Abschlußarbeit im Studiengang

Master of Science in Informatik

Efficient Normalization of IT Log Messages under Realtime Conditions

Rainer Gerhards

Mat.-Nr. 7033990

Themensteller: Prof. Dr. Wolfram Schiffmann

Betreuer: Prof. Dr. Schiffmann

Lehrgebiet Rechnerarchitektur

Fakultät für Mathematik und Informatik

Datum der Abgabe: 27. April 2016

Table of Contents

Abstract

In computer log analysis, normalization of a large variety of different message formats is often required. Ideally normalization is done in real-time. Traditionally regular expression based methods are used, but that approach is typically too slow for real-time normalization.

We show a novel normalization method based on an analysis and classification of log message formats and consideration of existing normalization approaches. Our method is evaluated both by theoretical reasoning and experiment. We show that while theoretical worst-case complexity is not considerably better than that of existing approaches, practical performance is superior. We provide backing theoretical argument why this is the case. Our method offers sufficient performance to handle large workloads under real-time conditions and is easily and efficiently parallelizable for even larger workloads.

1 Introduction

Computer log messages are important for many use cases including IT system operation management, IT security, (telecom) billing, and criminal investigation, to name a few. Log messages are generated by programs running both on regular as well as embedded systems. Typical log sources are general purpose operating systems (like Linux or Windows), applications installed on these systems (like mail servers or office applications), and devices like firewalls, routers, and switches. Unfortunately, log messages from different sources have very diverse formats, many of which look like free text, are not documented, and are hard to interpret automatically. Sometimes the format of messages is even different in different versions of the same source.

While for some use cases it is acceptable to store and process log messages "as is", many others require the transformation of the various incoming log message formats into some common format. We call this process "log (format) normalization" and it is especially important for applications like Intrusion Detection Systems (IDS) which need to handle messages from a diverse variety of log sources.

There have been various efforts to standardize log formats at the message generating software and make normalization obsolete. For example, a recent (2014) effort is Mitre's CEE [13]. Unfortunately, none of these efforts were able to attract sufficient attention. Instead, each just added to the format variations instead of reducing it. So it can be concluded from history that we need to deal with those format differences rather than trying to unify them at the message source, which would require change to all applications generating log messages. This is the core motivation behind log normalization.

The classical approach to normalization is to use a set of regular expressions (regex) to transform incoming log messages, where data from the message is extracted by sequentially applying the regexes until a match is found. That matched data is then transformed to the normalized output data. One of the earliest projects to employ this method was the "logcheck" tool [50]. According to source file copyright statement, "logcheck" was begun in 1996. It still is an active project today. The regex method used by it is also still state of the art. For example, the popular Logstash [8] log processing tool uses a system called "grok" [7] for its log normalization. Grok bases on the idea of sequentially iterating regexes. Grok has become very popular in recent years.

While regex-based solutions extract data reasonably well, it is known that they are computationally intense. In practice, many users complain that their runtime requirements make them unsuitable for large classes of applications, at least in en-

terprise environments with a large log volume. For example, Security and operation management tools usually require processing incoming log messages in realtime or at least very close to realtime. This is usually impossible to do with regex based approaches.

To solve that problem, two different approaches have been used: the traditional one is to hardcode parsers for each log format. Due to the wide variety of formats, this is only possible for very important formats or formats specifically requested by users. Nevertheless, this seems to be the approach taken by many commercial software packages. Note that we cannot proof this, as the source code for those packages is not available. But from private talks we had with developers, it is highly likely they take this approach.

An alternative approach, developed around 2010, is to use advanced data structures suitable for fast parsing. Details are given in section 2 "Related work". These normalizers offer a good compromise between speed and flexibility, and are in most cases useful for near-realtime normalization. However, there are still open questions: Memory consumption can become quite large, runtime can deteriorate and parallelization is not formally researched. Most importantly, none of these approaches have been published or formally documented.

Also, no formal survey of log formats or a classification of log format types exist. This is unfortunate, as knowing formats is fundamental to normalizers, yet still normalization methods base on "common understanding" rather than analysis. With this thesis, we will create a classification of log message formats. In support of this, we create a publicly accessible repository of log messages for research purposes.

We will develop an algorithm capable of realtime normalization that is parellizable in a well-defined way. Furthermore, we will create a prototype implementation of that algorithm and do experimental verification of our findings. The prototype also is of practical relevance, as it will be usable by millions of users worldwide as part of the rsyslog project [25] (the default Linux syslog daemon).

The reminder of this thesis is structured as follows: Section 2 discusses related work. Section 3 introduces basic concepts and definitions. Section 4 analyzes log formats. Section 5 describes the log normalization problem and explains normalization algorithms. Section 6 describes our improved normalization method. Section 7 describes the experimental verification of our method. Section 8 explains the results gained by it. And Section 9 provides the conclusion and an outlook for further work. Detail data of the experimental verification can be found in the Appendix.

2 Related work

Unfortunately, there are few publications on the topic of log message normalization itself.

Among others, we tried to research on this topic at the ACM digital library, CiteSeerX, the electronic catalog of the university as well as the SANS institute, Google Scholar and regular web search engines. Please note that Google Scholar includes results from IEEE, Springer and Elvesir publications, so they were included in the research. Search terms used were "log normalization", "computer log normalization", "log canonicalization", "computer log canonicalization" and other terms related to this topic. We also followed citations of the relevant syslog RFCs.

Very few results appeared, and the fast majority of them either referenced to standardization efforts (like CEE [13], DMTF [15], or SIP CLF [44]) or structured vendor formats like (WELF [33] or GELF [28]). These results discuss properties of the respective structured format and some also talked about the need of converting free-form log messages to structured format. However, most gave no information on that conversion process other than that it is desirable. Some vendor links, especially from Security Information and Event Management (SIEM) vendors, claim that their respective product can do this conversion based on proprietary technology. Again, no details were given on that process, but it commonly looked like they either used a hard-coded parser approach for each format or a regular expression based one.

One of the best sources of non-academic community information on logging formats and tools was the www.loganalysis.org web site [5]. Unfortunately, it was abandoned some time after 2005 and is no longer online. Some content is still available via the "Internet Archive Wayback Machine" [4]. Of course, that does not cover any recent developments.

Some information was available from related topics, namely the clustering and correlation of log records. A notable academic source of information is R. Varaandi's PhD thesis "Tools and Techniques for Event Log Analysis" [53] and associated research papers. Unfortunately, it does not cover the specifics of the log normalization process nor does it provide a precise description of the structure of log data.

Generally, descriptions of the nature of log data are sparse. Most papers simply imply that a log message is a single line of text, treated as a string. Or they quote standard formats, most notably RFC3164 or the RFC5424 series. However, all these RFCs describe the encapsulation format, but not the actual free-form content format. Concrete descriptions can be found in Vaarandi [52][Sect. III a], which is coarse, and in an early work of ourselves [21], which goes into some more detail.

Three larger projects exist which address the log normalization topic directly: "libgrok" [48] ("grok" in the following), "syslog-ng pattern db" (syslog-ng in the following), and "liblognorm" [20]. The latter two aim at real-time normalization.

Grok is an open source project. It is part of the popular Logstash [8] logging solution. No academic paper on it exists, but its regular expression based normalization approach is well described in the user manual. To gain more insight into the actual algorithm, we have analyzed the relevant source code [16, commit hash ecb6f358547952]. We concentrated our analysis on the code that builds the regular expressions.

The syslog-ng project is open sourced and also available as a feature-extended commercial solution. No paper is published over the project or the exact algorithm it uses. Its marketing brochure indicates it uses a "longest prefix match radix tree" [46], but no further details are available anywhere. To gain more insight into the method, we did a source code review based on the open source version [47, commit hash 99b3ee346443ae]. Reviewing the full source code would have been too time-consuming, so we concentrated on the code most probably associated with search tree handling. It resides in the file path "./modules/dbparser" and the most relevant code in files "radix.[ch]". The source code is mostly uncommented and contains no description of the algorithm. Syslog-ng uses a search tree based approach at normalizing.

The liblognorm project is open source. It was developed by us and the existing v1 version is a Proof of concept (PoC), but in production quality. It is being used frequently as part of the popular rsyslog [25] logging solution. No paper has been published about it, but we obviously know the algorithm details. The current form will be replaced by what is being developed in this thesis. Liblognorm uses a search tree based approach for normalization.

In a broader sense, the log normalization problem is a specialized text search problem. As such, the full tool set of text search algorithms is related to this work. Also, theory on languages and finite automata provides good ideas and solutions for log normalization.

3 Definition of Terms

Terminology in logging has a large number of variations with important slightly different meanings. In order to avoid ambiguity, we define even base terms precisely.

3.1 Base Terms

A *byte* is a storage unit which is used to store integers in the interval [0..255]. In data communications, this is often called an "octet". A *character* is the basic unit of information. For example "0", "a", "ä", and "α" are characters. A *code table* maps characters to integer values. A *(character) encoding* describes how code table values are mapped onto byte sequences. Many different encodings exist. Important ones in the western hemisphere are US-ASCII [9][1] and Unicode [51]. A *single byte character encoding* always uses a single byte to encode all characters supported by the encoding. US-ASCII is an example. A *multi byte character encoding* uses one or more bytes to encode characters supported by the encoding. Unicode is an example of such an encoding. Note that US-ASCII is part of Unicode and all US-ASCII characters are encoded in Unicode with exactly the same values as in US-ASCII. As such, all these characters use only a single byte. This also means that from a character value alone (in the US-ASCII range) one cannot deduce whether US-ASCII or Unicode encoding is used. A *single byte character* is a character that is represented by a single byte. A *multi byte character* is a character that is represented by two or more bytes. Note the difference to the definition of "character encoding". Here, a value inside the US-ASCII range is clearly defined as a "single byte character" regardless of the encoding used.

An *invalid character* is a sequence of bytes that does not properly correspond to a given encoding and thus cannot be assigned a character. A *control character* is a character that is intended to control the processing of other characters inside a string. For example, it can be used to control rendering (like US-ASCII LF) or data transmission (like US-ASCII ACK). In logging, the most important control characters are the US-ASCII characters with byte values $\{0, 1, \ldots, 31, 127\}$. Additional control characters exist in other encodings. A *printable character* is a character that is not a control character.

Let $\Sigma := \{0, \ldots, 255\}$ be the alphabet of byte values. A *string* $S := (s_1, s_2, \ldots, s_n)$ is an element of Σ^*. Note well that this definition permits arbitrary byte sequences, including invalid characters and incomplete multi-byte characters (these can happen in logging due to truncation).

The length of a string S is denoted by $|S|$. The string ϵ with $|\epsilon| = 0$ is called the *empty string*. In logging, we often need to work with parts of strings. As such, we

[1] We cite RFC20 rather than the original ASA standard X3.4-1963 because RFC20 contains additions more relevant to logging.

define *substring* $S_{i,j} := \begin{cases} s_i \dots s_j & \text{if } \forall i,j : 1 \leq i \leq j \leq n \\ \epsilon & \text{otherwise} \end{cases}$, and as special cases the *prefix* with $S_{0,j}$ and *suffix* with $S_{i,|S|}$.

We often need to split strings into substrings. This is frequently done based on specific bytes. A *word delimiter* is a byte that is used to indicate the begin and end of substrings. In log text, this is typically the US-ASCII SP character. Often more than one byte value is used for this purpose. Such a set D is called *word delimiters*. A *word* is a substring $S_{i,j}, \forall i \leq k \leq j : s_k \notin D$ inside a string that is delimited by word delimiters. A *subword* is a substring inside a word that is not delimited by word delimiters.

A *line* is a concept based on a line in printed publications. All characters of a string are "printed" at the same vertical position: let S be the string that is to be "printed" in one line on a two dimensional coordinate system with start coordinates (x, y) for the print operation, than s_i is printed at $(x + (i - 1), y)$

A *line terminator* is an operating system specific indication of the end of a line (where for the next character the y-coordinate needs to be advanced to $y + 1$ and the x-coordinate be reset to 0). Typically, control characters are used to indicate line termination. In Linux and Unix, US-ASCII LF is the line terminator, whereas under Windows the two-byte sequence CR LF is the line terminator.

3.2 Logging

A *log message* is a string that was created with the intention to log data. It can be sub-classed depending on the presence of line terminators:

- A *single line log message* is a log string that does not contain line terminators. This is the form usually expected by logging tools, like the Linux text processing tool-chain[2].

- A *multi line log message* is a log string that potentiality includes line terminators. As such, it is somewhat uncertain where a log message inside a file begins and where it ends. In practice, this is solved by using regular expressions to describe either the begin or the end of the log message (this varies by tool). For example, lines starting with US-ASCII SP characters are called to be "indented". Such indented lines may be treated as part of a multi line message which started by the first non-indented log message. Often, multi-line log messages are preprocessed and transformed into single line log messages.

[2]tools like grep, sed, wc, awk

This is, for example, usually done by replacing the line terminators by special character sequences, e.g. converting LF to "\n" [6] or "#012" [3].

In order to avoid the problems associated with multi-line log messages, we require that multi-line log messages be converted to single-line log messages before normalization. All of the methods described in this thesis will work on single-line log messages, only. As such, we will use the term *log message* as a synonym for single line log message in this thesis.

A *motif* is a substring inside a log string that is frequently being used and has a specific syntax and semantic (e.g. an IPv4 address). The term is based on the idea of "sequence motif" in bioinformatics [38, pg. 183]. A motif may spawn multiple words but may also be a subword.

A *motif parser* (also called "parser" for brevity) extracts substrings matching a given motif from log messages.

A set of log messages is called a *log data set*.

Logs are being processed by applications. These are usually called *(logging) tools* and we will use that nomenclature in this thesis as well. A logging tool is any program that is used to process log messages. This does not require that the program was originally written for log processing. Under unixoid operating systems, for example, many generic text processing applications (like "grep" or "sed") are often used as logging tools.

The *log processing chain* (also known as *tool chain*) is a sequence of programs that are coupled together in order to process log. Various ways exist to couple tools. Some important ones are:

- One tool writes an output file, which is read by the next tool as input. The tools may be executed immediately after another or some time may be left between.
- A variant of this method is using the Unix "pipe" method, where tools are executed concurrently and one tool's output is immediately forwarded to the other as input.
- Log messages are exchanged via the network.

Tools and systems can be classified according to their position inside the tool chain: *originators* are log sources, they originally create the log record, based on events that happen. Typical originators are firewalls, which generate traffic flow

[3] Using '#' followed by the octal representation is somewhat suggested by [23, Sect. 6.3.3].

messages, or authentication processes, which generate logs about successful and unsuccessful login attempts.

Relays forward log messages from some source to some destination. They may or may not apply transformations while relaying.

Consumers consume log records and do not transfer them to any other system. They form the tail end of the tool chain. Typical consumers are IDS systems or file stores (where log records be stored e.g. for legal reasons).

With these definitions a log chain can be more precisely defined as a directed graph where the nodes represent tools and the edges represent data flow between the tools. Originator nodes have an indegree of zero, whereas consumers have an outdegree of zero. Relays have non-zero in- and outdegrees.

In practice, the log processing chain usually serves multiple use cases. A single use case is usually represented by a single consumer o. This consumer induces a subset of the log processing chain specific to the use case. It consists of all nodes on all pathes that terminate in o. In this thesis, we always speak of such use case specific log chains, except when otherwise noted.

3.3 Realtime Requirements

For many use cases it is important to have log messages available in realtime or at least close to realtime. One obvious example is intrusion detection: in order to counter an attack, one must know as soon as possible that an attack is underway.

The IDS system is the consumer inside the log processing chain. For obvious reasons, all tools inside the chain must be tied tightly together and each one pass its output on to the next one as quickly as possible.

A typical intrusion detection tool chain consists of originators, one or more levels of relay, a collector, a normalization tool and the IDS (example in Fig. 1). Note that the normalization tool can also be placed on relays, and sometimes it is part of the IDS. The IDS often is also able to work on unnormalized data, in which case it is expected to work on meta data and a subset of events that it knows how to normalize (exact details are not available as the majority of major IDSes are closed source).

To specify the real time requirement precisely, we need to define two properties: latency and sustained rate.

With *latency* we mean the time it takes for a single log message to be transferred from the originator to the consumer. We further include the time the consumer requires to process the message. Latency is influenced by many factors, with the

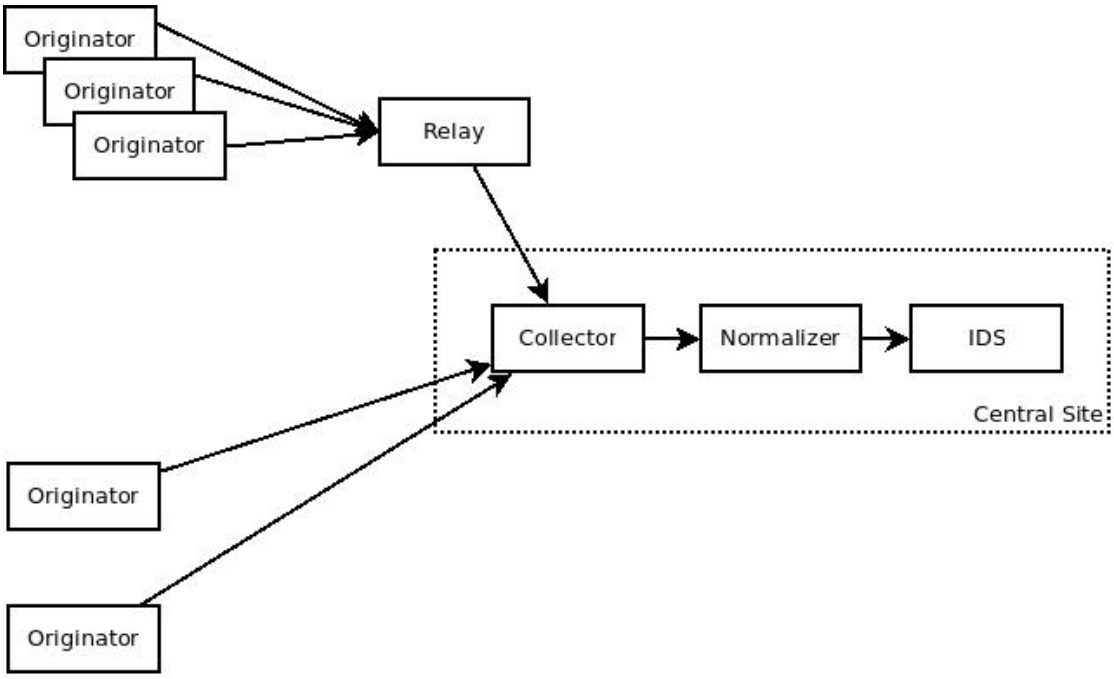


Figure 1: A typical workflow in log processing.

most important ones being available network connectivity, bandwidth and speed as well as the time normalization requires. Latency is usually measured in time units, in this context often in seconds.

