A New Approach to Fuzzy Regular Expression Parsers for Cybersecurity Logs

Trevor Martin
Intelligent Systems Lab
University of Bristol
Bristol, BS8 1UB, UK
Trevor.Martin@bristol.ac.uk

Alex Healing
BT Future Security Research
BT Research and Innovation
Adastral Park, IP5 3RE, UK
alex.healing@bt.com

Ben Azvine
BT Future Security Research
BT Research and Innovation
Adastral Park, IP5 3RE, UK
ben.azvine@bt.com

***Abstract*— As we move towards a zero-trust environment, collaborative intelligent systems are a vital tool in the cybersecurity workflow. These intelligent systems need to receive accurate and complete information from the multiplicity of low-level network monitoring systems. Typically, these low-level systems use regular expressions which are fast but not robust against minor changes - thus, information is often discarded. This paper outlines a new approach to low-level parsers, able to suggest new regular expression sequences and deal with approximate matching in an efficient way.**

Keywords*— *logfile, parser, cybersecurity, fuzzy regular expression

I. INTRODUCTION

We are moving towards a zero-trust environment in the digital sphere, in which a network is regarded as hostile until shown otherwise. In the past, network security has been assumed, and ensured by a range of tools such as firewalls (which use rules to permit or deny network connections), and intrusion detection / prevention systems (which use rules and pattern-recognition to identify, and possibly block, suspicious traffic). At higher levels such as social networks, other websites, email, filesystems, operating systems, etc. a host of more sophisticated tools analyse traffic and program behaviour with the aim of preventing malware, data loss, and related threats. Again, these systems rely on rules and simple pattern recognition to create the features for subsequent processing. In recent years, artificial intelligence (AI) has been proposed as an all-encompassing solution that will automate network security processes, aiming to match the security deployed in a system to the user, rather than to the network or device. Whether or not this is feasible, there is no doubt that AI is a vital tool in identification of anomalous events and sequences in networks, cutting out the 'noise' created by vast volumes of data.

At the start of the cyber-security monitoring chain, a variety of systems implement basic filtering operations and monitor traffic to generate the data for subsequent analysis. Regular expressions are an almost universal choice in analysing low-level logs, as they enable very fast recognition of patterns and extraction of key data fields. However, these low-level systems are inherently brittle, since they are based on strict patterns and data formats, and can discard large volumes of data if a tiny deviation from expected formats occurs. For example, when a monitored system changes it can cause a slight alteration in the generated data which might fall outside the expected range and hence no longer match the appropriate regular expression. Similar problems can arise when software is updated or other changes in configuration take place. In practice this often leads to data being discarded until the problem is resolved by manual re-configuration of the monitoring system. Initial set-up of monitoring systems (especially the need to manually create parsers) can also be labour-intensive, since regular expressions are often difficult to design and maintain - as recognised in the aphorism "*Some people, when confronted with a problem, think 'I know, I'll use regular expressions'. Now they have two problems.*"

In this paper we consider the problem of identifying and extracting data from a (mostly) formatted stream and propose a new approximate matching technique for identifying sequences of regular expressions. The method uses a fuzzy hierarchy (lattice) generated from the regular expressions, and is able to produce potential modifications to regular expression sequences from sample log data. The approach relies on three underlying themes - the graded (x-mu) approach to fuzzy sets [1], formal concept analysis [2, 3, 4] extended by the x-mu approach [5,6] , and a new approach described in this paper to calculate a graded distance, either between a logfile record (a sequence of fields, each represented as a string) and a sequence of regular expressions, or between two regular expression sequences. In the following section, we briefly introduce these underlying themes and subsequently show how the method can be used to categorise unseen logfile data and handle graded matching when data deviates from the expected format.

II. BACKGROUND

A. Problem Description

We focus on the initial analysis of log files and streams in the cybersecurity workflow, a vital step that provides data for subsequent analysis and classification of network events and trends. Logfiles vary in format, content and granularity, and examples include web proxy logs, firewall logs, IDS logs, netflow logs, DNS logs, etc.

Although content varies, these files and streams have a similar structure since they are time-stamped records of events in which various items of relevant data are stored (such as *id*s, *URL*s, IP addresses, flow data, messages, etc.). They are commonly processed in the first instance by regular

978-1-7281-6932-3/20/$31.00 ©2020 IEEE

expression-based parsers which can quickly and efficiently identify data types and extract key attributes into a centralised data storage system, for subsequent analysis.

