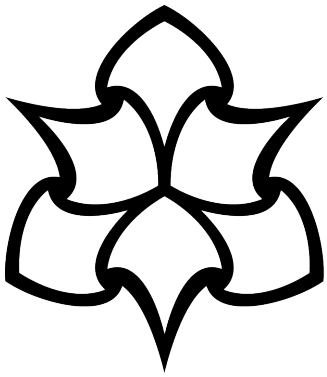

2016 4th International Conference on Future Internet of Things and Cloud Workshops

A Survey on Network Security Monitoring Systems

Ibrahim Ghafir
FI, Masaryk University
School of Computing
Manchester Metropolitan University
ghafir@mail.muni.cz

Vaclav Prenosil
Faculty of Informatics
Masaryk University
Brno, Czech Republic
prenosil@fi.muni.cz

Jakub Svoboda
Faculty of Informatics
Masaryk University
Brno, Czech Republic
svob.jak@gmail.com

Mohammad Hammoudeh
School of Computing
Manchester Metropolitan University
Manchester, UK
M.Hammoudeh@mmu.ac.uk

***Abstract*—Network monitoring is a difficult and demanding task that is a vital part of a network administrator's job. Network administrators are constantly striving to maintain smooth operation of their networks. If a network were to be down even for a small period of time, productivity within a company would decline; and in the case of public service departments the ability to provide essential services would be compromised. There are several approaches to network security monitoring. This paper provides the readers with a critical review of the prominent implementations of the current network monitoring approaches.**

***Keywords*—Network security monitoring, packet capture, deep packet inspection, flow observation.**

I. Introduction

A network monitoring system monitors an internal network to identify slow or failing system components. It can find, report and resolve problems. Whether it is a small business or a large enterprise, continuous monitoring network helps to maintain high performance networks with little downtime. Monitoring reports cater to different levels of audiences, i.e., the network and systems administrators, as well as to management. Therefor, a monitoring system should not be too complex to understand and use, nor should it lack basic reporting and drill down functionalities.

An effective network monitoring system covers every aspect of a networked system, including response time, availability, uptime and security. This makes network monitoring a difficult and demanding task. Network administrators are constantly striving to maintain smooth operation of their networks. If a network were to be down even for a small period of time, productivity within a company will decline; and in the case of public service departments the ability to provide essential services would be compromised. To offer proactive services, administrators need to optimize data flow and access in a complex and changing environment. Thus, these systems can help to identify specific activities and performance metrics, generating results that enable a business to address a variety of needs, including meeting compliance requirements, eliminating internal security threats and providing more operational visibility [1].

Network monitoring for security management aims to protect sensitive information on devices connected to a data network by controlling access points to that information. Securing sensitive information from both internal and external sources protects the network functionality from different forms of malicious attacks. There are many network monitoring approaches to ensure network security [2]. This paper provides the readers with a critical review of such approaches.

The remainder of this paper is organized as follows. Section 2 classifies the current network security monitoring implementations into three main classes. A comparison between presented implementations is provided in Section 3. Section 4 concludes the paper.

II. Network Security Monitoring Implementations

This section classifies the current network security monitoring implementations into packet capture representatives, deep packet inspection representatives and flow-based observation representatives. It also provides information about the suitability of particular tools for development of new network traffic analysis methods.

Packet capture is to intercept a data packet that is crossing or moving over a specific computer network. Once a packet is captured, it is stored temporarily so that it can be analyzed [3]. Deep packet inspection (DPI) is an advanced method of packet filtering that functions at the application layer of the OSI (Open Systems Interconnection) reference model. The use of DPI makes it possible to find, identify, classify, reroute or block packets with specific data or code payloads that conventional packet filtering, which examines only packet headers, cannot detect [4]. Traffic flow is a sequence of packets sent from a particular source to a particular unicast, anycast, or multicast destination that the source desires to label as a flow. A flow could consist of all packets in a specific transport connection or a media stream [5].