The *sustained rate* is that rate at which the overall system (from originator, through network, through other processing steps, to the final consumer) is able to process all generated messages without building up queues or notably increasing latency. The maximum sustained rate is the rate at which messages can be generated and being processed successfully over a long period of time. The sustained rate is measured in messages per second. The *expected maximum sustained rate* is the sustained rate that a user expects to occur within the system. This is in contrast to the *actual maximum sustained rate* that describes the processing capability of a system.

In practical use, occasional *traffic spikes* may occur and they may be higher than the actual maximum sustained rate. Usually the system buffers excess messages until they can be processed. If so, these spikes manifest as an increase in latency. As long as latency is not increased above the point the end system (e.g. IDS) requires, traffic spikes are unproblematic. Some systems may be forced to discard traffic spikes, what may be problematic (this can happen for example with syslog messages transported over UDP). In order to be able to handle traffic spikes sufficiently well, the maximum sustained rate of a log processing chain must be higher than the expected maximum

sustained rate.

The log normalization processor is an important part of the processing chain, and it can cause considerable latency. Message consumers typically expect a latency of several seconds (which is inevitable due to network traffic in any case), and may even tolerate some minutes of latency during traffic spikes. However, normalization processing also limits the maximum sustained rate, and this is the core problem behind realtime processing requirements.

A typical large installation generates several ten- to hundred-thousand messages per second. If an intrusion actually happens, one can often see a traffic spike, which should not immediately be increasing delay. As such, the systems need to have a sufficiently higher maximum sustained rate than the rate at which regular traffic is coming in. For attacks, this can mean at least twice the regular traffic rate.

For the log normalization process it means it must be fast enough to keep up with the expected maximum sustained rate, while not increasing the latency by more than a few seconds.

That requirement makes log normalization unavailable for real time processing when slow normalization algorithms are used.

4 Understanding Log Message Formats

In order to understand the need for a log normalization process and its limits, we need to understand log message formats as they exist in practice.

4.1 Format Variations

Log messages are generated by software programs. For obvious reasons, these have been developed by different companies and by different developers in different years. As usual in software engineering, standards and development trends change over time. Developing software is costly and consequently many parts of programs remain unchanged for a long time if there is no hard need to change them. Furthermore, logging formats are especially hard to change, because some consumers may depend on a specific format. Also, different companies have different development standards, and different developers have different educational background. Even worse, the logging format may change depending on user-selected configuration. For example, Cisco ISO devices optionally include a sequence number into log messages if configured to do so [10, pg. 29.8].

As said in the introduction, this leads to a large variety of different logging

formats. The lack of a well-accepted logging standard format is another reason for the large number of different formats. The only common ground is that a log message is a string, thus we use this definition.

As such the logging format depends on the application a generating the log message, but it also depends on the application's version v as well as its configuration k. So the triple (a, v, k) describes the log format being used to generate a specific message. Note that the mapping of a format is not bijective: the same log format may be generated by different (a, v, k) triples. In theory, two (a, v, k) triples generating the exact same format may convey different information. In practice, this case has not been seen and would be considered misleading or erroneous. So in the context of this thesis, we assume that (a, v, k) triples generating the same message will also convey the same information.

The set of potentially different formats inside an installation is finite but can be large. Let $\mathcal{L}$ be the set of all log messages seen at an installation and let $\mathcal{F}$ be the set of all message formats seen in $\mathcal{L}$. Then the function

$$\text{mfmt} : \mathcal{L} \mapsto \mathcal{F}$$

maps each $l \in \mathcal{L}$ to the format it was created with. Note that the format is usually not explicitly included and the message and thus must be deduced based on the message content itself.

4.2 Research Data Sets and Repositories

In order to analyze log message formats, real-life sample logs are very useful. There have been some efforts to create repositories of those, for example initiated by Balabit, the company that sponsors syslog-ng development [45]. We could not find any result of this effort. There has also been a very small repository at the loganalysis.org [5] web site (now offline and only available via the Internet Archive as cited). Another small set of examples can be found at the MonitorWare web site [22].

Most "log samples" that can be found online are actually not log messages but packet captures (in PCAP file format), mostly used for IDS challenges. They can be found at various sites, for example at NETRESEC [1]. PCAP files contain network packet captures, and usually only very few, if any, actual log messages. As such, there is little use in trying to mine these data source as the outcome is very little.

All in all, no easily accessible and sufficiently large repository with log samples for research existed when we began working on this thesis. So in order to support our own work as well as that of followers, we built our own as part of this thesis.

The goal of our repository, named "log samples for research", is not to have well-documented logs; the goal is to have logs at all. So if a format is used in practice, we would like to have a sample of it inside the repository, even if we do not know what exactly it means. Similarly, it is best but not mandatory to have large samples, because that enables us to use statistical methods to analyze the content. Also, a large variety of real world log samples is useful.

Unfortunately, it is very hard to obtain real-world log samples. A major concern is privacy, which often blocks potential contributions. A solution is anonymization, but this needs to be done by the contributor himself. That puts a burden on the potential contributor, which often causes unwillingness to contribute. Also, good log anonymization tools are not readily available.

So we ended up populating the repository with larger samples from our own systems, which are sufficiently anonymized. We also included smaller log samples that we got contributed. All of this is published as a publicly available git archive [24]. We plan to continue to maintain and promote the repository when this thesis work is concluded and hope the repository will become an important tool for researchers in the logging field.

In order to prove some points in our thesis, we amend the public repository by some large real-world log data sets which are not fully anonymized and are not available for publishing to the general public. However, upon request these data sets are available for researchers. In the medium term, we also plan to anonymize them and add them to the public repository (we have discussed this with the contributors).

The main data sets used for this thesis are:

- *PIX data set* - a real-world sample of log messages generated by a Cisco PIX device [26]. It contains roughly 2 million messages. This is a very good example for firewall syslog messages and includes many different variants of descriptions of traffic flow as well as host address formats.

- *enterprise combined data set* - this is a real world data set with combined log records from many sources typically found in large enterprises. It contains router and firewall logs, Windows logs, Linux logs, WiFi logs, and logs from various applications. Note that this data set contains all classes of logging formats. This data set contains nearly 56 million log records and is 27GiB in size. This data set is not yet included in the public repository but is the one that is available upon request from researchers.

4.3 Format Classification

In this chapter, we describe formats commonly found in practice and describe how they can be normalized.

There are three main classes of log formats

- free-text log formats
- semi-structured log formats
- structured log formats

Each of these will be described in the following subsections.

As we will see below, log messages can generally be viewed as a sequence of motifs (like IP addresses, numbers or user names) which are bound together via some literal text. At a very minimum, the literal text is used to discern the end of one motif and the beginning of the next one. The only exception from this rule is formats which contain fixed-width motifs (often called "fields" in this context), which are delimited by just their position.

We can unify this description by considering literal text as motifs as well. This is also useful from an application point of view because literal text sometimes contains important information, so it does not solely serve as delimiter.

It must be noted that motifs can be nested. For example, an IPv4 address is obviously a motif. However, we can describe the IPv4 address itself as consisting of seven motifs, 4 of which are integer numbers in the interval [0..255] and the remaining three are literal text motifs, namely the dot character. Multiple levels of motif hierarchy are possible inside log messages. An example is JSON [6] strings that are sometimes included in log messages. As JSON is recursively defined, the JSON motif will potentially contain multiple levels of sub-motifs.

When we talk about log normalization, we usually mean the top level motif by our term *motif*. If we need to refer to sub-motifs, we will do so explicitly.

If we have a motif m and a string s, we say m *matches* s (and vice versa) if and only if the structure of s is identical to the structure defined by m. For example, an IPv4 address motif matches any proper textual representation of an IPv4 address. In case of literal text motifs, *matches* means that the byte sequence in s is identical to the byte sequence described by m.

Tying this all together, we can improve our definition of a log message from chapter 3.2: A *log message* is a string that was created with the intention to log data and it consists of a sequence of substrings each matching a motif.

4.3.1 Free-text Log Formats

Free-text formats are commonly called "unstructured" in trade publications. However, this classification is incorrect. These logs have structure, but the structure is not well defined.

In general, these types of logs were originally meant to be human consumable, thus they resemble natural language text. Usually, this text is generated in a way that parameters (like account names or IP-addresses) are embedded in an otherwise fixed-text message.

As such, free-text formats always contain multiple motifs. If they would contain a single motif, only, we would have one of these cases:

- *motif with sub-motifs* - for example a message consisting only of a JSON body. In such cases, do not really have free-text messages but rather a well-defined structure.
- *a single motif without sub-motifs* - we have never seen an example of such a message in practice. However, one might consider a message containing only an integer value as an example. Such a value could be a measurement, like temperature or CPU utilization. The exact meaning would need to be conveyed implicitely, e.g. by the system the message originated from. In any case, such messages have a well-defined structure, even though this is not obvious to an external observer.

In both cases, the message has a specific structure and so cannot be classified as free-text format.

This shows that free-text messages are always a sequence of two or more motifs.

Generation of Free-text Log Messages Most operating systems provide APIs for generating free-text log messages.

Linux and Unix Under Linux and Unix, the POSIX syslog() API [31] and helpers are used, at least in most C programs. This API permits a programmer to log a message much like in the C language's printf() function. As such, the official API documentation does not mandate any specific log format, nor does it provide guidelines on how to log specific objects. In the Linux Kernel, an equivalent API named printk() exists.

In order to get some understanding of the consequences, we did a brief analysis of open source software code repositories. We use the FreeBSD github repository [11]

as well as the Linux Kernel git repository [49]. Both repositories were downloaded in May 2015.

After download, we did a search over the whole source tree for calls of the syslog() respective printk() API. We extracted all lines where these were called, but only the single line where the call started. This often did not provide the complete call parameters, but usually the formatting argument, which was sufficient for our needs. This resulted in 4638 code lines for FreeBSD and 37608 for the Linux Kernel. This data set can be found inside an online repository created as part of this thesis [24]. It must be noted that the result is probably a small subset of the actual logging calls, as many projects tend to wrap their own log handlers around the syslog() API and then call this log handler. This also explains the big difference in number between all of FreeBSD and Linux kernel (printk() looks like it is very seldomly, if ever, wrapped). We manually reviewed the data set and watched for consistency and format similarities. While this was not an exhaustive survey, it permitted us to find important properties.

Typical use cases (here from the FreeBSD tree) look like this:

```
1 syslog(LOG_CRIT, "Attempted login by %s on %s", user, tt);
2 syslog(LOG_ERR, "user: %s: shell exceeds maximum pathname size",
3 syslog(LOG_ERR, "tried to pass user \"%s\" to login",
4 syslog(LOG_DEBUG, "login name is +%s+, of length %d, [0] = %d\n",
5 syslog(LOG_ERR, "setlogin(%s): %m - exiting",
6 syslog(LOG_AUTH, "Attempt to use invalid username: %s.", name);
```

All of these samples output user names. Even from this very small sample, a couple of issues can be seen:

No formal indication of parameters happens inside the API. For a log consumer, the fixed and the variable text is not directly discernible.

Embedded spaces in parameters cause issues. While this should not be an issue in the case of operating system user (account) names, there are ample other motifs where embedded spaces may be possible. Depending on the platform, even user names may include spaces, if properly escaped. If we now look at line 1, we note that a space inside the user name is very hard to detect when just looking at the resulting message. Even relying on the fixed substring "on" would not always be sufficient without additional context, as the value for user could be "user on". In the second line, the situation is slightly better because a colon is less likely to be part of the motif. Depending on the parameter value this still can lead to misdetection. Lines 3 to 5 mostly solve that problem by embedding the motif inside characters that are unlikely to happen within the motif itself. This, too, will not always work

as the same character can potentially occur inside the motif and as such would need to be quoted.

Motifs may contain delimiters As can be seen in line 6 the user name is terminated by a period, obviously in an attempt to mimic a natural-language sentence. As user names may validly contain periods [34, Sect. 3.426], proper detection of the user name is problematic. As described above, this applies to a lesser extent to lines 3 to 5.

No consistency is found in the way the same motif is written to the log. It very much depends on the actual code how the motif is written. Even within the same code base, different formats are found to write the same motif.

Intended for human consumption Looking at these points, developers target log records obviously primarily for human consumption. In practice, though, enterprises need to process log records automatically due to the large volume of them.

Microsoft Windows Under Windows, log message are formatted and gathered via the Windows Event Log subsystem [12] introduced in 1993. A strong focus at that time was to provide an easy way to support message text localization to different natural languages [41, Pg. 21]. To support this, structural elements had to be added, being most importantly [41]:

- *common header* - containing some key metadata elements, like the creation time in precisely defined format
- *event id* - a key that uniquely can identify a specific message type. This provides a method to precisely define the semantics and the syntax of a specific message.
- *parameters* - each message can have multiple parameters (like account credentials, time values or IP addresses). These parameters were initially only indexed by a numerical index, but in combination with the event id and other header information it was possible to identify them.

A major new version of the Event Log subsystem was released with Windows Vista in 2007. This version of the subsystem is also in use today with current Windows releases. Most importantly, it offers more structured parameters, for example in XML format [37].

The Windows event log offers considerably more structure than the POSIX syslog() API, but only if used correctly. Unfortunately, when working with real-world Windows logs, one notices quickly that most software developers do not make good

use of that API. Instead, it is treated much like the POSIX syslog() API. As most Windows applications are closed source and licenses usually prohibit reverse engineering, we cannot cite exact samples of the misbehavior. However, we known both from personal experience as well as working with other researchers on log processing that the actual event log content does not offer much useful benefit. For example, third-party software vendors often tend to use a single event id, which contains a single parameter, which then contains the actual message string. So in essence, the output format is exactly like on Linux and Unix. One might speculate if this stems back to code that originally existed on Unix, uses the syslog() API and has been ported with little effort to Windows.

To further complicate things, when Windows events are integrated into enterprise log management systems, they are frequently sent via forwarding tools like Snare Windows Agent [42] or Adiscon EventReporter [27]. This is done because most enterprise log management systems require heterogeneous sources and as such do not support event log format natively. The forwarding tools usually take the Windows event log database and convert entries into syslog-like messages. During that conversion step, the distinction between constant text and dynamic parameter is often lost, so that the same problems mentioned for POSIX apply.

It must be noted that recent versions of these tools also support formats like JSON which permit to retain structure. However, that mode is currently seldom used in practice and so from a practical perspective the Windows event log format, as seen on a central log management system, is mostly equivalent to the typical Linux log.

Other Device Vendors Other important vendors of network equipment like Cisco, Huawei, or many others follow the POSIX paradigm. They either run Linux kernels on their systems, in which case using POSIX is obvious or they use closed source software. In the latter case, review of the source code is impossible, nor are there documented APIs. However, one can suspect the use of the POSIX syslog() paradigm from analyzing a large set of emitted messages.

4.3.2 Semi-Structured Log Formats

We call those formats *semi-structured* which have some structure, but are not well-defined or at least not easily recognizable. Prime examples are:

- comma-separated values (CSV)
- name-value pairs

While both of them are well known, many variants exist, some of which may not even be reliably parsable. Let us use CSV as an example: the format simply has values which are delimited by commas. In some variants, it is not clear how to represent a comma inside the value part. In some, this case is even undefined, so embedded commas will introduce errors. In some, quoted values exist. Those are values surrounded by quotation characters. Some demand that if quotation is used, all values must be quoted. Others permit both quoted and unquoted values. Some do not permit quoted values at all. These are just some quick examples. In general, those formats have some structure, but are usually equally hard to interpret like free-text formats.

Log normalizers still tend to represent these formats by a specific motif.

4.3.3 Structured Log Formats

Structured log formats have a well-defined structure and can be parsed based on that definition. If the parsing of a specific message fails, one can conclude that it is either erroneous or in a different format. So a clear rule for format detection and parsing exist.

Formats commonly found are:

- ArcSight CEF [32]
- CEE [13]
- GELF [28]
- plain JSON [6]
- RFC5424 Structured Data [23]
- SIP CLF [44]
- Web CLF [55] [4]
- WELF [33]

As each of these formats is well defined, it is a single motif from our point of view. These are obviously the easiest to work with formats in log normalization.

[4]Note: some variations of the Web CLF format exist, but all are well documented and discernible from each other.

4.4 Motif classification

Motifs are derived from real-world objects used in log messages. They can be classified in terms of complexity in comparison to regular expressions.

1. Motifs that can easily be implemented via regular expressions, like MAC layer addresses. The majority of motifs is in this class.

2. Motifs requiring elaborate regular expressions. An example is the ipv4address motif. The key point here is that each address octet must be in the interval [0..255]. As numerical checks are not supported, the regex must be built based on valid digit sequences. This class can occur quite frequently and is present in many actual log files.

3. Motifs actually outside the class of regular expressions as specified in computer science (cs). Very few motifs are in this class, but we have seen in 5.2.1 that some are even context-sensitive. A prime example of this class is the CEF format which requires look-ahead in order to differentiate names from values (see also Sect. B).

Furthermore, motifs can be classified in how precisely they match a substring. Take for example these two motifs which are frequently used in practice:

- *word* - a motif that describes a sequence of characters not containing the space character. The first occurrence of a space character or end of string terminates this motif. This is a base type that may be used, for example, to represent user names.

- *ipv4address* - a motif describing the textual representation of an ipv4address

If we take the string "192.0.2.1", it obviously matches both of these motif definitions. On the other hand, the string "hostname" matches the "word" motif, but not the "ipv4address" motif. In those cases, we say that the "ipv4address" *matches more specifically* than the "word" motif, or we may say that "word" is *broader* (in scope) than "ipv4address".

Usually, we have a subset relationship on the specificality of motifs, in that some motifs match a subset of some other motif. However, we may also have motifs which only have a partial overlap, e.g. an intersection of message formats. To show these, let us consider two other motifs:

- *hexnumber* - the hexadecimal representation of an integer number

- *float* - the representation of a floating-point number

The string "123" matches both of the motifs, while "1a3" matches only the "hexadecimal" motif and "1.3" matches only the "float" motif. We say that such motifs are *conflicting*.

We call both conflicting motifs and those with different specificality *ambiguous*. Also, we need to remember that motif structure is dictated by real-world objects, so we cannot forbid or easily overcome such ambiguities.

5 The Log Normalization Problem

5.1 Problem description

Let L be a set of log messages. Let the mfmt function be as described on page 11. Let $F = \{\text{mfmt}(l) | l \in L\}$ the set of all formats used inside L. Let f_o be a desired format, which not necessarily is in F. Let $F' = F \cup f_o$. Let $i = \text{extr}(l, f)$ be a function that extracts motifs from message $l \in L$ in format $f \in F$ and returns them in an interim format i. Let $\text{fmt}(i, f)$ be a function that takes a message in interim format i and formats it in format f. Let $l' = \text{fmt}(i, f)$. Note that l and l' may contain different information; deletion and insertion of information is permitted during the normalization process. For example, in practice it is common to annotate normalized log messages with some classification information.

Then the log normalization problem is: for each $l \in L$ detect its format and transform the message to be in f_o format. This is formalized in algorithm 1:

Algorithm 1 log normalization problem

for all $l \in L$ **do**
  $f = \text{mfmt}(l)$
  $i = \text{extr}(l, f)$
  $l' = \text{fmt}(i, f_0)$
end for

The core problem is the mfmt function implementation, because the message format is hard to detect. The other functions are rather simple.