Two significant bottlenecks in the acquisition of data are:

(i) the need to manually configure regular expression parsers for each data source, and for each variation in a data source. Monitoring systems are configurable, and are updated from time to time. Both factors contribute to changes in the format and content of data contained in the stream, requiring further manual configuration

(ii) systematic or random changes in the data from other causes (e.g. changes in the monitored system)

In both cases, significant loss of monitored data can occur when the data does not quite match the expected pattern. Table I shows an example of uniformly formatted data (note that headers/field descriptions are generally not available, but the nature of each field is "obvious" when a number of examples are considered). Table II shows more variation in format.

Because monitoring data is typically produced in large volumes and at high speed, it is impractical to store and reprocess records that do not match exactly - instead, they are usually discarded.

This work aims to:

- automatically create regular expression sequences that match samples of records from log data and are sufficiently general to identify and process additional records from the same source. These can be presented to a human expert for approval / alteration
- use the derived regular expression sequences to categorise new log data records (as a recognised log type) and identify components of the records
- allow a graded approach to the re-processing of records that cannot be categorised, by finding the closest sequence of regular expressions and considering the minimum number of changes required to make the record match that sequence.
- use rejected records to suggest updates to regular expression sequences for specific log streams.

B. *Regular Expressions*

Although regular expressions are widely used, there is no formal standard defining their exact syntax and behaviour (although partial standards such as POSIX exist). Consequently, there is an essential core of expressions, with various add-ons (see www.regular-expressions.info, regexr.com, etc for details). In this work, we assume the regular expressions follow a common "standard" and that there is an associated matching engine which, given a regular expression and a character string, returns true if the string is accepted by the regular expression and false otherwise. For consistency with other work (not reported here), we have used the java 9 regular expression engine docs.oracle.com/javase/9/docs/api/java/util/regex/Pattern. Note that the approach described in this paper is not dependent on the regexp engine.

We also assume a library of regular expression "primitives" representing basic data "types" such as *month*, *year*, *date*, *time*, *IP address, description, host name, mac address*, etc. (see regexlib.com for example). These regular expressions are the building blocks for the sequences. We further assume that each regular expression is associated with a tag indicating the data type recognised (see Table III). For readability we allow compound tags, built up from atomic tags or from other compound tags. For example, using *{ }* to delimit a tag, the compound expression

```
SHORTDATE := {MONTHDAY}-{MONTHNUM}
```

would match a string such as *23-09* or *12-10*.

A compound tag can be converted to a regular expression by substituting tags by definitions until all tags have been replaced (assuming no recursive or mutually recursive definitions).

Fuzzy matching of a string against a regular expression is possible by defining a membership function over the number of changes to the string necessary to make it acceptable to the

TABLE I. EXAMPLE OF UNIFORMLY FORMATTED LOG DATA

37.170.100.34	2018-01-01T09:40:29.276	37.170.100.200	80	7063	49 591
37.170.100.38	2018-01-01T09:43:08.861	37.157.76.124	80	5171	434 285
37.170.100.38	2018-01-01T09:47:41.282	37.170.30.250	25	32 818	182 798

TABLE II. EXAMPLE OF FORMATTED LOG DATA WITH VARIATIONS IN FORMAT

08-Sep-19	12:04:32	DNS Update Failed	10.0.3.151	XPM22.MAIN.COM	44:D3:CA:FD:7A:67
08-Sep-19	12:04:32	DNS Update Failed	10.0.3.151	MAIN	44:D3:CA:FD:7A:67
08-Sep-19	12:04:33	DNS Update Failed	10.0.3.151	XPM22.MAIN.COM	
08-Sep-19	12:06:29	Database Cleanup Begin			
08-Sep-19	23:49:56	DNS Update Failed	10.0.2.101	44:D3:CA:FD:7A:67	XPM206.MAIN.COM

POSINT	\b(?:[1-9][0-9]*)\b
MINUTE	(?:[0-5][0-9])
MONTHNUM	(?:0?[1-9]|1[0-2])
DATA	.*?
MONTHDAY	(?:(?:0[1-9])|(?:[12][0-9])|(?:3[01])|[1-9])
HOUR	(?:2[0123]|[01]?[0-9])
INT	(?:[+-]?(?:[0-9]+))
SECOND	(?:(?:[0-5][0-9]|60)

