ISSN 0361-7688, Programming and Computer Software, 2016, Vol. 42, No. 5, pp. 316–323. © Pleiades Publishing, Ltd., 2016.
Original Russian Text © A.I. Get'man, V.P. Ivannikov, Yu.V. Markin, V.A. Padaryan, A.Yu. Tikhonov, 2015, published in Trudy Instituta Sistemnogo Programmirovaniya, 2015, Vol. 27, No. 4, pp. 5–22.

Data Representation Model for In-Depth Analysis of Network Traffic

A. I. Get'man[a]*, V. P. Ivannikov[a,b,c,d], Yu. V. Markin[a]***, V. A. Padaryan[a,b]****, and A. Yu. Tikhonov[a]*****

[a] *Institute for System Programming, Russian Academy of Sciences, ul. Solzhenitsyna 25, Moscow, 109004 Russia*
[b] *Moscow State University, Moscow, 119991 Russia*
[c] *Moscow Institute of Physics and Technology, Institutskii per. 9, Dolgoprudnyi, Moscow oblast, 141700 Russia*
[d] *National Research University Higher School of Economics, ul. Myasnitskaya 20, Moscow, 101000 Russia*
*E-mail: *thorin@ispras.ru, **ivan@ispras.ru, ***ustas@ispras.ru, ****vartan@ispras.ru, *****fireboo@ispras.ru*
Received May 5, 2016

Abstract—This paper proposes a new object model of data for the in-depth analysis of network traffic. In contrast to the model used by most modern network analyzers (for example, Wireshark and Snort), the proposed model supports data stream reassembling with subsequent parsing. The model also provides a convenient universal mechanism for binding parsers, thus making it possible to develop completely independent parsers. Moreover, the proposed model allows processing modified—compressed or encrypted—data. This model forms the basis of the infrastructure for the in-depth analysis of network traffic.

DOI: 10.1134/S0361768816050030

1. INTRODUCTION

The problem of traffic analysis is now becoming increasingly important due to the progress in network technologies and the advent of a great number of new application layer protocols. Some practical aspects of such an analysis are

- detecting network malfunctions;
- testing (debugging) network protocols [1, 2];
- statistics gathering and network monitoring;
- intrusion detection and prevention [3].

There are a great many commercial and freeware network analyzers [4]. Each such tool is usually oriented to solving one particular problem, while its basic functions generally include protocol headers parsing and data streams reassembling.

The majority of modern network analyzers operate in two modes:

- the real-time analysis of the traffic (hereinafter, the *online* analysis);
- the analysis of previously saved file with traffic (hereinafter, the *offline* analysis).

In the online analysis mode, the tool has to operate continuously with the efficiency sufficient for parsing the traffic flowing through the network interface. In this case, the analyzer must be capable of processing a potentially infinite input data stream.

In the offline analysis mode, the tool receives input (finite) data from a file, which enables a more detailed analysis as compared to the online mode for similar traffic.

In practice, for most presently available tools, the offline analysis is completely equivalent to the online analysis except that the packets are read from the file rather than from the network interface. The freedom from the constraints on the data processing rate in the offline mode provides additional possibilities for

- visualizing the structure of all parsed data;
- analyzing the reassembled data streams of the application layer;
- applying other parsers (different from the parsers used by the system) to the data when viewing the results;
- debugging the protocol parsing modules;
- investigating the intrusion events.

The constraints imposed on the in-depth offline analysis are mostly due to the architectural design of network analyzers: most of them are primarily oriented to the online analysis. In this case, the subsequent parsing of reassembled data streams is impossible. Therefore, we propose to implement two *independent* systems for online and offline analysis, respectively. Both systems should use a unified infrastructure, including parsers (the complete list of requirements is given below). Such a unified infrastructure allows taking full advantage of the offline analysis, while enabling the protocol headers parsing and data streams reassembling in the online mode.

