An Introduction to log$\mathcal{W}$eaver (v2.8)

Jean Goubault-Larrecq[1,2]

[1] GIE Dyade, INRIA Rocquencourt
Domaine de Voluceau, BP 105
78153 Le Chesnay Cedex

[2] LSV, ENS Cachan
61, av. du président-Wilson
94235 Cachan Cedex

September 20, 2001

Contents

1 Introduction

Keeping and managing event logs is a standard and fairly universal way of ensuring basic security, whether at the application, system or network level. In particular, it is a cornerstone of *intrusion detection*, which relies on extracting useful information on potential or actual intruders to react accordingly.

Analyzing logs, however, is hard. Detecting intrusion patterns by hand quickly becomes infeasible as logs grow. Most intrusion detection systems include filtering and counting mechanisms [Pax98, Roe99], but this is not enough in general to eliminate false positives, and new mechanisms that attempt to detect combinations of patterns throughout the logs are required. To take an example from [Mou97], assume we would like to detect an intruder exploiting an old `sendmail` bug on Unix. This attack requires the intruder to copy some shell to `/usr/spool/mail/root` at a time where the latter does not exist, to set the setuid bit on it, and send a fake e-mail message to `root`; on old implementations of mail systems, as soon as `root` attempted to read his mail thereafter, the ownership of `/usr/spool/mail/root` was simply switched to `root`, therefore making a setuid-bit copy of a shell available to the intruder. Assume these events are logged. Detecting copies of shell files is a good clue that this attack or a similar one is attempted, and detecting that a non-`root` user is changing setuid bits too, however as a systems administrator we would like to be warned—automatically, if possible—only when the *same* user does both. Reports of one action without the other are *false positives*, where we are warned against a non-existent attack. Moreover, we might want to refine this by requiring that an e-mail was indeed sent to root after these two events happened. So we are looking at correlations between different entries in the log—the user has to be the same in each of the copy and setuid events—, together with constraints on the order in which events occur in the log.

logWeaver is a log auditing tool. That is, it takes a *log* as input, and processes it according to a *signature file*. The log is a list of events, like those produced by the `syslog` utility on Unix. logWeaver can read from several log formats, however, because it relies on a pre-processor to convert from several formats to a unique binary format that it understands (see Section 2). Moreover, logWeaver can work both *off-line*—using a log that may have been produced days ago—and *on-line*—detecting attacks as the log fills in. (Some call the latter mode of operation *streaming*.)

The signature file states which kinds of events should be monitored and reported on. logWeaver itself does not come with a standard library of attack signatures. The idea is that logWeaver may be included in a bundle, where various security utilities, along with logWeaver and one or several signature files, will be included. It is the responsibility of the packager to write signatures. Administrators at clients' sites may also change signatures, and in fact logWeaver allows one to modify the set of signatures while logWeaver is running, i.e., without having to stop it and relaunch it.

One of the main features of logWeaver is that it can filter, count and match regular expressions, but also detect correlations between events (insisting that the same user does both actions in the `sendmail` example above), while maintaining temporal relations (that the intruder copies the shell before it sends an email to `root`, for example).

Note also that logWeaver is a *generic* tool, which takes a log and a signature file and reports matches. While its typical application is in security, it is suited to any task that requires one to reach for complex sequences of events in large lists of events. Typical alternative applications are remote maintenance (detecting repeated failures, or correlated failures of hardware and software, or failures of different machines at the same customer's from lists of unsorted failures), or user preference tracking for example.

As of today, logWeaver compiles and runs on Unix and Windows. It has been tested on various Linux versions, and on Windows NT. More detailed information on the algorithms used in logWeaver can be found in [RGL01].

2 Architecture

The logWeaver tool is named `logw` under Unix. It is invoked typically by calling `logw` with the name of a preprocessor, whose role is to convert the log's format into logWeaver's own standard format, with the name of a signature file, and the name of a log to analyze. A typical command line is therefore:

```
logw -l⟨log-reader⟩ -s⟨signature-file⟩ ⟨log⟩
```

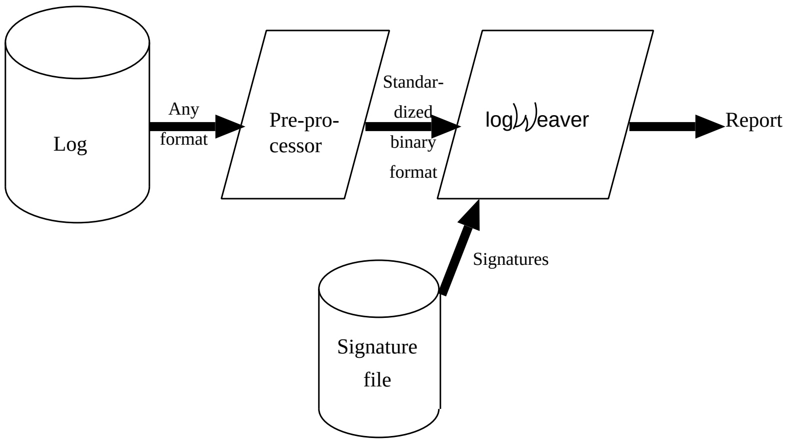

Figure 1: The log𝓦eaver architecture

```
Usage: logw [-h] [-V] -s spec-file [-l log-reader] [-e [neofs]] [log-file]
                [-c prefix] [-d [seconds]] [-r] [-b [seconds]] [-v [file]]
        -h: print this help message
        -V: print version and exit
        -s spec-file: monitor specs as given in spec-file.
        -l log-reader: use program log-reader as preprocessor for log-file
        -e [neofs]: report neofs end-of-file fake records at end of input
                    (default 1)
        -c prefix: checkpoint into and from prefix.ckp (default logweave)
        -d [sec]: checkpoint every sec seconds (default 10)
        -r: restart logw using last checkpoint file
        -b [sec]: block on read at end of log-file, polling every sec
                  seconds (default 0)
        -v [file]: verbose output to stderr [or file]
```

Figure 2: Command-line options

The architecture is as shown in Figure 1.

There are other command-line options to `logw`, which you can learn by calling `logw -h`. This should give you something like Figure 2.

The separation between log𝓦eaver and the preprocessor allows one to change the preprocessor at will. This way, log𝓦eaver accomodates log format changes independently from signatures, which can remain the same.

3 First Steps: Detecting Repeated Mouse Problems in Linux

Let us start with a simple example, and consider the `syslog` file given in the log𝓦eaver distribution. (This is one of the standard log files, coming from one of my laptops, covering three years of use.) Figure 3 shows an extract from this file, from line 27 to 33.

```
Jan 26 23:11:30 darkstar syslogd: exiting on signal 15
Jan 26 23:17:22 cecile syslogd: exiting on signal 15
Jan 26 23:20:16 cecile syslogd: exiting on signal 15
Jan 27 11:32:06 cecile syslogd: exiting on signal 15
Jan 27 12:23:14 cecile sendmail[103]: NOQUEUE: SYSERR(root):
      /etc/sendmail.cf: line 0: cannot open: No such file or directory
Jan 27 13:00:31 cecile insmod: Initialization of busmouse failed
Jan 27 13:00:32 cecile kernel: Unable to handle kernel paging
      request at virtual address c1005077
```

Figure 3: An extract from the `syslog` file

`line`	`date`	`machine`	`program`	`pid`	`comment`
27	Jan 26, 23:11:30	“`darkstar`”	“`syslogd`”	0	“`exiting on signal 15`”
28	Jan 26, 23:17:22	“`cecile`”	“`syslogd`”	0	“`exiting on signal 15`”
29	Jan 26, 23:20:16	“`cecile`”	“`syslogd`”	0	“`exiting on signal 15`”
30	Jan 27, 11:32:06	“`cecile`”	“`syslogd`”	0	“`exiting on signal 15`”
31	Jan 27, 12:23:14	“`cecile`”	“`sendmail`”	103	“`NOQUEUE: SYSERR(root):` [...]”
32	Jan 27, 13:00:31	“`cecile`”	“`insmod`”	0	“`Initialization of busmo`[...]”
33	Jan 27, 13:00:32	“`cecile`”	“`kernel`”	0	“`Unable to handle kernel`[...]”

Figure 4: An extract of the `syslog` file, as records

3.1 Log Format and Preprocessors

The format of syslog files on Linux is, as can be seen on the figure, a series of lines, with one line per event. Each line starts with the date, e.g., `Jan 26 23:11:30`. Then we find the name of the machine on which the event occurred: here `darkstar` or `cecile`. Follows the name of the service that emitted the event, for instance `syslogd` or `sendmail`. Optionally, the pid of the latter process is shown between square brackets (`[103]` on the `sendmail` line), then a colon followed by a free form message such as “`exiting on signal 15`” or the more exotic “`NOQUEUE: SYSERR(root): /etc/sendmail.cf: line 0: cannot open: No such file or directory`”.

The `linuxreadlog` executable in the log$\mathcal{W}$eaver distribution is the preprocessor for files obeying this format. Don’t try to call it yourself! (Unless you know what you are doing.) All preprocessors, whose names end in `readlog` by convention, are only meant to be called by `logw`.

Every log reader translates logs into sequences of *records* that `logw` can work on. The `linuxreadlog` log reader translates lines as shown in Figure 3 into records with a `date` field, which is a time value, a `pid` field, which is an integer (-1 is not present), and `machine`, `program`, `comment` fields that are just strings. In fact, `linuxreadlog` also adds an integer `line` field so as to help `logw` keep track of line numbers. For example, `linuxreadlog` will provide `logw` the sequence of records shown in Figure 4, given the lines of Figure 3. (We have used ellipses [. . .] to abbreviate parts of strings that are too long to fit on one line.) These records are transmitted in a simple binary format described in Section 7.

```
mouse_problems {
  .comment "mouse", .machine mach, .program prog,
                    .line line1, .date date1;
  .comment "mouse", .machine mach, .program prog,
                    .line line2, .date date2;
}
```

Figure 5: The `mouse_problems` rule

3.2 Basic Record Matching

Now look at the `t_mouse1.c` file in the log$\mathcal{W}$eaver distribution (see Figure 5). Although this looks like a C file, it is not: it is a signature file. The reason why its name ends in C is that the syntax of signatures is close enough to C that indenting mechanisms designed for C work on log$\mathcal{W}$eaver signatures.

The `t_mouse1.c` file declares one *signature*, or *rule*, called `mouse_problems`. The first line of its body only matches records whose `comment` field contains the word `mouse`, and stores its `machine` field into variable `mach`, its `program` field into variable `prog`, its `line` field into variable `line1`, and its `date` field into variable `date1`. Well, giving variable names x (e.g., `mach`, `prog`) does not exactly store values in x. Rather, it stores the value in x if x did not have any yet, otherwise it compares the value with the one x already had, and fails if they are not identical. Call this operation *match-or-store*.

