Fast and Scalable Classification of Structured Data in the Network

Sumantra R. Kundu[1], Sourav Pal[1], Christoph L. Schuba[2], and Sajal K. Das[1]

[1] Dept. of Computer Science and Eng.
University of Texas at Arlington
Arlington, TX 76019 – USA
Email: {kundu, spal, das}@cse.uta.edu
[2] Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054 – USA
Email: Christoph.Schuba@Sun.COM

Abstract. For many network services, such as firewalling, load balancing, or cryptographic acceleration, data packets need to be classified (or filtered) before network appliances can apply any action processing on them. Typical actions are header manipulations, discarding packets, or tagging packets with additional information required for later processing. Structured data, such as XML, is independent from any particular presentation format and is an ideal information exchange format for a variety of heterogeneous sources. In this paper, we propose a new algorithm for fast and efficient classification of structured data in the network. In our approach, packet processing and classification is performed on structured payload data rather than only packet header information. Using a combination of hash functions, Bloom filter, and set intersection theory our algorithm builds a hierarchical and layered data element tree over the input grammar that requires logarithmic time and tractable space complexity.

1 Introduction

Enterprise networks present today usually follow a multi-tiered architecture, wherein at the networking tier, different appliances perform different logical networking services, such as firewalling, load balancing, or cryptographic acceleration on data packets prior to their reaching the first tier application servers. In order for network appliances to decide which actions are to be applied to individual data packets, datagrams need to be classified using a set of predefined rules or service classes. Typical classification actions include header manipulation, routing, filtering, and tagging packets with additional information for subsequent processing. State of the art content-aware switches implementing deep (application layer) packet inspection technology are still limited to examining only the first few hundred bytes of application data; typically attempting to match preconfigured URLs or HTTP cookies. They cannot match generically specified structured data elements beyond these predefined, basic types of HTTP

protocol values. On the other hand, generic packet classification involving structured data represents significant progress beyond traditional and existing packet classification schemes. In this new approach, packets are classified not based on their destination address, but rather on the nature and type of data present in the payload.

Structured data payloads represent strictly ordered sequences of events and can be viewed as a formatted byte strings. Many applications today not only internally organize their data in a structured fashion, but also use similarly structured information for protocol exchanges between them. Such an approach is attractive since it combines data and metadata together and provides a common presentation protocol for a variety of heterogeneous data sources. More and more applications are no longer monolithic, but interact in a distributed fashion that involves extensive peer-to-peer or client-server interaction across the network. Thus, with application chatter occurring in a structured data format, inline network appliances can provide additional benefits if they are able to classify, identify, and operate on traffic flows based on the nature of their payload. Among numerous other possibilities, such content-aware packet classification facilitates:

- **Intelligent Network Partitioning:** where content and structured data--aware network appliances are able to direct traffic streams according to the *characteristics* and *capabilities* of the destination network resources. For example, in an enterprise business environment, all financial transactions involving structured data formats and matching certain well-defined criteria (e.g., credit card transactions exceeding a certain value), could be directed towards a pool of dedicated resources for better response time.
- **Preferential Data Dissemination:** networks deploying such content-aware appliances are able to provide *preferential information dissipation*. Thus, network traffic could be filtered and delivered according to the preferences of the end user, as embodied in publish/subscribe systems [6].
- **Selective Content Encryption:** instead of encrypting the entire packet payload during internal network transactions (e.g., email exchanges within different departments inside a university network and containing social security numbers of employees) *only sensitive portions* of the data stream (social security numbers, in this case) could be encrypted and decrypted by such content-aware network appliances.

However, such an approach has its own challenges. The first and foremost is the overhead and time involved in classifying such structured streams. In this paper, we address this issue and propose an algorithm for fast and efficient classification of structured data. Our algorithm executes in the data plane of network appliances and offers the ability to filter, redirect, and mark network traffic for controlled and preferential transmission. As a case study, we use the eXtensible Markup Language (XML) [5] as an example of structured data format for information exchange. We selected XML because it is predicted to become a large portion (as much as 35% [16]) of network traffic in the near future. As

shown in Figure 1, we use the hypothetical XML document as a running example throughout our paper.