A. Packet Capture Representatives

1) Tcpdump:

Tcpdump is a command line tool for packet capture analysis. Tcpdump can analyze both live traffic using the libpcap library and captured packet traces in PCAP format. Packets may be filtered both before and after the capture. Filtering before the capture can be done using BPF (Berkeley Packet Filter). Filtering after capture can be achieved using tcpdump's filters [6].

Data is printed out in text format. The output displays individual packets with information that include source and destination addresses, L4 protocol used, and L4 protocol flags. Figure 1 shows output listing two packets.

Packets to be displayed can be filtered using expressions. Filters can be imposed on source and destination addresses, ports, L3 and L4 protocols, and L4 protocol flags. Addresses

978-1-5090-3946-3/16 $31.00 © 2016 IEEE
DOI 10.1109/W-FiCloud.2016.30

```
$ tcpdump -r pcap -n \
"src host 10.0.2.15 and dst port 22 and tcp[13] = 2"
reading from file pcapfile, link-type EN10MB (Ethernet)
20:20:40.512613 IP 10.0.2.15.54346 > 192.168.1.42.22: Flags [S],
seq 3123841387, win 29200, options [mss 1460,sackOK,TS val 509640
ecr 0,nop,wscale 7], length 0
20:20:41.356843 IP 10.0.2.15.45749 > 192.168.1.41.22: Flags [S],
seq 1764864395, win 29200, options [mss 1460,sackOK,TS val 509851
ecr 0,nop,wscale 7], length 0
```

Fig. 1. Example of a tcpdump filter and tcpdump's output.

can be expressed in format of individual addresses or in CIDR notation. Multiple rules in the expression can be composed using boolean operators. The example on Figure 1 uses three filters. *src host 10.0.2.15* selects only packets originating in the IP address 10.0.2.15. *dst port 22* selects only packets destined for the port 22. Finally, *tcp[13] = 2* selects packets in which the decimal value of the 14th byte is 2. The filters are composed using the *and* word, which means only packets that meet all the criteria pass through the filter.

Tcpdump needs root privileges to open the network interface. Operation without full root is possible using SUID or Linux capabilities. Granting the *tcpdump* executable *cap_net_raw* and *cap_net_admin* capabilities allows tcpdump to be run as a regular user.

2) Wireshark:

Wireshark is a graphical tool for packet capture analysis. While Wireshark and tcpdump are implementations of the same architectural approach, their underlying ideas differ. Tcpdump is as close to the raw data as possible, while Wireshark strives to provide higher-level representation of the same data [7].

Wireshark can analyze both live traffic using the libpcap library and captured packet traces in PCAP format. Captured packets can be filtered both during and after the capture. Filtering after capture can be achieved using filter expressions. Filtering during capture can be done using BPF.

Data are displayed as text arranged in a scrollable colored list and expandable boxes. Wireshark's main window has two frames. The upper frame displays list of captured packets with their basic attributes displayed. A line may be colored, based on the protocol the individual packet belongs to. When the user selects packet in the upper frame, this particular packet is displayed in the lower frame. The lower frame's representation includes several boxes that can be expanded and collapsed. The boxes contain various attributes of the packet as well as representations of the packet's data in ISO OSI layers' data the packet is part of. Also, related data from other packets may be displayed there, HTTP TCP stream for instance.

Packets displayed in the upper frame can be filtered using expressions entered in the text box on the top of the window. The expression vocabulary is richer that the one of tcpdump. Figure 2 shows the architecture of Wireshark.

Wireshark uses a separate program to capture the traffic, dumpcap. The reason is to allow separation of privileges [8]. Wireshark can be run as a regular user and only dumpcap has to be given special permissions. Dumpcap needs root privileges to open the network interface. Operation without the user

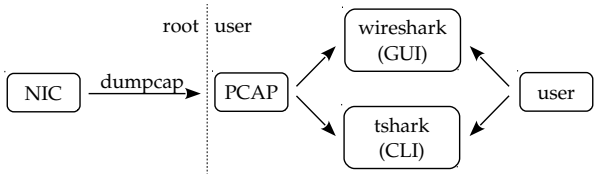

Fig. 2. Wireshark architecture showing privilege separation.

having root access is possible using either SUID or Linux capabilities. Granting the *dumpcap* executable *cap_net_raw* and *cap_net_admin* capabilities allows dumpcap to be run as a regular user without SUID.

