Network DVR: A Programmable Framework for Application-Aware Trace Collection

Chia-Wei Chang[1], Alexandre Gerber[2], Bill Lin[1]
Subhabrata Sen[2] and Oliver Spatscheck[2]

[1] University of California, San Diego, La Jolla, CA
[2] AT&T Labs-Research, Florham Park, NJ

Abstract. Network traces are essential for a wide range of network applications, including traffic analysis, network measurement, performance monitoring, and security analysis. Existing capture tools do not have sufficient built-in intelligence to understand these application requirements. Consequently, they are forced to collect *all* packet traces that might be useful at the finest granularity to meet a certain level of accuracy requirement. It is up to the network applications to process the per-flow traffic statistics and extract meaningful information. But for a number of applications, it is much more efficient to record packet sequences for flows that match some application-specific signatures, specified using for example regular expressions. A basic approach is to begin memory-copy (recording) when the first character of a regular expression is matched. However, often times, a matching eventually fails, thus consuming unnecessary memory resources during the interim. In this paper, we present a programmable *application-aware* triggered trace collection system called Network DVR that performs precisely the function of packet content recording based on user-specified trigger signatures. This in turn significantly reduces the number of memory copies that the system has to consume for valid trace collection, which has been shown previously as a key indicator of system performance [8]. We evaluated our Network DVR implementation on a practical application using 10 real datasets that were gathered from a large enterprise Internet gateway. In comparison to the basic approach in which the memory-copy starts immediately upon the first character match without *triggered-recording*, Network DVR was able to reduce the amount of memory-copies by a factor of over 500x on average across the 10 datasets and over 800x in the best case.

1 Introduction

Accurate trace collection of network traffic is the foundation of a wide range of network monitoring tasks. Traditionally, packet capture tools (e.g., TCPdump [1], Ethereal [2], Libpcap [3], and WinPcap [4]) are primarily focused on collecting and reconstructing packet sequences for flows by matching packets against simple packet header rules, such as the source/destination IP addresses, the port numbers, or the transmission protocol. The collected packets are often delivered to remote servers where monitoring and management applications can perform post-processing, as depicted in Fig. 1. Although the traditional network monitoring architecture has had some success in offering comprehensive insights about network traffic, the scalability of this architecture is limited

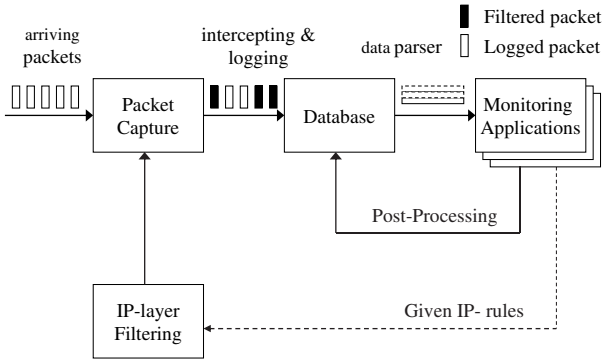

Fig. 1. Simplified traditional network monitoring architecture.

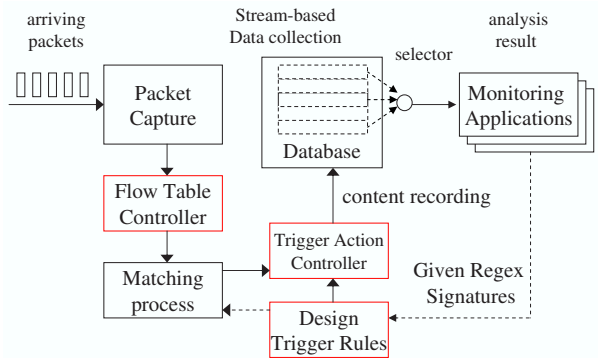

Fig. 2. Programmable network monitoring architecture.

in practice. In today's high-speed network, especially in core networks, the amount of traffic can be immense, with possibly millions of flows. Therefore, recording all packet traces in details is prohibitive in most cases.

