Performance Limitations of Parsing Libraries: State-of-the-Art and Future Perspectives

Antonino Manlio D'Agostino[1], Aleksandr Ometov[2][0000−0003−3412−1639], Alexander Pyattaev[3], Sergey Andreev[2][0000−0001−8223−3665], and Giuseppe Araniti[1][0000−0001−8670−9413]

[1] University Mediterranea of Reggio Calabria, Reggio Calabria, Italy
[2] Tampere University of Technology, Tampere, Finland
[3] Peoples' Friendship University of Russia (RUDN University), Moscow, Russia
Email: aleksandr.ometov@tut.fi

Abstract. The acceleration of mobile data traffic and the shortage of available spectral resources create new challenges for the next-generation (5G) networks. One of the potential solutions is network offloading that opens a possibility for unlicensed spectrum utilization. Heterogeneous networking between cellular and WLAN systems allows mobile users to adaptively utilize the licensed (LTE) and unlicensed (IEEE 802.11) radio technologies simultaneously. At the same time, softwarized frameworks can be employed not only inside the network controllers but also at the end nodes. To operate with the corresponding policies and interpret them efficiently, a signaling processor has to be developed and equipped with a fast packet parsing mechanism. In this scenario, the reaction time becomes a crucial factor, and this paper provides an overview of the existing parsing libraries (Scapy and dpkt) as well as proposes a flexible parsing tool that is capable of reducing the latency incurred by analyzing packets in a softwarized network.

Keywords: SDN · parsing · dpkt · Scapy · performance evaluation.

1 Introduction and Overview

Today, continuously growing numbers of interconnected devices push the telecommunication community towards developing new technologies for improved networking. Although several solutions have been proposed and implemented to address a steady increase in the mobile data consumption (e.g., the introduction of IPv6), they are still not ready for billions of new users/devices that are expected to join the network over a short period of time [1]. This projected acceleration suggests that the current and emerging (5G) mobile networks should evolve to become more "intelligent", efficient, secure, and, most importantly, scalable to enable future data communication that is incredibly diverse in nature [2, 3].

The Open Networking Foundation (ONF) [4] is a nonprofit consortium dedicated to the development, standardization, and commercialization of one of the 5G enablers – Software Defined Networks (SDNs). The ONF provided the most

explicit and well-received definition of SDN [5] as follows: "SDN is an emerging network architecture where network control is decoupled from forwarding and is directly programmable". Per this formulation, the SDN is shaped by main characteristics [6]: decoupling of control and data planes as well as programmability on the control plane. However, neither of these two SDN features is entirely new in the network architecture.

There is a pressing need to provide more capacity to the end users [7]. For this reason, LTE and WLAN integration may be attempted to improve the efficiency of mobile data offloading, which is a promising and low-cost solution to reduce the load on the cellular networks [8]. Further, the aggregate capacity of a heterogeneous network can be increased by utilizing short-range radio technologies [9] residing e.g., in unlicensed 2.4 GHz and 5 GHz spectrum [10]. This requires that two wireless interfaces are available on the smart phones [11].

Moreover, service operators have identified that offloading of bulky Internet traffic onto alternative access technologies constitutes a viable solution to relieve the high infrastructure costs [12]. Since 2000, there has been an extraordinary growth of research on SDN, initially in the area of wired networks and subsequently towards wireless technologies [13, 14]. SDN can be utilized to configure not only the radio side of access points but also the end-user terminals. While not explicitly mentioned as SDN, injecting operator-specific offloading policies into the end-user terminals is also possible. Abstraction of such policies from the terminal side constitutes an important strategy for SDN deployment.

There are several technologies that aggregate LTE and WLAN, such as Access Network Discovery and Selection Function (ANDSF) [15], LTE WLAN integration with IPsec tunnel (LWIP), and LTE-WLAN Aggregation (LWA) [16]. Furthermore, network-assisted device-to-device (D2D) offloading enables user equipment to communicate directly with each other, without relying on the conventional infrastructure of APs or BSs [17, 18]. 3GPP had invested considerable effort to ratify the IP traffic offloading solutions for the EPC: these approaches rely on tight cellular operator control and integration into the 3GPP network architecture.

