NON-INTRUSIVE, AUTOMATED LOG DISCOVERY AND PARSING

by

Kirk Adrian Max Rodrigues

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science
The Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto

© Copyright 2018 by Kirk Adrian Max Rodrigues
Abstract

Non-intrusive, Automated Log Discovery and Parsing

Kirk Adrian Max Rodrigues
Master of Applied Science

The Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto

2018

Today’s distributed systems generate log data for tasks such as debugging failures. When aggregated into a log analytics tool, this data can be queried to perform anomaly detection and failure diagnosis at a scale that is infeasible otherwise. However, before analysis, this data must be discovered and parsed.

harvester is a tool capable of automatically finding and parsing logs for indexing. harvester’s approach is novel in that it generates regular expressions for parsing using the constants of the executables that authored the messages. This allows harvester to parse logs for any software, without source code, so long as constants can be extracted from the software’s binary.

In our evaluation of two distributed systems written in Java and C, we are able to achieve an accuracy of 67.7% for the generated expressions. Our inaccuracies are remediable with slightly improved logging practices without having to overhaul existing logging.
Acknowledgements

Several individuals deserve praise and gratitude for their help with this thesis and my own personal development. Without these people, I would not be where or who I am today.

First and foremost, I am grateful to my supervisor, Ding Yuan, for recognizing potential in me and fostering my growth, patiently. I would not have achieved what I have achieved today, without him.

I am also grateful to Xu Zhao, Jack Luo, David Lion, and Michael Stumm whose work around this project was instrumental to its success. Thanks should also given to the rest of Ding’s group, Yongle Zhang, Serguei Makarov, Xiang Ren, and Adrian Chiu for their valuable feedback.

I would also like to thank the thesis committee members, Ashvin Goel, Michael Stumm, and Mireille Broucke for their thoughtful criticism and participation in the thesis’ defense.

Finally, I would like to thank those individuals who with or without knowing it changed my life.
# Table of Contents

Acknowledgements iii

Table of Contents iv

List of Tables vi

List of Figures vii

1 Introduction 1

2 Background 6
   2.1 Software Logging ......................................................... 6
   2.2 Linux’s /proc Pseudo-filesystem ....................................... 6
   2.3 Grok and Regular Expressions ......................................... 7
   2.4 Log Ingestion Engines .................................................. 8
   2.5 Log Aggregation and Analytics Tools ................................. 9

3 Motivating Example 10

4 Design 14
   4.1 Overview ................................................................. 14
   4.2 Log Discovery .......................................................... 15
   4.3 Log Parsing ............................................................. 16
   4.4 Identifier Inference .................................................... 17

5 Implementation 18
   5.1 Discovering Log and Executable Files ............................... 18
   5.2 Extracting Constants from Executables ............................... 19
5.3 Extracting Messages from Log Files ................................. 20
5.4 Heuristics .......................................................... 20
  5.4.1 Filtering Constants ........................................... 21
  5.4.2 Identifying Variable Boundaries ............................. 21
  5.4.3 Determining a Variable’s Semantic and Syntax ............. 21
  5.4.4 Identifier Identification ...................................... 23

6 Evaluation ............................................................. 24
  6.1 Workloads .......................................................... 24
  6.2 Configuration ....................................................... 24
  6.3 Accuracy Results .................................................. 25
  6.4 Logging Guidelines ................................................ 27

7 Limitations and Future Work ........................................... 28

8 Related Work .......................................................... 30
  8.1 Tools Based on Static-analysis .................................. 30
  8.2 Tools Based on Pattern-matching ............................... 31

9 Conclusion ............................................................. 32

Bibliography ............................................................. 33
List of Tables

5.1 Heuristics used to recognize file type .............................................. 18

6.1 Specifications of the evaluation cluster ............................................. 25

6.2 Inaccuracy classes and their occurrences .......................................... 26
List of Figures

1.1 An example showing Stitch’s ability to visualize a system’s hierarchy using log messages. 2

2.1 Example content for the /proc/<PID>/maps file 7
2.2 A log message with the Grok and regular expressions used to parse it 8

3.1 A log message generated using concatenation and the code which generates it 11
3.2 Two messages showing an attempt ID which is also part of a related client ID 11
3.3 A datanode UUID with its label specified in two different formats 11
3.4 Configuration necessary to discover and parse Hadoop logs using user-generated expressions 13

4.1 An example log message and the grok expressions used to parse it 15

6.1 Two log messages showing variable values logged and labelled as data structures rather than individually 26
6.2 A log message that logs variable values in a sentence 27
Chapter 1

Introduction

Today’s popular server and distributed software typically generates useful logging data. For example, these log messages may contain information about when requests begin and end, what components they traverse, performance metrics, software exceptions, and any other information useful for understanding how the system behaves. However, as a system scales in the number of machines, software components, and users it supports, this log data quickly becomes hard to inspect and manage manually. For example, in 2010, Facebook was already generating 30TB of log data [22] per day, necessitating a specialized solution to aggregate and analyze this data.

We designed such a tool, Stitch [27], to visualize this data so that it could be used to reconstruct a request’s execution flow across multiple machines and software layers potentially written in different languages. For example, consider the contrived log files in Figure 1.1 – the first contains log messages showing a user submitting two jobs while the second has messages showing when each job starts and ends. Figure 1.1b shows Stitch’s visualization of these files. Each value on the left identifies an object with its hierarchical relationship to other objects in the log files. The circles on the right represent log messages concerning the object in the same row, displayed along a timeline. In this example, user1 is intuitively shown as a parent of job1 and job2 with the timeline indicating the duration of each job. As we demonstrated in the Stitch project, this type of visualization can aid debugging better than analyzing the log messages alone. However, Stitch fundamentally depends on discovering and parsing each log file before it can analyze them.

For Stitch, discovering and parsing log files involves finding all log files on an application machine, extracting each log message, and then parsing it into two components: the message’s timestamp and any object identifiers labelled with their semantics. An object identifier is a variable value which identifies an...
object and its semantic is how it is labelled by a developer. For example, assume MongoDB [18] logged this message “2016-04-02T00:58:48.734 MongoDB starting : pid=22925 serverIp=172.16.196.133 serverPort=27017 numThreads=8” to /var/log/mongodb/mongod.log. Stitch would need to automatically discover this file and break the message up into these fields { timestamp: 1459573128, pid: “22925", serverIp: "172.16.196.133", serverPort: "27017" }. Apart from the timestamp, pid identifies the MongoDB process while serverIp and serverPort identify the port on which MongoDB is listening for new connections. Note that each identifier’s type (e.g., an IP address) is subtly different from its semantic (e.g., a server IP address) and non-identifier values like numThreads are omitted. In practice, these non-identifiers can still be used in other log-analytics tools to graph metrics, monitor performance, and so on.

