2014 6th International Conference on Cyber Conflict
P.Brangetto, M.Maybaum, J.Stinissen (Eds.)
2014 © NATO CCD COE Publications, Tallinn

Elastic Deep Packet Inspection

Bruce W. Watson
Dept. of Information Science
Stellenbosch University
Stellenbosch, South Africa
bruce@fastar.org

IP Blox
Kelowna, Canada
bruce@ip-blox.com

Abstract: Deep packet inspection (DPI) systems are required to perform at or near network line-rate speeds, matching thousands of rules against the network traffic. The engineering performance and price trade-offs are such that DPI is difficult to virtualize, either because of very high memory consumption or the use of custom hardware; similarly, a running DPI instance is difficult to 'move' cheaply to another part of the network. Algorithmic constraints make it costly to update the set of rules, even with minor edits.

In this paper, we present *Elastic DPI*. Thanks to new algorithms and data-structures, all of these performance and flexibility constraints can be overcome – an important development in an increasingly virtualized network environment. The ability to incrementally update rule sets is also a potentially interesting use-case in next generation firewall appliances that rapidly update their rule sets.

Keywords: *deep packet inspection (DPI), speed/memory performance, incremental defense*

1. INTRODUCTION

In this paper, we describe a new approach to deep packet inspection (DPI) – known as *Elastic DPI (EDPI)*. Next-generation firewalls (NGFW's, which include intrusion-detection and -prevention systems) usually consist of two main architectural components:

1. *Sensors*, which inspect network traffic (standard TCP/IP traffic, but also network area storage traffic, etc.), reporting back on which 'rules' matched. Sensors have previously been *shallow* packet inspectors for performance reasons, but DPI has now reached line-rate performance – and the *deep* inspection of network traffic makes it an obvious choice of sensor. A typical system will involve DPI instances deployed at various points in the network, perhaps with different rule sets to gain various types of insight into the traffic.

2. *Aggregators* and *correlators*, which take input from many sensors, assembling a broader picture of a threat. There may be multiple levels of aggregation or correlation, each feeding upwards to a more general level that eventually signals an alarm or threat.

Throughout this paper, we focus on DPI sensors. A good overview of the technical details or the field is given in (Varghese, 2005). Ongoing advances in this field are typically covered in conferences such as RAID[1].

Present day DPI is typically static in *where* it is run – either because of large computational needs or semi-custom hardware. Furthermore, while the rules recognized by a particular DPI instance are changeable, such changes (even for a single rule add, edit, or delete) are not fast enough to be done while processing a packet.
These aspects of current DPI (discussed in more detail in the next section) hint at the following problematic use-cases:

1. *Virtualization*. Software defined networking, cloud computing, and in-house virtual machine servers mean that network traffic may be in a virtual network. The soft (as in *software*) nature and dynamic topology of such networks means the traffic is not easily piped through semi-custom (or non-virtualizable) DPI hardware. Even when this is possible, the scale of such networks may overwhelm the DPI instances, making them a performance bottleneck. Ideally, DPI would be performed in COTS hardware that is also virtualizable.
2. *Fine-grained rule updates*. Current DPI offerings can make rapid rule updates as network administrators discover problems. Rule updates typically involve some actions (e.g. 'compiling' the rule set) offline before an update; as such, the updates are not performed *in-place*. The updates are usually not fast enough to be performed *while processing* traffic – i.e. there is some visible latency effect. Such fast updates could have a role in systems where rules are being *learned* automatically; that scenario would involve much more rapid rule generation/editing than done by human network administrators.
3. *Mobile DPI*. For DPI implemented on COTS hardware, load-balancing can be achieved by moving the DPI instances themselves as hardware becomes over-/under-loaded. This is doable today, but typically involves either already having a DPI instance at the destination, or bundling the DPI executable and compiled rules. Ideally, this would be lightweight and fine-grained, where a DPI instance can be partially moved downstream (in the network) – perhaps with the traffic it is already processing.

