sensitive Hashing (LSH) algorithm is implemented as well in order to measure possible efficiency gains that such hashing can provide in matching performance. Theory and implementation details of these algorithms are explained in the sections that follow.

B. *Data Characterization*

This section explores the possibilities of translation between and generation of names with different structures. The first subsection sets the stage by describing the format for CPE 2.3 names, the format for SWID tags, and the implied format for ACCM and for McAfee names.

C. *CPE 2.3 Names*

NIST IR 7695 states that “The WFN concept [explained below] and the bindings defined by the CPE Naming specification [also explained below] are the fundamental building blocks of all CPE functionality” [7] (page 1).

The “well-formed CPE name” (WFN) is a notation consisting of an “unordered set of attribute-value pairs that collectively (a) describe or identify a software application, operating system, or hardware device, and (b) satisfy the criteria specified in Section 5.2” [7] (page 8).

The syntax of a WFN is as follows [7] (page 9):

wfn:[a1=v1, a2=v2, …, an=vn]

where
a1, a2, and an are attributes and
v1, v2, and vn are values

The "criteria" that appear in the following sections of [9] lists the 11 permissible attributes.

The document provides an example of a WFN:

> wfn:[part = "a", vendor = "microsoft", product = "internet_explorer", version = "8\.0\.6001", update = "beta"]

Note that the values are surrounded by quotes, that spaces within values are replaced with an underscore ("_"), and that the periods embedded in a value are escaped via the backslash.

The WFN is a notation "used solely for the purposes of explaining and illustrating the concepts and procedures specified herein [i.e., in the CPE document]" [9] (page 9). However, a WFN can be bound to machine-readable representations "for interchange and processing" (page 8).

CPE 2.3 provides for two "bindings." The first binding is the Uniform Resource Identifier (URI) binding that provides backward compatibility for CPE 2.2. The example WFN above is shown below in the URI binding:

> cpe:/a:Microsoft:internet_explorer:8.0.6001:beta

Note that the URI binding is a set of ordered values delimited by the colon.

The second binding is the "formatted string binding." The example WFN above is shown below in this binding.

> cpe:2.3:a:microsoft:internet_explorer:8.0.6001:beta:*:*:*:*:*:*

The NIST document provides pseudocode for

- converting (or "binding") a WFN to a URI
- converting (or "unbinding") a URI to a WFN
- converting (or "binding") a WFN to a formatted string
- converting (or "unbinding") a formatted string to a WFN.

The pseudocode is written so that the WFN serves as an intermediary between the two bindings: converting a URI to a WFN and then to a formatted string, and then converting that formatted string to a WFN and then to a URI (or the entire process in reverse) ends up with the same URI that we began with (i.e., the pseudocode is "round trip safe"[9] (page 41)).

D. SWID Tags

A software identification (SWID) tag is an XML file as defined by ISO/IEC 19770-2:20093 "Software Identification (SWID) Tag." The file "resides along with the software on the device" [10] (page 8). A SWID tag consists of seven mandatory fields. The standard allows for "extended data" (i.e., additional fields). This document proposes an additional field named cpe_id. This additional field is a string structured as a CPE 2.3 formatted string binding. So the SWID tag described in this document is a proper superset of a CPE 2.3 formatted string, from which a CPE 2.3 formatted string could be automatically extracted.

The expectation is that a vendor creating a certified SWID tag for a software product will also automatically generate a corresponding CPE with this CPE name included in the SWID tag as the cpe_id…The benefits of this approach cannot be overstated because the CPE name and the SWID tag are managed through all aspects of the products [sic] lifecycle and will enable discovery, compliance, security, patching and many other logistical processes to use exactly the same reference details for a software product [10] (page 13).

E. ACCM Names

Both CPE 2.3 and SWID tags have documents that describe the format. Presumably there is a document that describes the format of ACCM names as well but it is not available for this research. As a result we have to infer the structure from the 161,377 example names provided.