Note that `.comment "mouse"` does not mean that the `comment` should *be* the string `mouse`, but that it should *contain* it as a substring. In fact, writing:

$$\texttt{.}\langle\text{field-name}\rangle\ \langle\text{variable-name}\rangle\ \langle\text{regular-expression}\rangle$$

asks for finding whether field ⟨field-name⟩ contains a substring that matches the given ⟨regular-expression⟩, if any, and if so match-or-stores it into ⟨variable-name⟩ (if any). (Both the variable name and the regular expression are optional.) So for example `.comment c "Init.*(bus|PS).*mouse.*fail"` will match any `comment` field that contains `Init`, followed by any number of characters, followed by either `bus` or `PS`, then by `mouse` a bit further away, and then `fail`. If matching succeeds, it will match-or-store the whole comment field into variable `c`. This way, we can keep only those messages where bus mice or PS/2 mice, but not serial mice, are reported to have failed some initialisation process.

In general, there is a seldom-used extension to this syntax which allows one to get back parts of the field that the regular expression matched. For example, writing :

```
.comment c "Init.*(bus|PS).*mouse.*fail" { mousetype = "\\1" }
```

will in addition match-or-store that part of the comment that matched the `(bus|PS)` part of the regular expression into `mousetype`. In general, `\\1`, . . . , `\\9` match the substring matched by the first, . . . , the ninth regular subexpression enclosed in parentheses. These features are those offered in Henry Spencer's regexp package, which was included in log$\mathcal{W}$eaver [Spe86].

3.3 Matching Rules

Whenever log𝒲eaver has matched the first line of `mouse_problems` against some record in the log, it will look for a subsequent record matching the second line of `mouse_problems`. This second line again asks for a record with a `comment` field containing `mouse`, with a `line` field that it will store into variable `line2`, a `date` field that it will store into `date2`; it must also have a `machine` field whose value, stored in `mach`, equals the one we have already gotten in matching the first line; it must also have the same `program` field than when matching the first line. This is where match-or-storing is important : the first time `.machine mach` is met, log𝒲eaver stores its `machine` field into `mach`, the second time it compares its `machine` field with the value stored in `mach`.

Match-or-storing may seem like a strange concept. It is just an operational explanation of a concept that is actually simpler when you put it formally, but has a less clear operational reading. The idea is that log𝒲eaver really only looks for pairs of records matching both lines of `mouse_problems`, looking at the same time for values of the `mach`, `prog` and other variables that will make matching successful.

Other variables are useful for reporting. Run log𝒲eaver with signature file `t_mouse1.c` on log `syslog` by typing:

```
logw -llinuxreadlog -st_mouse1.c syslog
```

You should get the output shown in Figure 6. (If not, consult Section 8.)

```
mouse_problems: mach=cecile line2=47 line1=33 prog=insmod
                date2=Sun Jan 28 10:36:47 2001 date1=Sat Jan 27 13:51:39 2001
mouse_problems: mach=cecile line2=61 line1=47 prog=insmod
                date2=Mon Jan 29 08:32:28 2001 date1=Sun Jan 28 10:36:47 2001
mouse_problems: mach=cecile line2=75 line1=61 prog=insmod
                date2=Tue Jan 30 07:27:40 2001 date1=Mon Jan 29 08:32:28 2001
mouse_problems: mach=cecile line2=89 line1=75 prog=insmod
                date2=Wed Jan 31 10:28:00 2001 date1=Tue Jan 30 07:27:40 2001
mouse_problems: mach=cecile line2=103 line1=89 prog=insmod
                date2=Thu Feb  1 15:30:03 2001 date1=Wed Jan 31 10:28:00 2001
```

Figure 6: Results of `mouse_problems` on `syslog`

Figure 6 shows 5 matches of rule `mouse_problems`. If you look at `syslog`, you'll realize that there are 6 lines where the string `mouse` occurs, corresponding to 6 "initialisation of busmouse" problems. These are lines 33, 47, 61, 75, 89, 103. Accordingly, log𝒲eaver reports 5 pairs of lines matching `mouse_problems`, namely 33–47, 47–61, 61–75, 75–89, 89–103. Notice how the extraneous `line` field provided by the log reader was used to collect line numbers into variables `line1` and `line2`, and how log𝒲eaver reports their values in successful matches.

```
mouse_problems synchronized(mach,prog) {
  .comment "mouse", .machine mach, .program prog,
                    .line line1, .date date1;
  .comment "mouse", .machine mach, .program prog,
                    .line line2, .date date2;
}
```

Figure 7: The refined `mouse_problems` rule

3.4 Managing Overlaps: Shortest Matches, Synchronization

However, most of these line pairs are redundant in that they overlap. For example, we might not be interested in knowing that there is an overlap between lines 47 and 61, since there is already an overlap between 33 and 47. This is redundant here because each of these lines must match with the same `machine` and `program` fields. In general, if you look for repeats of some event A in a log, and A occurs at lines i_1, i_2, …, i_n, you shall get reports of matches i_1–i_2, i_2–i_3, …, i_{n-1}–i_n. Notice that this is already better than reporting all possible pairs of matches : in the `mouse_problems` above, reporting all pairs would also report on lines 33–61, 33–75, 33–89, 33–103, 47–75, 47–89, 47–103, 61–89, 61–103 and 75–103, which would be too much indeed, and completely uninformative. The reason is that log$\mathcal{W}$eaver only reports on *shortest matches*: once 33–47 has been reported, attempts at reporting any other match between line 33 and some line later than 47 are silently abandoned.

One way to refine this a bit further is by requiring that no two reported matches of `mouse_problems` intersect, i.e., if `mouse_problems` matches from line a to b, and also from c to d, we require that $[a, b] \cap [c, d] = \emptyset$. More precisely, we may require that no two reported matches of `mouse_problems` with the *same values* of `mach` and `prog` intersect. The `synchronized` keyword then helps: defining a rule by adding the `synchronized (`x_1, …, x_n`)` declaration after its name tells log$\mathcal{W}$eaver to only report on matches of the rule that do not intersect in case the variables x_1, …, x_n received the same values. For instance, running the refined `mouse_problems` rule of Figure 7 with the command line :

```
logw -llinuxreadlog -st_mouse2.c syslog
```

yields the results shown in Figure 8.

```
mouse_problems: mach=cecile line2=47 line1=33 prog=insmod
                date2=Sun Jan 28 10:36:47 2001 date1=Sat Jan 27 13:51:39 2001
mouse_problems: mach=cecile line2=75 line1=61 prog=insmod
                date2=Tue Jan 30 07:27:40 2001 date1=Mon Jan 29 08:32:28 2001
mouse_problems: mach=cecile line2=103 line1=89 prog=insmod
                date2=Thu Feb  1 15:30:03 2001 date1=Wed Jan 31 10:28:00 2001
```

Figure 8: Results with the refined `mouse_problems` rule

3.5 Refining Rules with Constraints

We have seen in Section 3 how to write rules that mentioned a sequence of record patterns to be matched sequentially. Until now, rules were given by a name, optionally followed by a `synchronized` declaration, followed by a block (enclosed in curly braces), i.e., a sequence of record-matching statements terminated by semicolons (`;`). The semicolon has a precise meaning in log$\mathcal{W}$eaver: if there is any other record-matching statement after the semicolon, this means "then, later", i.e., matching a rule $S_1; S_2; \ldots; S_n;$ means finding some line i_1 where S_1 matches, some line $i_2 > i_1$ where S_2 matches (with the same values of the variables), …, and finally some line $i_n > i_{n-1}$ where S_n matches (with the same values of the variables).

```
bad_authentication synchronized(mach,uid) {
  .machine mach, .program "^PAM_pwdb$", .line line1, .date date1,
      .comment "authentic.*fail.*uid=([0-9]+)" { uid="\\1" };
  .machine mach, .program "^PAM_pwdb$", .line line2, .date date2,
      .comment "authentic.*fail.*uid=([0-9]+)" { uid="\\1" }
      | (date2 <= date1+3600);
}
```

Figure 9: Detecting two failed logins within one hour

Our mouse problem example is not really convincing, so let's consider the task of finding cases where somebody tried to login twice from the same account, and it failed twice within one hour. You can write it as in Figure 9 on Red Hat Linux (see file `t_auth1.c`). Note that we require that the program be exactly `PAM_pwdb` each time: the `^` caret symbol only matches at the beginning of the field, while the `$` dollar symbol only matches at the end of the field, so `^PAM_pwdb$` only matches the string "`PAM_pwdb`".

The complex regular expression used for both `comment` fields is meant to match comments in lines like the following (see the `messages.4` file in the distribution) :

```
Feb  9 16:54:11 abaca PAM_pwdb[2732]: authentication failure;
                              (uid=500) -> root for su service
```

Note how the `\\1` trick allows us to get back the value of the `uid` variable. By the way, although `uid` will contain the string "`500`" here, it is perfectly legal to use it as an integer. log$\mathcal{W}$eaver, like Perl, automatically converts strings, integers, and dates as needed.

Finally, observe that we have used a *constraint* `date2 <= date1+3600` in the second line of the `bad_authentication` rule. This allows us to only consider pairs of failed authentication attempts (by the same user `uid`) that fall within one hour (recall that 3600 is the number of seconds in one hour: `date1`, as a date, is first converted to an integer number of seconds, then 3600 is added, and `date2` converted to a number of seconds is then compared to the result).

Running this against the `messages.4` file as follows :

```
logw -llinuxreadlog -st_auth1.c messages.4
```

shows the message of Figure 10. Observe that there is another failed login attempt in `messages.4`, which was not considered because it was not followed within one hour with another login failure from the same user.

```
bad_authentication: mach=abaca line2=1264 line1=1263 uid=500
                    date2=Fri Feb  9 16:54:15 2001
                    date1=Fri Feb  9 16:54:11 2001
```

Figure 10: Results of `t_auth1.c` against `messages.4`

Using date constraints is also beneficial in that they help log*W*eaver reduce its memory consumption, at least in theory, and there is a caveat. For example, if you write a constraint of the form:

```
date2 <= "Sat Jun 30 21:49:08 2001"
```

then once log*W*eaver reaches a line dated after Saturday June 30th 2001, 21:49:08, it will recognize that it is too late to match this line and therefore any later line. In this case, it will quit trying to match the corresponding rule. (Note that log*W*eaver will convert the date string into the proper date automatically—but you have to adhere strictly to the `ctime` format: day of the week, then month [required], day of the month [required], time in the form HH:MM:SS and year.) The reason why log*W*eaver knows that when it is too late, it will be too late forever is that in the `syslog` format, as preprocessed by `linuxreadlog`, the `date` field is assumed to always increase, never decrease. Technically, this is due to `linuxreadlog` informing log*W*eaver that `date` fields have type `T` (monotonically increasing time values): see Section 7.