Our contribution is Elastic DPI (sensor implementation, consisting of algorithms, data-structures and implementation techniques), with several novel characteristics:

1. *Elastic resource consumption*. Most DPI implementations are built upon some form of regular expression ('regex') engine – all of which suffer from large memory or time consumption. Recent advances in data-structures and algorithms for regex processing allow us to adjust the DPI instance dynamically (both upwards and downwards), trading memory for time and vice-versa, even while processing a single packet.

1 Research on Attacks, Intrusions and Defenses, formerly Recent Advances in Intrusion Detection.

2. *Dynamic rule set*. A side effect of elasticity (in particular, JIT) is that the rule set can be edited on-the-fly, also within inspection of a single packet. Other attempts at this have involved recompiling at least part of the rule set, whereas our solution allows for incremental and in-place updates while processing traffic.
3. *Movable DPI engine*. Part of the elastic implementation involves a *domain specific virtual machine* (DSVM) for regular expressions[2]. The DSVM, along with the ability to reduce the memory footprint under EDPI, allows for the physical relocation (migration) of the DPI engine to other locations in the network, for robustness and load balancing. Such moves can be fine-grained, in which a DPI engine which has been partially run over a packet can then be migrated (perhaps after encryption and signing) with the packet itself, after which the DPI run can be completed.

A. Structure of this paper

We begin in the next section with an overview of present-day DPI, focusing on the architectures, rules for inspecting network packets, algorithms in use, implementation technologies, and perhaps most importantly: the performance trade-offs and constraints that arise from these.

The main new results – technical aspects and advances of elastic DPI – are described in the next section, with a focus on how it solves the problematic performance aspects of current solutions. This paper ends with the conclusions and future work.

The reader is expected to have a passing familiarity with regular expressions and/or finite state machines, or programming languages that use them, such as Awk, Perl, Python, etc. For a good introduction, see (Friedl, 2006), which covers both the user perspective and the under-the-hood workings of regex implementations.

2. DEEP PACKET INSPECTION TODAY

This section gives a brief overview of current DPI implementations, with a specific focus on Snort – see (Cox & Gerg, 2004) and (Cisco/SourceFIRE/Snort, 2014). While numerous other systems exist, they bear a general resemblance to Snort. Further coverage of DPI can be found in standard references such as (Varghese, 2005), (Nucci & Papagiannaki, 2009) and (Lockwood, 2008).

A. Architectural overview

Today, all academic and commercial/production DPI systems consist of the same basic architectural building blocks and interactions. These closely resemble a programming tool-chain, with a *programming language*, a *compiler*, and an *execution engine*. Indeed, current DPI systems are a form of *domain-specific language* (DSL[3]) and tools:

- *Rule sets*. These specify precisely what the system is to look for in the traffic. The rules are usually written in some domain specific language in which rule experts express interesting patterns or relationships between portions of the packet, session, flow, etc.

2 Virtual machines in this context are not new – see for example Russ Cox's work in this area (Cox R. , 2009). A comprehensive coverage of virtual machines can be found in (Smith & Nair, 2005), while (Fick, Kourie, & Watson, 2009) covers domain specific virtual machines.

3 Fowler provides a good overview of domain specific languages in (Fowler, 2010), while (Hudak, 1998) is one of the first papers explicitly treating DSL's.

- *Rule set compiler*. The human-readable rules are *compiled* (transformed) to a set of data-structures that are optimized for processing against the traffic. Compilation is usually an offline (batch) task, rerun for each change to the rule set. The compiler may support some form of incremental update, in which minor changes are much less time-consuming. Compilation is run by a network administrator, after which the new data-structures are downloaded to the matching engine.
- *Matching engine*. The precompiled rules (by now often consisting of hundreds of megabytes of data) are run against the traffic by an 'engine', in many cases consisting of specialized hardware to keep up with current network line-rates.

The following subsections consider each of these in some detail.

B. Rule sets

The rule language is a domain-specific language in which a rule engineer (a networking threat expert) can express the patterns in traffic corresponding to a threat. The best known such system is Snort[4], whose rules contain one or more clauses of the following types

- *IP addresses and ports*. A rule can apply to a specific IP address, a range of address, or a mask of addresses. Similarly, ports may be selected.
- *Flags*. A rule can apply to packets with certain flags (un)set.
- *Strings*. A specific string of bytes may be required in a packet; additionally, an offset range can be given, specifying where the bytes must appear.
- *Regex*. Regular expressions can express byte sequences that must appear in the packet, including repetitive sequences, alternative subsequences, etc.
- *Actions*. Most rules include some type of action, such as logging a message, raising an alarm, etc.

