The Interdisciplinary Center, Herzliya
Efi Arazi School of Computer Science

Snort Virtual Network Function with DPI Service

M.Sc. final project submitted in partial fulfillment of the requirements towards the M.Sc. degree in computer science

by

Asher Gruber

This work was carried out under the supervision of

Prof. Anat Bremler-Bar

and

Mr. Yotam Harhol

December 2016

Acknowledgments

I would like to thank my advisor Prof. Anat Bremler-Bar for giving me the opportunity to work on such an interesting project which combines theory and practice. I would also like to thank Mr. Yotam Harhol for his guidance and support throughout the project. His wise advices helped me to face all the challenges I have encountered along the way.

Abstract

Deep Packet Inspection (DPI) is a widespread functionality among middlebox applications. In many cases, DPI is performed by Intrusion Detection Systems (IDS) such as Snort [14]. Traditionally, each packet is re-scanned by multiple middleboxes until reaching its final destination. Recent studies show that DPI is one of the most processing intensive tasks of modern middleboxes.

The DPI as a Service paper [4], presents a framework which allows to extract the time-consuming task of DPI out of the middleboxes while providing it as a service. Alongside with the framework design, the authors introduce a reference implementation on a simulated environment, while demonstrating promising results through a set of experiments.

In this work we have enhanced the reference implementation in order to demonstrate that the framework can operate in a more realistic environment setup. First, and foremost, we have integrated the framework with the commonly used Snort IDS. Second, we have extended the DPI Service to support the Network Service Header (NSH) [11] protocol which allows passing of the pattern match results with the inspected packet. These significant enhancements, transformed the reference implementation to a more robust system, which can take proactive measures in an event of malicious pattern detection.

Finally, once the work on the framework was completed we were able to perform the basic experiments which were reported in the original paper. Our findings (see 4) indicate, that the original framework results are reproducible in a our version of the framework.

Contents

List of Figures

1 Introduction

Deep Packet Inspection (DPI) involves the inspection of packet payloads in order to identify predefined sets of patterns. Often the DPI process is carried out by Intrusion Detection Systems (IDS) in order to detect malicious content within packet payloads. IDS are usually installed on middleboxes which reside across the network. In the common architecture, each packet is inspected from scratch by multiple middleboxes on its path until reaching its final destination. It was shown that for numerous middleboxes the IDS task is the most time-consuming and can take most of the processing time [4]. Snort [14] is a well known open source IDS which supports protocol analysis alongside with content searching and matching. It is widely adopted by the security community and used by various enterprises [13].

The DPI as a Service paper [4], suggests to extract the common task of DPI out of the middleboxes and provide it as a service to the middleboxes within the network. The service will then inspect each of the packets once, against the rules of all the middleboxes, and report the results. The middleboxes will consume the results, while avoiding the highly expensive pattern search phase, and focus on performing more complex processing based on to the pattern match results. As part of the paper the authors introduces a framework for deploying the service and provide a reference implementation on a simulation environment. The suggested framework shows promising results, but it is yet to be proven that the achievements can be reproduced in a more realistic environment setup.

The goal of this work is to integrate the DPI as a Service framework with the complex "real world" Snort IDS. The integration demonstrates that the framework can operate in a more realistic environment setup. In order to allow the integration, enhancements were introduced to the original framework and the Snort code base was significantly extended.

Although Snort is one of the most popular open source IDS, applying modifications to the Snort code base was not a trivial task. The code base is very complex and was written in order to support real time processing under strict performance requirements, which often led to highly unreadable code. In addition, the Snort project is missing basic code readability aspects such as comments and explanations. Therefore, integrating Snort with

the DPI Service required comprehensive investigations, experiments and literature reviews which span across online resources, books and articles.

The DPI as a Service framework implementation supports the deployment of multiple DPI services across the network which are all controlled by a centralized DPI Controller. The controller is responsible for managing the overall DPI process such as middlebox registration, pattern set management, and DPI service initialization.

The paper suggests three options for reporting the pattern matches results to the middleboxes 1) via adding an additional layer (e.g. NSH) to the inspected packet. This method allows maximal flexibility and the best performance. 2) via the usage of flexible pushing and pulling of tags (e.g. MPLS labels). This method is supported by various Software Defined Network (SDN) implementations, yet it can introduce complication due to the fact that each matching result may require multiple tags. 3) via dedicated packets. This method is only useful when the middleboxes are in read-only mode and does not allow to take action according to the matching results. The provided reference implementation uses the third option for pattern match reporting, due to limitations introduced by it's underling SDN infrastructure. This solution is not applicable for most environments which require the IDS to take action in an event of malicious pattern detection.

Moreover, the framework reference implementation does not use a "real" IDS (e.g. Snort), but rather a simple IDS prototype which only counts the number of matched rules that were reported by the DPI service. This makes it hard to perform a real evaluation of the suggested framework, since "real" IDS do much more then merely count the results.

As part of the project, we have extended the DPI Service to support the Network Service Header (NSH) [11] protocol which allows passing the pattern match results with the inspected packet. The Snort IDS was extended to support the NSH protocol in order to allow integration with the DPI Service. In addition, Snort was enhanced in order to apply its rules without the need to re-scan the packet, while leveraging the match results reported by the DPI Service. Snort was furthermore enhanced in order to allow its registration to the DPI Controller. Finally, the Snort configuration mechanism was extended in order to allow control over the activation and settings of the DPI Service integration.

In addition to integrating Snort with the DPI Service framework, we analyze our implementation in order to conclude if the framework can perform in a "real world" environment. By repeating a subset of the original paper experiments, we show performance improvements of at least 38% in throughput and 16% in latency. These preliminary results support the promising results which were described in the original paper.

The structure of the remaining of this work is as follows: In section 2 we introduce the background for this work. Section 3, describe the code modifications which were performed as part of the project. Then we move to evaluate our implementation thorough as set of experiments which are presented in section 4. Finally, we conclude in section 5.

2 Background

2.1 Deep Packet Inspection (DPI)

Deep Packet Inspection (DPI) involves the inspection of packet payloads in order to identify predefined sets of patterns. Network Intrusion Detection Systems (NIDS) perform DPI in order to detect malicious content within packet payloads routed through the network. Snort is a well known open source NIDS which supports packet payload searching and matching.

Patterns searched during the DPI process usually consist of either strings or regular expressions. String matching is a core component in most DPI engines. In many implementations (such as Snort [14]), string matchings is used during the identification process of both pattern forms. When patterns are regular expressions, string matching is performed first as a pre-filter and constitutes most of the work performed by the engine. This procedure is commonly used, since regular expression engines are inefficient in comparison to string matching engines.

2.2 Snort Overview

Snort is an open source Network Intrusion Detection system (NIDS) created by Martin Roesch in 1998. It is one of the most widely spread open source IDS and supports protocol analysis alongside with content searching and matching. It is widely adopted by the security community and used by numerous enterprises.

Snort can be configured to run in three modes: Sniffer mode, which reads and displays packets of the network traffic. Packet Logger mode, which logs the packets to disk. NIDS mode, which performs detection and analysis of network traffic against a predefined set of rules. Once a rule is matched, Snort can take specific actions defined by the matching rule.

2.2.1 Packet Processing Flow

Network packets processed by Snort all follow a similar flow: packets are captured, decoded, preprocessed, rules are applied, and actions are taken.

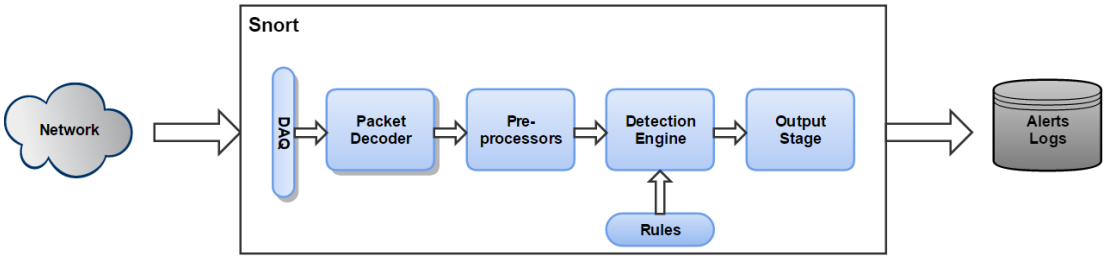


Figure 1: Packet processing flow

2.2.2 Packet Acquisition

Packet acquisition in Snort is performed via the Data AcQuisition library (DAQ) which provides an abstraction layer above system dependent packet capturing modules (such as libpcap). Snort supports packet capturing from a network interface or a provided pcap input file [1]. Snorts main control flow is built around the packet loop that reads packets acquired by the DAQ layer.

