Regex-Based Linkography Abstraction Refinement for Information Security

ABSTRACT

Linkographs have been used in the past to model behavioral patterns for creative professionals. Recently, linkographs have been applied to the context of cyber security to study the behavioral patterns of remote attackers of cyber systems. We propose a human supervised algorithm that refines abstractions to be used for linkographic analysis of common attack patterns. The refinement algorithm attempts to maximize the accuracy of computer-derived linkographs by optimally merging and splitting abstraction classes, represented as regular expressions (regexes). We first describe an algorithm to pick and perform a globally optimal merge of two abstraction classes. We then describe a counterpart algorithm to pick and split a single abstraction class into two separate ones. We build upon the concept of casting a regex as a conjunction of disjunctions and refining by adding and removing conjunctive and disjunctive elements. We also show how the Stoer-Wagner algorithm, normally used for least cost cuts of graphs, can be used to create two optimal subsets of a set of elements.

KEYWORDS

linkography, algorithm, security

ACM Reference Format:

. 2017. Regex-Based Linkography Abstraction Refinement for Information Security. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 10 pages.

https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Attacks authored by state sponsored actors, criminal outfits, ideological enclaves and recreational hackers continue to trouble public and private cyber systems. In order to maintain an advantage over their adversaries, cyber defenders need new and innovative intrusion detection, attribution and response methods. Linkography, a tool which relies on modeling the behavioral patterns of attackers, has been a recent development addressing this need. A linkographic model is improved over time by two distinct methods: ontology refinement and abstraction refinement. An ontology is a set of rules that link instances of behavior types, and an abstraction is a set of rules that classify behaviors. While substantial work has been done on ontology refinement, abstraction refinement has thus far seen limited development. This work takes a step closer towards a reliable abstraction refinement algorithm.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00

https://doi.org/10.1145/nnnnnnn.nnnnnnn


```
Move:
  copy
Execute:
  .*\.exe
Look:
  type
  ipconfig
```

Listing 1: Original abstraction.

```
0: copy alef.txt upload
1: copy bet.txt upload
2: ipconfig >>upload\gimel.txt
3: type dalet.txt >upload\dalet.txt
4: copy.exe he.txt upload
```

Listing 2: Windows console commands and their associated classes.

In this work, linkographs are used to model behavioral cause and effect relationships between sequences of shell commands executed by attackers. Instead of directly interacting with raw data, linkographs are modeled on top of commands abstracted into general classes. For example in the Unix shell: `pwd`, `ls` and `cat` can be categorized into a "Look" class. Each abstraction class is defined by a regular expression (regex). All commands that match a class-defining regex are subsequently placed under that class. An abstraction comprises all the class-defining regexes used to categorize commands.

We work through the following scenario to illustrate an information security application of linkography. The example begins with the abstraction in Listing 1. This abstraction has three classes: The first two classes (Move and Execute) have one stanza each while the Look class has two stanzas. The Move class's only stanza is a literal that will match any "copy" command. The Execute class's only stanza is a regular expression that will match any command ending in ".exe." Each of the Look class's stanzas are literals: The first matches any "type" command, and the second matches any "ipconfig" command. Listing 2 shows an attacker storing files and configuration data in an "upload" directory. Consider the simple self-loop ontology that Figure 1 shows. First, we use the abstraction shown in Listing 1 to classify the raw data set provided in Listing 2. Next, we use the self-loop ontology to specify what links are present in the resulting computer-derived linkograph pictured in Figure 2. At this point in the scenario, a human analyst looks at this raw data set and computer-derived linkograph and notices that all of the data points are related. In response, they propose the human-created linkograph that Figure 3 shows. This analyst intuits that all five events are related because of the "upload" artifact. Clearly, in all of these commands, the attacker is caching the defender's

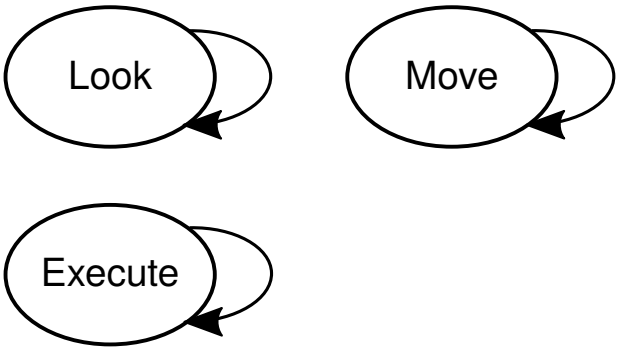

Figure 1: Self-loop ontology.

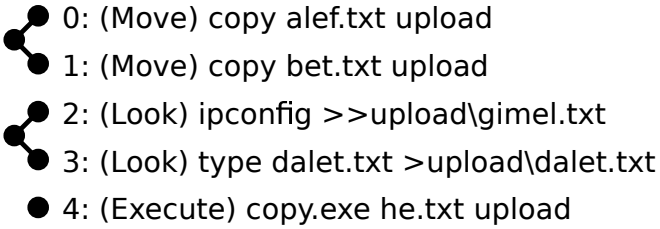

Figure 2: Original computer-derived linkograph.

```
Move:
  copy
  >.*
  copy\.exe
Execute:
  .*\.exe
Look:
  type
  ipconfig
```

Listing 3: Refined abstraction.