To be effective, industry-standard search and analytics tools like ElasticSearch [4], Splunk [20], Loggly [16], and Graylog [6] also need to discover and parse log files, but currently do so with some notable limitations. First, the user must manually specify what files and folders to monitor for log messages. This process can be tedious and error prone as the number of software components in the system grows. Some tools attempt to ease this effort by monitoring common log locations like /var/log/ on Linux machines, but this is ineffective when today’s distributed applications may log to different locations by default. For instance, Hadoop [8] and Spark [25] log to their installation directories whereas Redis [19] logs to stdout by default.

Once discovered, each file is parsed using manually-written regular expressions. For example, the MongoDB message from the previous example would be parsed by “%{TIMESTAMP:timestamp} MongoDB starting : pid=%{PID:pid} serverIp=%{IP:serverIp} serverPort=%{PORT:serverPort} numThreads= %{NUMBER:numThreads}”. This type of regular expression is called a Grok expression [7] and each
A %SYNTAX:semantic% pair captures a field labelled “semantic” which is parsed by the subexpression denoted by SYNTAX. Each log-analytics tool typically ships with several of these expressions for common software like Apache’s httpd server, but these expressions may not be exhaustive for each application and several applications like Spark have no prewritten expressions. Writing these manually can be infeasible due to the large number of logging statements in these applications. For example, Hadoop 2.7 contained over 5,800 logging statements [26] meaning 5,800 expressions would need to be written. Moreover, some of these statements are control/data-flow dependent requiring even more expressions depending on which path is taken.

In practice, customers deal with this problem in one of two ways. They either rewrite all their logging statements to be easier to write expressions for [23] or they eliminate parsing altogether by logging key-value pairs directly (e.g., by logging JSON objects) [12]. Either approach requires an overhaul of all existing logging that could easily take up to a year of effort.

While solving other problems, prior work has created automated expression generators, but these have limitations that make them impractical for our requirements. First, no approach has attempted to capture variables’ semantics. Sherlog [24] and lprof [28] can determine a variable’s type, but as noted before, this is different than a variable’s semantic. Sherlog and lprof use static analysis to construct a regular expression that can match the message generated by a particular logging statement. However, because a log message can be generated in many different ways (e.g., format strings, concatenation, etc.), this analysis can be difficult to write, and may need to be specially crafted for different languages. LogMine [9] and Spell [3] use the differences between similar messages to identify variable values in a message. They can then replace these values with capturing groups to generate a regular expression. However, these approaches require some amount of training to be accurate and even then, they are still probabilistic since they depend on the input messages being different enough to identify variable values accurately.

In contrast, harvester is a novel tool that can automatically discover log files, parse messages to extract variable values, determine their semantics and finally generate Grok expressions. harvester discovers log files by monitoring the system for open log files. To parse messages, our key idea is to extract the constants from the executable files which authored a log message, and then use these constants to filter out any variable values in the message. Since the variable values are, by definition, variable, they should be random enough such that they are not matched by constants. In practice, a variable’s value may contain constants itself which contributes to the syntax we generate for it. To determine its semantic, we use a set of heuristics based on the constant strings surrounding the value. Finally, since we designed harvester to support Stitch, we needed to determine which variable values were object
identifiers and which were not. Here, we again use heuristics to determine whether a variable’s semantic is likely an identifier or non-identifier.

We evaluated *harvester* on two distributed systems and found *harvester* achieved an overall accuracy of 67.7%, while successfully ingesting logs faster than they can be generated, on average. Our evaluation consisted of running benchmark workloads on Hadoop (written in Java) and Redis (written in C). These pieces of software are written by different teams and in different languages, suggesting that our method is general enough to be effective. The notable inaccuracies we observed are mostly due to variables whose values are constants (e.g., state values), variables logged within a data structure, and variables without semantics. Nonetheless, these are easily remedied by marginally better logging practices.

Our evaluation and the heuristics we use also imply simple guidelines for better logging. For example, developers should avoid implicitly stating the semantic of a value in a sentence. This is clear in “...initialized with 0 entries 0 lookups” where the first variable’s semantic should be similar to `numEntries`, but that requires understanding the mechanics of the English language. Similarly, developers should try to ensure that each variable’s semantic is globally unique so that it does not conflict with other variables’ semantics when aggregated or analyzed. Testing our suggested guidelines, we were able to increase accuracy to 90%.

Thus, this work makes the following contributions:

- A novel method of generating regular expressions for parsing log messages that requires only the executable files that authored each message.
- An effective way to discover log files and the executables writing to them.
- Several guidelines to improve plain-text logging while retaining its flexible nature.

However, *harvester* has a few limitations:

- Since *harvester* discovers log and executable files by monitoring open files, it is more suitable for long-running programs which stay open long enough for *harvester* to detect them. For example, *harvester* has trouble parsing the logs of periodic cron jobs since it cannot capture the executable’s path before the job finishes.
- Because *harvester* monitors open files, *harvester* may need root privileges to access all logs. However, this could be mitigated if the processes of interest run under one user or group of users for *harvester* to share.
- *harvester* cannot parse variable values which are constant strings (e.g., state variables).
This thesis is organized as follows: Chapter 2 describes the necessary background information to understand this thesis. Chapter 3 details a case where harvester can be very helpful. Chapter 4 then describes the high level algorithm design of harvester. Next, Chapter 5 explains how the design was implemented and made practical with several heuristics. Chapter 6 evaluates the accuracy of tool after which Chapter 7 describes notable limitations and how we hope to address them. Finally, Chapter 8 describes prior related work before we conclude in Chapter 9.
Chapter 2

Background

2.1 Software Logging

Software logging is a form of tracing where information about a system is captured in plain-text messages which are then typically stored in files. In most cases, these are written by developers to aid in debugging the software and monitor its execution without incurring significant performance overhead. They often consist of constant strings and variables values concatenated together in a way that is meaningful to the developer. Each message is also usually given a severity/verbosity level that indicates its importance according to the developer. In addition, when messages are used to trace and correlate events, they typically include timestamps. Finally, since logging is an on-going process, log files are typically segmented automatically to ensure they do not get too large to manage. Overall, it should be clear that software logging is a characteristically manual and somewhat ad-hoc effort at present.

2.2 Linux’s /proc Pseudo-filesystem

Linux’s /proc pseudo-filesystem contains a myriad of information about running processes, including what files they have open or memory mapped, each process’ view of the file system, each process’ current working directory, and the command line string used to start each process. All of this information is stored within a per-process directory, /proc/<PID>/, where <PID> is the process’ ID. Collectively, this information allows harvester to determine which logs files are being written to by each process, and which executable files are potentially being used to write each log.