TABLE III. SAMPLE REGULAR EXPRESSION TAGS AND DEFINITIONS

regular expression (or the proportion of this number to the total string length). For example, the string *1024703809* matches the *INT* regular expression (Table III) perfectly; a string *10247o3809* requires a single character to be changed (o → 0) in order to match the *INT* regular expression. This string has a high, though not full, membership in the set of strings accepted as *INT* (see Fig. 1). A more sophisticated approach could weight different substitutions (so that replacing "o" with "0" or "I" with "1" would have a lower cost than, say, replacing "m" with a digit). The details are described elsewhere (e.g. [7]) - the important point is that there are well-established methods for approximate matching of strings to regular expressions. For our purposes, it is better to use the inverse function and focus on the set of strings that are accepted by a regular expression with at least a specified membership, so that

$$INT_1 = \{1024703809, \ldots\}$$

$$INT_{0.8} = \{1024703809, 10247o3809, \ldots\}$$

where INT_α is the set of strings accepted with membership α or greater. See [8] for further discussion. This approach enables us to deal with a crisp set at each membership level, and hence we can use standard (non-fuzzy) methods to process the data rather than having to rewrite/re-engineer code so that it can handle memberships with the data.

C. Regular Expression Sequences

For the purpose of analysing logfiles, we assume the data is record-based, with each record split into fields as shown in Tables 1 and 2. Records are not necessarily all the same length but have some commonality - for example, in Table II we could compare the first field of each record to a regular expression identifying dates such as

```
DATE:={MONTHDAY}-{SHORTMONTH}-{TWODIGITYEAR}
```

with *MONTHDAY* defined in Table III and appropriate regular expression definitions for the other tags. A first guess at a parser for the data in Table II might identify records of length 3, 5 and 6, each starting with the sequence *{DATE} {TIME} {DATA}*, followed (in records of length 5 and 6) by *{IP4ADDRESS} {HOSTADDRESS}*, etc. We assume appropriate definitions for tags that are not shown in Table III. Clearly there is a degree of judgment and intuition in this process, as other representations would also match the records - to take an extreme example, each field is a string of zero or more characters so would also be matched by *DATA*. A sequence of *DATA* fields would be too general for the records in Tables I and II as the pattern would match every record and would not differentiate the content at all. To automate the generation of parsers from data, we therefore seek a way of capturing a subset of patterns which matches the sample records without over-generalising.

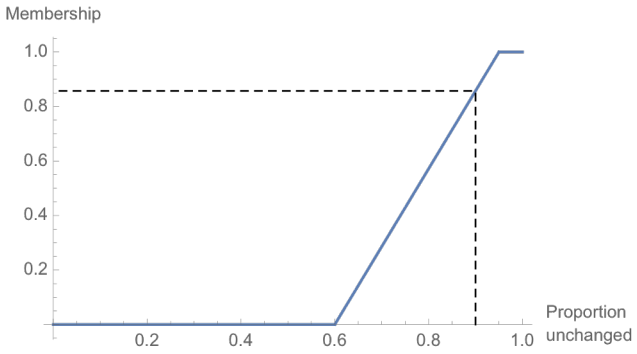

Fig. 1. An example membership function for fuzzy regular expressions. A string in which 9 out of 10 characters match the regular expression has membership 0.86

III. A FRAMEWORK FOR GRADED GENERALISATION

A. From Regular Expressions to Formal Concepts

Whilst regular expressions are powerful tools for string matching, their design and maintenance can be error-prone. In part this arises from the lack of a full standard, but is largely due to the compact representation using punctuation symbols and the context-sensitive nature of the symbols - for example, the ^ character can represent a start of line (e.g. *^[a-z]* matches a single lower case letter at the beginning of a line) or a complement operator (e.g. *[^a-z]* matches any single character except a lower case letter). Whilst experience and the use of predefined libraries can help, it is often difficult to identify the full set of strings accepted by a regular expression, and to see the relation between different regular expressions. For example, intuitively one would expect the set of strings accepted by *MONTHNUM* (above) to be a subset of those accepted by *MONTHDAY* which in turn is contained in *INT*, as is suggested by the example strings shown in the table.

As a more complex illustration, consider the following four expressions intended to represent different forms of MAC address

```
MAC1 (?:(?:[A-Fa-f0-9]{4}\.){2}[A-Fa-f0-9]{4})
MAC2 (?:(?:[A-Fa-f0-9]{2}-){5}[A-Fa-f0-9]{2})
MAC3 (?:(?:[A-Fa-f0-9]{2}:){5}[A-Fa-f0-9]{2})
MAC4 (?:(?:[A-Fa-f0-9]{1,2}:)?(?:(?:[A-Fa-f0-9]
    {1,2}:){5}[A-Fa-f0-9]{1,2}))
```