In implementations, the mfmt and extr functions are often combined, because the extraction of data items is usually a side-effect of the detection algorithm.

Log normalizers use format databases to implement the mfmt function. Various names exist for these databases and they vary greatly in content and structure. In

this thesis, we use the term *rulebase* to refer to such a database. A *rule* is a formal description of a specific format. A rulebase is set of rules. In some normalizers rule bases are ordered sets.

As we have seen in section 4.3, the format of log messages belongs to different language classes. As such, rules must be able to detect all of them.

Each rule describes a specific log message. The rule is like a template for a specific message: it contains the same sequence of motifs as the message itself. Each motif can be constant or dynamic text. Let M be the set of all possible motifs. Than a rule r is formally defined as follows:

$$r = (m_1, \ldots, m_n) | 1 \leq i \leq n : m_i \in M$$

Reconsider that a log message $l \in L$ is a sequence of substrings s_i representing motifs:

$$l = (s_1, \ldots, s_n)$$

A rule is said to *match* a message if and only if both the rule and the log message have the same number of components n and substring s_i matches m_i for all $1 \leq i \leq n$.

The naive implementation for the mfmt(l) function is as follows:

Algorithm 2 mfmt function (naive)

```
for all r ∈ R do
  if l matches r then
    return r (success)
  end if
end for
return no match (failure)
```

Let c_m be the time complexity of the "matches" operation. Then the time complexity of the naive algorithm is $O(|R|c_m)$. If we assume that c_m is $O(1)$, then the naive algorithm is linear in $|R|$. So this algorithm does not scale well for large rulebases. As we will soon see, it nevertheless is used in practice.

5.2 Normalization Algorithms

For the rest of this section let R be a rulebase of required type, $r \in R$ a rule, and $n = |R|$. Let L be a set of log messages, $l \in L$ an individual log message in some

input format, and l' the normalized message in desired output format f_o. Further let k be the maximum configured log message size in bytes (so $\forall l \in L : |l| \leq k$).

5.2.1 Regular Expression based Normalizer

As said in the introduction, the traditional approach to normalization is to use regular expressions to describe motifs (except for literal text). However, it must be noted that the term "regular expression" is not used in a strict cs sense. Rather, it refers to implementations used in practice, like PCRE [17]. These implementations support many extensions to the cs model, for example back references. As Aho has shown in [3, Sect. 2.3] regular expressions with backreferences do not describe "regular or even context-free languages". In [3, Sect. 6.1] the author shows that deciding these regular expression class is NP-complete, so based on the assumption $P \neq NP$ there exists no polynominal-time algorithm for deciding them. This is in sharp contrast to cs regular expressions, which can be decided in $O(k)$. As Ross Cox shows in [14] the feature-rich regular expression libraries used in practice actually have a very bad runtime performance. This is in line with user reports of the slowness of regex-based normalizers (for an example report for "grok", see [40]).

The regex-based normalizers treat rulebases as an ordered set of rules. Format detection happens by iterating over the rules until a match is found. In each iteration, all rule regular expressions are executed. This leads to normalization algorithm 3.

Algorithm 3 mfmt function (typical regex approach)

```
for all r ∈ R do
  if l matches r via regular expression then
    return r (success)
  end if
end for
return no match (failure)
```

This basically is the naive algorithm, but now the "matches" operation has time complexity $O(\text{exp})$. This leads to $O(\text{exp}) = O(|R|) \cdot O(\text{exp}))$ time complexity of the overall algorithm. In practice, few regular expressions actually have exponential time complexity. Nevertheless, they usually have runtime costs that cause notable delay if repeatedly executed. From a practice point of view, the time required to execute a single rule is considered constant, but "high". So this algorithm is considered to have runtime cost linear to the number of rules being used. That is problematic because due to the large variance in message formats large rule sets are often required.

Due to the "high" cost of evaluating a single rule, even small rulebases have problematic runtime requirements (see our test results in 7.2.1). See [40] for a practical report where even after optimizing the regex only 500 messages per second could be processed, far less than what is usually required by larger organizations.

As a further problem, some of the to be processed formats, like ArcSight CEF [32] are context-sensitive languages and cannot be expressed by the regex engines used. While such formats are rare, they create the need for special handling in regex based normalizers or are simply not supported by them.

5.2.2 Prefix Search Normalizer

The performance of regex-based normalizers could be improved by removing the iteration over R. Looking at the deterministic finite automaton (DFA) form of the regular expression, this can be done by combining all individual DFAs into a single one. It will remove the $O(|R|)$ overhead for looping through R, which means that we could support large rule sets, just as we would like to. This thought path leads to prefix search normalizers.

Furthermore, when looking at existing regex rule sets, one notices that motifs are usually simple objects like user names, IP addresses, timestamps, literal text and so on. Full regexp capabilities are not required to extract them. Finally, matching always occurs from the initial character of the message towards the end, so no matching within a larger text is required.

This leads to the idea to use a data structure specialized in high-performance matching as basis for a single DFA. That was the core idea used around 2010 in syslog-ng and liblognorm.

This approach is based on the idea of prefix search trees in string processing [35, Sect. 6.3]. In order to understand prefix search normalizers, we first need to understand search trees.

Tries

The first search tree proposed [35, pg. 492] was the trie data structure [19]. It was introduced in 1960 by E. Fredkin and developed for high performance searches. A trie is is a search tree which does not contain the key in each node, but rather key prefixes as edge labels. A sample is depicted in figure 2. For any path, the key can be constructed by concatenating the edge labels. In the example, "and" can be constructed by walking the path (a, n, d). Parsing a string $s = (s_1, \ldots, s_n)$ is done selecting the top-level edge by s_1, then the second level edge by s_2 and so on, until either a mismatch or a terminal is found. This means each s_i is only evaluated once,

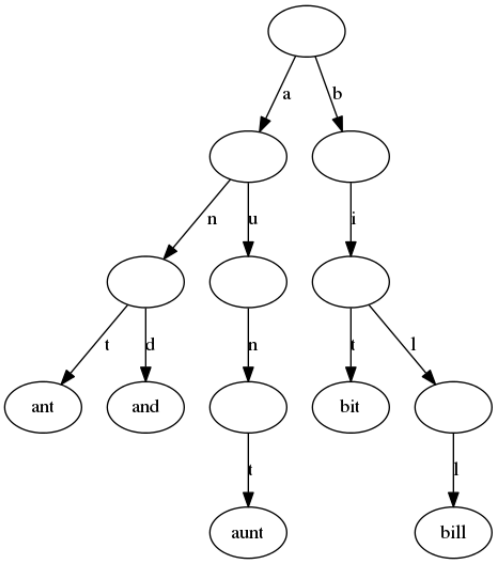


Figure 2: A sample Trie.

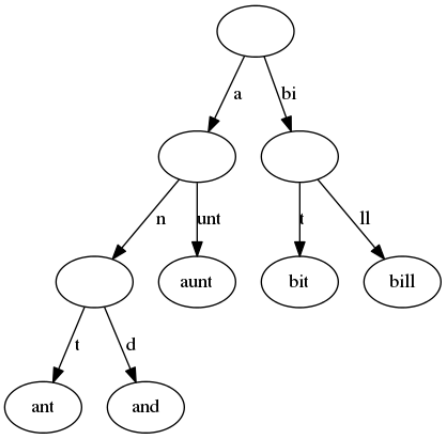


Figure 3: A sample PATRICIA tree.

and so the trie decides s in $O(|s|)$ time. If an upper bound for $|s|$ exists, a trie decides in constant time.

The original trie had some problem areas associated, among others excessive memory use for inner nodes. Inner nodes have large fixed-length branching tables representing edge labels. Tries have continuously been improved. For example, Morrison described the PATRICIA tree [39] in 1968, which addresses the problem of memory usage by nodes not strictly necessary. Let $(s_1, \ldots, s_n)$ by a tree path. Then, an *empty path* is any subpath $(s_i, s_{i+1}, \ldots, s_{i+m})$ with $1 < i \leq n, i + m \leq n$ where $\text{indeg}(s_k) = \text{outdeg}(s_k) = 1$ with $i \leq k \leq i + m$. As there are no alternatives branches, an empty path must be walked completely if the string matches the search tree. As such, all empty path nodes can be collapsed into the corresponding edge

label in node s_{i-1}, which then matches multiple characters. As trie nodes have high memory requirements, this can considerably reduce memory usage. This process is called *empty path compression*.

Our sample as PATRICIA tree can be found in figure 3. Note that PATRICIA trees are also referred to as *radix trees*. We will use that term in our thesis, as it seems more popular in recent literature.

Several improvements of tries lead to the 2013 introduction of the "adaptive radix tree" [36] (ART) by Leis et al. It provides even better space compression and works better with the modern computer main memory system, namely CPU caches. Note that Leis, in Section III, offers a good overview of radix tree advantages, which we refer the interested reader to.

Parse Radix Tree Radix trees in their pure form provide matching of literal text only. So for a string s, they can guarantee $O(|s|)$ complexity.

We need to adapt the definition for use in log normalizing, where the matching bases on motifs. To support this, we introduce the *parse radix tree* (PRT). It is a radix tree where edges are labeled with motifs rather than constant text.

As we have seen in Sect. 4.4, this introduces a new problem: different motifs may match the same substring. For example, let us consider the "word" and "ipv4address" motif from 4.4.

If we now look at the log message "Attempted login by guest on 192.0.2.1", the substring "192.0.2.1" matches both of these motifs.

So after our modification the PRT is no longer represented by a simple DFA. Instead, it now is a nondeterministic finite automata (NFA) [2, sect. 3.6.1] $N = (Q, \Sigma, \delta, q_0, F)$ with

- Q being the set of internal states
- Σ being the set of input symbols, which consist of motifs
- δ is the transition function, which is represented by a tree
- q_0, the start state, is at begin of the log message
- F being the set of terminal states contains all states in which a rule has completely matched the to-be-processed log message.

The PRT is the tree-representation of δ. When we implement an algorithm for N, we can no longer rely that the matching is always done by constantly evaluating a single path in the PRT. Matching can now require a much more elaborate walk.

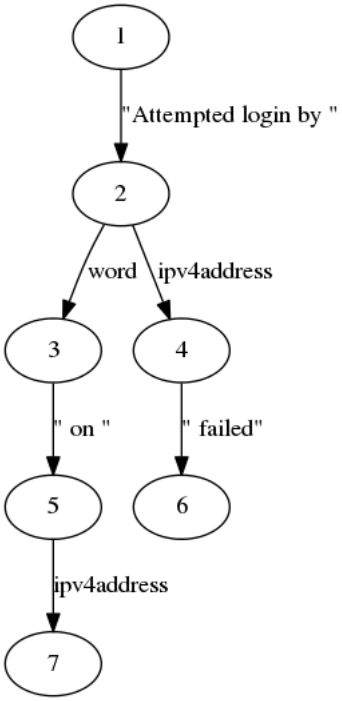


Figure 4: A sample parse radix tree with potentially conflicting motifs.

Let us consider a case from practice. We have potential log messages "Attempted login by guest on 192.0.2.1" and "Attempted login by 192.0.2.1 failed". The following rulebase describes both of these messages:

```
("Attempted login by ", word, " on ", ipv4address)
("Attempted login by ", ipv4address, " failed")
```

Note that constant text motifs are shown by giving just the literals. If we construct the PRT out of this rulebase, it looks like shown in figure 4.

Node 2 shows a potential problem during tree walk: there are two branches with motifs that both matches a set of common substrings. Now let us consider the message "Attempted login by 192.0.2.1 failed".

From a theoretical perspective, the NFA evaluates all potential pathes. As such, the message is matched and the path for this match is $(1, 2, 4, 6)$.

In an implementation, however, we need to do matching sequentially. Let us assume the "word" motif would be evaluated first. As the substring "192.0.2.1" perfectly matches the "word" motif, parsing would continue with node 3. There, the substring " failed" is tried to match, but that fails. In a regular trie, parsing would now be completed and a mismatch returned. In contrast, the PRT needs to take into account that it does motif matching and as such another path might provide a match. So the matching algorithm needs to go back one layer (from node 3 to

node 2 in our example), and check if there are other edges available whoms motif match ("ipv4address“ in our sample). If so, the algorithm must try to match via that edge. In our case, this means it branches to node 4, will now correctly match " failed“ and return match success. So the complete walk for this matching operation is $(1, 2, 3, 2, 4, 6)$.

This ability to go back upwards inside the tree and continue the evaluation of not yet tested motifs is a new property of the parse radix tree. We call this process *backtracking*. Note that this is a recursive process, which can happen on multiple tree levels. As such, a PRT unfortunately no longer has time complexity of $O(|s|)$ for a string s. In theory backtracking can cause exponential runtime. For now just let us note that this does not happen in practice due to the structure of log messages. We will show details in section 6.4 "Time Complexity“ on page 42. Nevertheless, backtracking, if it happens too frequently, has a negative effect on PRT performance.

Algorithm 4 describes the core idea of a recursive PRT matching algorithm.

Data Structures used by Popular Projects Syslog-ng uses a PRT basing on an ART. It is unknown if syslog-ng initially used a different algorithm in the past (the ART was published in 2013, and the first version of syslog-ng pattern db around 2010) and it is also unknown, though likely, if the commercial version uses the same algorithm.

Liblognorm utilizes a PRT which bases on a PATRICIA tree. Additionally, the PRT includes some trie-like quick lookup capability for children based on the next literal character. In each node, it contains large literal lookup tables with one pointer entry for each of the 256 potential byte values. The intent is to increase lookup speed, but it comes at the cost of large memory consumption.

Both syslog-ng and liblognorm v1 use a variant of algorithm 4. Most importantly, they treat literal text different from other motifs. The liblognorm v1 algorithm gives literal matches priority, whereas syslog-ng prioritizes other motifs. This is done by evaluating literals in different places inside the algorithm. The liblognorm method is shown in algorithm 5, where literal matches are done after other motifs. For syslog-ng, this is just the opposite. Note that other details may also be different for syslog-ng. Liblognorm v1 does empty path compression by storing the literal that corresponds to the empty path as a string inside the node. We call this "empty path prefix“ in the algorithm description.

Algorithm 4 prefix search normalizer (basic algorithm)

```
if s is the empty string then
  if n is a terminal node then
    return success
  else
    return failure
  end if
else
  for all edges e of n do
    if e matches prefix of s then
      recursively call ourselves with node pointed to by e and unmatched suffix
      of s
      if success returned then
        return success
      end if
    end if
  end for
end if
return failure
```

For a log message l, the algorithm receives the substring s, which is the not yet matched suffix of l. Also, it receives the node n from where the PRT is to be walked. Initialization happens by calling the algorithm with the complete message l and the root of the PRT.

Algorithm 5 liblognorm v1 search algorithm

```
if s is the empty string then
  if n is a terminal node then
    return success
  else
    return failure
  end if
else
  if prefix of s is equal to the empty path prefix then
    set s to first char after empty path prefix
    for all non-literal motifs e of n do
      if e matches prefix of s then
        recursively call ourselves with node pointed to by e and unmatched suffix
        of s
        if success returned then
          return success
        end if
      end if
    end for
    use literal lookup table to see if edge exist labeled with first character of s
    if such edge exists then
      recursively call ourselves with node pointed to by e and unmatched suffix
      of s
      if success returned then
        return success
      end if
    else
      return failure
    end if
  else
    return failure
  end if
end if
return failure
```

For a log message l, the algorithm receives the substring s, which is the not yet matched suffix of l. Also, it receives the node n from where the PRT is to be walked. Initialization happens by calling the algorithm with the complete message l and the root of the PRT.

6 An improved normalization method

6.1 Goals and alternatives for an improved method

Our PoC with liblognorm v1 proved that a prefix search normalizer works reasonably well in practice. However, we noticed a number of problem areas with the PoC:

- *space requirements* - memory consumption is rather high, especially for larger rulebases. Among others, a core problem is the edge label tables, which alone consume 2KiB per node on a 64-bit system.
- *motif evaluation order* - as we have seen, some strings are matched by multiple motifs. This can lead to mismatches. With liblognorm v1, we have seen some of those cases in practice. It must be noted that this ambiguity can not simply be avoided: there is ambiguity within the language of the rulebase. As described in [30, Sect. 5.4] such ambiguity cannot automatically be removed.
- *speed increases* - even though the prototype is already much faster than regular expression based normalizers, we would like to further improve the speed.
- *CPU cache friendliness* - the prototype algorithm was designed without regard to cache performance. This shall be changed not only to improve runtime performance but also put less burden on system resources in general.
- *limited set of motifs* - the prototype includes only a very small set of motifs, what often poses a problem in practice.
- *user-defined motifs* - the success of grok is partly claimed to the fact that it is very easy for users to extend the set of supported motifs. For liblognorm v1, this requires C programming skills, which means no extensibility for ordinary users.
- *one normalizer for all formats* - it is very desirable to be able to use a single normalizer for all kinds of message formats, because this is what is seen in typical log streams. If we would restrict our algorithm to the normalization of free-text log formats, in practice other system components would need to be extended to cover the other ones. So it is best to include that capability in the algorithm itself.
- *runtime analysis* - no runtime performance analysis of any normalizer has been published so far. Thus is is unclear which runtime behavior can be expected for typical workloads.

We can base the new algorithm on regex or search tree concepts to reach these goals. The following paragraphs describe advantages and disadvantages of the approaches. In the following, let l be a log message.

regex-based normalizer As we have seen in Section 5.2, the main problem of the *regex* based approach currently used in practice is that it executes regexes sequentially and regex libraries support advanced features outside the class of regular languages, what leads to exponential runtime.

As we have already said, we can overcome the sequential execution problem by building a single automata for the complete rulebase. That would be the approach we would need to look into.

We discuss this based on the motif classification in regard to regular expressions in Sect. 4.4. These classes need different implementation in a regex-based normalizer:

Classes 1 and 2 can be dealt with by an ϵ-NFA [30, Sect. 2.5] which then could be converted to a DFA. The result could then be amended by an algorithm that handles the few remaining class 3 cases. This could be implemented, for example, by a hierarchy of different automata which work together. It must be noted, though, that a pure DFA implementation is impossible because we need to support motif class 3, which is outside the class of regular languages.

This approach has the advantage that it provides an $O(|l|)$ runtime guarantee whenever no class 3 motif is used. This would be in the vast majority of cases and as such be an advantage. However, the ϵ-NFA would have a large number of states because of class 2 motifs. Further, it is expected (but not experimentally proven) that the translation from ϵ-NFA to DFA will lead to a very large state set. Implementation-wise this would mean large state tables and a transition function implementation that would frequently need to access spatially remote areas of memory. That leads to a very cache-unfriendly implementation. Finally, the ϵ-NFA does not solve the ambiguity problem described in 4.4.

search tree-based normalizer With the search tree concept, all motif classes can be handled in an uniform way. The ambiguity problem can be solved with relative ease by motif prioritization.