B. Deep Packet Inspection (DPI) Representatives

1) Snort:

Snort is an intrusion detection system performing deep packet inspection using pattern matching. The pattern matching is implemented in the form of rules [9]. Rules are structured text files describing network traffic data of interest. Typically, rules are used to generate alerts when a security-related incident occurs, such as malware activity, attack, or breach of security policies. A rule contains information specifying when the rule should be triggered. An important part of these information is one or more patterns that are searched in the network traffic. The pattern can be a sequence of characters or bytes or a regular expression. Figure 3 shows Snort rule structure.

```
rule header    alert tcp any any -> 192.168.1.0/24 111
rule options   (content:"|00 01 86 a5|"; msg:"mountd access";)
```

Fig. 3. Snort rule structure

Snort rules have a specific structure [10]. The beginning of the rule before the parentheses describes which network flows the rule refers to. This is called the rule header. The rule header specifies the action the rule should perform (*alert* for instance), L3 protocol and source/destination IP addresses and ports on which to match. Variables may be used in place of IP addresses. The rest of the rule inside the parentheses is called the rule options. Rule options specify content on which the rule matches and other properties of the rule, its name, classification type, etc. The most important part of the rule is the *content* keyword, which specifies a pattern to be found in the packet's payload. The *content* keyword's function can be further changed using modifiers. For instance, modifiers offset, distance, depth, and *within* control in which areas of the packet the rules are matched [11].

Snort has three special keywords, *byte_test*, *byte_jump*, and *byte_extract*, that allow to adjust the pattern matching based on data in an individual packet [12]. The first two keywords behave as patterns that match when their conditions are true. *byte_test* performs arithmetic ($<$, $\leq$, $=$, $>$, $\geq$) and bitwise (AND, OR) comparisons on sequences of bytes. *byte_jump* puts a space before the next pattern with size inferred from payload's byte value. If the jump is possible, *byte_jump* also behaves as a match. This behavior can be used to match packets with a specific length based on specific data in the payload. *byte_extract* converts specified bytes into a numerical

variable that can be used later in the rule. These keywords do not allow more complicated decoding or processing.

The Snort's architecture allows the implementation of so-called preprocessors [13]. Preprocessors read the packet before rule evaluation, serially in the order specified by Snort's configuration. This allows implementation of additional rule keywords. Moreover, preprocessors allow implementation of functionality more complicated than just pattern matching, such as data decoding and anomaly detection. For instance, the Normalizer preprocessor converts equivalent values to a unified format with the goal of making IDS evasion harder. Snort's architecture including the preprocessors is depicted in Figure 4.

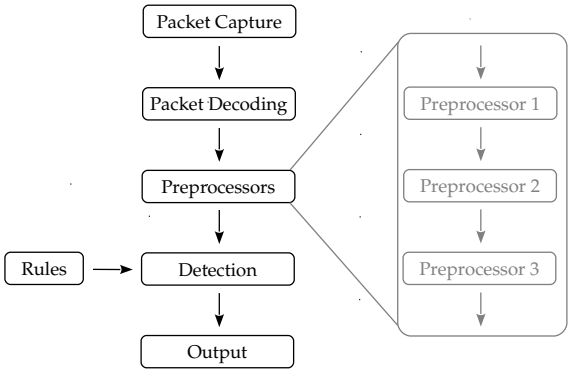

Fig. 4. Snort architecture.

There are several preprocessors for anomaly detection available. Frag3 and stream5 preprocessors are integrated in the official Snort distribution and detect protocol anomalies. SPADE [14], PHAD [15], and snortad [16] are 3rd party preprocessors detecting traffic anomalies. Snort preprocessors are usually implemented in C. They allow implementation of similar concepts to those that can be implemented in the event-based architecture.

Snort needs root privileges to open the network interface. It is possible to configure Snort to drop its privileges to a non-root user once it opens the network interface.

