Searching Massive Data Streams Using Multipattern Regular Expressions

Jon Stewart, Joel Uckelman

Chapter 4

SEARCHING MASSIVE DATA STREAMS USING MULTIPATTERN REGULAR EXPRESSIONS

Jon Stewart and Joel Uckelman

Abstract This paper describes the design and implementation of `lightgrep`, a multipattern regular expression search tool that efficiently searches massive data streams. `lightgrep` addresses several shortcomings of existing digital forensic tools by taking advantage of recent developments in automata theory. The tool directly simulates a nondeterministic finite automaton, and incorporates a number of practical optimizations related to searching with large pattern sets.

Keywords: Pattern matching, regular expressions, finite automata

1. Introduction

The regular-expression-based keyword search tool `grep` has several important applications in digital forensics. It can be used to search text-based documents for text fragments of interest, identify structured artifacts such as Yahoo! Messenger chat logs and MFT entries, and recover deleted files using header-footer searches.

However, while digital forensic investigations often involve searching for hundreds or thousands of keywords and patterns, current regular expression search tools focus on searching line-oriented text files with a single regular expression. As such, the requirements for digital forensic investigations include multipattern searches with matches labeled by pattern, graceful performance degradation as the number of patterns increases, support for large binary streams and long matches, and multiple encodings such as UTF-8, UTF-16 and legacy code pages.