Nodes are the search tree analogon to DFA states. The search tree requires fewer nodes than the DFA states because a) we can compact them and b) motif parsers themselves do not use explicit states. The current memory consumption can be reduced by following the ART paradigm. Fewer nodes increase spatial proximity of frequently accessed data items, which can be further improved by proper layout

of the data structures as part of algorithm engineering. This makes the approach attractive from a memory consumption and cache performance point of view.

On the contrary, we cannot prove the desired $O(|l|)$ time complexity, not even for the important subset of motif classes 1 and 2. However, practical experience with the liblognorm v1 PoC indicates that observed run time is "sufficiently" fast and backtracking happens only infrequently, so that it does not negatively affect the heuristically expected $O(|l|)$ runtime. As with the regex-based normalizers, the use of class 3 motifs inside rulebases causes worst-case exponential runtime.

Conclusion Weighing these arguments, we conclude that an algorithm based on the search tree concept is best for solving our problem. The main arguments are:

- it provides a simple solution for solving the ambiguity problem
- it provides a unified solution for all motif classes
- it promises less memory consumption and better cache performance

6.2 The data structure

We base our data structure on the PRT as described in Section 5.2.2, but change the structure from tree to DAG. In the following, we continue to use the terminology we introduced for search trees and mean the analogon for DAGs. Let us further define: if $(n_1, \ldots, n_n)$ is a walk inside a graph, then any subsequence of nodes in that walk is a *subpath*.

This DAG offers important new capabilities to improve search trees:

common subpathes It frequently happens that two log messages l_1, l_2 consist of mostly the same motifs, but have small differences in some places. For example, l_1 may contain a host name at the same location where l_2 has an IP address. In such cases, the subpath from that location to the next difference is equal for both messages. We call this a *common subpath* when walking the DAG. A common subpath that leads to the same terminal symbol is also called a *common suffix*. If we assume the user is interested in parsing such messages via different rules, inside a tree we will duplicate common subpathes (see figure 5 for an example). With a DAG, we can eliminate these duplicates and need to include the path only once inside the DAG (see figure 6 for the example as DAG). This results in reduced memory usage. Also, cache performance increases, because we need to walk only a single memory area for all instances of the common subpath.

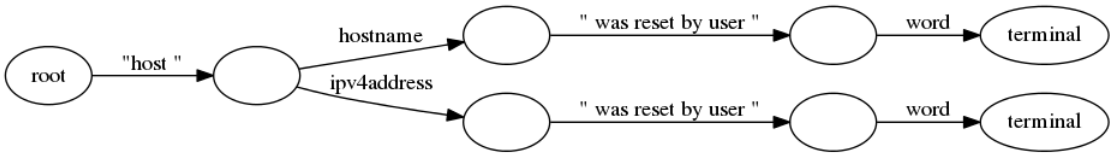


Figure 5: A search tree with a common suffix

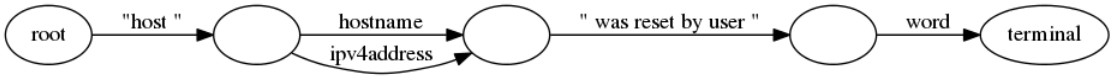


Figure 6: A search DAG with a common suffix

disconnected components Disconnected components permit to store common subpathes as their own component inside the DAG. Let us show a simplified example based on what is found inside the PIX data set. Let us assume we have messages like in the following:

```
traffic flowing from hosta to hostb permitted
traffic flowing from 192.0.2.1 to hostb permitted
traffic flowing from 192.0.2.1/80 to hostb/5432 permitted
traffic flowing from 192.0.2.1/80 to 192.0.2.2/5432 permitted
traffic flowing from hosta to 192.0.2.2/5432 permitted
```

As one might suspect from these samples, the actual motif structure is as follows:

```
("traffic flowing from", HOSTSPEC, " to ", HOSTSPEC, " permitted")
```

Here, "HOSTSPEC“ is a combined motif which consists of a host name (which may be an IP address) optionally followed by the slash character and a port number. Let us assume a motif "hostname“ exists, which represents either an alphanumeric name or an IP address. Let us further assume "HOSTSPEC“ is not within the motif set provided by the core implementation. This is quite likely for many of such combined motifs, because there are so many possibilities. So we need to build it out of the available motifs hostname, literal, and number.

In a pure search tree, we need to represent this as shown in figure 7. This is the structure that liblognorm v1 actually generated. With the search DAG, a first simplification is possible, shown in figure 8. Note that this representation is also much closer in structure to the actual message format.

With the introduction of disconnected components, we can finally fully model the actual message structure. We permit to use individual disconnected components for combined motifs like "HOSTSPEC“ in our sample. This results in a further

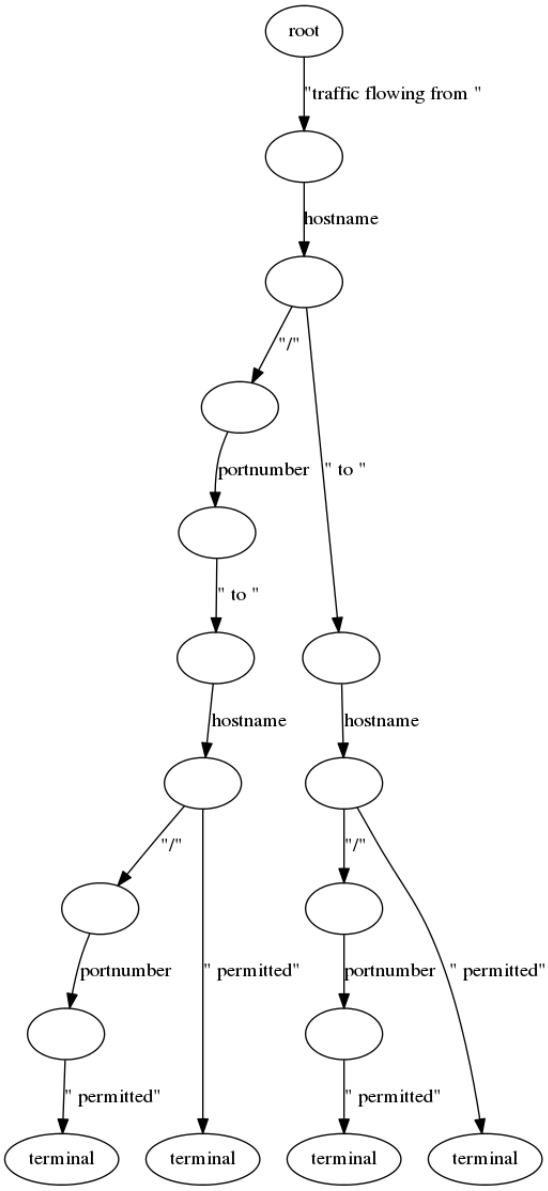


Figure 7: search tree of the firewall sample

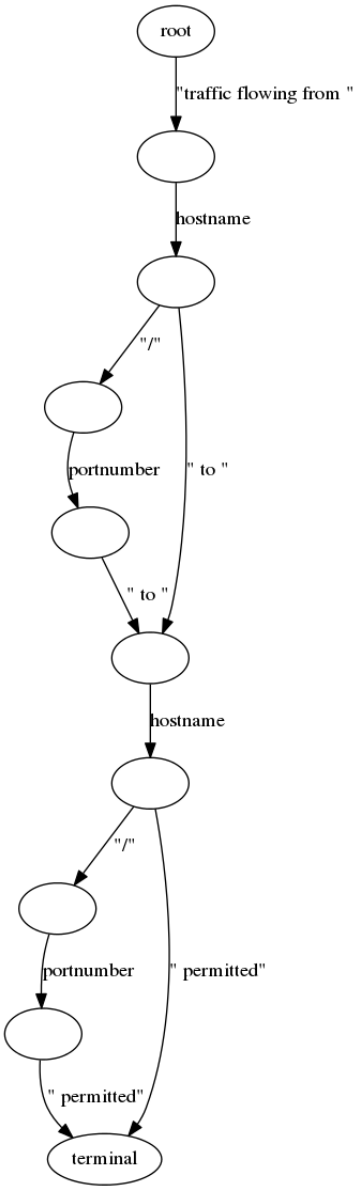


Figure 8: search DAG of the firewall sample

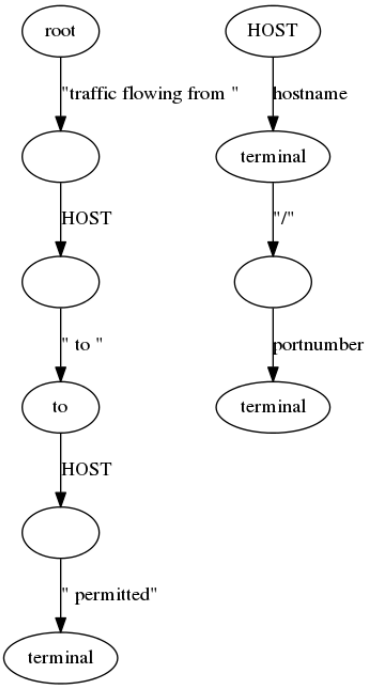


Figure 9: search DAG with disconnected component of the firewall sample

simplification of the search DAG, as can be seen in figure 9.

In practice, the simplification due to disconnected components is even more radical: In real Cisco PIX logs, a "HOSTSPEC" is a much more complex construct, with more optional motifs. It is specified as follows:

```
[interface:]ip/port [SP (ip2/port2)] [[SP](username)]
```

Brackets indicate optional parts. Also, this combined motif may occur multiple times inside a log message. So a search tree will grow rapidly in size. This is a well-known problem in liblognorm v1. Actually, we implemented this combined motif as a base motif in liblognorm v1. We did so because it is a very important object for many users. However, there are many similar combined motifs in practice, which cannot be supported by adding native types for each of them. Consequently, it is very hard to handle such cases with regular search tree based algorithms.

unified edge labels We do not treat literals and motif parsers differently. Instead, we introduce a new "literal" motif, which provides the match over a literal. This enables us to keep the data structure small and relieves us from special case handling for literals inside the algorithm, resulting in a simplified solution. Furthermore, this removes the need for the large edge label tables liblognorm v1 kept in each tree node. This is a considerate space saving but comes at the cost of slightly worse lookup

times when many literals exist in a single node. As evaluation results show, this is not a real issue, especially as usually there are not many edges labeled with literal motifs.

prioritized edge labels As we have shown, motif ambiguity causes problems during normalization. We introduce the ability to prioritize motifs and so affect the search order. This will solve all issues where motifs have a subset relationship and it enables to solve conflicting motif issues by proper configuration.

rooted DAG Each disconnected component d has exactly one node with indegree 0. This is called the *root node* of d. Parsing always starts at the root node. Exactly one of the PDAGs component is designated as the *root component*. Parsing of a log message starts with the root node of the root component. For convenience, we call this node the *root of the PDAG*.

loops Some motifs have a set of one or more values. During normalization, we would like to extract the individual items of this set and present them to the application (for example as an array). This requires looping capabilities. One way to design this is to natively support loops inside the data structure, which then no longer would be a DAG. Another approach is to implement such loops as a motif, outside of the DAG itself. We follow this latter approach because loops actually only occur in motifs, not within the normalization process itself. So implementing them on the motif level is the right thing to do.

parse DAG We call the data structure described in this Section a parse DAG (PDAG).

6.3 The Algorithms

There exist two important operations for a PDAG:

- construction of the PDAG
- parsing a message via the PDAG

Also a number of utility algorithms, like PDAG deconstruction, are needed. We will not describe them as they are obvious.

Note that we do not describe an algorithm for searching the PDAG but rather for parsing a message via the PDAG. The reason is that parsing the message and

searching is done at the same time because of efficiency reasons. We prefer the term "parsing" over "normalization" in this somewhat broader context.

The anticipated use case is that the PDAG will be constructed once and then be used unaltered for parsing many messages, possibly over many hours or days. So PDAG construction is not a time-critical operation, whereas parsing via the PDAG is. Realtime requirements exists only for the parsing algorithm. Consequently, we will permit the construction algorithm to use extra runtime if that helps to speed up the parsing process. This includes potentially long-running optimizations.

The only constraint on construction time complexity is that the construction must be "sufficiently fast" to permit acceptable restart times of the normalizer. With the implementation developed for this thesis, this is the case. But let us assume this would not be the case: then, we could split the construction process into two subprocesses:

1. read the rulebase, do all lengthy operations and store the finally resulting PDAG in an easy-to-load binary format in file f
2. read f and proceed with parsing

Step 1 would be done in its own process, which could be run concurrently to the active instance of the normalizer which runs unaffected. Once step 1 is done, the normalizer can re-read the PDAG via step 2, which would only introduce a small delay. As we see, we can always ensure that PDAG construction is sufficiently fast.

The fact that the PDAG is read-only after construction is also helpful for parallelization: we do not need to place restrictions on the number of concurrent operations that utilize the data structure itself at the same time.

6.3.1 PDAG Construction

The construction process works in two stages: the actual load stage and preparation and optimization of the data structure.

During load, it builds PDAG components and nodes based on the provided rulebase. For each rule, the already existing PDAG is walked with the motif sequence specified inside the rule. When a not yet known edge motif is found at some node, the edge is added and a new node created for it. During this process, literal strings are converted into individual one-character literal motifs. So the string "and" becomes the literal motif sequence (a, n, d). This initially creates a very deep PDAG, but simplifies the construction process greatly. The PDAG will be shrunk again in

the optimization phase. Terminal nodes are flagged as such when the end of a rule is reached.

In the preparation and optimization phase all PDAG components are processed individually. Motif priority order is established by simply keeping motifs in an array and sorting this in order. This permits fast processing in motif order in the parsing algorithm.

More importantly a process similar to empty path compression inside a PATRICIA tree is performed: all pathes of the DAG are evaluated for sequences of nodes n with $\text{indeg}(n) = \text{outdeg}(n) = 1$ where the only edge is labeled by a constant. If those are found, these nodes can be compressed. Let $(n_1, n_2, \ldots, n_{k-1}, n_k)$ be a subpath where $\text{indeg}(n_2) = \text{outdeg}(n_2) = \ldots = \text{indeg}(n_{k-1}) = \text{outdeg}(n_{k-1}) = 1$ which contains only literal motifs, then $(n_2, \ldots, n_{k-1})$ are removed and their literals are concatenated to the motif of the (n_1, n_2) edge, which is further updated so that the subpath becomes (n_1, n_k).

We call this process *literal path compression*. It undoes the expansion of literals done in the load step, but now this is very simple to do. If we would instead do this at load time, we would permanently need to reconstruct parts of the PDAG when literal nodes would need to be broken up due to new rules. As we do not have any parsing operations concurrently to building the PDAG, there is no need to follow that more complex paradigm.

The algorithm is shown more formally in algorithm 6.

Algorithm 6 PDAG Construction

```
{load phase}
set current component to root component
for all rules r in rulebase do
  if r requires change of current component then
    create new component, if required
    switch current component
  else
    set node n to root of current component
    split r into motif sequence M {each character of a literal is its own motif}
    for all m ∈ M do
      if n does not yet have an edge identical to m then
        create new node n'
        add edge for m, let it point to n'
      end if
      set n to next node pointed to be edge labeled m
    end for
    flag n as terminal node
  end if
end for
{preparation and optimization phase}
for all disconnected components do
  sort all edge labels by motif priority
  do literal path compression
end for
```

6.3.2 Parsing via the PDAG

The parsing operation is essentially a search, with the added functionality that parsed data items are assigned to the respective to-be-extracted fields. This way, we can do search and parsing in a single step. Note that if we were to split these two operations, we would during the search stage need to record information of where the extraction stage needs to extract the data items. That recording overhead is very similar in processing time to the time required for the actual extraction. As such, the combined algorithm performs faster and thus is preferable.

This algorithm 7 is conceptually very similar to the basic prefix search normalization algorithm 4. This is a great simplification over the liblognorm v1 PoC algo-

Algorithm 7 Parsing via PDAG

```
if s is the empty string then
  if n is a terminal node then
    return success
  else
    return failure
  end if
else
  for all edges e of n {in priority order} do
    if e matches prefix of s then
      recursively call ourselves with node pointed to by e and unmatched suffix
      of s
      if success returned then
        extract matching data item
        return success
      else
        discard already extracted data items
      end if
    end if
  end for
end if
return failure
```

For a log message l, the algorithm receives the substring s, which is the not yet matched suffix of l. Also, it receives the node n from where the PDAG is to be walked. Initialization happens by calling the algorithm with the complete message l and the root of the PDAG.

rithm (5). This simplification was possible due to the data structure changes, most importantly the treatment of literals as regular motifs. The new algorithm increases maintainability of the actual implementation. It also helps to improve processing speed, because the CPU instruction path is shorter and so better fits into caches. It must be noted that even though the new algorithm is simpler, it is more capable than the previous one. For example, it supports user-defined types (disconnected components) as well as looping structures.

One important detail that not directly visible from the algorithm is the way disconnected components are treated. They represent motifs and are named. So a generic motif parser for this type is defined, which recursively invokes the algorithm at the root of the disconnected component with s. In regard to algorithm 7, the only special handling required is that this recursive invocation terminates successfully when a terminal node is reached and does not require all of s to be parsed (because a motif is usually not a suffix). We have not added this detail to algorithm 7 because it would detract from actual algorithm structure and is easy to add in implementation.

6.4 Time Complexity

6.4.1 Theoretical Worst-Case Complexity

The PDAG consists of potentially many disconnected components. As shown in Algorithm 7, parsing is done by walking the PDAG. Disconnected components are used for user-defined motifs, to which the regular parsing algorithm is applied. So without loss of generality, we can focus our analysis on a single component.

We call the time complexity of the parsing algorithm $O(\text{PDAG})$. Without loss of generality, only successfully matching operations are considered. Non-matchings ones always terminate earlier. Time complexity depends on

- motif parser complexity
- amount of backtracking

Individual motif parser complexity is shown and analyzed in Appendix B.

In a first approach to $O(\text{PDAG})$, let us assume that the rule base does not need to use backtracking. Note that this can happen in practice, as we have seen in Experiment 7.2.6.

Let l be a log message and $|l|$ be its size in bytes. The empty word is not permitted as motif. As such, all motif parsers need to obtain at least one byte from l during the parsing process. As the parsing algorithm always works on the yet-unparsed suffix s of l, a longest walk inside the PDAG can at most include $|l|$ nodes.

Let us now consider motif parser behavior. Let us consider two cases

1. for all nodes, always the first edge matches
2. for all nodes, always the last edge matches

Case 1 cannot be used for worst-case complexity, but provides us some interesting insight. In it, if a motif parser matches, it processed b bytes from l. These bytes are removed from s. So after this step, the walk can mostly contain additional $|s| - b$ nodes. This means the number of nodes inside a walk is limited so that the sum B of bytes processed by each parser equals $|l|$. In other words, no matter how many nodes are present inside a walk, their combined complexity is always $O(|l|)$ as that is the exact number of bytes which are being processed. So in case 1, we have complexity $O(|l|)$.

