Finite State Machine Parsing for Internet Protocols: Faster Than You Think

Robert David Graham
Errata Security
robert_david_graham@yahoo.com

Peter C. Johnson
Department of Computer Science
Dartmouth College
Hanover, NH USA
pete@cs.dartmouth.edu

***Abstract*—A parser's job is to take unstructured, opaque data and convert it to a structured, semantically meaningful format. As such, parsers often operate at the border between untrusted data sources (e.g., the Internet) and the soft, chewy center of computer systems, where performance and security are paramount. A firewall, for instance, is precisely a trust-creating parser for Internet protocols, permitting valid packets to pass through and dropping or actively rejecting malformed packets. Despite the prevalence of finite state machines (FSMs) in both protocol specifications and protocol implementations, they have gained little traction in parser code for such protocols. Typical reasons for avoiding the FSM computation model claim poor performance, poor scalability, poor expressibility, and difficult or time-consuming programming.**

In this research report, we present our motivations for and designs of finite state machines to parse a variety of existing Internet protocols, both binary and ASCII. Our hand-written parsers explicitly optimize around L1 cache hit latency, branch misprediction penalty, and program-wide memory overhead to achieve aggressive performance and scalability targets. Our work demonstrates that such parsers are, contrary to popular belief, sufficiently expressive for meaningful protocols, sufficiently performant for high-throughput applications, and sufficiently simple to construct and maintain. We hope that, in light of other research demonstrating the security benefits of such parsers over more complex, Turing-complete codes, our work serves as evidence that certain "practical" reasons for avoiding FSM-based parsers are invalid.

I. Introduction

Parsers are responsible for translating unstructured, untrusted, opaque data to a structured, implicitly trusted, semantically meaningful format suitable for computing on. Parsers, therefore, are the components that facilitate the separation of *data* from *computation* and, hence, exist in nearly every conceivable useful computer system.

Any program that presents the contents of data files contains a parser: word processors show documents, media players play video or audio, databases store and retrieve arbitrary data. More generally, any program that accepts input (interactively or not) contains a parser to translate from the unstructured input data to whatever internal structures the program uses to process that input. The source of input varies, of course—it could be bytes read off a disk, events from a keyboard or mouse, or data over a network interface—but the basic task remains the same: create structure from chaos.

As a result of this mandate, parsers are nearly always tasked (explicitly or otherwise) with determining whether a reasonable structure can even be derived from the chaotic input; that is, *parsers imbue input with trust*. For example, a firewall examines data attempting to enter or exit a network; packets approved by the firewall policy are allowed through, packets that are not approved are dropped or rejected. Having passed through, internal machines assume these packets to be more trustworthy than those that failed to pass muster at the perimeter; the firewall has essentially blessed them with its seal of approval. The same can be said of parsers for documents or data files: if the parser finds a problem with the data file, the program will (or at least should) emit an error rather than attempt to load a broken file.

These examples highlight two vital requirements of parsers: they must be *correct*, so that only valid input is blessed with trust; and they must be *efficient* so that enormous documents and torrential datastreams (for word processors and firewalls, respectively) don't bring systems to their knees. Correctness is reasonably straightforward (though perhaps not "simple", per se), but efficiency has some interesting subtleties because it encompasses all resources a parser might use. The obvious resource to consider is CPU cycles but, at scale, memory becomes an issue as well.

Both of these performance axes affect projects we have worked on, best illustrated by `masscan` [4], an Internet port scanner that can cover the entire IPv4 address space in under 6 minutes, at 10 million packets per second, from a single machine. To achieve this rate, `masscan` is extremely efficient in both the number of CPU cycles it spends parsing each byte received over the wire as well as the memory it uses to store state for the millions of simultaneous connections it maintains. This scanning rate is an order of magnitude faster than alternatives such as nmap [7] and Zmap [3].