A multipattern engine must identify all the occurrences of patterns in a byte stream, even if some matches overlap. The patterns must have

```
<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. HTML code fragment.

full use of the regular expression syntax, and must not be limited to fixed strings. For example, when carving an HTML document, a digital forensic examiner might run a search for the keywords `<html>.*</html>` and `osama.{0,10}bin.{0,10}laden`. A correct multipattern search implementation would report a hit for both keywords in the HTML code fragment in Figure 1.

The search algorithm must degrade gracefully as the number of patterns increases, so that it is always faster to search for all the patterns in a single pass of the data than to perform multiple search passes. Most digital forensic examiners desire competitive and predictable performance. Worst-case guarantees are important as they afford digital forensic examiners greater control over case management.

It is also necessary to efficiently search byte streams many times larger than main memory and to track pattern matches that are hundreds of megabytes long. In particular, the search algorithm used must mesh nicely with input/output concerns.

Finally, because digital forensic data is unstructured, it is often necessary to search for occurrences of the same patterns in different encodings. This is especially important when searching for text in foreign languages, where numerous encodings exist and it is unrealistic to demand that a digital forensic examiner master all the regional encodings.

This paper discusses several regular language text search implementations and describes the implementation and key features of `lightgrep`, a simple regular expression engine for digital forensics inspired by Google's RE2 engine [5]. Experimental results are also presented to demonstrate the advantages of `lightgrep`.

2. Finite Automata

A finite automaton consists of a set of states, one of which is the initial state, and some of which may be terminal states. Pairs of states may

have arrows corresponding to transitions from one state to the other. Each transition has a label corresponding to a character in the input alphabet.

A finite automaton reads characters from an input string. The current state of the finite automaton changes as it follows transitions with labels that match the characters read from the input string. If a terminal state is reached, the finite automaton has matched the input. If a non-terminal state is reached that has no transition for the current character, the finite automaton has not matched the input. A finite automaton is a deterministic finite automaton (DFA) if no state has two or more outgoing transitions with the same label; otherwise, it is a nondeterministic finite automaton (NFA). Every NFA is equivalent to some regular expression, and vice versa [15].

3. Pattern Searching Approaches

Pattern searching is not a new problem, and several tools exist that address the problem. Of these, some are forensics-specific tools (e.g., FTK [1] and EnCase [7]) while others are regular expression search tools used in general computing (e.g., `grep`, `lex` and RE2 [4]).

Resourceful digital forensic examiners often use the Unix `strings` command to extract ASCII strings from binary files and then perform a search by piping the output to `grep`. This works for a quick search, but using `strings` filters out unprintable characters and segments the data. As a result, searches for non-ASCII text, as well as for many binary artifacts, are not possible, and examiners are limited to fixed strings when searching for multiple patterns. Note, however, that GNU `grep` does offer good performance for single patterns.

AccessData's FTK [1] uses the open source Boost Regex library [12]. Boost Regex offers a rich feature set and competitive performance. It uses a backtracking algorithm, which can lead to poor performance on certain expressions (consequently, searches are terminated when the run time becomes quadratic). Like most regular expression libraries, however, Boost Regex does not support multipattern searches.

Guidance Software's EnCase [7] supports multipattern searches for regular expressions with a limited syntax, and also allows users to specify which encodings to consider for each keyword. Performance is acceptable with fixed-string patterns and it degrades gracefully as the number of patterns increases. However, the search time increases significantly as regular expression operators and character classes are used in more patterns. Repetition is limited to a maximum of 255 bytes and EnCase is unable to parse some complex patterns correctly. Finally, while the

search algorithm used by EnCase is proprietary, its results are consistent with a backtracking approach, wherein an increasing degree of alteration in a multipattern automaton leads to performance loss.

The Unix `lex` utility can be used to search for multiple regular expressions in a limited fashion. `lex` compiles all patterns into a single DFA and reports on the longest match. The use of a DFA leads to good performance, but it also means that `lex` cannot match overlapping occurrences of different patterns. Most problematically, `lex` may backtrack unless all the patterns are specified in a deterministic form, rendering its use with non-trivial or inexpertly constructed patterns infeasible. Additionally, because `lex` generates a C program to perform a search, examiners have the burden of compiling the generated program and maintaining a well-configured Unix environment. Nevertheless, good results can be obtained with `lex` when it is used to extract a fixed set of common, mutually-exclusive patterns [8].

Google's RE2 [4] is a regular expression engine that is used by Google Code Search. RE2 implements much, but not all, of the POSIX and Perl regular expression syntax, guarantees linear performance with respect to pattern and text size, and allows for efficient submatches of captured groups. The RE2 syntax is strictly limited to patterns that can be searched without backtracking to avoid the evaluation of expensive patterns that might be used in denial-of-service attacks.

RE2 converts the specified pattern to an NFA and then generates DFA states as needed, caching the DFA states for repeated use [14]. RE2 represents the NFA in terms of specialized machine instructions and treats the current state as a thread operating on a certain instruction in the program. NFA searches require the execution of multiple threads in a lock-step, character-by-character examination of the text; DFA searches utilize only a single thread. This is consistent with the $O(nm)$ and $O(n)$ complexity of NFA and DFA simulations, respectively, where n is the size of the text and m is the number of states in the automaton.

The starting and ending points of matches on captured groups are tagged in the automata [10]. 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.

As with `lex`, RE2 can approximate a multipattern search by combining patterns into alternating subpatterns of the form $(t_1) \mid (t_2) \mid \ldots \mid (t_n)$. However, because the submatch array size is $O(m)$, performance begins to degrade substantially due to the copying of the thread state as the number of patterns increases beyond 2,000 to 4,000. RE2 has other properties that limit its use in digital forensics applications. When us-

`literal` c	If the current character is c, increment the instruction and suspend the current thread. Otherwise, kill the current thread.
`fork` n	Create a new thread at instruction n at the current offset and increment the instruction.
`jump` n	Go to instruction n.
`match` n	Record a match for pattern n ending at the current offset and increment the instruction.
`halt`	Kill the current thread and report a match if one exists.

Figure 2. Basic bytecode instructions.

ing a DFA search, RE2 generates a reverse DFA that it runs backwards on the text when a match occurs in order to find the starting point of the match. This is clearly inefficient for the very long matches required during file carving. RE2 also assumes that the text is small enough to fit in main memory, and it has no facility for continuing searches from one segment of text to another.

4. lightgrep

The `lightgrep` tool, which is inspired by the design of RE2, is a regular expression search tool for digital forensics. It directly simulates an NFA and can search for thousands of patterns in a single pass without exhibiting pathological performance problems. All occurrences of all patterns are reported without having to refer backwards in the data, allowing for a streaming input/output model. The number of patterns is limited only by the amount of system RAM and matches are reported regardless of their size.

Correct multipattern searching is achieved by the application of tagged transitions to pattern matches, not to submatches. Instead of using an array of submatch positions, each state has scalar values for the starting offset of the match, ending offset and value of the last tagged transition. Transitions are tagged to match states with the corresponding index numbers of the patterns. While the worst-case complexity of NFA search is $O(nm)$, several practical optimizations are incorporated in `lightgrep` to obtain reasonable performance with large automata.

4.1 Implementation

Rather than using an NFA directly, `lightgrep` compiles patterns into a bytecode program using the instructions in Figure 2. Given a list of patterns to match and a stream of input, the bytecode program is then