information for later exfiltration. An abstraction refinement algo-

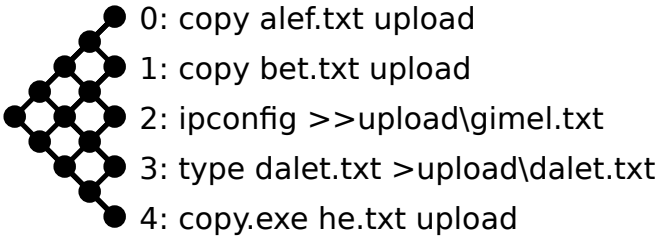

Figure 3: Human-created linkograph.

rithm guided by Figure 3 could produce the abstraction that Listing 3 shows. The reader will notice that this refined abstraction still has three classes. However, the first class now has three stanzas, while the Execute class still has the same one stanza and the third class still has the same two stanzas. The original stanza of the Move class remains. However, one new stanza is a regex that will match any command beginning with ">.", and the second new stanza is a literal that will match any "copy.exe" command. Figure 4 shows the computer-derived linkograph that results from labeling the raw data set provided in Listing 2 with this new abstraction and applying the self-loop ontology shown in Figure 1.

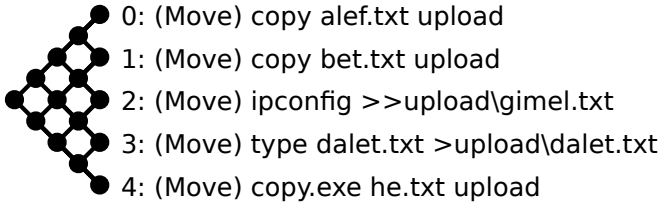

Figure 4: Machine-derived linkograph based on the refined abstraction.

Accuracy is our metric of interest in this study. This is the degree to which a computer-derived linkograph based a particular abstraction and ontology resembles the human-created linkograph from the same raw data set. Mitchell, et al. formally define accuracy in [14], but to summarize: Accuracy is one minus the number of differences (underlinks and overlinks) between two linkographs divided by the number of possible links. Underlinks are the links that are present in the human-created linkograph but not in the computer-derived linkograph. Conversely, overlinks are the links that are present in the computer-derived linkograph but not in the human-created linkograph. Abstraction refinement uses accuracy as the supervisory metric to follow the lead of human-created linkographs and therefore the insight of the human analyst.

Abstraction refinement involves iteratively improving class-defining regexes such that computer-derived linkographs more closely resemble human-created exemplars. In past works, abstraction classes have largely been defined by hand-made regexes [9, 14]. This work aims to automate the creation and improvement of abstractions given only raw data and a target linkograph. Our process begins with a trivial abstraction: a cold start where every command in the raw data defines its own class, or a reverse cold start where a single class matches all the commands in the raw data. Next, we iteratively improve the abstraction by merging or splitting classes via their respective regexes.

The rest of this paper is organized as follows: First, Section 2 surveys other work related to this topic. Next, Section 3 proposes an abstraction refinement algorithm. Third, Section 4 analyzes this algorithm's complexity. Next, Section 5 presents numerical data and figures that quantify and visualize the performance of our abstraction refinement algorithm. Finally, Section 6 contains our conclusions from the study and identifies future lines of research.

2 LITERATURE SEARCH

Fisher, et al. [9] published the first paper relating linkography to computer network defense (CND). These authors formally defined linkographs and related terminology, proposed and proved three theorems about linkographs and introduced the concept of deriving linkographs using abstractions and ontologies. Mitchell, Fisher, Watson and Jarocki [14] identified the problem of linkography ontology refinement and proposed two basic approaches. These authors formally defined ontology and derived linkograph, proposed

a theorem about maximizing the accuracy of an ontology and proposed two algorithms to refine ontologies based on human supervision. Later, Mitchell, McBride and Jarocki [15] identified the problem of linkography abstraction refinement and proposed one basic approach. These authors proposed an algorithm to refine abstractions based on human supervision and introduced the concept of cold start abstraction refinement.

Numerous algorithms have been built in order to approximately learn regexes to classify positive and negative examples. Kearns and Vazirani describe an equivalent problem of learning a discrete finite automaton (DFA) from a set of positive examples [11]. Brāzma and Čerāns have described several methods to generate regexes by folding repeated sequences found in positive examples [5, 6]. Kinber takes a similar approach, but utilizes an oracle which informs the learner if an example is positive or negative. This allows Kinber's algorithm to learn from positive and negative examples [12]. Fernau's algorithm deduces repeating sequences by blocking similar sequences of characters and performing a union operation on different sequences of characters [8]; this technique is used by the algorithm described in this paper. A line of work by Alberto Bartoli, et al. iteratively converges upon an ideal regex from positive and negative examples by using genetic algorithms [1–3]. Bex, et al. and Bui and Zeng-Treitler take a similar domain-specific approach by learning limited regexes for Extensible Markup Language (XML) parsing and clinical data classification, respectively [4, 7]. Galassi, et al. introduce learning regexes from noisy data and false positives. They introduce the concept of atomic elements in a regex that cannot be split and two operators; one of these operators deduces distinct episodes, and one deduces repeated patterns [10]. Myers and Miller describe an algorithm that performs approximate pattern matching. While this is not directly relevant to the algorithm, it is useful for classification and testing the effectiveness of produced regexes [16].