Overall, rule-based policies like ANDSF are insufficient to represent the complex and/or stateful operation, such as in LWA and LWIP. For this reason, there is a need to introduce an appropriate finite state machine (FSM), which allows creating the desired stateful protocol operation via simple primitives (e.g., transmit a message, establish an IPsec tunnel, associate with an AP, etc.) that are pre-implemented in the device. Notably, the SDN technology evolved further due to the utilization of OpenFlow as a realistic and viable platform to the switch hardware [19]. The heart of OpenFlow is the "match/action" abstraction, which comprises {*rule, action*} pairs: if a rule is matched by the incoming packet, an action associated with this rule is executed.

There are three main operations that correspond to the said abstraction: (i) selection of the fields to be matched; (ii) query in the MAT (Match-Action Table) being efficiently supported in hardware by Ternary Content Addressable Memories (TCAMs); and (iii) execution of the corresponding action(s) selected

among a fixed set of the standardized ones. Lately, the need for a more flexible OpenFlow emerged. It is thus vital to improve the programmability and the flexibility of the matching procedure, as well as the way we analyze the packets [20, 21]. It is possible to develop a packet processor, and it is easy to understand that it can represent a bottleneck with respect to the delay of entry/exit of packets, as it is shown in Fig. 1.

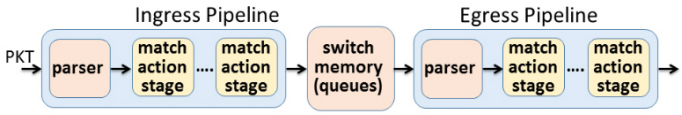


Fig. 1. A typical OpenFlow pipeline architecture

In this work, we evaluate the existing software libraries that allow for dynamic packet analysis and modification. We also elaborate on the development of flexible tools that enable fast and straightforward packet parsing, which may play a significant role in future softwarized networks. The rest of this paper is organized as follows. Section 2 outlines the design of the proposed parsing toolbox, which enables fast and reliable software oriented packet parsing. Further, the main functionality is detailed in Section 3. Section 4 provides a comparison of several existing parsers with the developed one. The main conclusions are drawn in the last section.

2 Design of Parsing Software for SDN

The best way to capture packets, analyze them, and understand which kinds of packets are to be processed is through a dynamic parser. Numerous packet parsers have been developed over the years, but it is still difficult to find *not a machine-oriented* one. For this reason, there is a need to develop a new more flexible parser. Our goal is to create a framework that is easily modifiable (including the source code), machine-oriented, and friendly to use. The developed parser should be compatible with any existing packet.

Our parser is written in Python, which contains all of the necessary functions required to analyze a packet (e.g., read data, compare data, convert data, etc.). It has a JSON "instruction file", which contains all the needed instructions and details to analyze the protocols as well as extract the requested conditions (see Fig. 2). Here, JSON was selected for its broad adoption. The purpose is to store primitive types as supported by JSON in a human-readable and straightforward format.

The bottleneck with parsing JSON and XML usually is not the parsing itself, but the interpretation/representation of the data. An event-based XML parser is typically very fast, but constructing a complex DOM tree with thousands of small objects is not. If it is necessary to parse XML to the nested native data

structures, such as lists and dictionaries, the slow part will be the interpretation of the parsing results, not the actual string analysis. Since JSON parses directly into those primitive types rather than a complex object tree, it will likely be faster.

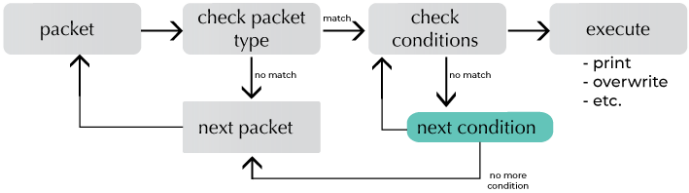


Fig. 2. Diagram of a custom parser

2.1 Configuration File Structure

Essentially, a JSON file that we need for retrieving the fields from the packets is divided into three blocks, and an example is given in Fig. 3.

```
{
  "enc_type": "1",
  "objects": [{
    "name": "ETH",
    "properties": [{
      "name": "Destination",
      "read_from": 0,
      "read_to": 6
    },{
      "name": "Source",
      "read_from": 6,
      "read_to": 12
    }],
    "read_from": 0
  }],
  "zones": [{
    "name": "ALL",
    "read_from": 0,
    "find": {
      "object": "ETH",
      "multiple": false,
      "print": ["Destination","Source"]
    }
  }]
}
```

Fig. 3. MAC address retrieval example

Explanation of "enc_type" is a field, which contains the encapsulation type of the entire capture. An integer within this field must be provided, which should

correspond to the encapsulation type from the *pcap* file. In a logical order, it is actually the first parameter that the algorithm checks. Table 1 represents a part of the list of possible types of encapsulation.