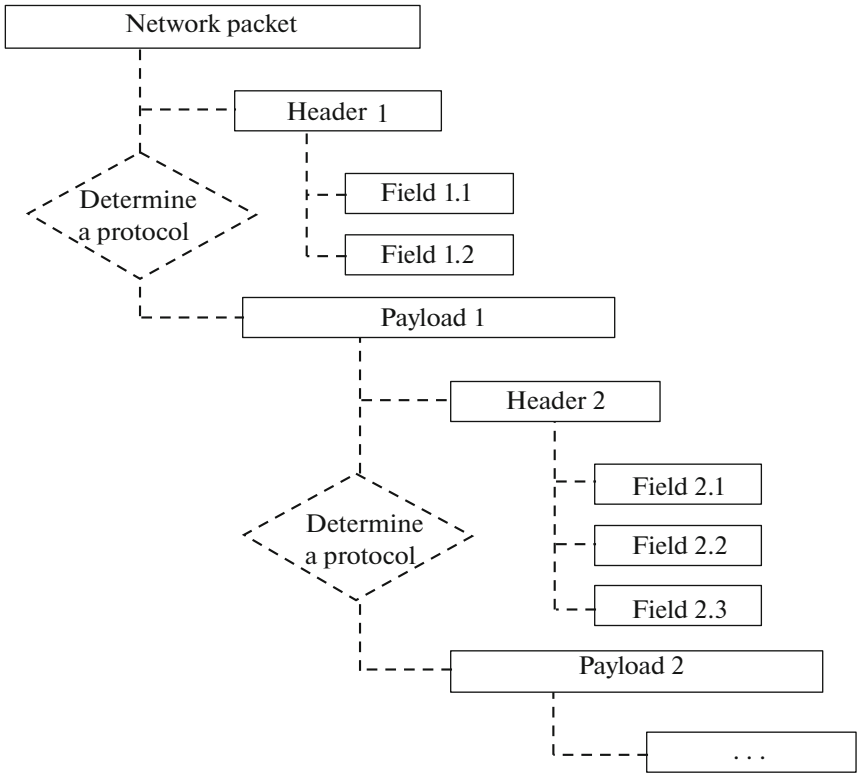

Fig. 1. Protocol headers extraction and parsing.

The main result presented in this paper is an object model that is used by these two systems.

The requirements to the online systems for in-depth traffic analysis are quite elaborated: the ITU-T recommendations were approved in 2012 [5]. In contrast, the requirements for the offline systems have not yet been refined. Section 2 presents a list of refined functional requirements for both the systems (based on these requirements, the corresponding requirements for the proposed model are formulated). Section 3 describes the model used by modern network traffic analyzers (for example, Snort [6], Wireshark [7], and Bro Network Security Monitor [8]) with the emphasis being placed on its various limitations. Section 4 discusses how exactly these limitations are overcome in the proposed model. The results are summed up in the Conclusion section.

2. REQUIREMENTS FOR A PARSING SYSTEM

Hereinafter, network data are assumed to be transmitted in packets. A network packet consists of control information and payload. In the process of parsing, the protocol headers in the packet are marked up (detailed) and the values of the header fields are analyzed. While the header structure is specified, the payload can contain arbitrary data. The latter is usually a packet of a higher-layer protocol that should be identified by the analyzer. If the identification is successful, then parsing continues (see Fig. 1).

According to the OSI model, the network packet data can be interpreted using the stack of protocol headers. The standard sequence order of these headers is from lower-layer to higher-layer protocols. In the case of tunneling, however, this sequence order can be violated. The parsing system should handle such situations correctly.

Some network protocols (for example, IPv4 [9]) are limited by the maximum allowed size of the payload per a packet, which is referred to as the maximum transmission unit (MTU) of a protocol. Thus, if the MTU value is exceeded, then the payload is divided into fragments of a permissible size so as to be transmitted in several packets. In such cases, the parsing system has to carry out the defragmentation of data (see Fig. 2).

To provide a higher security of network communication, many protocols (particularly, TLS [10–12] and SSH [13]) support data encryption. To parse the encrypted data, they first need to be decrypted with the key provided by the user (see Fig. 3). Generally speaking, the parsing system should provide the user with the interface for entering some additional data required for parsing. This requirement is not mandatory according to the ITU-T recommendations.