2.2.3 Packet Decoding

Following the capturing of a packet it is decoded according to it's network protocol stack. Based on the link layer protocol the packet is decoded layer by layer (e.g. Ethranet, IP, IPv4, TCP). The decoding process is modular and allows the addition of layer decoders over time. Snort also supports encapsulation protocols via it's Layer 3 and Layer 4 decoders. However, only one level of encapsulation is supported, due to the limiting design of the packet data structure.

2.2.4 Packet Preprocessing

Once the decoding step is completed the packet is handed over to the Snort preprocessors. The preprocessors can perform a variety of operation, but usually perform packet checks and alerting, packet data normalization, and addition of rule options to the static detection layer rules (e.g. the HttpInspect preprocessor normalizes URIs which are HTTP-specific).

2.2.5 Snort Detection

The Snort detection engine is focused around identifying predefined sets of attack patterns. Snort rules allow users to define their specialized NIDS policies using a simple and flexible syntax.

Snort Rule Engine Snort rules start with a header which includes an action (log, alert, drop, etc.), protocol (ip, tcp, etc.), direction (source, destination) and endpoints (address and port). The header is then followed by a body which consists of rule metadata and evaluation options [13]. Rule options are used in order to identify a specific packet. Among all options, the content and PCRE [10] are the most important from a DPI perspective.

The content option allows to match a specific content within a packet payload. The content may be defined as an ASCII string or binary data in the form of hexadecimal characters. Whenever a specified content is exactly found withing the packet payload additional content related options are evaluated.

The PCRE option allows to search a packet payload for a pattern using a regular expressions syntax. Upon execution, Snort extracts the strings that appeared within the PCRE regular expression (called anchors) and uses them in order to perform regular content search. Only after all anchors are matched, Snort actually evaluates the regular expression against the packet payload. Snort follows this process, since the regular expression evaluation engine is relatively inefficient. Whenever the content and PCRE options reside together in the same rule, Snort first performs the content and the anchor search and only then the regular expression is evaluated.

A common Snort setup usually includes a vast number of rules, hence the rule engine performance is extremely important. In order to allow quick evaluation of packets, the rules are grouped by protocol, direction and ports and stored in special data structures which support real-time mapping of a packet to a rule group. Then for each group of rules a dedicated Aho-Corasick (AC) DFA is constructed (see 2.2.5) in order to enable Snort to quickly determine which of the rule patterns are matched withing a given packet payload. Following the construction of the AC DFA it is wrapped with the MPSE (see 2.2.5) which

is associated to the rule group.

```
alert tcp 192.168.1.0/24 any -> any any (content: "POST"; msg: "POST matched";)
```

Figure 2: Snort Rule Example

Multi-Pattern Search Engine In order to perform pattern matching on real-time network traffic, Snort uses its Multi Pattern Search Engine (MPSE). The engine was introduced in 2002, as part of a performance improvement initiative. The core of the MPSE is the implementation of the Aho-Corasick (AC) algorithm which is used to perform multi-pattern searches [8].

Aho-Corasick (AC) is a string matching algorithm [2] which is commonly used by NIDS. It matches multiple strings simultaneously by first constructing a DFA that represents the pattern set; then, using the DFA, the algorithm scans the text in a single pass. Whenever an accepting state is reached a pattern from the set is matched. An efficient implementation of the algorithm is crucial, since it can significantly impact the overall performance of Snort.

The AC algorithm is used by the MPSE in order to perform DPI for both string and regular expression patterns. The AC DFA is constructed from the patterns defined withing the rules provided to Snort. The AC implementation is used to support both the content and PCRE rule options. In both cases the DFA is constructed from the strings provided or derived (PCRE string anchors) from the options, and used to perform serial text search in a single pass.

In the Snort AC DFA each accepting state corresponds to a pattern match associated to at least one rule (multiple rules can have the same pattern). During the AC DFA construction whenever a rule has more then one pattern defined (multiple contents or anchors) the longest content is added to the DFA as a representative of that rule. Choosing the longest content is useful, since a long content is less likely to be found and therefore allows quicker rule elimination.

During the detection phase, after the packet is mapped to a rule group it is searched using the group's AC DFA. Once an accepting state is reached, the state's corresponding rules are sent for advanced examination. The rules are then re-evaluated against the packet

in order to verify that all the DPI content options which were not included in the DFA search are met (e.g. additional content options, PCRE regular expression, etc.). Finally the matched rules are added to the event queue (see Section 2.2.5).

2.2.6 Snort Output

During the detection phase of a given packet, all output actions (log, alert) defined by the matching rules are collected in an event queue. Once the detection step in over, the actions in the queue are triggered. The event queue also provides a set of more complex functionalities such as rate (limit the number of events) and threshold (perform an action only after an event occurred more then a given threshold) which are used to control Snort's output.

2.2.7 Snort Configuration

The Snort distribution includes the snort.conf file which is loaded at startup in order to configure Snort. The file allows to control various configurations such as rule operations (e.g. path to rule files), activation of components/plug-ins (e.g. preprocessors) and much more. The default configuration file supports common use cases, but is usually customized in order to adapt it to the environment in which Snort is deployed.

2.3 Service Chaining

The term Service Chaining is used to describe a modern way for deploying and delivering composite services. A service chain can be viewed as a set of service functions a packet needs to go through, before reaching its final destination (e.g. a packet need to go through a NIDS). The services are commonly assembled while leveraging technologies such as Software Defined Networks (SDN). In order to implement the service chain in a network, a service path is required. The service path is created using a network overlay topology which connects between the actual service functions instances [12].

Service chaining enables an elastic, simple and modern service deployment model by treating service functions as resources which can be scheduled and consumed. Network packets, are simply "steered" to the next required service function in the path, along with metadata that can be used by the service function.

Although service chaining is currently achieved using various technologies, they all suffer from known limitations [6], mainly because of their tight coupling with the underlying network topology. In order to address these limitation the NSH protocol was proposed.

2.3.1 Network Service Header (NSH)

NSH [11] is a data plane header format which is added to frames/packets in order to create a dedicated service plane that is independent from the underlying transport layer. NSH provides the decoupling of the service topology and the actual network topology.

The dedicated service plane created via NSH enables the following:

1. Topological Independence - Service forwarding occurs within the service plane, the underlying network topology does not require modification. NSH provides an identifier used to select the network overlay for network forwarding.
2. Service Chaining - NSH enables service chaining, since it contains path identification information needed to realize a service path. Furthermore, NSH provides the ability to monitor and troubleshoot a service chain end-to-end.
3. NSH provides a mechanism to carry shared metadata between participating entities and service functions.
4. Classification and re-classification - sharing the metadata allows service functions to share classification results with downstream service functions saving re-classification, where enough information was enclosed.
5. NSH offers a common and standards-based header for service chaining to all network and service nodes.

6. Transport Agnostic - NSH is transport independent. An appropriate network transport protocol can be used to transport NSH-encapsulated traffic.

A NSH contains metadata and service path information that are added to a packet or frame and used to create a service plane. The packets and the NSH are then encapsulated in an outer header for transport. The service header is added by a service classification function (a device or application) that determines which packets require servicing, and correspondingly which service path to follow to apply the appropriate service.