Table 1. Examples of possible types of encapsulation

LINKTYPE_name	**enc_type**	**DLT_ name**	**Description**
LINKTYPE_ETHERNET	1	DLT_EN10MB	IEEE 802.3 Ethernet (10Mb, 100Mb, 1000Mb, and higher).
LINKTYPE_IEEE802_11	105	DLT_IEEE802_11	IEEE 802.11 WLAN.

Explanation of "objects" is an array containing the definitions and instructions of the objects. Inside this JSONObject, there are two mandatory fields and one optional field, as it is possible to observe in Table 2.

Table 2. List of fields inside "objects"

Name	**Type**	**Mandatory**
name	String	Yes
read_from	int/JSONArray	No
read_to	int/JSONArray	No
match	JSONArray	No
match_or	JSONArray	No
properties	JSONArray	Yes

Further, we describe the meaning of the following fields:

- *"name"* contains an easy to read string for the object to find;
- *"read_from" and "read_to"* indicate the position of bytes, which should be read from the packets;
- *"match" and "match_or"* check if the data extracted matches the chosen interval data;
- *"properties"* contains additional properties of the object.

Fields "read_from" and "read_to" represent the precise relative bytes where the parser can start (or stop) reading data (e.g., if TCP payload starts at the 50th byte, but it is only the 20th byte in an IP packet, *"read_from"* should be 20). For more complex protocols, there is no fixed position to start (or stop) reading, so we have to acquire this value from the packet itself. In this case, we

can use a JSONObject instead of a simple *integer* number, which can contain all of the information to obtain the *read_from* value. The *"convert"* field can convert the read data (usually obtained in a byte format) into one of these formats: *int*; *int-DWORD* (it is an int value multiplied by four); *string*; or *binary*.

For more complex situations (e.g., IP header length), we have to use another parameter, *"edit_selection"*, which contains a JSON Object required to extract information from data. For example, an IP PDU starts after an IP header. The IP header length is stored inside the second half of the first IP byte, and the value is stored as an *int-DWORD*. Hence, one has to extract the first byte and convert it into a binary format. Then, one needs to acquire the last 4 bits, convert them into an integer, and multiply by four. The JSON code for this situation is represented in Fig. 4.

```
{
"read_from": {
  "read_from": 0,
  "read_to": 1,
  "convert": "binary",
  "edit_selection": {
    "convert_from": "binary",
    "read_from": 4,
    "read_to": 8,
    "convert": "int-DWORD"
    }
  }
}
```

Fig. 4. Example of read_from and edit_selection fields

Field "properties" is a JSONArray containing all the snippets of information that one desires to extract from a packet. When a property has been extracted, it can be overwritten or printed (see below). It is composed of the following parameters: *"name"*; *"read_from"*; *"read_to"*; or *"convert"*.

Explanation of "zones" Further, we have to define the relative properties of an object required to be found inside a packet. We should also consider how to nest one object inside another. For example, if the parser is attempting to find an IP packet inside a TCP packet, there will be no output. Every zone must have the fields within Table 3.

While *"name"* is but a simple label for the zone to find, *"read_from"* behaves precisely as demonstrated in the previous text.

The field "find" is basically a JSONObject or a JSONArray of objects constructed with the parameters within Table 4. Here, *"object"* can set the name of the object to find inside this data interval. It must be one of the object names that have been declared previously; *"multiple"* is useful when multiple instances of the same object need to be found. For example, it can be used when multiple tags do not have a fixed length and position; *"label"* is printed when the parser finds the required object. It is useful for debugging purposes; *"print"* prints the

Table 3. List of fields inside "zones"

Name	Type	Mandatory
name	String	Yes
read_from	int/JSONArray	Yes
find	int/JSONArray	Yes

properties of an object. The array of strings must contain only the valid property names from the object zone, or a string "all" to print all of the packet sections.