The set of such clauses making up a rule can be combined in a Boolean expression, indicating when the rule has matched.

Several observations can be made, given that the rule language is a DSL:

- The structure of all such rule languages is not only a reflection of the domain (as captured during a domain modeling phase before designing the domain specific rule language), but also of the underlying algorithmic and computational model used in the matching engine. We will return to this later as a point for optimization.
- Rules can vary dramatically in granularity, meaning that some rule authors use a one-to-one mapping between threats and rules ('coarse' rules), whereas others favor fine-grained rules in which a threat is made up of several such smaller rules. While this is often a question of style (as in other programming languages), coarser ('fatter') rules can be so complex as to also impede optimization, and therefore performance. The total number of rules in current systems is well over 1000, even with relatively coarse-grained rules.
- Often, rule sets consist of several subsets, each of which are actually written for different applications – e.g. intrusion detection rules, load-balancing rules, and quality of service rules. In the interests of performance, these application-specific

4 We occasionally refer to Snort, however, all major vendor's systems bear a close resemblance to Snort. Snort is used as an example here because it has both open source and commercial versions. (Cisco/SourceFIRE/Snort, 2014)

rules are often combined into a single large rule set for deployment in a single DPI instance. Later, we consider the performance implications of such combinations.

C. Matching engine

(We discuss the matching engine before the rule compiler, as the engine choices determine the compiler's characteristics.) The matching engine is designed with three competing engineering requirements:

1. *Speed*. The maximum bandwidth of the network is a given, and the engine must typically deal with full line-rates. In addition, only limited latency is permitted.
2. *Accuracy*. The engine must faithfully match rules. If the engine becomes overloaded with network traffic, some applications allow for lossy matching, in which some false positives or negatives are allowed.
3. *Price*. Balancing speed versus accuracy is also a price tradeoff. High speed and accuracy is computationally and memory intensive and may require semi-custom hardware.

The rule set (which is a domain specific language and its underlying domain model) in some sense dictate an abstract computational model for the engine – in some sense a domain specific virtual machine (Smith & Nair, 2005). In the case of Snort (which is representative of most DPI systems), there is a combination of two things:

1. *Finite state machines (FSM's, also known as automata)*. These are an efficient representation of string patterns and also regex's[5]. The finite state machines are run over the packet, indicating matches of regex's or strings. There are several types of FSM, with the best known being6:
   a. Deterministic: fast, predictable, but potentially massive memory consumption, or
   b. Nondeterministic: can be much slower and less predictable, but with modest memory needs. Additionally, there are efficient bit-parallel versions of these, which can be fast but require wide bit-vector machines or custom hardware.
2. *Decision trees*. Most of the other clauses in Snort rules are best compiled into decision trees, which are then evaluated by the engine in conjunction with what is found by the finite state machines.

These data-structures are loaded into the engine at startup time, and are not easily modified on-the-fly. The engine is typically so performance constrained (barely enough clock cycles for line-rate traffic) that on-the-fly optimizations and modifications are rarely done. This means any data-structure optimization is necessarily pulled into the compilation phase. One optimization that would be ideal (though not realized in current DPI implementations) is *incremental construction* of the data-structures in the match engine – only fleshing out those parts that are actually needed while processing traffic.

5 Regex's are typically compiled into FSM's using any of a number of algorithms, many of which are found in compiler textbooks such as (Cooper & Torczon, 2012). To date, the most comprehensive (and only taxonomic) treatment of such algorithms is (Watson, 1995). Dozens of regex compilation algorithms were devised in the years 1958-1995, with only modest advances since then. As a result, such a taxonomy remains a good overview of the field, despite its age.

6 Most interesting alternatives require additional hardware support, such as TCAM memory, etc. Given that one of our requirements is to push for COTS implementation, we avoid such state machines here.