Another project where performance reigns supreme is `robdns` [5], a DNS server designed to service 10 million requests per second. One interesting aspect of this project is that responding to individual DNS requests is not the only performance-sensitive operation: at its intended scale, just parsing the DNS data ("zone files") from disk is a potential bottleneck! Some DNS servers, such as Knot [6] and Yadifa [9], have optimized their parsing of DNS zone files but, just like `masscan` outperforms its competitors, `robdns` is

DOI 10.1109/SPW.2014.34

faster than all of them.

Finally, an intrusion detection system (IDS) needs to not only parse data, it must also search otherwise-opaque fields for questionable patterns. For instance, where a typical HTTP header parser would find the `Host` field and blindly return all following characters through the end of line as the value, an IDS needs to be able to search that value for potentially-malicious contents. To analyze data at line speed, the parser inside an IDS must be both fast, to keep up with data rates in excess of 1 Gbps, and compact, to maintain state for a potentially huge set of concurrent connections.

What's the secret sauce? you ask. *Finite state machines.*

No way! you say. *Way.*

Not just finite state machines, but FSMs that parse input byte-by-byte, sequentially, without backtracking, and are constructed to maximize throughput by taking into account architectural characteristics such as L1 cache hit rate and branch misprediction penalties. In the remainder of this research report, we will demonstrate that hand-written FSM parsers can handle real-world protocols such as DNS zone files, X.509 certificates, and HTTP headers; that these parsers are extremely efficient with system resources; and that they scale better than existing implementations in software such as the nginx and Apache web servers and the `openssl` cryptography library.

We next review finite state machines and how they apply to parsing (Section II), then we describe the architectural considerations we kept in mind while writing our parsers (Section III), next we provide an overview of three of the parsers we implemented along with performance metrics (Section IV), discuss future directions (Section V), and, finally, conclude (Section VI).

II. Finite State Machines for Parsing

A finite state machine—also known as a deterministic finite automaton (DFA)—is defined by a set of states, exactly one of which is the *start state*, some subset of which are the *accepting states*, a set of input tokens called the *alphabet*, and a *transition function* that, given the next input token and the current state, determines the next state. A DFA is said to *accept* a particular input if and only if, starting in the start state and repeatedly applying the transition function to each input token in sequence, one winds up in a accepting state when one reaches the end of the input. Any input that is not accepted is *rejected*. The *language* of a DFA is the set of all inputs it accepts; equivalently, the DFA is said to *recognize* that language. Furthermore, the class of languages that can be recognized by a DFA are called *regular languages*.

To parse a particular protocol, we require a DFA that recognizes the language of all valid messages in that protocol. Consider the task of parsing HTTP headers, an example that we will refer to throughout this section. Without loss of generality, and to save space, let's assume we only have to deal with a single header field: `Host`.

One way to represent a DFA is graphically; our toy HTTP header parser is shown in Figure 1. The circles are states, with

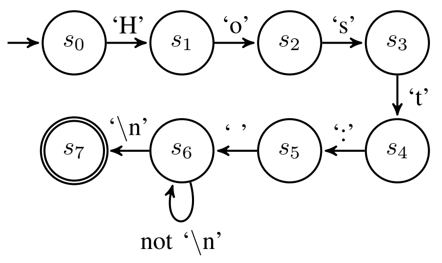

Fig. 1. A simple DFA that recognizes the HTTP "Host" header. Note that if a transition does not exist for a given input character from a given state, the entire input is rejected. State s_0 is the start state and state s_7 is the accepting state.

	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7
'H'	s_1							
'o'		s_2						
's'			s_3					
't'				s_4				
':'					s_5			
' '						s_6		
'not \n'	s_1						s_6	
'\n'	s_1						s_7	

TABLE I
Tabular representation of transition function for DFA shown in Figure 1. An empty cell indicates immediate failure for a given current state and next input token.

the start state signified by a tailless arrow pointing to it and the accepting states shown as double circles. The transition function is shown as arrows from current state to next state, labeled by the alphabet token that induces the transition. If there is no appropriately labeled arrow given the current input token, the entire input is rejected.

Equivalently, a DFA can be represented in tabular format; Table I shows the same DFA from Figure 1 as such.