Now let us consider case 2, where always the last edge matches. This is the worst case and so includes all other cases where interim edges match.

Let us first assume a maximum size PDAG where all edges are literal motifs. Then, by PDAG construction, we can at worst have 1 byte per edge label and so have 256 edges per node. By applying the principle from case 1, all walks from PDAG root to a terminal node will then involve $|l|$ nodes. The worst case here is that always the last edge of each node matches. In this specific case, the literal parser calls are $O(1)$, because each one matches exactly 1 byte. As a result, in this scenario we have a maximum of 256 matching operations per nodes, and we have a maximum of $|l|$ nodes. This gives us a complexity of $O(256|l|)$, which is in $O(|l|)$.

Now let us consider what happens if motifs other than literal are used. Now, the situation is different:

- we no longer have a limit of 256 edges per node. Let m be the number of motifs other than literal. Then, we have a maximum of $256 + m$ edges per node.
- we cannot guarantee that edge labels be evaluated in $O(1)$

The number of additional motifs c is relatively small in the current implementation, but may be greatly extended. In any case, the maximum number of edges $256+c$ would still be a constant, and so the resulting time complexity would $O((256+c)|l|)$ and thus still be in $O(|l|)$. The more severe problem is edge label evaluation time. Several motifs exist which have $O(|l|)$ complexity. Let use assume all those c motifs are $O(|l|)$ and are used in the topmost edges of each node and thus be evaluated first. For the worst case, let us further assume none of them matches and the matching edge is always the last literal as discussed above. In this case, each node needs to

evaluate $256 + c$ edges, where c edges are in $O(|l|)$. So the evaluation process for a single node is $O(|l|) = O(c|l|) = O(256 + c|l|)$. As the parsing algorithm needs to evaluate $|l|$ nodes, we have an overall complexity of $O(|l|^2) = O(|l||l|)$.

Other cases cannot exist if no backtracking is involved. So the worst case time complexity of the PDAG parsing algorithm is

$$O(|l|^2)$$

if no backtracking is involved.

With backtracking involved, edge label matches are no longer final. If edge e matches, the associated sub-PDAG is evaluated and a mismatch during this evaluation is not considered a terminal mismatch. Instead, processing continues with next edge following after e. This means that the complexity of edge evaluation is the combination of the complexity of the sub-PDAG evaluation. Again, let us consider a PDAG with $256 + c$ edges in every node where c edges are $O(|l|)$. We can the recursively define the cost function $\mathrm{cf}(h)$ for evaluating edge with is the root of a sub-PDAG of height h.

$$\mathrm{cf}(h) = \begin{cases} 256 + c|l|, & \text{if } h = 1 \\ (256 + c|l|) \cdot \mathrm{cf}(h-1), & \text{otherwise} \end{cases}$$

This leads to the closed formula

$$\mathrm{cf}(h) = (256 + c|l|)^h$$

As such, evaluation of a sub-PDAG representing a suffix s of l has the complexity $O((256 + c|l|)^{|s|}$. We could now rightfully argue that there is some cost amortization involved, because the $O(|l|)$ motifs at suffix s only require time $|s|$. However, that does not change the complexity class. Let us set $|l| := 1$ for the inner part of the formula, than we still have $O(c^{|s|}) = O((256 + c)^{|s|})$ and as such exponential complexity. This leads to this somewhat unfortunate fact:

The PDAG parsing algorithm with backtracking has exponential theoretical worst-case time complexity of

$$O(c^{|l|})$$

In conclusion, the theoretical worst-case complexity is not better than the theoretical worst-case complexity of regex-based normalizers. It must be noted, however, that regex-based normalizers are actually $O(|R| \cdot c^{|l|})$ for the rule base R. The factor $|R|$ is irrelevant in theoretical analysis, but is important for practical considerations.

6.4.2 Practical Worst-Case Complexity

Thankfully, the structure of real-world log messages is greatly different from what we assumed for the theoretical worst case scenario:

1. the *length of motifs* is usually larger than a single byte. For example, an IP v4 address requires at least 7 bytes ("1.2.3.4") and a date motif 8 bytes ("01/01/15"). Also, literal motifs often match more than one byte. For messages containing structured data, long motifs are even more frequent. Often structured messages may consist of a single motif, for example "JSON".

   So in real-world log messages, the longest path from PDAG root to terminal node is usually much shorter than $|l|$ nodes. For example, the maximum path length was 44 in Experiment 7.2.6 (Table 14a), which represents a large enterprise rule base.

2. *motifs usually detect mismatches early* inside the matching operation. The reason is again the structure of log data as well as the specificality of the motif. Other motifs are so broad that the will never return a mismatch: for example, the "word" motif will always succeed because it simply matches data up to the next space or the end of the message. Either of this conditions will always hold, so no mismatch happens (but if the messages as whole is mismatched, backtracking may occur).

3. the *number of edges per node* is considerably smaller in practical cases. For example, over 90% of the nodes had at most two edges in Experiment 7.2.6. We suspect that the reason for this is that even free-form log messages differentiate each other relatively early in the message and so only few nodes close to the root have a larger number of edges.

4. *full sub-PDAG evaluation* will usually not happen in practice. Matching usually will stop relatively quickly because motifs in the suffix are considerably different. This is probably the result of the same effect that keeps the number of edges per node low.

Reason 4 is the most interesting in regard to our theoretical worst case result: if we can assume that the sub-PDAG evaluation terminates early, the cost of backtracking is considerably reduced. Let us assume that on average only v levels of the sub-PDAG are evaluated. Then the equations change as follows:

$$\mathrm{cf}(v) = \begin{cases} 256 + c|l|, & \text{if } v = 0 \\ (256 + c|l|) \cdot \mathrm{cf}(v-1), & \text{otherwise} \end{cases}$$

This leads to the closed formula

$$\mathrm{cf}(h) = (256 + c|l|)^v$$

This is in $O(c|l|^v)$ for a constant v. Note that this is only the complexity for the mismatch case. We still need $O(|l|^2)$ for the matching case, resulting in an overall complexity of

$$O(|l|^{2+v}) = O(|l|^2 + |l|^v) = O(|l|^2 + c|l|^v)$$

for the PDAG parsing algorithm. So under our assumptions, the algorithm is no longer of exponential complexity.

Now let us now consider the effect of motif size and early mismatch detection. This primarily affects the matching case. In this scenario, edge label evaluation time drops from $O(|l|)$ to $O(k)$ for some constant $k \ll |l|$. This leads to $O(k)$ for evaluating each edge label and results in $O(|l|^v) = O(k|l|^v)$ for the overall PDAG parsing algorithm. Note that we may have some cases where $k \simeq |l|$. Then we usually have a structured motif, which usually matches either completely or mismatches quickly. Under this assumption, our result still holds. This reasoning is most important if we consider cases without backtracking. Then, the overall algorithm complexity is

$$O(|l|)$$

under our assumptions.

6.4.3 Conclusion

The worst-case time complexity of the PDAG parsing algorithm is

$$O(c^{|l|})$$

Due to the structure of real-world log messages, we expect much better worst-case behavior in practice. Here, for a constant v, we expect

$$O(|l|^{2+v})$$

for cases where backtracking is involved and

$$O(|l|)$$

for cases without backtracking.

Out expectation cannot be formally proven, but is backed by the experiments we carried out (Sect. 7) as well as the structure of log messages.

In any case, it must be noted that the size of the rule base does not have any influence on the runtime performance. The complexity solely depends on the length of the log message. This is a big practical advantage over regex-based normalizers, where the rule base size is typically the limiting factor.

6.5 Algorithm Engineering

With current memory architectures, good cache hit rates are vital to implement highly performing code. As such, we also propose to apply several methods of algorithm engineering and have used these in liblognorm v2.

Read-only PDAG We have made all PDAG data structures read-only after initial construction. This also is desirable because we do not want to re-do the optimization stage. From an algorithm engineering perspective, it avoids costly writes to these frequently accessed data structures. Also, it facilitates parallel processing. Any write state is either kept inside the log message object or on the stack. This keeps writes at to the currently processed item. Also, it ensures that PDAG values are less likely to be evicted from cache as these are more often accessed than a single message object.

Replace Pointers with Indexes In liblognorm v1, motif parsers where stored as pointers to their implementation inside the nodes. In v2, we have replaced them with 1 byte wide indexes into a global motif parser call table. This provides two benefits:

- Pointers require 8 bytes on current hardware. So by using the 1 byte index, we can save 7 bytes for each pointer entry. Especially nodes with larger edge label tables benefit from the resulting decreased cache utilization.
- The read-only table with the pointers to motif parsers is very likely to stay in cache for extended periods of time because it is frequently being accessed and very compact.

Structure Sizes We have limited data type sizes inside PDAG data structures where useful. We used this methods:

- Flag values (for example "terminal node" status) are stored inside bit fields, usually represented by a single bit. While this requires some masking overhead for reading the value, we save some bytes inside the data structure, resulting in more nodes per cache line.

- For numerical data types, we have carefully thought about the interval that needs to be supported and have used a data type that provides this and can sufficiently fast accessed. For example, edge labels are represented by a dynamically sized table. We keep the current table size inside an 8-bit unsigned integer (native data type "unsigned char"). We could have further reduced the size by just using 6 bits and packing these into an empty slot inside a bit field definition. From the table growth perspective, 6 bits would have been sufficient. However, this data item is frequently being accessed in comparisons and it is more likely that the native "unsigned char" data type provides gain over the bit field access operation.

- We did not use compiler structure packing, as the resulting unaligned accesses can be very costly and will probably more than outweigh the gain from better cache utilization.

inlined parser tables All but terminal nodes have an out-degree of 1 or higher. Many are anticipated to have an out-degree of exactly 1. This happens when literal and non-literal motifs are intermingled, in which case empty path compression cannot be applied. The motif parser tables is dynamically allocated because of its dynamic size. The node structure contains a pointer to it. Now let us assume the table were of fixed size. Then, we could "inline" this table directly into the node, as part of the node data structure. This would have the advantage of close spatial proximity so it becomes more probable that the table is in the same cache line. On the other hand, the node structure would be large, and thus reduce the likelihood of a cache hit. We can build a hybrid solution: we can use a dynamic table, but inline a fixed size 1 element table directly into the node data structure. Almost all nodes have at least one motif parser, so we do not waste space. Many nodes are anticipated to have only one motif parser, so we keep close spatial proximity in these cases. If, however, more motifs are used, we fall back to allocating the table dynamically. In this case, we can store the pointer to the dynamically allocated area in the same memory that otherwise holds the 1-element table (as is done via a "union" in the C programming language). This method provides benefits from both approaches and does not notably increase code complexity.

array embedding of PDAG It is expected that further improvement could be gained by embedding nodes into an array instead of dynamically allocating memory for them. The reason is that the whole PDAG would be stored more compact and nodes would be in closer spatial proximity. This is especially true as each dynamically allocated memory block has a system header, which also requires some space and even puts nodes sequentially allocated into distributed regions of memory. Finally, the 8-byte node pointers can be replaced by 4-byte integer indexes, what both reduces memory requirements and decreases processing time by smaller items that need to be computed. On the contrary, pointers need to be computed from array indexes, what may outweigh that effect.

Side-note on external libraries It must be noted that runtime libraries and other libraries being used on an implementation have strong influence on the performance.

We experienced this in our implementation as well. Even after algorithm engineering, we noticed that performance was somewhat less than anticipated. We analyzed this with the "callgrind" profiler from the valgrind system [54]. It showed that runtime performance was largely dominated by functions inside the json-c library [29], up to the point that performance counters for our own code were very hard to find. Unfortunately, we needed to use this library to persist normalized data in order to keep compatible with the liblognorm v1 API. As a solution, we created a new project for optimizing json-c. This was successful and enabled us to greatly improve library performance. We needed to make some incompatible changes and so had to release our optimized version of json-c under the new name of libfastjson [43]. It must be noted that libfastjson also brought some notable speedup to liblognorm v1.

Another area of potential improvement is the dynamic memory allocation subsystem. The standard system provided on Linux is outperformed by the jemalloc project [18]. We experimentally built liblognorm with jemalloc support and noticed some performance improvement. However, we stayed with the default allocator in order to base our method evaluation on a system being as standard as possible. For very high demands, it is suggested to switch to building with jemalloc.

6.6 Parallelization

In order to find a suitable appropriate parallelization method, we need to consider the use case: we want to process log messages under real-time conditions. If the message volume is low, we obviously meet this goal without special effort and do not need

speedup from parallel processing. So a sufficiently large workload is a necessary precondition to consider parallelizing. Further, we normalize messages to supply them to some intelligence tools. Some of these tools may consider message sequence important. For these, the normalization process should not re-order messages. It must be noted, though, that log message often occur in a sequence different from the sequence in which they were emitted by the originator. A simple example is syslog messages sent over UDP protocol. By UDP design, there is no provisioning inside the protocol for keeping message sequence. Neither is there any provisioning inside the syslog protocol. As such, messages often arrive slightly out of sequence. Thus analytics tools should be able to handle this situation in order to work fully correct. If we know we normalize message for well-behaved tools, we can relax the requirement to preserve sequence.

The PDAG is read-only after construction. Also, the parse algorithm (7) only modifies state attached to the log message currently being processed. As such, we have no interdependencies other than sequence between messages. This enables us to select from these parallelization methods:

- *fine-grained parallelism* can be found inside the PDAG parsing algorithm. An approach would be to evaluate more than one search path in parallel.

- *coarse-grained parallelism* can be found easily by running the PDAG parsing algorithm on multiple messages in parallel. For this, we need to split the log message set into independent partitions and process these individually.

Looking at the use case, it is clear that using coarse-grained parallelism is preferable: it has lower overhead and promises near-linear speedup because the partitions are independent of each other and very little coordination is required.

We call the partitions (message) *batches*. Each batch is processed sequentially by one thread. However, multiple threads concurrently process multiple batches. As the consumer expects to receive a sequential stream of messages, we must serialize the batches before passing messages to it. This leads to a design with one input queue, which receives messages to be normalized, a pool of worker threads which do the actual normalization, and one output queue, which receives the finished batches and serializes them for consumption by the destination. The work flow is shown in figure 10.

Preserving sequence, if required, can be done by keeping a monotonically incrementing batch ID. It is assigned on batch creation. Then, the batch is processed and once it is finally completed, given to the output queue. The output queue re-

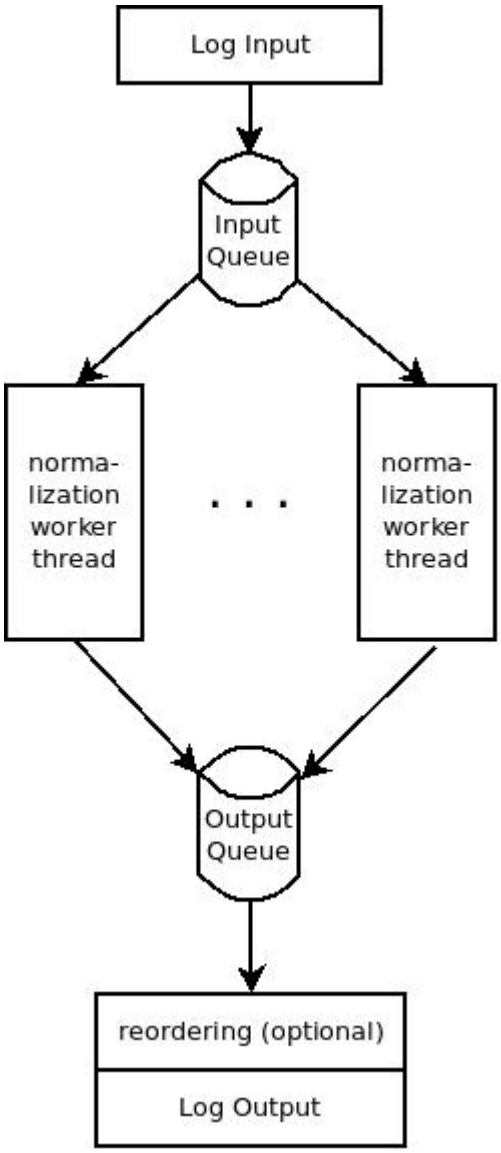


Figure 10: Workflow for parallel normalization of log messages.

The basic unit of work is the batch, which consists of multiple messages. The actual number of messages inside a batch is decided when dequeuing from the input queue (up to a configured maximum). The reordering step after the output queue is optional and only needs to be done if the destination does not accepts slight reordering of messages.

ceives the batch and emits it, and possibly other waiting batches, to the consumer if suitable.

With a sufficiently large batch size, the overhead for queue locking is negligible. Sizes around 1024 or above seem reasonable for busy systems. Unfortunately, large batches can introduce non-tolerable delays in low-traffic situations. To mitigate this problem, we introduce a mechanism for dynamically selecting batch sizes. This is based on a similar method as used by our rsyslog [25] project. An upper limit u is specified for the batch size. If a new batch needs to be extracted from the input queue, up to u messages are obtained from it. However, if only $n < u$ messages are currently queued, only these n messages are extracted and batch construction is completed. The algorithm does not wait for the arrival of the missing $u-n$ messages. This obviously results in a smaller batch size and can potentially lead to additional overhead in batch processing. However, this scenario can only happen under low-load conditions, because otherwise the input queue would have held u or more messages. When the system receives larger load, the input queue will build up and so under heavy load steady-state conditions, always batches of u messages will be extracted. This way, batch processing adjusts itself to current demands and guarantees low latency under low-load conditions while being able to keep the overhead minimal at high load.

Algorithm 8 details worker thread processing. Algorithm 9 details consumer side processing if preserving order is required. We do not provide details for the input queue enqueue and output queue dequeue algorithms (non-reordering case), as these are straightforward standard algorithms.

Note that the parallel algorithm can also easily be adapted to large non-shared-memory, message-passing clusters, for example via the Message Passing Interface (MPI) [57, Sect. 6]. As a rough sketch, if we have $n > 2$ processors, rank 0 can run the master node, which does

- loading the PDAG
- broadcasting it to all other nodes
- reads input data, builds batches and sends each one to a compute node (ranks $[1..(n-1)]$).

Note that the read-only nature of the PDAG after load makes it very simple to broadcast it once during initialization. The compute nodes carry out work described above for the normalization worker thread. Once finished with a batch, they send

Algorithm 8 parallel normalizer worker thread

while worker thread shall not terminate **do**
  wait for data in input queue {signaled by producer}
  lock input queue mutex
  n = number of messages in queue
  if $n < u$ **then**
    dequeue n messages from queue to batch B
  else
    dequeue u messages from queue to batch B
  end if
  assign batch sequence ID $i + 1$ to B {if required}
  unlock queue mutex
  {main worker activity begins here}
  for all log message $l \in B$ **do**
    normalize l
    store normalized result in B
  end for
  {main worker activity done}
  lock output queue mutex
  enqueue all normalization results from batch
  signal output queue not empty to consumer
  unlock output queue mutex
end while

Details of mutex and signaling behavior are not described and need to be matched to the actual system used, for example POSIX threads. Most importantly, an implementation need to re-query predicates like "queue data present" to ensure proper operations. The batch sequence ID i is maintained inside the queue data structure.