From the definitions, it is difficult to see the relation between the sets of strings accepted by these regular expressions. Testing would reveal (for example) that

abcd.1222.34f5	is accepted by MAC1
ab-cd-12-22-34-f5	is accepted by MAC2
ab:cd:12:22:34:f5	is accepted by MAC3 and MAC4
a:b:cd:12:22:34:f5	is accepted by MAC4

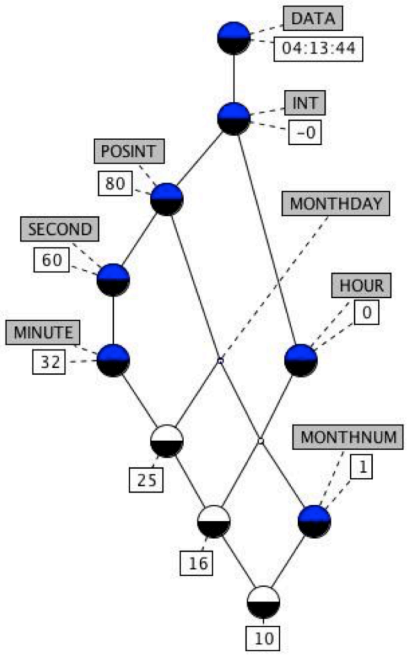

Fig. 2. Formal Concept Lattice corresponding to the context in Table IV

We might postulate that *MAC4* accepts a superset of the strings accepted by *MAC3*. A more structured way to approach this problem is by use of formal concept analysis (FCA) [2, 3] where we consider a set of sample strings and the regular expressions that accept each string. In FCA terms, the strings are objects and the regular expressions are attributes. Given a set of strings and regular expressions, each string (object) is associated with the set of regular expressions that accept it (attributes), and each regular expression is associated with the set of strings it accepts. This defines a formal context - a Boolean table in which rows correspond to strings and columns to regular expressions. Table IV shows a formal context for a set of strings (shown as row labels) and regular expressions (shown as column labels). The table entry is 1 (true) where the string is accepted by the regular expression and 0 otherwise. (NB the ISO *SECOND* includes 60, a leap second).

The formal context defines a lattice, as shown in Fig 2. Each node is a formal concept, defined by a set of strings (the concept extension) and a set of attributes (the concept intension) that accept all strings in the extension. To avoid repetition, the lattice uses reduced labelling [2, 3] in which a concept's intension is the set of regular expressions attached to the node and to all of its ancestor nodes; a concept's extension is the set of strings attached to the node and to all of its descendant nodes. For example, the node labelled

(*HOUR* , 0) has the extension {0, 1, 10, 16},

corresponding to the set of strings which all satisfy the regular expressions

HOUR, *INT* and *DATA* (the intension of the concept).

The lattice shows the subset/superset relation between the concepts - e.g. any string accepted by *MONTHNUM* will also be accepted by *HOUR* or *MONTHDAY*. The lattice also highlights differences between the regular expressions, e.g.16 is accepted by *HOUR*, *MINUTE* and *MONTHDAY* (plus the labels attached to its ancestor nodes) but not by *MONTHNUM*. With a suitable string generator, the lattice can reveal errors in regular expression definitions but this is not the focus here. Instead, we use the lattice to define a graded notion of generalisation for a regular expression when compared to a second regular expression.

For clarity, we restrict the context to single strings representing different concepts; in practice, there may be many strings that are accepted by a given set of regular expressions. Each concept extension has a size (cardinality) that can be found by considering a large sample of strings, or by analysis of the regular expressions themselves. We also note that each regular expression is associated with a node in the lattice, and that some nodes in the lattice do not have associated regular expressions. These correspond to conjunction and disjunction of regular expressions (or, equivalently, they correspond to the union and intersection of sets of strings). For any pair of regular expressions, R_1 and R_2, the lattice represents a partial order, i.e. we have one of the following cases:

$R_1 \geq R_2$ (R_1 accepts all strings accepted by R_2)

$R_2 \geq R_1$ (R_2 accepts all strings accepted by R_1)

or neither of these, so that R_1 accepts some strings that R_2 does not accept, and vice-versa.