This DFA only recognizes a single line of an HTTP header. We can compose several such DFAs to recognize the entire set of headers, as shown in Figure 2. To save space, we have taken a bit of a shortcut in notation by not including distinct states for each character read, but those are easy to extrapolate.

A. Implementation Methods

In practice, there are many ways to implement deterministic finite automata. One popular method is as a two-dimensional array precisely like the one shown in Table I along with a loop similar to this:

```
cur_state = start_state;
while(i < input_length) {
  c = input[i];
  cur_state = table[cur_state][c];
}
```

Alternatively, one could use a massive `switch` statement, conditioned on the current state, where each case individually sets the next state. Another option is to use a *jump table*, which is similar to the tabular method described above, with the exception that each entry is a function pointer to the handler

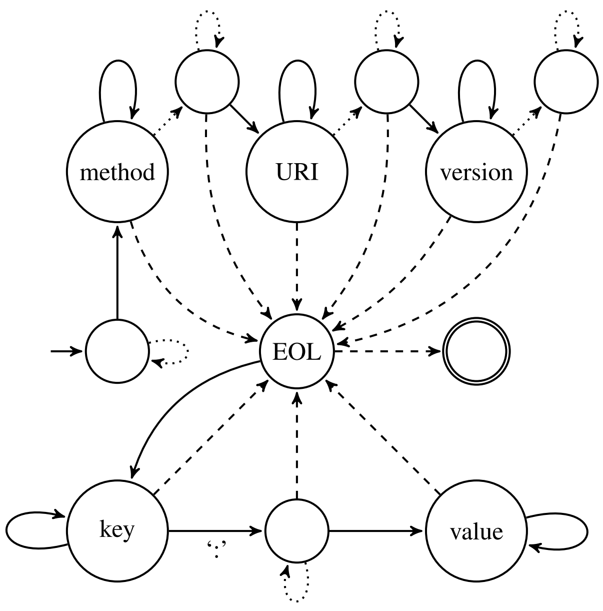

Fig. 2. DFA that recognizes HTTP headers, with the request on the first line, followed by arbitrary key-value pairs on subsequent lines, ending with two consecutive newlines. Solid edges represent any printable character, dotted lines represent a space, dashed lines represent a newline. The unlabeled states just eat spaces.

for that case. Finally, a true sadist (or masochist, depending on your perspective) could desecrate Dijkstra's memory by using a complex pile of `goto` spaghetti.

These relatively naïve approaches are quite widely used, though they leave room for improvement, especially when one keeps in mind the underlying hardware architecture upon which these programs run.

III. Design Considerations

We have identified a number of hardware design parameters that affect parsing using DFAs. Some of these, such as cache hit latency, help us calculate maximum theoretical throughput for DFA-based parsers, whereas others guide our efforts to reach this maximum.

A. Branch (Mis)prediction

As processor pipelines have grown deeper and deeper, the importance of the branch prediction unit has increased: every incorrect prediction induces a flush of the entire pipeline, wasting precious cycles. Despite the fact that modern branch prediction units have gotten extremely accurate (well above 90% for some workloads), the inherently unpredictable nature of parsing random input runs the risk of running a pipeline full of bubbles. Or does it?

As described above, massive `switch` statements are often used to implement DFAs, with one loop iteration for each input token. We have found, however, that the things we want to parse frequently take the shape of surprisingly linear DFAs; refer back to Figure 1 for such an example.

microarchitecture	target platform	pipeline depth
AMD Bobcat	mobile	15 stages
AMD Bulldozer	desktop/server	16 to 19 stages
ARM Cortex-A8 (in-order)	mobile	13 stages
ARM Cortex-A9 (out-of-order)	mobile	8 stages
IBM Power8	server	16 stages
Intel Core	desktop/server	14 to 19 stages
Intel Silvermont	mobile	16 stages

TABLE II
Pipeline depths of modern microarchitectures.