A casual scan suggests that all of the names begin with cpe:/: and that the fields appear to be delimited with colons (they do not appear to be attribute value pairs, for example). CPE 2.2 names begin with cpe:/a for applications, cpe:/o for operating systems, and cpe:/h for hardware. And CPE names—both 2.2 and 2.3—are ordered with fields delimited by colons, not attribute value pairs. So the ACCM names appear to be formatted similar to CPE 2.2 names.

F. Mcafee MA Names

Like the names in the ACCM file reviewed in the previous section, the 870 names in the McAfee file appear to have a structure similar to CPE 2.2.

Each line of the file begins with a or o. If the line begins with a, it ends with %; if the line begins with o, it sometimes ends with:, suggesting that the last field in the name is blank. Lines that begin with o include windows or linux or hp-ux, for example, and lines that begin with a include symantec_antivirus and policy_editor_agent and host_intrusion_protection, for example, so we presume that the convention follows CPE with o for operating system and a for application. In general it appears that the lines use the following structure:

> a:<company name>:<application name>:<date>
> o::<operating system name>:<operating system version>:<patch level>
> o:<company name>:<operating system name>:<operating system version>:<patch level>

Sometimes <patch level> is missing or is n%2fa which is n/a,

As with ACCM names presented above, the McAfee name structure is like CPE 2.2 but not actually CPE 2.2.

G. *Matching Algorithms and Methodology*

After conducting a review of a number of string matching algorithms including sources [3-13], three primary measurement and performance enhancing algorithms are included in the analysis and were selected due to their unique approaches, efficiencies and lower complexity. A brief description of each algorithm is described below.

Vagner-Fisher: The Vagner–Fischer algorithm computes edit distance of each string matching another string and holds them in a matrix. The pseudo code in table 1 below finds the edit distance between two strings, *s* of length *m*, and *t* of length *n*.

Algorithm 1: *Vagner-Fisher(***char** s[1..m], **char** t[1..n]*)*

```
1   let d be a 2-d array of int with dimensions [0..m, 0..n]
2   for i in [0..m]
3       d[i, 0] ← i
4   for j in [0..n]
5       d[0, j] ← j
6   for j in [1..n]
7     for i in [1..m]
8       if s[i] = t[j] then
9         d[i, j] ← d[i-1, j-1]
10      else
11        d[i, j] ← minimum of
12             (
13                 d[i-1, j] + 1,
14                 d[i, j-1] + 1,
15                 d[i-1, j-1] + 1
16             )
17  return d[m,n]
```

The distance array d[i, j] holds distances between the first *i* characters of *s* and the first *j* characters of *t*. The array *d* is a 2-dimensional array holding [0..m, 0..n] values. Lines 2-3 assign Levenshtein distance values of any first string to an empty second string. Lines 4-5 assign distance values of any second string to an empty first string. In lines 7-16, character comparisons are made for strings *s* and *t*. If the characters of the two strings are not the same, then the edit distance is determined in lines 11-16. The algorithm can be adapted to use less space from $O(mn)$ to $O(m)$, but it is not easily parallelized due to data dependencies. A threshold *k* can be set as a minimum distance and a diagonal of width $2k + 1$in distance matrix *d* can be computed. The algorithm can then be run in $O(kl)$ time with *l* the length of the shortest string.

Sequence Neighborhood Kernel (SNK):

Scoring similarity between pairs of sequences between strings can be based on fixed, spectral representations of sequential data and the use of mismatch kernels [11, 13]. In this scoring context, the counts of all short substrings, termed *k-mers*, contained within a sequence are found. A *k-mer* refers to all the possible substrings, of length *k*, that are contained in a string. Like Vagner-Fisher, similarity scores are established by measuring the number of transformations required on a k-mer based on different models of deletions, insertions and mutations. Efficiency for large alphabet sizes and loosely defined matching models are often computationally intensive. As an example, the complexity of well-known trie-based matching algorithms using a mismatch kernel between two strings depends on the alphabet size and the number of mismatches allowed, and are only efficient when employed with shorter strings with smaller alphabets [9, 10]. Predictive models are more capable in terms of being useful with large alphabets and string sizes, but are computationally complex [13].