Algorithm 9 reordering algorithm

```
while system shall not terminate do
  wait for data in output queue {signaled by worker}
  lock output queue mutex
  dequeue batch B from output queue
  insert B into 𝓑
  unlock output queue mutex
  while ID of batch on head of 𝓑 equals i_l + 1 do
    remove head batch B from 𝓑
    i_l := i_l + 1
    for all log message l ∈ B do
      provide l to consumer
    end for
  end while
end while
```

The algorithm keeps the batch ID i_l of the last batch sent to the output destination. This needs to be properly initialized on startup (details not shown for clarity). The utility list $\mathcal{B}$ is a list of all batches which cannot be processed because they wait for a batch to arrive with a lower batch ID. $\mathcal{B}$ is ordered on ascending batch ID, that is the lowest batch ID is at the top of the list, the highest at the tail.

the result to the node with rank n, which re-orders and outputs them as described above.

7 Experimental Verification

7.1 Method

We have two classes of experiments:

- *timing experiments*, which time runtime performance of an implementation. Absolute results are obviously sensitive to the lab environment.
- *structural experiments*, which analyze the structure of the parsing process or the PDAG. These are insensitive of the lab environment, because given the same input data, those experiments will always lead to the exact same results.

7.1.1 Timing Experiments

To keep timing experiments consistent, all were executed in a controlled environment. We used the same lab machine for all experiments. We used a single machine only, because no machine specifics are being used, so the results are comparable to each other. We used a real machine, not a virtual environment. The machine had the following specification according to Linux tool "lshw":

CPU	Intel Core 2 Quad
Cores per CPU	4
CPU-Sockets	1
Clock Speed	2,40 GHz
L1-Cache	64 KiB
L2-Cache	4 MiB
Mainboard	Gigabyte 965P-DS3P
Main Memory	8 GiB
Disk	RAID 5; 3 WDC WD5000AAKS-2 (ATA, 500 GiB)

The operating system was Ubuntu 14.04LTS. Whenever possible, we used distribution provided packages for software components. When we had to build components ourselves, we built with default C compiler (gcc) with optimization set to "-O2". We used the default memory allocator. Both liblognorm v1 and v2 were build with libfastjson, commit hash f96b6c6.

This lab machine was dedicated to the experiment. No other processing in parallel was permitted. Besides these restrictions, timing can be slightly different due to

- background actions, like the operating system checking for package updates
- different disk access patterns, e.g different distribution of blocks in dynamically updated files (including the OS page file)
- operating system scheduling policies, e.g. frequency of change of processor core (which results in higher cache access times after each switch)

In order to limit these factors, output was only generated if inevitable. We did this because we are interested in the computational performance of the algorithm. If we write output, we get a strong dependency on disk speed, which blurs experiment results. Avoiding output was not an option with grok and syslog-ng, which do not provide an option to suppress it. For consistency, we also generated output with liblognorm in those cases. Output was always written to `/dev/null`, which means it was immediately discarded after being generated. So the time overhead for output generation and API calls is included in the experiment results, only the actual disk speed is not reflected.

To further limit the influence of environmental fluctuations, each experiment was executed 25 times. This was done automatically via a script. There was a pause of 10 seconds between executions which was given to permit the system to settle back to idle state. The time for each execution was taken and the average computed. We also manually checked for strong variations in execution times, which would be an indication of problems with the experiment. We were prepared to investigate such cases, but did not discover any.

The computed average of the 25 controlled execution times is then used as test result. Detail results are shown in tables in the Appendix.

7.1.2 Structural Experiments

Structural experiments explore PDAG properties and are not timing-sensitive. They can be reproduced on any type of machine. Consequently, no specific controlled environment was used for them. We manually verified that the result was valid by doing at least one control run and verifying the correctness of the program output. Structural experiment data is generated by special instrumentation code inside liblognorm. In the following, this structural data is presented. Again, the Appendix contains detail data.

7.1.3 Test Data Sets

We use log data sets from those described in Sect. 4.2. Note that these data sets are real life data and contain some malformed records. For use in our experiments, we have removed those malformed records. From those cleaned-up data sets, we have derived the actual data sets use in the experiments.

The following data sets are used inside the experiments:

PIX2m data set This data set is based on the "PIX data set" described in 4.2. The original data set contains duplicate time stamps and some malformed records which possibly stem from networking problems. We have processed this data set as follows:

- removed duplicate timestamps and other extra header information. The result is the format usually found in Linux log files. Most importantly, this is the only format that the syslog-ng tool understands, at least without deep configuration.
- removed malformed log records. This was necessary only for a small number of log messages. The result is a data set the parsed cleanly with all tools we use in our experiments.
- duplicated some records to achieve a data set of exactly 2 million records. As log normalization itself has no interdependencies between log messages, we picked random messages for duplication.

The result of this transformation is called the "PIX2m data set" and is 289MiB in size.

PIX20m data set This data was build based on PIX2m by simply duplicating it 10 times. So it now consists of 20 million records. The larger record count is necessary for better performance evaluation as the fast tools do not run sufficiently long with the PIX2m to provide good results. This data set has a size of 2.9GiB.

PIX20m-single First, we identified a rule r that was frequently matching records inside the PIX2m data set. The rule r matched approximately 288 thousand messages. Then, we extracted those messages from PIX2m and duplicated them multiple times to reach a 20 million message data set. This is what we call PIX20m-single. Note that while the whole data set matches a single rule, the actual messages are different (except for duplication).

ent55m data set This data set is based on the "enterprise combined data set" described in 4.2. The original data set contains some malformed records which most probably stem from networking problems or failure of the recording applications. We have removed these malformed records and received a slightly smaller 25GiB data set of 55,057,137 messages. We have not done any other modifications to the original data set.

7.1.4 Experiment Rule Bases

We used a consistent set of rule bases for each test.

PIX For the PIX family of data sets, we created different rule bases for use with

- liblognorm v2 (37 rules)
- liblognorm v1 (37 rules)
- syslog-ng patterndb (37 rules)
- grok (34 rules)

The slight differences in rule numbers stem back to different motifs being used in the respective description languages. These rule bases match all records inside the PIX family of data sets and were consistently used for all tests involving them. It must be noted that the liblognorm rule bases were intentionally constructed in a way that in made them least specific: log messages often contain a message type identifier early in the message (the so-called syslog "TAG"). This is a literal string, which usually identifies a class of formats. The syslog-ng tool, for example, bases its normalization on that TAG and selects the rules based on TAG. This can result in fewer conflicting rules and as such reduce backtracking. For liblognorm, we have used the generic "word" motif instead of specific TAG values, so this in theory makes parsing slower, what was our intent to fully validate the performance of the algorithm under hard conditions. Note that some experiments made modifications to the rulebase. If so, details are contained in the experiment description.

ent Similarly, we used a rule base for the ent data set. Here, only rule bases for liblognorm v1 and v2 were created. These rule bases were contributed from the same user who also contributed the combined enterprise data set. Both rule bases consist of 1,124 rules which are representative for large enterprise normalization needs.

7.2 Experiments

7.2.1 Scalability of Regular Expression based Normalizers

Description We used the PIX20m-single data set where all messages match rule r. We ran experiments with different rule bases:

- r is the only rule inside the rule base (1/1)
- r is the first rule in a rule base of 34 rules (1/34)
- r is the last rule in a rule base of 34 rules (34/34)

Note: in the (x/y) description, x denotes the position of r inside the rule base and y denotes the size of the rulebase.

Results As can be seen from table 1 on page 71, the rule base structure has strong influence on performance. At (1/34), more than 18k messages per second (mps) could be processed, but at (34/34) only less than 1k mps could be processed. This difference between is exactly as expected and shows the typical behavior of regex-based normalizers.

The (1/1) 103k mps vs. (1/34) 18k mps result is a bit surprising: in theory, a regex-based normalizer iterates over all regexes and terminates as soon as the first match is found. So if all messages match the first regex, it should not make any notable difference in performance if more regexes follow the first matching rule. Note that grok has a setting that tells whether or not processing should continue after the first match (`break-tf-match:true`), which we enabled. We do not have enough insight into the way grok operates to provide a good explanation of this situation. So this cannot be the cause for the performance difference. We assume that some reordering of rules happens.

This kind of makes the experiment results in regard to the position of rule questionable. However, the core intent of this experiment was to show that regex-based normalizers scale very badly in regard to growing rule base sizes and the results clearly shows that this is the case.

7.2.2 Scalability of liblognorm

Description This experiment is similar to Experiment 7.2.1. We used the PIX20m-single data set where all messages match rule r. We ran experiments with different rule bases:

- r is the only rule inside the rule base (1/1)
- r is the first rule in a rule base of 37 rules (1/37)
- r is the last rule in a rule base of 37 rules (37/37)

Note: in the (x/y) description, x denotes the position of r inside the rule base and y denotes the size of the rulebase. In comparison to Experiment 7.2.1 note that liblognorm needs a slightly larger rule base to process the data set.

Results As can be seen from Table 2 on page 72, the performance did not notably change with the number of rules. We achieved around 381k mps in all cases. For an equal number of rules, position of the matching rule inside the rule set is irrelevant. The small difference between (1/1) 386k mps and (1/37) 381k mps is explained by the PDAG size. With more rules, more literal motifs need to be evaluated, what very slightly decreases performance.

7.2.3 Comparison of different Normalizers

Description We use the PIX20m data set and have created rule bases for liblognorm v2 and v1, syslog-ng, and grok. All of these rule bases match all messages from the data set.

Results As can be seen from Table 3 on page 73, the performance of the grok regex-based normalizer (2k mps) is much worse than that of the search tree based normalizers (80/150/192k mps). Even for such a small rule base solely aimed at normalizing firewall log messages the performance is insufficient for realtime processing of any but very low message rates.

The search tree based normalizers all achieve message rates that make them suitable for realtime processing, with syslog-ng being able to handle only a more moderate workload. Liblognorm v2 performs notably better than liblognorm v1, which shows that the new algorithm not only creates a more compact parsing data structure but also performs faster.

7.2.4 Performance in Relation to Rule Base Structure

Description We use the PIX20m data set and the rule base created for it. We amend this rule base by the one created for the ent55m data set. The, we do one sub-experiment where the rule base consists of the PIX rules followed by the ent55m rules and another one where the ent55m rules are followed by the PIX rules. Note

that this is an extreme scenario: the PIX rule base has 37 rules and the ent55m ruleset has 1124 rules none of which match the input data. All matching is done by the PIX rules. The intent of this experiment is to show how far rule base structure affects performance.

Results As can be seen from Table 4 on page 74, the performance is somewhat affected by the load order of the rule base. If the matching rules are loaded last, we see roughly a 20% increase in runtime. The reason is that load order influences edge order and as such the number of motif evaluations during the parsing algorithm. In an extreme case like here, where only 3% of the rules match and they are all loaded last, edge order becomes somewhat important. In essence, we have "polluted" PDAG nodes with dead edges that are never needed. In practice, matching is more evenly distributed among nodes, so the performance loss is expected to be even milder.

It must be noted that it would be a wrong conclusion to split rule bases to avoid the small performance loss. If splitting requires a selection operation on which rule base to use, this selection operation may be more costly than running a combined rule base. Also, if such a split is possible, it could possibly be used to change the rule base in a way that the PDAG itself can rely on that split.

7.2.5 Scalability in Relationship to Message Size

Description Here, we evaluate how message size influences performance. We have fabricated both the data sets as well as the rule base in a way that shows the relationship in a simple but closely controlled case.

Real-world log data is hard to analyze in regard to message sizes. The reason is that a specific message type usually fluctuates only relatively little in size. So large size fluctuations usually mean that we deal with different message types, which in turn have different (and usually more) motifs. As such, getting a solid analysis out of real-world log data on this topic is very difficult. This is the prime reason why we used these fabricated data in this experiment.

We use a single rule inside the rule base which extracts four "word" motifs and has a single space character between each of them. The data sets exactly match this rule and have four equal-length part. The messages are all the same and are 20, 200, 800, and 2000 bytes in size.

Results As can be seen from Table 5 on page 75, the performance of PDAG parsing depends on the message size. This is to be expected because motif parsers

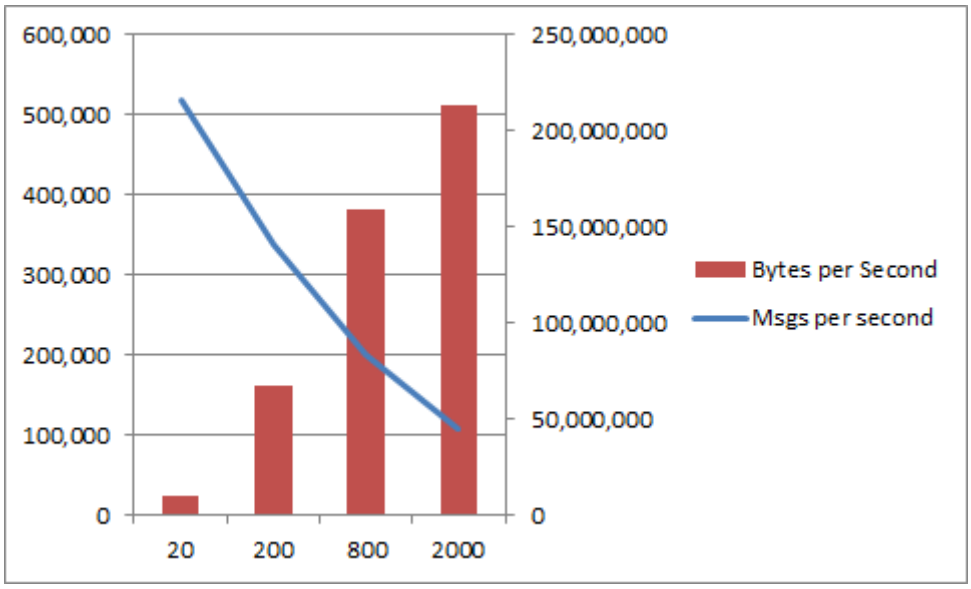


Figure 11: Performance in Relation to Message Size.

need to process more data with increasing size and also more data needs to be put into normalized form.

Figure 11 on page 62 plots the relationship. At 20 byte messages, we can process approximately 518K messages per second. If the message is 100 times larger, we can still process roughly one fifth of that rate (106K messages per second). Consequently, the amount of data processed per second (shown in red) increases with increasing message size. This is also expected, because there is some fixed processing time required for the parsing algorithm itself.

Again, we caution against using this result on real-world data without proper consideration of the different nature of real-world log message.

7.2.6 Evaluation of Backtracking

This is a structural experiment which aims at generating insight into how the PDAG parse algorithm performs in practice. Most importantly, we want to have information about the backtracking behavior, which is important for upper bound performance behavior.

In order to support this, the liblognorm v2 implementation has been amended with code that permits to gather

- *static structural data*, which gives insight into properties of the PDAG itself, like the number of nodes and motifs used.
- *dynamic data*, which gives insight into the parsing algorithm itself. Among

others, this records the exact PDAG walk that was being used during parsing of a message, and how often backtracking was necessary during the walk.

The code outputs both summary records as well as statistics for individual rules. At a very verbose level, dynamic data can also be amended to each message being processed. This instrumentation is turned on with command line options. We have used the `-S` option to the `lognormalizer` utility, which outputs structural data as well as dynamic summary data and per-rule statistics. This output is still very verbose, so we have only included the most relevant information in tables shown inside the thesis. The full, verbose, information is available upon request or may be regenerated by redoing the experiment again based on our input data.

Relatively small PDAG

Description This experiment uses the PIX2m data set and accompanying rule base. It shows behavior for a smaller PDAG.

Results The result can be seen in Table 13a on page 76. The most interesting finding is that there was no backtracking at all during this experiment. This is especially interesting as the rule base was deliberately constructed less specific than it could be (see 7.1.4).

Further, Table 12b on page 70 shows that almost all nodes have an outdegree of less than 2, and 179 of 230 have an outdegree of exactly 1. This proves that inlining of parser tables into the node as part of algorithm engineering is a useful technique (see 6.5 on page 48).

Large PDAG

Description This experiment uses the ent55m data set and accompanying rule base. It shows behavior for a large PDAG.

Results In this experiment, we had some backtracking, see Table 15a. However, still the majority of messages (over 97%) were parsed without any backtracking involved. For the remaining ones, there were on average less than 6 backtracking operations required. Judging from the path length (Tables 15b, 15c) backtracking never caused excessive computational cost: the average path length was below 18 nodes and more than 45 nodes needed to be walked only in very few cases. The worst-case path length was just 101 nodes.

Table 14b on page 77 also shows that almost all nodes have an outdegree of less than 2, and 70% (4121 of 5860) have an outdegree of exactly 1. This again proves our algorithm engineering approach (see 6.5 on page 48).

7.2.7 Execution Profiler Analysis of liblognorm v2

Description This experiment provides insight into the flow of execution during the normalization process. Most importantly, we wanted to know which parts of the code use most of the execution time. To do so, we use the valgrind profiling tool "callgrind" and execute the liblognorm v2 normalizer under its control. We use the ent55m data set, but only the first 200,000 (200k) records from it. This is because the profiler slows operations dramatically down and may also get integer overflows on extended runs. We manually verified that the subset contains sufficient diversity, even those this is not critical to the correctness of this test. We used the kcachegrind call graph viewer tool [56] to interpret callgrind results.

Results Figure 16 on page 79 shows an image of the kcachegrind viewer (it has no export capability). The tool is configured to sort functions based on the percentage that the execution spent in themselves (column "Self"). This is purely time spent in the function, including any inlining done by the optimizer but excluding any functions called (the latter is included in the "Incl." column). The "Function" column has the name of the function in question and "Location" (truncated) shows in which software component the function is located.

We have analyzed only functions which contribute at least 1% to the overall execution time. Table 17a on page 80 tell us that those functions summed up to over 81% of the total execution time. It also tells us that liblognorm itself only used 17% of the time, while libfastjson needed 34% and the memory allocator subsystem needed 30%. Accompanying Diagram 17b shows that the execution time of our algorithm is dominated by those third-party components.

One must further note that this result is obtained after we have replaced the original json-c library by libfastjson, because json-c required considerably more processing time.

In conclusion, it is very important to keep aspects other than the core algorithm and its optimization in focus when further improvements are considered. It obviously is more promising to help speed up those third party components, which requires 64% of execution time than to speed up the 17% liblognorm requires. Of course, liblognorm uses this components in order to perform specific tasks, so they cannot simply be removed. However, optimizations can be possible by, for example:

- improve the performance of those third-party components. We have actually done this by developing libfastjson based on json-c. Also, replacing the system default memory allocator by jemalloc usually results in increased performance.

- algorithm engineering can be used to improve some aspects of the implementation. For example, we could reduce the number of memory allocations by using different allocation methods. In our concrete example, however, there is limited capability to do so, as in-depth analysis shows that most memory allocations stem back to libfastjson. As a side-note, this technique has very successfully been used in the optimization of libfastjson and so is known to work well.

