ISSN 2229-5518

Secured Data Transmission Using Snort Rules and Mining Technique

[1]Mr.R.venkatramana ,[2]Mrs.M.Sreedevi

Abstract— Network traffic analysis becomes more and more crucial in the IP network infrastructure as the amount of IP packets transmitted on the Internet at any given moment of time increases enormously. A thorough understanding of the IP traffic will help us better design our network topology and utilize bandwidth more effectively. From the perspective of security, it can also protect our system from attacks, such as intrusions, our model employs feature selection so that the binary classifier for each type of attack can be more accurate, which improves the detection of attacks that occur less frequently in the training data. Based on the accurate binary classifiers, our model applies a new ensemble approach which aggregates each binary classifier's decisions for the same input and decides which class is most suitable for a given input. During this process, the potential bias of certain binary classifier could be alleviated by other binary classifiers' decision. Our model also makes use of multi boosting for reducing both variance and bias. The clients have some rules to communicate between them using snort rules. Any Communications (such as FTP, SMTP, etc) between the clients are monitored by the snort. If it continues again, then that particular client will be disconnected from this network (means cannot be able to communicate with other clients in that network.) But, that client will be physically connected with the network. The proposed work describes a network traffic analysis software tool, which provides searching, visualization, and preprocessing functions with a user-friendly GUI implemented in Java language. Within the huge network traffic data collected, a user can identify any particular packets using various searching functions provided. Visualization presents the analyzed result in a different setting to further enhance the analysis. The GUI in Java allows the tool to be used in different platforms. This tool is tested and demonstrated through several real network datasets.

Index Terms—Algorithms, filtering algorithms, finite-state automata (FSA), mathematics, packet filters, packet processing, predicate optimization, protocol description languages (PDLs), run-time safety, snort rules and mining techniques.

I. Introduction

PACKET filters are a class of packet manipulation programs used to classify network traffic in accordance to a set of user-provided rules; they are a basic component of many networking applications such as shapers, sniffers, demultiplexers, firewalls, and more. The modern networking scenario imposes many requirements on packet filters, mainly in terms of processing speed (to keep up with network line rates) and resource consumption (to run in constrained environments). Filtering techniques should also support modern protocol formats that often include cyclic or repeated structures (e.g., MPLS label stacks, IPv6 extension headers).

Finally, it is also crucial that filters preserve the integrity of their execution environment, both in terms of memory access safety and termination enforcement, especially when running as an operating system module or on the bare hardware. Although at first sight this aspect might not seem crucial, it is a fact that many of the limitations built into existing packet filters derive directly from safety issues. As an example, the impossibility of automatically proving termination for a generic computer program led the BPF [1] designers to generate acyclic filters only, thus preventing the parsing of packets with multiple levels of encapsulation or repeated field sequences.

Existing packet filters focus invariably on subsets of these issues but, to the best of our knowledge, do not solve all of them at the same time. As an example, two widely known generators, BPF [2] and PathFinder [3], do not support recursive encapsulation; NetVM-based filters [4], on the other hand, have no provision for enforcing termination, either in filtering code or in the underlying virtual machine. This paper presents Stateless Packet Filter (SPAF), a finite-state automata (FSA)-based technique to generate fast and safe packet filters that are also flexible enough to fully support most layer-2 to layer-4 protocols, including optional and variable headers and recursive encapsulation. The proposed technique specifically targets the lower layers of the protocol stack and does not directly apply for deep packet inspection nor for stateful filtering in general. Moreover, for the purpose of this paper, we consider only static situations where on-the-fly rule set updates are not required. While these limitations exclude some interesting use cases, SPAF filters are nevertheless

[1]R.Venkatramana,Research Scholar(M.Tech) Department of CSE Madanapalli Institute of Technology and Science, Madanapalli, Andhra Pradesh, India. Mail: rvramana.r@gmai.com

[2]Mrs.M.Sreedevi, Associate Professor Department of CSE Madanapalli Institute of Technology and Science, Madanapalli, Andhra Pradesh, India. Mail: srikundu@yahoo.co.in

ISSN 2229-5518

useful for a large class of applications, such as monitoring and traffic trace filtering, and can serve as the initial stage for more complex tools such as intrusion detection systems and firewalls.

A stateless packet filter can be expressed as a set of predicates on packet fields, joined by boolean operators; often these predicates are not completely independent from one another, and the evaluation of the whole set can be short-circuited. One of the most important questions in designing generators for high-performance filters is therefore how to efficiently organize the predicate set to reduce the amount of processing required to come to a match/mismatch decision. By considering packet filtering as a regular language recognition problem and exploiting the related mathematical framework to express and organize predicates as finite-state automata, SPAF achieves by construction a reduction of the amount of redundancy along any execution path in the resulting program: Any packet field is examined at most once. This property emerges from the model, and it always holds even in cases that are hard to treat with conventional techniques, such as large-scale boolean composition. Moreover, thanks to their simple and regular structure, finite automata also double as an internal representation directly translatable into an optimized executable form without requiring a full-blown compiler. Finally, safety (both in terms of termination and memory access integrity) can be enforced with very low run-time overhead.