In a key departure from prior formulations, the learning algorithm presented by Li, et al. takes as input not just labeled examples but also an initial regex [13]. Li's algorithm iteratively transitions from a general to a more specific regex by using two transformations: drop disjunction and include intersection. Inspired by this approach, we specifically focus on learning improved regexes by merging and splitting previous ones. We treat merging two regexes as a generalizing transformation and splitting two regexes as a specializing transformation. Finally, given the nature of shell commands, we do not deal with handling repeating sequences.

3 ALGORITHM

An abstraction class is defined by a single regex. A labeling convention can be used to improve readability.

To be analyzed by this algorithm, regexes are read in the form of a disjunction of conjunctions of disjunctions (DCD). More specifically, a regex consists of a disjunction of command patterns that represent a class. Each command pattern is a conjunction of atomic components. Each atomic component is a disjunction of several phrases or symbols. The goal of the algorithm is to build, merge and split regexes of this form to create accurate computer-derived linkographs.

```
(
  (cat )
  (Desktop/)
  (folder1/|folder2/)
  (attack|readme)
  (.exe|.txt)
) | (
  (cd )
  (../)
  (Desktop/|Documents/)
  (folder1/|folder3/)
)
```

Listing 4: Example of a class-defining regex.

Listing 4 shows an example of a class-defining regex. We can think of regexes of this form as a DFA. Each conjunction can be thought of as a node, and each of the many "inner" disjunctions can represent the possible paths to get to the next node. Casting a regex as a DFA only serves as a metaphor to better reason about merge and split algorithms and does not afford any additional expressibility. From the previous example, an equivalent DFA would look like Figure 5.

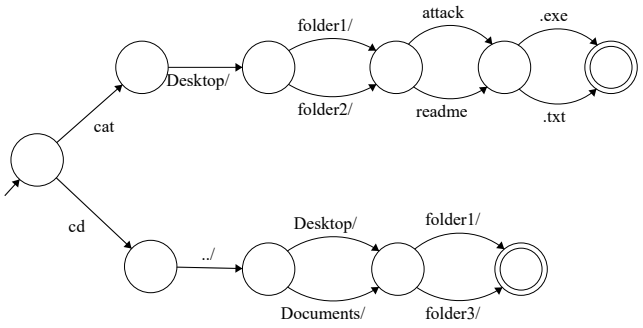

Figure 5: Equivalent DFA for Listing 4.

3.1 Ontologies

An ontology is a set of rules that defines how to link instances of behavioral classes. Once an abstraction categorizes raw data into general classes, an ontology is used to create the final linkograph. Ontologies are formally and comprehensively defined in Fisher, et al. [9]. To summarize, an ontology is a directed graph where vertices represent classes and directed edges represent a cause-and-effect relationship. An ontology is needed at the end of each refinement step to derive a linkograph to compare against the human exemplar.

3.1.1 Self-Loop Ontologies. A self-loop ontology is one of the simplest possible ontologies; each class only points to itself. A full definition of self-loop ontologies can be found in Mitchell, et al. [14]. Due to its simplicity, a self-loop ontology can be generated without any human intervention, given only a list of class definitions. This is especially useful for the purpose of automated abstraction refinement, where the classes are constantly changing and therefore warrant a new ontology with each refinement step. Therefore, for the purpose of measuring the accuracy of automated

abstraction refinement, we fix a self-loop ontology, with constantly changing classes to reflect abstraction refinement updates.

3.2 Approaches to Find the Best Merge

3.2.1 Order Classes by Frequency. As the authors intuited in [15], abstraction classes that occur more frequently in the raw data have a higher chance of impacting the final linkograph when merged with other classes. To find a close-to-optimal merge, we first order classes by the frequency with which instances of these classes appear. We pick the class with the highest frequency as a candidate for the first class to merge. We pick the second merge class by iterating through the remaining classes and picking the first one which increases the accuracy of the derived linkograph. If no possible merge leads to a higher accuracy, we continue to the class with the second highest frequency and so on.

3.2.2 Order by $2 \times \text{underlinks}(a,b) - \text{totalpossiblelinks}(a,b)$. This is only applicable to self-loop ontologies, which are explained in Section 3.1.1. $\text{underlinks}(a,b)$ is defined to be the number of links that are in the human-created linkograph between nodes of class a and nodes of class b but are not in the linkograph derived by a computer using the given abstraction. $\text{totalpossiblelinks}(a,b)$ is defined to be the total number of possible links between class a and class b. We can estimate the benefit of merging two classes a and b using $\text{underlinks}(a,b) - (\text{totalpossiblelinks}(a,b) - \text{underlinks}(a,b))$. This is because merging classes a and b fixes all the underlinks between these classes under the self-loop ontology but will also draw links between instructions that were not human-linked between classes a and b. Now we can find a globally optimal merge by calculating $2 \times \text{underlinks}(a,b) - \text{totalpossiblelinks}(a,b)$ for all classes a and b.

3.3 Approaches to Find the Best Split

3.3.1 Order Classes by Frequency. Similar to finding a close-to-optimal merge, we can find a close-to-optimal split by ordering by frequency. However, unlike finding the best merge, we do not need to pick a second class for split. Therefore, we simply order classes by frequency and split the most frequent class by one of the several split methods listed in Subsection 3.5.