In this paper, we present a programmable *application-aware* trace collection architecture called *Network DVR* that can perform the *recording* of *packet contents* highly selectively by matching captured packets against application-specific signatures. This is depicted in Fig. 2. The name Network DVR is loosely analogous to a digital video recorder for television that can be intelligently programmed to record discriminately. Our design of Network DVR is based on a novel concept of *triggered-recording* that allows a user to flexibly define rules for triggering the *start* and *termination* of packet content recordings. In particular, Network DVR can be programmed to only begin recording when some *start rule* has been matched, which can significantly reduce the likelihood of recording matchings that will eventually fail. We have applied our Network DVR approach on a practical example application by using 10 real datasets that were gathered from a large enterprise Internet gateway. Our evaluations show that Network

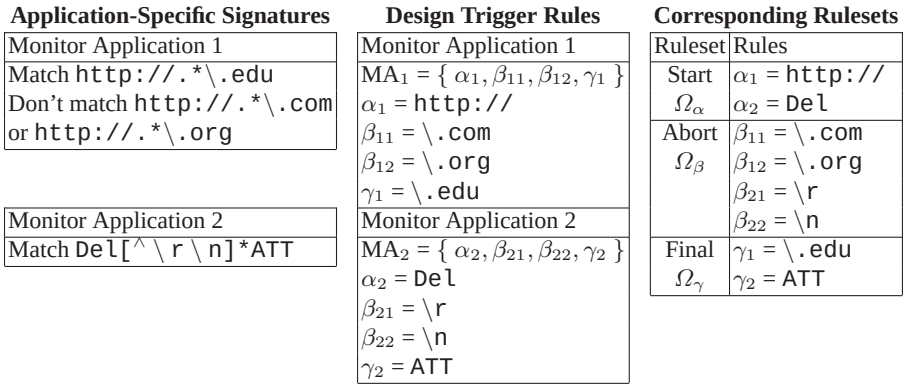

Fig. 3. Construction of trigger rulesets.

DVR can dramatically reduce the amount of memory-copies by a factor of over 500x on average across the 10 datasets and over 800x in the best case.

The rest of the paper is organized as follows. Section 2 presents a high-level overview of our proposed triggered trace collection approach called Network DVR. Section 3 presents evaluation results. Section 4 discusses related work. Finally, Section 5 concludes the paper.

2 Proposed Triggered Trace Collection Concept

In this section, we present the concept of application-aware *triggered* trace collection, which aims to record packet contents based on application-specific signatures that control when the recordings should *start* and *abort*. These signatures are captured as regular expressions. Although regular expression matching has been widely used for intrusion detection, regular expression matching is used differently in our context to trigger the start and termination of packet content recordings. Specifically, we define three trigger rulesets, one that defines when the recording module should *start* recording, one that defines when the recording should *abort*, and one that defines if a recording is a valid *final match*. These three regular expression rulesets are respectively called the *start* ruleset (Ω_α), the *abort* ruleset (Ω_β), and the *final match* ruleset (Ω_γ)). Incoming traffic can be matched against these trigger rulesets by means of a deterministic finite automaton (DFA). In particular, each *accept state* will correspond to one or more rules matched from these three rulesets. Accordingly, the corresponding *start*, *abort*, and/or *finalized* messages would be triggered to control the recording behavior, depending to which rulesets the matching rules belong. To simplify the presentation, we will assume a conventional DFA representation for these rules. However, in general, our framework can make use of any state-of-the-art DFA variants [21, 11, 20, 10, 7, 16, 9] for efficient representation and matching.

In the following, Section 2.1 describes how trigger rulesets are constructed, Section 2.2 presents the high-level triggered trace collection procedure, and Section 2.3 presents our memory management scheme.