Implementing each state as a distinct case in a `switch` statement is certainly a viable approach but we can use our knowledge of what we're parsing to take a shortcut. We know that there is only a single valid transition from, e.g., state s_2 to state s_3, so we can avoid looping back around to the beginning of the `switch` and instead fall through to the next state by omitting the `break` statement in the code for s_2. This saves us cycles on a number of fronts: we take the branch prediction unit out of the equation entirely, thus avoiding the danger of an incorrect prediction; the processor then (correctly) performs its speculative execution of state s_3; and we maintain spatial locality of the instruction cache.

Table II surveys the pipeline depths of a number of modern microprocessor architectures. The deeper the depth, the greater the impact of a missed branch prediction, and the more savings we realize by eliminating as many branches as we can in otherwise branch-heavy code like parsing.

B. Memory Overhead

The `masscan` Internet scanning tool achieves its staggering scanning rate with a number of tricks, many of which are intended to minimize the memory overhead of managing millions of concurrent connections. While these techniques don't necessarily affect parsing, per se, we believe they are instructive examples for those looking to implement high-performance parsers (which is essentially what `masscan` is).

First and foremost, `masscan` bypasses the kernel network stack and uses a NIC driver that deposits raw packets directly to user memory. This avoids a great deal of switching between user mode and supervisor mode in both hardware and software. Recall that on every mode switch, the processor must flush the pipeline! Avoiding these switches is extremely beneficial to performance.

Secondly, `masscan` discards all fragmented packets with the understanding that the sending machine will just re-send the packets whole. Ignoring fragments allows `masscan` to parse every frame sequentially, without the need to buffer *anything*: the only memory it must devote to any given connection is a few integers indicating its current state in the parser. For example, due to using nested state machines, our X.509 parser requires 58 bytes per TCP connection to store its current state. At 10 million concurrent connections, that amounts to 580 megabytes. In contrast, the Linux kernel uses in excess of 2 kilobytes per TCP connection [8]—not counting buffers for storing packet fragments. `masscan` is two orders of magnitude more efficient in its use of memory!

C. Pattern Matching using Aho-Corasick

Parsers are used to assign semantic meaning to otherwise-opaque data, and in many cases the parser isn't immediately concerned with the precise contents of the data. For instance, when a web server parses incoming HTTP headers, one of the fields it identifies is `Host`. All the parser will do is identify the associated value (i.e., all characters following "Host: " through to the line terminator) and send that data up to a higher level in the program for further intepretation.

There are, however, applications that benefit from putting more logic in the parser; network intrusion detection systems (IDS) and antivirus software are a few. Traditionally, IDSes and antivirus software maintain a library of *signatures* (i.e., byte strings) they believe identify invalid and/or malicious data; to perform their duties, they scan network or disk data for the presence of these character strings. In DFA parlance, these applications *recognize* the language of all strings that contain *any* of the signatures as a substring.

We could, of course, construct DFAs for this purpose by hand, but it turns out that Al Aho and Margaret Corasick figured out how to programmatically generate them many years ago. Given a set of strings to search for, the Aho-Corasick [1] algorithm produces a DFA that finds all occurrences of those strings (verbatim) within a particular input string. Essentially, it *recognizes* the language of all input strings that contain one or more of the search strings within it (with some sugar on top to take action whenever a search string is found). This is precisely what intrusion detection systems and antivirus software do!

`masscan` makes extensive use of Aho-Corasick parsers: to find specific fields within HTTP headers, or X.509 certificates, or values within fields, and so on. These automatically generated parsers are especially useful when matches require no further processing. We found hand-written parsers were more effective when the matched bytes required some processing, e.g., converting ASCII integers to binary for later use, as in the case of run-length encoded values.

D. Input Token Translation

Another technique to improve parser performance is to compress the input token space with a translation function. Because we're parsing one byte at a time, our input alphabet is technically the entire range of 256 possible values, but (especially for ASCII protocols) many are not used in valid parses. When we encode the transition function as a table, this sparse use of the input token space results in a sparsely-populated table. Even a relatively modest number of states will therefore cause the table to, if not exceed the size of the L1 cache on its own, at least cause otherwise premature eviction of useful entries.