The rest of this paper is structured as follows. Section II presents an overview of the main related filtering approaches developed to this date. Section III provides a brief introduction to the FSAs used for filter representation and describes the filter construction procedure. Section IV focuses on executable code generation and on enforcing the formal properties of interest, Finally, Section V reports conclusions and also highlights possible future developments.

II. Related Work

Given their wide adoption and relatively long history, there is a large corpus of literature on packet filters. A first class of filters is based on the CFG paradigm; the best-known and most widely employed one is probably BPF [1], the Berkeley Packet Filter. BPF filters are created from protocol escriptions hardcoded in the generator and are translated into a bytecode listing for a simple, *ad hoc* virtual machine. The bytecode was originally interpreted, leading to a considerable run-time overhead impact that can be reduced by employing JIT techniques [5]. BPF disallows backward jumps in filters in order to ensure termination, thus forgoing support for, e.g., IPv6 extension headers; memory protection is enforced by checking each access at run-time. Multiple filter statements can be composed together by boolean operators, but in the original BPF implementation, only a small number of optimizations are performed over predicates, leading to run-time inefficiencies when dependent or repeated predicates are evaluated. Two relevant BPF extensions are BPF and xPF. BPF [2] adds local and global data-flow optimization algorithms that try to remove redundant operations by altering the CFG structure. xPF [6] relaxes control flow restrictions by allowing backward jumps in the filter CFG; termination is enforced by limiting the maximum number of executed instructions through a run-time watchdog built into the interpreter, but its overhead was not measured, and extending this approach to just-in-time code emission has not been proposed and might prove difficult.

Afurther CFG-based approach, unrelated to BPF, is described in [4]. Its main contribution is decoupling the protocol database from the filter generator by employing an XML-based protocol description language, NetPDL [7]. Filtering code is executed on the NetVM [8], a special-purpose virtual machine targeting network applications that also provides an optimizing JIT compiler that works both on filter structure and low-level code. The introduction of a high-level description language reportedly does not cause any performance penalties; this approach, however, delegates all safety considerations to the VM and does not provide an effective way to compose multiple filters. In general, CFG-based generators benefit from their flexible structure that does not impose any significant restriction on predicate evaluation order; for the same reason, however, they are prone to the introduction of hard-to-detect redundancies, leading to multiple unnecessary evaluations if no further precautions are taken. Even when optimizers are employed and are experimentally shown to be useful, they work on an opportunistic basis and seldom provide any hard guarantees on the resulting code.

A second group of filter generators chooses tree-like structures to organize predicates. PathFinder [3] transforms predicates into template masks (atoms), ordered into decision trees. Atoms are then matched through a linear packet scan until a result is reached. Decision trees enable an optimization based on merging prefixes that are shared across multiple filters. PathFinder is shown to work well both in software and hardware implementations, but it does not take protocol database decoupling into consideration, and no solution to memory safety issues is proposed for the software implementation. FSA-based filters share a degree of similarity with PathFinder as packets are also scanned linearly from the beginning to the end, but predicate organization, filter composition, and safety considerations are handled differently. DPF [9] improves over PathFinder by generating machine code just-in-time and adding low-level optimizations such as a flexible switch emission strategy. Moreover, DPF is capable

IJSER © 2012

of aggregating bounds checks at the atom level by checking the availability of the highest memory offset to be read instead of considering each memory access in isolation; our technique, described in Section IV-E, acts similarly but considers the filter as a whole, thus further reducing run-time overhead. While organizing predicates into regular structures makes it easier to spot redundancies and other sources of overhead, it also introduces different limitations. As an example, generators restricted to the aforementioned acyclic structures do not fully support tunneling or repeated protocol portions. Moreover, it has been noted that performing prefix coalescing is not sufficient to catch certain common patterns, resulting in redundant predicate evaluation [2].

A third approach is to consider packet filtering as a language recognition problem. Jayaram *et al.* [10] use a pushdown automaton to perform packet demultiplexing; filters are expressed as LALR(1) grammars and can therefore be effectively composed using the appropriate rules. This solution improves filter scalability, but there are downsides related to the push-down automaton: A number of specific optimizations are required to achieve good performance. It is also quite unwieldy to express protocols and filter rules as formal grammars that must be kept strictly unambiguous: The authors marginally note that the simpler FSA model would be sufficient for the same task.