This also works with constraints of the form `date2 <= date1+3600`, such as in Figure 9, which are more useful: when the first line was matched, the value of `date1` was recorded, and if `date2` starts being too late in the second line (i.e., `date2` is greater than `date1` plus one hour), then log*W*eaver will realize there is no point in looking for a match of `bad_authentication` starting from `date1`.

However, there is a caveat… in fact, log*W*eaver won't realize this in most cases: the constraint `date2 <= date1+3600` is only evaluated when the record pattern that precedes it matches. So, in the example of Figure 9, log*W*eaver won't even evaluate the constraint on lines that have a `program` field other than `PAM_pwdb`, or that have a `comment` field that does not match the given regular expression. If no line matches the record pattern, log*W*eaver will never quit monitoring this rule, even though dates might grow arbitrarily.

The proper way to solve these problems is to write the rule as in Figure 11. This uses a slightly different record pattern as before (Section 3.2), of the form:

`.`⟨field-name⟩ ⟨variable-name⟩ ⟨op⟩ ⟨expression⟩

where *op* is `==`, `!=`, `<=`, `<`, `>=`, or `>`. The ⟨expression⟩ should be only mention variables that already have values from matches in previous lines. Anyway, if you use variables that don't have values from matches in previous lines, log*W*eaver will reject your rule with an error message.

```
bad_authentication synchronized(mach,uid) {
  .machine mach, .program "ˆPAM_pwdb$", .line line1, .date date1,
      .comment "authentic.*fail.*uid=([0-9]+)" { uid="\\1" };
  .date date2 <= date1+3600,
      .machine mach, .program "ˆPAM_pwdb$", .line line2,
      .comment "authentic.*fail.*uid=([0-9]+)" { uid="\\1" };
}
```

Figure 11: Detecting two failed logins within one hour, making sure time windows are obeyed

As an application of the remarks above, observe that it is important to put the `.date date2 <= date1+3600` pattern first in the second pattern of rule `bad_authentication` if you wish to have log𝓦eaver quit monitoring this rule as soon as dates are too late, not waiting to match the `machine`, `program`, `line` and `comment` fields first.

Try the following, and compare with our previous use of `t_auth1.c`:

```
logw -llinuxreadlog -st_auth2.c messages.4
```

This should take approximately as long as before. However, the difference should be clear when failed authentication events are more frequent, in which case `t_auth2.c` will be faster. (For experts: you may compare the number of active threads in each case by issuing the `-v` command-line option—verbose—and exploring the resulting trace file. See e.g. Question 8.7 in Section 8.)

4 Going Further: Loops, Flexible Variables, Checkpointing, and All That

It is often awkward to specify repetitions of events by repeating lines. With what we have seen of log𝓦eaver until now, the only way to check whether some event E occurs 42 times, is to write $E; E; \ldots; E$ with 42 Es. Fortunately, log𝓦eaver provides a loop mechanism to alleviate this problem.

4.1 Repeated Modprobe Problems in Linux

```
repeated_modprobe_problems synchronized(mach) {
  while (loop: _$loop<=3) {
    .machine mach, .program "ˆmodprobe$", .line $line;
  }
}
```

Figure 12: Detecting three consecutive `modprobe` related problems

Consider for example the `repeated_modprobe_problems` rule of Figure 12, which is meant to operate on Linux syslog files—that is, we shall use the `linuxreadlog` preprocessor. The line inside the `while` loop matches any line such that the `program` field is exactly `modprobe`, and gets the machine name in the variable `mach`. The new features in this rule are the funny variable name `$line`, the even funnier variable `_$loop`, the `loop` label, and the `while` loop. We now explain all of them.

The `$line` is an example of a *flexible* variable. Until now, all variables were *rigid*: once they get a value, they keep it forever; that is, rigid variables have one and only one value that does not change through time. On the contrary, flexible variables have values that evolve through time.

In logWeaver, flexible variables start with a dollar ($) sign, possibly preceded by underscores (_), while rigid variables start with a letter.

In the example of Figure 12, each run through the loop body should match a new line with the same `machine` field (the `mach` variable is rigid), but with varying `line` fields (the `$line` variable is flexible).

The value of a flexible variable such as `$line` is the last value it got. If it did not have any yet, and depending on whether you wish to use `$line` as a number, a date, or a string, using `$line` will return 0, the epoch (i.e., the origin of time on your operating system; on Unix systems, this is `Jan 1 00:00:00 UTC 1970`), or the empty string.

Although the current value of any flexible variable is the last one it got through pattern-matching, logWeaver keeps the full history of its past values. (Except for some, see below.) This is for reporting reasons. Consider again the `repeated_modprobe_problems` rule of Figure 12. This is in file `t_modprobe1.c`; try it on the `syslog` file, typing:

```
logw -llinuxreadlog -st_modprobe1.c syslog
```

The results you should get are shown on Figure 13. As you can see by looking at the first line of the report, the reported values for the `$line` flexible variable are lines 1, 2, and 3. This allows you to know which were the matching lines.

The flexible variables whose name start with an underscore (_) are a bit special, in that logWeaver does not keep the history of their past values. For example, the `_$loop` does not have three values: only its final value 4 is reported in Figure 13.

What is this variable `_$loop` anyway? The idea is that since there is a label `loop`, logWeaver automatically maintains a *counter variable* `_$loop`, which is a special kind of flexible variable that counts how many times it has been through that label. In effect, writing:

```
while (loop: _$loop<=⟨n⟩) ...
```

says: loop n times through the `while` loop. More precisely, initially `_$loop` is not initialized, and is therefore converted to 0. Once control reaches the `loop` label, that is, just before logWeaver tries to evaluate `_$loop<=`$\langle n\rangle$, `_$loop` is incremented by one. At the next turn through the loop, `_$loop` will be incremented to two, then to three, then to four, and here the test `_$loop<=`$\langle n\rangle$ fails (when $n = 3$), so it has gone three times through the loop. (This is also why `_$loop` is printed as 4 in the end, since this is the value it has when the loop exits.)

```
repeated_modprobe_problems: mach=darkstar _$loop=4 $line=1,2,3
repeated_modprobe_problems: mach=darkstar _$loop=4 $line=4,4,4
repeated_modprobe_problems: mach=darkstar _$loop=4 $line=4,4,4
repeated_modprobe_problems: mach=darkstar _$loop=4 $line=4,5,6
repeated_modprobe_problems: mach=darkstar _$loop=4 $line=7,8,8
repeated_modprobe_problems: mach=darkstar _$loop=4 $line=8,8,8
repeated_modprobe_problems: mach=darkstar _$loop=4 $line=8,8,15
repeated_modprobe_problems: mach=darkstar _$loop=4 $line=16,17,18
repeated_modprobe_problems: mach=darkstar _$loop=4 $line=18,18,18
repeated_modprobe_problems: mach=darkstar _$loop=4 $line=18,18,18
repeated_modprobe_problems: mach=darkstar _$loop=4 $line=19,20,21
repeated_modprobe_problems: mach=darkstar _$loop=4 $line=22,22,22
repeated_modprobe_problems: mach=darkstar _$loop=4 $line=22,22,22
repeated_modprobe_problems: mach=darkstar _$loop=4 $line=22,24,25
repeated_modprobe_problems: mach=darkstar _$loop=4 $line=26,27,27
repeated_modprobe_problems: mach=darkstar _$loop=4 $line=27,27,27
repeated_modprobe_problems: mach=cecile _$loop=4 $line=127,128,376
repeated_modprobe_problems: mach=cecile _$loop=4 $line=377,379,380
repeated_modprobe_problems: mach=cecile _$loop=4 $line=381,382,383
repeated_modprobe_problems: mach=cecile _$loop=4 $line=407,408,747
repeated_modprobe_problems: mach=cecile _$loop=4 $line=748,748,1020
repeated_modprobe_problems: mach=cecile _$loop=4 $line=1021,1022,1240
repeated_modprobe_problems: mach=cecile _$loop=4 $line=1241,1241,1241
repeated_modprobe_problems: mach=cecile _$loop=4 $line=1242,1242,1323
repeated_modprobe_problems: mach=cecile _$loop=4 $line=1496,1501,1502
```

Figure 13: Results of `t_modprobe1.c` against `syslog`

You may wonder why the second line of Figure 13 reports three times the same line number: 4, 4, and 4. This is not a bug. In fact, if you look at the contents of the `syslog` file, you shall see that although line 3 is perfectly normal:

```
Jan 26 19:36:06 darkstar modprobe: Can't locate module block-major-8
```

line 4 is an abbreviation for seven repeats of this line:

```
Jan 26 19:36:09 darkstar last message repeated 7 times
```

The `linuxreadlog` preprocessor takes this as an indication that it should output seven times the contents of line 3, but all these repeats have the same line number, namely 4. This is why log$\mathcal{W}$eaver outputs the seemingly strange `$line=4,4,4` result of line 2 of Figure 13.

4.2 Counting, Accumulating Information

Until now, we have always let log$\mathcal{W}$eaver look for well-defined sequences of events. Once a matching sequence is found, it is reported. For example, count 42 events E, then report where these 42 were found. However, it is often useful to take a log, and report *how many* events E occurred.

Let us take the example of the log nwcnx.ulm.cg. This is an edited version of a log for Evidian's Netwall firewall, monitoring IP traffic. Ignoring the first three records, the rest is comprised of a lists of all IP packets over some network from October 28th, 1999, 09:03:24 to October 29th, 1999, 23:36:46 (line 31095), and from November 2nd, 1999, 03:20:22 to 03:54:16. Record format is made of several ⟨label=data⟩ fields, dates are in YYYYMMDDHHMMSS format, e.g. 19991029233627 is October 29th, 1999, 23:36:27. The proper preprocessor is nwreadlog.

```
probing_attack synchronized (source) anchored {
  while (loop: true) {
    .ACTION==0, .SRC source, .DST $dest, .line $line
                | ($dest =~ "^clio");
    || <<EOF>>; return | (_$loop>100);
  }
}
```

Figure 14: Detecting probing attacks on clio*

First look at the t_nw2.c file, whose contents is shown in Figure 14. The purpose of the probing_attack rule is to detect attacks where the same source address sends packets to all sorts of destination addresses (collected in $dest). Only attacks where the same source sends at least 100 packets should be reported, and we wish to only consider these packets sent to some machine whose name starts with clio. (Imagine clio_1, clio_r12, clio_a2, ..., are particularly sensitive machines.) On the other hand, we don't want to be warned as soon as logWeaver detects 100 messages from the same source to sensitive destinations. Rather, logWeaver should proceed to the end of file, still counting, and only then report the incriminated source, the list of all destinations $dest, and the list of matching lines $line.