```
 1: for p := 0 to end of stream do
 2:     create a new thread ⟨0, p, ∅, ∅⟩
 3:     for all live threads t := ⟨s, i, j, k⟩ do
 4:         repeat
 5:             execute instruction s
 6:         until t dies or is suspended
 7:     end for
 8: end for
 9: for all live threads t := ⟨s, i, j, k⟩ do
10:     repeat
11:         execute instruction s
12:     until t dies
13: end for
```

Figure 3. Bytecode interpreter.

executed by the bytecode interpreter (Figure 3) to produce a list of matches.

Each thread is a tuple $\langle s, i, j, k \rangle$ where s is the current instruction, 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. Note that $\emptyset \neq 0$: A zero (0) for the start or end of a match indicates that a match starts or ends at offset 0; a null ($\emptyset$) indicates no match.

```
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`.

To clarify how `lightgrep` works, consider the stream `qabcabdbd` and a search request for the patterns `a(bd)+` and `abc`. Figure 4 shows the bytecode produced for these patterns. For comparison, the NFA corresponding to these patterns is shown in Figure 5.

To illustrate the procedure, we step through the execution of the bytecode as the stream is advanced one character at a time. The leftmost column lists the thread ID, the second column specifies the thread and the third column provides an explanation of the step.

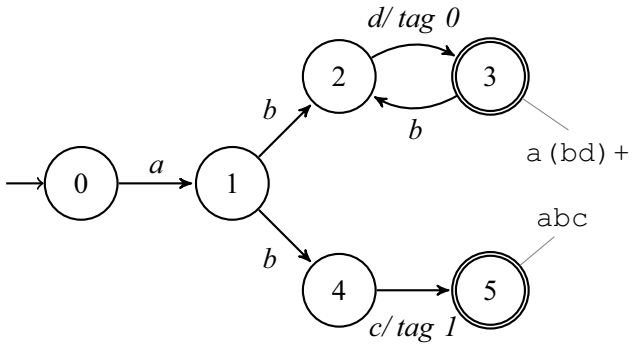

Figure 5. NFA matching `a(bd)+` and `abc`.

1: **q**abcabdbd

0	$\langle 0, 0, \emptyset, \emptyset \rangle$	thread 0 created
0	$\langle 0, 0, \emptyset, \emptyset \rangle$	`literal 'a'` fails; thread dies

2: q**a**bcabdbd

1	$\langle 0, 1, \emptyset, \emptyset \rangle$	thread 1 created
1	$\langle 0, 1, \emptyset, \emptyset \rangle$	`literal 'a'` succeeds
1	$\langle 1, 1, \emptyset, \emptyset \rangle$	advance instruction and suspend

3: qa**b**cabdbd

2	$\langle 0, 2, \emptyset, \emptyset \rangle$	thread 2 created
2	$\langle 0, 2, \emptyset, \emptyset \rangle$	`literal 'a'` fails; thread dies
1	$\langle 1, 1, \emptyset, \emptyset \rangle$	`fork 6` creates thread 3
3	$\langle 6, 1, \emptyset, \emptyset \rangle$	thread 3 created
1	$\langle 2, 1, \emptyset, \emptyset \rangle$	advance instruction
1	$\langle 2, 1, \emptyset, \emptyset \rangle$	`literal 'b'` succeeds
1	$\langle 3, 1, \emptyset, \emptyset \rangle$	advance instruction and suspend
3	$\langle 6, 1, \emptyset, \emptyset \rangle$	`literal 'b'` succeeds
3	$\langle 7, 1, \emptyset, \emptyset \rangle$	advance instruction and suspend

4: qab**c**abdbd

4	$\langle 0, 3, \emptyset, \emptyset \rangle$	thread 4 created
4	$\langle 0, 3, \emptyset, \emptyset \rangle$	`literal 'a'` fails; thread dies
1	$\langle 3, 1, \emptyset, \emptyset \rangle$	`literal 'd'` fails; thread dies
3	$\langle 7, 1, \emptyset, \emptyset \rangle$	`literal 'c'` succeeds
3	$\langle 8, 1, \emptyset, \emptyset \rangle$	advance instruction and suspend