The remainder of the paper is organized as follows: In Section 2, we provide a brief introduction to structured data classification in the context of XML and introduce the terminology used in our algorithm. Then, in Section 3 we introduce our algorithm and study its various aspects. In Section 4, we evaluate the performance of one aspect of the classification algorithm and highlight the prototype development. Finally, in Section 5, we conclude with remarks and directions for future work.

```
<workorder value="1313",priority="6">
 <customer name="UTA", URL="www.cse.uta.edu">
   <priority value="2">
   <cardnum>123456789087</cardnum>
   <description>
       <shipping carrier="UPS">overnight<shipping>
       <type="storage server"></type>
       <price unit="USD">4567</price>
       <qty>2</qty>
    </description>
  </priority>
 </customer>
</workorder>
```

Fig. 1. Sample XML Document used in our study.

2 Background

Information exchange using XML consists of structured data sets representing strictly ordered sequences of events. For example, in web services, applications use XML messages in the form of Simple Object Access Protocol (SOAP) messages to exchange data objects among themselves. Among others, studies involving the use and application of XML in publish/subscribe peer-to-peer (P2P) networks have been documented in [12][14]. Most of these studies have focused on preferential data dissemination and appropriate message semantics in P2P and ad-hoc networks. In large databases, studies on structured data have focused on database access strategies and query subsystems development [11].

We differ from both the above approaches by looking at structured data classification *from the perspective of a network packet classifier*. The central idea is to design an XML-aware packet classifier which is capable of understanding XML vocabulary independently of any specific XML grammar scheme. In Figure 2 we depict the building blocks of our classifier. It consists of three main functional blocks: a rule parser, an event generator, and an event manager. The function of the event generator is to generate tokens based on a specific defined stream parsing schema. For XML, we use the schema as described in W3C standards

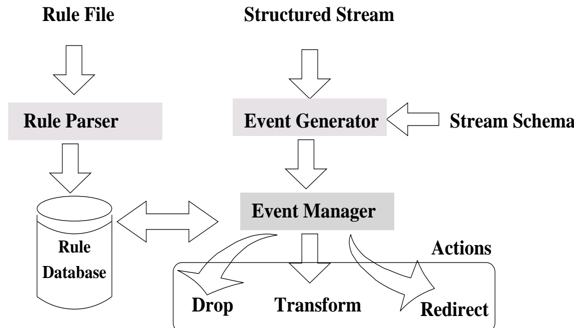

Fig. 2. Architecture of the Network Classifier used for Structured Data Routing.

[13] to generate the tokens. The function of the event manager is to decide upon the most relevant and appropriate action that matches a given rule for the XML stream under consideration. Typical actions include dropping and/or tagging the data stream for further processing, transforming the stream to a different document format (e.g., HTML), or redirecting the stream to a specific destination endpoint (IP address).

The Problem: In the light of the above discussion, we define the classification problem in the following way: *Given a structured data stream where data objects appear in an ordered sequence, how to decide upon an action that best matches the input grammar?*

We observe that in order to decide upon the best action to be initiated on a particular data stream, data sets in the data stream need to be *continuously* and *efficiently* compared against an *in-built structured tree* of data sets built over the input grammar. Furthermore, because *multiple actions* might match a given data stream, the algorithm needs to select the most appropriate action that needs to be applied. We now define some of the terminology used in this paper.

Nodes: Nodes in a structured tree correspond to data defined over the input alphabet. For XML, new nodes are identified by data associated with the opening tag (<). For example, in Figure 1, *customer* is a valid element node in the structured tree.

Rules and Actions: Rules and actions together define the structure of the input rule file that is fed to the rule parser (see Figure 2). Each rule is represented as a tuple; a grammar and an associated action, where the grammar is defined over the domain of an application, conforming to an established schema. Thus,

$$\text{Rule} :=<\text{Grammar, Action}>$$

For XML traffic, we have defined the grammar of the input rule file similar to XPath Expressions (XPE) [13] with relative path names, followed by an action.