This is what the rule of Figure 14 accomplishes. Try it by issuing the command:

```
logw -lnwreadlog -st_nw2.c -e nwcnx.ulm.cg
```

This will churn a while, then output exactly one attack, of the form:

```
probing_attack: source=frec _$loop=115
                $dest=clio_safe,clio_safe,...
                $line=476,510,18720,...
```

where we have used ellipses to abbreviate long series of (115) machine names or line numbers.

There are, again, several new features in this example. (We shall explain them in this section, and in Section 4.3.) In order of appearance, they are: the anchored keyword, the =~ operator, the || alternative, the <<EOF>> pattern, the return instruction, and the -e option to logw. Moreover, you may have noticed that logWeaver issued a few warning messages while reading the nwcnx.ulm.cg file:

```
nwreadlog: unknown field name USER, line 1.
nwreadlog: unknown field name USER, line 2.
nwreadlog: unknown field name USER, line 3.
nwreadlog: unknown field name SRC.PORHOST, line 31095.
```

Actually, these are warnings issued by the `nwreadlog` preprocessor, which could not parse lines 1, 2, 3, and 31095. In those cases, preprocessors merely ignore the faulty lines, and go on to the next. We don't care about the first three lines. Line 31095 is more interesting: it actually results from the loss of part of the log, which was truncated in the middle of line 31095 (in the middle of a `SRC.PORT=` field), then the log resumes at a new line (`HOST=` etc.): this results in line 31095 being garbled. log$\mathcal{W}$eaver, or rather its preprocessor, survives such format errors and ignores the malformed line 31095 altogether.

Let us now explain what all the other new constructions are for. The `=~` operator is Perl's regular expression matching operator: `$dest =~ "^clio"` means that we constrain `$dest` to match the regular expression `^clio`, that is, to start with the four letters "`clio`". As in Perl, there is an operator for not matching, `!~`. Since `!` is negation, `$dest !~ "^clio"` is equivalent to `!($dest =~ "^clio")`. Constraints may actually appeal to much more sophisticated regular expression matching; the `switch` expression may be used in more complex cases, e.g.:

```
... | (switch ($dest) {
          case "^clio": true
          case "net[_a-z]*([0-9]+)" { _net = "\\1" }: (_net!=$net)
          default: false
        })
```

This would match provided `$dest` either starts with "`clio`"; or it does not but it contains "`net`" followed by a few letters or underscores, then by a number, and this number (which we get back in the local variable `_net`) should not be the same as the one currently in the flexible variable `$net` (imagining we are using `$net` someplace else in the rule). Note the use of local variables, whose names start with an underscore, and denote temporary variables whose scope does not exceed the `case` branch where they are defined.

So the first line of the `while` loop tries to match records with the same `SRC` field and possibly different `DST` fields, provided the latter does not start with "`clio`". The `<<EOF>>` pattern, on the other hand, only matches end of files, and `||` is the committed choice operator: writing ⟨pattern-1⟩; ⟨action-1⟩ `||` ⟨pattern-2⟩; ⟨action-2⟩ means we are looking for some record that matches ⟨pattern-1⟩ (then we proceed to the sequence ⟨action-1⟩ of instructions), or that does not match ⟨pattern-1⟩ but does match ⟨pattern-2⟩ (then we proceed to ⟨action-2⟩). As a result, the body of the `while` loop in rule `probing_attack` does one of three things: it matches a record with the same `SRC` field each time, and a `DST` field that does not start with "`clio`", then it proceeds to the start of the loop again (⟨action-1⟩ is empty); or it matches the end of file, then it does `return | (_$loop>100)`; or it ignores the current record and waits for one that falls in one of the previous two cases.

The `return` instruction exits the rule and *accepts*: when a rule accepts, log$\mathcal{W}$eaver reports the values of variables. This used to be implicit in our previous examples: when control reaches the end of a rule, as in C, log$\mathcal{W}$eaver implicitly does a `return`. The `_$loop>100` is, as before, a constraint: `return` is only executed provided the value of the loop counter `_$loop` exceeds 100, that is, provided we went at least 101 times through the start of the `while` loop (labeled by `loop:`). And indeed we go once through `loop:` to enter the loop, then once more each time we match a record in the first line. So if we matched 100 times the first line of the body of the `while` loop, `_$loop` will be 101 at the time control reaches the `return` statement. If we reach the end of file, but `_$loop` is ≥ 100, then `return` is not executed, and as no other instruction can be executed, `probing_attack` just fails silently, not reporting any attack.

Finally, the `anchored` keyword is a kludge, resembling Prolog's cut. In principle, it is not really needed, and you may check this by removing the `anchored` keyword in `probing_attack` (see file `t_nw2_na.c` for instance). You may check that the presence or absence of `anchored` gives the same result by launching log𝓦eaver on `nwcnx1.ulm.cg` with or without the `anchored` keyword. However the version without `anchored` is slower and uses more memory, sometimes inacceptably more. The deep reasons are explained in Section 5.4.

Without going into details, the main thing to know is that, in general, you should not use `anchored`. Using `anchored` improperly may indeed contribute to masking real attacks, which will then go undetected (see Section 5.4 for an illustration.) Moreover, the increase in efficiency afforded by `anchored` is not always decisove. The `anchored` keyword is however sometimes useful, in cases of signatures that are meant to count or filter some sequence of events from the start to the end of the log, like `probing_attack` for example, as opposed to signatures that try to find matches in the middle of logs like `mouse_problems`. If log𝓦eaver is too slow, you may want to use the `anchored` option, and check that it yields the same result as without the `anchored` option, but faster. If this does not solve your efficiency problem, see Question 8.6.

4.3 Modes of Operation, End of Files, Streaming and Checkpointing

Note that using the `<<EOF>>` pattern is required to have log𝓦eaver report anything at all. If we had just written:

```
probing_attack synchronized (source) anchored {
  while (loop: true) {
    .ACTION==0, .SRC source, .DST $dest, .line $line
                | ($dest =~ "^clio");
  }
}
```

log𝓦eaver would never had had a chance of getting outside of the loop and returning. In fact, it would just collect matching lines, to no avail.

This leads us to explain how log𝓦eaver deals with logs, what end of files mean to log𝓦eaver, and why we used the `-e` option.

In its default mode of operation (without the `-e` option), log𝓦eaver takes a log and considers it as an unfinished sequence of events: once it reaches the end of file, log𝓦eaver saves its internal state and exits. This allows log𝓦eaver to be relaunched: once the log has grown again, relaunch `logw` with the `-r` (recovery) option, then log𝒲eaver will resume work exactly where it had stopped. (This should not be used when the log may grow while log𝒲eaver analyzes it, because log𝒲eaver may see the end of file while being in the middle of a record: then it will fail to parse this record and simply ignore it.) In fact, log𝒲eaver checkpoints its state not only when it reaches the end of file, but also when you interrupt it with control-C, and at regular intervals; see below.

In this default mode of operation, no log𝒲eaver rule may ever match an end of file (`<<EOF>>`). This is because an end of file is just taken by log𝒲eaver as an indication that there might be further records in the log later on, but they are not there yet.

If you use the `-e` option, then on encountering the end of file, log𝒲eaver will insert a fake empty record in the stream of records. This is the record that `<<EOF>>` matches. Providing the `-e` option therefore enables `<<EOF>>` to be matched at all. In the `probing_attack` rule, this allows log𝒲eaver to dump reports on accumulated information on current attacks. The `-e` option is particularly useful for offline auditing.

Another variation on the theme of end of files is provided by the `-b` option. If you provide it to `logw`, then, instead of checkpointing its state and returning at the end of the log, log𝒲eaver will wait (forever) for the log to fill in. (This implements *streaming*, where events are dealt with as they arrive, and correlations are reported as soon as they are complete.) Each time new records come in, log𝒲eaver will do a bit more work. Just giving the `-b` option is therefore ideally suited to online auditing tasks, using attacks such as those of Sections 3 or 4.1. Using both `-b` and `-e` is meaningless, and will work just as though only `-b` had been provided to `logw`.

Let us explain how log𝒲eaver deals with checkpointing in a more precise way. By default, log𝒲eaver checkpoints its internal state regularly, every 10 seconds. This can be changed by using the `-d` option; e.g., using `-d60` will force checkpoints to occur only every minute. The checkpoint file is named `logweave.ckp` by default, and is stored in the current directory from which you launched log𝒲eaver. The name of the checkpoint file can also be changed, using the `-c` option; e.g., using `-cmyfile` will checkpoint into `myfile.ckp`. log𝒲eaver also checkpoints when it is stopped by typing control-C or sending log𝒲eaver one of the signals `SIGINT`, `SIGQUIT`, `SIGHUP`, or `SIGALRM` (on Unix).

It also simultaneously checkpoints the current state of all active rules (see Question 8.5, Section 8) in a file called `logweave_ckp.c` (or `myfile_ckp.c` if you used the `-cmyfile` option).

To recover, relaunch log𝒲eaver from the same directory, with the `-r` option. If you used a non-standard checkpoint file name, like `myfile.ckp`, you have to give the `-cmyfile` option again, too. You also have to give the same `-s` option, with the same signature file as for your first call. If you don't remember what this signature file was, or if rules were fed to log𝒲eaver through, say, a named pipe (again, see Question 8.5, Section 8), then you may use `logweave_ckp.c` as signature file. However, you have to first change its name first, otherwise log𝒲eaver will read from it to get rules but also write into it when it checkpoints again, which will make things messy. For example, type:

```
mv logweave_ckp.c myspec.c
./logw -r -l./myreadlog -smyspec.c mylog
```

5 How It Really Works

It is sometimes necessary to understand some of the principles behind logWeaver in certain situations. In most cases, this is because you wrote a signature but you don't get the results you expected. Rather complete details are given in [RGL01], except the latter does not include any information on how `synchronized` or `anchored` is actually implemented. Moreover, there is a slight bug in the description of the way shortest runs, i.e., shortest matches, are discovered; this was also a bug in the algorithm of logWeaver in versions $\leq$2.7.

5.1 Basic Notions

The first thing you have to understand is that every signature compiles to an automaton. An automaton is a rather abstract graph-like structure that is composed of *states* and *transitions* between them. An example will explain the idea: let us look at how logWeaver handles the `t_mouse1.c` example. This can be done by asking logWeaver to produce a trace of everything it does, using the `-v` (verbose) option. Add the `-vt_mouse1.trace` to the command line to get a trace in file `t_mouse1.trace`, as follows:

```
logw -llinuxreadlog -st_mouse1.c -vt_mouse1.trace syslog
```

Now look at the `t_mouse1.trace` file. This starts with the following lines, which show logWeaver taking the `mouse_problems` signature into account.

```
* Adding rule:

mouse_problems {
  .comment "mouse", .machine mach, .program prog, .line line1,
     .date date1;
  .comment "mouse", .machine mach, .program prog, .line line2,
     .date date2;
}
  mouse_problems: 2 states, start state=0.
  - State 0:
     0. Match .comment "mouse", .machine mach, .program prog,
              .line line1, .date date1, then goto state 1.
  - State 1:
     0. Match .comment "mouse", .machine mach, .program prog,
              .line line2, .date date2, then accept.
```

logWeaver first reprints the text of your signature, then shows you the automaton it compiled. This automaton has two states. In state 0, it is waiting for a record whose `comment` field matches the regular expression `"mouse"`, and which will match-or-store its `machine` field into the `mach` variable, the `program` field into the `prog` variable, the `line` field into the `line1` variable, and the `date` field into the `date1` variable. Once it has found such a matching record, it will proceed to state 1. Once in state 1, it will wait for a record whose `comment` field also matches the regular expression `"mouse"`, whose `machine` and `program` fields are the same as for the first matching record (at line `line1`), and it will store its `line` field into `line2` and its `date` field into `date2`. Once these two records have been found, log𝒲eaver will *accept*, i.e., it will return from the signature and report the values of variables that were matched-or-stored during the match.