Apart from the specialized solutions for fast packet filtering mentioned, one of the most widely used packet filtering programs is the NetFilter framework.1 NetFilter is a component of the Linux kernel that performs packet filtering, firewalling, mangling operations (e.g., network address translation), and more, acting through a set of hooks and callbacks that intercept packets as they traverse the networking stack. In contrast with all the aforementioned approaches, NetFilter uses the relatively simple method of applying all the specified rules in sequence when performing packet filtering, leading to poor performance and scalability; moreover, it appears not possible to specify an arbitrary predicate, filters being limited to preset protocols and statements that are specialized by specifying actual network addresses and ports.

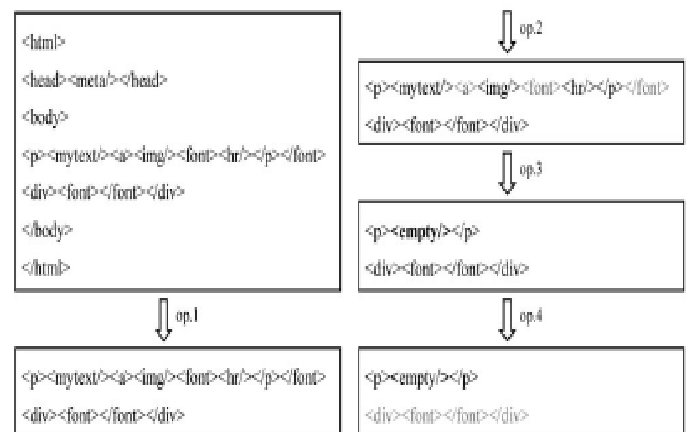
Besides the generation technique, there have also been improvements along other dimensions such as architectural considerations, as demonstrated by xPF, FFPF [19], and nCap [20], or dynamic rule sets support, as shown by the SWIFT tool [21]. We consider these aspects out of scope for the purpose of this paper, being either orthogonal to the technique we present or the object of future works.

Definition of Near-Duplicate

The central idea of near-duplicate spam detection is to exploit reported known spams to block subsequent ones which have similar content. For different forms of e-mail representation, the definitions of similarity between two e-mails are diverse. Unlike most prior works representing e-mails based mainly on content text, we investigate representing each e-mail using an HTML tag sequence, which depicts the layout structure of e-mail, and look forward to more effectively capturing the near-duplicate phenomenon of spams.

Let I ¼ ft1; t2; . . . ; ti; . . . ; tn; <mytext=>; <anchor>g be the set of all valid HTML tags with two types of newly created tags, <mytext=> and <anchor>, included. An e-mail abstraction derived from procedure SAG is denoted as <e1; e2; . . . ; ei; . . . ; em>, which is an ordered list of tags, where ei 2 I. The definition of near duplicate is: "Two e-mail abstractions _ ¼ <a1; a2; . . . ; ai; . . . ; an> and _ ¼ <b1; b2; . . . ; bi; . . . ; bm> are viewed as near-duplicate if 8ai ¼ bi and n ¼ m."

The tag length of an e-mail abstraction is defined as the number of tags in an e-mail abstraction.

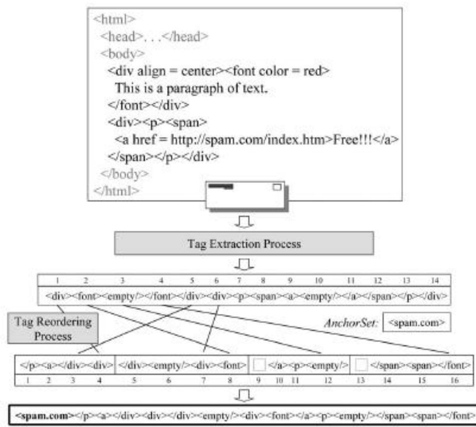


The following sequence of operations is performed in the preprocessing step.

1. Front and rear tags are excluded.
2. Nonempty tags2 that have no corresponding start tags or end tags are deleted. Besides, mismatched nonempty tags are also deleted.
3. All empty tags2 are regarded as the same and are replaced by the newly created <empty=> tag.
Moreover, successive <empty=> tags are pruned and only one <empty=> tag is retained.
4. The pairs of nonempty tags enclosing nothing are removed.