5: qabc**q**abdbd

5	$\langle 0, 4, \emptyset, \emptyset \rangle$	thread 5 created
5	$\langle 0, 4, \emptyset, \emptyset \rangle$	`literal 'a'` fails; thread dies
3	$\langle 8, 1, \emptyset, \emptyset \rangle$	`match 1`
3	$\langle 8, 1, 4, 1 \rangle$	set match pattern and end offset
3	$\langle 9, 1, 4, 1 \rangle$	advance instruction
3	$\langle 9, 1, 4, 1 \rangle$	`halt`; report match on pattern 1 at $[1, 4)$; thread dies

6: qabcq**a**bdbd

6	$\langle 0, 5, \emptyset, \emptyset \rangle$	thread 6 created
6	$\langle 0, 5, \emptyset, \emptyset \rangle$	`literal 'a'` succeeds
6	$\langle 1, 5, \emptyset, \emptyset \rangle$	advance instruction and suspend

From here on, we do not mention the creation of threads that die immediately due to a failure to match the current character.

7: qabcqa**b**dbd

6	$\langle 1, 5, \emptyset, \emptyset \rangle$	`fork 6` creates thread 7
7	$\langle 6, 5, \emptyset, \emptyset \rangle$	thread 7 created
6	$\langle 2, 5, \emptyset, \emptyset \rangle$	advance instruction
6	$\langle 2, 5, \emptyset, \emptyset \rangle$	`literal 'b'` succeeds
6	$\langle 3, 5, \emptyset, \emptyset \rangle$	advance instruction and suspend
7	$\langle 6, 5, \emptyset, \emptyset \rangle$	`literal 'b'` succeeds
7	$\langle 7, 5, \emptyset, \emptyset \rangle$	advance instruction and suspend

8: qabcqab**d**bd

6	$\langle 3, 5, \emptyset, \emptyset \rangle$	`literal 'd'` succeeds
6	$\langle 4, 5, \emptyset, \emptyset \rangle$	advance instruction and suspend
7	$\langle 7, 5, \emptyset, \emptyset \rangle$	`literal 'c'` fails; thread dies

9: qabcqabd**b**d

6	$\langle 4, 5, \emptyset, \emptyset \rangle$	`match 0`
6	$\langle 4, 5, 8, 0 \rangle$	set match pattern and end offset
6	$\langle 5, 5, 8, 0 \rangle$	advance instruction
6	$\langle 5, 5, 8, 0 \rangle$	`jump 2`
6	$\langle 2, 5, 8, 0 \rangle$	goto instruction 2
6	$\langle 2, 5, 8, 0 \rangle$	`literal 'b'` succeeds
6	$\langle 3, 5, 8, 0 \rangle$	advance instruction and suspend

10: qabcqabdb**d**

6	$\langle 3, 5, 8, 0 \rangle$	`literal 'd'` succeeds
6	$\langle 4, 5, 8, 0 \rangle$	advance instruction and suspend

11: Having reached the end of the stream, the remaining threads run until they die:

6	$\langle 4, 5, 8, 0\rangle$	`match 0`
6	$\langle 4, 5, 10, 0\rangle$	set match pattern and end offset
6	$\langle 5, 5, 10, 0\rangle$	advance instruction
6	$\langle 5, 5, 10, 0\rangle$	`jump 2`
6	$\langle 2, 5, 10, 0\rangle$	goto instruction 2
6	$\langle 2, 5, 10, 0\rangle$	`literal 'b'` fails; report match of pattern 0 at $[4, 9)$; thread dies

The execution of this bytecode reports a match for `abc` at $[1, 4)$ and a match for `a(bd)+` at $[5, 10)$.

4.2 Optimizations

This section describes the optimizations implemented in `lightgrep`.

Minimization. Minimizing thread creation from unnecessary alternation is the key to improving performance in an NFA simulation. Rather than treating each pattern as a separate branch of the NFA, patterns are formed into a trie by incrementally merging them into the NFA as they are parsed. (A trie, also known as a prefix tree, is a tree whose root corresponds to the empty string, with every other node extending the string of its parent by one character. A trie is a type of acyclic DFA.) Merging must take into account not only the criteria of the transitions, but also the sets of source and target states.

To facilitate minimization, we use the Glushkov NFA form [6] instead of the Thompson form [16]. Constructing a Glushkov NFA is computationally more expensive, but it has only $m+1$ states, while a Thompson NFA has $O(2m)$ states. Additionally, a Glushkov NFA is free of no epsilon transitions, simplifying both the compilation process and the resulting bytecode.