All open file descriptors are stored as symbolic links under the /proc/<PID>/fd directory. Each link either contains a path to a file on the file system, information about an open pipe/socket, or
Figure 2.1: Example content for the \texttt{/proc/<PID>/maps} file

information about system file descriptors like those corresponding to timers. In each case, the \texttt{lstat} call or the \texttt{/proc/<PID>/fdinfo} file can be used to determine the file descriptor’s open mode (e.g., open for writing). For links concerning pipes and sockets, the content of the link is \texttt{<type>:[<inode#>]} where \texttt{<type>} is \texttt{pipe} or \texttt{socket}, and \texttt{<inode#>} is the inode number corresponding to the open pipe or socket. For the last category of links, the link contains \texttt{anon_inode:[<type>]} where \texttt{<type>} indicates the type of descriptor. For example, the link contains \texttt{anon_inode:[eventpoll]} for epoll file descriptors.

A list of memory mapped regions is stored in the \texttt{/proc/<PID>/maps} file. Figure 2.1 shows some example content for this file. The last field is either the path of the file backing the region, a hint at the region (e.g., heap), or nothing if the region is not backed. Note that the process’ view of the file system may be different to harvester’s view since other file systems or folders may have been mounted by the process or the process’ root directory (/) may have been changed using the \texttt{chroot} command. Therefore, any paths in the process’ \texttt{maps} file must be accessed relative to \texttt{/proc/<PID>/root}.

The command line string used to start a process is stored in the \texttt{/proc/<PID>/cmdline} file. This file contains a list of null-terminated strings corresponding to the command and arguments used to start the process. The list ends with a final null character. If the command is relative to the process’ current working directory, the \texttt{/proc/<PID>/cwd} symbolic link indicates the path of this directory.

2.3 Grok and Regular Expressions

Grok expressions are an extension of typical regular expressions. Accordingly, they use the same syntax with the exception that Grok allows a user to specify subexpressions. Consider the message in Figure 2.2a parsed by the Grok expressions in Figure 2.2b. The last line in Figure 2.2b is an expression containing subexpressions \texttt{%{USER:user}}, \texttt{%{HOST:host}}, and \texttt{%{DURATION}}. Each subexpression has the format \texttt{%{SYNTAX:semantic}} where \texttt{SYNTAX} refers to the Grok expression used and \texttt{semantic} to the value’s semantic. Therefore, \texttt{%{USER:user}} refers to the Grok expression on line 1 captured with the semantic \texttt{user}. Similarly, \texttt{%{HOST:host}} is on line 2 captured with the semantic \texttt{host}. Finally, \texttt{%{DURATION}} is on line 3 and is not captured.

Figure 2.2c shows a typical regular expression used to parse the same message. The syntax (\texttt{(?<user>
user1 logged out of the machine host1 in 5s

(a) An example log message

USER user[0-9]+  
HOST host[0-9]+  
DURATION [0-9]+s  
MESSAGE %{USER:user} logged out of the machine %{HOST:host} in %{DURATION}

(b) Grok expression to parse the message

(?<message> (?<user>user[0-9]+) logged out of the machine (?<host>host[0-9]+) in [0-9]+s)

(c) Regular expression to parse the message

Figure 2.2: A log message with the Grok and regular expressions used to parse it

user[0-9]+) is a named capturing group with name “user” and regular expression “user[0-9]+”. Notice that this syntax would duplicate “user[0-9]+” across all regular expressions which contained a user variable. This can become unwieldy as the number of expressions increases.

In practice, Grok expressions are the popular standard for specifying expressions to parse log messages.

2.4 Log Ingestion Engines

Log ingestion engines like LogStash [17] and fluentd [5] typically perform three functions: monitoring and reading log messages from different data sources, applying filters, and outputting the filtered data to aggregation and analytics tools. A variety of different data sources are supported including log files, pipes, and databases. Similarly, a variety of data sinks are supported including log analytics tools, files, and database. Log files are typically monitored using the monitoring layer of the system’s file system (e.g., inotify on Linux) or periodically checking the file for new log messages. When messages are found, they are parsed according to the enabled filters. A filter could be a Grok parser or a mutator to modify the message. Finally, the message is sent to the configured data sinks like ElasticSearch.

In recent years, LogStash has aimed to split off its monitoring functionality to avoid the performance penalty of parsing messages on user application nodes. This monitoring has been taken over by the lightweight Beats [1] component. For example, FileBeat can be used to monitor log files and send their content to a centralized LogStash.
2.5 Log Aggregation and Analytics Tools

Log aggregation and analytics tools like ElasticSearch [4] and Splunk [20] function as data sinks that can index a user’s data for search as well as perform various analytics. For example, ElasticSearch can ingest parsed log data as JSON objects output by LogStash, index each field in the object, and then allow users to search the data as full-text or individual fields. This data can also be graphed and monitored by web applications like Kibana [11].

Stitch is a performance analytics and debugging tool that uses the cardinality relationship between variables to infer their hierarchical relationship. For example, if a variable with semantic $A$ is logged together with multiple variables with semantic $B$, and not vice-versa, Stitch would infer variables with semantic $A$ had a one-to-many relationship with variables with semantic $B$. This in turn would mean variables with semantic $A$ were parents of variables with semantic $B$. A concrete example of this is a user starting multiple MapReduce jobs. A user would be associated with multiple jobs but a job would not typically be associated with multiple users. Stitch visualizes these relationships for each log message across multiple software stacks, giving users a structural view of their system. Thus, the parser that feeds Stitch must be able to extract variable values with their semantics from log messages across multiple software stacks.
Chapter 3

Motivating Example

To help users get started, LogStash ships with several manually generated Grok expressions for popular pieces of software like Amazon Web Services (AWS), MongoDB, and Nagios. However, for some software like Nagios, not all types of log messages are covered by the expressions and for software like Hadoop and Spark, no expressions are provided. Therefore, suppose we are first tasked with generating Grok expressions for Hadoop (sufficient for use with Stitch). Then second, we must configure FileBeat and LogStash to use these expressions to parse Hadoop’s log files.

Since Hadoop is open source, we might first try to generate patterns directly from the source code. However, there are over 5,800 logging statements in Hadoop at the default INFO level verbosity or higher [26]. Furthermore, they are difficult to parse since parts of the log message may be generated through concatenation statements spread across the code base. Figure 3.1a shows such a log message and Figure 3.1b shows the logging statement which generates it. aId is a TaskAttemptId variable whose string representation is shown in Figure 3.1c. The variable itself is a concatenation of the task’s ID, a timestamp, the parent job’s ID, a character indicating whether the task is a map or reduce, the attempt ID, and underscores to separate each part. Since these concatenations may be decision based, the source parser would need to track control and data-flow to determine the precise format of the variable. At the very least, determining the precise format requires inspecting each of the classes which generate these concatenated variables. This problem is exacerbated when Hadoop is updated since the execution paths need to be reevaluated.