In the engine, FSM's and decision trees can both be implemented along the following spectrum: pure/portable software, accelerated software (GPU[7]), FPGA[8], ASIC[9]. That spectrum is increasing in price, performance and time-to-market, but decreasing in flexibility. With the aforementioned performance and accuracy requirements, line-rate DPI engines often involve FPGA's or ASIC's, as well as highly optimized data-structures, significantly reducing flexibility.

D. Rule compilers

With the rule language defined and the matching engine's computational model selected, rule compilation is a straightforward problem of producing the correct data-structures. As with general purpose programming languages, the optimizer in a rule compiler is the most time consuming component: ideally, all of the rules are compiled together and co-optimized. Editing, adding or removing, even a single rule therefore requires an incremental recompile step and perhaps a global re-optimization step. Compilation is also very ill suited to running on the hardware hosting the matching engine (which is geared to high performance traffic stream processing) – reinforcing the notion that this is a task for the network administrator. An overview and taxonomy of algorithms involved in regex compilation can be found in (Watson, 1995), though several conferences cover new developments in such algorithms (e.g. the *International Conference on Implementations and Applications of Automata*).

E. Performance tradeoffs and constraints

The performance tradeoffs in current systems can be summarized as:

1. Current rule sets consist of >1000 rules, and growing. Rule sets often consist of subsets for different application areas. In practice, they are compiled together, yielding data-structures. An alternative solution would be to separate them and compile and deploy separate match engines – leading to at least partial duplication of data-structures.
2. Deterministic FSM for regex's: fast, can be implemented on standard hardware, but can require exponential memory against the number of rules.
    a. Cheap execution unit (can be standard CPU) for the engine.
    b. Potentially exponential memory costs.
    c. Can require exponential running time and memory for compilation, giving very slow update time when rules are edited.
3. Nondeterministic FSM for regex's: fast, but only when implemented with bit-parallelism on wide bit-vector custom hardware; memory requirements linear in the size of the rule set.
    a. Expensive execution unit consisting of custom hardware.
    b. Cheap memory for linear-sized FSM.
    c. Compilation is usually quadratic in the rule set size – still too slow for incremental updates after rule edits.
4. With both types of FSM, compilation is a network administrator task, and the resulting data-structures are relatively static once moved to the match engine. As such, *all rules* in the set are present (in compiled form), even if they are not used,

7 Graphics Processing Unit – e.g. from NVIDIA. Numerous papers have been written on network processing acceleration using GPU's.

8 Field Programmable Gate Array – a 'soft' silicon chip which is 'programmed', e.g. from Xylinx or Altera. Network processing acceleration using FPGA's is covered in (Lockwood, 2008).

9 Application-Specific Integrated Circuit – essentially a custom silicon chip. ASIC solutions to DPI are typically proprietary or secret.

they conflict, or are from different rule application areas. This can be a significant system overhead, given that practical situations see only a fraction of the total rule set in use while processing typical network traffic. (Of course, that is alleviated when DPI runs on a system with virtual memory and not all data-structures are in physical memory at a time.)

5. Many rule languages use Perl-compatible regex's (PCRE's). Pure regex's compile and optimize very well for FSM's, but PCRE's contain numerous features (such as backtracking, greedy operators, etc.) that impede the match engine's implementation and performance. As a result, rule writers shy away from regex unless absolutely needed, preferring to use the other rule clauses – making the rules very heterogeneous and difficult to optimize (Friedl, 2006).

These tradeoffs have some DPI-system-wide implications:

- DPI is not suited to a virtualized environment:
  - Deterministic FSM: the match engine uses COTS hardware, but with high memory consumption (incompatible with virtualization, in which the virtual machines are expected to not appropriate all resources).
  - Nondeterministic FSM: the match engine uses custom hardware not found in a virtual environment.
- For similar reasons, it is not movable, even in a virtualized environment. Either the system is consuming large amounts of memory (making it costly to move), or using custom hardware (impossible to move).
- In the deterministic FSM scenario (the most common one in practice), rule set edits do not allow for incremental compilation (where only the impacted parts of the data-structures change). The illusion of incremental compilation is given by some systems – though this is accomplished by compiling a separate set of tables for the rules that have changed, thereby further raising system overheads as those new FSM's must also be run over the packet.