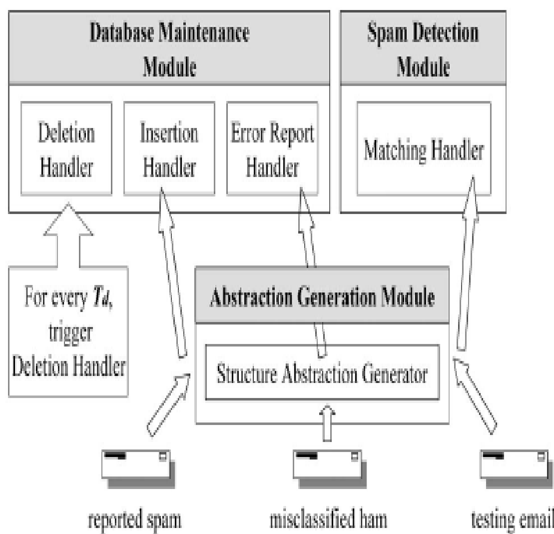
Example for Mail

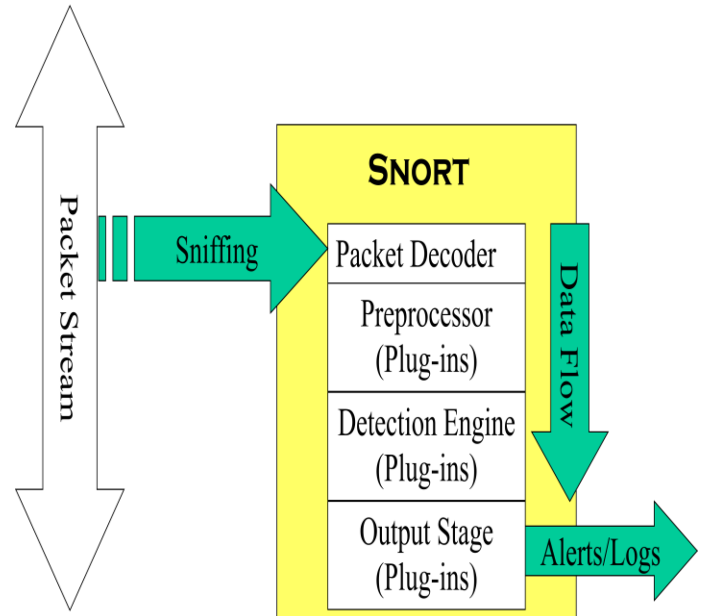
ISSN 2229-5518



SYSTEM ARCHITECTURE

In the packet filtering process with the increasing popularity of electronic mail (or e-mail), several people and companies found it an easy way to distribute a massive amount of unsolicited messages to a tremendous number of users at a very low cost. These unwanted bulk messages or junk emails are called spam messages. The majority of spam messages that has been reported recently are unsolicited commercials promoting services and products including sexual enhancers, cheap drugs and herbal supplements, health insurance, travel tickets, hotel reservations, and software products. They can also include offensive content such pornographic images and can be used as well for spreading rumors and other fraudulent advertisements such as make money fast.

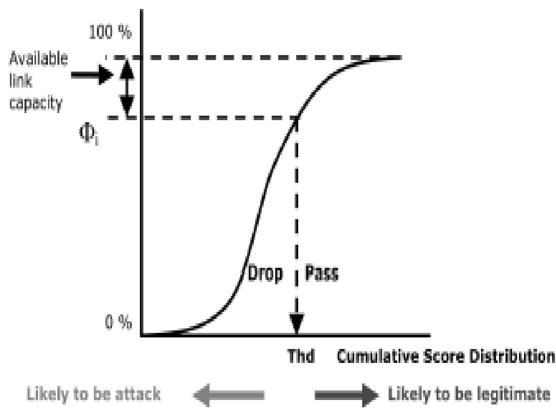




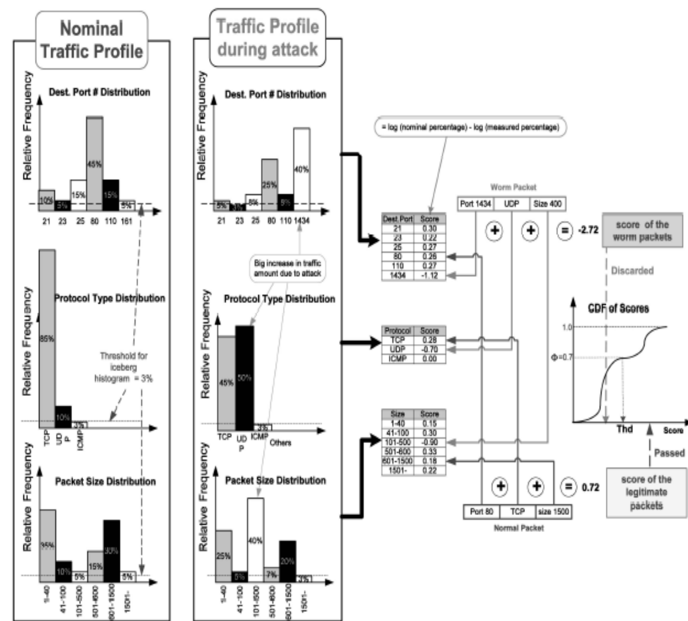
As a result, spam has become an area of growing concern attracting the attention of many security researchers and practitioners. In addition to regulations and legislations, various anti-spam technical solutions have been proposed and deployed to combat this problem. Front-end filtering was the most common and easier way to reject or quarantine spam messages as early as possible at the receiving server. However most of the early anti-spam tools were static; for example using a blacklist of known spammers, a white list of good sources, or a fixed set of keywords to identify spam messages. Although these list-based methods can substantially reduce the risk provided that lists are updated periodically, they fail to scale and to adapt to spammers' tactics.