Table 4. List of fields inside "find"

Name	Type	Mandatory
object	String	Yes
multiple	Boolean	Yes
label	String	No
print	JSONArray	No

3 Main Algorithm Functionality

Among the several developed functions, the most important one inside our main is *parseZones()*. This function, which is called from the main function, calls others two crucial functions, *readData()* and *findInData()*. These will be explained in the following text.

Function parseZones() divides a packet into one or more zones as well as performs operations on them by following the instructions in the JSON.

```
1  def parseZones(data, json):
2      json_zones = json['zones']
3      newData = data
4      for json_zone in json_zones:
5          name = json_zone['name']
6          zone_data = readData(newData, json_zone)
7          newZoneData = findInData(zone_data, json_zone)
8          replace_from = getReadFrom(zone_data, json_zone)
9          replace_to = getReadTo(zone_data, json_zone)
10         newData = replace(newData, replace_from, replace_to, newZoneData)
11     return newData
```

Fig. 5. Representation of parseZones() function

The term "zone" refers to one or more parts of a packet separated from each other, where it is possible to perform operations defined by the JSON file. The partitions can be useful if there is a need to divide a packet and perform different operations for each of the corresponding zones. For example, a hypothetical 100-byte packet can be divided into four equal parts, but the payload may be located in a different position in each of them. Relying on the concept of zones, one can search for a specific payload in each partition – efficiently and timely.

Essentially, the said function performs the following two steps:

- reading the bytes obtained from the function *readData()*;
- processing the information contained by the JSON from the function *findInData()*.

This procedure is performed cyclically for each zone described in the JSON file. At the end of a cycle, the modified package is returned if specified by the JSON file. Otherwise, a complete copy of the original package is obtained. Additionally, the script has the functionality to overwrite the original zone with that eventually modified according to the instructions.

Function readData() is used to read the bytes of a packet. Locating the zones from which it is possible to read the desired bytes is done by the *getReadFrom()* and *getReadTo()* methods. These two methods return an integer that represents the index where to start reading and the index where to finish reading the packet, respectively.

```
def readData(data, json):
    output = data
    read_from = getReadFrom(data, json)
    read_to = getReadTo(data, json)
    if 'match' in json:
        output = matchData(output, json)
        if output == None: return None
    if 'match_or' in json:
        output = matchDataOr(output, json)
        if output == None: return None
    if read_from != None:
        if read_to == None:
            output = output[read_from:]
        else:
            output = output[read_from:read_to]
    output = convertData(output, json)
    output = editSelection(output, json)
    return output
```

Fig. 6. Representation of readData() function

After that, any checks are performed by using the methods *matchData()* and *matchDataOr()*, which verify that a certain part of the packet is equal to a certain

value present in the JSON file. At this point, the function extracts the real bytes from a packet and converts them, if necessary, into another format (e.g., byte ← string) via the *convertData()* function. After this conversion, it may be necessary to extract an even smaller part of the selection; in these cases, the *editSelection()* function is used.

Function findInData() allows to search for the objects specified in the JSON file within a zone. This search can be of either type: single or multiple.

```
def findInData(data, json):
    if 'find' not in json:
        return data
    else:
        find = json['find']
        if isinstance(find, list):
            output = bytearray()
            for my_find in json['find']:
                find_object = my_find['object']
                find_multiple = my_find['multiple']
                output = output + findObjectsInDataAndDoThings(data, find_object, my_find, find_multiple)
            return output
        else:
            find_object = find['object']
            find_multiple = find['multiple']
            output = findObjectsInDataAndDoThings(data, find_object, find, find_multiple)
            return output
```

Fig. 7. Representation of findInData() function

In the first case, the search stops when an object is found for the first time in the specified range of bytes, and then only a range is returned where to perform all of the following operations – as specified by the JSON file – using the *findObjectsInDataAndDoThings()* function. In the second case, the search stops when the last byte of the search interval has not arrived. In this way, if n intervals corresponding to the search terms are found, all of the subsequent procedures specified by the JSON for each of these n intervals are performed using *findObjectsInDataAndDoThings()*.

