

lightgrep: A Multipattern Regular Expression Search Tool for Digital Forensics

Jon Stewart
Lightbox Technologies, Inc
Arlington, VA, USA
jon@lightboxtechnologies.com

Joel Uckelman
Institute for Logic, Language and Computation
University of Amsterdam
Amsterdam, The Netherlands
j.d.uckelman@uva.nl

1. INTRODUCTION

Online keyword searching, commonly called “grep”, has several important applications in computer forensics. Investigators can search for text-based documents and fragments with potential relevance to a matter, identify structured artifacts such as Yahoo! Messenger chat logs and MFT entries, and recover deleted files using a header-footer search. Examiners often want to search for hundreds, sometimes thousands, of keywords and patterns.

We describe `lightgrep`, a regular expression engine for digital forensics, which can efficiently search huge data streams for multiple patterns. `lightgrep` addresses several shortcomings of existing tools used in digital forensics, by taking advantage of some recent (and some not-so-recent) developments in automata theory. We state the characteristics of regular expression searching in forensics, review current approaches, and then discuss `lightgrep`'s implementation in detail, based on direct simulation of nondeterministic finite automata. A number of practical optimizations will be given.

2. STATEMENT OF THE PROBLEM

Traditional regular expression libraries have focused on searching line-oriented text files with a single regular expression. The requirements for digital investigations are different, and present several challenges:

- Multipattern, with matches related to patterns
- Graceful degradation as number of patterns increases
- Support for large binary streams and long matches
- Multiple encodings (UTF-8, UTF-16, code pages)

A multipattern engine must identify all occurrences of the given patterns in a byte stream, even if some matches overlap. For example, an investigator may run a search for the keywords `<html>.*</html>` and `osama.{0,10}bin.{0,10}laden.`

```
<html>
<head>
  <title>Welcome!</title>
</head>
<body>
  <p>Welcome to our friendly homepage on the
    internet!</p>

  <p>Send us <a href="mailto:osama@binladen.org">
    email!</a></p>
</body>
</html>
```

Figure 1: Some interesting HTML

A correct multipattern search implementation would report a hit for both keywords on the fragment in Figure 1.

The search algorithm must degrade gracefully as the number of patterns increases. An investigator should be confident that doubling the number of keywords will at most double the search time. Worst-case guarantees are important in forensics as they afford investigators greater control over case management.

It must be efficient to search byte streams many times larger than main memory and to track pattern matches hundreds of megabytes in size. In particular, the search algorithm used must mesh nicely with I/O concerns.

Finally, since forensic data is by nature unstructured, it is necessary to search for occurrences of the same patterns in different encodings.

3. RE2

Recently, Google has released the regular expression engine used for Google Code Search, RE2 [3], and its author, Russ Cox, has described it in depth [2]. RE2 works by converting the specified pattern to an NFA, or a DFA if memory permits. RE2 represents the automata as specialized machine instructions and treats the current state as a thread operating at a certain instruction in the program; NFA searching requires the execution of multiple threads in lock-step examination of the text, character-by-character [6]. This is consistent with the $O(nm)$ complexity of NFA simulation, where n represents the size of the text and m the number of states in the automata [1].

Additionally, the start and end of matches on captured groups are tagged in the automata, as described by Laurikari [5]. Each thread maintains a small array recording the starting and ending offsets of submatches, indexed by the transition tags. In this manner, RE2 is able to record submatch boundaries without backtracking.

literal c	If current character is c , increment state and suspend the current thread; kill the current thread otherwise
fork n	Create a new thread in state n at the current offset, and increment state
jump n	Goto state n
match n	Record a match for pattern n ending at current offset n and increment state
halt	Kill the current thread, reporting a match if any

Figure 2: Basic bytecode instructions

4. LIGHTGREP

Inspired by RE2, we have created `lightgrep`, a regular expression search tool for digital forensics. Simulating an NFA directly, it can search for thousands of patterns in a single pass of data. All occurrences of all patterns are correctly reported, and `lightgrep` never needs to refer backwards in the data. The number of patterns is limited only by the amount of RAM and it will cheerfully report that `.*` matches the entire byte stream, regardless of size.