Selective Packet Discarding

Once the score is computed for a packet, selective packet discarding, and overload control can be performed using the score as the differentiating metric. Since an exact prioritization would require offline, multiple-pass operations, e.g., sorting and packet buffering, the following alternative approach is taken into account. First, the cumulative distribution function (CDF) of the scores of all incoming packets in time period (Ti) is maintained. Second, the cut-off threshold score is calculated. Third, the arriving packets in time period T (i+1) if its score value is below the cut-off threshold are discarded. At the same time, the packets arriving at T (i+1) create a new CDF.



Selective packet discarding



Discarding SQL Slammer Worm attack packets

III. Filter Generation Technique

In this section we describe about filtering techniques using snort rules and mining techniques.

One of the key concepts in PacketScore is the notion of "Conditional Legitimate Probability" (CLP) based on Bayesian theorem. CLP indicates the likelihood of a packet being legitimate by comparing its attribute values with the values in the baseline profile. Packets are selectively discarded by comparing the CLP of each packet with a dynamic threshold. The concept of using a baseline profile with Bayesian theorem has been used previously in anomaly-based IDS (Intrusion Detection System) applications, where the goals are generally attack detection rather than real-time packet filtering.

In this research, the basic concept to a practical real-time packet filtering scheme using elaborate processes is extended. In this method, the PacketScore operations for single-point protection is described, but the fundamental concept can be extended to a distributed implementation for core-routers.

To make it more suitable for real-time processing, conversion of floating-point division/multiplication operations into subtraction/addition operations is made. Scoring a packet is equivalent to looking up the scorebooks, e.g., the TTL scorebook, the packet size scorebook, the protocol type scorebook, etc. After looking up the multiple scorebooks, the matching CLP entries in a log-version scorebook are added. This is generally faster than multiplying the matching entries in a regular scorebook. The small speed improvement from converting a multiplication operation into an addition operation is particularly useful because every single packet must be scored in real-time. This speed improvement becomes more beneficial as the number of scorebooks increases. On the other hand, generating a log-version scorebook may take longer than a regular scorebook generation. However, the scorebook is generated only once at the end of each period and it is not necessary to observe every packet for scorebook generation; thus, some processing delay can be allowed. Furthermore, scorebook generation can be easily parallelized using two processing lines, which allows complete sampling without missing a packet.

The purpose of a stateless packet filter generator is to create a program that, given a finite-length byte sequence (a packet) as its input, returns a binary match/mismatch decision. The input of the generator itself consists of a set of filter rules provided by the user that specify the desired properties of matching packets; each rule, in turn, consists of multiple predicates expressed in a simple high-level language (where header fields and protocols appear symbolically), combined together with boolean operators. In older generators, the set of supported protocols was fixed; in modern ones protocol header formats are kept into an external database that can be updated without modifying the generator.

In order to develop a successful FSA-based filtering technique, it is first of all necessary to show that any filter of interest can be expressed as a finite automaton, then

provide a method to transform a high-level filter statement and a protocol database into FSA form. Finally, the resulting automaton must be translated into an efficiently executable form.

A. Protocol Database Compilation

The first phase in the SPAF generation process consists of parsing the protocol database and building template automata that recognize all the correctly formatted headers for a given protocol. These automata will be reused and specialized in later phases to create the final filter.

In order to decouple filter generation from the protocol database, we have employed an XML-based protocol description language (NetPDL [7]) designed to describe the on-the-wire structures of network protocols and their encapsulation relationships. NetPDL descriptions are stored in external files that can be freely edited without modifying the generator itself.