3. ELASTIC DPI

Elastic DPI uses recent advances in algorithms and data-structures (for regex's and FSM's) to provide solutions to the problems sketched in the last section.

A. Simplifying the rule language

As mentioned earlier, two of the performance penalties in DPI systems are the use of: elaborate rule structures (e.g. thanks to the different clause types in Snort rules) that require decision trees, and regex's, specifically PCRE.

In EDPI, we have chosen to only use regex's and actions in rules:

- Regex's can be used to express IP address, port and flag aspects that must match. In the match engine, the regex is run against the entire packet, including any headers and trailers containing such information.

- Strings, including their offsets within the packet, are written as regex's. Indeed, also in Snort string clauses are actually a form of regex in a different notation, as offsets are readily written in regex's as well using counting quantifiers (Friedl, 2006).
- The dialect of regex's chosen is much purer than PCRE, leaving out the computationally heavy backtracking and capture mechanisms[10]. In return, our dialect allows for exotic extended regex operators such intersection, negation, shuffle, cut, etc., which gain more than enough expressive power. Those operators allow us to directly combine what would previously have been multiple clauses and Boolean expression in the rule, yielding a single regex for the rule. In fact, the rule compiler merges all of the rules' regex's into a single large regex (of the form Expression1 | Expression2 | …). See (Brzozowski, 1964) for more on compiling extended regular expressions to FSM's.

This unification of rule notation, and underlying computational formalism is both elegant (rule writers can think in one formalism) and also computational efficient, as discussed below.

B. Speed versus memory

In this section, we detail three groups of algorithmic, implementation, and optimization techniques that, independently, are already significant advances, but together are key enablers for Elastic DPI.

1) On-demand construction

As mentioned earlier, most current compilers from regex's to FSM's are *batch compilers*, meaning they compile the entire regex (in our case, the composite regex consisting of all rules) into a single massive FSM without regard to which parts of the FSM will actually be used. At run time, usually only a fraction of the FSM is used (because not all DPI rules match over the traffic) – imposing an unfortunate system overhead. Ideally, we would like to only build those parts actually in-use – a kind of *hot state/path* optimization. Such algorithms have been known since the early days of regex and FSM implementation (Thompson, 1968). In DPI systems, for performance the match engine is often running on hardware highly tuned for the matching process, or COTS hardware fully devoted to DPI – not the environment in which to run the compiler or perform such on-the-fly construction. EDPI rests on a new class of algorithms and data-structures that are fast enough for on-the-fly construction and optimization while simultaneously performing matching.

A regex/FSM co-representation is presented in (Watson, Frishert, & Cleophas, 2005) and (Frishert & Watson, 2004), and we have extended that work for EDPI. The algorithm (our *continuation engine*) takes two parameters: a regular expression to be matched, and an input byte of the traffic. It returns another regular expression, known as the *continuation*. Essentially, the continuation[11] encodes the 'remainder' of the pattern to be matched in the input, and computing the continuation is equivalent to taking a transition in an FSM corresponding to the regex. Continuations date to Janusz Brzozowski's original work in this area in the late 1950's (Brzozowski, 1964), though the algorithm has been oddly underused in compilers and other applications.

10 Those mechanisms are not only computationally heavy, but also nondeterministic, making them problematic when making real-time performance promises, as are required in DPI.

11 Also known in the literature as the derivative.

In our *continuation engine (CE)*, we have made two important optimizations over Brzozowski's original work:

1. The continuations (over all possible input bytes) of a regex share most subexpressions with the original regex. As such, we can apply *common subexpression sharing* – a well-known technique in compilers (Cooper & Torczon, 2012) – to dramatically reduce space. In addition to this effect in continuations, many rules in a rule-set share subexpressions (Massicotte & Labiche, 2011) – leading to further savings under EDPI.
2. As continuations are generated (by processing traffic), they are *cached* in lookup tables and do not need to be recomputed. Whenever a continuation is needed which has not yet been computed, the cache entry is empty and it is computed once-off relatively cheaply.