4 Performance Evaluation and Benchmarks

One of the goals of this work was to develop a fast parser to reduce the analysis time and overall delay as much as possible. A major challenge in this scenario is a large number of packets that can arrive simultaneously. The first version of the developed parser only includes analysis from a *pcap* file to evaluate the maximum reachable processing speed. Further versions will include the feature of directly scanning the interface (e.g., from a rooted Android smartphone one can access LTE or WLAN interfaces).

To evaluate the performance of the developed packet processor, simulations have been conducted with a *pcap* file composed of 1,000 DNS packets (UDP);

10,000 TCP packets; and 100 HTTP packets. To confirm the usability of our tool, we compared it with the well-known parsing libraries: dpkt[4] and Scapy[5].

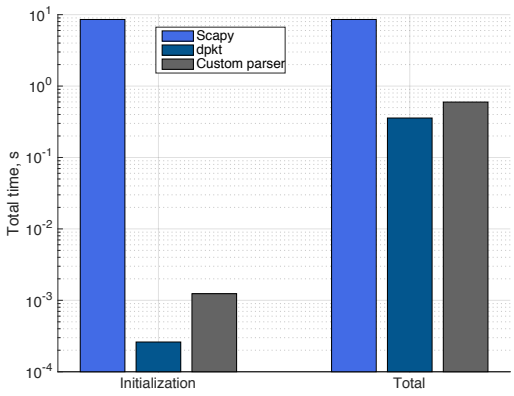


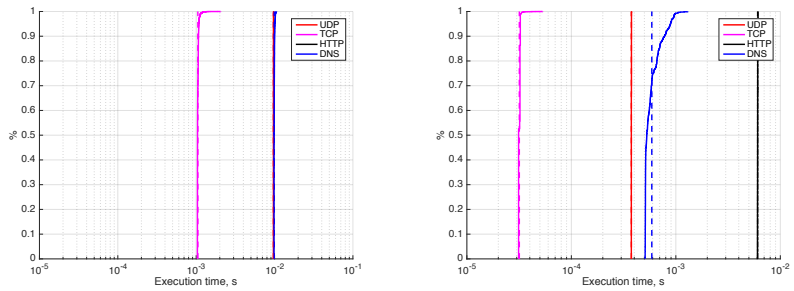
Fig. 8. Initialization and total time comparison

Initialization phase Before proceeding with the actual results, it is important to analyze the initialization time of the parsing software, see Fig. 8. Here, it is easy to see that the Scapy library is operating in the 'offline' mode, i.e., while working with a *pcap* file, it executes the actual parsing during the file read procedure. Our custom framework operates similarly to the dpkt tool, i.e., the actual parsing occurs when a standalone packet is analyzed. Hence, the initialization phase is high-speed. Another effect shown in this figure is a comparison of the total parsing time for the same set. The developed software demonstrates a relative gain even compared to dpkt.

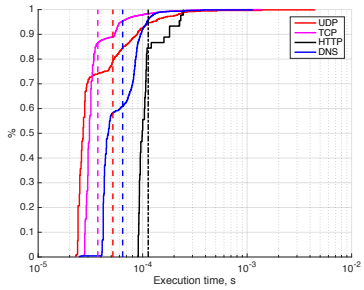
Comparison of parsers Further, we analyze different packets per parser in the form of a cumulative distribution function (CDF). As it is displayed in Fig. 9(a), the parsing time for most of the packets is relatively similar. This is due to the effect of pre-parsing during the file read procedure. However, TCP parsing is consuming the most effort.

Further, the dpkt framework is analyzed. As it is shown in Fig. 9(b), this parser operates under entirely different conditions. Since dpkt conducts parsing based on a pre-validation of the packet type, each of those provides completely different results. At the same time, the behavior in case of TCP remains the fastest.

[4] "dpkt 1.9.1", 2018: `https://pypi.python.org/pypi/dpkt`
[5] "Scapy library", 2018: `http://www.secdev.org/projects/scapy/`



(a) Scapy (b) dpkt



(c) Custom parser

Fig. 9. Parsing time comparison

Finally, we evaluate our developed tool, and the results are collected in Fig. 9(c). The fluctuating behavior of CDFs may be explained as a result of the different payloads of packets involved in the analysis. Overall, we conclude that the custom parser operates faster than Scapy or dpkt.