A precise description of NetPDL is beyond the scope of this paper. Nevertheless, we shall provide a quick overview of the features supported by the FSA generator. The language provides a large number of primitives that enable the description of header formats of layer-2–7 protocols, but for the scope of this work we have restricted our support to those designed for layer-2–4 decoding. The basic building block of a protocol format is the header field, a sequence of bytes or bits that can be either fixed or variable in size. Adjacent fields are by default laid out in sequence, but more complex structures such as optional or repeated sections can be created using conditional choices and loops; these statements are controlled by expressions that can contain references to the values of previously encountered fields.
A second NetPDL portion contains a sequence of control flow operations (if, switch) that predicate encapsulation relationships. In general, the control flow is followed until a nextproto tag is encountered, specifying which is the next protocol to be found in the packet. A NetPDL database thus

```
<protocol  name =”ipv6”>
  <format>
    <field>
<field type=”bit” name = ”ver” mask=”0xF0000000” size =”4”/>
<field type=”bit” name = ”tos” mask=”0x0F000000”size =”4”/>
<field type=”bit” name = ”flabel” mask=”0x00FFFFFF”size =”4”/>
<field type=”fixed” name=”plex”  size =”2”/>
<field type=”nexthdr” name=”plex”  size =”1”/>
<field type=”hop” name=”plex”  size =”16”/>
<field type=”src” name=”plex”  size =”16”/>
<field type=”dst” name=”plex”  size =”16”/>
<loop type =”while” expr=”1”>
<switch expr=”nexthdr”>
<case value=”0”><includeblk name=”HBH”/></case>
<case value=”0”><includeblk name=”AH”/></case>
<default>
<loopctrl type=”break”/>
</default>
  </switch>
    </loop>
      </fields>
        </format>
<encapsulation>
<switch expr=”nexthdr”>
<case value=”4”> <nextproto proto=”#ip”/></case>
<case value=”4”> <nextproto proto=”#tcp”/></case>
<case value=”4”> <nextproto proto=”#udp”/></case>
</switch>
  </encapsulation>
    </protocol>
```

IPV6 NetPDL excerpt

Describes an oriented encapsulation graph where the vertices are protocols and the edges are encapsulation relationships. Currently, the graph begins with a single user-specified root that usually represents the link-layer protocol, but an extension to multiple ones would be trivial. Starting from this root, the FSA generator follows the encapsulation graph and builds a FSA for every reachable protocol using the method explained later in this section. As an example, a simplified NetPDL description of the IPv6 header format is presented in Fig. 1. IPv6 starts with a sequence of fixed-size fields; bitfields (such as ver) are specified by the mask attribute. The initial portion is followed by a set of extension headers, each one containing a “next header” information (nexthdr). This sequence is of unspecified (but implicitly finite, as any packet is finite) length, and it is described using a switch nested within a

loop: At each iteration, the newly read nexthdr field is evaluated, and if no more extension headers are present, the loop terminates. Encapsulation relationships are also specified in a similar fashion by jumping to the correct protocol depending on the value of the last nexthdr encountered. SPAF currently supports the full versions of the most common layer-2–4 protocols in use nowadays, such as Ethernet, MPLS, VLAN, PPPoE, ARP, IPv4, IPv6, TCP, UDP, and ICMP; this set can be easily extended as long as no stateful capabilities are required.

An important point regarding FSA creation from NetPDL descriptions is that, as long as it is correctly performed, it is not be a critical task for filter performance: Any resulting automaton ultimately will be determinized and minimized, yielding a canonical representation of the filter that does not depend on the generation procedure. For this reason, and given the complexity involved, the NetPDL-to-FSA conversion procedure is not fully described in this paper, and it can be regarded as an implementation detail. Nevertheless, in order to exemplify how the conversion can be done, we report the key steps for translating the NetPDL snippets of Fig. 2 into the corresponding automata.

The purpose of this initial conversion step is not to generate automata immediately suitable for filtering. On the contrary, the results are templates for the following generation steps, representing the "vanilla" version of protocol headers, with no other conditions imposed, to be specialized according to the filter rules. Since they are strictly related to header format, any inputconsuming transition in these templates can be related to a specific portion of one3 header field; this information must be preserved to accommodate the imposition of filtering rules. For this reason, template automata are augmented by marking all the relevant transitions with the related field's name.4 The simplest example is generating an automaton that parses a fixed-length header field [Fig. 2(a)]: It is sufficient to build a FSA that skips an appropriate amount of bytes, resulting in Fig. 2(b). During the construction process, header fields are given well-defined start and end5 states that are used as stitching points to join with any predecessors or successors by -transitions, as required. A more complex example involving a conditional choice is shown in Fig. 2(c). The generation procedure starts by creating automata representations for all the initial fields in the NetPDL description; upon encountering the switch construct, however, the generator backtracks the transition graph until it encounters the type field. Once found, all the states/transitions that follow type (the block in the figure) are replicated. The original copy is left as is, while in the replica the transitions for type are specialized to recognize the bytes of interest for the switch, so the right path will be taken depending on the actual input values. Finally, the correct trailing block (or) is joined in the right place. The last example [Fig. 2(e) and (f)] shows the automata generated for a header structure similar to the IPv6 extension headers case. In this case, a loop is interlocked with a switch construct, and a greater amount of block replication is required to ensure that independent paths exist into the automaton for every possible combination of the current nexth value (upon which the outcome of the switch depends) and the next nexth value, which might cause the loop to end.