3.3.2 Order by $\text{overlinks}(a_1,a_2) - \text{correctlinks}(a_1,a_2)$. This is only applicable to self-loop ontologies. We will actually order by minimizing cost which we estimate using $\text{overlinks}(a_1,a_2) - \text{correctlinks}(a_1,a_2)$. (a_1,a_2) is defined to be the optimum split of nodes within class a. $\text{overlinks}(a_1,a_2)$ is defined to be the number of links existing between a_1 and a_2 in the computer-derived linkograph that are not in the human-created linkograph. $\text{correctlinks}(a_1,a_2)$ is defined to be the number of links existing between a_1 and a_2 in the computer-derived linkograph as well as in the human-created linkograph. We can solve for (a_1,a_2) as follows: create a fully-connected undirected graph of all the nodes. If there is a link in the human-created linkograph between two nodes, give their corresponding edge a weight of -1. If there is no link between two nodes (and therefore an overlink under self-loop), give their corresponding edge a weight of 1. Then run the Stoer-Wagner algorithm [17] to find the cost-minimizing split of the graph. The Stoer-Wagner algorithm is a recursive algorithm that solves the minimum-cut problem. Specifically, the minimum-cut problem involves cutting a graph with weighted edges into two subgraphs in the least expensive way. The Stoer-Wagner algorithm will maximize the edges with weight -1 and minimize the edges with weight 1 along the split, which in turn minimizes $\text{overlinks}(a_1,a_2) - \text{correctlinks}(a_1,a_2)$.

```
(cat )
(../)
(Desktop/)
(folder1/|folder2/)
(attack)
(.txt)
```

Listing 5: Class 1.

```
(cat )
(../)
(Downloads/)
(folder3/)
(raw_attack)
(.hex)
```

Listing 6: Class 2.

3.4 Merge

The merge algorithm requires three basic steps:

(1) If we are attempting to merge a raw string we need to first split the atomic components. A proposed splitting regex is: `/|\s|(?<!\.)\.(?!\.)`. This proposed regex splits commands, directories, flags, file names and file types. This regex also prevents us from splitting on `..` which commonly refers to the parent directory. A raw string split into atomic components can then trivially be converted into a class-defining regex which accepts only that string. If we are attempting to merge a class-defining regex instead, this step is not required as the atomic components are already defined.
(2) Next, we align the disjunctive clauses of a regex. This will pair up disjunctions with similar commands, directories, files and filenames. To align disjunctive clauses, we count the number of common elements between two clauses. The number of common elements between two clauses serves as a similarity score when using a standard alignment algorithm.
(3) Third, perform a generalizing transformation. To create a generalized regex from two unique regex classes, we are presented with two options: Add phrases to the disjunctive layer and remove clauses from the conjunctive layer. A generalized regex should accept elements from both previous classes as well as additional patterns that we "infer."
   (a) Add Elements to a Disjunction: For each pair of disjunctions in a series of aligned disjunctions from two separate classes, we can simply create a new disjunction that consists of all the elements of the pair of disjunctions. This is guaranteed to accept all elements of both classes. Additionally, this allows us to match new patterns that could not have been matched by either class. For example, consider the classes in Listings 5 and 6. The merged class

```
(cat )
(../)
(Desktop/|Downloads/)
(folder1/|folder2/|folder3/)
(attack|raw_attack)
(.txt|.hex)
```

Listing 7: Merged class.

```
(cat )
(../)
(Desktop/|Downloads/)
(
  folder1/
  |folder2/
  |folder3/
  |folder4/
  |folder5/
)
(attack|raw_attack)
(.txt|.hex)
```

Listing 8: New merged class.

```
(cat )
(../)
(Desktop/|Downloads/)
(
  folder1/
  |folder2/
  |folder3/
  |folder4/
  |folder5/
  |(.)*
)
(attack|raw_attack)
(.txt|.hex)
```

Listing 9: New merged class after removing conjunction.

in Listing 7 is able to identify elements of both original classes but also has the benefit of detecting similar patterns such as "attack.hex" or "raw_attack.txt" being analyzed directly in any of the six combinations of Downloads/Desktop and folder1/folder2/folder3. From a DFA perspective, this can be thought of as adding more paths between conjunctive states.

(b) Remove Elements from a Conjunction: This is a little more tricky. We do not want to remove a conjunction entirely because that could actually make a regex more general. For example, if we originally have regex `(cat )(folder1/)(attack)(.py)`, removing the disjunctive clause `(attack)` will create `(cat )(folder1/)(.py)`, which does not make sense and will fail to match certain previously matched commands. Instead, we can add a general clause to a disjunction to let it match everything. For example, say we add several more classes to our merged example from before, and we end up with the class shown in Listing 8.

If we decide to generalize once a disjunction reaches a certain number of phrases, say four, we will end up with the class shown in Listing 9.

```
(
  folder1/
  |folder2/
  |folder3/
  |folder4/
  |folder5/
)
```

Listing 10: Retained information.

```
(cat )
(../)
(Desktop/|Downloads/)
(
  folder1/
  |folder2/
  |folder3/
  |folder4/
  |folder5/
)
(attack|raw_attack)
(.txt|.hex)
```

Listing 11: General class.

Essentially, this transformation allows us to match commands regardless of which subfolder they are in. Intuitively, this makes sense because it would seem that the subfolder varies largely and does not give us much information about what specific class we are in. For splitting purposes, we chose to retain the information shown in Listing 10. We have several choices for when to remove elements of a conjunction:

Trivial Merge
This choice only adds elements to a disjunction. It does not subsequently remove elements from a conjunction.

Simple Threshold Merge
This is shown in the example illustrated by Listings 8, 9 and 10. This choice removes a disjunctive clause once its number of phrases exceeds a certain threshold.