A sample input grammar (defined in the input rule file) for our XML document of Figure 1 might have semantics as shown in Table 1. In the example, *priority* is

Rule1	/priority(value== 1);IP:10.11.12.13 : 8234
Rule2	/price> 20000;IP:10.11.12.14 : 8080
Rule3	action:transform:HTML;IP:10.11.12.15 : 80

Table 1. Example of grammar executed on XML stream.

an element *node* and (*value* == 1) is its *attribute*. We say an event has occurred when we encounter the data element *priority* that has the attribute *value* equal to 1 in the XML data stream. Rule 1 directs the XML stream with the element node, *priority*, having an attribute (*value* == 1) to the tuple <IPaddress:port> 10.11.12.14:8234. In Rule 2, we are only interested in events for which the element node *price* has values greater than 20000. All attributes of *price* if present in the data stream constitute *don't care* conditions. In Rule 3, the XML stream is converted to an HTML document format and sent to the IP address tuple 10.11.12.18 : 80. Due to the fact that structured data is a formatted byte string where the element nodes are arranged in a hierarchical fashion, an input rule file defines the element nodes and the associated events the user is interested in. Thus, *events occur only if the element nodes defined in the rule file are also present in the data stream.* The taxonomy of the element nodes in the rule file determine the event list with attributes determining the transition function between the events.

3 Classification Algorithm

Since we only need to find a *subset of events* from the structured data stream, we model the element nodes and the relationship between them as a directed acyclic graph. This graph is referred to as the *structured data tree* (SDT) in our paper. Events occur and traverse along the edges of an SDT. The rule database contains the SDT defined by the input rule file. This graph is created at system startup or whenever the input rule file is updated by the input rule parser. It is important to note that normally we do not attempt to construct the whole XML data stream in memory (except for, e.g., document transformations).

In order to determine if an element node exists in the rule database, the algorithm maintains a simplified bloom filter matrix with controllable degree of false positives. The event generator drives the finite state machine (FSM) until it reaches a node for which the action is defined. Such nodes are referred to as *reachable states.* However, multiple actions might match at each reachable state. A *Jaccard coefficient vector* is calculated to select the most relevant action based on the current event state. This approach is equivalent to a depth-first search of the structured data without the overhead of building the entire streaming data

tree in computer memory. In the next section, we elaborate how we use these concepts in our classification algorithm.

3.1 Bloom Filters

Bloom filters [3] are commonly used for probabilistic membership query tests on a set of $S = \{s_1, s_2, \dots s_n\}$ elements. Each element of the set S, is mapped to a constant space representation spanning k hash functions, $[h_1, h_2, \dots h_k]$ to create a *Bloom Filter Matrix* (BFM). The individual hash functions are bit vectors of length v bits and together define the Bloom filter space. Initially all the bits of the hash function i, h_i, are set to 0. Insertion of a new element, s, into the bloom filter space is accomplished by setting all the corresponding bits of each of the hash vectors, h_i, equal to 1. False positives occurs when a different element, s', is incorrectly predicted to be present in the filter space because all its h_i are found equal to one. This happens when both elements s and s' suffer collisions in their hash values.

In our paper, we use the BFM for identifying and filtering relevant element nodes from the event generator. Such an approach greatly reduces the amount of valid FSM state search required during state transitioning. Further, instead of using k *different* hash functions, we break up the SHA1 hash function values into k different partitions; thus, mimicking k different hash functions. This partitioning is utilized for creating the BFM and for insertions and search operations of element nodes in the rule database. In this study, each partition is referred to as a *block* and we assume all the blocks to be of equal length.

3.2 BFM: Bloom Filter Matrix

In our algorithm, the BFM is created by defining blocks of length n from the 160 bits SHA1 message digest. This division corresponds to $\lceil (160/n) \rceil$ number of hash functions (or blocks). For each of the hash functions, the integer value corresponding to the n bits, is set to 1. Thus, for blocks of n bits, the BFM would require system memory equal to $2^n \times \lceil (160/n) \rceil \times sizeof(int)$ bits. This approach is illustrated in Figure 3 for an arbitrary block size of n bits.

The percentage of false positives depends on the number of hash functions (k) and on the ratio of the size of the filter (f) to the size of the data set (n). For $h_i = (0 \dots 2^v - 1)$, to a good approximation, the false positive rate (P_f) is approximately given by [3] [9]:

$$P_f = (1 - p)^k \qquad \text{where} \qquad p = e^{-kn/(2^v - 1)} \tag{1}$$

For memory constrained systems, there are two available options for limiting memory usage:

Decreasing the size of individual blocks. For example, a BFM with blocks of size 8 bits will require approximately 68% less memory than an equivalent BFM with blocks of size 10 bits.