Comparison of packet types Here, the focus is set on the packet parsing time comparison per parser. We show which one to select for the corresponding needs of a developer. As it is demonstrated in Fig. 10, UDP, DNS, and HTTP packets all confirm the benefits of utilizing our custom parser over the conventional alternatives. Only for TCP in case of dpkt, some difference in the execution speed is present. However, it can be considered negligible.

5 Conclusions

Wireless networks are constantly evolving in the offered connectivity levels, thus strongly consolidating in our lives as a necessity. More and more devices are joining the networks to request continuous high-quality service and produce a vast amount of mobile data, which poses unprecedented challenges for the network

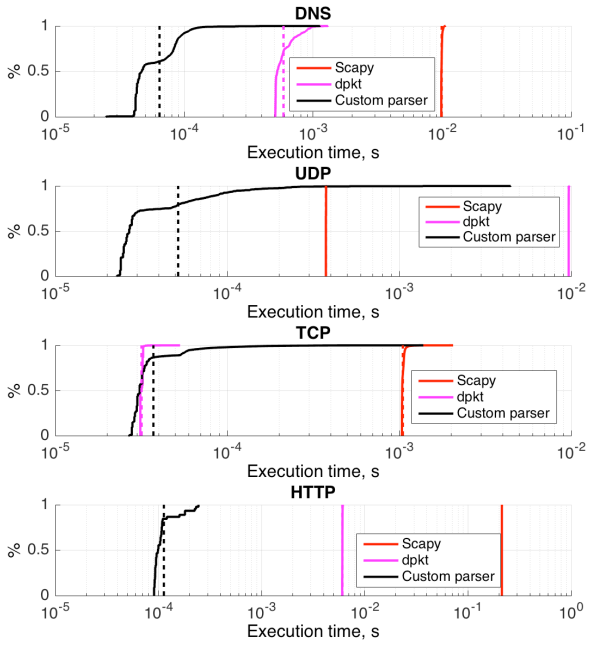


Fig. 10. Parsing time per packet comparison

design and implementation in the upcoming 5G era. Transformation of mobile user experience demands complex changes in both network infrastructure and device operation, where user experience is optimized by taking into account the surrounding network context.

Along these lines, Software Defined Networking can become essential to mitigate the network overload due to its programmable and centralized controller features, which decide – via the use of a finite state machine – how to manage the network offloading efficiently. The software libraries that exist today (e.g., Scapy and dpkt) may not be effective enough to support the requirements of emerging systems. In contrast, our proposed parser may be employed on any machine to help improve the SDN performance as well as introduce new features due to its universal compatibility with any packet. It demonstrates significant benefits over the counterpart parsing libraries with respect to the execution times.

Acknowledgment

This work is supported by the “RUDN University Program 5 – 100”.