The problem is similar if we try to work backwards from the generated log messages. To generate all possible expressions, we need to exercise every execution path to generate all possible types of log messages. Even if this were possible to a large extent, there would still be several hundred variable value
Figure 3.1: A log message generated using concatenation and the code which generates it.

Figure 3.2: Two messages showing an attempt ID which is also part of a related client ID.

Figure 3.3: A datanode UUID with its label specified in two different formats.
semantics to label across the thousands of log message types. Simply using find-and-replace to label each variable’s semantic with a Grok expression could make subtle errors. Figure 3.2 shows two log messages where the attempt ID in the first message is a substring of the client ID in the second message. A find-and-replace operation for the attempt ID incorrectly modify both IDs, even though they represent separate objects. Find-and-replace also does not work well when the variable value’s semantic is not specified consistently. Figure 3.3 shows two log messages which both print the datanode’s universally unique identifier (UUID). However, one specifies the name of the variable’s type as “Datanode UUID” while the other specifies it as “datanodeUuid”. Finally, if the source code is not available, it is conceivably easy for us to miss or incorrectly parse a variable value.

harvester can solve these problems in more than 76% of cases (see Chapter 6). Since it generates Grok expressions on demand, harvester does not need to exercise every execution path before use. Furthermore, harvester can operate without source code, using only the binaries of the program(s) which author each log file. harvester operates by trying to find all constant segments in a message, effectively finding all variable segments as well. As Chapter 6 shows, this process is not entirely accurate. But it is a good approximation and at least automates several steps before the user needs to manually refine the expressions.

Once we have a satisfactory list of Grok expressions, we need to configure FileBeat to discover Hadoop’s log files and stream them to LogStash; and we need to configure LogStash to parse the messages using our generated expressions. Figure 3.4a shows the configuration necessary to monitor Hadoop’s logs directory for log files. The fields property is used in LogStash to specify which expressions should be used for parsing; see the filter section of Figure 3.4b. Alternatively, we could allow LogStash to try all known expressions (including our generated ones) on every log file, but this has the potential to use an existing expression which is too general (instead of our more specific expressions). Nonetheless, the minimum configuration necessary to parse Hadoop’s log files is to have FileBeat monitor Hadoop’s logs directory and send any messages to LogStash. This is not very difficult considering the amount of configuration distributed systems usually require, but it is an added step that must be done for every application, increasing the possibility of user error.
filebeat.prospectors:
- type: log
- enabled: true
- paths:
  - /home/user1/hadoop-2.8.3/logs/*.log
- fields: {log_type: hadoop}

(a) filebeat.yml: FileBeat’s configuration file

input {
  beats {
    port => "5043"
  }
}

filter {
  if ([fields][log_type] == "hadoop") {
    grok {
      patterns_dir => "./patterns"
      patterns_glob => "hadoop"
      match => { "message" => "%{MESSAGE}" }
    }
  }
}

output {
  ...}

(b) logstash.conf: LogStash’s configuration file

Figure 3.4: Configuration necessary to discover and parse Hadoop logs using user-generated expressions.
Chapter 4

Design

The primary goal of harvester is to find all log files in a system and parse them into a format appropriate for input into applications like ElasticSearch or Stitch. For a given log file, these applications intake a single message at a time, parsed into three components: the complete message string, the message’s timestamp, and any variable values in the message. In addition, each variable value is labelled with its semantic (e.g., “datanodeUid”) and whether it is an object identifier (e.g., an HDFS block ID) or not (e.g., a file size). Figure 4.1a shows an example message from Spark parsed into the fields shown in Figure 4.1b. Besides the complete message string and timestamp in milliseconds from the epoch, there are two variable values; one is an identifier, the application ID, and one is a non-identifier, the application’s exit status. Similar to LogStash, each message is parsed using one of many Grok expressions.

Figure 4.1c shows a Grok expression which matches the message in the previous example. The last line is the complete expression with each field being matched using one of the subexpressions on the previous lines. All fields in the expression correspond to variables, with the exception of the timestamp field. We prefix semantics with a dollar sign to denote identifier variables. Note that the timestamp expression is not actually a Grok expression; we discuss its format and use in Chapter 5.

4.1 Overview

There are four high-level steps to harvester’s operation: First, harvester finds open log files and the executables used to write each one. Next, for each file, harvester extracts a message at a time. For each message, harvester uses the constant (hard-coded) strings (henceforth referred to as constants) stored in the executables to eliminate all constants from the message. The remaining segments of the message, are treated as variable values and labelled using a series of heuristics. This finally results in a Grok
Figure 4.1: An example log message and the grok expressions used to parse it.

expression that can be used to parse the corresponding type of message.

### 4.2 Log Discovery

*harvester* discovers log files and the executables used to write each one by periodically scanning all open files in the system. This has two noteworthy limitations. First, old log files will not usually be opened and thus will not be discovered. Or if they are opened, but not by their authoring application, they will only be partially parsed since we will not have the constants originally used to author each log file. The second limitation is similar in that a log or executable file may be opened and closed faster than we can discover it. This makes *harvester* more suitable for long-running applications rather than ephemeral ones. Nonetheless, a user can always remedy either limitation by specifying the necessary log or executable files.

To ensure the accuracy of *harvester*’s algorithm, we must find the precise set of executables used to author each log file. If we include too many executables when parsing a log file, we may mistakenly eliminate a variable value. For example, Hadoop’s unit-test jars include constants that mimic common variable values. On the other hand, if we include too few executables, we may create an erroneous variable value because we did not identify a constant as such. Thus, if we discover an log file open for writing, we associate the writing process’ executables with the log file. In addition, we include the
executables of any other processes passing messages to the current process, since these messages may be logged. For example, on Linux, several processes write messages to a socket connected to a logging daemon (syslog) that writes these messages to disk or forwards them to other machines.