The *Sequence Neighborhood Kernel* (SNK) algorithm uses inexact mismatch string comparisons, which greatly improve runtimes for string comparison operations. The algorithm relies on an efficient use of mismatch neighborhoods and k-mer statistics on sets of sequences that result in a mismatch kernel algorithm of complexity $\boldsymbol{O}\left(\boldsymbol{c_{k,m}}(|\boldsymbol{X}| + |\boldsymbol{Y}|)\right)$, where $c_{k,m}$ is a constant independent of the alphabet, *X* and *Y* are comparison strings, *k* is continuous substring length, and *m* is the number of mismatches.

Although not performed in our analysis, it is possible to create alphabets consisting of substrings in order to increase match efficiency. The consequence of this strategy is a potentially very large alphabet with thousands of symbols. In such cases, SNK is highly efficient acting as a semi-supervised learning method [16].

Locality Sensitive Hashing (LSH):

Locality Sensitive Hashing can be used to reduce the dimension of high-dimensional data by hashing input data such that similar data is segregated into the same buckets. The goal of LSH is to create hashes for similar data, thereby maximizing the probability of collision. Once data is hashed, it is possible to conduct highly efficient nearest neighbor searches [14, 15]. The hashing and search processing works as follows:

1. ***Pre-processing Step 1 (Random Hash Function Selected):*** A family of G hash functions is obtained by concatenating k functions $h_1 ... h_k$. A random hash function g is obtained by concatenating k randomly chosen hash functions from the LSH family F.
2. ***Pre-processing Step 2 (Constructing Hash Tables):*** A set of hash tables L, each corresponding to a different randomly chosen hash function g, is constructed. On a data set S, each of the data points n are hashed into each of the L hash tables.
3. ***Search Process:*** Given a query point q, the algorithm iterates over the set of hash functions g. For each g considered, it retrieves the data points that are hashed into the same bucket as q. The process stops when a point within a specified distance cR from q is found, where R is a distance threshold and c is an approximation ratio.

The following is the performance for the preprocessing steps and the search process:

Pre-processing: $O(nLkt)$ where t is the time to evaluate a hash function h.

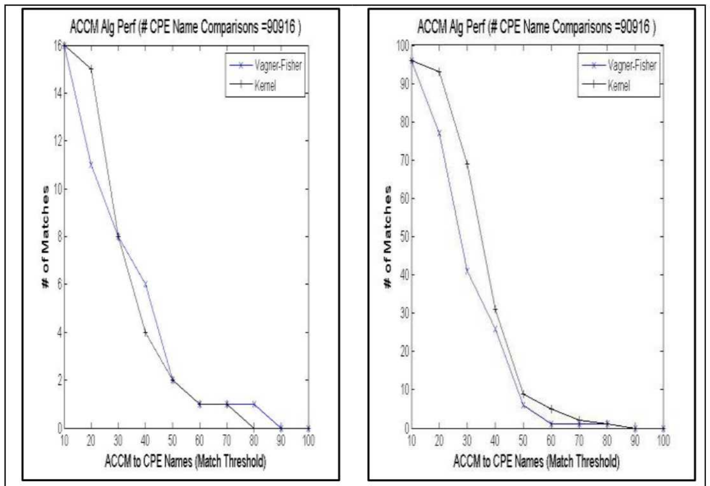
Searching: $O(L(kt + dnP_2^k))$ where d is the distance threshold for a match with probability P_2^k.

IV. Matching Results

A. *Accuracy of Matching Algorithms*

After extraction and cleansing of the ACCM and MA datasets, each of the ACCM and MA names are compared to each of the over 90K CPE 2.3 names. Edit distances, and *k-mer* (defined below) commonality are measured using the Vagner-Fisher (VF) and Sequence Neighborhood Kernel (SNK) algorithms respectively. LSH hashing is also evaluated for its ability to bucket CPE names into various hash tables for efficient nearest neighbor comparison with ACCM and MA names.