In order to exchange NSH packets between participating network nodes an NSH-aware control plane is required. The control plan is responsible for providing the network nodes with requisite information such as service path ID to overlay mapping. NSH does not mandate a rigid control protocol in order to operate, which leaves room for innovation. Any existing control plane can simply add support for service chaining via NSH. For example, the open source OpenDaylight (ODL) [9] Service Function Chaining project provides an SDN approach to service chaining via a controller.

See the NSH Protocol Format appendix for more details.

3 Implementation

In order to demonstrate that the DPI as a Service framework works with a complex "real world" middlebox we have integrated the existing DPI Service components (DPI Controller and DPI Service) with Snort.

Upon initialization Snort registers itself to the DPI Controller using a dedicated message. The message includes a list of all the rules which were loaded into Snort according to the policy configuration. The DPI Controller then registers the rule patterns to the DPI Service which will use them in order to report matched rules on a packet metadata using NSH.

Now whenever a packet requires DPI it will be scanned by the service which will report all detected patterns as metadata with the packet. The packet will then continue it's path according to the service chain until it reaches the Snort middlebox. Snort will retrieve the DPI Service match results and apply it's policies without the need to scan the packet from scratch using it's traditional pattern matching engines.

In order to allow the integration both Snort and the DPI Service where enhanced. The DPI Service was enhanced to support the NSH protocol and can now pass the pattern match results with the inspected packet. Snort was enhanced in order to apply its rules without the need to re-scan the packet, while leveraging the match results reported by the DPI Service. Snort was furthermore enhanced in order to allows it's registration to the DPI controller. In order to support the change, the Snort code base was extended significantly and modifications were performed across the standard packet processing flow and initialization phase.

The following sections describe the changes which were performed in each of the components in order to support the enhancements.

3.1 NSH Support

We have decided to report the DPI Service match results using the NSH protocol which will be added as an additional layer to the inspected packet. NSH was chosen, since it provides

(among other benefits) a mechanism for metadata exchange along the packets service path. We will be using this mechanism in order to transfer the pattern match results between the DPI Service and Snort.

NSH supports two types of metadata: type 1, which is used for fixed length metadata; type 2 which allows a variable length metadata size. We choose to use type 2, since we will be using the NSH metadata section to send the report matched rule information which is of variable length.

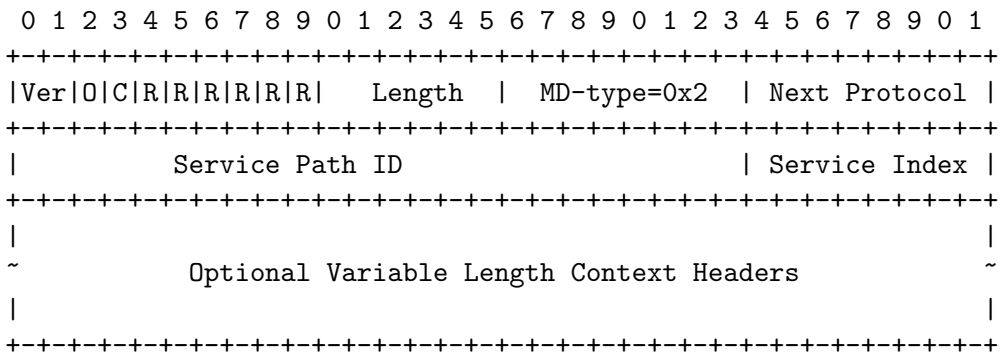


Figure 3: NSH MD-type=0x2

When NSH is added to a packet, an outer encapsulation is used to forward the original packet and the NSH through the service chain. Although multiple encapsulation protocols are possible, we choose to use VXLAN-gpe (see Figure 4), since it is described in the IETF [11] in more details then other protocols.

The VXLAN RFC [5] specifies that the protocol should be encapsulated with an outer UDP header with a set destination port of 4789, an outer IP header and an outer Ethernet header. In our implementation we use the well known UDP destination port in order to support NSH in Snort and the DPI Service.

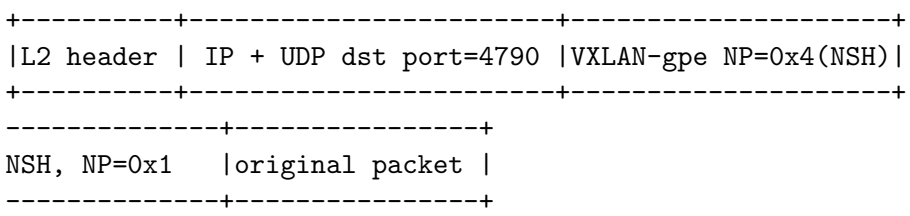


Figure 4: VXLAN-gpe + NSH

3.2 DPI Service

The original implementation of the DPI Service used a dedicated packet in order to pass patten match results. We have extended the implementation in order to allow the reporting of pattern match results withing the inspected packet using the NSH protocol. The code is available at https://github.com/gruberasheridc/moly. The original implementation mode is still supported and can be activated by changing a single flag (see `USE_NSH` flag in the `Sniffer.c` file). In both modes, whenever the searched patterns are not found within the inspected packet it is forwarded as it was received.

When the DPI Service is initialized it is provided with a list of rule structures (see Figure 5) for whom to search for patterns. When a pattern is matched for a given packet the service reports the match using the rule id (`rid`) which was provided in the relevant rule structure. In our implementation we always provide the DPI Service rules with the original Snort rule ID, in order to simplify the integration with Snort.

```
{className:'MatchRule',rid:17236,pattern:'-moz-column-',is_regex:false}
{className:'MatchRule',rid:20607,pattern:'gwagents|2E|css',is_regex:false}
```

Figure 5: DPI Service Input Rule Format

Once the DPI Service finds matching patterns within an inspected packet, the packet is encapsulated with the NSH protocol (see the `build_nsh_result_packet` function, in the `Sniffer.c` file). First the matching results are aggregated and transformed to the relevant match report structure. Then the NSH is constructed with the match report results lined in a sequence as the metadata (MD-type=0x2). In addition, during the NSH construction

all the necessary protocol fields are set. It is important to note that we always set the NSH next protocol to IPv4.

Following the NSH encapsulation, the result is encapsulated with VXLAN-gpe and the next protocol field is set to NSH. The result is then encapsulated with an outer UDP header with the destination port always set to 4790. The result is once again encapsulated with an outer IP header and an outer Ethernet header as the VXLAN protocol defines. We set both the IP header and the Ethernet header with the values of the corresponding layers of the original packet.

In the original implementation each rule match is reported in a separate result structure, even if the same rule pattern is found multiple times in a sequence. In our implementation we introduce the ability to report a range of rule matches by supporting two result structures. One for a single rule match (see Figure 7) and an additional structure for a range of rule matches (see Figure 6). Supporting the report range match allows us to minimized the encapsulated packet as much as possible. The only difference between the result structures is that the range structure includes an additional field at the end (`length`). This design allows our code to be more efficient when processing the report match results. In order to represent the rule id in the structures we have decided to use a new type (`rule_id_t`). The type allows to choose the underling native C type which is used in order to represent the rule id. Out-of-the-box we have decided to use `uint16_t` in order to save space, but if needed this default can be changed as desired. The construction of the result structures can be found in the `find_detection_results` function, in the `Sniffer.c` file.

```
typedef struct {
    rule_id_t rid;
    uint8_t  is_range;
    uint16_t position : 15;
    uint16_t length;
} MatchReportRange;
```

Figure 6: Report Range Match Structure

```
typedef struct {
    rule_id_t rid;
    uint8_t is_range;
    uint16_t position : 15;
} MatchReport;
```

Figure 7: Report Single Match Structure

All the logic that was added in order to implement the NSH support can be found in the `Sniffer.c` file. In order to support the logic, multiple structure were added. These structures can we roughly divided into two groups: NSH structures and report match structures. The NSH structures are corresponding to the NSH protocol specification (see A) and are located under the `/Common/NSH` folder. The report match structures are located under the `Sniffer` folder. We use the exact same structures in our Snort implementation in order to support the NSH protocol and the metadata decoding.

3.3 Snort

In order to integrate Snort with the DPI Service and DPI Controller multiple enhancements were required.