### 4.3 Log Parsing

To parse a message, *harvester* first searches its list of known Grok expressions for one that matches the message; if *harvester* finds no match, it then generates a matching Grok expression. Generating an expression primarily requires determining what parts of the message are constant and what parts are variable (corresponding to the message’s variable values). To do this, *harvester* uses the set of constants from the executables which potentially authored the message. This set is used to create a combination of constants which match non-overlapping parts of the message. The combination which matches the maximum number of characters in the message should accurately identify the constant parts of the message. Formally, let

- \( L[0..n-1] \) be the unmatched log message of length \( n \)
- \( |L[0..n-1]| \) be the length of \( L[0..n-1] \)
- \( \mathbb{C} \) be the set of all constants extracted from the executables
- \( M[n-1] \) be the combination of non-overlapping constants (from \( \mathbb{C} \)) which matches the maximum number of characters in \( L[0..n-1] \)
- \( |M[n-1]| \) be the number of characters matched by \( M[n-1] \)

We then define \( M[i] \) recursively as,

\[
M[i] = \max\{match(0, i), \max_{0 \leq j < i} \{M[j] + match(j + 1, i)\}\}
\]

where \( match \) is defined as,

\[
match(a, b) = \begin{cases} 
  b - a + 1 & \text{if } L[a..b] \in \mathbb{C} \\
  0 & \text{otherwise}
\end{cases}
\]

*harvester* uses the dynamic programming in Algorithm 1 to compute \( M[n-1] \). Intuitively, \( M[i] \) should either match more characters than \( M[i-1] \) or the same number of characters. Therefore, we keep track of the maximum number of characters matched so far in the variable \( max \). At the \( i \)th iteration, the algorithm first tries to find a constant which matches the entire segment \( L[0..i] \) (line 5).
Algorithm 1: Computing $M[n-1]$

**Input:** $C$ - constants from executables, $L[0..n-1]$ - the log message

**Output:** $M[n-1]$ - combination of non-overlapping constants matching the most characters in $L[0..n]$

1. begin
2. $M \leftarrow$ array of length $n$
3. /* Number of characters matched by best combination */ $max \leftarrow 0$
4. for $i \leftarrow 0$ to $n$ do
5.   if $L[0..i] \in C$ then
6.     /* Combination of $L[0..i]$ is optimal */
7.     $M[i] \leftarrow L[0..i]$
8.   else
9.     for $j \leftarrow 1$ to $i$ do
10.    if $L[j..i] \in C$ and $|M[j-1]| + |L[j..i]| > max$ then
11.       /* Combination of $M[j-1]$ and $L[j..i]$ is optimal */
12.       $M[i] \leftarrow M[j-1] \cup L[j..i]$
13.     end
14.   end
15. if $i > 0$ and $|M[i]| \equiv 0$ then
16.   /* Copy combination from previous iteration */
17.   $M[i] = M[i-1]$
18. end
19. $max = |M[i]|$
20. end

If no such constant exists, the algorithm tries to find a constant which matches the segment $L[1..i]$ (line 10). If this constant exists, and combined with $M[0]$ it would match more characters than $max$, then $M[i]$ is the union of $M[0]$ and this constant (lines 10-11). However, if this constant does not exist, this process is repeated for $L[2..i]$ and $M[1]$, $L[3..i]$ and $M[2]$, and so on, shrinking the segment from the left by one character each time (line 9). If no combination exceeds $max$ and there is a previous combination to copy, $M[i]$ is set to $M[i-1]$ (lines 15-16). Overall, this algorithm has a worst case time complexity of $O(n^2)$ since for every $i$, it must try $L[0..i]$ to $L[i..i]$. The space complexity is $O(n)$ since we only store the optimal solution for each $i$.

4.4 Identifier Inference

Algorithm 1 is effective at finding constant segments. However, given the ad-hoc nature of logging, several heuristics described in Chapter 5 are necessary to improve the accuracy of each variable’s boundaries as well as to determine each variable’s semantic.
Chapter 5

Implementation

We have implemented harvester in C++ for Linux, with the goal of having it continuously monitor each node in a cluster. No feature of harvester is specific to Linux, so the design principles in this and the previous chapter should be portable. In order to monitor and read all log files in a cluster, harvester must be run as the root user. However, if not all applications need to be monitored, harvester could be run with fewer privileges corresponding to the applications of interest. We currently support two types of executables – files using the executable and linkable format (ELF), typically generated by C/C++ applications, and Java classes.

5.1 Discovering Log and Executable Files

harvester periodically scans Linux’s /proc pseudofilesystem to discover log and executable files as well as open pipes and sockets. We chose a period of five seconds to minimize average CPU usage. For each process in /proc, harvester creates a temporary data structure to keep track of its files, pipes, and sockets. Each log file is identified by its device ID, inode ID, and path, in case the file is unlinked and

<table>
<thead>
<tr>
<th>Conditions</th>
<th>Path Properties</th>
<th>Type Detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>File-type String</td>
<td>Path Properties</td>
<td>Type Detected</td>
</tr>
<tr>
<td>Contains &quot;text/&quot;</td>
<td>Contains &quot;.log&quot;</td>
<td>Log file</td>
</tr>
<tr>
<td>Contains &quot;x-application&quot; or &quot;x-executable&quot; or &quot;x-sharedlib&quot;</td>
<td></td>
<td>ELF file</td>
</tr>
<tr>
<td>Contains &quot;java-archive&quot; or &quot;.jar&quot; or &quot;.war&quot;</td>
<td></td>
<td>JAR file</td>
</tr>
<tr>
<td>Contains &quot;.zip&quot;</td>
<td>Extension is &quot;.jar&quot; or &quot;.war&quot;</td>
<td>JAR file</td>
</tr>
<tr>
<td>Contains &quot;.x-java-applet&quot;</td>
<td></td>
<td>Java class file</td>
</tr>
</tbody>
</table>

Table 5.1: Heuristics used to recognize file type
replaced while we are reading it (e.g., due to log rollover). *harvester* searches for log files within each process’ open file descriptors. For executables, *harvester* also searches within each process’ open file descriptors, its memory mapped regions, and its command line string. To determine if a file is a log or executable, we inspect its file-type string using the libmagic library [14] and then use the heuristics summarized in Table 5.1. Note that if our heuristics are wrong about a log or executable, *harvester* will simply fail to parse the file, ignoring it if it is found again. Once the scan is complete, we analyze the open pipes and sockets.

*harvester* needs to associate all executables which may have contributed to each log file, including those communicating using message passing. To do this, it first builds a communication graph of all processes. Each process writing to a pipe is given an edge from them to the process reading from the same pipe; the two ends of a pipe can be associated since they use the same inode number. This method cannot be used for sockets however, since connected sockets do no use the same inode. In this case, *harvester* first builds an inode to process map and then uses the Netlink service to ask the kernel for all socket pairs. These are then used to connect the two corresponding processes in the direction of the sockets’ open modes (e.g., the process writing to a socket has an edge to the process reading from the same socket). Finally, *harvester* runs a breadth-first search from each process in the graph, adding its executable files to its children, so that any logs written by the children will have an accurate set of authoring executables.

Next, *harvester* creates or updates the log parser corresponding to each log file. If a parser already exists, *harvester* verifies that it is using the same set of executables as the scan discovered; any missing executables are parsed, and their constants added to the parser. If the parser is new, *harvester* checks every other parser to see if they are using the same set of executables. If so, the parsed constants can be shared (using a shared pointer) rather than incurring the time and memory cost of re-parsing. This means that each parser’s set of executables must be treated as immutable once it is established so that updating one log file’s constants does not affect another log file’s constants. If no existing parser has the same set of executable files, the executables’ constants are extracted and inserted into a new hash set for the parser.