Figures 2 below shows the matching accuracy for the VF and SNK algorithms. It should be noted that on a standard 4 core processor, the execution times for comparing 100 randomly selected ACCM and MA samples with the 90K CPE names using VF and SNK is 5.8 and 1.4 hours respectively. With limited computing resources available, comparisons between VF and SNK were only made for 100 samples.



(a) Accuracy SNK & VF 20 ACCM Samples.

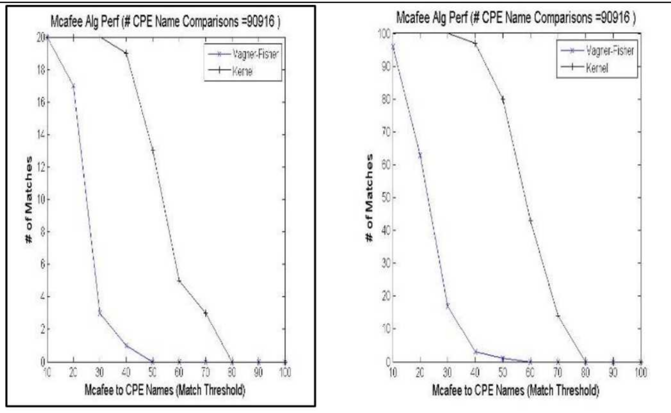
(b) Accuracy SNK & VF 100 ACCM Samples.

Figure 2: Matching Accuracy of SNK & VF with ACCM Data.

Later analysis for the more efficient SNK algorithm was performed with 500 randomly selected samples in order to show accuracy with a higher number of samples. In Figure 2(a) and (b) 20 ACCM and 100 ACCM samples were compared with 90916 CPE 2.3 names. On the horizontal axis the match threshold percentage is indicated with values between 0-100. For VF, the match threshold represents the edit distance as a percentage of the string length while for SNK it represents the k-mer statistic as a percentage of the string length. The vertical axis is a measure of these percentages as compared to the match thresholds. As shown, as the match threshold is increased, the number of positive string matches meeting the distinct match thresholds decrease.

An algorithm that yields a large number of string matches across the entire range of thresholds is an accurate matching algorithm. In Figures 2(a) and (b) the match accuracy decreases rapidly as the match threshold values increase. In 2(a), VF and SNK exhibit approximately the same accuracy while in 2(b) SNK shows a slightly higher accuracy than VF. As the number of ACCM (*A*) and MA (*M*) samples compared to the fixed number of CPE names (*C*) increases, the probability of a match increases (*A/C, M/C*). In 2(b) there are 5 times as many ACCM points sampled as compared to 2(a) which indicates that SNK exhibits a greater computational affinity to the ACCM data than VF, and accuracy is expected to increase as the number of samples compared increases.

Figure 3 below shows the accuracy of VF and SNK for MA data samples. SNK is clearly more accurate than VF. Additionally, on average, both VF and SNK exhibit greater accuracy than with the ACCM data. This indicates that the MA names are closer in structure to CPE names than ACCM names are.

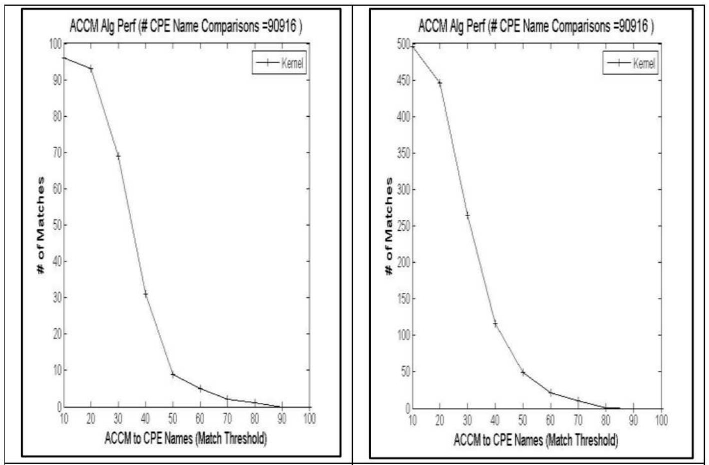


(c) Accuracy SNK & VF 20 Mcafee Samples.