We have extended Snort to support communication with the DPI Controller during the initialization phase. The communication is required in order to register the Snort instance to the DPI Controller which in return registers the Snort rule patterns to the DPI Service.

Snort was furthermore extended to supports the NSH protocol in order to allow processing of the DPI Service pattern match reports which are now sent withing the inspected packet using the NSH protocol. Since we have decided to use VXLAN-gpe as our NSH encapsulation protocols, adding support for VXLAN was essential for supporting NSH. Therefore, Snort was extended to support VXLAN.

Enabling Snort to apply its rules without the need to re-scan the packet, while leveraging the match results reported by the DPI Service, required multiple modifications to the standard packet processing flow (e.g. decode). Additional changes were introduce to the

Snort initialization phase in order to support the packet processing modifications and increase performance.

Finally, the Snort configuration mechanism was extended in order to allow control over the activation and settings of the DPI Service integration.

The sections below describe in more details the various enhancements which were introduced to the Snort code base. The code is available at https://github.com/gruberasheridc/snort-2.9.8.0.

3.3.1 Snort DPI Service Configuration

Snort support for the DPI Service framework can be controlled via the Snort configuration file (i.e. `snort.conf`). The section seen in Figure 8 was added to the configuration file in order to support the various options using the common Snort configuration directive syntax.

```
config dpi_service: dpi_rule_export_mode file
   dpi_controller_ip 127.0.0.1, \
   dpi_controller_port 3000, \
   dpi_rule_export_dir /home/asher/packet_dump/
```

Figure 8: Snort DPI Service Configuration

The `dpi_service` config directive is used to activate the support for the DPI Service framework. The directive also instructs Snort to parse the remaining DPI Service configuration options. The `dpi_rule_export_mode` option allows to configure how Snort will be exporting it's rules (in a format supported by the DPI Controller). Three export modes exist:

1. `file` which directs Snort to write the rules to the path defined by the `dpi_rule_export_dir` option.
2. `controller` which directs Snort to send the rules to the controller network location defined by the `dpi_controller_ip` and `dpi_controller_port` options.
3. `console` which directs Snort to write the rules to the console.

The DPI Service configuration parsing logic was incorporated into the standard Snort configuration parsing phase which is performed during Snort initialization (See the `ConfigDPIaaService` function in the `parser.c` file). The logic parses the configuration options and sets their corresponding values to dedicated variables which were added to the global Snort configuration structure (see Figure 9). The variables are used across the implementation to enable and control the added DPI Service support.

```
typedef struct _SnortConfig
{
    ...
    bool dpi_service_active;
    DPI_EXPORT_MODE dpi_export_mode;
    char *dpi_controller_ip;
    int dpi_controller_port;
    char *dpi_rule_export_dir;
    ...
} SnortConfig;
```

Figure 9: SnortConfig DPI Service variables

3.3.2 Controller Communication

In order to implement a more realistic environment setup Snort was extended to support communication with the DPI Controller. According to the framework suggested by the DPI as a Service paper, the middleboxes (i.e. Snort) are responsible for registering themselves to the DPI Service via the DPI Controller. The communication between the middlebox and the DPI Controller is achieved by exchanging JSON messages. The DPI Controller address is pre-configured and known to the middlebox (see 3.3.1), so is the middlebox's unique identifier.

As part of the project, we have extended Snort to allow the sending of a registration message to the DPI Controller (see 3.3.1 for configuration options). The JSON registration message (see Figure 10) includes the middlebox's ID (`snort_id` property) and a list of all the rule patterns (`rules` array) which are registered to Snort via the rule mechanism. Each of the rules includes the string pattern searched by the rule (`pattern` property), the rule's

identifier (`rid`) and additional properties. The message contains all the information needed for the DPI Controller in order to register the middlebox's rules to the DPI Service and for the DPI Service to report the matched rules on a packet metadata using NSH.

```
{
    "snort_id": "master_snort",
    "rules": [{
        "className": "MatchRule",
        "pattern": "-moz-column-",
        "is_regex": false,
        "rid": 17236
    },
    {
        "className": "MatchRule",
        "pattern": "gwagents|2E|css",
        "is_regex": false,
        "rid": 20607
    }]
}
```

Figure 10: Snort Registration message

In order to construct the registration message, knowledge of the loaded rules is required. We have decided to leverage the special rule mapping data structures (see 2.2.5) which are constructed during Snort initialization in order to obtain the list of rules. Though there are easier ways to obtain the list of Snort rules, using these data structures allows us to get access the MPSE instance of each rule group. We are then able to access the AC DFA of the group and extract the rules and their corresponding patterns by traversing the accepting states. Finding the rules via the AC DFA is required, since we want to send to the DPI Controller only the pattern which are actually searched by the AC DFA and not all the patterns which were registered with each rule (see 2.2.5). Sending the AC DFA patterns is essential, since we want to bypass the AC DFA execution by Snort and use the rule match results sent by the DPI Service instead.

Once the registration message is built, it is sent by Snort according to the settings provided in the configuration file (see 3.3.1).

The construction and sending of the registration message was added to the Snort initialization phase (see the `RegisterContentRulesToDPIController` function in the `snort.c` file). The operations are performed only in case the DPI Service functionality was activated in the configuration file.

3.3.3 DPI Service Integration

Enabling Snort to use the DPI Service pattern match results in order to avoid re-scanning of the packet payload, required multiple modification in the Snort code base. The section below describe these modifications in depth.

Snort Initialization During Snort initialization a registration message is sent to the DPI Controller (see 3.3.1) in order to register the Snort rule patterns to the DPI Service. The registration information allows the DPI Service to search for the given patterns and notify upon their finding using the Snort rule IDs (see 3.1).

Traditionally when packets are processed by Snort in order to perform DPI they are scanned using the MPSE and more specifically the AC DFA. We would like to avoid the need to re-scan the packet from scratch using the AC DFA, since it has a significant impact on the overall performance of Snort [8]. Snort still needs to have the ability to perform other DPI operations using the MPSE and AC DFA, since finding a matched pattern using the AC DFA does not guarantee a rule match. The packet still needs to be re-evaluated against the additional rule options which are associated to the matched DFA accepting state in order to verify that all the DPI content options of the rule are met (see 2.2.5).

In order to take advantage of the DPI Service results, while skipping the AC DFA search, a mapping between a Snort rule and its associated AC DFA accepting state is required (see 3.3.3). Having the mapping will allow Snort to use the accepting state in order to perform the addition DPI rule operations which are required for every matched rule that is reported by the DPI Service. The creation of the mapping is performed during the Snort initialization phase while constructing the registration massaged, since it involves traversing the same data structures (see 3.3.1).

The registration message is built while visiting the AC DFA accepting states associated to each of the rule groups (one AC DFA per rule group). Each of the DFA accepting states represents a pattern of at least one or more rules (see 2.2.5). Obtaining an accepting state provide access to the associated rules and pattern. During the registration construction whenever a rule-pattern pair is added to the message the rule-accepting state pair is added to the mapping.

Since every rule group has a dedicated AC DFA, a rule-accepting state map will be created per rule group (or rule group AC DFA). Therefore another mapping is created in order to represent the association between rule group and it's rule-accepting state map. The additional mapping is essential, since it is possible that the DPI Service will report a rule match within a packet that does not meet the packets rule group. Therefore having a rule-accepting state map which is shared amount all rule groups will potentially cause false positive rule match alerts. Once the map creation is complete, the results are stored in a dedicated variable which was added to the global Snort configuration structure (see Figure 11). The variable is then used during the packet detection phase (see 3.3.3) in order to fetch an accepting state for a given rule which was matched by the DPI Service.

```
typedef struct _SnortConfig
{
    ...
    SFGHASH *dpi_acsm_map;
    ...
} SnortConfig;
```

Figure 11: SnortConfig map from AC SM to a map of rule ID to AC DFA accepting state

Snort Packet Processing Allowing Snort to take advantage of the match results reported by the DPI Service, required multiple changes to the standard Snort packet processing flow.