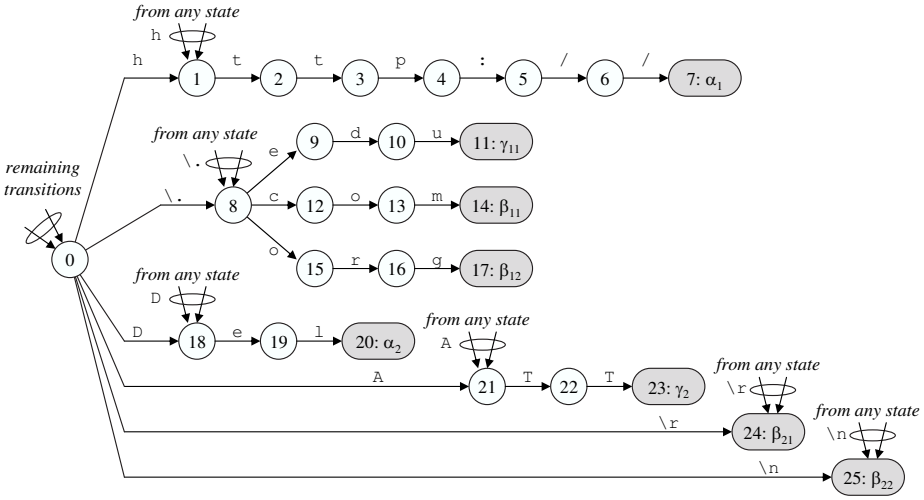

Fig. 4. Compiled DFA.

2.1 Trigger Rulesets Construction

An essential part of trigger trace collection is the trigger ruleset construction (i.e., to build the *start* (α), *abort* (β) and *final match* (γ) rules, which control when to activate, abort, or finalize the trace collection). Naturally, these trigger rules can be defined directly by the monitoring applications, but these rules can also conceivably be automatically generated in a systematical way from a set of application signatures.

Consider the examples shown in Fig. 3. The first monitoring application is for tracking valid URLs in the form of "http://.*\.edu" to find say the "top 100" sites of educational institutions on the web, excluding non-educational ".com" and ".org" sites. The second monitoring application aims to identify the characters except carriage returns or new line characters between the two sub-patterns "Del" and "ATT" (this example is truncated from a Snort ftp rule). Each monitoring application signature generates its individual trigger rule set and each signature is mainly split into two parts, a prefix part and a suffix part. Generally, the prefix part serves as a start condition for the recording while the suffix part serves as a finish condition for the recording. In addition, there may be conditions that will allow us to abort a recording. For example, the complement syntax, "[^]", is used to indicate the characters that should be *excluded* from a match. In such cases, these exclusive characters can serve as abort conditions for the recording (e.g., "\r" and "\n" in application 2). In other cases, the abort conditions may be explicitly specified (e.g., "\.com" and "\.org" in application 1).

2.2 Triggered Trace Collection Procedure

Given the trigger rulesets, we construct a DFA to perform the matching. The DFA for the example depicted in Fig. 3 is shown in Fig. 4. The *accepting* states are shown in the shaded ovals (i.e., States 7, 11, 14, 17, 20, 23, 24, 25) with the corresponding matching

rules specified. In order to implement cross-packet inspection, the matching module needs to remember for each flow the *last* state that the matching module visited in the DFA to serve as the starting state for the next packet for the same flow. To remember the last state visited for each flow, we use a flow state table where each entry corresponds to a flow, and the value of the entry is the "last visited" state of the flow in DFA. The flow state table can be implemented as a hash table and a new hashed flow can be initialized to the initial state (e.g., State 0).

With every incoming packet, the matching module will find its corresponding last visited state in the flow state table and adjust the matching procedure to start at the right position of the packet payload based on the header information. Each symbol is read sequentially from the packet payload and transferred to the next state by following the DFA structure. If it matches a *start* rule, the symbol-copy/recording behavior will be activated, also the matching index and the recording-begin memory position will be logged. If this flow is under the recording process, for each symbol, the memory allocator will unlink one memory cell from the free memory cell list, copy the symbol, link it to the temporary recording strings. After a corresponding *abort* rule is matched, this temporary recording string will be recycled and connected back to the free list of memory cells. On the other hand, if a corresponding *final match* rule is matched, the temporary recording string will be appended to a flush queue for writing to disk, and its recording-end memory position will be logged. Here, "corresponding" means for the trigger rules either in *start*, *abort*, or *final match* ruleset that come from the same application-specific signature (e.g., have the same matching index).