To mitigate this, before we perform the transition table lookup, we translate the raw input token to a tightly-packed space of tokens that are meaningful in this particular parse. If we need only concern ourselves with printable ASCII, this technique reduces the memory footprint of the transition table by at least 50%.

E. Cache Latency

Recall the loop shown in Section II-A for evaluating a DFA against an input; it is a particularly tight loop, especially because the body,

```
state = table[state][c];
```

can be executed as a single instruction in x86 assembly:

```
mov ebx[eax+ecx],%eax
```

Prior to execution, register `ebx` holds the base address of `table`, `eax` holds the value of `state`, and `ecx` holds the value of `c`. After execution, `eax` holds the new value of `state`.

In the absolute best case, this memory-register operation will result in an L1 cache hit. Modern processors are often able to hide the latency of retrieving a value from cache by executing instructions out of order. If we unroll the loop a few times, however, we can see that each instruction depends on the result of the previous:

```
mov ebx[eax+ecx],%eax
mov ebx[eax+ecx],%eax
mov ebx[eax+ecx],%eax
mov ebx[eax+ecx],%eax
```

The second instruction can't complete until the result of the first instruction has been loaded from cache.[1] Therefore, we conclude that the absolute lower bound on parser execution is c cycles per byte, where c is the L1 cache hit rate in cycles. This exposition isn't merely academic. By understanding the architectural limitations on the execution of parsers, we can evaluate their performance against the ideal rather than against the rest of the horses in the stable.

Additionally, we are better-informed when writing our parsers. If we know the L1 cache hit rate is c cycles, then we know we have c instructions we can execute between each of those `mov` instructions in the parser's inner loop. We can use those cycles to do things like the input token translation described in Section III-D and to check if we've found a match. In the following section, we will show precisely how this works.

IV. Implementations

As mentioned previously, we are more concerned with comparing our parsers' performance against the theoretical maximum rather than other implementations. (We do plan to compare against others, though; we will elaborate in Section V.) In the following subsections, we describe the performance of both synthetic microbenchmarks we have developed as well as full-fledged applications that parse X.509 certificates and DNS zone files.

[1]The presence of pipeline stages and superscalar execution present further potential slowdows, but modern processors seem to be quite efficient at mitigating these, as we will show in Section IV.

				synthetic		Aho-Corasick	
Processor	clock rate	L1 speed	max theo.	asm-idx	asm-ptr	cycles/byte	parse speed
AMD Bobcat	1.6 GHz	3 cycles	4.3 Gbps	4.2 cycles	3 cycles	6.770	1.89 Gbps
AMD Bulldozer (Piledriver)	4 GHz	4 cycles	8 Gbps	5 cycles	3.8 cycles	4.974	6.46 Gbps
Intel Atom (Cedarview)	1.6 GHz	3 cycles	4.3 Gbps	4.2 cycles	3 cycles	14.257	895 Mbps
Intel Core (Ivy Bridge)	2.5 GHz	4 cycles	5 Gbps	5 cycles	4 cycles	5.015	3.99 Gbps
Intel Core (Sandy Bridge)	3.2 GHz	4 cycles	6.4 Gbps	5 cycles	4 cycles	5.033	5.07 Gbps
Intel Core (Westmere)	2.13 GHz	4 cycles	4.3 Gbps	4 cycles	4 cycles	4.068	4.21 Gbps

TABLE III
PERFORMANCE METRICS FOR VARIOUS MODERN PROCESSORS, RESULTS OF RUNNING BOTH SYNTHETIC BENCHMARKS TESTING L1 CACHE LATENCY AND A PARSER GENERATED BY THE AHO-CORASICK ALGORITHM.

A. Synthetic Microbenchmark

We wrote two programs to establish a lower bound for our parser's inner loop:

- `asm-idx` is a series of raw x86 assembly instructions that look up values in a two-dimensional array such that the result of one instruction is used as an array index in the subsequent instruction.
- `asm-ptr` is the same as `asm-idx` with the exception that it adds a pointer dereference, the purpose being to exercise an alternate addressing mode in Intel's implementation of the x86 ISA.