These two techniques allow us to process the input traffic (taking transitions in FSM terms, but actually computing continuations in CE terms) while effectively only building those parts of the FSM that are actually in-use.

There are two performance implications:

1. *Startup costs*. With an initially empty cache, every traffic byte processed triggers a continuation computation in the CE. This continues until a the cache consists of the 'hot states/path' – a critical mass of reusable cache entries is reached. In many DPI applications, this occurs within the first megabyte of input traffic. With suitable traffic profiles, such *cache preloading* is something that can be done offline, saving startup costs.
2. *Processing costs*. Most traffic bytes processed result in a cache lookup – essentially an FSM transition, making this as fast as any other FSM-based solution, providing the cache implementation is highly tuned. Occasionally, a cache miss occurs, giving some overhead in building a new continuation and cache entry. With buffer management in EDPI, the latency from a cache miss is smoothed, and this does not cause any throughput or latency issues. In the worst case, the CE can 'cache thrash', consuming as much memory as a traditional DPI system and having some startup latency[12].

These caching techniques were explored in (Thompson, 1968), but became less interesting as available memory grew. More recently, the performance has been quantified in (Ngassam, Watson, & Kourie, 2006).

These performance characteristics make EDPI competitive with traditional DPI in practice[13], also because hot path optimization (computing only those FSM parts that are actually in-use) reduces total memory load and improves processor cache (not to be confused with CE cache) utilization. To contrast, EDPI can have as little as a few kilobytes in use (representing the regex rule set and some caching) whereas traditional DPI has megabytes of memory in use at a given time for a comparable rule set.

12 This worst-case scenario would amount to pulling some of the compilation costs of traditional DPI into the match engine area of the system, since one of EDPI's architectural advantages is to support compilation in the match engine via the CE.

13 despite the overhead of the continuation engine.

2) Restricting memory
Like other caches in computational systems (e.g. memory and disk caches), the CE's cache can be flushed without errors, but with a computational penalty for rebuilding. The match engine's memory budget may be reduced during processing (that is, in a 'hot/live' system). The CE will discard the cache entries, leaving the entries 'undefined' and triggering recomputing the continuations later. In memory constrained systems, such cache flushing can also be done selectively – when memory is full and a new continuation is being constructed, the least-recently used cache entry is flushed. Least-recently used is tracked using the time-stamp (clock) counter present on most modern processors.

Flushing some cache entries frees up memory used for the transitions, but additional memory may also be freed. In particular, the representation of the additional derivatives (the continuations) consumes memory – even with common subexpression sharing. The CE marks the original regex (as opposed to the continuations, which are *derived* from the original regex), and can discard the non-original regex's (the continuations) when reclaiming memory, since the continuations are easily reconstructed by the CE. This is particularly useful for reducing the state and regex set to a *kernel* that can then be moved to a new compute location (perhaps with the packet being processed), the CE then reconstructing the cache at the new location.

3) Approximate EDPI
Using cache management techniques similar to those in the previous section, EDPI also allows for *approximate DPI* (also known as lossy matching) in very memory constrained systems. That is not presented here, but may be found in (Watson, Kourie, Ketcha, Strauss, & Cleophas, 2006).

4) Stretching and jamming
Stretching and jamming are two additional optimization techniques (they are the reverse of each other) that can move the FSM along the speed versus memory axis (de Beijer, Cleophas, Kourie, & Watson, 2010). DPI typically processes the traffic an 8-bit byte at a time, implying that regex's are also expressed as bytes, and the FSM is represented with transitions on bytes. Stretching allows us to process 4-bit nybbles at a time – each byte of the traffic is separated into a high- and a low-order nybble; the high-order nybble is processed through an FSM transition, followed by the low-order nybble. (The order of the two nybbles can be swapped for processing, giving an endianness optimization which sometimes yields faster processing – though this has not yet been quantified.)