### 5.2 Extracting Constants from Executables

To parse constants from each executable, *harvester* reads their constant section(s) and processes any constants it finds. For each ELF executable, *harvester* uses the libelf-dev [13] library to parse the section headers and find the `.rodata` and `.rodata1` sections (if they exist). *harvester* then extracts all constants
from these sections. For each Java class file, harvester reads constants with the CONSTANT_String, CONSTANT_Class, and CONSTANT_NameAndType tags from its constant pool. The CONSTANT_Class tag is used to represent class names while the CONSTANT_NameAndType tag is used to represent field and method names. Since there are least three different ways to print a class’ name in Java, harvester assembles and adds the name in all possible forms. Java archives are decompressed first using the libzip [15] library. For both executable formats, harvester ignores any constants containing non-printable ASCII characters since all the English logs we observed used just ASCII. Additionally, constants containing potential format-specifiers (e.g., “%s”) are duplicated and split on these format specifiers, so that the constant parts around them may be matched by harvester’s algorithm. Once complete, harvester is ready to parse each log file.

5.3 Extracting Messages from Log Files

The first step in parsing a message is to extract it from the log file. Since messages may contain multiple lines, harvester uses the message’s timestamp as a delimiter. Extraction begins with harvester trying all known timestamp patterns against the current line in the file. If a match is found, harvester adds this line to the current message, and tries the patterns on the next line, starting with the pattern which was just matched. If no timestamp is found in the next line, a line break is appended to the message, followed by the line. This continues until a line with a timestamp is found, at which point the message has been completely extracted. If a line without a timestamp is found when the message is still empty, we assume the line is itself a message. This sometimes occurs at the beginning of log files during the software’s initialization sequence. However, if this is done for the entire file, then we will have parsed a file without timestamps by mistake. To prevent this, we search up to 25 lines of the file to ensure it contains timestamps.

Since harvester relies on known timestamp patterns to parse timestamps, we must treat the timestamp field specially in Grok expressions. The Grok expression for the timestamp type is a list of possible patterns specified using format specifiers defined by the C strftime [21] function. Since strftime does not define a specifier for milliseconds, we define one ourselves – %3.

5.4 Heuristics

After implementing Algorithm 1, we found several heuristics were necessary to refine the variables detected.
5.4.1 Filtering Constants

At first, when we ran *harvester* on logs from Hadoop, we found *harvester* identified all messages as constant. This was because every permutation of one and two-character ASCII words existed in the authoring executables. Ignoring all constants less than three-characters allowed *harvester* to find some variables, but characters that used to be single-character constants (e.g., a colon) were now being detected as variable. Thus, *harvester* also ignores any variables without alphanumeric characters. Nonetheless, most variables containing numbers were still being detected as constant. Adding a condition to ignore all constants containing numbers without any alphabetical characters produced the best set of variable matches. A final heuristic we applied was to treat constant strings corresponding to enum values as variable values instead.

5.4.2 Identifying Variable Boundaries

Since variable values can contain constants themselves (e.g., “blk_” in “blk_1073741825_1001”), *harvester* extends each variable segment to the boundary of the variable value it overlaps. Our solution to this relies on the observation that most variable values are printed between two delimiters. Thus, we can extend any variable segment that falls between common delimiters (e.g., a space, colon, equals sign, slash, etc.) to the extent of the delimiters. This can end up including trailing punctuation like periods and exclamation marks, so after the extension, *harvester* trims these characters from the segment. Note however, that some variable values can indeed contain these delimiters; for example, an IPv6 address contains colons. *harvester* solves this when determining the variable’s syntax by combining consecutive variables with no possible semantic in-between.

File paths and universal resource identifiers (URIs) are another category of variable values which may span multiple delimiters in addition to containing nested variable values. For example, `hdfs://node1:8200/user134/job_151529522293_0005/job.jar` is a URI containing a hostname, port, username, and job ID. We refer to these types of variable values as compound values. These values are typically the result of the system using existing individual variable values in file paths. Thus, *harvester* recognizes them specially using simple regular expressions.

5.4.3 Determining a Variable’s Semantic and Syntax

The syntax used to match a variable value is simply the combination of its constant segments and “.*?” in place of its variable segments. For example, the syntax for “attempt_151529522293_0002_r_000003_0” is “attempt_.*?_r_.*?”. We use “.*?” instead of “.*” for improved performance; “.*” is greedy and
first assumes that the segment following the variable value may be part of the value. It must then backtrack if it is wrong. For example, the “.*” in the expression “abc.*def” applied to the string “abc123def” would first consume “123def” and then backtrack when it realizes no characters follow “def”. In practice, the segment following the variable is rarely (if ever) part of the variable, so “.*?” is better for performance.

A variable’s semantic may be embedded or standalone. We define an embedded semantic as one where the variable value itself identifies its semantic. For example, we treat the “blk” in “blk_1073741825_1001” as the variable’s semantic. This is assembled by concatenating all constant segments of the variable value. In contrast, a standalone semantic is one that is separated from the variable value. For example, we treat “ipcPort” in “ipcPort=8808” as the variable’s semantic. Determining a standalone semantic is more challenging since the name may be spread across several words preceding the variable. For example, in “retry cache entry expiry time is 600000 millis”, the semantic is “retry_cache_expiry_time” and precedes even the word directly before the variable value. We solve this with a series of heuristics.

We search for the standalone semantic by successively considering each word preceding the variable value. The goal of this search is to find a noun and any associated verbs that are likely to be the value’s semantic. Prepositions (e.g., “at”) directly preceding the variable are skipped over since they do not function as part of the semantic. If a word appears to be in camel-case (e.g., “ipcPort”) or has non-whitespace delimiters, we assume this is the semantic since camel-case or custom delimiters perform the same function as whitespace in semantics that are a series of words. Once we find the first noun (searching backwards), we stop the search at the next observed non-noun (i.e., a verb, adverb, adjective, etc.) or the next non-whitespace delimiter; both function to delimit where the set of words associated with this noun end. Finally, we concatenate all the words together using underscores (Grok expressions do not support spaces in semantics).