Snort is a single-threaded application. Multithreaded Snort setups work in the following way: The monitored traffic is divided by flows into multiple parts; each part of the traffic is fed to a single Snort instance

2) Suricata:

Suricata is an intrusion detection system performing deep packet inspection using pattern matching [17]. Figure 5 shows Suricata rule structure, the rule header and rule options are as same as the Snort rule structure.

```
rule header    alert tcp any any -> 192.168.1.0/24 111
rule options   (content:"|00 01 86 a5|"; msg:"mountd access";)
```

Fig. 5. Suricata rule structure.

Suricata uses similar rules to Snort and is compatible with Snort rules. The rule structure is the same for both Snort and Suricata. The difference between the two is in the keywords and protocols that can be specified. Suricata allows specification of several L7 protocols on top of the L3 protocols supported by Snort, *http*, *ftp*, *tls*, *smb* and *dns* [18]. Some keywords behave differently than in Snort, for instance, the *fast_pattern*:only keyword does not make a difference in processing, unlike in Snort. Some keywords are supported only by Suricata, such as the *iprep* keyword for matching IP reputation data and the *dns_query* keyword for analyzing only the DNS response body.

Suricata's architecture is similar to Snort's one with a difference. What corresponds to the preprocessor part in the Snort's architecture is divided in two in Suricata, decoding and detection. Decoding modules add information to the internal representation of packets in Suricata. Detection modules rely on this internal representation and provide keywords for use in rules. Overview of the Suricata's architecture is shown in Figure 6.

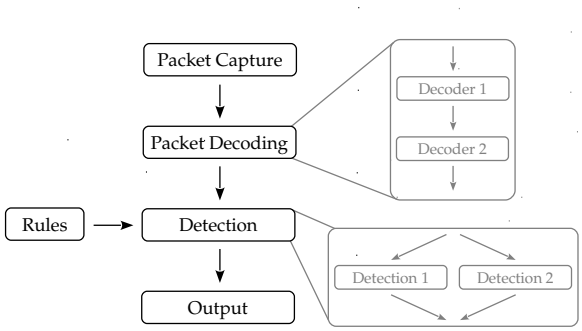

Fig. 6. Suricata architecture.

Each packet is first processed in decoding functions and then in detection modules. Decoding functions read the packet and save the decoded data into an internal representation of the packet. The decoding functions are called one at a time on the packet. Extending the decoding functionality is possible by implementing a new decoding function and placing it into the decoding pipeline. The decoding pipeline starts with the source of captured packets, then L2 is decoded, and then protocols on higher layers are decoded.

Upon decoding, the packets pass detection. The detection is governed by rules and depends on the decoding step. The rules are matched with the internal packet representation. The matching process is broken into several detection modules in all of which the matching takes place. Unlike decoding, detection is parallelized and one packet can be processed in multiple detection modules at the same time. Extending the detection functionality is possible by implementing a new detection module and registering it in the table of detection methods.

Suricata is written in C and the modules for Suricata have to be written in C. There are no plans supporting C++. C requires greater programming expertise than the Bro language. Therefore, this property makes Suricata not the best prototyping tool available.

Suricata is multithreaded out of the box. Even though it is not as fast as Snort on a single-CPU computer, Suricata is

designed to scale on computers with tens of CPUs [19]. The multithreading approach is different from Snort. Multithreaded Snort setups divide the monitored traffic by flows into multiple parts, each processed by an individual Snort instance. Suricata, on the other hand, does not require the traffic balancing since it manages multithreading itself. This approach makes it more user-friendly.

3) Bro:

Bro [20] is a network security monitor performing deep packet inspection using event-based analysis. In contrast to Snort and Suricata, Bro is not rule-driven. Instead, it implements a Turing-complete scripting environment [21]. Rule-based detection and arbitrary detection algorithms can be implemented in this environment. Bro detection rules are described by scripts. Figure 7 shows Bro architecture.