Using an Objective Function
We can iterate over all possible conjunction-removing transformations and pick the one that maximizes accuracy. When viewed from a DFA perspective, this can be viewed as allowing a "free" transition from one conjunctive state to another.

3.5 Split

The split function is more simple and only requires one step:

(1) Perform an Ungeneralizing Transformation. To split a generalized regex into two unique ones we are presented with two options: Remove phrases from the disjunctive layer and add clauses to the conjunctive layer.

(a) Removing Elements from a Disjunction: For each disjunctive clause in a conjunction, we can split the phrases of that clause into two separate classes. For example, we can split as shown in Listing 11.

We have several options for how to split elements from a disjunction:

```
(cat )
(../)
(Desktop/|Downloads/)
(folder1/|folder2/|folder3/)
(attack|raw_attack)
(.txt|.hex)
```

Listing 12: Split class 1.

```
(cat )
(../)
(Desktop/|Downloads/)
(folder4/|folder5/)
(attack|raw_attack)
(.txt|.hex)
```

Listing 13: Split class 2.

Trivial Split
This option removes the most beneficial element from each disjunctive clause.

Even Split
This option splits each disjunctive clause evenly and randomly.

Using an objective function
We can iterate over all possible splits and pick one that maximizes accuracy. This can potentially be done greedily, or just approximated.

(b) Adding Elements to a Conjunction: Adding elements to a conjunction involves picking atomic elements from the raw data and incorporating them into the conjunctive clauses of regexes to make them more specific. However, picking close-to-optimal atomic elements from the raw data is a non-trivial task. We can also indirectly restore elements already in a conjunction by removing the `(.)*` element in a conjunction.

3.6 Cold Start Abstractions

We assume that we are provided with raw data containing a series of commands. We can automate the process of creating abstractions from the provided data by starting with one of the following trivial abstractions and refining it using merges and splits. Ideally, refinement starting from standard and reverse cold starts should converge.

Standard Cold Start
Every command in the raw data is turned into a class. Refinement on this abstraction will consist of mainly merges.

Reverse Cold Start
Every command in the raw data is placed in a single class. Refinement on this abstraction will consist of mainly splits. One caveat is that the algorithm assumes that a single command is a single disjunction class. For example, the following class-defining regex is not allowed:

```
((cd )(Desktop/|Documents/))
| ((cd )(Workspace/|Downloads/))
| ((ls )(Desktop/|Documents/))
```

The format we expect is shown in Listing 14.

```
(
  (cd )
  (
    Desktop/
    |Documents/
    |Workspace/
    |Downloads/
  )
) | (
  (ls )
  (Desktop/|Documents/)
)
```

Listing 14: Expected format.

3.7 Implementation Choices

We implement the refinement algorithm to handle raw data from the Windows command line environment. The find best merge and find best split actions are implemented using the ordering classes by frequency approaches described in Sections 3.2.1 and 3.3.1. Merge is implemented using the proposed regex. Component alignment is implemented using a similarity score based on the number of common elements between two disjunctions. Elements are removed from a conjunction using a simple threshold merge. As for split, we remove elements from a disjunction using an even and random split. Because of the computational complexity of optimally adding elements from the raw data to regexes, adding elements to a conjunction only involves removing the `(.)*` if it increases accuracy.

Algorithm 1 Find the best class to merge with.

```
1:  function FIND_BEST_MERGE(C, c, b, d, l)
2:      bma ← calculate_accuracy(b, d, l)
3:      for c′ ∈ C do
4:          if c ≠ c′ then
5:              c′′ ← merge(c, c′)
6:              b′ ← b − c
7:              b′ ← b′ − c′
8:              b′ ← b′ + c′′
9:              u ← calculate_accuracy(b′, d, l)
10:             if u > bma then
11:                 return bma, b′
12:             end if
13:         end if
14:     end for
15:     return bma, b
16: end function
```

3.7.1 Find Best Merge. Algorithm 1 requires five input parameters:

C holds the list of abstraction classes sorted by the frequency with which they appear in the raw data set, c is the abstraction class to be merged with another, b holds the current working abstraction, d holds the raw data set, and l holds the human-created linkograph. This algorithm uses five state components: bma holds the best accuracy measured so far. c' iterates over the list of sorted abstraction classes. c'' holds the provisional merged class. b' holds

the provisional abstraction. u holds the accuracy of the provisional abstraction.

Algorithm 2 Find the best way to split a class.

```
1:  function FIND_BEST_SPLIT(c, b, d, l)
2:      bsa ← calculate_accuracy(b, d, l)
3:      c_0, c_1 ← split(c)
4:      b' ← b − c
5:      b' ← b' + c_0
6:      b' ← b' + c_1
7:      u ← calculate_accuracy(b', d, l)
8:      if u > bsa then
9:          return bsa, b'
10:     end if
11:     return bsa, b
12: end function
```

3.7.2 Find Best Split. Algorithm 2 requires four input parameters: c is the abstraction class to be split, b holds the current working abstraction, d holds the raw data set, and l holds the human-created linkograph. This algorithm uses six state components: bsa holds the best accuracy measured so far. c_0 and c_1 hold the splits of c. b' holds the provisional abstraction. u holds the accuracy of the provisional abstraction.