The rest of the `t_mouse1.trace` file shows how log𝒲eaver reads records and tests the `mouse_problems` signature against them by lauching threads that travel along the automaton above and eventually report signature matches when they reach an `accept` state.

So first log𝒲eaver reads the first record of file `syslog`, as shown in the trace file:

```
===============================================
Reading line 1.
0 active threads.
* Launching new thread monitoring rule mouse_problems,
    pid=0 [goto state 0, transition 0].
* pid=0, trying to match transition 0 of state 0: does not match:.
```

Then it launches a new thread to monitor possible matches of rule `mouse_problems` starting at line 1. Threads are numbered, much as processes under Unix, and the first thread that log𝒲eaver creates is numbered 0: its `pid` (process identifier) is 0. This thread proceeds to state 0 of rule `mouse_problems`. However, this thread expects a record whose `comment` field matches the regular expression `"mouse"`. This is not the case, as the first line is:

```
Jan 26 19:36:05 darkstar modprobe: Can't locate module block-major-22
```

So log𝒲eaver kills thread 0, and proceeds to next line. Similarly, log𝒲eaver will create threads at lines 1–33 and kill them right away.

The first line where something more interesting happens is line 33:

```
Jan 27 13:00:31 cecile insmod: Initialization of busmouse failed
```

Indeed, this is the first line where the `comment` field, which is everything after the last colon (`:`), matches `"mouse"`. Let us look at what log𝒲eaver does here:

```
===============================================
Reading line 33.
0 active threads.
* Launching new thread monitoring rule mouse_problems, pid=62
 [goto state 0, transition 0].
* pid=62, trying to match transition 0 of state 0: OK, values now:
  mach=cecile line1=33 prog=insmod date1=Sat Jan 27 13:00:31 2001
 [goto state 1, transition 0].
```

As before, it launches a new thread, with `pid` 62. This time, however, the match it was looking for succeeds. This is what the `* pid=62, trying to ...` line above says. As you may check, log𝒲eaver also recognizes that the `mach` variable should contain `cecile`, that the value of `line1` should be `33`, and so on. Now it proceeds through the transition from state 0 to state 1 (`[goto state 1, transition 0]`). State 1 will now wait for a record matching the second line of the `mouse_problems` rule.

Since logWeaver has now done all it could on record 33, it reads record 34, and proceeds:

```
==============================================
Reading line 34.
1 active threads.
* Launching new thread monitoring rule mouse_problems, pid=63
 [goto state 0, transition 0].
* pid=62, trying to match transition 0 of state 1: does not match:
   wait for next line.
* pid=63, trying to match transition 0 of state 0: does not match:.
```

Note that while logWeaver again launches a new thread (number 63) to monitor matches of `mouse_problems` (which, by the way, gets killed right away), it also continues to execute thread 62. The latter is waiting, at state 1, for a record matching the second line of the `mouse_problems` rule. Since this does not match, logWeaver decides to reschedule thread 62 at the subsequent record (`wait for next line`), leaving thread 62 at state 1.

Everything works this way until logWeaver reaches record 47:

```
Jan 27 13:05:33 cecile insmod: Initialization of busmouse failed
```

Then two things happen:

```
==============================================
Reading line 47.
1 active threads.
* Launching new thread monitoring rule mouse_problems, pid=76
 [goto state 0, transition 0].
* pid=62, trying to match transition 0 of state 1: OK, values now:
  mach=cecile line2=47 line1=33 prog=insmod
  date2=Sat Jan 27 13:05:33 2001 date1=Sat Jan 27 13:00:31 2001
     accept! [kill pid 62].
* pid=76, trying to match transition 0 of state 0: OK, values now:
  mach=cecile line1=47 prog=insmod date1=Sat Jan 27 13:05:33 2001
     [goto state 1, transition 0].
```

First, logWeaver again tries to make thread 62 advance. This times, this works: the transition from state 1 to the `accept` state matches, enriching the set of values that thread 62 has got. Now it knows `line2=47`, in particular. Since thread 62 accepts, i.e., returns, logWeaver reports all values, printing the following to `stdout`:

```
mouse_problems: mach=cecile line2=47 line1=33 prog=insmod
    date2=Sat Jan 27 13:05:33 2001 date1=Sat Jan 27 13:00:31 2001
```

Now that thread 62 has returned, it is reclaimed, and won't run on later records.

The second thing that logWeaver does on record 47 is launch a new thread, as usual. This is thread 76 here, and it proceeds along the transition from state 0 to state 1, preparing for a new match of rule `mouse_problems`.

5.2 Thread Management

As the previous section has shown, log𝒲eaver creates and runs threads in *parallel*. Less obvious is the fact that each thread is actually *non-deterministic* as well. Consider the example of Figure 15 (file `t_setuid1.c`).

```
setuid_attack synchronized(M,user) {
  .line $line, .HOST M, .PROG "^cp$", .SRC "sh$",
   .DST file "^/usr/spool/mail/(.*)$" { victim = "\\1" },
   .RULE user;
 again:
  .line $line, .HOST M, .PROG "^set.*id$", .SRC file, .RULE user;
  {
    .line $line, .HOST M, .PROG "^clr.*id$", .SRC file, .RULE user;
        goto again;
    | .line $line, .HOST M, .PROG "mail", .SRC file, .DST victim;
  }
}
```

Figure 15: The `setuid1` example

This is meant to detect a mail attack where the attacker first copies, using the Unix `cp` utility, a shell (detected as some file whose name ends in `sh`) to some file in the `/usr/spool/mail/` hierarchy, where unread mails are stored. Then the attacker sets the effective user id bit of the file, or its effective group id bit. Then either it clears the bit and sets the bit again, and so on, or it sends a mail to the owner `victim`. (This is a rough version of an old attack which allowed one to get a root shell from any user account.)

Run the `t_setuid1.c` file against the toy example `mailattack2`:

```
logw -l nwreadlog -s t_setuid1.c -v t_setuid1.trace mailattack2
```

The trace file `t_setuid1.trace` shows the resulting automaton:

```
setuid_attack: 4 states, start state=0.
  - State 0:
     0. Match .line $line, .HOST M, .PROG "^cp$", .SRC "sh$",
              .DST file "^/usr/spool/mail/(.*)$" { victim="\\1" },
              .RULE user, then goto state 1.
  - State 1 (label _$again):
     0. Match .line $line, .HOST M, .PROG "^set.*id$", .SRC file,
              .RULE user, then goto state 2.
  - State 2:
     0. Match .line $line, .HOST M, .PROG "mail", .SRC file,
              .DST victim, then accept.
     1. Match .line $line, .HOST M, .PROG "^clr.*id$", .SRC file,
              .RULE user, then goto state 3.
  - State 3:
     0. goto state 1.
```

Compared with the example of Section 5.1, this automaton shows two differences. The first one is that label `again` gives rise to the counter variable `_$again`, which will be incremented each time a thread running along this automaton comes to state 1. The second one is that there are now two transitions going out of state 2, one that leads to acceptance (transition 0), one that goes to state 3 (transition 1).

As in previous section, log𝓦eaver launches a thread (pid 0) to detect the mail attack:

```
==============================================
Reading line 1.
0 active threads.
* Launching new thread monitoring rule setuid_attack, pid=0
  [goto state 0, transition 0].
* pid=0, trying to match transition 0 of state 0: OK, values now:
  M=abaca file=/usr/spool/mail/root $line=1 user=Joe victim=root
  [goto state 1, transition 0].
```

This matches the first record, which indeed is a copy of a shell to `/usr/spool/mail/root`:

```
HOST=abaca PROG=cp SRC=/bin/bash DST=/usr/spool/mail/root RULE=Joe
```

Once thread 0 has matched this copying record, it proceeds to state 1:

```
==============================================
Reading line 2.
1 active threads.
* Launching new thread monitoring rule setuid_attack, pid=1
  [goto state 0, transition 0].
* pid=0, trying to match transition 0 of state 1: OK, values now:
  M=abaca _$again=1 file=/usr/spool/mail/root $line=1,2 user=Joe
  victim=root
  [goto state 2, transition 0] [goto state 2, transition 1].
* pid=1, trying to match transition 0 of state 0: does not match:.
```

Apart from the newly launched thread 1, which gets killed immediately, what happens here is thread 0 managing to match the transition from state 1 to 2, i.e., it finds the first `setuid` action. Now there are two transitions coming out of state 2, one which is waiting for the final `mail` action (transition 0), and one which is waiting for a `clruid` action (transition 1). Which one is the one to follow cannot be decided by log𝓦eaver at this point. So log𝓦eaver does not decide, and splits thread 0 into two copies, each with pid 0: one goes to state 2, waiting for transition 0 to fire, the other goes to state 2, waiting for transition 1 to fire.

When log𝓦eaver reaches record 3, there are now two threads with pid 0:

```
==============================================
Reading line 3.
2 active threads.
* Launching new thread monitoring rule setuid_attack, pid=2
  [goto state 0, transition 0].
* pid=0, trying to match transition 0 of state 2: does not match:
  wait for next line.
* pid=0, trying to match transition 1 of state 2: does not match:
  wait for next line.
* pid=2, trying to match transition 0 of state 0: OK, values now:
  M=iroko file=/usr/spool/mail/root $line=3 user=Joe victim=root
  [goto state 1, transition 0].
```

Furthermore, thread 2 is starting up, trying to find a match of rule `setuid_attack` starting from line 3. Meanwhile, both threads 0 are rescheduled to wait for record 4.

At record 4, which is as follows: log*W*eaver will notice that the second thread 0, which is waiting on transition 1 of state 2, can actually fire, whereas the first thread 0 cannot:

```
==============================================
Reading line 4.
3 active threads.
* Launching new thread monitoring rule setuid_attack, pid=3
  [goto state 0, transition 0].
* pid=0, trying to match transition 0 of state 2: does not match:
  wait for next line.
* pid=0, trying to match transition 1 of state 2: OK, values now:
  M=abaca _$again=1 file=/usr/spool/mail/root $line=1,2,4 user=Joe
  victim=root
  [goto state 1, transition 0].
* pid=2, trying to match transition 0 of state 1: does not match:
  wait for next line.
* pid=3, trying to match transition 0 of state 0: does not match:.
```

Notice that thread 3 is killed right away (there is no new instance of `setuid_attack` starting from line 4), while thread 2 waits at state 1 for some `setuid` action.