In particular, for each application-specific signature $\text{MA}_i = \{\alpha_i, \beta_{i1} \ldots \beta_{ij}, \gamma_{i1} \ldots \gamma_{ik}\}$, there is a corresponding variable v_i that gets set where its corresponding start rule has been matched. This is depicted in Fig. 5. Upon encountering an abort or a final match rule, we check if there is an active recording for this rule by testing v_i. If it is set, then the corresponding actions for abort or final match are to reset v_i, and the recording is either aborted or flushed, respectively. Since we support recordings on a per-flow basis, we must keep track of a separate set of these v_i variables for each flow. This can be dynamically managed using a hash table. In particular, to test if v_i is set for flow f, we can perform a hash lookup on the key $f{:}v_i$, where the key is constructed by combining the flow ID and the variable name. To set v_i for flow f, we can perform a hash insert (or lookup-then-insert) with the key $f{:}v_i$. Finally, to reset v_i for flow f, we can perform a hash delete with the same key $f{:}v_i$. Since we are not storing a value in this hash table, this hash table can as well be efficiently using a counting Bloom filter [22].

Note that multiple recordings may be triggered for a single flow if multiple *start* rules have been matched. Therefore, the worst-case bound on memory bandwidth/processing time is $O(N)$, where N is the number of given total application-specific signatures[3]. Fortunately, instead of having multiple recording strings for each matched pattern in the *start* ruleset, we use a single aggregated recording string for each flow to guarantee there is always one memory copy for each incoming symbol. By logging the recording-begin/end memory positions of each valid matching result in

[3] Here the memory bandwidth requirement is expressed in terms of the number of memory operations (e.g., copies) to be performed for each input character processed.

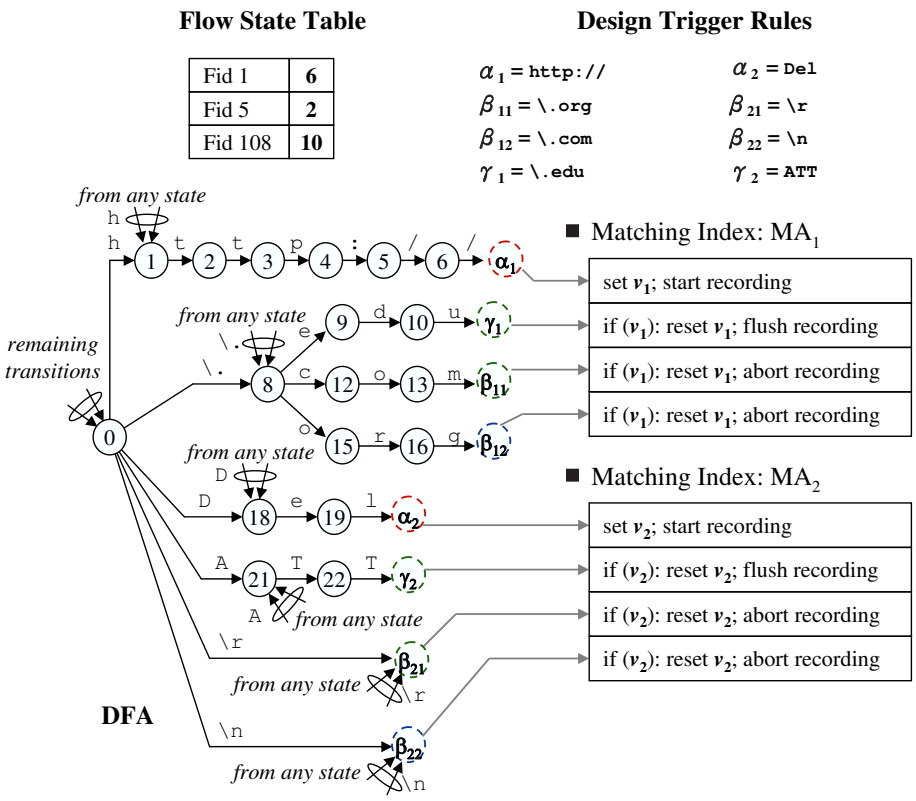

Fig. 5. Flow state table, DFA, and actions.

the aggregated recording string, the system can output all recorded matching strings for each application-specific signature.