- in the practical implementation, APIs are important. We could remodel liblognorm's APIs so that, for example, it does not provide JSON back to callers. This could probably speed up liblognorm, but at the cost of compatibility breakage.

8 Results

Our goal was to develop an algorithm that is sufficiently fast for real time normalization under practical conditions.

As we have seen in Sect. 6.4, we unfortunately cannot proof that our algorithm performs better than regex-based normalizers. However, we have argued why we heuristically expect much better performance and experimental results backs that expectation.

A key property of our algorithm is that its performance is independent from the rule base size. We have seen in Experiment 7.2.2, for a smaller, yet practical, rule base size of 37 rules that liblognorm performance does not notably change by rulebase growth. Neither is the position of a matching rule inside the rule base of importance. We have further verified this with Experiment 7.2.4 which uses a much larger rulebase of 1,161 rules. Here, we see that the position of rules actually has some effect. Out of the total rules, only 37 matched. Processing was by a factor of 1.2 slower if we moved those 37 matching rules to the bottom of the rulebase versus the top. It is also interesting to compare this to Experiment 7.2.3 (liblognorm part), in which the same data set and the same matching 37 rules were used. The difference between both experiments is only that the non-matching 1,124 rules were not not included in 7.2.3. The first thing we notice is that we get almost exactly the same performance compared for both rule bases when the 37 matching rules

are on top of the rule base (the very small difference can easily be explained by influences on the lab machine, like scheduling order). This effect proves the point that the PDAG is still somewhat sensitive to load order, as we have explained in the Experiment descriptions for each experiment individually. Nevertheless, the effect on rule position and rule base size is marginal: while the rule base grew by a factor of 31.4 in Experiment 7.2.4, the execution time just increased by a factor of 1.2. So for practical purposes we can say that the algorithm performs almost independent from rule base size and rule position.

Most importantly, the small effect rule base structure and growth has on performance is in sharp contrast to regex-based normalizers, as we have seen in Experiment 7.2.1. Consequently, our algorithm performs much better. Take for example Experiment 7.2.3 which bases on real-world firewall log messages. Here, regex-based grok was only able to process 2k mps, whereas liblognorm v2 could process 192k mps. One might argue that we can use the same parallelization methods we have proposed for our algorithm in Sect. 6.6 in order to improve the overall speed of a grok-based system. This seems possible. However, we would need roughly 100 times the hardware resources liblognorm needs (assuming linear parallel speedup). So from a practical point of view this is not a desirable solution.

Also from Experiment 7.2.3 we learn that the syslog-ng normalizer, also influenced by the search tree paradigm, performs at 80k mps, much closer to the performance of liblognorm v2. The liblognorm v1 prototype performed at 150k mps which is good for many use cases but still slower than the 192k mps the new PDAG structure in liblognorm v2 can handle.

Absolute performance numbers need to be considered with care. For example, we have seen that liblognorm v2 can handle 380k mps in Experiment 7.2.2 but only 192k mps in Experiment 7.2.3. This difference results from differences in input data, most notably the average log message size as well as on the motif structure. Experiment 7.2.2 used a single, smaller, message from the full data set used in Experiment 7.2.3. To proof that effect, we did Experiment 7.2.5 in which we can clearly see that effect message size has on performance, even though a sub-linear one. We also need to be a bit careful if we apply the findings from Experiment 7.2.5 to real-world log messages, because

- real data usually has more motifs in larger messages. So processing time increases not only for effects explained in Experiment 7.2.5 but there are also increases due to processing a larger motif parser (edge) set.

- the likelihood of backtracking increases. For example, in Experiment 7.2.6 we

see that the PIX20m data set with its 37 rule rule base requires no backtracking whereas the ent55m data set with its 1,124 rule rule base requires 3% of backtracking.

In any case, the degradation of performance due to these circumstances is far from the theoretical worst case exponential time complexity. For practical purposes, the algorithm can be considered to scale linearly depending on the message size and be almost constant in regard to rule base size and structure. This corresponds to what we expect by theoretical argument (Sect. 6.4.3). Also, based on Experiment 7.2.7 we suspect that part of the performance degradation is related to third party components, which could be optimized to gain even better results.

This good performance is offered with sequential processing. To further increase performance, we also have shown a simple and efficient algorithm to partition large workloads for parallel processing (Sect. 6.6). If desired, it can keep message sequence intact and does so with very limited additional overhead. The algorithm works without introducing additional latency and is self-optimizing based on workload, which is important for real-world cases where workload varies greatly over time periods. As we use very coarse-grained parallelism, it is expected that this method can be used to run many instances of the parsing algorithm in parallel. The practical limit depends on the reordering need as well as available hardware. On a current typical enterprise multi-processor, multi-core machine, it is expected that all cores (16, for example) can be used with close to linear speedup. The parallel algorithm can also be used with large non-shared-memory, message-passing clusters.

Our algorithm implementation in liblognorm v2 does not include the parallel processing algorithm. There are two reasons for this:

- liblognorm is a library and intended to be used with any threading model a potential caller might already employ. Implementing multi-threading at the library level without affecting caller's choices in regard to threading models and abstraction libraries is complex and worth a project in itself.

- a prime user of liblognorm is rsyslog, which already employs multi-threading for workload processing. Rsyslog provides the necessary support to automatically execute multiple normalizer invocations in parallel, what provides almost the same benefits like native threading support. This can easily be used because the PDAG is read-only after construction.

As we did not implement multi threading, we could not experimentally proof our expectations in regard to the achievable speedup by the proposed parallelization

method. However, we did some experiments with liblognorm inside rsyslog, which makes us believe that the speedup actually is near linear.

9 Conclusion and Outlook

As discussed in Sect. 8 our algorithm has shown to be able to provide sufficient speed for real-time performance: a single instance can process message rates of 100k messages per second or more on practical workloads and on a very modest machine. The algorithm scales well and is only mildly dependent on the size of the rule base. The message size has a somewhat larger impact but this was expected and as we have argued in Sect. 8 is a property that it shares with the other methods we have examined. Furthermore, the algorithm can easily be parallelized and is expected to offer near-linear speedup with growing number of processors. So we expect it to be able to handle very large workloads on real deployments. Most importantly, experimental results correspond to theoretical expectations.

Besides our primary goal of real time capability, the new algorithm and its implementation provides additional benefits. Most importantly, the PDAG architecture supports custom data types via disconnected components. This is an important feature from a practice point of view. The ability for users to easily extend the available motifs is, for example, a key advantage of grok and a key reason why grok is being used even though it is extremely slow in comparison. From a maintenance perspective, the new implementation is actually simpler than its predecessor, the liblognorm v1 prototype, because its processing is more generic.

Results (especially Experiment 7.2.7) suggest that the implementation can further be speeded up. However, when doing so one should put the the focus on supporting libraries and algorithm engineering rather than on improving the theoretical approach. The reason is that in the current implementation the majority of time is spent outside of the core algorithm (Fig. 17b). This argument is backed by the fact that we speeded up and changed the JSON support library, because the one originally used was too slow to actually measure core liblognorm performance. See the result description of Experiment 7.2.7 for further details of how performance could potentially be improved without changing the algorithm itself. Work on core algorithm improvements should be done only when it can be proven that the core algorithm becomes a bottleneck.

For potential further work we see these promising areas:

- implementation optimization, as just suggested.

- the proposed parallel processing mode should be considered if the algorithm shall be used by applications which do not natively support workload partitioning.

- as a loosely related item, we think that some of our findings on log file classification and formats can be useful for clustering log records. This has useful applications in the try to semi-automatically generate rule bases.

- the current state can be very useful for creation of anonymization tools. A core function for an anonymization tool is to detect motifs and decide for which motifs actual values need to be anonymized (e.g. IP adresses need to, byte counts probably not). Our current implementation provides the ability to amend output data so that an anonymizer can easily obtain motif information.

Finally, we have also created a repository of log messages usable for research purposes and provided a classification of message types. We hope that both will be useful for research in the field and plan to maintain and extend the repository in the future.

A Detail Data from Experimental Verification

The following tables contain the detail results from experimental verification. They are discussed in section 7 on page 55.

Figure 12: PDAG structural data (PIX2m Data Set)

(a) Objects, overall

nodes	230
terminal nodes	37
edges (motifs)	229
longest path	23

(b) nodes by outdeg

outdegree	node
0	35
1	179
2	12
3	1
4	1
5	1
14	1

(c) motif counts

motif	occurs
literal	119
date-rfc3164	1
number	20
ipv4	29
word	22
rest	2
quoted-string	6
duration	2
cisco-interface-spec	24
string-to	1
char-to	3

Table 1: execution times (in seconds) for grok in relation to rule base structure

rule structure:	**1/1**	**1/34**	**34/34**
	18.95	107.74	2865.92
	18.99	107.80	2867.90
	19.00	107.86	2868.54
	19.01	107.90	2868.55
	19.09	108.02	2869.58
	19.10	108.09	2870.69
	19.11	108.12	2870.76
	19.13	108.12	2871.11
	19.17	108.19	2871.12
	19.23	108.21	2871.13
	19.25	108.22	2871.24
	19.34	108.27	2874.40
	19.37	108.28	2874.45
	19.41	108.33	2874.74
	19.43	108.34	2876.45
	19.44	108.35	2876.70
	19.44	108.38	2878.35
	19.48	108.45	2879.09
	19.49	108.47	2880.87
	19.51	108.51	2880.98
	19.52	109.15	2881.10
	19.52	110.14	2884.60
	19.61	111.53	2891.10
	19.69	111.62	2906.92
	19.84	111.72	2924.13
average	19.32	108.71	2878.02
msgs per sec	103,494	18,397	695

Table 2: execution times (in seconds) for liblognorm v2 in relation to rule base structure

	1/1	**1/37**	**37/37**
	50.95	51.81	51.96
	51.09	51.85	52.00
	51.15	51.86	52.00
	51.27	51.87	52.00
	51.28	51.91	52.00
	51.30	51.91	52.05
	51.30	51.99	52.08
	51.35	52.02	52.14
	51.60	52.02	52.21
	51.62	52.05	52.29
	51.72	52.08	52.31
	51.73	52.11	52.37
	51.74	52.16	52.38
	51.76	52.21	52.41
	51.78	52.21	52.42
	51.83	52.46	52.49
	51.83	52.48	52.50
	51.86	52.56	52.54
	51.88	52.60	52.55
	51.98	52.70	52.57
	52.08	52.94	52.60
	52.42	53.04	52.71
	52.55	53.15	53.19
	52.58	53.71	53.19
	52.63	53.99	53.26
average	51.73	52.39	52.41
msgs per sec	386614	381770	381615

Table 3: Comparison of different Normalizers

	liblognorm v2	**liblognorm v1**	**syslog-ng**	**grok**
	103.26	133.49	245.90	882.80
	103.33	133.65	247.25	893.37
	103.40	133.78	247.42	893.40
	103.49	133.79	247.53	893.51
	103.57	133.84	247.89	893.85
	103.63	133.84	247.90	893.95
	103.71	133.84	248.22	894.36
	103.72	133.88	248.25	894.43
	103.76	133.93	248.28	894.67
	103.83	133.96	248.33	894.71
	103.85	133.99	248.33	894.75
	103.85	134.07	248.38	894.75
	103.87	134.08	248.38	894.93
	103.90	134.11	248.53	894.97
	103.90	134.12	248.55	895.00
	104.03	134.13	248.88	895.01
	104.06	134.14	248.99	895.04
	104.06	134.14	248.99	895.08
	104.13	134.15	249.12	895.13
	104.14	134.15	249.16	895.20
	104.17	134.18	249.28	895.28
	104.17	134.21	249.33	895.44
	104.18	134.22	249.33	895.46
	104.24	134.22	249.37	895.65
	104.24	134.22	249.41	895.72
average	103.88	134.03	248.55	894.74
msgs per sec	192,521	149,224	80,468	2,235

All execution times are shown in seconds.

Table 4: Execution times (in seconds) in Relation to Rule Base Structure

rule order	PIX+ent55m	ent55m+PIX
	102.48	123.78
	102.55	123.89
	102.71	123.96
	102.86	124.03
	102.93	124.06
	103.02	124.13
	103.16	124.22
	103.18	124.43
	103.19	124.43
	103.30	124.50
	103.30	124.52
	103.31	124.63
	103.33	124.80
	103.34	124.85
	103.43	124.86
	103.46	124.92
	103.54	124.93
	103.55	124.96
	103.57	125.22
	103.77	125.26
	103.85	125.30
	104.02	125.31
	104.15	125.49
	105.38	126.98
	113.72	127.89
average	103.80	124.85
msgs per sec	192,671	160,187

Table 5: Execution times (in seconds) in Relation to Message Size

bytes/msg:	**20**	**200**	**800**	**2000**
	3.76	5.81	9.73	18.26
	3.76	5.83	9.83	18.36
	3.77	5.83	9.85	18.50
	3.78	5.84	9.88	18.53
	3.80	5.86	9.91	18.56
	3.80	5.86	9.91	18.60
	3.81	5.87	9.96	18.62
	3.83	5.88	9.97	18.63
	3.84	5.88	9.98	18.65
	3.84	5.88	9.98	18.66
	3.85	5.89	9.99	18.67
	3.85	5.90	10.00	18.67
	3.85	5.90	10.01	18.67
	3.86	5.91	10.01	18.69
	3.87	5.92	10.04	18.71
	3.88	5.93	10.05	18.71
	3.88	5.93	10.05	18.73
	3.88	5.93	10.10	18.86
	3.89	5.93	10.11	18.92
	3.89	5.95	10.13	18.93
	3.90	5.95	10.16	19.01
	3.92	5.96	10.16	19.10
	3.92	5.97	10.20	19.20
	3.94	5.98	10.25	19.21
	3.98	6.03	10.59	19.22
average	3.85	5.90	10.03	18.75
msgs per sec	518,941	338,707	199,322	106,685
bytes per sec	10,378,820	67,741,400	159,457,600	213,370,000

Figure 13: PDAG dynamic data (PIX2m Data Set)

(a) Backtracking Count

Backtracking count	messages
0	2000000
avg	0
max	0
msgs total	2000000
msgs with backtr.	0
% backtracking	0

(b) Path Lengths

length	messages
avg	16.114158
max	23
10	11
11	9806
12	3934
13	141149
14	551509
15	648384
16	62
17	333290
18	5926
20	594
22	16865
23	288470

Figure 14: PDAG structural data (ent55m Data Set)

(a) Objects, overall

nodes	5860
terminal nodes	1108
edges (motifs)	5859
longest path	44

(b) nodes by outdeg

outdegree	node
0	1068
1	4121
2	512
3	81
4	33
5	21
6	7
7	6
8	1
9	3
11	1
13	2
14	1
18	1
21	1
25	1

(c) motif counts

motif	occurs
literal	3461
repeat	39
date-rfc3164	5
number	284
float	6
hexnumber	1
kernel-timestamp	1
whitespace	86
ipv4	322
ipv6	1
word	479
rest	163
date-iso	1
time-24hr	1
duration	2
cisco-interface-spec	91
name-value-list	8
json	5
cee-syslog	1
mac48	55
string-to	12
char-to	825
char-sep	3

Figure 15: PDAG dynamic data (ent55m Data Set)

(a) Backtracking Count

Backtracking count	messages
0	53411698
1	458055
2	6464
3	60404
4	262517
5	634679
6	16289
9	112327
12	10204
43	84500
avg	0.176912
max	43
msgs total	55057137
msgs with backtr.	1645439
% backtracking	2.99
avg backtr. if needed	5.91

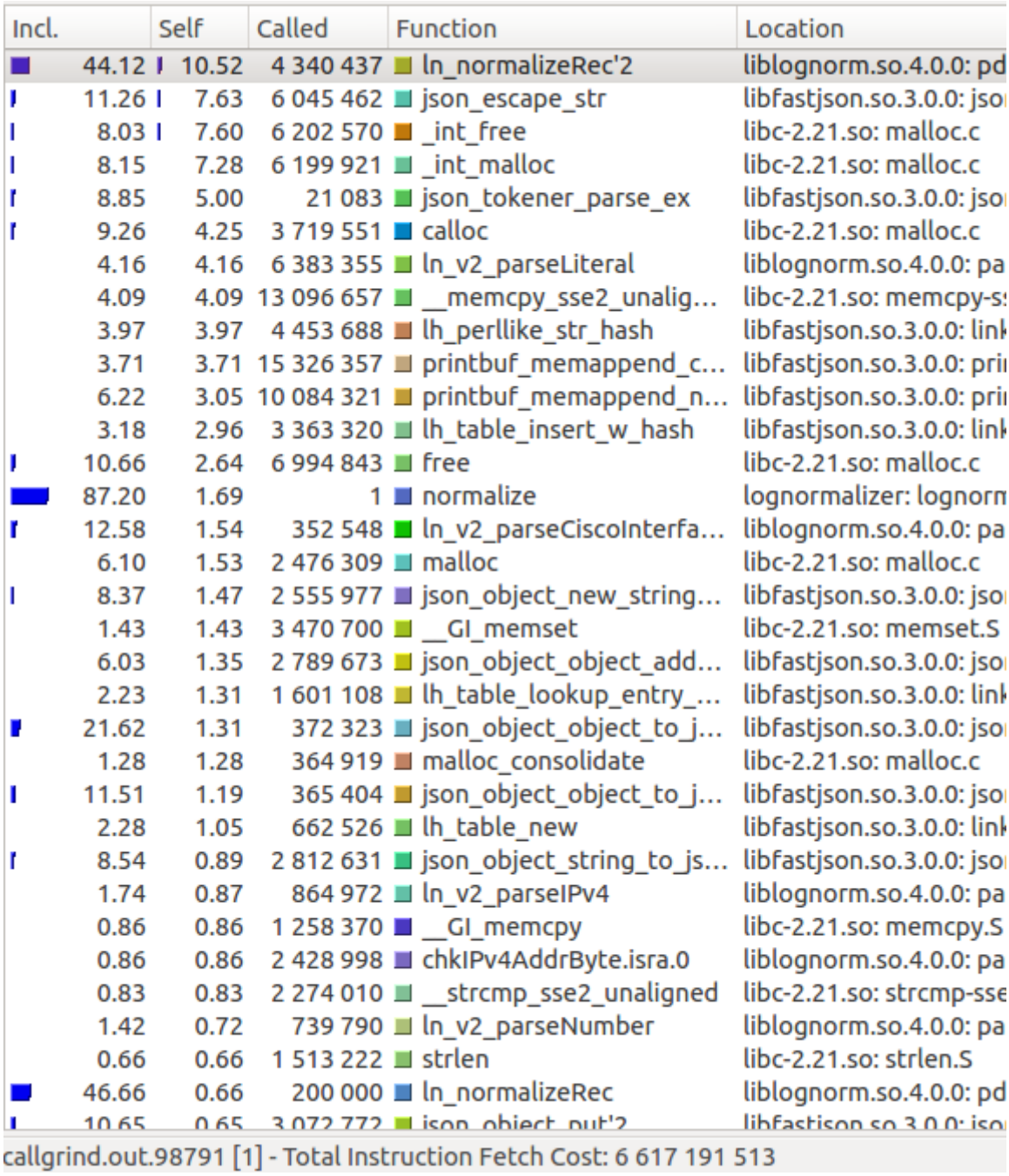
(b) Path Lengths

length	messages
avg	17.711578
max	101
7	1241559
8	716
9	19964706
10	56708
11	100139
12	313725
13	18313
14	175689
15	135809
16	161927
17	586429
18	134392
19	6417985
20	82321
21	6904664
22	2108
23	7167369
24	7031
25	1642200
26	718
27	8025324
28	975213
29	728