Where two regular expressions are related under the partial

TABLE IV. FORMAL CONTEXT FOR A SAMPLE SET OF STRINGS AND THE REGULAR EXPRESSIONS IN TABLE III

	POSINT	MINUTE	MONTHNUM	DATA	MONTHDAY	HOUR	INT	SECOND
0	0	0	0	1	0	0	1	0
10	1	1	1	1	1	1	1	1
04:13:44	0	0	0	1	0	0	0	0
16	1	1	0	1	1	1	1	1
0	0	0	0	1	0	1	1	0
1	1	0	1	1	1	1	1	0
80	1	0	0	1	0	0	1	0
60	1	0	0	1	0	0	1	1
25	1	1	0	1	1	0	1	1
32	1	1	0	1	0	0	1	1

order, e.g. $R_1 \geq R_2$, we say that the expression R_2 is more specific than R_1, since it accepts some of the same strings as R_1 but not necessarily all (conversely, R_1 is more general than R_2)

Define R to be the set of all regular expressions in our library, $R = \{ R_1, R_2 \ , ..., R_n\}$, and R^* to be a subset $R^* \subseteq R$, then the greatest lower bound (GLB) and least upper bound (LUB) of R^* are unique and guaranteed to exist.

$LUB(R^*) = R_k$ such that $R_k \in R$ and $\forall R_j \in R^* : R_j \leq R_k$

$GLB(R^*) = R_k$ such that $R_k \in R$ and $\forall R_j \in R^* : R_k \leq R_j$

(in case of multiple upper / lower bounds, we take the least / greatest respectively). i.e. the LUB is the most specific regular expression that accepts all strings accepted by either R_1 or R_2 and the GLB is the most general regular expression that accepts all strings accepted by either R_1 or R_2

The top element of the lattice corresponds to a regular expression such as .* which accepts any sequence of characters.

B. *Minimal Record Labelling*

Definition : the *minimal labelling* of a record

$r = s1\ s2\ ...\ sn$

is given by $minLab(r) = R_1\ R_2\ ...\ R_n$

where each Ri is a regular expression defined by

$$R_i = GLB\left(\left\{E \middle| E \in R \wedge E \text{ accepts } s_i\right\}\right)$$

(s_i is the string value of field i)

Thus, for each field in the record, we choose the lowest regular expression that accepts the field.

C. *Graded Generalisation of Regular Expression Sequences*

Using the regular expression lattice, it is straightforward to take a record and label each field with the most specific regular expression that accepts the field. We simply need to find the greatest lower bound of all regular expressions accepting the string. In practice this search can be guided by the lattice, since if a string is not accepted by a regular expression, it will not be accepted by any descendant nodes.

Each sample record yields a regular expression sequence; our task is to find a small set of sequences that accept all sample records (and unseen records of the same logtype). To illustrate, assume we have a record containing three fields which represent an hour, minute and second. Examples might be

```
minLab(15,14,32)=
       HOUR∧MINUTE, HOUR∧MINUTE,  MINUTE
minLab(0,25,60)=
       HOUR,  MINUTE∧MONTHDAY,  SECOND
```

In this case the second sequence is the more general, and completely subsumes the first sequence (i.e. all records accepted by the first sequence are also accepted by the second). This is obvious by considering the pairwise least upper bound for each position in the sequences, since

$HOUR \wedge MINUTE \leq HOUR$

$HOUR \wedge MINUTE \leq MINUTE \wedge MONTHDAY$

$MINUTE \leq SECOND$

This relation can be denoted by extending the partial order to sequences of the same length, so that if

$S_1 = R_{11}\ R_{12} \ldots R_{1n}$

$S_2 = R_{21}\ R_{22} \ldots R_{2n}$

then

$S_1 \geq S_2$ *iff* $R_{11} \geq R_{21}$ and $R_{12} \geq R_{22}$ … $R_{1n} \geq R_{2n}$

We can discard the sequence S_2 as a candidate for recognising the log records since any sequences it accepts are also accepted by S_1

We also define a graded (or fuzzy) version which allows us to generalise a regular expression, based on extension cardinalities and a fuzzy proportion of inclusion. This allows us to replace two regular expressions R_1 and R_2 with their least upper bound, as long as the extension of R_1 is *most* of the extension of $LUB(R_1, R_2)$. Formally, given a membership function *most* , a threshold membership level α , an initial sequence

$$S_1 = R_{11}\ R_{12} \ldots R_{1n}$$

and a new candidate sequence

$$S_2 = R_{21}\ R_{22} \ldots R_{2n}$$

we generalise S_1 to

$$S_3 = LUB(R_{11}, R_{21})\ LUB(R_{12}, R_{22}) \ldots LUB(R_{1n}, R_{2n})$$

in the case that for all i, R_{1i} generalises R_{2i} at α, *most*

We denote this relation $R_{1i} \underset{\alpha, most}{\succcurlyeq} R_{2i}$, defined as

$$R_{1i} \underset{\alpha,most}{\succcurlyeq} R_{2i} \text{ iff } R_{1i} \geq R_{2i} \quad OR \quad most\left(\frac{|ext(R_{1i})|}{|ext(LUB(R_{1i}, R_{2i}))|}\right) \geq \alpha$$