2.3 Constant Time Memory Allocation Structure

Initially, the memory allocator maintains a free-list of memory cells as a linked list, and each unit of memory cell can hold one byte (for a character) and a pointer. Depending on the implementation, it may be more memory efficient to have the memory allocation granularity be multiple bytes for each memory cell instead of one. For simplicity of presentation, we will assume each memory cell holds one symbol. For recording, the recording module stores the string being recorded as a linked list of symbols as well. When the recording string needs more memory, the memory allocator unlinks cells at the front of the free list sequentially and puts them to the tail of the linked list corresponding to the recording string (constant time memory allocation). It can connect the entry of the bitmap table to the recording string by setting a pointer at the bitmap table entry to the head of the corresponding linked list.

Suppose this recording string is confirmed as a valid recording. The memory allocator can simply put the head of this linked list to an output queue for flushing to disk and

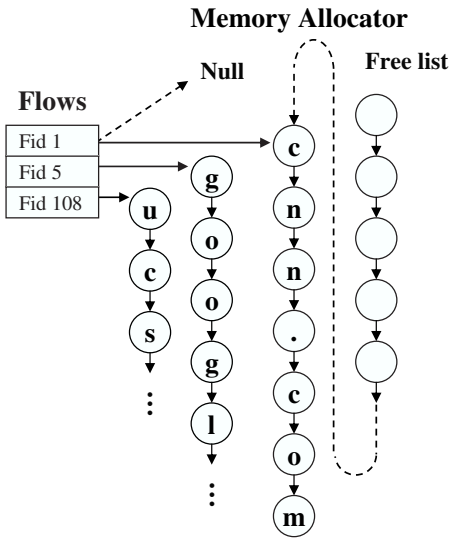

Fig. 6. Trace collection module.

reset the flow state entry pointer to NULL. After written to disk, it can be linked back to the tail of the free-list (constant time recovery of the memory cells). If a recording is aborted (e.g. Fid 1 records "cnn.com" which is not ended by ".edu" in Fig. 6), the memory allocator will directly set the flow state entry pointer to NULL, and link this linked list to the tail of the free-list, which again is a constant time operation.

3 Evaluation

In this section, we evaluate the proposed Network DVR framework. We show the scalability of the modified regular expression extraction module, which uses an application-aware trigger mechanism to control the recording process. We then demonstrate the performance that Network DVR can achieve by measuring the number of memory copies consumed during the recording process. Indeed, for an initial evaluation, the number of memory copies is a good proxy for the future performance of a real time system.

3.1 Simulation Setup

We gathered real data traces from a couple of servers for a day by using Gigascope [8]. These traces were collected on a trunk with four 1-Gigabit Ethernet links at a large enterprise Internet gateway by using IP-filtering for specific IP-groups in which we are interested, and we used these traces to verify our application-aware data collection approach. The collected traces were then partitioned into 10 datasets based on the packets collected every 60 minutes. Each of these datasets has approximately 3,500 flows. For each dataset, given the application-specific signatures, we replay the complete trace and calculate the number of memory copies that Network DVR needs and compare the result to a basic approach in which the recording starts when the first character of a regular expression is matched. We have employed an efficient public domain DFA variant

Table 1. Evaluate the number of memory copies needed for Snort rules.