(b) Accuracy SNK & VF 100 Mcafee Samples.

Figure 3: Matching Accuracy of SNK & VF with ACCM Data.

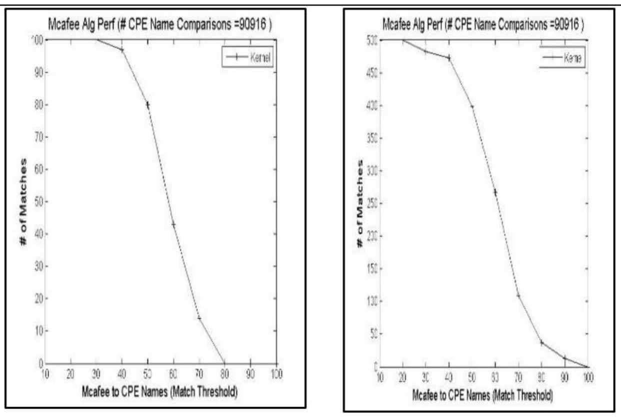
Additionally, the figures above show that SNK is more accurate with ACCM and MA names than VF is.
Figures 4 and 5 below show the accuracy for SNK alone in matching 100 ACCM names & 500 MA names with all 90916 CPE 2.3 names. Again, accuracy increases as sample size increases.



(a) Accuracy SNK 100 ACCM Samples.

(b) Accuracy of SNK 500 ACCM Samples.

Figure 4: Matching Accuracy of SNK with ACCM Data.



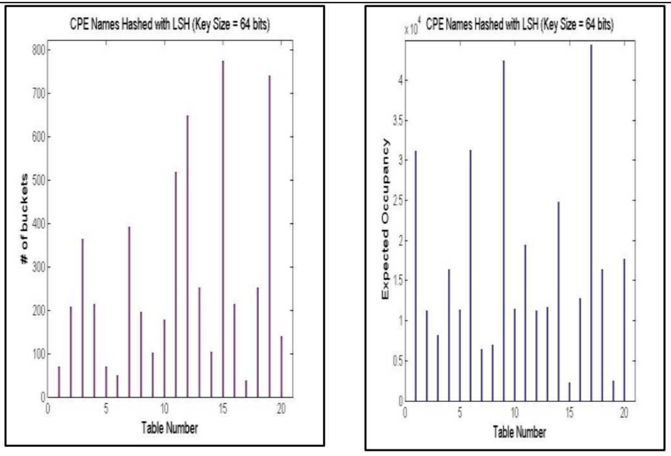
(a) Accuracy of SNK 100 Mcafee samples.

(b) Accuracy of SNK 500 McAfee samples.

Figure 5: Matching Accuracy of SNK with McAfee MA Data

The scope of this investigation did not include measuring the accuracy of the LSH nearest neighbor search algorithm. Rather the research was confined to analyzing LSH hashing as a means to sort the data into hash tables, and with the goal of reducing the overall number of string compares.

Figure 6 below shows how well LSH is able to order CPE names into 20 hash tables with a hashing key length of 64 bits.



(a) LSH and # of buckets per table.

(b) LSH and Expected Occupancy per bucket.

Figure 6: CPE Name Hashing using LSH

In figure 6(a) the horizontal axis lists the number of each hash table generated while the vertical axis lists the number of buckets in each of the tables. In Figure 6(b) the horizontal axis lists the number of each of hash table while the vertical axis lists the expected occupancy of the number of names that potentially could be placed in each bucket of the table.

When generating the tables, the hash key length in bits and the number of tables is specified. The resulting structure is a set of hash tables with each table containing the 90K+ hashed CPE names distributed within the table buckets in quantities based on name similarities. Looking at Figures 6 (a) and (b), it is apparent which tables have the well distributed names. Those tables having the greatest number of buckets and the lowest expected occupancy rate exhibit the greatest distribution of names. When performing nearest neighbor searches, selecting tables with large numbers of buckets and low expected per bucket occupancy rates yields the shortest nearest neighbor search execution times.