Decoding Traditionally when a packet is captured by Snort it is decoded according to it's network protocol stack (see 2.2.2). We have extended the Snort decoding functionality

in order to support the NSH protocol which is required in order to allow integration with the DPI Service. Due to the fact that we have decided to use VXLAN-gpe as our NSH encapsulation protocols, adding support for VXLAN was also required in order to support the NSH decoding. While implementing our decoding enhancements we have used the same structure model that was used in the DPI Service implementation to allow maximal compatibility. The protocol model declarations can be found in the `decode.h` file and the added decoding logic can be found in the matching `decode.c` file.

According to the VXLAN RFC [5] the protocol is always encapsulated with an outer UDP header with a destination port set to 4789 (see 3). Based on this definition we have extended Snort's UDP decoding function to support the VXLAN protocol (see the `DecodeUDP` function). Once all the standard UDP decoding operations are completed we check if the packet destination port is equal to 4789; If so, we start to decode the VXLAN layer (see the `DecodeVxLAN` function). When the layer is decoded we verify that the next protocol field was set to NSH. In case it was, we continue to the NSH protocol decoding (see the `DecodeNSH` function).

We start by decoding the NSH Base Header and validating that the metadata type is equal to 2 (see 3), since we use the metadata for reporting on pattern match results which are of variable length. Then we start decoding the pattern match reports in sequential order. Each decoded match report (See 7 and 6) is added to a list which is saved on the standard Snort packet structure (see 12). We later use the list of match reports to bypass the AC DFA search during the packet detection phase.

```
typedef struct _Packet
{
    ...
    SF_LIST *dpi_service_match_reports;
    ...
} Packet;
```

Figure 12: Packet DPI Service Match Reports List

When the decoding of the match results is completed we move to decode the original inner packet which was encapsulated with NSH. First, we check the NSH next protocol field

which was set according to the protocol of the inner packet; Then we call the appropriate Snort decoding function (e.g. DecodeIP). Calling the decoding function hooks us back to the standard Snort decoding stack which will decode the inner packet. Once the inner packet decoding is completed, the Snort standard packet structure (see 12) will be fully populated with the packet decoded information and will also include the DPI Service match report list.

Detection Following the decoding phase, the decoded packet continues it's journey through the packet processing flow (see 3.3.3) to the detection phase. The detection phase is responsible for determining if the packet's payload contains any of the rule patterns which were registered to Snort. Whenever a pattern is found within the payload it's corresponding rules are registered to the event queue which is evaluated during the output phase (see 2.2.5).

In order to detect patterns within the packet payload, Snort leverages it's MPSE (see 2.2.5) which internally uses an AC DFA. The AC DFA allows Snort to quickly determine which of the rule patterns are matched withing a given packet payload (see 2.2.5). The usage of the AC DFA has a significant impact on the overall performance of Snort. Hence, we would like to limit it's usage as much as possible. Therefore, we have modified the detection phase in away that allows the MPSE to take advantage of the rule match results provided by the DPI Service instead of re-scanning the packet using the AC DFA.

When a packet arrives to the detection phase (see the `Detect` function in the `detect.c` file) it is first analyzed in order to identify its rule group (see 2.2.5). Once the rule group is obtained the packet continues to the next evaluation step (see the `fpEvalHeaderSW` function in the `fpdetect.c` file). During this step packets which require DPI are sent to the MPSE for comprehensive content analysis based on their rule group (see the `mpseSearch` function in the `mpse.c` file). We have modified this step, so that if the DPI Service functionality is active, the packet is sent to an alternative content analysis that avoids re-scanning the packet using the AC DFA (see the `mpseSearchDpiSrv` function in the `mpse.c` file). Once the content analysis is completed the packet continues through the standard packet processing flow.

The introduced content analysis functionality leverages the DPI Service results sent with the packet (see 3.3.3) alongside with the rule data structures that were prepared during Snort initialization (see 3.3.3). We start our analysis by checking if any rule match results were associated with the packet data structure during the decoding phase (see 12). If no matches were reported, we complete the analysis and conclude that no matches were found. Otherwise we continue the analysis in order to determine if any of the reported pattern matches actually satisfy the rule definition. This operation is required, since the packets are initially scanned for one pattern per rule (or accepting state). Therefore rules are not considered matched until all their DPI options are met (see 2.2.5).

The analysis continues by first obtaining the packet's MPSE instance which provides access to the rule group's AC DFA. Using the AC DFA we fetch the rule-accepting state map which is relevant to the packet (see 3.3.3). The map will allow us to find the AC DFA accepting state which is associated to each of the rules. Next we go over each of the reported rule match results and obtain their matching accepting state. In case the accepting state is found in the map, we call the standard Snort Match operation which required information provided by the accepting state. The Match operation is also called in the standard Snort analysis functionality whenever an accepting state is reached in the AC DFA search. By calling the Match operation we assure that all the DPI content options which were not included in the DPI Service search are met before the rule is considered as matched. Once the Match operation concludes that all the options of a given accepting state are met it registrars a match event to the queue. The analysis functionality ends when all the rule match results are evaluated. Then the packet continues through the standard Snort packet processing flow to the output phase.

3.4 Rule Parsing Tool

As part of this project, Snort was extended in order to register itself (and it's rules) upon startup to the DPI Controller. The extension exports the rules in a format which is supported by the DPI Controller. The Controller is then responsible for sending these rules (along with additional information) to the relevant DPI Service instance in a different format which is supported by the DPI Service. Due to the fact that the integration between

the DPI Service and the DPI Controller was not yet implemented, there was a need to develop an ad hoc parsing tool, which will convert the rules exported by Snort from the DPI Controller format to the DPI Service format. The conversion allows to perform comprehensive framework experiments until the integration between the DPI Controller and the DPI Service will be implemented.

The tool was implemented as a Node.js [7] application and is available at https://github.com/gruberasheridc/dpi-snort-rule-converter.

4 Experimental Results

In this section we evaluate the performance of the DPI as a Service framework after it was integrated with the Snort NIDS (see 3). Moreover, we compare the results, to those of the original paper (see [4, Section 6]). Repeating the original experiments will allow us to evaluated if the promising results presented in the paper, can be reproduced in a more realistic environment setup. It is important to note that we have only attempted to repeat a partial set of the original experiments.

4.1 Experimental Environment

All the results presented in the section were performed on a machine with Intel Xeon E3-1270 v3 CPU, quad-core, each core having two hardware threads, 32 KB L1 data cache (per core), 256 KB L2 cache (per core), and 8 MB L3 cache (shared among cores). The system runs Linux Ubuntu 14.04 LTS (Trusty). While conducting the experiments we used 3498 rules from Snort [14]. The DPI Service experiments were performed using exact-match patterns of length 4 characters or more which were extracted from the rules and sent by Snort to the DPI Controller as described in section 3.3.1. The middlebox/Snort experiments were performed using the original Snort rules. As input traffic we used a 148MB HTTP traffic trace crawled from most popular websites [3]. Both trace and rules are similar to the ones that were used in the original paper.

In our experiments we concentrate on evaluating the integration between the DPI Service and Snort which is the main contribution of this work. Since our focus is on the data plane, the measurements presented in this section where performed on an environment setup which consisted of the DPI Service and Snort, without including the DPI Controller due to time constraints. As the DPI Controller was not part of the experiments, we had to convert the Snort messages which were sent to the controller, to messages which are acceptable by the DPI Service. The conversion was done using a dedicated utility program we developed (see 3.3.3).

4.2 Reproducing the Pipelined Middlebox Scenario

In order to verify that the results of the original paper can be reproduced in a more realistic environment setup, we decided to repeat the pipeline middlebox scenario experiment from the original paper [4]. We have conducted the same experiment while swapping the middlebox simulation with our version of Snort and the original DPI Service with our enhanced version which supports the NSH protocol.