Splitting traffic bytes into nybbles is done on-the-fly – a very fast operation on modern processors. The FSM, however, needs some preparation, with each transition on a byte being stretched into two transitions on the corresponding pair of nybbles. The FSM must typically also be made deterministic again: an FSM state with transitions on bytes b_0 (= nybbles n_{0high} and n_{0low}) and b_1 (= nybbles n_{1high} and n_{1low}) is deterministic when b_0 and b_1 are different, but in the stretched transitions we may have $n_{0high} = n_{1high}$, making the FSM nondeterministic. Stretching doubles the number of steps required to process the traffic, so what does stretching gain us? The narrower alphabet (4-bit nybbles) yields much narrower transition tables: 16 columns now compared to the 256 columns for the full 8-bit byte alphabet, and this is often a significant space savings despite doubling the number of transitions and adding new states. Jamming is the opposite transformation, changing the alphabet from 8-bit bytes to 16-bit short-words. This equates to processing two adjacent traffic bytes at a time by merging two subsequent transitions – halving of the processing time[14]. This speed win is traded against the fact the transition tables may now have $2^{16} = 65536$ columns compared to 256 – a massive increase, despite the halving of transitions and reduced number of states. Both stretching and jamming may be applied again, respectively yielding transitions on 2-bit half-nybbles or on 32-bit words, etc.

The optimization sweet spot for stretching and jamming is difficult to find a priori, though some benchmarks are presented in (de Beijer, 2004). The CE in EDPI allows us to dynamically stretch and jam, by rebuilding the FSM in stretched or jammed form (under the hood, the FSM is actually modified in places where it has already been built), based on speed or memory requirements at that moment. A key future optimization is for the CE to locally stretch and jam – an optimization for only part of the regex and FSM where it may be particularly profitable.

C. Incremental rule set modifications

The co-representation of the regex set with the FSM, along with CE, has a critical side-effect: the regex's may be edited on-the-fly. Parts of the rule regex's (which are, in turn, parts of the combined regex) can be added, deleted, or modified. The CE observes this and discards those parts of the continuation data-structures that are no longer valid; they are then rebuilt as needed. This elegant solution brings incremental rule set modification to EDPI, even in running ('hot') systems. Of course, if the modified regex is to be rerun on the packet, this requires backtracking to the beginning of the packet – a relatively small penalty for incremental rules.

D. Location flexibility

The ability to move a running DPI match engine to another machine is a natural side effect of two EDPI aspects:

- EDPI can be virtualized (thanks to more virtualization-friendly memory consumption), and the resulting virtual stack is easily migrated within existing hypervisor products.
- Even without virtualizing EDPI, the data-structures are shrinkable to a kernel – as mentioned earlier. Without harming the matching process, the data-structures can be shrunk to the size of the original regex set (usually measured in *kilobytes*), which can then be moved along with the partially-processed packets, and restarted at a new location. This allows for a form of data-flow architecture with the computation (EDPI instance consisting of the CE) traveling with the data to more appropriate locations (in terms of load balancing, for example).

4. CONCLUSIONS AND FUTURE WORK

In this paper, we gave an overview of the current state of deep packet inspection (DPI) systems, with a particular focus on their engineering tradeoffs and potential performance problems in an increasingly virtualized environment. Against that backdrop, we presented Elastic DPI as a new approach with some key differentiators:

14 There are some nontrivial issues that we do not discuss here, for example: a packet consisting of an odd number of bytes must be specially handled with the last byte which has nothing to jam with.

1. The amount of memory in use can be grown or shrunk dynamically, trading speed against memory consumption. This is key to enabling virtualization.
2. The set of DPI rules may be edited on-the-fly, allowing for highly dynamic systems.
3. The actual DPI engine can be sufficiently shrunk (in service, while processing) to be moved efficiently to another computing resource.
4. The domain specific language for expressing rules can be made uniform, in terms of extended regular expressions which capture all of the presently used clauses in other rule languages.

These aspects are significant advances in this field and are made possible by recent advances in the algorithmics of pattern matching, as well as new implementation techniques.
The primary direction for future work is to integrate and measure the Elastic DPI system in a production environment, yielding benchmarking data. Additional foci are on parallelism in the Elastic DPI algorithms – especially given the current trends towards multicore hardware.

Acknowlegements:

The anonymous referees provided particularly strong, useful and well thought out feedback, which is appreciated.