As Figure 6 shows, tables 15 and 19 meet these criteria and should be selected when performing a nearest neighbor search for matching an ACCM or MA name to a CPE name. When conducting such a search, a max number of nearest neighbors to be returned is specified. Subsequently, the indices of the closest hash values are returned. From this cursory analysis, it appears that the CPE 2.3 names dataset lends itself well to LSH hashing as tables with both large numbers of buckets and low per bucket expected occupancy rates are present.

B. *Computational Complexity*

Table 1 below shows the average execution times and computational complexity for Vagner-Fisher (VF), Sequence Neighborhood Kernel (SNK), and Locality Sensitive Hashing (LSH) algorithms, given the complexity variables:

d = length of string
k = continuous substring length
m = number of mismatches
n = number of strings
X = a comparison string
Y = a comparison string
w = width parameter
t = time to evaluate function
ρ = probability of collision
s = vector dimension
c_{km} = constant independent of alphabet

Computational complexity indicates the resources needed by a particular algorithm to solve a problem and is generally expressed in Big-O notation. Big-O notation is a mathematical representation of an algorithm's data manipulations. Providing a detailed analysis of the topic of computational complexity is beyond the scope of this study, and reference [15] provides full coverage of the topic.

Algorithm	Average Execution Time (20 ACCM & MA samples)	Avg. Exe Time (100 ACCM & MA samples)	Average Exe Time (500 MA samples)	Complexity
VF	5065.947 s	21092.917 s	Not Demonstrated	$2^{O(\sqrt{\log d \log\log d})}$
SNK	1098.406 s	4978.787 s	26026.967 s	$O(c_{k,m}(\lvert X\rvert + \lvert Y\rvert))$
LSH	Not Demonstrated	Not Demonstrated	Not Demonstrated	$O(n^{\rho}(wt + s))$

Table 1: Algorithm Complexity and Execution Times

Vagner-Fisher has the highest complexity: it exhibited execution times nearly 5 times longer than SNK. The efficiency analysis indicates that VF and SNK algorithms are linear in the number of samples. Although analysis of nearest neighbor search performance was not conducted for LSH, the average execution time for generating a 20 table LSH structure using the approximately 90K CPE names was 11 minutes.

C. *Suggested Fusing Algorithm*

Algorithm 2 below illustrates a suggested data fusing algorithm that will provide greater accuracy and decrease the execution time for finding the most probable CPE name match for ACCM and MA names.

Algorithm 2: *FusingAlgorithm(C, A, M)*

```
1    begin
     /* The LSH key size array $K$ and LSH hash table size array $S$
       are initialized. */
2    initialize $i, q, j$=0; K = {$k_i$, …, $k_B$}; S = {$s_i$, …, $s_R$}; where $k_i$, $s_i$
3      do /* Run LSH algorithm, hash structs $L$ = {$l_i$, …, $l_w$} */
4        $l_i \leftarrow lsh(C, k_i, s_i)$;
5        $i = i + 1$;
6      end do
     /* Finding table $T$, with greatest # bins lowest expected occup*/
7        $T \leftarrow$ optimalTable($L_i$);
     /* Find nearest neighbors $NNA$ and $NNM$ within distance $cR$ */
8      for each $A_j$ do
9        $NNA_j \leftarrow lshNN(T, A_j)$;
10       if($NNA_j <= cR$)
11         $Ns \leftarrow SNK(NNA_j, A_j)$; /* quantify match
```

```
12          end if
13          $j = j + 1;$
14      **end for**
15      **for** each $M_l$ **do**
16          $NNM_q \leftarrow lshNN(T, M_q)$
17          if($NNM_q <= cR$)
18              $Ms \leftarrow SNK(NNM_q, M_q)$; /* quantify match
19          end if
20          $q = q + 1;$
21      **end for**
```