Jump Tables for States with High Branching. Typically, one thread is forked to handle each successor of a given state. Some NFA states may have a large number of successors, making the creation of new threads costly. For example, the first state often has a large number k of outbound transitions when many patterns are specified. Therefore, every character read from the input stream causes k new threads to be created, almost all of which die immediately due to the lack of a match. Determining which threads will survive and spawning only these threads would be a significant practical improvement.

To accomplish this, we use the `jumptable` instruction. This instruction sits at the head of a list of 256 consecutive instructions, one for each possible value of the current byte. When the `jumptable` instruction is

reached with byte b, execution jumps ahead $b+1$ instructions and continues from there. The instruction offset $b+1$ from `jumptable` is generally a `jump` in the case of a match (in order to get out of the jump table), or a `halt` otherwise. If more than one transition is possible for byte b, then a list of appropriate `fork` and `jump` instructions is appended to the table and the `jump` instruction for byte b targets this table. In this manner, only the threads that succeed are spawned. The compiler takes care to specify jumps to states just beyond their `literal` instructions, ensuring that b is not evaluated twice. A sibling instruction, `jumptablerange`, used when the difference between the minimum and maximum accepted byte values is small, operates by checking that the byte value is in range and only then indexes into the table; this allows the jump table itself to be the size of the range, rather than the full 256 bytes.

Reduced State Synchronization. A typical simulation of an NFA uses a bit vector (containing a bit for each state) to track which states are visited for the current character in the stream in order to avoid duplicating work [2]. The number of NFA states depends on the combined length of the search patterns used; therefore, a search using a large number of patterns (even fixed-string patterns) forces this bit vector to be quite long. Either the bit vector must be cleared after each advance of the stream, or a complex checking process must be performed after each transition to update the bit vector.

Note that it is impossible for two threads to arrive at the same state at the same character position unless the state has multiple transitions leading to it. Therefore, only these states with multiple predecessors require bits in the current state vector; the bits for the other states are wasted.

The `lightgrep` implementation presented above makes no provision for such deduplication. In order to handle this, `lightgrep` uses the `chkhalt` instruction, which associates an index with each state having multiple incoming transitions. This instruction is inserted before the outbound transition instructions associated with a state requiring synchronization. The index associated with the state is specified as an operand to `chkhalt`, which uses it to test the corresponding value in a bit vector. The bit is set if it is currently unset, and execution proceeds. If the bit is already set, then the thread dies. In this manner, the size of the bit vector is minimized and safe transitions, which occur frequently in practice, are left unguarded.

Complex Instruction Set. As noted in the discussion of `jumptable` and `chkhalt`, it is easy to introduce new instructions to handle common

cases. For example, `either` has two operands and continues execution if the current byte matches either operand. Similarly, `range` has two operands and continues if the current byte has a value that falls within their range, inclusively. More complex character classes can be handled with `bitvector`, an instruction followed by 256 bits, each bit set to one if the corresponding byte value is permitted. If several states have the same source and target states, their transitions can be collapsed into a single `bitvector` instruction. In general, it is worthwhile to introduce a new instruction if it can eliminate sources of alternation.

Compilation. The `lightgrep` tool uses a hybrid breadth-first/depth-first search scheme to lay out the generated instructions. Instructions for states are first laid out in breadth-first order of discovery; the discovery switches to a depth-first search when a parent state has a single transition. This hybrid scheme has two advantages. First, subsequent states are generally close to their parent states due to breadth-first discovery. Second, the total number of instructions used can be reduced significantly in linear sequences of states since `jump` and `fork` instructions need not appear between them.

4.3 Additional Usability Features

This section describes additional usability features implemented in `lightgrep`.

Greedy vs. Non-Greedy Matching. As discussed in [4], it is possible to introduce non-greedy repetition operators such as `*?` that result in the shortest possible matches instead of the longest. Thread priority for alternations and repetitions can be controlled by executing forked threads before continuing execution on the parent thread and by careful ordering of `fork` instructions during compilation.

Non-greedy matching can be quite useful in digital forensics. Our prior example of the pattern `<html>.*</html>` is not appropriate for carving HTML fragments from unallocated space in a file system. The pattern matches the first fragment, but a thread will continue trying to match beyond the fragment, eventually producing a match at the end of the last such fragment (if it exists) and reporting one long match. In contrast, `<html>.*?</html>` generates one match for each fragment.