As originally performed, we took the patterns of Snort and randomly divided them into two sets, A and B, simulating a configuration where we have two standalone middleboxes, Snort1 and Snort2. In this setting, traffic should go through a pipeline of the two middleboxes, one with pattern set A and the other with pattern set B. We first evaluate this setting while the middleboxes run a vanilla version of Snort. We then compared the throughput results of the vanilla Snort to a setup consisted of two virtual DPI instances running on two machines simultaneously, where the load is equally distributed between them. The second setting was evaluated while the middleboxes run our version of Snort which takes advantage of the DPI Service match report results.

Figure 13 shows the comparison between the two setups in the form it was presented in the original paper [4, Figure 9 (a)]. Looking at the figures it is evident that we were able to reproduce the results of the original paper. Both experiments clearly demonstrate that the virtual DPI solution is superior in relation to the two separate middleboxes setup.

It should be noted, that in the original experiments the DPI solution was at least 86% faster then the two separate middleboxes, while in our experiment it was at least 38% faster. This difference can be explained, due to the fact that the IDS reference implementation which was presented in the paper only counted the number of matched rules that were reported by the DPI service. Snort on the other hand, does much more then merely count the results such as performing advanced analysis per rule match and taking actions in case all rule options are met. Therefore the impact of extracting only the exact-pattern match logic to the DPI Service has a smaller (though significance) impact on the overall performance of Snort.

In addition to comparing the throughput, we also compared the latency of both setups.

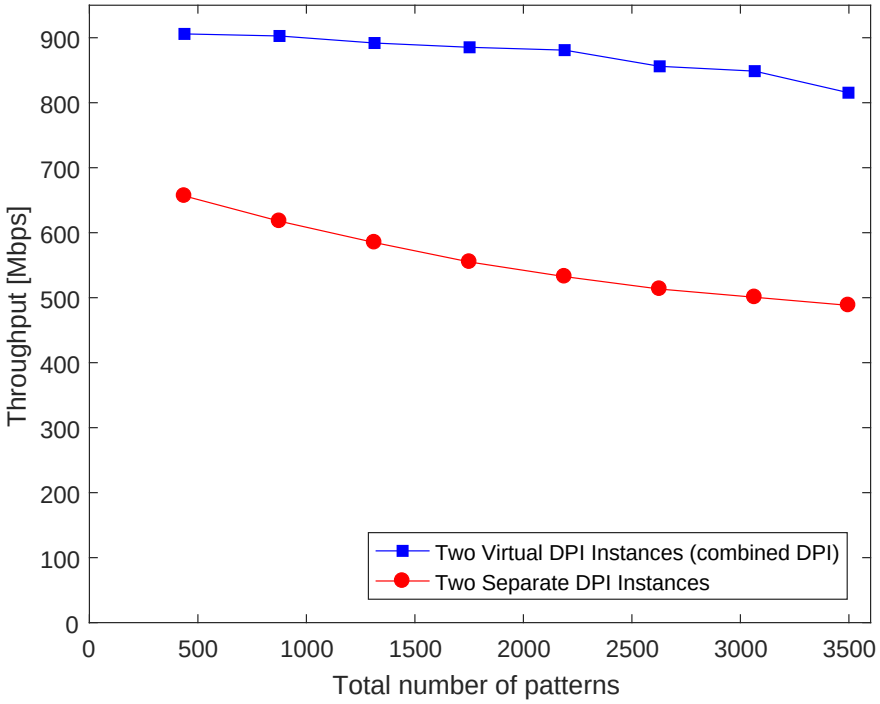


Figure 13: Comparing the throughput that can be handled by two pipelined middleboxes, and by our Virtual DPI

The latency measurements were taken when the total number of rules was maximal (i.e. 3498 rules). The latency of the two separate middleboxes setup was 32.73 μs/packet, while the latency of the two virtual DPI instances was 27.57 μs/packet. This translates to 16% improvement in latency in this scenario.

4.3 Virtual DPI Performance

As part of the Virtual DPI performance analysis which was conducted in the original paper [4, Section 6.3 (a)], it was found that the middlebox DPI applications operate much faster then the virtual DPI instances. In contrast to that, our experiments show that the middlebox which run the Snort NIDS are in fact a bottleneck in the system from both a latency and throughput perspective in relation to the DPI Service.

Figure 14: Comparison between the throughput of the Snort middleboxes and the DPI Service in the pipeline middlebox scenario

Figure 14 shows a comparison between the throughput of the Snort middleboxes and the throughput of the DPI Service in the pipeline middlebox scenario (see 4.2). The figure presents the information which was the basis for the first data point of Figure 13, where the total rule number of patterns is equal to 438. It is important to note that while each of the Snort instances was operating with a pattern set of 219 rules (total of 438 for both Snort instances), the DPI Service was operating with the maximum set of 3498 rules.

Looking at the figure it is clear that the middleboxes which run the Snort NIDS (i.e. Snort1, Snort2) are a bottleneck in the system. Although the figure shows the analysis of only the first data point from the pipeline middlebox scenario, we can deduce that this system behavior is consistent, since the point correlates to the maximal throughput measured from the middleboxes throughout the experiment. From this point forward the middleboxes throughput continued to decrease while the number of rules increased.

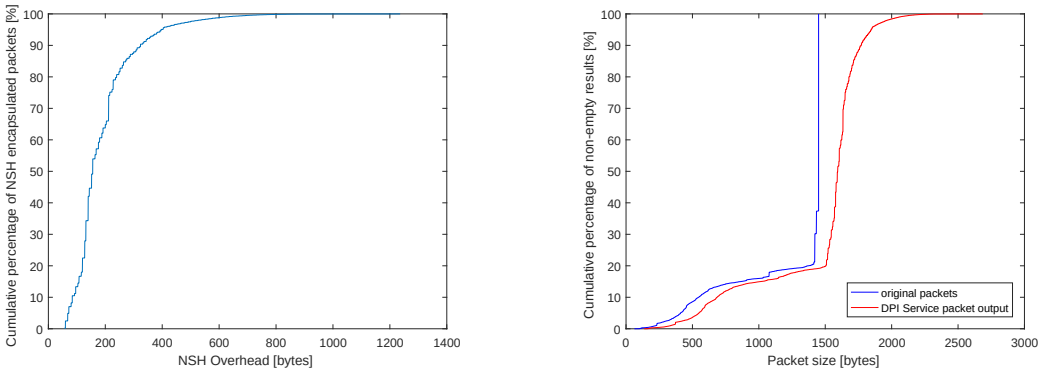
The fact that our DPI performance results conflict with the original paper can be explained, due to the fact that the original analysis used a simulated middlebox DPI application which was only counting the number of matched rules that were reported by the DPI Service. In contrast, our analysis used Snort as the middlebox DPI application. Snort which is a robust NIDS supports various functionalities in addition to the DPI capabilities which were extracted to the DPI Service. It is also important to recall that the DPI Service currently only reports exact-pattern matches. Once a match is reported, it still needs to go through an advanced examination process in Snort (see 2.2.5). Therefore, it makes perfect sense that the middleboxes which run Snort are a bottleneck in the system.

4.4 Analysis of Reporting Results with NSH

As part the project we have extended the DPI Service to allow the reporting of pattern match results within the inspected packet using the NSH protocol (see 3.1). Reporting the results using NSH has the overhead of adding information to the packet (see 3). In this section we analyze this overhead.

Whenever pattern matches are found within a packet, the NSH is constructed with the match report results lined in a sequence as the metadata. Following the NSH encapsulation, the packet is encapsulated with VXLAN-gpe and then re-encapsulated with an outer UDP header. The result is once again encapsulated with an outer IP header and an outer Ethernet header as the VXLAN protocol defines (see 3.1 and 3). The combination of all these layers, including the match report results, forms the NSH overhead analyzed in the section.

Figure 15a shows the cumulative distribution (CDF) of the NSH encapsulation overhead in the network trace used for the experiments (see 4.1). Note that 27% of the packets in the trace have pattern matches, but only these packets are represented in the figure. The percentage of packets in the trace that had pattern matches, is significantly higher then the original experiment which measured around 10% (see [4, Section 6.5]). The difference can be explains, due to the fact that in the original experiment (see [4, Section 6.2]) the DPI Service experiments were performed using exact-match patterns of length 8 characters or more, while we performed the experiments using patterns of length 4 or more. Obviously



(a) CDF of NSH encapsulation overhead per packet (b) CDF of non-empty results packet size vs original

Figure 15: NSH encapsulation overhead Analysis

using smaller patterns yields more matches.