In compound variable values, the nested values sometimes do not specify a semantic. For example, in the previous URI, “user134” is not labelled as a username. In these cases, if harvester has previously come across the variable value, it uses the previously assigned semantic. Naturally, this requires that the variable value is random enough that it will not map to two different semantics. Therefore, we only use this heuristic when the value contains both alphabetical and numeric characters. To ensure we encounter these individual variable values before they are used in compound values like file paths, we attempt to parse log messages from separate files in order of their timestamp.
5.4.4 Identifier Identification

We determine whether a variable value is an identifier or not using the value’s semantic. If the semantic is embedded, or the word preceding the value is a noun, we automatically assume it is an identifier. This is intuitive since the variable segment of an identifier is typically part of the noun which denotes the identifier. For example, in “port=80”, together “port” and “80” is a noun, whereas in “containers allocated = 9”, “allocated” and “9” together are not a noun. Additionally, we treat any noun semantic containing one of 21 words like “capacity”, “threshold”, or “number” as non-identifier semantics. Finally, all other semantics are treated as non-identifiers with one exception – all host:port combinations are treated as identifiers.
Chapter 6

Evaluation

Our evaluation focuses on determining the accuracy of the Grok expressions *harvester* generates. *harvester*'s accuracy is correlated with the amount of automation it provides in generating Grok expressions, and by extension, parsing messages. In other words, if the expressions do not need any refinement, *harvester* has fully automated the process. If a few edits are necessary compared to the number of generated expressions, *harvester* may still have saved hours of developer time.

6.1 Workloads

We evaluated *harvester* on a series of benchmark workloads for Hadoop 2.8.3 and Redis 4.0.6. For Hadoop, we used the TeraSort, PageRank, and Bayesian Classification workloads from the HiBench [10] benchmark suite. The TeraSort workload uses a series of map and reduce tasks to sort a large amount of data stored in HDFS. The PageRank workload runs the PageRank algorithm [2] on a series of webpages. The Bayesian Classification workload runs the Naive Bayes algorithm on a series of documents. For these workloads, HiBench was configured to use its “large” scale factor. Collectively, these workloads should touch every component of Hadoop. For Redis, we used the *redis-benchmark* tool distributed with Redis. We configured it to run 1,000,000 requests with 50 concurrent clients.

6.2 Configuration

All workloads were run on a cluster of 24 lightly-loaded server machines. Table 6.1 lists the specifications of the machines. For each application, one of the two-processor nodes was arbitrarily configured as the master and the other 23 as slaves.


<table>
<thead>
<tr>
<th></th>
<th>4 Machines</th>
<th>20 Machines</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
<td>2 x Intel Xeon E5-2630 v3</td>
<td>1 x Intel Xeon E5-2630 v3</td>
</tr>
<tr>
<td></td>
<td>16-core @ 2.4GHz</td>
<td>16-core @ 2.4GHz</td>
</tr>
<tr>
<td><strong>Memory</strong></td>
<td>256 GB DDR4 RAM</td>
<td>128 GB DDR4 RAM</td>
</tr>
<tr>
<td><strong>Hard Disk</strong></td>
<td></td>
<td>2 x 7,200RPM</td>
</tr>
<tr>
<td><strong>Network</strong></td>
<td></td>
<td>10Gbps interconnect</td>
</tr>
<tr>
<td><strong>Operating System</strong></td>
<td></td>
<td>Linux 4.4.0</td>
</tr>
</tbody>
</table>

Table 6.1: Specifications of the evaluation cluster

*harvester* was started under an isolated non-root user before the target software was started, and shutdown only after the target software was shutdown. Running *harvester* and the target under an isolated user ensures that the generated Grok expressions correspond only to the target software. Encompassing the startup and shutdown phases of the target application ensures *harvester* sees and generates Grok expressions for a complete series of log messages (from startup to shutdown).

### 6.3 Accuracy Results

We measure accuracy as the simple ratio between the number of correct Grok expressions versus the total number of generated expressions. Overall, *harvester* generated 728 expressions of which 206 were erroneous, 29 were extraneous (i.e., they could be replaced by just three expressions), and 493 were correct. This works out to an accuracy of 67.7%. By disabling the heuristic which treats enum constant strings as variable values, we found accuracy only decreased by 1.4%, suggesting this heuristic was not very effective. However, in truth, 90 expressions are affected by this change. It just so happens that a majority of them are also affected by another inaccuracy meaning the heuristic’s benefits cannot yet be seen. The heuristic which treats `host:port` combinations as identifiers contributes to 0.77% of the total accuracy. Overall, this level of inaccuracy might present a challenge to users of *harvester*, but several inaccuracies are easily resolved.

We observed eight classes of inaccuracies across the target software, summarized in Table 6.2.

The most prevalent inaccuracy occurs when variable values have no semantic specified in the log messages. This means even when generating expressions manually, a user would need to refer to the source code to determine the variable values’ semantics. Since most of these values are enums, we could label them with their enum type as a best-effort solution; we hope to explore this in future work.

The next most prevalent inaccuracy is due to variable values which match constants such as “root” and “default”. We hope to address this also in future work, beyond just detecting constants that match enum values.
Inaccuracy Class | # Occurrences
--- | ---
**Variable values without a semantic**: these values cannot be labelled with a semantic without domain knowledge | 54
**Variable values which are constants**: these values are not recognized as variables because they match constants | 38
**Non-identifiers recognized as identifiers**: these values are labelled as identifiers but in reality, they are non-identifiers | 37
**Non-descriptive type-name**: these values have a type-name which can be ambiguous in a global context (e.g., “port”) | 23
**Variables values logged as data structures**: the members of these values are labelled but not including their parent’s semantic | 20
**Variables with file extensions**: these values are suffixed with a file extension making them appear as other values”) | 18
**Variable values logged in a sentence**: the semantics of these values are implicit or obfuscated by the structure of the sentence | 16

Table 6.2: Inaccuracy classes and their occurrences

![Log messages](image)

Figure 6.1: Two log messages showing variable values logged and labelled as data structures rather than individually.

The third highest inaccuracy is when non-identifiers are mistakenly identified as identifiers. However, the majority of this inaccuracy (35 incorrect expressions) is due to how software versions are specified. For example, Hadoop’s version is frequently logged as “hadoop-2.8.3” which harvester’s heuristic treats as an identifier with an embedded semantic. We can resolve this by generating constants from the path of each executable. This would raise the overall accuracy to 72.5%.

A notable category of inaccuracy is caused by logging values within the structure of an array or dictionary. Figure 6.1 shows two examples of this. The first is where a list of replica host:port combinations is logged. Since each combination is not preceded by a type-name, we combine the values and give them the type-name “replicas”. The second example is where a type of tuple is logged. The individual variable semantics are general while the parent semantic is specific; however, we do not detect the parent semantic in our parsing. Both inaccuracies could be resolved by flattening the semantics at the cost of more verbose logging. Thus, the array of values would have each value preceded by its semantic and the tuple would have each field’s semantic preceded by its parent’s semantic. Since Hadoop’s logging is already quite verbose, this should not be a significant cost.