Encapsulation relationships are handled in a similar fashion by spawning new paths in the automaton graph that end with a special state marked with the protocol that should follow. The exact usage of these marked states is explained in Section III-D. The generation procedure acts to counter the absence of explicit storage locations in the FSA model; when it becomes necessary to use the values of previously encountered fields for subsequent computations, the only solution is to spawn a number of parallel branches within the automaton, each one associated with a specific value of the field under consideration.

B. Multicast Packet Delivery

Here we discuss about packet forwarding to the nodes

Packet sending from the source

After the multicast tree is constructed, all the sources of the group could send packets to the tree and the packets will be forwarded along the tree. In most tree-based multicast protocols, a data source needs to send the packets initially to the root of the tree.

The source node want send the data to the members at that time we perform the security action, i.e. whenever the source node want to send the data , the source node can encrypt the data by using AES (Advanced Encryption Standers) the encrypted data can be transferred to the group members , in the transmission of packets the intermediate nodes want to read the data , if suppose the nodes can access the data that time we don't have any problem because the data is in the encryption form i.e. cipher text , due to this text the intermediate nodes can't get the data it can simply transfer the data to the destination, in the destination side the receiver can decrypt the data using AES algorithm.

For providing the security we use the Advanced Encrypted Standards Algorithm

The algorithm described by AES is a symmetric-key algorithm, meaning the same key is used for both encrypting and decrypting the data. The strength of a 128-bit AES key is roughly equivalent to 2600-bits RSA key. AES data encryption is a more mathematically efficient and elegant cryptographic algorithm the time required to crack

an encryption algorithm is directly related to the length of the key used to secure the communication (It takes less time). AES allows you to choose a 128-bit, 192-bit or 256-bit key, making it exponentially stronger than the 56-bit key of DES (RSA). The algorithm was required to be royalty-free for use worldwide .AES has defined three versions, with 10, 12, and 14 rounds. Each version uses a different cipher key size (128, 192, or 256), but the round keys are always 128 bits.

IV. CONCLUSION

We have designed, prototyped, and evaluated SPAF, a packet filter generator based on the creation of finite-state automata from a high-level protocol format database and filter redicates. SPAF aims at emitting fast and efficient filters while preserving all the relevant safety properties, both in terms of memory access correctness and termination. The PacketScore scheme is used to defend against DDoS attacks. The key concept in PacketScore is the Conditional Legitimate Probability (CLP) produced by comparison of legitimate traffic and attack traffic characteristics, which indicates the likelihood of legitimacy of a packet. As a result, packets following a legitimate traffic profile have higher scores, while attack packets have lower scores. This scheme can tackle never-before-seen DDoS attack types by providing a statistics-based adaptive differentiation between attack and legitimate packets to drive selective packet discarding and overload control at high-speed.

Thus, PacketScore is capable of blocking all kinds of attacks as long as the attackers do not precisely mimic the sites' traffic characteristics. The performance and design tradeoffs of the proposed packet scoring scheme in the context of a stand-alone implementation is studied. By exploiting the measurement/scorebook generation process, an attacker may try to mislead PacketScore by changing the attack types and/or intensities. We can easily overcome such an attempt by using a smaller measurement period to track the attack traffic pattern more closely. We are currently investigating the generalized implementation of PacketScore for core networks.

In order to prove this technique on the field, we have developed a filter generator that creates filters from an external protocol database and user-specified rules. Filter DFAs can be used as they are by existing hardware or software engines or translated into C code by the back end.We also developed an ad hoc DFA execution engine that adapts its operations to the word size of the underlying machine instead of processing a byte at a time and enforces memory safety and termination through run-time fully aggregated bound checks. The run-time performance and memory occupation of SPAF filters have been evaluated both in synthetic and real-world benchmarks. Test results show that FSA-based filters perform on a similar or improved level as other modern approaches such as BPF+, both on simple and complex filters; SPAF filters are also shown to scale better with increasing numbers of filtering rules. The measured overhead of run-time safety checks is small and does not cause any significant penalties both in times of run-times (few checks are executed per packet) and memory occupation (few checks are inserted per filter). Overall, the SPAF approach is an effective and simple way to generate packet filters that are easy to compose and efficient to run, even with increasing complexity.Among the potential problems, a widely known issue affecting specifically DFAs is an explosion occurring in the state space when treating certain critical patterns; this problem is the limiting factor for DFA adoption in other pattern-based detectors such as intrusion detection systems

The SPAF approach can be easily extended to perform packet demultiplexing in addition to packet filtering. This is partial supported by our current generator by labeling final states with identifiers of the matching filtering rules; full support would require dynamic automata creation and code generation, tasks that will be the object of future studies. Another future extension to SPAF could be enabling interactions (e.g., look-ups and updates) with stateful constructs such as session tables, useful for higher-layer filtering and traffic classification. In conclusion, SPAF has been shown as an approach that improves the state of the art by generating packet filters that combine most of the desired properties in terms of processing speed, memory consumption, flexibility and simplicity in specifying protocol formats and filtering rules, effective filter composition, and low run-time overhead for safety enforcement. The development of the filter generator and the test results support the viability of our claims.