Algorithm 3 Threshold merge two classes.

```
1: function MERGE(c, c')
2:     c'' ← add_disjunction(c, c')
3:     c'' ← remove_conjunction(c'')
4:     return c''
5: end function
```

3.7.3 Merge. Algorithm 3 requires two input parameters: c is the first abstraction class to merge, and c' is the second abstraction class to merge. This algorithm uses a single state component: c'' first stores the result of add_disjunction(). c'' is then passed through remove_conjunction() to remove conjunctive elements with more elements than the threshold.

Algorithm 4 Random and even split of a single class.

```
1: function SPLIT(c)
2:     c_0, c_1 = remove_disjunction(c)
3:     c_0 = add_conjunction(c_0)
4:     c_1 = add_conjunction(c_1)
5:     return c_0, c_1
6: end function
```

3.7.4 Split. Algorithm 4 requires a single input parameter: c is the abstraction class to split. This algorithm uses two state components: c_0 and c_1 first store the results of remove_disjunction(). Then they are individually passed through add_conjunction() to add back previously removed conjunctive elements by removing all instances of `(.)*`. Note that the overall split is not accepted unless this action increases accuracy.

Algorithm 5 Combines the disjunctions of two classes.

```
1:  function ADD_DISJUNCTION(c, c')
2:      c'' ← ∅
3:      D ← get_disjunctions(c)
4:      D' ← get_disjunctions(c')
5:      AD, AD' = align(D, D')
6:      for d ∈ AD, d' ∈ AD' do
7:          d'' ← d ∪ d'
8:          c'' ← c'' + d''
9:      end for
10:     return c''
11: end function
```

3.7.5 Add_Disjunction. Algorithm 5 requires two input parameters: c, c' hold the input classes to merge. This algorithm uses eight state components: D, D' hold the disjunctions of c, c'. AD, AD' are the aligned results of D, D'. d, d' iterate through elements of AD, AD'. d'' holds the union of d, d'. c'' holds the new class to return.

Algorithm 6 Remove a conjunction from a class.

```
1: function REMOVE_CONJUNCTION(c)
2:     D ← get_disjunctions(c)
3:     for d ∈ D do
4:         if |d| > t then
5:             d ← d + (.)*
6:         end if
7:     end for
8:     return c
9: end function
```

3.7.6 Remove_Conjunction. Algorithm 6 requires one input parameter: c is the abstraction class to remove a conjunction from. This algorithm uses three state components: D holds the disjunctions of c, d iterates through D, and t holds the preset threshold.

Algorithm 7 Remove a disjunction from a class.

```
1:  function REMOVE_DISJUNCTION(c)
2:      D = get_disjunctions(c)
3:      c_0 ← ∅
4:      c_1 ← ∅
5:      for d ∈ D do
6:          d_0, d_1 = split(d)
7:          c_0 ← c_0 + d_0
8:          c_1 ← c_1 + d_1
9:      end for
10:     return c_0, c_1
11: end function
```

3.7.7 Remove_Disjunction. Algorithm 7 requires a single input parameter: c is the abstraction class to remove disjunctive elements from. This algorithm uses six state components: D holds all the disjunctions in c, d iterates through D, d_0 and d_1 hold a random

split of disjunctive elements in d. c_0 holds the disjunctive elements not removed from c. c_1 holds the disjunctive elements removed from c.

Algorithm 8 Add conjunctions to c.

```
1: function ADD_CONJUNCTION(c)
2:     D ← get_disjunctions(c)
3:     for d ∈ D do
4:         if (.)* ∈ d then
5:             d ← d − (.)*
6:         end if
7:     end for
8:     return c
9: end function
```

3.7.8 Add_Conjunction. Algorithm 8 requires a single input parameter: c is the abstraction class to remove conjunctions from. This algorithm uses two state components: D holds the disjunctions of c, and d iterates through D.