The next inaccuracy is due to variable values that are used as filenames with extensions. The extensions are automatically captured as part of the variable value, thus making it appear as a dif-
ferent value altogether. For example, “container_151529522293_0005_01_000001” versus “container_151529522293_0005_01_000001.tokens” Since the extension is constant and at the end of the value, it is easy to automatically eliminate, further raising accuracy to 75%.

Another notable inaccuracy is due to logging values in a sentence. Figure 6.2 shows an example of this. Reading the sentence, it is clear that the first value’s semantic should be something like “numEntries” and the second, “numLookups”. However, none of our heuristics are able to deal with this because the semantic is specified after the variable and there is no mention of it being a number.

6.4 Logging Guidelines

Based on the inaccuracies we observed and the heuristics we implemented, we have determined a few practices which should improve the ease of parsing plain-text developer-written logs. The first is to always clearly label values with their semantic in a globally unique way. This means avoiding writing out verbose sentences when a simple key-value pair will suffice. More specifically, spaces should be avoided for semantics, in favour of another delimiter such as an underscore. In addition, the semantic should not be so general that it easily conflicts with other variables’ semantics. Next, variables reporting the count of an item such as the number of blocks should be prefixed with “num” for easier identification as a non-identifier. Finally, if a data-structure needs to be logged, ensure each of its members are prefixed with the data-structure variable’s semantic. Applying these guidelines and rerunning our evaluation, we were able to increase accuracy to 90%.
Chapter 7

Limitations and Future Work

For the process of discovering log files, harvester has two limitations: First, in order to discover all log files on a system, it requires root privileges since some files may be protected. This could be avoided by having the user configure the appropriate privileges, but may defeat the purpose of log discovery if these privileges are configured per application. The second limitation is that harvester’s log-discovery may not be suitable for short-running applications. This is because harvester discovers log files and the executables that write to them by periodically scanning the system for open files. Short-running applications like a typical cron job may exit before harvester has a chance to discover all of their log and executable files. We could address this by extending harvester to search and monitor the directories containing existing log files. In this way, old log files could be discovered and new log files would notify harvester, triggering a scan before they are closed again.

harvester has four limitations in parsing log files: First, harvester cannot support files which do not use timestamps as a delimiter or which use a timestamp format that harvester does not recognize. In the future, we hope to develop a more automated method of detecting different timestamp formats. Second, harvester does not currently support log files written in the UTF-8 format because we did not observe them in our evaluations on open-source software. However, log files written in other languages typically use UTF-8. Similarly, harvester’s heuristics are developed primarily for the English language. We hope to remedy both these limitations with further evaluation on UTF-8 log files, especially those in other languages to see if they follow the same logging patterns as those in English. Moreover, we hope to use more formal Natural Language Processing (NLP) techniques to better our existing heuristics. Finally, harvester does not support variable values which are constants themselves. We hope to address this by trying to detect which values are names of types or which follow the pattern of how other variable values
are printed.
Chapter 8

Related Work

Of the three requirements motivating this work (discovering log files, extracting variable values, and assigning semantics), existing tools only tackle automated log parsing to extract variable values. They approach this by building regular expressions in one of two ways: using static analysis on the application’s source or bytecode, or using pattern matching across several log messages. Static analysis can be more precise at identifying variables than harvester’s constant-matching algorithm, but it can also be more difficult to write. Pattern-matching can work without the applications’ source code or binaries, but is probabilistic and requires some amount of training before accuracy is achieved.

8.1 Tools Based on Static-analysis

Sherlog [24] uses the format string of a logging statement to build a corresponding regular expression. For example, the statement `printf("removing directory, \%s", dir)` might generate the expression "removing directory, (.*)"\(^1\). However, logging statements may be more complicated when constant strings are logged or when basic printing functions are wrapped in functions that provide extra functionality. For example, Gnulib’s `error(int status, int errnum, const char *message, ...)` function prints the message with the provided arguments, in addition to printing the error string corresponding to “errnum” and exiting with “status” if it is non-zero. Thus, to use Sherlog, developers must indicate the format of their logging statements as well as which variables in an expression should be considered constant.

lprof [28] statically analyzes the application’s Java bytecode to determine how a log message is constructed through the concatenation of several constant strings and variables, eventually using this

\(^1\)Details of the exact regular expression generated are not included in the Sherlog paper. This expression is an approximation based on the functionality required.
to build a regular expression. For example, the logging statement 
\texttt{LOG.info("Clients are to use \\
\quad \texttt{clientNamenodeAddress + "to access"...\})} would result in bytecode which called the append method of a 
\texttt{StringBuilder} object to add each constant string. To add \texttt{clientNamenodeAddress}, its string representation 
would first need to be generated by calling the variable’s \texttt{toString} method. Thus, static analysis must look 
for each logging statement and then reverse-engineer the \texttt{StringBuilder} object passed to this statement, 
if at all. Naturally, this again requires that the developer specify the format of their logging statements.

Overall, static analysis requires some manual effort and is not invulnerable to errors. The process 
of indicating the format of logging statements needs to be repeated for every application using a new 
logging library. Moreover, the static analysis needs to be written for every common logging format in all 
languages that need to be supported. For example, C applications typically use format strings, but C++ 
or Java applications might use a combination of format strings and concatenation. Finally, this effort 
may still mistake constants for variable values when constant strings are logged as variables. In contrast, 
\texttt{harvester} can mistake variable values as constant strings but the manual effort required is significantly 
reduced; \texttt{harvester} only needs to implement a constant string extractor for each supported executable 
format. As a rough indication of the difference in difficulty, lprof’s expression generation required 7,435 
lines of code, whereas \texttt{harvester} was able to generate expressions including the semantic of each variable 
for two executable formats with less than 5,000 lines of code.

8.2 Tools Based on Pattern-matching

Tools like Spell [3] compare incoming messages to find the longest common subsequence (LCS) that 
matches at least half of an existing message. These matching segments are considered constant and the 
non-matching segments are treated as variable. Tools like LogMine [9] operate similarly, but use the 
number of edits necessary to change one message to the other as well as clustering. Although these tools 
can be run in full isolation with only access to the log files, their use of similarity is probabilistic and 
may not be as precise as \texttt{harvester}, nor Sherlog or lprof. Moreover, these tools need to process some 
amount of log messages before they can generate precise expressions, whereas \texttt{harvester} can work on any 
message, in isolation of all others.
Chapter 9

Conclusion

*harvester* is a novel tool able to extract and label variable values from log messages. These parsed messages are invaluable for data analytics, debugging, and performance analysis. *harvester* extracts values using the constants extracted from the executables that authored the messages. *harvester* then labels the values using a series of heuristics based on our evaluation on real-world distributed systems logs. Finally, *harvester* automatically generates a type of regular expression which can be used to parse these messages in the future. Overall, *harvester* achieves 67.7% accuracy in the expressions it generates, although this could be improved with marginally better logging practices.
Bibliography