V. References

[1] S. McCanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture," in *Proc. USENIX*, 1993, p. 2.

[2] A. Begel, S. McCanne, and S. L. Graham, "BPF+: Exploiting global data-flow optimization in a generalized packet filter architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 29, no. 4, pp. 123–134, 1999.

[3] M. L. Bailey, B. Gopal, M. A. Pagels, L. L. Peterson, and P. Sarkar, "PathFinder: A pattern-based packet classifier," in *Proc. Oper. Syst. Design Implement.*, 1994, pp. 115–123.

[4] O. Morandi, F. Risso, M. Baldi, and A. Baldini, "Enabling flexible packet filtering through dynamic code generation," in *Proc. IEEE ICC*, May 2008, pp. 5849–5856.

[5] L. Degioanni, M. Baldi, F. Risso, and G. Varenni, "Profiling and optimization of software-based network-analysis applications," in *Proc. 15th Symp. Comput. Arch. High Perform. Comput.*, Washington, DC, 2003, p. 226.

[6] S. Ioannidis and K. G. Anagnostakis, "xPF: Packet filtering for lowcost network monitoring," in *Proc. HPSR*, 2002, pp. 121–126.

[7] F. Risso and M. Baldi, "NetPDL: An extensible XML-based language for packet header description," *Comput. Netw.*, vol. 50, no. 5, pp. 688–706, 2006.

[8] L. Degioanni, M. Baldi, D. Buffa, F. Risso, F. Stirano, and G. Varenni, "Network virtual machine (NetVM): A new architecture for efficient and portable packet processing applications," in *Proc. 8th Int. Conf. Telecommun.*, Jun. 15–17, 2005, vol. 1, pp. 163–168.

[9] D. R. Engler and M. F. Kaashoek, "DPF: Fast, flexible message demultiplexing using dynamic code generation," in *Proc. ACM SIGCOMM*, New York, 1996, pp. 53–59.

[10] M. Jayaram, R. Cytron, D. Schmidt, and G.Varghese, "Efficient demultiplexing of network packets by automatic parsing," in *Proc. Workshop Compiler Support Syst. Softw.*, 1996.

[11] S. Kumar, J. Turner, and J. Williams, "Advanced algorithms for fast and scalable deep packet inspection," in *Proc. ACM ANCS*, New York, 2006, pp. 81–92.

[12] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley, and J. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection," in *Proc ACM SIGCOMM*, New York, 2006, pp. 339–350.

[13] M. Becchi and P. Crowley, "An improved algorithm to accelerate regular expression evaluation," in *Proc. ACM ANCS*, New York, 2007, pp. 145–154.

[14] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *Proc. ACM ANCS*, New York, 2006, pp. 93–102.

[15] M. Becchi and P. Crowley, "A hybrid finite automaton for practical deep packet inspection," in *Proc. ACM CoNEXT*, New York, 2007, pp. 1–12.

[16] R. Smith, C. Estan, and S. Jha, "XFA: Faster signature matching with extended automata," in *Proc. IEEE Symp. Security Privacy*, 2008, pp. 187–201.

[17] M. Becchi, M. Franklin, and P. Crowley, "A workload for evaluating deep packet inspection architectures," in *Proc. IEEE Int. Symp. Workload Characterization*, Sep. 2008, pp. 79–89.

[18] T. Hruby, K. van Reeuwijk, and H. Bos, "Ruler: High-speed packet matching and rewriting on NPUs," in *Proc. ACM ANCS*, New York, 2007, pp. 1–10.

[19] H. Bos, W. D. Bruijn, M. Cristea, T. Nguyen, and G. Portokalidis, "FFPF: Fairly fast packet filters," in *Proc. OSDI*, 2004, pp. 347–363.

[20] L. Deri, "nCap: Wire-speed packet capture and transmission," in *Proc. IEEE E2EMON*, Washington, DC, 2005, pp. 47–55.

[21] Z. Wu, M. Xie, and H. Wang, "Swift: A fast dynamic packet filter," in *Proc. 5th USENIX Symp. Netw. Syst. Design Implement.*, Berkeley,CA, 2008, pp. 279–292.