4 ANALYSIS

The refinement process is allowed a maximum number of changes. Each change consists of an iteration of refinement which first orders classes by frequency, then finds the best merge and finds the best split for each class and exits once a beneficial merge or split is found. In order to sort classes by frequency, we first need to compute the number of instances of each class. This is linear with respect to the number of raw data elements, d. After counting the number of instances of each class, ordering classes using a standard sorting algorithm has a runtime of $O(N \log(N))$, where N is the number of classes. This gives a combined runtime of $O(d + N \log(N))$. Finding the best merge takes as input a single candidate for a merge and iterates through the remaining classes, provisionally merges them, measures their accuracy and returns the best merge results. This gives it a runtime of $O(N)$ with respect to the number of classes. Finding the best split simply splits a single class, so it has a runtime of $O(1)$ with respect to the number of classes. In a single refinement step, we first order classes once, then we sequentially perform the find best merge and find best split techniques until we find a merge or split that increases accuracy. In the worst case, the find best merge and find best split techniques will be run N times each. This gives an overall runtime of $O(d + N \log(N) + N(N + 1)) = O(d + N^2)$ for a single refinement step. In a cold start, the number of classes is equivalent to the number of raw data elements simplifying the worst-case overall runtime to $O(d + d^2) = O(d^2)$. In a reverse cold start, the first refinement step only contains a single class reducing the complexity to $O(d + 1^2) = O(d)$. In the worst case however, we can have r classes where r is the maximum allowable refinement steps. This is because each of the r refinement steps can generate a new class. In this case, the worst case time complexity is $O(d + r^2)$. For each pair of classes, the merge action must align their respective regexes and invoke the remove conjunction and add disjunction subroutines. The alignment algorithm has a runtime of $O(N^2)$ with respect to the number of atomic elements. For each pair of class-defining regexes, their merge action must align their respective regexes and subsequently invoke the remove conjunction and add disjunction subroutines. The alignment algorithm uses a standard dynamic programming approach, which is capable of aligning two strings s_1 and s_2 in $O(|s_1||s_2|)$. In our case, however, we align disjunctions instead of characters. In the worst case, each disjunction can have t elements, where t is the preset threshold. Alignment scoring, which counts the number of common elements in two disjunctions can therefore be done in $O(2t) = O(t)$ time. Therefore, we are able to find the best alignment of two regexes in $O(tx_1x_2)$ where x_1 and x_2 are the number of disjunctions in the first and second regexes, respectively. The add disjunction action simply creates a new merged regex of all the aligned elements in $O(a_1 + a_2)$ where a_1 and a_2 are the number of atomic elements in each regex. Remove conjunction scans the new regex and adds a `(.)*` element for disjunctions which contain more than t elements; this also runs in $O(a_1 + a_2)$. This gives merge an overall runtime of $O(tx_1x_2 + 2(a_1 + a_2)) = O(tx_1x_2 + a_1 + a_2)$. We know that the number of disjunctive clauses is bounded by the number of atomic elements, therefore merge runs in $O(ta_1a_2)$ overall. Finally, the split action only invokes the remove disjunction and add conjunction subroutines. Remove disjunction performs a random but even split of all the atomic elements in a class and therefore runs in $O(a)$ time where a is the number of elements in the class to split. Add conjunction functions similarly to remove conjunction but removes `(.)*` instead of adding it, meaning it also runs in $O(a)$ time. The overall split algorithm therefore runs in $O(a)$ time. The add disjunction action simply creates a new merged regex of all the aligned elements in $O(N)$ time with respect to the number of atomic elements. The remove conjunction scans the new regex and adds a `(.)*` element for disjunctions which contain more than a threshold number of elements; this also runs in time $O(N)$ with respect to the number of atomic elements. Finally, the split action invokes the remove disjunction and add conjunction subroutines, both of which run in linear time as shown earlier.

5 RESULTS

The success of the refinement algorithm was tested using 47 hand drawn linkographs and accompanying raw data sessions captured from an actual attacker actioning a real victim. Each session consisted of 33.2 shell commands on average. We tested the refinement process with both the cold start and the reverse cold start abstraction. Below, we measure the average accuracy over the 47 sessions based on the maximum allowed changes, measured in the number of merges and splits done. Each setting of allowed changes was run as a separate instance of the refinement algorithm. We also vary the merge threshold across all possible number of allowed changes.

In the cold start abstraction, every command has its own class, so the refinement process almost exclusively relies on merging classes to increase accuracy. A typical cold start abstraction scores around 0.76 accuracy before refinement. We can see that increasing the number of allowed changes steadily increases the accuracy up until around seven changes when the accuracy plateaus at around 0.91. We can also see that a smaller threshold typically leads to better accuracy results. Intuitively, this means the algorithm is more aggressively generalizing regexes, which in turn leads to broader

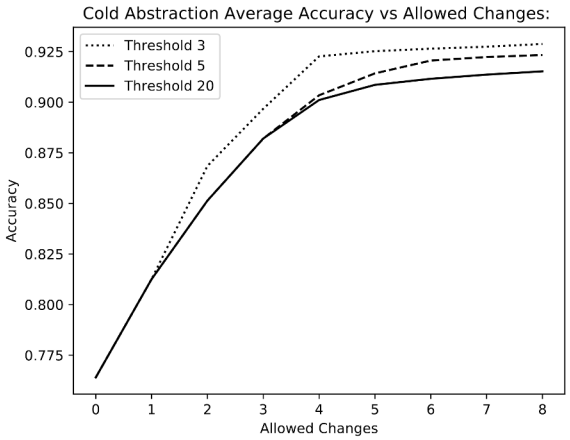

Figure 6: Cold start results.

and fewer classes, which leads to better linkographs when beginning from a cold start. Figure 6 visualizes the impact of threshold and the number of allowed changes on accuracy when beginning with the cold start abstraction.

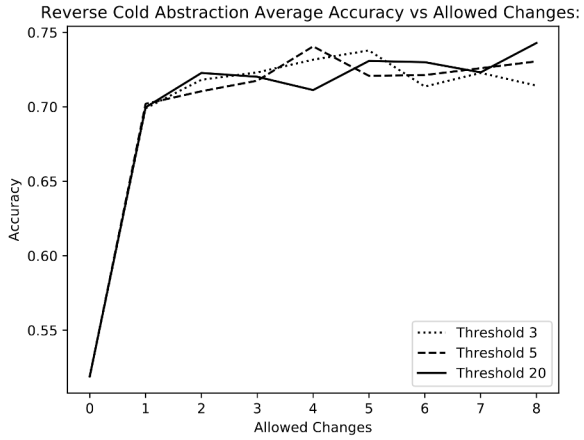

Figure 7: Reverse cold start results.

In the reverse cold start abstraction, a single class matches all the commands, so the refinement process almost exclusively relies on splitting classes to increase accuracy. A typical reverse cold start abstraction scores around 0.52 accuracy. Interestingly, the first refinement step pushes the accuracy to around 0.70, and the remaining refinement steps only marginally improve the accuracy. Given eight changes, the final accuracy is around 0.725. As expected, threshold plays little to no role when beginning with a reverse cold start abstraction. This is understandable as splitting does not have to merge disjunctions under any circumstance. The threshold variable, however, does play a small role in creating the initial reverse cold start abstraction when commands are merged into a single class. Figure 7 visualizes the impact of threshold and the number of allowed changes on accuracy when beginning with the reverse cold start abstraction.