After some time, log*W*eaver reaches record 8, where there are now three threads 0:

```
==============================================
Reading line 8.
5 active threads.
* Launching new thread monitoring rule setuid_attack, pid=7
  [goto state 0, transition 0].
* pid=0, trying to match transition 0 of state 2: does not match:
  wait for next line.
* pid=0, trying to match transition 1 of state 2: does not match:
  wait for next line.
* pid=0, trying to match transition 0 of state 2: does not match:
  wait for next line.
* pid=2, trying to match transition 0 of state 1: OK, values now:
  M=iroko _$again=2 file=/usr/spool/mail/root $line=3,5,6,8 user=Joe
  victim=root
  [goto state 2, transition 0] [goto state 2, transition 1].
* pid=2, trying to match transition 0 of state 2: does not match:
  wait for next line.
* pid=7, trying to match transition 0 of state 0: does not match:.
```

Looking at what thread 0 investigates in more detail, we may discover that, starting from record 1, user `Joe` on machine `abaca` does a `setuid` at record 2, then a `clruid` at record 4, a `setuid` at record 7 (don't count `Joe`'s `setuid` at record 5, which occurs on `iroko` instead of `abaca`, and will be detected by thread 2 instead), and finally a `sendmail` action at record 9. At record 8, each thread 0 has seen both `setuid`'s and the `clruid` in the middle. The first thread 0 is trying to find an attack that does record 2's `setuid`, then record 4's `clruid`, then record 7's `setuid`; it is currently waiting for the `sendmail` action. The second one is also trying to find an attack going through records 2, 4 and 7, and is currently waiting for yet another `setuid` action. The third one has gone through record 2, and chose to wait directly for the final `sendmail` action. (Note that, in theory, there should be other instances of thread 0, namely at least one that didn't go through record 2 and record 4, but waited until record 7 to match a `setuid` action. log$\mathcal{W}$eaver actually recognizes that any such attack would be subsumed by the previous ones.)

On reaching record 9, which is:

```
HOST=abaca PROG=sendmail SRC=/usr/spool/mail/root DST=root
```

log$\mathcal{W}$eaver advances the first thread 0 past transition 0 of state 2 to the acceptance state:

```
===============================================
Reading line 9.
6 active threads.
* Launching new thread monitoring rule setuid_attack, pid=8
  [goto state 0, transition 0].
* pid=0, trying to match transition 0 of state 2: OK, values now:
  M=abaca _$again=2 file=/usr/spool/mail/root $line=1,2,4,7,9
  user=Joe victim=root accept!
  [kill pid 0].
* pid=0 [killed].
* pid=0 [killed].
* pid=2, trying to match transition 0 of state 2: does not match:
  wait for next line.
* pid=2, trying to match transition 1 of state 2: does not match:
  wait for next line.
* pid=2, trying to match transition 0 of state 2: does not match:
  wait for next line.
* pid=8, trying to match transition 0 of state 0: does not match:.
```

Having done so, it reports the result (see the `accept!` line above). More importantly, it *kills* thread 0, i.e., it removes every other thread 0: remember that all copies of thread 0 were actually different attempts to find an attack starting from record 1. Once one has been found, reporting the others would in general only increase information glut. Precisely, it is shown in [RGL01] that the match that log𝒲eaver reports this way is the *shortest* one in a precise sense, and it is argued that this shortest match is the one that carries the most useful information.

5.3 Synchronized Rules and Merging Pids

Let us return to the `t_mouse1.c` example of Section 5.1. We have seen in Section 3.2 that this was not completely satisfying. This is why we introduced the `synchronized` keyword in Section 3.4. Consider the `t_mouse2.c` example, which only differs from `t_mouse1.c` by the fact that rule `mouse_problems` is now synchronized on variables `mach` and `prog`. Write:

```
logw -l linuxreadlog -s t_mouse2.c -v t_mouse2.trace syslog
```

and look at the trace file `t_mouse2.trace`. Compared with the trace we analyzed in Section 5.1, the first difference occurs at record 47, on launching thread 76. Remember that thread 76 was the new successfully launched thread trying to match rule `mouse_problems` starting from record 47. Recall also that thread 62 was about to accept on reaching record 47. Here is what the new synchronized rule `mouse_problems` does here:

```
===============================================
Reading line 47.
1 active threads.
* Launching new thread monitoring rule mouse_problems, pid=76
  [goto state 0, transition 0].
* pid=62, trying to match transition 0 of state 1: OK, values now:
  mach=cecile line2=47 line1=33 prog=insmod
  date2=Sat Jan 27 13:05:33 2001 date1=Sat Jan 27 13:00:31 2001
    accept! [kill pid 62].
* pid=76, trying to match transition 0 of state 0:
  possibly conflicts with non-anchored synchronized declaration,
  changing pid to 62: [just killed].
```

The interesting difference is what the new thread 76 does. It matches transition 0 of state 0, match-or-storing `cecile` into `mach` and `insmod` into `prog`. Now it recognizes that another thread, namely thread 62, already holds a lock on rule `mouse_problems` with these values of the synchronized variables. (Although thread 62 just got killed, the lock is retained until all rules have been dealt with at record 47. Hence, technically, thread 62 still holds this lock.)

It might seem natural to think that, since thread 76 violates the synchronization condition, it should be killed right away. However, it might be that thread 62 never accepts, while thread 76 will. In this case, we would like thread 76 to report eventually. The trick to implement this is to *merge* pids 62 and 76: log𝒲eaver changes the given occurrence of thread 76 to have pid 62. This is another example of non-deterministic choice: log𝒲eaver cannot decide which of thread 62 or 76 will eventually succeed, so it merges their pids, leaving the decision to a later time when one of the threads accepts.

It turns out, though, that changing thread 76 to have pid 62 means renaming thread 76 to get the pid of a thread that just got killed, because thread 62 just accepted. The `[just killed]` comment signals this: log𝓦eaver detects that pid 62 was just killed, so thread 76 has to be killed, too.

By the way, although `setuid_attack` in `t_setuid1.c` (Figure 15) appears to do what we expect of it, this is not the case. Consider the following records (see file `mailattack`):

```
HOST=abaca PROG=cp SRC=/bin/bash DST=/usr/spool/mail/root RULE=Joe
HOST=abaca PROG=setuid SRC=/usr/spool/mail/root RULE=Joe
HOST=abaca PROG=clruid SRC=/usr/spool/mail/root RULE=Joe
HOST=abaca PROG=sendmail SRC=/usr/spool/mail/root DST=root
```

and run log𝓦eaver:

```
logw -l nwreadlog -s t_setuid1.c -e mailattack
```

This reports the following non-existent attack:

```
setuid_attack: M=abaca _$again=1 file=/usr/spool/mail/root $line=1,2,4
               user=Joe victim=root
```

You may need some time to convince yourself that this is actually a legal match of `setuid_attack`. This is due to the fact that the choice operator `|` we use in the last line of `setuid_attack` does not force log𝓦eaver to commit to the first alternative when it matches. Concretely, when log𝓦eaver has matched the `cp` of line 1, the `setuid` of line 2, and come to the `clruid` of line 3, it has the choice of either taking the first alternative and waiting for a later `setuid` (which will never occur), or of taking the second alternative and waiting for a later `sendmail`, which occurs at line 4.

In this example however we probably wish to forcibly take into account every `setuid` or `setuid` as soon as they arrive. In other words, we require log𝓦eaver to *commit* to the first alternative when it matches, by using the committed choice operator `||` instead of the ordinary choice operator `|`. This is file `t_setuid.c`, and indeed running log𝓦eaver with the latter on `mailattack` reports no attack.

5.4 Anchored Signatures

In Section 5.3 we saw that synchronized rules was implemented by merging pids. Returning to the `t_mouse2.c` example, we have seen that when the possible conflict between thread 62 and thread 76 is detected, log𝓦eaver cannot decide which of the two is going to succeed, and therefore cannot just kill thread 76 right away. Using the `anchored` keyword modifies this behavior so that log𝓦eaver indeed kills the conflicting thread 76 instead. This is faster than the pid merging trick of Section 5.3, but does not always leads to the same results. In particular it may lose some valid attacks.

Consider for example the `setuid_attack` example with the `anchored` keyword added. (This is file `t_setuid2.c`.) Now compare `t_setuid1.c` (without `anchored`) and `t_setuid2.c` (with `anchored`) on file `mailattack3`:

```
HOST=abaca PROG=cp SRC=/bin/bash DST=/usr/spool/mail/lp RULE=Joe
HOST=abaca PROG=cp SRC=/bin/bash DST=/usr/spool/mail/root RULE=Joe
HOST=abaca PROG=setuid SRC=/usr/spool/mail/root RULE=Joe
HOST=abaca PROG=clruid SRC=/usr/spool/mail/root RULE=Joe
HOST=abaca PROG=sendmail SRC=/usr/spool/mail/root DST=root
```

Running `t_setuid1.c` produces the following valid attack:

```
setuid_attack: M=abaca _$again=1 file=/usr/spool/mail/root
               $line=2,3,5 user=Joe victim=root
```

while running `t_setuid2.c` returns no attack whatsoever. This may be explained by looking at respective traces obtained with the `-v` option in each case. Briefly, what happens is as follows. In both cases, log$\mathcal{W}$eaver creates a thread at record 1 to look for an attack perpetrated by `Joe` on machine `abaca`, here against user `lp`. At record 2, log$\mathcal{W}$eaver wishes to create a new thread to look for an attack again perpetrated by `Joe` on `abaca`, this time against `root`. Note that the attack against `lp` is never complete, while the attack against `root` eventually completes at record 5. By using `anchored`, as in `t_setuid2.c`, you force log$\mathcal{W}$eaver to quit monitoring the second attack, on the false premises that it would overlap the first attack. But there is actually no first attack, so the only effect is to force log$\mathcal{W}$eaver to overlook the only valid attack. Looking at traces, here is what log$\mathcal{W}$eaver does at record 2 when we do not use `anchored`:

```
==============================================
Reading line 2.
1 active threads.
* Launching new thread monitoring rule setuid_attack, pid=1
  [goto state 0, transition 0].
* pid=0, trying to match transition 0 of state 1: does not match:
  wait for next line.
* pid=1, trying to match transition 0 of state 0:
  conflicts with anchored synchronized declaration:.
```

Thread 1, which is the one that would eventually succeed, is killed on the grounds that it conflicts with thread 0.