	data	basic	netDVR	actual	reduction	overhead
Set 1	7.00×10^8	3.13×10^7	7.06×10^4	4.69×10^4	444.04	1.51
Set 2	6.50×10^8	2.89×10^7	6.39×10^4	4.38×10^4	451.64	1.46
Set 3	6.60×10^8	3.09×10^7	7.22×10^4	4.63×10^4	427.74	1.56
Set 4	6.30×10^8	2.84×10^7	6.96×10^4	4.68×10^4	407.39	1.49
Set 5	6.10×10^8	2.74×10^7	6.83×10^4	4.50×10^4	401.05	1.52
Set 6	7.90×10^8	3.60×10^7	5.07×10^4	3.16×10^4	709.49	1.60
Set 7	7.60×10^8	3.53×10^7	7.77×10^4	5.12×10^4	453.73	1.52
Set 8	7.60×10^8	3.25×10^7	8.24×10^4	6.44×10^4	394.35	1.28
Set 9	8.20×10^8	3.44×10^7	5.66×10^4	4.08×10^4	607.21	1.39
Set 10	9.00×10^8	3.77×10^7	4.66×10^4	3.09×10^4	808.92	1.51
				average	510.56	1.48
				max	808.92	1.60

provided by Becchi and Crowley [7] to serve as the matching module in Network DVR, but we note that other efficient DFA variants [21, 11, 20, 10, 16, 9] may be used in our framework as well.

For our evaluation, we considered a practical application in which Network DVR is used to record the details of matched results for an intrusion detection system (IDS). To perform this recording task without our trigger concept, current IDSs would be forced to begin copying symbols to memory when the first symbol of a regular expression is matched. Here, we chose one category of signatures from the Snort 2007 signature set [12] for our evaluation. In particular, we considered the ftp signatures (consisting of 58 regular expressions) for our experiment to evaluate the amount of unnecessary memory copies that Network DVR can reduce by using the proposed triggered-recording concept. For each signature, we manually decomposed the signature into start, abort, and final match rules. Specifically, we used the prefix of each signature as a start rule, the suffix of each signature as a final match rule, and any exclusion characters as abort rules.

3.2 System Performance Comparison on Memory Copy Times

Table 1 summarizes the results for the 10 datasets considered in terms of memory copies. The column labeled `data` shows the size of the total incoming symbols which is replayed by using the real traces collected by Gigascope [8] using IP-filtering only (unaware of the application). The column labeled `basic` shows the results for a basic approach that begins copying symbols to memory when the first character of a regular expression is matched. The column labeled `netDVR` shows the results using our triggered-recording approach. The column labeled `actual` shows the total number of times actual memory copies were needed for successful matches. The column labeled `reduction` shows the reduction factor in memory copies that `netDVR` can achieve relative to the `basic` approach (i.e., `reduction` = `basic` / `netDVR`). The last column labeled `overhead` shows the overhead factor in memory copies that `netDVR` incurs relative to the `actual` memory copies for successful matches (i.e., `overhead` = `netDVR` / `actual`).

As can be seen in Table 1, our Network-DVR approach can achieve a reduction factor of over 500x across the 10 datasets and over 800x in the best case (i.e., for Set 10). This reduction is achieved by only activating the recording when a start rule has been matched. However, even with this start ruleset pre-filtering, it may still be possible that a matching eventually fails. This accounts for the difference in results between the `netDVR` column and the `actual` column. However, this overhead is only a factor of 1.48x on average with 1.6x in the worst case.

4 Related Work

Packet recordings gathered by core routers provide valuable coarse-granularity traffic information for a variety of measurement-related applications. However, because of the large volumes of traffic, filtering unnecessary data becomes an important issue. Packet filtering at the IP-header level is among the first techniques applied to solve this issue [13, 14]. BLINC [15], which observes and identifies patterns of host behavior at the transport layer, is designed for traffic classification. Deep packet inspection (DPI) [7, 9] methods, which identify and classify traffic based on a signature database that includes information extracted from the data part of a packet, allows for finer control than classification based only on header information. These approaches are what we describe as the "basic" implementations, which start copying symbols to memory when the first symbol of a regular expression is matched in order to output the details of the matching results. Our approach performs much better by triggering the memory copies later.