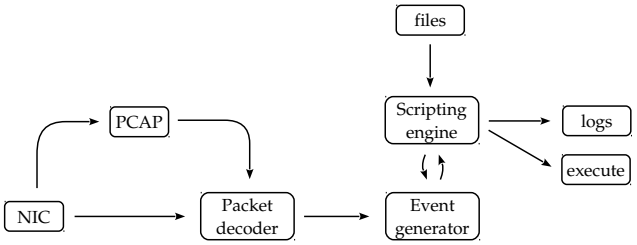

Fig. 7. Bro architecture.

The Bro's programming environment uses the Bro language, which is an interpreted, typed language. What makes the Bro language special is the domain-specific types. For example, the *addr* type holds an IP address [22]. Variables of structured types are reference type variables. This makes processing of large sets or tables efficient, since only the references are copied, not the data itself. There are two types of collections, sets and tables. Loops are available in the form of iteration through collections. The Bro programming language lacks other forms of loop control, presumably serving as a deterrent against overly complex algorithms. This is a reasonable requirement for network traffic monitoring when the processing is done in real time. And that exactly is the most significant goal of Bro, to allow real-time network traffic analysis and save already processed results to log files.

The default installation of Bro contains many scripts implementing various sorts of traffic analysis. Some of the items the default Bro setup monitors are: Bidirectional flows, DHCP leases, DNS queries and responses, MD5 and SHA1 hashes of files transmitted over unencrypted protocols, HTTP requests and user agents, port scans, email headers from SMTP traffic, successful and unsuccessful SSH connections, SSL certificates, SYSLOG messages, traffic tunnels. Since the preinstalled scripts usually expose an API in the form of events, they can be used by user scripts, extending the default functionality.

The core of Bro, implemented in C, processes network traffic, performs DPI and generates events about what is happening in the traffic. Events generated by the core are listed in the bif files [23]. Many events are generated, spanning L2 through L7. Examples are a new ARP packet, closed TCP connection, HTTP request, etc. In other words, this type of DPI performs semantic matching of network events instead of simple pattern matching, as opposed to Snort and Suricata. Majority of the events provide context, typically in the form of information about the relevant connection. The events are then processed by the Bro scripts.

Bro scripts use so-called event handlers to listen to the events. The usual reactions to events vary. On the one hand, the simplest possible processing saves the event information to a log file. On the other hand, some scripts implement fairly complex processing and generate additional types of events. This further extends DPI abilities of Bro. Scripts can handle events generated both by the core and by other scripts. Figure 8 shows a very short module that just writes "Hello world!" to the standard output when Bro starts.

```
module helloworld;
event bro_init() {
 print "Hello world!";
}
```

Fig. 8. A simple Hello world! script.

The scripting engine hosts the scripts and dispatches events generated both by the scripts and the core to the scripts listening to these events. It also allows operations like file access and execution of applications native to the operating system. This functionality can be used by advanced scripts. File access may be used to fetch information from external sources, e.g., a blacklist. Execution facility may be used for many purposes. One example is reporting issues to a ticket managing software via email. The *sendmail* executable can be used by such a script. Another example is automatic triggering of a remotely triggered black hole by executing a program that does the blackholing.

Bro scripts are organized in so-called modules. A module can be implemented wholly in one file or can be broken into several files. Two identifiers with the same name in two different modules do not collide with each other. Cross-module references can be made using the name *name_of_module::name_of_identifier*. A module can define types, variables, functions, and event handlers. These entities can be either local to the module or globally accessible from other modules.

It is possible to define custom types using *enum*, *set*, *table*, *vector*, and *record*. *Enum* in Bro is similar to *enum* in other languages. *Set* is similar to *HashSet¡T¿* in C# in its functionality [24], albeit the syntax is different. *Table* is similar to *Dictionary¡TKey,TValue¿* in C# [25] with the difference that C# allows only one key while Bro allows multiple keys. *Vector* is a table indexed by count. *Count* is the name for int in Bro. *Record* is similar to C# class [26] that contains only fields [27]. Both Bro *record* and C# *class* are reference types, meaning assignment of its instance copies only the reference (pointer), not the whole instance. This can be compared to C# *struct* which is a value type, meaning assignment of its instance copies the whole instance.