By not using `anchored`, as in `t_setuid1.c`, log$\mathcal{W}$eaver tries to detect both possible attacks; the fact that the `setuid_attack` rule is synchronized on `M` and `user`, and that both possible attacks have the same values for `M` and `user` respectively means that log$\mathcal{W}$eaver would only report one attack in case both succeeded. Looking at the trace, we would get the following at record 2:

```
==============================================
Reading line 2.
1 active threads.
```

```
* Launching new thread monitoring rule setuid_attack, pid=1
  [goto state 0, transition 0].
* pid=0, trying to match transition 0 of state 1: does not match:
  wait for next line.
* pid=1, trying to match transition 0 of state 0:
  possibly conflicts with non-anchored synchronized declaration,
  changing pid to 0:
  OK, values now:  M=abaca file=/usr/spool/mail/root $line=2
                   user=Joe victim=root
  [goto state 1, transition 0].
```

which allows logWeaver to proceed and eventually report a successful attack starting at record 2.

6 Signature Syntax

We describe here the syntax for the *signature files*, that is, those files that describe attack scenarios and usually have names of the form `t_`whatever`.c`. (The syntax of logs is whatever you wish, provided it has a preprocessor, see Section 7.)

We start with the lexical rules:

id An *identifier* is a letter or an underscore (_) followed by zero, one or more alphanumeric characters. *Letters* are A-Z, a-z. *Alphanumeric characters* are A-Z, a-z, 0-9, the dot (`.`), and the underscore (_.) Identifiers typically denote rule names or rigid variables.

field A *field name* is a dot followed by an identifier, without intervening spaces. For example, `.line` is a field name, while `line` is an identifier.

temp-var A *temp-var* is a sequence of at least one dollar sign (`$`), followed optionally by an identifier, without intervening space. For example, `$line`, `$$`, `$$date` are temp-vars. They typically denote flexible variables.

local-var A *local-var* is a sequence of at least one underscore (_) followed by a possibly empty sequence of dollar signs (`$`), followed by an identifier. They typically denote local variables in expressions of the form `(`$x_1 = e_1, \ldots, x_n = e_n, e$`)` which bind the local-vars x_1 to the value of e_1, ..., x_n to e_n, then evaluate e.

string A *string literal* is enclosed in two double quotes `"`, and is a sequence of characters. The backslash \ has special meaning: `\n` denotes a line-feed character (ASCII 10), `\t` a tab (ASCII 9), `\r` a carriage-return (ASCII 13), `\b` a bell (ASCII 7), `\f` a form-feed (ASCII 12). Additionally, a backslash followed by 1 to 3 octal digits denotes the corresponding ASCII code in octal; for example, `\012` is synonymous with `\n`. Otherwise, backslash followed by a character is just this character, so `\"` denotes the double-quote character.

int An *integer literal* is any non-empty sequence of digits 0-9, or `true` (denoting 1), or `false` (denoting 0). They denote integers.

Spaces, tabs, form-feeds, carriage-returns and line-feeds separate lexical tokens, and are otherwise ignored during parsing. Finally, the following are reserved keywords:

```
anchored  break   case    default       else    false
goto      if      int     len           new     reject
return    string  substr  synchronized  switch  time
true      while   &&      ||            ==      !=
=~        !~      <=      >=            =?      sum
<<EOF>>
```

‘,’	right-associative
‘?’ ‘:’	left-associative
‘||’	right-associative
‘&&’	right-associative
‘|’	right-associative
‘==’ ‘!=’	not associative
‘<’ ‘>’ ‘<=’ ‘>=’	not associative
‘+’ ‘-’	left-associative
‘*’ ‘/’ ‘%’	left-associative
‘=~’ ‘!~’ ‘=?’	not associative
‘!’ ‘#’ ‘new’	not associative

Figure 16: Operator precedence

The syntax of signature files is given by the following grammar, which recognizes rule commands (adding a rule, removing a rule) by the spec-commands production. We use the notation $[X]$ to denote any string matched by X or the empty string, $[X]^*$ to denote any sequence (possibly empty) of strings matched by X, and $[X]^+$ to denote any non-empty sequence of strings matched by X. The notation ‘abc’ denotes the corresponding string literal or keyword. Precedences are as given in Figure 16, where lower operators bind more strongly than higher operators.

spec-commands	::=	id [synchronized] [‘anchored’] block	add a new rule
	|	‘-’ id	remove a rule

synchronized ::= ‘synchronized’ ‘(’ var [‘,’ var]* ‘)’

var	::=	id	rigid variable
	|	temp-var	flexible variable

block	::=	‘{’ choice ‘}’	instruction block

choice	::=	[instruction]$^+$	sequence of instructions
	|	choice ‘|’ choice	non-deterministic choice
	|	choice ‘||’ choice	committed choice
	|	choice ‘+’ choice	conjunction (both arguments should be matched, in any order)

```
instruction   ::=  ‘return’ [constraint] ‘;’                  | accept
                |  ‘reject’ [constraint] ‘;’                  | silently fail
                |  ‘break’ [constraint] ‘;’                   | exit current scope
                |  ‘goto’ id [constraint] ‘;’                 | branch to label
                |  ‘if’ parenthesized-expr instruction        | conditional
                     ‘else’ instruction                       |
                |  ‘while’ while-expr block                   | while loop
                |  immediate-match [constraint] ‘;’           | record patterns
                |  block                                      | instruction block
                |  id ‘:’ instruction                         | label
                |  ‘;’                                        | no operation

immediate-match   ::=  field [id] string [regexp-bindings]          | string field matching
                    |  immediate-match ‘,’ immediate-match          | conjonction of patterns
                    |  field [id] [simple-constraint]*              | numeric field matching
                    |  ‘<<EOF>>’                                    | matching end-of-file
                    |  ‘(’ immediate-match ‘)’                      |

simple-constraint   ::=  ‘=’ expr    | equality constraint
                      |  ‘!=’ expr   | difference constraint
                      |  ‘<’ expr    | strict less than
                      |  ‘>’ expr    | strict greater than
                      |  ‘<=’ expr   | less than or equal to
                      |  ‘>=’ expr   | greater than or equal to

regexp-bindings   ::=  ‘{’ regexp-binding [‘,’ regexp-binding]* ‘}’
 regexp-binding   ::=  var ‘=’ string

constraint   ::=  ‘|’ parenthesized-expr

parenthesized-expr   ::=  ‘(’ local-expr ‘)’         | expression between parentheses
        local-expr   ::=  [binding ‘,’]* expr        | expression with value bindings
           binding   ::=  local-var ‘=’ expr         | binding a local variable to a value
```

expr	::=	simple-expr	atomic expression
	|	expr ‘&&’ expr	boolean conjunction
	|	expr ‘||’ expr	boolean disjunction
	|	expr ‘==’ expr	equality test
	|	expr ‘!=’ expr	difference test
	|	expr ‘=~’ string	regular expression matching
	|	expr ‘!~’ string	r.e. matching failure
	|	expr ‘=?’ history-expr	membership test
	|	expr ‘<’ expr	strict less than
	|	expr ‘<=’ expr	less than or equal to
	|	expr ‘>’ expr	strict greater than
	|	expr ‘>=’ expr	greater than or equal to
	|	expr ‘+’ expr	addition
	|	expr ‘-’ expr	subtraction
	|	expr ‘*’ expr	multiplication
	|	expr ‘/’ expr	integer division (quotient)
	|	expr ‘%’ expr	integer modulo (remainder)
	|	expr ‘?’ expr ‘:’ expr	conditional

simple-expr	::=	id	rigid variable
	|	local-var	local variable
	|	history-expr	flexible variable (or one of its previous values)
	|	string	string constant
	|	int	integer constant
	|	parenthesized-expr	expression between ‘(’ and ‘)’
	|	‘switch’ expr ‘{’ [expr-case]$^+$ [expr-default] ‘}’	regular expression matching
	|	‘if’ parenthesized-expr simple-expr [‘else’ simple-expr]	conditional
	|	‘!’ simple-expr	negation
	|	‘#’ history-expr	set cardinality
	|	‘-’ simple-expr	minus
	|	‘sum’ history-expr	sum of all values
	|	‘|’ history-expr ‘|’	list length
	|	‘len’ parenthesized-expr	string length
	|	‘‘’ simple-expr	previous value
	|	‘new’ history-expr	freshness test (succeeds if current value is not in set of old values)
	|	‘substr’ ‘(’ expr ‘,’ expr [‘,’ expr] ‘)’	substring
	|	‘int’ parenthesized-expr	conversion to integer
	|	‘string’ parenthesized-expr	conversion to string
	|	‘time’ parenthesized-expr	conversion to time

history-expr	::=	[‘‘’]* temp-var	flexible variable (or previous value: ‘$x is the previous value of $x, ‘‘$x is the value before that, etc.)

while-expr	::=	parenthesized-expr	expression between parentheses
	|	‘(’ id ‘:’ local-expr ‘)’	… possibly preceded by a label

expr-case	::=	‘case’ string [local-regexp-bindings] ‘:’ expr	alternative of switch (regular expression matching)
expr-default	::=	‘default’ ‘:’ expr	default alternative

local-regexp-bindings	::=	‘{’ local-regexp-binding [‘,’ local-regexp-binding]* ‘}’	
local-regexp-binding	::=	local-var ‘=’ string	binding a local variable

7 Writing Your Own Preprocessor

8 Frequently Asked Questions

8.1 I cannot manage to launch logWeaver, why?

Make sure logWeaver's executable `logw` is in your current path. Also make sure that the preprocessor you want to use (whose name typically ends in `readlog`) is also in your path; on Unix systems, check your `PATH` variable. For security reasons, your OS may insist on excluding your current directory from your current path. In that case, and if you are in the same directory as all executables, instead of typing:

```
logw -llinuxreadlog -st_mouse1.c syslog
```

type:

```
./logw -l./linuxreadlog -st_mouse1.c syslog
```

8.2 logWeaver complains about `.⟨field-name⟩: unknown field name`, what can I do?

This means that you ran it against a signature file that used ⟨field-name⟩ as a field name, but the log you are analyzing does not have such fields. Check the spelling of your field names. Call your preprocessor without arguments to get the list of all fields it produces (with their types, see Section 7). For example, calling `linuxreadlog` returns:

```
line:M;date:T;pid:I;machine:A;program:A;comment:A;
```

meaning that there will be an integer field `line` that can only increase through time (type `T`), a date field `date` that can only increase too (type `T`; type `D` is the type of dates that may vary freely), an integer field `pid` (type `I`), and three text fields `machine`, `program`, and `comment` (type `A`, like ASCII).

It might also be the case that you instructed logWeaver to use a preprocessor, through the `-l` option, that it could not find. Since it is the responsibility of the preprocessor to tell logWeaver which are the fields it will provide to logWeaver, with their types, failing to launch a preprocessor means that no field name will be recognized.