We measured execution time of our programs using the `rdtsc` instruction. The results of running these synthetic benchmarks on six different processors are shown in the `asm-idx` and `asm-ptr` columns of Table III. Notice how the performance of `asm-ptr` closely tracks the L1 cache hit latency for each processor. Intel's desktop- and server-centric microarchitectures (Ivy Bridge, Sandy Bridge, and Westmere) are especially good, where the Atom and AMD's Bobcat and Bulldozer cores lag by one clock cycle.

We also generated an Aho-Corasick parser that searches for six words within the text of the King James Bible. Our performance measurements for this are shown in the final two columns of Table III. Especially important are two comparisons:

- cycles per byte parsing rate we observed in our code (column 7) and L1 cache hit rate (column 3);
- and observed parsing speed (column 8) and max theoretical parsing speed (column 4).

Performance of the Aho-Corasick parser is especially good on Intel's Westmere architecture, getting within 2% of the theoretical maximum. In contrast, performance is especially poor on Intel's Cedarview, achieving barely 20% of the theoretical maximum. This is likely due in large part to the fact that Cedarview is an in-order architecture; Intel's newest Atom processors (Silvermont) use an out-of-order execution model that we expect would realize significant benefits for the workloads we care about.

B. X.509 Certificates

The internationally-recognized X.509 standard [2] defines a format for cryptographic certficates in public key infrastructure (PKI) systems. One of the tasks for which we created `masscan` was performing a census of PKI certificates presented by Internet-accessible machines. To meet our performance goals, we needed a much faster method of parsing these certificates than that provided by the widely-used OpenSSL library.

Certificates are so large that they typically cross TCP packet boundaries. That means block parsers, which is what OpenSSL uses, must first allocate a large chunk of memory to which it then copies the contents of multiple packets as they arrive. Moreover, SSL may fragment a certificate in addition to fragmentation that may be induced at the TCP layer, meaning that multiple buffers are used and many copies are performed. This requires not just CPU cycles, but also many kilobytes per TCP connection; kilobytes we don't have to spare when attempting to support 10 million simultaneous connections.

As you have probably guessed, `masscan` solves this problem in part by using a streaming parser for X.509 certificates (in addition to the userspace TCP stack discussed in Section III-B. Due to the complexity of the standard, we in fact use several nested state machines and thus require 58 bytes to remember the state of each connection, plus 100 bytes to buffer specific fields that we extract. Even so, we use two orders of magnitude less memory than OpenSSL running on the kernel's TCP stack.

C. DNS Zone Files

DNS has two formats: an on-the-wire format for transmitting data between machines and a file format for storing the database on disk. One popular format for this file is the "zone file" popularized by BIND and later supported by many other servers. (Listing 1 shows a typical example.) On startup, the DNS server must parse this text file to build its in-memory database before servicing any client requests.

Very large zone-files can take a long time to parse, incurring a long delay before the DNS server can begin responding to requests; a delay which exacerbates unexpected outages. For reference, the zone-file for the ".com" top-level domain presents a particularly difficult problem: it contains 200 million domains and is over 8 gigabytes in size. BIND can take 2000 seconds to parse and load this file; other DNS servers, such as yadifa and knot-dns, have focused on improving this problem and can load it in about 450 seconds.

We have written a prototype DNS server that parses and loads the ".com" zone-file in 233 seconds using a state-machine parser and a single thread on similar hardware

```
$ORIGIN example.com.
$TTL 1h
example.com.  IN  SOA  ns.example.com. (
              2007120710 ; serial number
              1d         ; refresh period
              2h         ; retry time
              4w         ; expiration
              1h         ; max cache time
              )
example.com.  NS    ns
example.com.  NS    ns.somewhere.example.
example.com.  MX    10 mail.example.com.
@             MX    20 mail2.example.com.
@             MX    50 mail3
example.com.  A     192.0.2.1
              AAAA  2001:db8:10::1
ns            A     192.0.2.2
              AAAA  2001:db8:10::2
www           CNAME example.com.
wwwtest       CNAME www
mail          A     192.0.2.3
mail2         A     192.0.2.4
mail3         A     192.0.2.5
```