Bro can be run both as a single-threaded application and as a multithreaded distributed application. The single-threaded mode is called standalone while the multithreaded one is called cluster. If Bro is used as a platform for development of proof-of-concept methods, the standalone mode is usually

more appropriate than the cluster mode. Development for the cluster mode is more difficult than for the standalone mode because additional functionality has to be used by scripts [28].

C. Flow-based Observation Representatives

Flow-based observation architecture contains two main components; a flow exporter and a flow collector. This section covers representative implementations of both flow exporters and flow collectors.

1) Flow Exporters:

nProbe [29] is a commercial open-source flow exporter. Data can be exported in NetFlow v5, NetFlow v9, and IPFIX formats. nProbe has an application visibility (nDPI) ability, which is used for detection of application-specific protocols. This information is saved in a custom column in NetFlow v9 or IPFIX format. It is difficult to obtain nProbe source code for free.

YAF [30] is an open-source flow exporter. Data are exported in the IPFIX format. A passive OS fingerprinting functionality based on the p0f software can be compiled into YAF. YAF supports modules that implement DPI. However, YAF does not provide DPI in default setup.

QoF [31] is a fork of YAF. It removes all payload inspection abilities and instead focuses on passive performance measurements.

ipt-netflow [32] is a plugin for iptables for flow export. Data can be exported in NetFlow v5, NetFlow v9, and IPFIX formats. There is no special functionality besides standard network flows. There is also no apparent focus towards high-throughput networks. ipt-netflow is open-source.

pmacct [33] is an open-source flow exporter and flow collector. Data can be exported in NetFlow v5, NetFlow v9, sFlow v5, and IPFIX formats. Supports high-throughput networks using PF_RING. No DPI-related functionality is available in pmacct.

softflowd [34] is an open-source flow exporter performing export to NetFlow v1, v5, and v9 formats. There is no apparent effort to provide anything on top of regular NetFlow data export.

2) Flow Collectors:

nProbe is not only a flow exporter, it is also a flow collector. Available storage backends are MySQL, SQLite, text files, and binary files. The nProbe flow collector was created because its author deemed other collectors available at the time to be too cumbersome to use.

IPFIXcol [35] is an IPFIX collector designed for high-throughput networks. IPFIXcol claims to be flexible. Storage backend can be customized using output plugins. IPFIXcol also allows implementation of so-called IPFIX mediators, used for processing of the collected data before it hits the collector.

flowd [36] is a NetFlow v1, v5, v7, and v9 collector. It is created under the UNIX philosophy to do just one thing. The collected data are saved in a binary format. flowd is provided with Perl and Python interfaces for reading the binary data. flowd strives for security using privilege separation of components. flowd is open-source and freely available.

nfdump [37] consists of several tools. The nfcapd tool listens to NetFlow v5, v7, v9 streams and saves them to nfcap files. The nfdump tool can be used for analysis of nfcap files. nfdump uses similar filter syntax to tcpdump. nfdump is open-source and freely available.

pmacct [38] as a collector has several storage backends available. It can use MySQL, PostgreSQL, SQLite, MongoDB, BerkeleyDB, and flat files. Among other formats, it can collect NetFlow v1-v9 and IPFIX.

SiLK [39] is a collector for NetFlow v5, v9, and IPFIX data. It is designed for high-throughput networks. SiLK consists of multiple tools and plugins for filtering, analysis, and processing of flow data.

III. Comparison

With respect to the selection of network traffic monitor suitable for DPI, the following criteria have been evaluated for each mentioned traffic monitor:

- *Prototyping*: Is the network traffic monitor suitable for creation of method prototypes?
- *Developer-friendliness*: Does the network traffic monitor allow development of new traffic analysis methods in an easy to use way?
- *Extensibility*: Is it possible to extend the existing functionality of the network traffic monitor in a reasonable way? What programming language does the API use?

The descriptions of individual network traffic monitors in this paper indicate answers to these criteria. Table 1 shows the summary.