Analyzing the graph, reveals that the average NSH size is 189 bytes, while the median size is 156 byte.

Figure 15b shows the cumulative distribution (CDF) of original packet sizes of packets with match reports, in the network trace used for the experiments (see 4.1) alongside with the cumulative distribution (CDF) of packet sizes after adding the NSH header to the same packets. Comparing the two graphs of the figure clearly shows that the NSH encapsulation increases the packet sizes significantly.

5 Conclusions

In order to demonstrate that the DPI as a Service framework [4] can operate in a more realistic environment setup, we have integrated the original framework implementation with the commonly used Snort NIDS. In addition, we have extended the DPI Service to support the NSH protocol which allows passing of the pattern match results with the inspected packet.

These two main enhancements to the system, allowed us to re-evaluate the framework by preforming a subset of the original experiments. Our experimental results (see 4) show, that the original promising results of the framework can in fact be reproduced in a more realistic environment setup. Although our results demonstrate less improvement then originally reported, they are still significant and solid.

References

[1] Snort users manual. https://www.snort.org/#documents/.

[2] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. of the ACM*, 18(6):333–340, 1975.

[3] Alexa: The web information company, 2013. http://www.alexa.com/topsites.

[4] Anat Bremler-Barr, Yotam Harchol, David Hay, and Yaron Koral. Deep packet inspection as a service. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 271–282, New York, NY, USA, 2014. ACM.

[5] Mallik Mahalingam, T. Sridhar, Mike Bursell, Lawrence Kreeger, Chris Wright, Kenneth Duda, Puneet Agarwal, and Dinesh Dutt. Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. RFC 7348, August 2014.

[6] Thomas Nadeau and Paul Quinn. Problem Statement for Service Function Chaining. RFC 7498, April 2015.

[7] Node.js. https://nodejs.org.

[8] Marc Norton. Optimizing pattern matching for intrusion detection. *Sourcefire, Inc., Columbia, MD*, 2004.

[9] Opendaylight. https://www.opendaylight.org/.

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

[11] Paul Quinn, Puneet Agarwal, Rajeev Manur, Rex Fernando, Jim Guichard, Surendra Kumar, Abhishek Chauhan, Michael Smith, Navindra Yadav, and Brad McConnell. Network service header. IETF Internet-Draft, February 2014. https://datatracker.ietf.org/doc/draft-quinn-sfc-nsh.

[12] Paul Quinn and Jim Guichard. Service function chaining: Creating a service plane via network service headers. *Computer*, 47(11):38–44, 2014.

[13] Martin Schütte, Thomas Scheffler, and Bettina Schnor. Development of a snort ipv6 plugin - detection of attacks on the neighbor discovery protocol. In *SECRYPT*, 2012.

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

A NSH Protocol Format

The appendix below contains the sections from the NSH Internet draft [11] which are most relevant to our work.

An NSH is composed of a 4-byte base header, a 4-byte service path header and context headers, as shown in Figure 16 below.

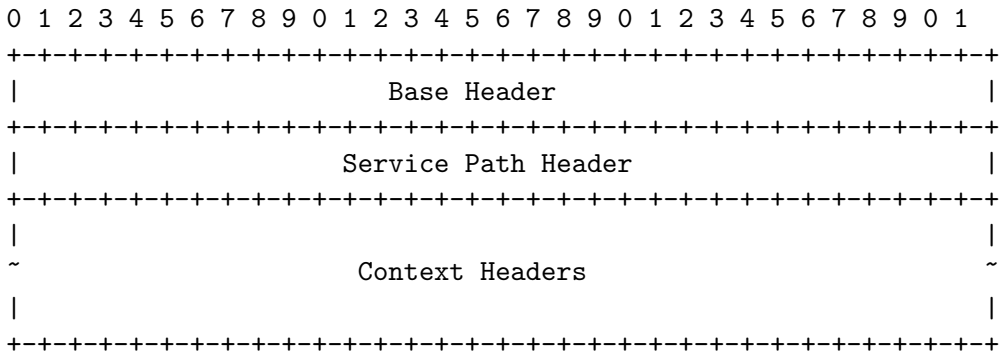


Figure 16: Network Service Header

Base header: provides information about the service header and the payload protocol.

Service Path Header: provide path identification and location within a path.

Context headers: carry opaque metadata and variable length encoded information.

NSH is imposed between the original packet or frame, and an outer network transport encapsulation such as MPLS, VXLAN/VXLAN-GPE, GRE or IP in IP. NSH is transport agnostic, and can be carried by many widely deployed transport protocols.

A.1 NSH Base Header

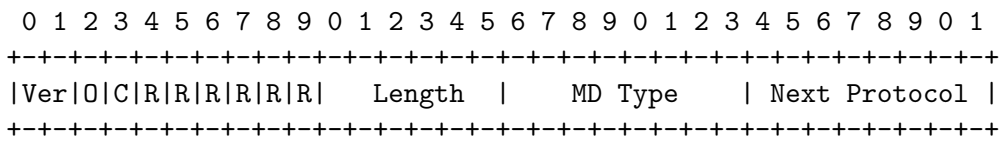


Figure 17: NSH Base Header

Version: The version field is used to ensure backward compatibility going forward with future NSH updates.

O bit: Indicates that this packet is an operations and management (OAM) packet.

C bit: Indicates that a critical metadata TLV is present. This bit acts as an indication for hardware implementers to decide how to handle the presence of a critical TLV without necessarily needing to parse all TLVs present. The C bit MUST be set to 1 if one or more critical TLVs are present.

All other flag fields are reserved.

Length: total length, in 4-byte words, of the NSH header, including optional variable TLVs.

MD Type: indicates the format of NSH beyond the base header and the type of metadata being carried. This typing is used to describe the use for the metadata. NSH defines two MD types: 0x1 which indicates that the format of the header includes fixed length context headers. 0x2 which does not mandate any headers beyond the base header and service path header, and may contain optional variable length context information.

Next Protocol: indicates the protocol type of the original packet. This draft defines the following Next Protocol values: 0x1 : IPv4 0x2 : IPv6 0x3 : Ethernet

A.2 Service Path Header

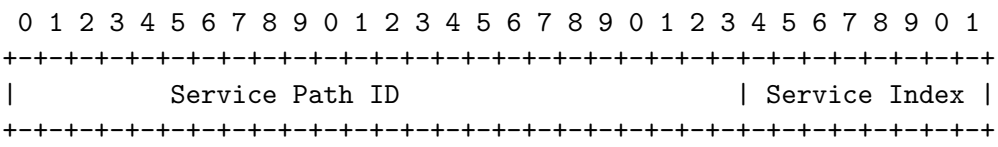


Figure 18: NSH Service Path Header

Service path ID (SPI): 24 bits : identifies a service path. Service index (SI): 8 bits : provides location within the service path.

Service Path Identifier (SPI): identifies a service path. Participating nodes MUST use this identifier for path selection. An administrator can use the service path value for reporting

and troubleshooting packets along a specific path.

Service Index (SI): provides location within the service path. Service index MUST be decremented by service functions or proxy nodes after performing required services. MAY be used in conjunction with service path for path selection. Service Index is also valuable when troubleshooting/reporting service paths. In addition to location within a path, SI can be used for loop detection.

A.3 NSH MD-type 1

When the base header specifies MD Type 1, NSH defines four 4-byte mandatory context headers, as per Figure 19. These headers must be present and the format is opaque as depicted in Figure 20.