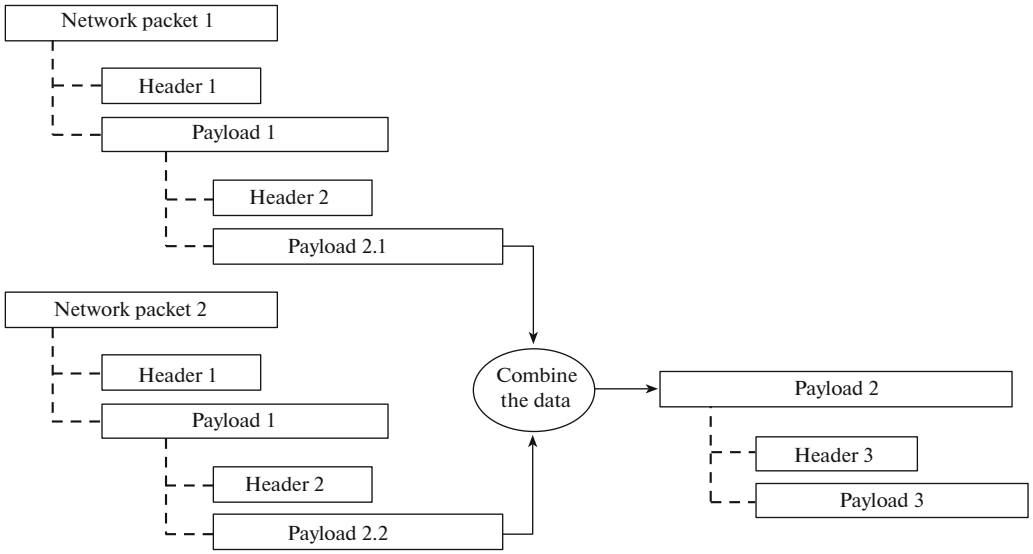

Fig. 2. Data defragmentation (combining).

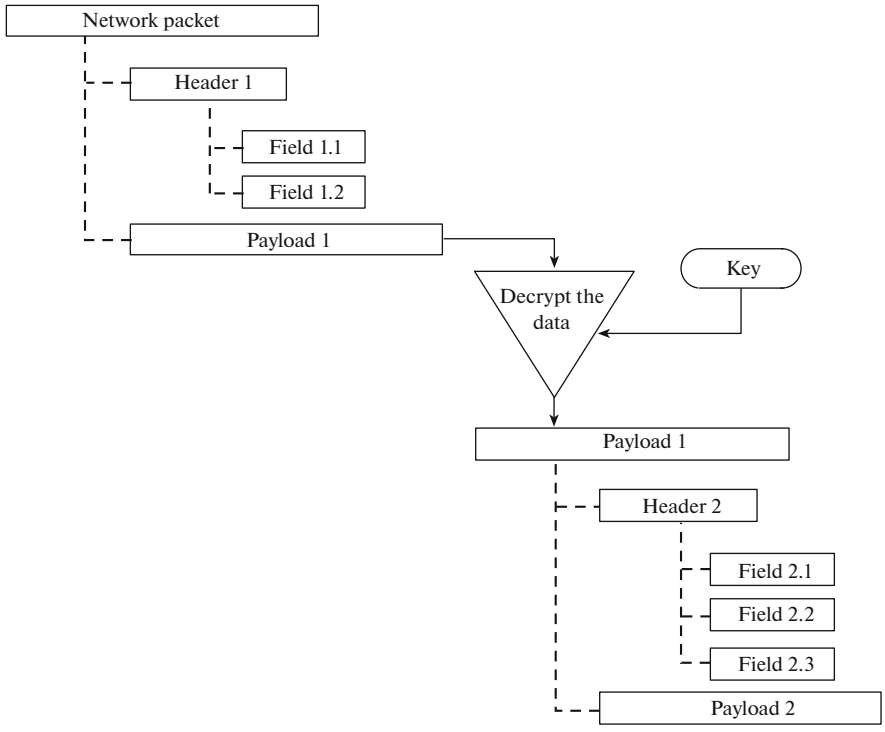

Fig. 3. Data decryption (transformation).