TABLE I. Bro is the Suitable Tool for Creation of Prototypes

Monitor	Prototyping	Developer friendliness	Extensibility
Tcpdump	No	No	No API
Wireshark	No	No	No API
Snort	No	No	C language
Suricata	No	No	C language
Bro	Yes	Yes	Bro language

IV. Conclusion

There are several approaches to network security monitoring. There is no best approach, each approach performs best in a certain environment and fit different purposes. Wireshark is an effective tool for manual analysis, predominantly of small capture files. Tcpdump is packet-oriented approach that works well in scenarios where filtering individual packets by L3/L4 attributes, like IP address, TCP flags and payload bytes, is sufficient. It does not work well for stream reassembly or L7 protocol analysis. Snort and Suricata work well when the objective is to match patterns in network data. Bro allows development of advanced detection methods. Bro offers the best software/environment for the development of novel detection or processing techniques. It can be used for continuous monitoring of high-throughput networks. The scripting environment is extensible in a memory-safe language specialized in

network data processing. It is not constrained by belonging to a single paradigm for network monitoring like the other tools. Unfamiliarity is a disadvantage, compared to more known tools like wireshark, tshark, snort and suricata.

REFERENCES

[1] O. B. Kodical, S. Srinivasan, and N. Srinath, "Tool tracker: A toolkit ensembling useful online networking tools for efficient management and operation of a network," *World Academy of Science, Engineering and Technology, International Journal of Computer, Electrical, Automation, Control and Information Engineering*, vol. 2, no. 6, pp. 2013–2018, 2008.

[2] J. Svoboda, I. Ghafir, and V. Prenosil, "Network monitoring approaches: An overview," in *Proceedings of International Conference on Advances in Computing, Communication and Information Technology*, birmingham, UK, 2015. ISBN: 978-1-63248-061-3.

[3] V. Moreno, J. Ramos, P. M. Santiago del Rio, J. L. Garcia-Dorado, F. J. Gomez-Arribas, and J. Aracil, "Commodity packet capture engines: tutorial, cookbook and applicability," *Communications Surveys & Tutorials, IEEE*, vol. 17, no. 3, pp. 1364–1390, 2015.

[4] A. Bremler-Barr, Y. Harchol, D. Hay, and Y. Koral, "Deep packet inspection as a service," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. ACM, 2014, pp. 271–282.

[5] R. Hofstede, P. Celeda, B. Trammell, I. Drago, R. Sadre, A. Sperotto, and A. Pras, "Flow monitoring explained: From packet capture to data analysis with netflow and ipfix," *Communications Surveys & Tutorials, IEEE*, vol. 16, no. 4, pp. 2037–2064, 2014.

[6] F. Fuentes and D. C. Kar, "Ethereal vs. tcpdump: a comparative study on packet sniffing tools for educational purpose," *Journal of Computing Sciences in Colleges*, vol. 20, no. 4, pp. 169–176, 2005.

[7] V. Ndatinya, Z. Xiao, V. R. Manepalli, K. Meng, and Y. Xiao, "Network forensics analysis using wireshark," *International Journal of Security and Networks*, vol. 10, no. 2, pp. 91–106, 2015.

[8] J. Keuter, "Privilege separation," http://wiki.wireshark.org/Development/PrivilegeSeparation, accessed: 12-01-2016.

[9] H. Li, G. Liu, W. Jiang, and Y. Dai, "Designing snort rules to detect abnormal dnp3 network data," in *Control, Automation and Information Sciences (ICCAIS), 2015 International Conference on*. IEEE, 2015, pp. 343–348.

[10] "Snort syntax and simple rulewriting," http://www.anotherchancecomputers.com/uncategorized/snort-syntax-and-simple-rule-writing/, accessed: 12-01-2016.

[11] J. Esler, "Offset, depth, distance, and within," http://blog.joelesler.net/2010/03/offset-depthdistance-and-within.html, accessed: 12-01-2016.

[12] "Writing good rules," http://manual.snort.org/node36.html#testing_numerical_values, accessed: 12-01-2016.

[13] J. Esler, "Preprocessors," http://manual.snort.org/node59.html, accessed: 12-01-2016.