ProgME [17] introduces flowsets based on packet headers for defining aggregates in the context of network measurements (specifying counting). ATMEN, a triggered measurement infrastructure to communicate and coordinate across various administrative entities, is proposed in [18] to reduce wasted measurements by judiciously reusing measurements along three axes: spatial, temporal, and application. Another approach, in which any particular application can express the classes of traffic of its interest, is proposed in [19] for application-specific flow sampling in many monitoring applications, such as SNORT [12], BLINC [15]. Although the above methods make flow recording feasible by using traffic classification, faster deep packet inspection, hardware pre-matching, concept of flowsets or application-aware packet sampling, none of them perform content-state-aware filtering of packet content for a given monitoring application.

5 Conclusion

In this paper, we presented Network DVR, a framework for a programmable content-aware packet trace collection system. Our main contribution is programmable recording framework that can be programmed to record packet traces that are relevant to a given monitoring application. Our framework employs a trigger recording concept that defers recording until some start conditions have been matched, which significantly minimizes the number of memory copies that the system consumes for valid trace collection (a key indicator of performance). Modules with knowledge about application requirements enable Network DVR to collect traffic data in accordance to the application at hand. Our evaluation using real datasets from a large enterprise Internet gateway shows that Network DVR can collect relevant packet traces effectively in a concise manner and also reduce the amount of memory copies dramatically. Given these encouraging results,

we plan to develop a real-time implementation of Network DVR with larger sets of application signatures to better quantify the actual performance improvement of our proposed approach.

References

[1] V. Jacobson, V. C. Leres, and S. Mccanne. The TCPdump Manual Page. *Lawrence Berkeley Laboratory*, Berkeley, CA, June 1989.

[2] A. D. Orebaugh and G. Ramirez. Ethereal Packet Sniffing. Syngress Publishing, 2004

[3] Libpcap. http://www.tcpdump.org/#documentation.

[4] WinPcap: The Windows Packet Capture Library. http://www.winpcap.org/

[5] C. Estan and G. Varghese. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Trans. Computer Systems*, 2003.

[6] PCRE: Perl Compatible Regular Expressions http://www.pcre.org/

[7] M. Becchi, and P. Crowley. A hybrid finite automaton for practical deep packet inspection. *CoNEXT*, 2007.

[8] C. Cranor, T. Johnson, O. Spataschek, V. Shkapenyuk Gigascope: a stream database for network applications. *ACM SIGMOD*, 2003.

[9] R. Smith, C. Estan, S. Jha, and S. Kong. Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. *ACM SIGCOMM*, 2008.

[10] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. *ACM ANCS*, 2006.

[11] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. *ACM SIGCOMM*, 2006.

[12] Snort. http://www.snort.org.

[13] A. W. Moore and D. Zuev. Traffic classification using bayesian analysis techniques. *SIGMETRICS*, 2005.

[14] S. Kundu, S. Pal, K. Basu, and S. Das. Fast classification and estimation of Internet traffic flows. *PAM*, 2007.

[15] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. BLINC: multilevel traffic classification in the dark. *ACM SIGCOMM*, 2005.

[16] M. Becchi, and S. Cadambi Memory-efficient regular expression search using state merging. *INFOCOM*, 2007.

[17] L. Yuan, C. N. Chuah, and P. Mohapatra. ProgME: towards programmable network measurement. *ACM SIGCOMM*, 2007.

[18] B. Krishnamurthy, H. V. Madhyastha, O. Spatscheck. ATMEN: a triggered network measurement infrastructure. *ACM WWW*, 2005.

[19] H. V. Madhyastha, and B. Krishnamurthy. A generic language for application-specific flow sampling. *ACM CCR*, 2008.

[20] S. Kumar, J. Turner, J. Williams Advanced Algorithms for Fast and Scalable Deep Packet Inspection *ACM ANCS*, 2006.

[21] N. Tuck, T. Sherwood, B. Calder, and G. Varghese. Deterministic memory-efficient string matching algorithms for intrusion detection. *INFOCOM*, 2004.

[22] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An improved construction for counting Bloom filters. *14th Annual European Symposium on Algorithms, LNCS 4168*, 2006.