6 CONCLUSIONS

In future work, we will study a number of open questions from this article. We hope future iterations of abstraction refinement can find a globally optimal solution similar to ontology refinement [14]. This might require us to consider linkographs that are constrained in some way: limited to a self-loop ontology, for example. While picking the next best merge and split in order of frequency is generally a successful approach, it is not always optimal. Ordering by the number of overlinks and underlinks is a closer to optimal solution, but it still may suffer from premature optimization when considering the overall refinement process. Properly merging and splitting classes after they are picked is a separate optimization problem. Threshold merging of aligned regexes is currently a fairly straightforward and intuitive solution. In future solutions, we may seek more robust methods of merging regexes. Even but random splitting of regexes is also a straightforward and intuitive solution that can be improved. Future implementations can potentially perform a more optimal split by using the Stoer-Wagner algorithm. Additionally, it is important to note that a refinement beginning with the reverse cold start abstraction is not nearly as successful as one beginning with the cold start abstraction. Refinements that start from reverse cold start abstractions can potentially be improved if the split algorithm does not rely on randomness but rather on a heuristic to decide how to split a class. We also suspect that finding a computationally feasible method of adding conjunctions into regexes can improve the overall success of the split function. Ideally, we can convince ourselves that we are finding near-optimal abstractions if refinement ends with almost identical accuracies starting with either a cold start or a reverse cold start abstraction. We can also consider alternative strategies to abstraction refinement. We can potentially use genetic algorithms to improve regexes as seen with Bartoli, et al. [1–3]. We can also define abstractions based on traditional classification methods such as bag of words or feature similarity. The end goal, regardless of method, is to create accurate and automated linkographs quickly for the purpose of reliable intrusion detection, correlation and attribution.

REFERENCES

[1] Alberto Bartoli, Giorgio Davanzo, Andrea De Lorenzo, Eric Medvet, and Enrico Sorio. Automatic synthesis of regular expressions from examples. *Computer*, 47(12):72–80, 2014.
[2] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. Learning text patterns using separate-and-conquer genetic programming. In *European Conference on Genetic Programming*, pages 16–27, Copenhagen, Denmark, April 2015. Springer.
[3] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. Inference of regular expressions for text extraction from examples. *Transactions on Knowledge and Data Engineering*, 28(5):1217–1230, 2016.
[4] Geert Jan Bex, Wouter Gelade, Frank Neven, and Stijn Vansummeren. Learning deterministic regular expressions for the inference of schemas from XML data. *ACM Transactions on the Web (TWEB)*, 4(4):14, 2010.
[5] Alvis Brāzma. Learning of regular expressions by pattern matching. In *Computational Learning Theory*, pages 392–403, Barcelona, Spain, March 1995. Springer.
[6] Alvis Brāzma and Kārlis Čerāns. Efficient learning of regular expressions from good examples. In *Algorithmic Learning Theory*, pages 76–90, Reinhardsbrunn Castle, Germany, October 1994. Springer.
[7] Duy Duc An Bui and Qing Zeng-Treitler. Learning regular expressions for clinical text classification. *Journal of the American Medical Informatics Association*, 21(5):850–857, 2014.

[8] Henning Fernau. Algorithms for learning regular expressions from positive data. *Information and Computation*, 207(4):521–541, 2009.

[9] Andrew Fisher, Kent Carson, David Zage, and John Jarocki. Using Linkography to Understand Cyberattacks. In *Conference on Communications and Network Security*, Florence, Italy, September 2015.

[10] Ugo Galassi, Attilio Giordana, L Saitta, and M Botta. Learning regular expressions from noisy sequences. *Lecture notes in computer science*, 3607:92, 2005.

[11] Michael J Kearns and Umesh Virkumar Vazirani. *An introduction to computational learning theory*. MIT press, 1994.

[12] Efim B Kinber. Learning Regular Expressions from Representative Examples and Membership Queries. In *International Colloquium on Grammatical Inference*, pages 94–108, Valencia, Spain, September 2010. Springer.

[13] Yunyao Li, Rajasekar Krishnamurthy, Sriram Raghavan, Shivakumar Vaithyanathan, and HV Jagadish. Regular expression learning for information extraction. In *Conference on Empirical Methods in Natural Language Processing*, pages 21–30, Honolulu, Hawaii, USA, October 2008. Association for Computational Linguistics.

[14] Robert Mitchell, Andrew Fisher, Scott Watson, and John Jarocki. Linkography Ontology Refinement and Cybersecurity. In *Computing and Communication Workshop and Conference*, Las Vegas, Nevada, USA, January 2017.

[15] Robert Mitchell, Marci McBride, and John Jarocki. Linkography Abstraction Refinement and Cyber Security. In *Conference on Communications and Network Security*, Las Vegas, Nevada, USA, October 2017.

[16] Eugene W Myers and Webb Miller. Approximate matching of regular expressions. *Bulletin of mathematical biology*, 51(1):5–37, 1989.

[17] Mechthild Stoer and Frank Wagner. A Simple Min-cut Algorithm. *J. ACM*, 44(4):585–591, July 1997.