References

1. VNI Cisco: Global mobile data traffic forecast 2016–2021. White Paper (2018)
2. Mäkitalo, N., Ometov, A., Kannisto, J., Andreev, S., Koucheryavy, Y., Mikkonen, T.: Safe and secure execution at the network edge: A framework for coordinating cloud, fog, and edge. IEEE Software (2018)
3. Karakus, M., Durresi, A.: A survey: Control plane scalability issues and approaches in Software-Defined Networking (SDN). Computer Networks **112**, 279–293 (2017)
4. Florea, R., Ometov, A., Surak, A., Andreev, S., Koucheryavy, Y.: Networking Solutions for Integrated Heterogeneous Wireless Ecosystem. CLOUD COMPUTING p. 103 (2017)
5. Xia, W., Wen, Y., Foh, C.H., Niyato, D., Xie, H.: A Survey on Software-Defined Networking. IEEE Communications Surveys & Tutorials **17**(1), 27–51 (2015). https://doi.org/10.1109/COMST.2014.2330903
6. Ordonez-Lucena, J., Ameigeiras, P., Lopez, D., Ramos-Munoz, J.J., Lorca, J., Folgueira, J.: Network slicing for 5G with SDN/NFV: Concepts, architectures, and challenges. IEEE Communications Magazine **55**(5), 80–87 (2017)
7. Volkov, A., Khakimov, A., Muthanna, A., Kirichek, R., Vladyko, A., Koucheryavy, A.: Interaction of the IoT traffic generated by a smart city segment with SDN core network. In: Proc. of International Conference on Wired/Wireless Internet Communication. pp. 115–126. Springer (2017)
8. Laselva, D., Lopez-Perez, D., Rinne, M., Henttonen, T.: 3GPP LTE-WLAN Aggregation Technologies: Functionalities and Performance Comparison. IEEE Communications Magazine **56**(3), 195–203 (2018)
9. Ometov, A.: Short-range communications within emerging wireless networks and architectures: A survey. In: Proc. of 14th Conference of Open Innovations Association (FRUCT). pp. 83–89. IEEE (2013)
10. Galinina, O., Andreev, S., Gerasimenko, M., Koucheryavy, Y., Himayat, N., Yeh, S.P., Talwar, S.: Capturing spatial randomness of heterogeneous cellular/WLAN deployments with dynamic traffic. IEEE Journal on Selected Areas in Communications **32**(6), 1083–1099 (2014)
11. Ometov, A., Masek, P., Urama, J., Hosek, J., Andreev, S., Koucheryavy, Y.: Implementing secure network-assisted D2D framework in live 3GPP LTE deployment. In: Proc. of International Conference on Communications Workshops (ICC). pp. 749–754. IEEE (2016)
12. Andreev, S., Galinina, O., Pyattaev, A., Hosek, J., Masek, P., Yanikomeroglu, H., Koucheryavy, Y.: Exploring synergy between communications, caching, and computing in 5G-grade deployments. IEEE Communications Magazine **54**(8), 60–69 (2016)
13. Feamster, N., Rexford, J., Zegura, E.: The road to SDN: an intellectual history of programmable networks. ACM SIGCOMM Computer Communication Review **44**(2), 87–98 (2014)
14. Volkov, A., Muhathanna, A., Pirmagomedov, R., Kirichek, R.: SDN Approach to Control Internet of Thing Medical Applications Traffic. In: Proc. of International Conference on Distributed Computer and Communication Networks. pp. 467–476. Springer (2017)
15. Grebeshkov, A., Gaidamaka, Y., Zaripova, E., Pshenichnikov, A.: Modeling of vertical handover from 3GPP LTE to cognitive wireless regional area network. In: Proc. of 9th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT). pp. 1–6. IEEE (2017)

16. Määttanen, H.L., Masini, G., Bergström, M., Ratilainen, A., Dudda, T.: LTE-WLAN aggregation (LWA) in 3GPP Release 13 & Release 14. In: Proc. of Conference on Standards for Communications and Networking (CSCN). pp. 220–226. IEEE (2017)
17. Shen, X.: Device-to-device communication in 5G cellular networks. IEEE Network **29**(2), 2–3 (March 2015). https://doi.org/10.1109/MNET.2015.7064895
18. Pyattaev, A., Johnsson, K., Surak, A., Florea, R., Andreev, S., Koucheryavy, Y.: Network-assisted D2D communications: Implementing a technology prototype for cellular traffic offloading. In: Proc. of Wireless Communications and Networking Conference (WCNC). pp. 3266–3271. IEEE (2014)
19. Gerasimenko, M., Moltchanov, D., Florea, R., Himayat, N., Andreev, S., Koucheryavy, Y.: Prioritized centrally-controlled resource allocation in integrated multi-RAT HetNets. In: Proc. of 81st Vehicular Technology Conference (VTC Spring). pp. 1–7. IEEE (2015)
20. Pontarelli, S., Bruschi, V., Bonola, M., Bianchi, G.: On offloading programmable SDN controller tasks to the embedded microcontroller of stateful SDN dataplanes. In: Proc. of IEEE Conference on Network Softwarization (NetSoft). pp. 1–4 (July 2017). https://doi.org/10.1109/NETSOFT.2017.8004225
21. Pontarelli, S., Bonola, M., Bianchi, G.: Smashing SDN "built-in" actions: Programmable data plane packet manipulation in hardware. In: Proc. of IEEE Conference on Network Softwarization (NetSoft). pp. 1–9 (July 2017). https://doi.org/10.1109/NETSOFT.2017.8004106