[14] S. Biles, "Detecting the unknown with snort and the statistical packet anomaly detection engine (spade)," http://webpages.cs.luc.edu/~pld/courses/447/sum08/class6/biles.spade.pdf, accessed: 12-01-2016.

[15] M. Mahoney, "Network anomaly intrusion detection research at florida tech," http://cs.fit.edu/~mmahoney/dist/, accessed: 12-01-2016.

[16] "Anomalydetection: Home - snort.ad," http://www.anomalydetection.info/?home,1, accessed: 12-01-2016.

[17] J. S. White, T. Fitzsimmons, and J. N. Matthews, "Quantitative analysis of intrusion detection systems: Snort and suricata," in *SPIE Defense, Security, and Sensing*. International Society for Optics and Photonics, 2013, pp. 875 704–875 704.

[18] "Suricata rules," https://redmine.openinfosecfoundation.org/projects/suricata/wiki/Suricata_Rules, accessed: 12-01-2016.

[19] V. Julien, "On suricata performance," http://blog.inliniac.net/2010/07/22/on-suricataperformance/, accessed: 12-01-2016.

[20] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Computer networks*, vol. 31, no. 23, pp. 2435–2463, 1999.

[21] "The bro network security monitor," http://www.bro.org/documentation/overview.html, accessed: 12-01-2016.

[22] "Types and attributes - bro 2.2 documentation," http://bro.org/sphinx/scripts/builtins.html, accessed: 12-01-2016.

[23] "All bro scripts," http://bro.icir.org/sphinx/scripts/scripts.html, accessed: 12-01-2016.

[24] "Microsoft: Hashset¡t¿ class," http://msdn.microsoft.com/en-us/library/bb359438.aspx, accessed: 12-01-2016.

[25] "Microsoft: Dictionary¡tkey, tvalue¿ class," http://msdn.microsoft.com/en-us/library/xfhwa508\%28v=vs.110\%29.aspx, accessed: 12-01-2016.

[26] "Microsoft: Classes and structs (c# programming guide)," http://msdn.microsoft.com/en-us/library/ms173109.aspx, accessed: 12-01-2016.

[27] "Microsoft: Classes and structs (c# programming guide)," http://msdn.microsoft.com/en-us/library/ms173118.aspx, accessed: 12-01-2016.

[28] "base/frameworks/cluster/main.bro," https://www.bro.org/sphinx/scripts/base/frameworks/cluster/main.html, accessed: 12-01-2016.

[29] L. Deri and N. SpA, "nprobe: an open source netflow probe for gigabit networks," in *TERENA Networking Conference*, 2003.

[30] B. T. Christopher Inacio, "Yaf: Yet another flowmeter," http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.368.3172&rep=rep1&type=pdf, accessed: 12-01-2016.

[31] B. Trammell, "Yaf-derived flow meter for passive performance measurement," https://github.com/britram/qof, accessed: 12-01-2016.

[32] "Netflow iptables module," http://sourceforge.net/projects/ipt-netflow/, accessed: 12-01-2016.

[33] P. Lucente, "pmacct: steps forward interface counters," http://www.pmacct.net/pmacct-stepsforward.pdf, accessed: 12-01-2016.

[34] "softflowd - a software netflow probe," https://code.google.com/p/softflowd/, accessed: 12-01-2016.

[35] P. Velan and R. Krejčí, "Flow information storage assessment using ipfixcol," in *Dependable Networks and Services*. Springer, 2012, pp. 155–158.

[36] "flowd - small, fast and secure netflow collector," http://code.google.com/p/flowd/, accessed: 12-01-2016.

[37] "Nfdump," http://nfdump.sourceforge.net/, accessed: 12-01-2016.

[38] P. Lucente, "pmacct project: Ip accounting iconoclasm," http://www.pmacct.net/, accessed: 12-01-2016.

[39] C. Gates, M. P. Collins, M. Duggan, A. Kompanek, and M. Thomas, "More netflow tools for performance and security." in *LISA*, vol. 4, 2004, pp. 121–132.