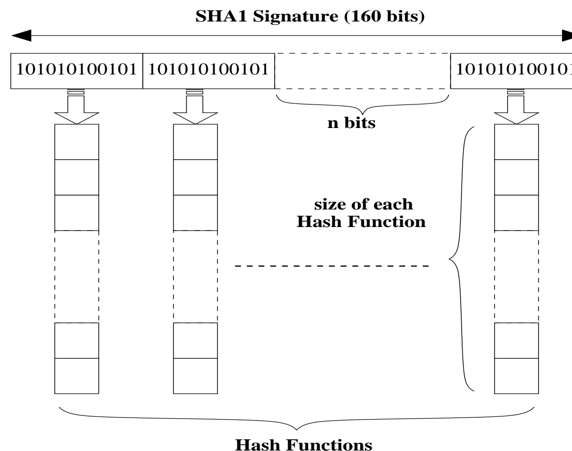

Fig. 3. Bloom Filter Matrix (BFM) created out of a single SHA1 hash function.

Using a truncated integer representation for each block. For example, a block length of 8 bits can represent 256 values. However, to conserve memory, the system might want to represent it with only 5 bits, allowing 32 unique values. Decreasing the block length size to 5 bits, will decrease the BFM memory requirement by 87% (as compared to a BFM with blocks of size 8 bits) at the cost of increased false positive rate. However, we amortize this cost by organizing the SDT in system memory so as to minimizing the overhead of node search and retrieval operations.

3.3 Storage of the Element Nodes

In order to eliminate false positives during element node membership queries, we have to make sure that the node exists in the rule database. Blocks of length n bits defined over the SHA1 hash function not only identify the hash vectors h_i of the BFM, but also define levels in the rule database memory. In such a case, it is not difficult to see that the maximum depth of the SDT along any search path is $\lceil(160/n)\rceil$. In the best case when the signatures of all element nodes have at least one bit different among their first n bits (i.e., in the first block), there are no collision, and all elements fit in the first level of the SDT.

Associated with each level of the SDT is a Signature Map Table (SMT). It contains entries which are individual element node signature digests corresponding to level i and are mapped to the associated address in system memory where nodes reside. The size of the SMT is equal to the number of entries times the space required to store the element signatures. Collisions occur when two elements have identical signature bit patterns of length n at level i. In this case, the Signature Map Table (SMT) contains a reference to the next level of the tree.

The depth of the SDT traversal is directly related to the probability of a collision occurring between two nodes with a signature digest of length n. The

probability of two element nodes having identical bit patterns for a message digest of length n bits is $2/2^n$. This value is equal to approximately $1.35e-48$ for $n = 160$ (total length of the message digest of SHA1). The probability of two element nodes having at least one bit *different* is equal to $(1 - \frac{n}{^{2^n}C_2})$, for a message digest of length n bits.

We choose SHA1 for our algorithm because of its low collision probability and its collision resistance. The latter property implies that the probability of node collision decreases rapidly as our algorithm traverse the SDT along any search path. Figure 4 shows the organization of the system memory for any level i.

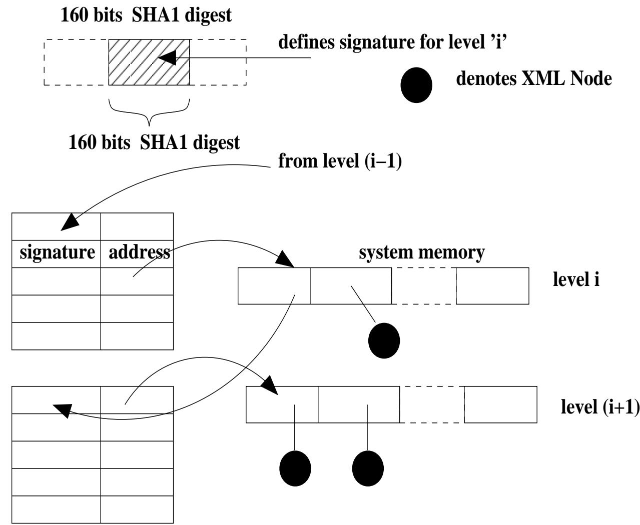

Fig. 4. Organization of Element Nodes in System Memory (RAM).