Listing 1. Example DNS zone file

(Westmere 2.13 GHz). Parsing the zone-file takes 42 seconds, with the remainder of the time taken inserting the parsed data into an in-memory database. There is no way to directly compare the parsing efficiency with the other DNS servers, but this demonstrates that a state-machine parser is useful for systems that focus on speed.

A more direct comparison is comparing our DNS parsing code to `wc`, the standard UNIX word-count program. Whereas parsing a zone-file is quite complex, counting characters, words, and lines is simple. On the same system (IvyBridge 3.2 GHz), our DNS program parses the ".com" zone-file in 35 seconds of user-time, whereas word-count takes 103 seconds of user-time.

V. Future Work

As this is a research report and not a full-fledged research project, we envision a great deal of work to strengthen the promising results presented.

We plan to pick apart a suite of existing applications that contain parsers for Internet protocols to see how they work. Specifically, we are interested in the methods employed by nginx, Apache, openssl, Yadifa, and Knot to parse HTTP headers, X.509 certificates, and DNS zone files. We plan to extract the parsing routines from these pacakges as much as we can to compare their performance against our own parsers. We also hope to precisely quantify the benefits of the various optimization techniques we described in Section III.

Using the lessons we learned from this investigation, we plan to produce a system to generate DFA-based parsers tailored to the characteristics of the hardware they are to be run upon. Ideally, we would feed in parsing rules and architectural parameters such as L1 cache hit rate and produce a tuned binary.

VI. Conclusion

In the preceding sections, we have described the demanding applications that drove the designs and implementations of our parsers, and the not-at-all-novel but still (in our opinion) underappreciated finite state machine as a tool to help us meet those demands. Being mindful of the underlying hardware architecture, we have measured the maximum theoretical throughput of FSM-based parsers and have demonstrated that the parsers we have written for real-world applications such as text searching can, in the best case, achieve performance within 2% of that maximum. We have also presented other optimization techniques we have employed that help us hit our performance targets which, while perhaps not directly applicable to parsing in specific, nonetheless can serve to guide those looking to implement tremendously scalable systems.

It is our hope that this research report serves as evidence that arguments such as "they don't scale" and "they can't parse meaningful protocols" are invalid when considering using finite state machines for parsing real-world protocols. Additionally, we hope that the cycles-per-byte metric presented in Section IV, along with the target of meeting the L1 cache hit latency, receive higher visibility among those looking to implement parsers where performance is paramount.

References

[1] Alfred V. Aho and Margaret J. Corasick. "Efficient String Matching: An Aid to Bibliographic Search". In: *Communications of the ACM* 18.6 (June 1975).

[2] David Cooper et al. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. RFC 5280. May 2008. URL: http://www.rfc-editor.org/rfc/rfc5280.txt.

[3] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. "ZMap: Fast Internet-Wide Scanning and its Security Applications". In: *Proceedings of the 22nd USENIX Security Symposium*. 2013.

[4] Robert David Graham. *MASSCAN: Mass IP port scanner*. URL: https://github.com/robertdavidgraham/masscan.

[5] Robert David Graham. *`robdns`: a fast DNS server*. URL: https://github.com/robertdavidgraham/robdns.

[6] *Knot DNS*. URL: https://www.knot-dns.cz/.

[7] *Nmap*. URL: http://nmap.org.

[8] Kumiko Ono and Henning Schulzrinne. "One Server Per City: Using TCP for Very Large SIP Servers". In: *Principles, Systems and Applications of IP Telecommunications. Services and Security for Next Generation Networks*. Ed. by Henning Schulzrinne, Radu State, and Saverio Niccolini. Springer-Verlag, 2008.

[9] *Yadifa*. URL: http://www.yadifa.eu/.