(c) Path Lengths, cntd

length	messages
30	13428
31	59847
32	8257
33	29
34	2291
35	17491
36	22792
37	9392
38	3964
39	180
40	11161
41	17060
42	149
44	628587
45	2985
46	2088
52	84514
53	14
54	8052
56	1
72	34
85	12510
91	14817
95	21720

Figure 16: Normalizer Execution Profile (kcachegrind screenshot)



Incl.	Self	Called	Function	Location
44.12	10.52	4 340 437	ln_normalizeRec'2	liblognorm.so.4.0.0: pd
11.26	7.63	6 045 462	json_escape_str	libfastjson.so.3.0.0: jso
8.03	7.60	6 202 570	_int_free	libc-2.21.so: malloc.c
8.15	7.28	6 199 921	_int_malloc	libc-2.21.so: malloc.c
8.85	5.00	21 083	json_tokener_parse_ex	libfastjson.so.3.0.0: jso
9.26	4.25	3 719 551	calloc	libc-2.21.so: malloc.c
4.16	4.16	6 383 355	ln_v2_parseLiteral	liblognorm.so.4.0.0: pa
4.09	4.09	13 096 657	__memcpy_sse2_unalig...	libc-2.21.so: memcpy-s
3.97	3.97	4 453 688	lh_perllike_str_hash	libfastjson.so.3.0.0: link
3.71	3.71	15 326 357	printbuf_memappend_c...	libfastjson.so.3.0.0: pri
6.22	3.05	10 084 321	printbuf_memappend_n...	libfastjson.so.3.0.0: pri
3.18	2.96	3 363 320	lh_table_insert_w_hash	libfastjson.so.3.0.0: link
10.66	2.64	6 994 843	free	libc-2.21.so: malloc.c
87.20	1.69	1	normalize	lognormalizer: lognorm
12.58	1.54	352 548	ln_v2_parseCiscoInterfa...	liblognorm.so.4.0.0: pa
6.10	1.53	2 476 309	malloc	libc-2.21.so: malloc.c
8.37	1.47	2 555 977	json_object_new_string...	libfastjson.so.3.0.0: jso
1.43	1.43	3 470 700	__GI_memset	libc-2.21.so: memset.S
6.03	1.35	2 789 673	json_object_object_add...	libfastjson.so.3.0.0: jso
2.23	1.31	1 601 108	lh_table_lookup_entry_...	libfastjson.so.3.0.0: link
21.62	1.31	372 323	json_object_object_to_j...	libfastjson.so.3.0.0: jso
1.28	1.28	364 919	malloc_consolidate	libc-2.21.so: malloc.c
11.51	1.19	365 404	json_object_object_to_j...	libfastjson.so.3.0.0: jso
2.28	1.05	662 526	lh_table_new	libfastjson.so.3.0.0: link
8.54	0.89	2 812 631	json_object_string_to_js...	libfastjson.so.3.0.0: jso
1.74	0.87	864 972	ln_v2_parseIPv4	liblognorm.so.4.0.0: pa
0.86	0.86	1 258 370	__GI_memcpy	libc-2.21.so: memcpy.S
0.86	0.86	2 428 998	chkIPv4AddrByte.isra.0	liblognorm.so.4.0.0: pa
0.83	0.83	2 274 010	__strcmp_sse2_unaligned	libc-2.21.so: strcmp-sse
1.42	0.72	739 790	ln_v2_parseNumber	liblognorm.so.4.0.0: pa
0.66	0.66	1 513 222	strlen	libc-2.21.so: strlen.S
46.66	0.66	200 000	ln_normalizeRec	liblognorm.so.4.0.0: pd
10.65	0.65	3 072 772	json_object_put'2	libfastjson.so.3.0.0: jso

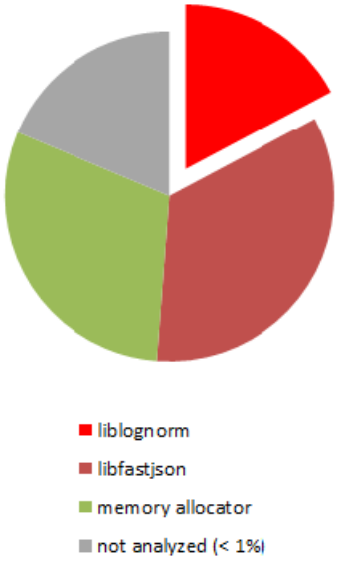
callgrind.out.98791 [1] - Total Instruction Fetch Cost: 6 617 191 513

Figure 17: Profile analyzer result of execution time distribution between liblognorm and runtime library

(a) Frequently Executed Functions (> 1%)

Function	Exec Time (%)
ln_normalizeRec	10.52
ln_v2_parseLiteral	4.16
normalize	1.00
ln_v2_parseCiscoInterfa..	1.54
liblognorm	**17.22**
json_escape_str	7.63
json_tokener_parse_ex	5.00
lh_perllike_str_hash	3.97
printbuf_memappend_c..	3.71
printbuf_memappend_n..	3.05
lh_table_insert_w_hash	2.96
json_object_new_string..	1.47
json_object_object_add	1.35
lh_table_lookup_entry	1.31
json_object_object_to_j..	1.31
json_object_object_to_j..	1.19
lh_table_new	1.05
libfastjson	**34.00**
_int_free	7.60
_int_malloc	7.28
calloc	4.25
_memcpy_sse2_unalig..	4.09
free	2.64
malloc	1.53
__GI_memset	1.43
malloc_consolidate	1.28
memory allocator	**30.10**
total analyzed	**81.32**

(b) Time Distribution



B Supported Motifs and their Time Complexity

Table 6 on page 83 lists motifs available in liblognorm v2 together with their computational complexity. The complexity is shown in relation to a log message l and determined as follows:

- *simple* motif parsers read at most as many characters from l as there can be inside the motif. No looping is done. As such, all simple parsers are in $O(|l|)$.
- *very specific simple* motif parsers, like the IP address or date parsers only read a fixed maximum amount of characters from l. So they are $O(1)$.
- The *cisco-interface-spec* motif parser is a combination of multiple simple, very specific parsers. Each of these parsers is $O(1)$, but cisco-interface-spec needs to do some limited look-ahead, so it is classified as $O(|l|)$.
- *structured* motif parsers also read l only once, so they are $O(|l|)$.
- The *CEF* structured parser is an exception: at worst case, it needs to read the input twice, so it is $O(2|l|)$, which also is in $O(|l|)$. The reason for the need to read twice is that CEF is defined in a very complex way: there is no clear indication of the end of a parameter value. To detect its end, the next parameter name needs to be read, which can theoretically result in most parts of the message being read twice.
- The *alternative* "motif" is actually a structural element implemented as a motif. It permits to branch between different sub-PDAGs. As such, it is subject to backtracking and so its computational complexity is O(PDAG) what denotes the complexity parsing the PDAG as whole has.
- The *repeat* "motif" is also a structural element implemented as motif. It permits to loop over the message and extract repeated elements. Those sometimes occur in log messages, for example to show multiple users doing the same operation or showing connection flags associated with a traffic flow in firewall traffic. With repeat, one can specify motifs that need to be repeatedly matched. As such, the complexity of repeat is that of those parsers. For many practical cases, this means "repeat" is $O(|l|)$. However, the "alternative" motif may be used, and so in general "repeat" is O(PDAG).
- As *user-defined* motifs are disconnected PDAG components, and as such are PDAGs themselves, they also have O(PDAG) complexity.

The O(PDAG) complexity for the PDAG as whole is discussed in Sect. 6.4 on page 42.

Note that some motif parsers mainly exist for compatibility reasons with liblognorm v1. This is a practical requirement to provide compatibility to existing rule bases. In v2, many of them can be replaced with base-type parsers constrained in their parameter range or by using user-defined motif parsers.

Table 6: Motif Parsers in liblognorm v2

motif	type	complexity
date-iso	simple	$O(1)$
date-rfc3164	simple	$O(1)$
date-rfc5424	simple	$O(1)$
ipv4	simple	$O(1)$
ipv6	simple	$O(1)$
mac48	simple	$O(1)$
time-12hr	simple	$O(1)$
time-24hr	simple	$O(1)$
kernel-timestamp	simple	$O(1)$
alpha	simple	$O(\|l\|)$
char-sep	simple	$O(\|l\|)$
char-to	simple	$O(\|l\|)$
duration	simple	$O(\|l\|)$
float	simple	$O(\|l\|)$
hexnumber	simple	$O(\|l\|)$
literal	simple	$O(\|l\|)$
number	simple	$O(\|l\|)$
op-quoted-string	simple	$O(\|l\|)$
quoted-string	simple	$O(\|l\|)$
rest	simple	$O(\|l\|)$
string	simple	$O(\|l\|)$
string-to	simple	$O(\|l\|)$
whitespace	simple	$O(\|l\|)$
word	simple	$O(\|l\|)$
cisco-interface-spec	simple (combined)	$O(\|l\|)$
json	structured	$O(\|l\|)$
v2-iptables	structured	$O(\|l\|)$
checkpoint-lea	structured	$O(\|l\|)$
cee-syslog	structured	$O(\|l\|)$
name-value-list	structured	$O(\|l\|)$
cef	structured	$O(\|l\|) = O(2\|l\|)$
alternative	branching construct	O(PDAG)
repeat	loop construct	O(PDAG)
user-defined motif	sub-PDAG	O(PDAG)

Literatur

[1] NETRESEC AB. Publicly available pcap files. `http://www.netresec.com/?page=PcapFiles`. Online, last access November, 23rd 2015.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers principles, techniques, and tools (second edition).* Addison-Wesley, Reading, MA, 2007.

[3] Alfred V. Aho. Algorithms for finding patterns in strings. *Algorithms and Complexity*, 1:255, 2014.

[4] The Internet Archive. Internet archive wayback machine. `https://archive.org/web/`. Online, last access June, 29th 2015.

[5] T. Bird and M Ranum. Loganalysis.org. Online via the Internet Archive, last access June, 29th 2015; original site no longer available.

[6] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159 (Proposed Standard), March 2014.

[7] Elasticsearch BV. grok. `https://www.elastic.co/guide/en/logstash/current/plugins-filters-grok.html`. Online, last access June, 1st 2015.

[8] Elasticsearch BV. Logstash. `https://www.elastic.co/products/logstash`. Online, last access November, 20st 2015.

[9] V.G. Cerf. ASCII format for network interchange. RFC 20 (INTERNET STANDARD), October 1969.

[10] Inc. Cisco Systems. *Catalyst 2960 and 2960-S Switch Software Configuration Guide, Cisco IOS Release 12.2(55)SE.* Cisco Systems, Inc., 170 West Tasman Drive, San Jose, CA 95134-1706, USA, 2010. also online at `http://www.cisco.com/c/en/us/td/docs/switches/lan/catalyst2960/software/release/12-2_55_se/configuration/guide/scg_2960.pdf`, last access November, 23nd 2015.

[11] The FreeBSD Developer Community et al. Freebsd src tree. `https://github.com/freebsd/freebsd`. Online, last access June, 1st 2015.

[12] Microsoft Corporation. Windows event log service. https://technet.microsoft.com/en-us/library/dd315601(v=ws.10).aspx. Online, last access March, 15th 2016.

[13] MITRE Corporation and CEE Editorial Board. Common event expression. http://cee.mitre.org/. Online, last access June, 1st 2015.

[14] Russ Cox. Regular expression matching can be simple and fast. https://swtch.com/~rsc/regexp/regexp1.html. Online, last access July, 2nd 2015.

[15] DMTF. Distributed management task force inc. https://www.dmtf.org/. Online, last access March, 15th 2016.

[16] Jordan Sissel et al. grok on github. https://github.com/jordansissel/grok. Online, last access July, 2nd 2015.

[17] Philip Hazel et al. Pcre - perl compatible regular expressions. http://www.pcre.org. Online, last access July, 2nd 2015.

[18] J. Evans et al. jemalloc, an alternative memory allocator. http://www.canonware.com/jemalloc/. Online, last access December, 8th 2015.

[19] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[20] R. Gerhards. liblognorm. http://www.liblognorm.com/. Online, last access June, 1st 2015.

[21] R. Gerhards. On the nature of syslog data. http://www.monitorware.com/en/workinprogress/nature-of-syslog-data.php. Online, last access June, 29th 2015.

[22] R. Gerhards. Syslog log samples. http://www.monitorware.com/en/logsamples/. Online, last access November, 23rd 2015.

[23] R. Gerhards. The Syslog Protocol. RFC 5424 (Proposed Standard), March 2009.

[24] R. Gerhards et al. a repository with sample log files for research purposes. http://git.adiscon.com/?p=log-samples-for-research.git;a=summary. Online, last access November, 24st 2015.

[25] R. Gerhards et al. rsyslog, a modern syslogd. http://www.rsyslog.com. Online, last access June, 9th 2015.

[26] R. Gerhards et al. sample pix log messages. http://git.adiscon.com/?p=log-samples-for-research.git;a=tree;f=freetext/cisco;h=879bff34d58532220db7025dd1df2bff84254a60;hb=HEAD. Online, last access November, 24st 2015.

[27] Adiscon GmbH. Eventreporter - windows event monitoring & forwarding. http://www.eventreporter.com/en/. Online, last access November, 24th 2015.

[28] Graylog. Gelf — graylog extended log format. https://www.graylog.org/resources/gelf/. Online, last access November, 24th 2015.

[29] E. Haszlakiewicz et al. Json-c, a json implementation in c. https://github.com/json-c/json-c/wiki. Online, last access December, 8th 2015.

[30] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[31] The IEEE and The Open Group. closelog, openlog, setlogmask, syslog - control system log. http://pubs.opengroup.org/onlinepubs/9699919799/functions/closelog.html. Online, last access June, 11st 2015.

[32] ArcSight Inc. Common event format, revision 16. https://kc.mcafee.com/resources/sites/MCAFEE/content/live/CORP_KNOWLEDGEBASE/78000/KB78712/en_US/CEF_White_Paper_20100722.pdf. Online, last access July, 2nd 2015.

[33] Trustwave Holdings Inc. Info: What is the welf log file format? https://www3.trustwave.com/support/kb/article.aspx?id=10899. Online, last access March, 15th 2016.

[34] Andrew Josey, DW Cragun, N Stoughton, M Brown, C Hughes, et al. The open group base specifications issue 6–ieee std 1003.1. *The IEEE and The Open Group*, 20(6), 2004.

[35] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[36] Viktor Leis, Alfons Kemper, and Tobias Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 38–49. IEEE, 2013.

[37] Val Menn. New tools for event management in windows vista. https://technet.microsoft.com/en-us/magazine/2006.11.eventmanagement.aspx. Online, last access March, 15th 2016.

[38] R. Merkl and S. Waack. *Bioinformatik Interaktiv (2. Auflage)*. WILEY-VCH Verlag GmbH & Co. KGaA, Weinheim, 2009.

[39] Donald R Morrison. Patricia - practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM (JACM)*, 15(4):514–534, 1968.

[40] Chris Moultrie. Logstash grok speeds. http://ghost.frodux.in/logstash-grok-speeds/. Online, last access July, 2nd 2015.

[41] James D Murray. *Windows NT Event Logging*. O'Reilly & Associates, Inc., 1998.

[42] InterSect Alliance Pty. Snare enterprise agent for windows. https://www.intersectalliance.com/our-product/snare-agent/operating-system-agents/snare-agent-for-windows/. Online, last access November, 24th 2015.

[43] E. Haszlakiewicz R. Gerhards et al. libfastjson, a fast json library for c. https://github.com/rsyslog/libfastjson. Online, last access December, 8th 2015.

[44] G. Salgueiro, V. Pascual, A. Roman, and S. Garcia. Indicating WebSocket Protocol as a Transport in the Session Initiation Protocol (SIP) Common Log Format (CLF). RFC 7355 (Informational), September 2014.

[45] Balazs Scheidler. Collecting log samples. https://bazsi.blogs.balabit.com/2010/11/collecting-log-samples/. Online, last access November, 23rd 2015.

[46] Balabit IT Security. Pattern db. https://www.balabit.com/network-security/syslog-ng/opensource-logging-system/features/pattern-db. Online, last access June, 1st 2015.

[47] Balabit IT Security. syslog-ng on github. https://github.com/balabit/syslog-ng. Online, last access June, 26th 2015.

[48] Jordan Sissel. libgrok. https://github.com/jordansissel/grok. Online, last access November, 20st 2015.

[49] L. Torvalds et al. Linux kernel source tree. `https://github.com/torvalds/linux`. Online, last access June, 1st 2015.

[50] Tod Troxell et al. logcheck. `http://logcheck.org/`. Online, last access June, 1st 2015.

[51] CORPORATE Unicode Staff. *The Unicode Standard: Worldwide Character Encoding*. Addison-Wesley Longman Publishing Co., Inc., 1991.

[52] Risto Vaarandi. A data clustering algorithm for mining patterns from event logs. In *Proceedings of the 2003 IEEE Workshop on IP Operations and Management (IPOM)*, pages 119–126, 2003.

[53] Risto Vaarandi. *Tools and Techniques for Event Log Analysis, PhD Thesis*. Tallinn University of Technology Press, 2005.

[54] The valgrind developers. Valgrind, a system for debugging and profiling linux program. `http://valgrind.org`. Online, last access December, 8th 2015.

[55] w3c consortium. Extended log file format, w3c working draft wd-logfile-960323 (work in progress). `http://www.w3.org/TR/WD-logfile.html`. Online, last access November, 25th 2015.

[56] J. Weidendorfer. kcachegrind call graph viewer. `http://kcachegrind.sourceforge.net/html/Home.html`. Online, last access December, 15th 2015.

[57] B. Wilkinson and M. Allen. *Parallel programming: techniques and applications using networked workstations and parallel computers*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 2004.

Ich versichere, dass ich diese Master-Abschlussarbeit selbstständig verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt habe. Die Stellen, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, sind in jedem einzelnen Fall unter Angabe der Quelle als Entlehnung (Zitat) kenntlich gemacht. Das gleiche gilt für beigefügte Skizzen und Darstellungen. Außerdem räume ich dem Lehrgebiet das Recht ein, die Arbeit für eigene Lehr- und Forschungstätigkeiten auszuwerten und unter Angabe des Autors geeignet zu publizieren.

Hagen, den 2016-04-27

Rainer Gerhards