For instance, assume an additional definition *NONNEGINT* in Fig 1, accepting non-negative integers. Given a pattern with *POSINT* and another with *HOUR*, we could generalise to *NONNEGINT* since this has almost the same extension as *POSINT* (differing only in accepting 0). However, we would not generalise to *INT* since this accepts (roughly) twice as many strings. There is an element of pragmatism here in comparing the number of strings accepted as INT with the number accepted as *POSINT*. Since we typically use a large set of sample strings, this is a valid comparison.

```
INPUT :  L, a sample of logfile records
         most, a fuzzy set representing acceptable ratio for generalisation
         α, a membership threshold
OUTPUT :outputSet, a minimal set of regexp sequences that accepts the logfile records

outputSet = empty set
FOREACH record, r
 FOREACH pattern sequence, S ∈ outputSet
  IF minLab(r) generalises S at (α, most)
    OR S generalises minLab(r) at (α, most)
  THEN
      replace S in outputSet by LUB(minLab(r),S)
  ENDIF
 ENDFOR
ENDFOR
IF r did not cause any change to outputSet
  add minLab(r) to outputSet
ENDIF
RETURN outputSet
```

Fig. 3. Algorithm to create general patterns from samples of logfile records

IV. Algorithm to Generate Regular Expression Sequences from Record Samples

The algorithm is shown in Fig 3. Each record in the sample is read and its minimal labelling is compared to the current set of pattern sequences. If an existing pattern sequence subsumes the minimal labelling, no further action is necessary - for example, if we have a pattern sequence:

HOUR, MINUTE, SECOND

and a record containing

15,14,32

with minimal labelling

HOUR∧MINUTE, HOUR∧MINUTE, MINUTE

then the existing pattern sequence is more general than the record. This is covered in the algorithm by the condition

S generalises *minLab(r)*

and the action

replace *S* in outputSet by LUB(*minLab(r),S)*

is (effectively) a null operation since it leaves *S* unchanged.

If one or more of the labels in the existing pattern sequence are graded generalisations of the corresponding labels derived from the record (as described in Section III.C), the existing pattern sequence is modified. Similarly, if the minimal labelling of the record is a graded generalisation of the pattern sequence, then the pattern sequence is modified. Otherwise, a new pattern sequence is added to the set when there is insufficient similarity to existing sequences.

In the worst case, this algorithm is quadratic in the number of records (if every record leads to a new pattern that can't be combined with any existing pattern). In practice most logtypes give rise to only a few patterns and the runtime is roughly linear in the number of records used for the learning phase.

V. Experimental Validation

We focus on two key features of the method - categorising unseen records as the correct logfile type (and hence extracting the correct data components) and minimising the number of regular expression sequences that are used for each logfile type.

We have used a set of public domain logfiles from sources including:

www.stratosphereips.org/datasets-ctu13

(a dataset of "botnet traffic that was captured in the CTU University, Czech Republic, in 2011. The goal of the dataset was to have a large capture of real botnet traffic mixed with normal traffic and background". This dataset is netflow data in CSV and TSV format)

csr.lanl.gov/data/2017.html

(a dataset of (some) network and computer (host) events collected from the Los Alamos National Laboratory enterprise network)

log-sharing.dreamhosters.com

(multiple log samples from various security and network devices and applications, mostly unmodified and all collected from real systems.

vast challenge datasets (www.vacommunity.org)

(various synthetic datasets in a number of formats)

	bluecoat	netflow2-CSV	netflow1-TSV	IPLog	stratosphere	snort	cisco Firewall	DHCP	Rejected	read Failures
firewall_log_1.csv	0	0	0	0	0	0	90190	0	9810	0
	0	0	0	0	0	0	95623	0	4377	0
	0	0	0	0	0	0	95650	0	4350	0
firewall_log_4.csv	0	0	0	0	0	0	99924	0	76	0
	0	0	0	0	0	0	100000	0	0	0
	0	0	0	0	0	0	100000	0	0	0
IPLog3.5.csv	0	0	0	100000	0	0	0	0	0	0
	0	0	0	100000	0	0	0	0	0	0
	0	0	0	100000	0	0	0	0	0	0
bluecoat1.log	94150	0	199	0	0	0	0	7	5830	13
	95380	0	199	0	0	0	0	7	4600	13
	95385	0	199	0	0	0	0	7	4595	13
stratospher.log	0	0	0	0	46689	0	0	0	0	0
	0	0	0	0		0	0	0	0	0
	0	0	0	0		0	0	0	0	0
DhcpSrvLog-Sun.log	0	0	0	0	0	0	0	10492	195	0
	0	0	0	0	0	0	0	10492	195	0
	0	0	0	0	0	0	0	10492	195	0
alertcsv.csv	0	0	0	0	0	99923	0	0	77	0
	0	0	0	0	0	99923	0	0	77	0
	0	0	0	0	0	99923	0	0	77	0
capture20110818.tsv	0	0	99946	0	0	0	0	0	54	0
	0	0	100000	0	0	0	0	0	0	0
	0	0	100000	0	0	0	0	0	0	0
capture20110810 .binetflow	0	99646	0	0	0	0	0	0	4	0
	0	100000	0	0	0	0	0	0	0	0
	0	100000	0	0	0	0	0	0	0	0