Algorithm 2 arguments CPE names (*C),* ACCM names (*A*), and MA names (*M*) are input in order to first find the optimal search table *T* within the LSH structure *L* (lines 1-7). The optimal search table will be a table that has the highest number of bins per table with the lowest possible per bucket expected occupancy. Once the optimal search table is identified, the nearest neighbors within distance *cR* are then found for all ACCM and MA names (lines 8-20). In order to ensure the closest match and to quantify the match similarity, the SNK algorithm is used to measure the similarity or match statistics *Ms* and *Ns* between the CPE nearest neighbors for each ACCM (A_q) and MA (M_q) name (lines 11, 18).

I. CONCLUSIONS

A. Matching Analysis Summary

As shown in section 3, the variation between ACCM, MA and CPE names is substantial. The number of matches decreases precipitously as the match threshold is increased for both algorithms tested. ACCM formatting requires that cleansing be performed prior to matching and subsequent matching shows low accuracy. MA data does not require cleansing and its names do more closely conform to CPE names than do ACCM names; however, the probability of a match for MA names using the most efficient and accurate algorithm (SNK), is only 0.8 at a 50% match threshold.

Another issue lies in the shear number of string-wise comparisons that will be needed to classify ACCM and MA names on a global scale. In the testing performed thus far, the execution time using the most efficient algorithm (SNK) with 500 MA samples was 7 hours. Applying this algorithm to the projected 10 million existing names would require 20K, 4 core processors each processing for 7 hours.

A practical approach would be to use lower cost GPU equipped computing clusters containing tens of thousands of processor cores. Each of these processors using Algorithm 2 and subsets of the MA and ACCM dataset could simultaneously and efficiently conduct search and compare operations.

Even with the above stated efficiencies in place and considering the variance of ACCM and MA names from the CPE dictionary, it is unlikely that a definitive, high-confidence matching method can be derived without changes to the naming formats themselves. The following sections discuss possible name generation schemes that address these naming differences.

B. Possible Naming Solutions

Given the descriptions of the four name types presented above, this subsection explores the possibilities of translation between and generation of those names, either after the names have been generated ("downstream") or as the names are generated ("upstream").

Downstream:

The ACCM and McAfee names are sufficiently regular that matching to CPE 2.3 names is possible.
However, the fields do not match the CPE 2.3 fields. So it appears that the most efficient way to do the matching may be to parse the names. An ACCM name such as

cpe:/:seagate:seagate_manager_installer:2.01.0013

can be recognized as the seagate_manager_installer application by the company Seagate. And perhaps the version is dated 2.01.0013. A similar approach could be used for the McAfee names. However, in both cases not all of the lines follow this syntax. This complicates the parsing, possibly requiring some lines to be discarded. In some ways this is partly like converting the names to attribute value pairs—a subset of WFN. It does not appear that SWID tags could help here. Granted, SWID tags permit an embedded CPE 2.3 name but in order for a CPE 2.3 name to be embedded, it must first be created. For the purposes here, there is no value in embedding a CPE 2.3 name in a SWID tag.

If the publishers of the software and hardware create names according to a specification imposed upon them, instead of according to their own specification, then, as the SWID tag document noted above proposes, the ideal would be to create a SWID tag with an embedded CPE 2.3 name.

REFERENCES

[1] P.-N. Tan, M. Steinbach & V. Kumar, *Introduction to Data Mining*, Addison-Wesley (2005), chapter 8; page 500.

[2] A. Bronselaer, G. De Tr´ e "Properties of Possibilistic String Comparison", IEEE TRANSACTIONS ON FUZZY SYSTEMS, VOL. 18, NO. 2, APRIL 2010

[3] Rasmussen, C. E., C. K. Williams (2006), "Guassian Processes for Machine Learning*", MIT Press*.

[4] Wolpert, D. (1992), “Stacked Generalization”, *Neural Networks, 5(2)* 241-259.