BIBLIOGRAPHY:

Watson, B. W. (1995). *Taxonomies and Toolkits of Regular Language Algorithms* (Ph.D dissertation ed.). Eindhoven: Eindhoven University of Technology.

Thompson, K. (1968). Regular expression search algorithm. *Communications of the ACM , 11* (6), 419-422.

Brzozowski, J. (1964). Derivatives of Regular Expressions. *Journal of the ACM , 11* (4), 481-494.

Varghese, G. (2005). *Network Algorithmics*. Morgan Kaufmann.

Watson, B. W., Frishert, M., & Cleophas, L. (2005). Combining Regular Expressions With Near-Optimal Automata. In A. Arppe, L. Carlson, K. Linden, J. Piitulainen, M. Suominen, M. Vainio, et al., & A. Copestake (Ed.), *Inquiries Into Words, Constraints And Contexts. Festschrift for Kimmo Koskenniemi on his 60th Birthday* (Online ed., pp. 163-171). Stanford: CSLI Studies in Computational Linguistics, Stanford University.

Cox, K., & Gerg, C. (2004). *Managing Security with Snort and IDS Tools*. O'Reilly.

Smith, J. E., & Nair, R. (2005). *Virtual Machines*. Morgan Kaufmann.

Nucci, A., & Papagiannaki, K. (2009). *Design, Mesurement and Management of Large-Scale IP Networks*. Cambridge University Press.

de Beijer, N., Cleophas, L., Kourie, D., & Watson, B. W. (2010). Improving Automata Efficiency by Stretching and Jamming. In J. Holub, & J. Zdarek (Ed.), *Prague Stringology Conference* (pp. 9-24). Prague: Czech Technical University.

Watson, B. W., Kourie, D., Ketcha, E., Strauss, T., & Cleophas, L. (2006). Efficient Automata Constructions and Approximate Automata. In J. Holub (Ed.), *Prague Stringology Conference* (pp. 100-107). Prague: Czech Technical University.

Cox, R. (2009, December). *Regular Expression Matching: the Virtual Machine Approach*. Retrieved from Regexp2: http://swtch.com/~rsc/regexp/regexp2.html

Cooper, K. D., & Torczon, L. (2012). *Engineering a Compiler* (Second ed.). Morgan Kaufmann.

Frishert, M., & Watson, B. W. (2004). Combining Regular Expressions with Near-Optimal Brzozowski Automata. In K. Salomaa (Ed.), *Conference on Implementations and Applications of Automata*. Kingston: Queen's University Press.

Fick, D., Kourie, D. G., & Watson, B. W. (2009). A Virtual Machine Framework for Constructing Domain Specific Languages. *IEE Proceedings - Software*, 3 (1).

Ngassam, E. K., Watson, B. W., & Kourie, D. G. (2006). Performance of Hardcoded Finite Automata. *Software - Practice & Experience*, 35 (5), 525-538.

Ngassam, E. K., Watson, B. W., & Kourie, D. G. (2006). Dynamic Allocation of Finite Automata States for Fast String Recognition. *International Journal of Foundations of Computer Science*, 17 (6), 1307-1323.

Massicotte, F., & Labiche, Y. (2011). An Analysis of Signature Overlaps in Intrusion Detection Systems. *41st International Conference on Dependable Systems & Networks* (pp. 109-120). Hong Kong: IEEE/IFIP.

Lockwood, J. W. (2008). Network Packet Processing in Reconfigurable Hardware. In S. Hauck, & A. Dehon, *Reconfigurable Computing* (pp. 753-778). Morgan Kaufmann.

Friedl, J. E. (2006). *Mastering Regular Expressions* (Third ed.). O'Reilly.

Cisco/SourceFIRE/Snort. (2014). *Snort Homepage*. Retrieved March 17, 2014, from Snort: www.snort.org

Fowler, M. (2010). *Domain Specific Languages*. Addison-Wesley.

Hudak, P. (1998). Domain Specific Languages. In P. H. Salas, *Handbook of Programming Languages: Little Languages and Tools* (Vol. 3, pp. 39-60). MacMillan.

de Beijer, N. (2004). *Stretching and Jamming of Automata* (M.Sc thesis ed.). Eindhoven: Eindhoven University of Technology.