TABLE V. NUMBER OF RECORDS CATEGORISED BY EACH LOGFILE PARSER. EACH CELL SHOWS A VALUE (FROM TOP) WITH α=1, 0.8 AND 0.6 AS SHOWN IN THE ALGORITHM (FIG 3). TESTS ARE OVER THE FIRST 100000 RECORDS READ, EXCEPT IN CASES WHERE THE FILE CONTAINS FEWER THAN 100000 RECORDS

These logfiles vary in size from a few thousand to several million records; for test purposes below we use the first 500 records to create logfile parsers and (up to) 100K records to test the categorisation performance.

Results are given for alpha = 1, 0.8 and 0.6 to illustrate the balance between accuracy in recognising logfile types, and compactness of the derived regular expression patterns.

Table V shows the categorisation performance of the parsers. The logfiles are completely unchanged from the downloaded versions, some of which are not cleaned at all - hence there are some unreadable records containing control / extended ASCII characters as indicated in the final column. Execution times are of the order of a few seconds on a mid-range laptop (exact times are not given - clearly, with any experimental software, run time can be improved significantly by re-engineering code and using faster hardware).

Table VI gives the number of patterns for each record length at α = 1 and 0.8. The *bluecoat* data in particular shows various record lengths. This is due to the presence of records periodically giving timestamped information on software versions and monitor configuration - in particular, switching between records of length 23 and 26. It is noticeable that the number of patterns from records of length 23 and 26 is reduced significantly by allowing greater generalisation of fields (i.e. using alpha of 0.8 instead of 1, see algorithm in Fig 3). A

Logtype	L_1 C_1	$L_{0.8}$ $C_{0.8}$
bluecoat	2 1	2 1
	3 2	3 2
	23 2	23 1
	24 1	24 1
	26 13	26 5
	27 1	27 1
netflow2-CSV	15 9	15 8
netflow1-TSV	2 1	2 1
	12 1	12 1
IPLog	7 2	7 2
stratosphere	17 2	17 2
snort	27 2	27 2
ciscoFirewall	15 2	15 2
DHCP	1 1	1 1
	7 1	7 1
	8 4	8 4

TABLE VI. LENGTH (L) AND NUMBER OF PATTERNS (C) FOR EACH LOGFILE TYPE AT α=1, 0.8.

TIMESTAMP_ISO8601	*BASE16FLOAT*	*URIPROTO*	*URIHOST*	*NOTSPACE*	...
TIMESTAMP_ISO8601	*BASE16FLOAT*	*SYSLOGPROG*	*MAC3*	*NOTSPACE*	...
match	*match*	*Generalise*	*Replace/combine*	*match*	...
2011-08-18 10:19:17.534	*0.000*	*IPX/SPX*	*0:15:17:2c:e5:2d*	*->*	...

TABLE VII (PART) RESULT OF APPROXIMATE MATCHING REJECTED RECORD (SHOWN IN THE BOTTOM ROW) AGAINST THE NEAREST MATCHING PATTERN IN THE PARSER (TOP ROW). THE SECOND ROW IS THE MINIMAL LABELLING OF THE RECORD, AND THE THIRD ROW HIGHLIGHTS WHERE THE LABELS MATCH OR THE PATTERN LABELS NEED TO BE CHANGED IN ORDER TO ACCEPT THE RECORD

possible extension to this work would look at the sequencing of records with different lengths (for example using the mechanism described in [9])

The Cisco firewall data (first row of Table V) shows the poorest level of performance because this particular log contains a number of records in a format not seen in the data used to create the parser patterns.