8.3 I have written a rule, but it never detects anything, although it really ought to, what is the matter?

There are several possible reasons to this.

- Perhaps your rule has never been taken into account by logWeaver: have you checked whether logWeaver complained about your rule? If there was a syntax error or a semantic error in your rule, logWeaver reports the error, and ignores the rule.

- Perhaps your rule was correctly formed, but it was still not taken into account: make sure that your rule definition is followed by a control-L character (ASCII 12 = 0xc). (You may have noticed this if you have read the example files `t_mouse1.c`, `t_auth1.c`, and so on, that were mentioned in this manual.) You may check that this is what happen by generating a trace file with the `-v` option (see Question 8.7). Let us take as example the `mouse_problems` of file `t_mouse1.c`. Then the trace file should show a few lines looking like:

  ```
  * Adding rule:

  mouse_problems synchronized(mach,prog) {
    .comment "mouse", .machine mach, .program prog,
                      .line line1, .date date1;
    .comment "mouse", .machine mach, .program prog,
                      .line line2, .date date2;
  }
    mouse_problems: 2 states, start state=0.
    - State 0:
       0. Match .comment "mouse", .machine mach, .program prog,
                .line line1, .date date1, then goto state 1.
    - State 1:
       0. Match .comment "mouse", .machine mach, .program prog,
                .line line2, .date date2, then accept.
  ```

  This says that log$\mathcal{W}$eaver correctly parsed and accepted your rule definition, and log$\mathcal{W}$eaver prints its text followed by a description of the automaton it has compiled it to.

  As said above, a typical cause for log$\mathcal{W}$eaver not taking a rule into account is failing to write control-L after the rule. The control-L character tells log$\mathcal{W}$eaver that everything preceding it is a sequence of commands: rule definitions tell log$\mathcal{W}$eaver to add the corresponding rules, and entries of the form $-\langle$rule-name$\rangle$ tell log$\mathcal{W}$eaver to remove rule $\langle$rule-name$\rangle$ from its set of active rules.

  In fact, the signature file is really a sequence of commands telling log$\mathcal{W}$eaver to add or remove rules. You may always append new commands at the end of the signature file, but they will only be taken into account when you type control-L. This is admittedly a kludge, and we plan to be able to dispense with this control-L trick in future versions of log$\mathcal{W}$eaver.

  The purpose of all this is so that you can add or remove rules while log$\mathcal{W}$eaver is running. log$\mathcal{W}$eaver starts matching newly added rules from the line it is currently reading in the log. Removed rules won't be triggered longer, but if log$\mathcal{W}$eaver was in the process of detecting a series of lines matching a rule and this rule gets removed, it will still try to complete the match.

- Or perhaps your rule was taken into account, but it just never matched anything. Again, this can be investigated through the use of the `-v` option: see Question 8.7.

8.4 My machine crashed, or the `logw` process got killed, while it was monitoring some real-time stream of events, how do I recover from this?

Don't panic. log$\mathcal{W}$eaver checkpoints its internal state at regular points in time. You may relaunch log$\mathcal{W}$eaver with the `-r` option to get it running again, starting from where it stopped. See Section 4.3 for detailed information.

8.5 Is it possible to add or remove rules from the signature file and have log$\mathcal{W}$eaver take the modifications into account?

Yes. There are actually two ways to do this. The first is to stop log$\mathcal{W}$eaver (e.g., type control-C), modify the signature file, and relaunch it with `-r`. Using the checkpointing mechanism (the `-r` option, see Section 4.3) allows you to resume log auditing without losing any information, and without having to reanalyze all previous lines of the log.

The second possibility is to append new commands at the end of the signature file without stopping log$\mathcal{W}$eaver. In doing this, don't forget to add a control-L at the end: see Question 8.3.

8.6 log$\mathcal{W}$eaver uses a lot of memory. What should I do?

There is no best answer to this question. Detecting complex patterns scattered across lines can sometimes require quite a lot of time and memory. What costs most is complex patterns that match lines very far apart, because log$\mathcal{W}$eaver will have to execute many long-lived threads, which all consume resources. In fact, complex patterns that might match lines arbitrarily far apart but actually never match are monitored by threads that never terminate, and clog the system.

Sometimes this explosive behavior is unavoidable. Typically, if you try to find lines matching some record pattern A, followed by lines matching some other pattern B, but B might match 100, 1000, or one million lines later, or never. Then log$\mathcal{W}$eaver will launch a thread to try to match A at each line, and for each line that matched A, the corresponding thread will wait until it finds a line where B matches. If A matched 5000 times and B never matches, log$\mathcal{W}$eaver will eventually run 5000 useless threads in parallel. Using constraints on dates, as in Section 3.5, can help relieve log$\mathcal{W}$eaver by killing threads that have waited too long. You may also use constraints on line numbers to kill threads that have waited for too many lines already. If this does not help, see Question 8.9, or (for experts) use the `-v` option: see Question 8.7.

Another case where log$\mathcal{W}$eaver will use a lot of memory is if you used the ordinary choice operator `|`. Whenever you write something like $A \,|\, B$ in a signature, log$\mathcal{W}$eaver will spawn two threads (see Section 5 on threads), one waiting for A, the other for B. If this is inside a loop, then the number of threads will double at each turn of the loop, which will quickly lead to memory saturation and drastic performance reduction. In this case, you probably in fact want to say $A \,||\, B$, where `||` is the committed choice operator. This is not equivalent: while $A \,|\, B$ means "wait for either A or B to happen", A || B means "if A holds now, then choose A, otherwise try to match B".

8.7 I have written a rule, but it never matches, or it matches unexpected series of lines, is there a bug in log$\mathcal{W}$eaver?

While it is certainly true that log$\mathcal{W}$eaver, like any complex piece of software, is likely to contain bugs, it might be that you wrote something that actually means something else than what you meant. For example, in versions 2.5 and before of log$\mathcal{W}$eaver, you could write an instruction of the form `if` (⟨condition⟩) ⟨instruction⟩ without an `else` clause. This had the effect of going to ⟨instruction⟩ if ⟨condition⟩ held, and blocking otherwise. But most people read it as though the failure of ⟨condition⟩ meant that execution would proceed to the next line.

While strange behaviors are puzzling, the `-v` (verbose) option to log$\mathcal{W}$eaver can be used to understand what is really going on. For example, add the `-vmytrace` option to the `logw` command line. This will produce a `mytrace` file containing a mostly human-readable explanation of what log$\mathcal{W}$eaver actually tried to do. You may then use this information to adapt your rules accordingly. To understand the `mytrace` file, you may have to understand how log$\mathcal{W}$eaver works, though. See Section 5.

Now if you wrote a rule that never matched, look into your `mytrace` file. It may be that the rule was never triggered, or that it was triggered but cannot reach an accepting state, or that it was triggered but waits for an end-of-file indicator that will never occur, typically. The most puzzling cases are the latter two, and happen typically with rules that try to count or accumulate information along the whole log. (It may be a good idea to re-read Section 4.2 to understand how you should write your rules.) For example, writing a rule with a body like:

```
while (loop: true) {
     .field1 = "blah", ...;
  || .field1 = "foo", ...;
  || ...
}
```

will typically never exit the loop. On encountering the end of file, it won't automatically exit the loop: the `true` condition in the `while` directive does not say "while we are not at the end of file", rather "while true is true", that is, forever. So you typically have to add a line matching the end of file explicitly, say:

```
  || <<EOF>>; return;
```

meaning that end of files should be matched and then the rule should match.

Even though you may have put this `<<EOF>>` pattern already, your rule may still fail to match. The typical reason is that, by default, log$\mathcal{W}$eaver does not understand end of files: to log$\mathcal{W}$eaver, a log does not end, and is always in the process of being filled in. To inform log$\mathcal{W}$eaver that your log won't grow, or at least that you would like your rules to be aware of the existence of the end-of-file, use the `-e` option: on encountering the end of file, log$\mathcal{W}$eaver will then insert a dummy empty record that only matches the `<<EOF>>` pattern, therefore allowing the `|| <<EOF>>; return;` construction to fire.

Another example of unexpected behaviour is explained at the end of Section 5.3, where the `t_setuid1.c` signature file detects false attacks because of the use of `|` instead of `||`. There seems to be a rule that you are in general safer in using the committed choice operator rather than the ordinary choice operator `|`. Also, the latter tends to make logWeaver consume more memory, see Question 8.6.

8.8 Why is logWeaver complaining about ifs without elses?

See Question 8.7.

8.9 I have written a constraint on dates as in Section 3.5 but logWeaver keeps gobbling up memory. What is happening?

Basically, reread carefully Section 3.5 and understand why you should write your rule in the style of Figure 11 instead of Figure 9. If this does not work, use the `-v` option: see Question 8.7.

8.10 How do I interface logWeaver with `logrotate` or other log rotation mechanisms?

Good question. You almost can now, but I still have to work on it.

8.11 Is it possible to use a variable whose value will not be reported?

Yes: use a name starting with two dollar signs, e.g., `$$date`, `$$$date`, etc. Note that such variables are always flexible. You may consult, e.g., `t_nw8.c` for an example.

8.12 Can I have the values of a flexible variable printed without duplications?

Regular flexible variables (e.g., `$dest`) print as the *sequence* of the values they took during a match. If you wish to print them as a *set*, with every value listed exactly once, give your flexible variable a name starting in `$_`. Consult `t_nw8.c` for an example.

8.13 Some line numbers repeat, or two instances of the same `synchronized` rule overlap, what is the matter?

It may be the case that there are two instances of the same `synchronized` rule that match records with the same line number. In this case, you will get the feeling that they overlap. This might not be the case, though, since in some log formats, several records may share the same line number. See the example of Figure 13, which exhibits this behavior, and read Section 4.

8.14 Do I need spaces after command-line options, e.g., do I write `-l./nwreadlog` or `-l ./nwreadlog`?

There is no difference between the two, starting from version 2.8. You were only required to leave the space off in versions 2.7 and earlier of log$\mathcal{W}$eaver.

References

[Mou97] Abdelaziz Mounji. *Languages And Tools for Rule-Based Distributed Intrusion Detection*. PhD thesis, Facults Universitaires Notre-Dame de la Paix, Namur, Belgium, September 1997.

[Pax98] V. Paxon. BRO: A system for detecting network intruders in real-time. In *Proceedings of the 7th USENIX Security Symposium*, 1998.

[RGL01] Muriel Roger and Jean Goubault-Larrecq. Log auditing through model-checking. In *IEEE Computer Security Foundations Workshop XIV (CSFW'14)*, June 2001.

[Roe99] Martin Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the 13th Conference on Systems Administration (LISA'99)*, pages 229–238. USENIX Associations, November 1999.

[Spe86] Henry Spencer. regexp package. Available at `http://arglist.com/regex/`, 1986.

Index