[5] Leidig, J., C. Trefftz (2007), “A Comparison of the Performance of four Exact String Matching Algorithms”, *IEEE EIT Proceedings*.

[6] Zhenhong, L., F. Hongbo (2011), “Fast Single Pattern String Matching Algorithms Based on Multi-Windows and Integer Comparison”, *Washington DC: IEEE Conference Publishing Services*.

[7] Lecroq, T. (2007), “Fast exact string matching algorithms”, Information Processing Letter, 2007, 102(6): 229–235.

[8] Faro, T. (2013), “The Exact Online String Matching Problem: a Review of the Most Recent Results”, *ACM Computing Surveys, 2013, 45(2)*.

[9] Brant A. Cheikes, David Waltermire, and Karen Scarfone, NIST Interagency Report 7695 “Common Platform Enumeration Naming Specification,” Version 2.3, August 2011.

[10] Steve Klos, “TagVault.org: Certified SWID Tag Integration with Common Platform Enumeration names,” April 9, 2012, Version 2.0.

[11] Christina S. Leslie, Eleazar Eskin, Jason Weston, and William Stafford Noble. Mismatch String Kernels for SVM Protein Classification. In *NIPS,* pages 1417-1424, 2002.

[12] John Shawe-Taylor and Nello Cristianni, *Kernel Methods for Pattern Analysis.* Cambridge University Press, NY, NY, USA, 2004.

[13] Rui Kuang, Eugene le, Ke Wang, Kai Wang, Mahira Siddiqi, Yoav Freund, and Chritina S. Leslie, Profile-Based String Kernels for Remote Homology Detection and Motif Extraction. In *CSB,* pages 152-160, 2004.

[14] Gurmeet Singh, Manku; Das Sarma, Anish, "Detecting near-duplicates for web crawling", *Proceedings of the 16th international conference on World Wide Web. ACM, 2007.*

[15] Das, Abhinandan S., et al., "Google news personalization: scalable online collabortive filtering", *Proceedings of the 16th international conference on World Wide Web. ACM, 2007.*

[16] Jason Weston, Christina Leslie, Eugene le, Dengyong Zhou, Andre Elisseeff, “Semi-supervised Protein Classification Using Cluster Kernels”, *Bioinformatics,* 21(15):3241-3247, 2005.

[17] Trevisan, Luca, “Lecture Notes on Computational Complexity”, *U.C. Berkely Computer Science Dept.,* 2004.

[18] F. Damerau, “A technique for computer detection and correction of spelling errors,” Commun. ACM, vol. 7, no. 3, pp. 171–176, 1964.

[19] S.Needleman and C.Wunsch,“A general method applicable to the search or similarities in the amino acid sequence of two proteins,” J.Mol.Biol., vol. 48, no. 3, pp. 443–453, 1970.

[20] M. Waterman, T. Smith, and W. Beyer, “Some biological sequence metrics,” Adv. Math., vol. 20, no. 4, pp. 367–387, 1976.

[21] A. Monge and C. Elkan, “The field matching problem: Algorithms and applications,” in Proc. Int. Conf. Knowl. Discov. Data Mining, 1996, pp. 267–270.

[22] M.Jaro,“Unimatch: Arecord linkage system: Users manual,”U.S.Bureau of the Census, Washington, DC, Tech. Rep., 1976.

[23] V. N. Vapnik. Statistical Learning Theory. Wiley-Interscience, 1998.

[24] C. S. Leslie, E. Eskin, J. Weston, and W. S. Noble. Mismatch string kernels for svm protein classification. In NIPS, pages 1417–1424, 2002.

[25] R. Kuang, E. Ie, K. Wang, K. Wang, M. Siddiqi, Y. Freund, and C. Leslie. Profile-based string kernels for remote homology detection and motif extraction. J Bioinform Comput Biol, 3(3):527–550, June 2005.

[26] J. Weston, C. Leslie, E. Ie, D. Zhou, A. Elisseeff, and W. S. Noble. Semi-supervised protein classification using cluster kernels. Bioinformatics, 21(15):3241–3247, 2005.