Positional Assertions. The `vi` text editor offers users the ability to specify positional assertions in patterns. For example, a pattern can assert that it must match the pattern on a certain line, in a certain column. Positional assertions can have useful applications in searching

binary data for forensic applications. A file format may have an optional record that can be identified with a pattern, but that is known to occur only at a given offset. Further, file carving may be limited to data that is sector-aligned. To accomplish this, we introduce the syntax `(?`*i*`@`*regex*`)` and `(?`*i*`%`*j*`@`*regex*`)`, where i is either an absolute or modulo byte offset and j is a divisor. Thus, `(?0%512@)PK` would match sector-aligned ZIP archive headers.

Multiple Encodings. Many regular expression libraries with Unicode support rely on data to be decoded to Unicode characters before consideration by the search routine, on the assumption that the data to be searched is stored in a single encoding. This is not a valid assumption in digital forensics—when searching unstructured data, encodings may change capriciously from ASCII to UTF-16 to UTF-8 to a legacy code page. `lightgrep` is explicitly byte-oriented. In order to search for alternate encodings of a pattern, its various binary representations must be generated as separate patterns in the NFA. Matches can then be resolved back to the user-specified term and appropriate encoding using a table.

`lightgrep` can search for ASCII-specified patterns as ASCII and as UTF-16. Full support for various encodings is under active development; the open source ICU library [9] is being used to eliminate platform dependencies. In addition to specifying the particular encodings to be used for a given search term, users may choose an automatic mode, where the characters of a keyword are considered as Unicode code points. All unique binary representations are then generated from the list of supported ICU encodings, which will aid searches for foreign-language keywords.

5. Experimental Results

In order to benchmark `lightgrep`, we created a list of 50 regular expressions suitable for use in investigations, with moderately aggressive use of regular expression operators. Some of the terms are for text, others for artifacts and files. Testing used increasing subsets of the terms, from five terms to 50 in five-term increments. Of the search algorithms mentioned in Section 3, only EnCase has enough features in common with `lightgrep` for a head-to-head performance comparison to be meaningful. Therefore, we compared only EnCase and `lightgrep` in our experiments.

With both EnCase and `lightgrep`, the tests ran each group of keywords against a 32 GB Windows XP `dd` evidence file. The file systems in the evidence file were not parsed. The workstation used had two

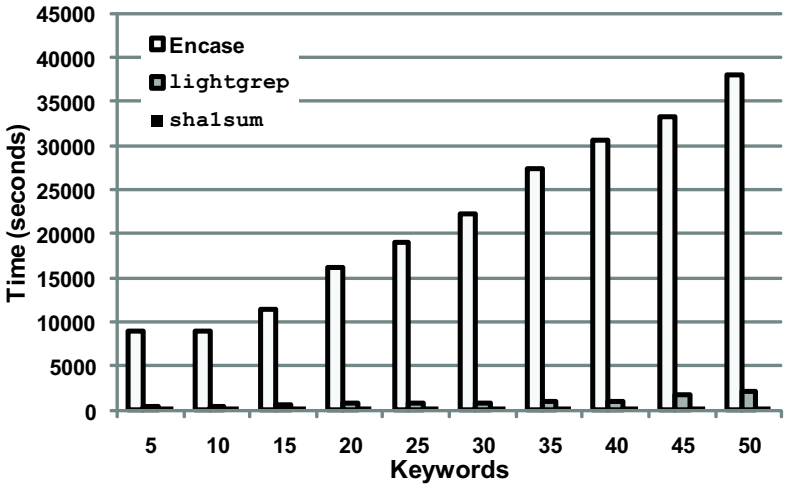

Figure 6. Wall-clock time for EnCase and `lightgrep`.

Intel Xeon 5160 3 GHz dual-core processors with 4 GB RAM and a 7,200 rpm SATA2 hard drive. Figure 6 shows the wall-clock execution times. `lightgrep` dramatically outperformed EnCase on the test data – by more than a factor of ten in all cases.

As a further benchmark, we compared `lightgrep`'s completion time with the time required to hash the evidence file with SHA-1 using the `sha1sum` command. SHA-1 hashing reads every byte of the input and is input/output-bound, so the SHA-1 timings provide a lower bound for search performance. The results show that `lightgrep` comes very close to matching hash performance with small sets of keywords.