The key to correct multipattern searching is applying tagged transitions to the pattern matches. Instead of having an array of submatch positions, each thread has scalar values for the starting offset of the match, the ending offset, and the value of the last tagged transition. We tag transitions to match states with the corresponding index numbers of the patterns. While the worst-case complexity of NFA search is $O(nm)$, we have applied several practical optimizations to provide reasonable performance with large automata.

4.1 Implementation

Rather than using an NFA directly, we compile patterns into bytecode, using the instructions in Figure 2. A program in our bytecode language consists of a (numbered) sequence of bytecode instructions. Then, given a list of patterns to match and a stream of input, a bytecode program is executed by the bytecode interpreter to produce a list of matches.

Each thread is a tuple $\langle s, i, j, k \rangle$, where s is the current state, i is the start (inclusive) of the match, j is the end (exclusive) of the match, and k is the index of the matched pattern. When a thread is created, it is initialized to $\langle 0, p, \emptyset, \emptyset \rangle$, where p is the current position in the stream.

To make this concrete, we give an example. Suppose we have the stream `qabcabdbd` which we want to search for the patterns `a (bd) +` and `abc`. The NFA corresponding to this pattern list can be seen in Figure 3 and the bytecode produced for these patterns appears in Figure 4. The run of this bytecode reports a match for `abc` at [1, 4) and a match for `a (bd) +` at [4, 9).

4.2 Multiple Encodings

Many regular expression libraries that support Unicode rely on the assumption that the data is stored in a single encoding. This does not hold in forensics; when searching unstructured data, encodings may change capriciously. `lightgrep` is explicitly byte-oriented. To search for alternate encodings of a pattern, its binary representations are generated as separate patterns in the NFA. Matches can then be resolved back to the user-specified term and appropriate encoding via a table. `lightgrep` currently can search for ASCII-specified patterns as ASCII and as UTF-16. Full support for various encodings is under active development, using the open source ICU library [4].

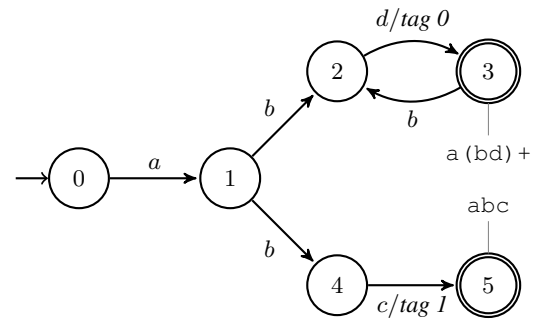


Figure 3: NFA matching `a (bd) +` and `abc`

```

0  literal 'a'
1  fork 6
2  literal 'b'
3  literal 'd'
4  match 0
5  jump 2
6  literal 'b'
7  literal 'c'
8  match 1
9  halt

```

Figure 4: Bytecode matching `a (bd) +` and `abc`

5. CONCLUSION

Tagged NFAs can easily be applied to the multipattern problem and several practical optimizations can keep the observed performance competitive as the size of the automata increases. `lightgrep` provides investigators with a sorely-needed capability, allowing evidence to be searched for large keyword sets. The $O(nm)$ running time of NFA search establishes an upper bound for forensic keyword searches and, with a combination of implementation optimizations, observed performance far exceeds that of the nearest substitute tool, EnCase.

References

- [1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, & Tools*. Pearson, second edition, 2007.
- [2] R. Cox. Regular expression matching: the virtual machine approach, December 2009. <http://swtch.com/~rsc/regexp/regexp2.html>.
- [3] R. Cox. RE2: an efficient, principled regular expression library, May 2010. <http://code.google.com/p/re2>.
- [4] IBM. International Components for Unicode (ICU), May 2010. <http://icu-project.org>.
- [5] V. Laurikari. NFAs with tagged transitions, their conversion to deterministic automata and applications to regular expressions. In *Proceedings of the 7th International Symposium on String Processing and Information Retrieval*, pages 181–187. IEEE, 2000.
- [6] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.