3.4 Checking the Existence of an Element Node

Determining the existence of the element node in the rule database is a two step process. In the first step, the SHA1 message digest is calculated and is partitioned into blocks of n bits. Then, the corresponding entries of the BFM are checked in parallel. If any one of the entries corresponding to the integer representation of the block of n bits among the k, $(k = 1, 2 \ldots \lceil 160/n \rceil)$ hash functions, is equal to 0, then the element node is not present in the rule database and the algorithm terminates. Else, the node exists with a certain probability of false positive.

To treat false positives, in the *second step* of the search operation, the algorithm searches through the SDT until an entry in the SMT is found that contains the element node or until a path of length $\lceil 160/n \rceil$ in the SDT has been traversed.

Building the Structured Data Tree (SDT): After all the element nodes have been inserted into the rule database, individual nodes need to be connected to form the SDT. This step is performed based on the relationship enforced between the element nodes in the input rule file. The SDT, thus, defines the event list and the flow of events in our classifier architecture.

3.5 Selecting the most appropriate action in the context of an element node

In structured data, individual element nodes have predicates which are valid within the name space of the node only. For example, the element node *priority* has the predicate ($value = 2$) in our example of Figure 1. The predicates define the events in the context of the element node and are stored as members of a set. There might be boolean relationship between the members themselves.

Suppose we have the following grammar present in the input rule file:

```
/workorder(value>''1000'') OR (priority>''2'') OR (currency=''USD'');
/workorder(value<=''2765'') OR(customer=''sun.com'');
/workorder(department!=''sales'') OR(value>''345'' );
```

Now, in the context of XML element *workorder* of Figure 1, we observe that all the above rules have one or more members which match this data stream. Our aim is to find the action which best fits the given data stream.

The *Jaccard coefficient* between two sets A and B is defined as the quotient of the cardinality of set $A \cap B$ and set $A \cup B$ and is used to find the degree of similarity between the two sets A and B:

$$J = |A \cap B|/|A \cup B|$$

Thus, given sets A and B, we can calculate the degree of resemblance between the two. The rule corresponding to the set with the highest value of J is chosen for the given structured data stream. In case of identical values of J one of the possible actions is chosen at random.

4 Experimental Evaluation

We have implemented our classification algorithm in C++ and conducted experiments on a Pentium IV 2.4 GHz Intel CPU with 1 GB main memory running SuSe 9.2 Professional edition on Linux kernel 2.6.8. We used the Xerces C++ XML parser [15] in our preliminary evaluation phase.

The following events are generated by the XML parser: `startStream()`, `startElementNode()`, `elementConstraints()`, `endElementNode()`, `endStream()`. Events `startStream()` and `endStream()` denote the start and end of the structured data stream. Event `startElementNode()` triggers the context of the element node while event `endElementNode()` closes the context. Event `elementConstraints()` defines all the events and transition functions within the name space of the element node.

We have evaluated the redirection feature of the classification algorithm for

XML documents with an input rule file containing 2500 rules as defined in Section 2. From Figure 5, we observe that the overhead of redirection increases with the increase in the number of rules, but the increase in overhead is still manageable.

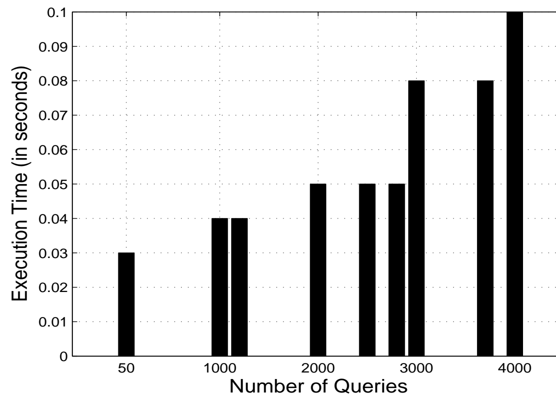

Fig. 5. Performance of the algorithm with number of queries in the rule file

A prototype for Content Based Routing (CBR) is currently being developed using the the Click software router [7]. Such a router will have the capability to perform inline network filtering and forwarding of structured data and will directly interface with the network device driver as a loadable kernel module. The classification algorithm presented in this paper is currently being integrated inside our CBR testbed for structured data routing.