Additionally, a `lightgrep` search was conducted with 114,743 fixed strings from an English word list (not shown in Figure 6). This search completed in 523 seconds, just 50 seconds more than the time needed to hash the evidence. Because fixed strings collapse into a DFA for matching (but not searching), this indicates that performance improvements with complex patterns can be achieved by further determinization of the NFA.

6. Conclusions

Tagged NFAs are easily applied to the multipattern problem and optimizations can keep the observed performance below the worst-case $O(nm)$ running time as the automata sizes increase. `lightgrep` uses these mechanisms to provide digital forensic examiners with a sorely-

needed capability, allowing evidence to be searched for large keyword sets in a single pass.

The `lightgrep` tool is currently undergoing robust acceptance testing to ensure confidence in its results and support for generating alternative patterns for matching in multiple encodings. An obvious optimization is to multiplex the execution of virtual machine threads onto system threads, exploiting multicore processors.

The malleability of the bytecode representation for automaton matching enables it to be used with newer matching algorithms that can skip bytes in the text based on precomputed shift tables. For example, Watson [17] describes a sublinear multipattern matching algorithm that combines a Commentz-Walter fixed-string search for prefixes of matches with full automaton evaluation for the complete pattern. Also, pattern matching research related to network packet inspection and rule matching in intrusion detection systems can be applied to digital media searching. An example is the work of Becchi and Crowley [3] on optimizations related to counted repetitions in patterns.

The current version of `lightgrep` does not search for near-matches. Near-matching can be performed using Wu and Manber's algorithm [18], which is implemented in the `agrep` and `TRE` [11] search utilities. Alternatively, fuzzy matching functionality may be implemented using a bit-parallel algorithm as in `nrgrep`.

References

[1] AccessData, Forensic Toolkit, Lindon, Utah (www.accessdata.com/forensictoolkit.html).

[2] A. Aho, M. Lam, R. Sethi and J. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Boston, Massachusetts, 2007.

[3] M. Becchi and P. Crowley, Extending finite automata to efficiently match Perl-compatible regular expressions, *Proceedings of the International Conference on Emerging Networking Experiments and Technologies*, 2008.

[4] R. Cox, Regular expression matching: The virtual machine approach (swtch.com/∼rsc/regexp/regexp2.html), 2009.

[5] R. Cox, RE2: An efficient, principled regular expression library (code.google.com/p/re2), 2010.

[6] V. Glushkov, The abstract theory of automata, *Russian Mathematical Surveys*, vol. 16(5), pp. 1–53, 1961.

[7] Guidance Software, EnCase, Pasadena, California (www.guidance software.com).

[8] S. Garfinkel, Forensic feature extraction and cross-drive analysis, *Digital Investigation*, vol. 3(S), pp. 71–81, 2006.

[9] International Business Machines, ICU – International Components for Unicode, Armonk, New York (icu-project.org), 2010.

[10] V. Laurikari, NFAs with tagged transitions, their conversion to deterministic automata and applications to regular expressions, *Proceedings of the Seventh International Symposium on String Processing and Information Retrieval*, pp. 181–187, 2000.

[11] V. Laurikari, TRE – The free and portable approximate regex matching library (laurikari.net/tre), 2010.

[12] J. Maddock, Boost.Regex (www.boost.org/doc/libs/1_43_0/libs/regex/doc/html/index.html), 2009.

[13] G. Navarro, NR-grep: A fast and flexible pattern-matching tool, *Software Practice and Experience*, vol. 31(13), pp. 1265–1312, 2001.

[14] G. Navarro and M. Raffinot, *Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences*, Cambridge University Press, Cambridge, United Kingdom, 2007.

[15] M. Sipser, *Introduction to the Theory of Computation*, PWS Publishing, Boston, Massachusetts, 1997.

[16] K. Thompson, Regular expression search algorithm, *Communications of the ACM*, vol. 11(6), pp. 419–422, 1968.

[17] B. Watson, A new regular grammar pattern matching algorithm, *Proceedings of the Fourth Annual European Symposium on Algorithms*, pp. 364–377, 1996.

[18] S. Wu and U. Manber, Agrep – A fast approximate pattern-matching tool, *Proceedings of the USENIX Winter Technical Conference*, pp. 153–162, 1992.