Importantly, the rejected data from each logfile source is saved and can be used to suggest new or updated versions of the parser patterns. This process can be fully-automated or part of a collaborative system in which the human analyst can judge whether or not the suggested change is reasonable. For example, the automatic analysis of the rejected Cisco firewall records shows a significant number matching the alternative pattern depicted in Table VII. An analyst can decide whether this is a valid pattern and should be identified as an addition to the parser, i.e. whether it should be combined with the existing pattern by generalising the *URIHOST* tag to an alternative tag matching *URIHOST* or *MAC3*

VI. RELATED WORK

Many studies have focused on the problem of learning regular expressions from data, and the related problem of learning regular expressions to match sequences.

The difficulty of designing good regular expressions is illustrated in [10] where regular expression design was described as a "highly non-trivial task" and used to illustrate the (potential) benefit of crowd sourcing as a solution. In a similar vein, [11]designed a system to generate test strings for regular expressions, focusing on areas that typically cause problems in regexp design and using a number of heuristics to create strings that (semantically) should not be accepted but would match the given definitions. They found more than 260 errors in a set of almost 700 regular expressions taken from a standard regular expression library (regexlib.com), including a regexp for a floating point number that would accept a single decimal point with no digits.

Many studies have focused on the problem of learning regular expressions from data, although this is an NP-complete task in principle. Regular expressions are frequently used for sequence extraction in DNA - see for example the work of [12] which outlines an approximate matching mechanism for regular expressions involving alternation (the | symbol).

Another application of regular expressions to sequences [13] examined a regular-expression based approach to identifying sequences of events (in testing GUI) and found it to give comparable performance to an SVM-based classifier.

We are not aware of any similar work in the area of logfile analysis.

VII. SUMMARY

There is immense scope for the use of collaborative (human + machine) intelligent systems in monitoring and analysing logfile data to maintain security of networked systems. We have outlined a new approach to the first stage of the cybersecurity workflow, gathering the low level data from logs, prior to its analysis and transformation into event sequences and higher-level structures. The approach follows the collaborative intelligence paradigm, in which the machine does the initial work and human judgment / insight can refine the results.

Further work is underway to improve the mechanism for graded generalisation and to develop an interface so that this tool can be used in practical cyber-defence applications.

REFERENCES

[1] T. P. Martin, "The X-mu representation of fuzzy sets," *Soft Computing,* vol. 19, pp. 1497 - 1509, 2014/05/31 2015.

[2] B. Ganter and R. Wille, *Formal Concept Analysis: Mathematical Foundations*: Springer, 1998.

[3] U. Priss, "Formal Concept Analysis in Information Science," *Ann Rev of Information Science and Technology,* vol. 40, pp. 521 - 543, 2006.

[4] R. Belohlavek, V. Sklenar, and J. Zacpal, "Crisply Generated Fuzzy Concepts," *Lecture Notes in Computer Science,* pp. 269-284, 2005.

[5] T. P. Martin, "Change mining in evolving fuzzy concept lattices," *Evolving Systems,* vol. 5, pp. 259-274, 2014/06/21 2014.

[6] T. P. Martin, "Representation of Fuzzy Concept Lattices," in *UK Computational Intelligence (UKCI), 2015*, Exeter, UK, 2015, pp. 1-8.

[7] T. P. Martin, Y. Shen, and B. Azvine, "Incremental Evolution of Fuzzy Grammar Fragments to Enhance Instance Matching and Text Mining," *IEEE Transactions on Fuzzy Systems,* vol. 16, pp. 1425-1438, 2008.

[8] T. P. Martin, "The X-mu representation of fuzzy sets," *Soft Computing,* vol. 19, pp. 1497-1509, 2015.

[9] T. P. Martin and B. Azvine, "A Virtual Machine for Event Sequence Identification using Fuzzy Tolerance," in *IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, Vancouver 2016, pp. 1080-1087.

[10] R. A. Cochran, L. D'Antoni, B. Livshits, D. Molnar, and M. Veanes, "Program Boosting: Program Synthesis via Crowd-Sourcing," presented at the Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Mumbai, India, 2015.

[11] E. Larson and A. Kirk, "Generating Evil Test Strings for Regular Expressions," *2016 9th Ieee International Conference on Software Testing, Verification and Validation (Icst),* pp. 309-319, 2016.

[12] P. Powell, "RESIM-An Algorithm for Finding the Similarity of Regular Expression Based Patterns and Strings," *Asilomar Conference on Signals Systems and Computers,* p. 283, 1992.v

[13] R. Gove and J. Faytong, *Identifying Infeasible GUI Test Cases Using Support Vector Machines and Induced Grammars*, 2011.