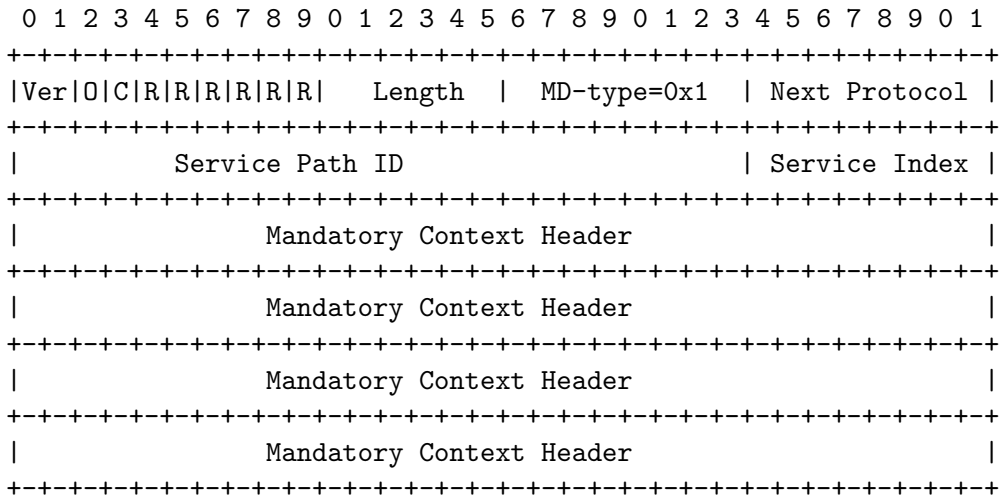


Figure 19: NSH MD-type=0x1

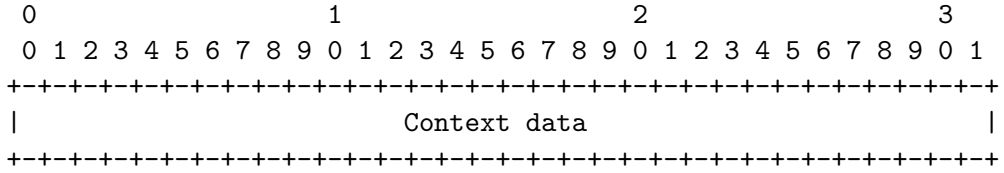


Figure 20: Context Header

A.4 NSH MD-type 2

When the base header specifies MD Type 2, NSH defines variable length only context headers. There may be zero or more of these headers as per the length field.

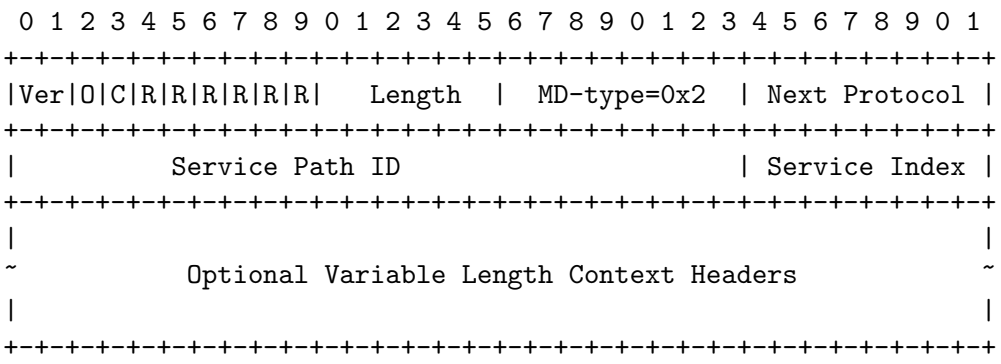


Figure 21: NSH MD-type=0x2

A.4.1 Optional Variable Length Metadata

NSH MD Type 2 MAY contain optional variable length context headers. The format of these headers is as described below.

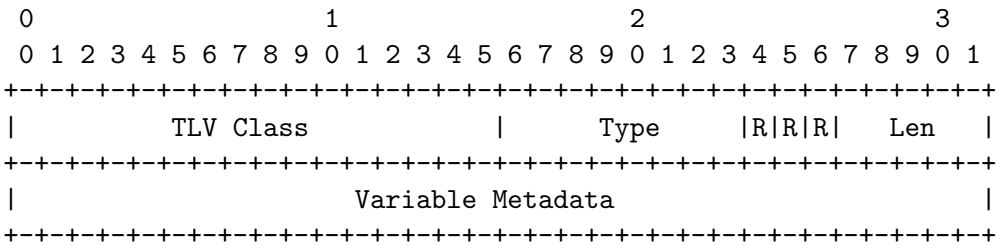


Figure 22: Variable Context Headers

TLV Class: describes the scope of the "Type" field. In some cases, the TLV Class will identify a specific vendor, in others, the TLV Class will identify specific standards body allocated types.

Type: the specific type of information being carried, within the scope of a given TLV Class. Value allocation is the responsibility of the TLV Class owner. The most significant bit of the

Type field indicates whether the TLV is mandatory for the receiver to understand/process. This effectively allocates Type values 0 to 127 for non-critical options and Type values 128 to 255 for critical options. Figure 23 below illustrates the placement of the Critical bit within the Type field.

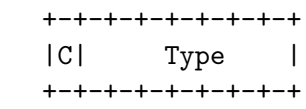


Figure 23: Critical Bit Placement Within the TLV Type Field

Encoding the criticality of the TLV within the Type field is consistent with IPv6 option types.

If a receiver receives an encapsulated packet containing a TLV with the Critical bit set in the Type field and it does not understand how to process the Type, it MUST drop the packet. Transit devices MUST NOT drop packets based on the setting of this bit.

Reserved bits: three reserved bit are present for future use. The reserved bits MUST be zero.

Length: Length of the variable metadata, in 4-byte words.

תקציר

Deep Packet Inspection (DPI) היא פונקציונליות נפוצה בקרב ישומי רכיבי ביניים (middlebox) ברשת. במקרים רבים, ה- DPI מבוצע על ידי מערכות איתור מצבים חריגים (Intrusion Detection Systems), כגון סנורט (Snort). באופן מסורתי, כל פקטה נסרקת מחדש על ידי מספר רכיבי ביניים בדרך ליעדה הסופי. מחקרים עדכניים מראים, כי DPI היא מהפעולות היקרות ביותר מבחינת משאביי העיבוד ברכיבי הביניים.

מאמר עדכני, מציג מערכת (framework) המאפשרת להוציא את פונקציונליות ה- DPI מתוך רכיבי הביניים, ולהנגיש אותה כשירות (DPI as a Service) לרכיבים. בנוסף לאיפיון המערכת, כותבי המאמר מספקים מימוש לדוגמא ומציגים תוצאות מבטיחות אשר נמדדו במסגרת סדרת ניסויים.

בעבודה זו הרחבנו ושיפרנו את המימוש לדוגמא, כדי להוכיח שהמערכת שהוצעה במאמר יכולה לפעול בסביבה מציאותית יותר. בראש ובראשונה, שילבנו את סנורט במערכת המקורית. בנוסף לכך, הרחבנו את ה- DPI Service והוספנו תמיכה בפרוטוקול ה- Network Service Header (NSH), המאפשר דיווח על מציאת דפוסים חשודים בפקטה יחד עם הפקטה המקורית. תוספות משמעותיות אלו, הפכו את המימוש לדוגמא למערכת איתנה, אשר יכולה להגיב בעת זיהוי דפוס זדוני בתעבורת הרשת.

לבסוף, כאשר הושלמה העבודה על הרחבת המערכת, ערכנו את הניסויים הבסיסיים אשר דווחו במאמר המקורי. הממצאים שלנו מראים, כי המערכת המשופרת מצליחה לשחזר את תוצאות המערכת המקורית.

המרכז הבינתחומי הרצליה
בית הספר למדעי המחשב על שם אפי ארזי

רכיב רשת וירטואלי Snort עם שירות סריקת עומק של חבילות

פרויקט גמר המוגש במילוי חלק מהדרישות לקראת תואר מוסמך במדעי המחשב

על ידי

אשר גרובר

עבודה זו בוצעה בהנחיית

פרופ .ענת ברמלר-בר

מר .יותם הר-חול

דצמבר 2016