5 Conclusions and Future Work

In this paper, we have introduced a novel algorithm for classification of structured data in the network. In the classifier architecture, a combination of hash functions, a Bloom filter, and a Jaccard coefficient vector has been used to create a structured data tree and to select the most appropriate action corresponding to an input structured data stream. We have implemented the classification algorithm and evaluated the redirection feature of the algorithm. It is observed that while the execution time increases with respect to the number of queries over a set of 2500 tags, it is well within tractable limits. While the results are encouraging, we intend to carry out further investigations with real time network traffic. There exists several opportunities to extend the results of our work. Using *Merkle trees* [8] it is possible to improve the process of instantiation and searching the SDTs. Such an approach would guarantee both logarithmic time and space complexity for all combinations of SDT operations. We are also looking

upon improving the action matching algorithm and defining an efficient structured data parser that can extract tokens when the structured data is found to cross network packet boundaries.

Acknowledgment: The material of this work is supported by NSF ITR grant IIS-0326505. We would also like to thank the anonymous reviewers for their helpful comments and suggestions which improved the quality of this study.

References

1. S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo, "Dynamic XML Documents with Distribution and Replication", *ACM SIGMOD*, pp. 527-538, June 2003.
2. M.R. Anderberg, "Cluster Analysis for Applications", *New York Academic Press*, 1973.
3. B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors", *Communications of the ACM*, pp. 422-426, 1970.
4. J. W. Byers, J. Considine, M. Mitzenmacher, and S. Rost, "Informed Content Delivery Across Adaptive Overlay Networks", *IEEE/ACM Transactions on Networking (TON)*, vol.12, pp 767-780, 2004. 2002.
5. eXtensible Markup Language (XML), Information available online at: http://www.w3.org/XML/
6. P. W. Foltz and S. T. Dumais, "Personalized information delivery: an analysis of information filtering methods", *Communications of the ACM*, vol. 35, pp 51-60, 1992.
7. R. Morris, E. Kohler, J. Jannotti, and M. Frans Kaashoek, "The Click Modular Router", *ACM Symposium on Operating Systems Principle (SOSP)*, pp. 217 -233, 1999.
8. R. Merkle, "Protocols for public key cryptography", *IEEE Symposium on Security and Privacy*, pp. 122-134, 1980.
9. M. Mitzenmacher, "Compressed bloom filters", *ACM Symposium on Principles of Distributed Computing*, pp. 144-150, 2001.
10. National Institute of Standards and Technology (NIST), "Announcing the secure hash standards", *Federal Information Proceeding Standards Publication*, August 2002.
11. D. Olteanu, T. Furche, F. Bry, "An Efficient Single-Pass Query Evaluator for XML Data Streams", *ACM Symposium on Applied Computing*, pp. 627 - 631, 2004.
12. S. Ratnasamy, M. Handley, R. M. Karp, and S. Shenker, "Application-Level Multicast using Content-Addressable Networks", *Third International COST264 Workshop on Networked Group Communication*, pp. 14-29, 2001.
13. W3C XML Schema, Information available online at: http://www.w3.org/XML/Schema
14. A. Snoeren, K. Conley, and D. K. Gifford, "Mesh-based Content Routing using XML", *ACM Symposium on Operating Systems (SOSP)*, pp. 160-173, 2001.
15. Xerces C++ Parser, Information available online at: http://xml.apache.org/xerces-c/
16. ZapThink, Information available online at: http://www.zapthink.com/
17. N. Duffield, C. Lund, and M. Thorup, "Estimating Flow Distributions from Sampled Flow Statistics", *IEEE/ACM Transactions on Networking (TON)*, vol. 13, pp. 933-946, October 2005.
18. A. Kumar, J. Xu, O. Spatschek, and L. Li, "Space-code Bloom Filter for Efficient Per-Flow Traffic Measurement", *IEEE INFOCOM*, vol. 3, pp. 1762-1773, August 2004.
19. D. Shah, S. Iyer, B. Prabhakar and N. McKeown, "Maintaining Statistics Counters in Router Line Cards", *IEEE Micro*, vol. 22, pp. 76-81, 2002.