Traffic analysis is inevitably accompanied by *parsing errors*. A parsing error is an inconsistency between the protocol specification (parser code) and the network packet parsed according to this specification. Parsing errors occur due to various reasons:

- errors in the parser code;
- undocumented features of protocols;
- corruption of data during transmission.

The parsing system has to enable the localization of the data parsed with errors. If an error detected is not critical, then the analysis should continue.

Network traffic analyzers (for example, Wireshark and Bro Network Security Monitor) generally have a modular architecture, which is due to the continuous

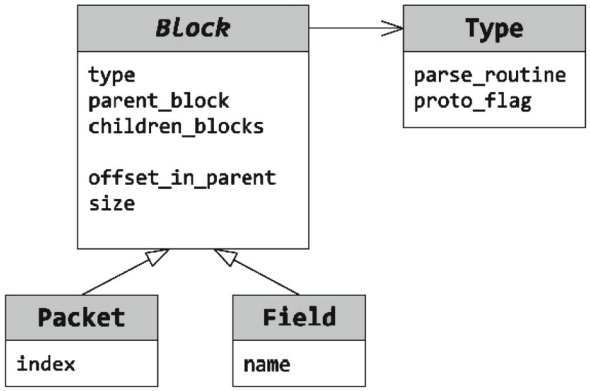

Fig. 4. Basic model of data.

development of new network protocols that need to be supported. It is difficult to extend a system in which all parsers are localized in one functional module. The modular architecture allows creating for each particular protocol an individual module to parse header(s) of this protocol. Therefore, a problem of inter-module dependencies arises: when adding a new module, the other modules must be "informed" about it. Modifying the code of already implemented modules is difficult and ineffective (it is also a potential source of errors). Moreover, such a modification requires rebuilding the module. Therefore, it seems necessary to implement a mechanism for adding new parsing modules without modifying the existing ones.

The in-depth analysis of network traffic assumes an *exhaustive* parsing: parsing the headers of the protocols of all OSI layers and saving the reassembled data streams. The support of the exhaustive parsing is a key requirement for the system.

Taking into account all the requirements formulated above, let us consider the following (main) operating scenario. The online analysis system parses packets (received from a certain network interface) 24/7. The *parsing results*—the set of all reassembled data streams (see below) for all protocols—are saved on the hard drive. The analyst monitors at certain regular intervals the error messages from the parsing system. If the number of parsing errors for a particular parser exceeds a predetermined threshold, then it seems reasonable to fix the source code of this parser. For the subsequent debugging of the parser code, the corresponding network trace is saved. The decision on modifying the parser code is made based on the results of the offline analysis of the saved network trace.

In fact, the offline analysis system is intended to acquire and accumulate sufficient knowledge about the data structure for the further improvement of the online analysis system. Therefore, it is highly important for the parsing modules of the two systems to be compatible with each other. Such compatibility can only be achieved by elaborating the architectures of these systems.

The main (quite expectable) constraint imposed on the online analysis system is a lack of computing resources. Since input data stream is potentially infinite, it is necessary (from time to time or upon occurrence of a particular event) to save the parsing results on the hard drive or to hand them over to another analyzer. It should be noted that the online analysis system does not provide any visualization of the parsed data structure.

In contrast, the offline analysis system works with the finite network trace, which imposes no constraints on the computing resources. The main purpose of this system is parser debugging. Visualizing the structure of the parsed packets considerably facilitates the debugging process. It is also not unreasonable to implement a convenient mechanism to navigate between the packets and properly reassembled streams.

3. MODERN NETWORK TRAFFIC ANALYZERS

Most modern network analyzers support both online and offline analyses. In the first case, network card acts as a source of packets; in the second case, such a source is a network trace file. Packet parsing consists in extracting the fields of all protocol headers in the packet. The extraction of a field means that a certain continuous data *block* of a known size is made to correspond to this field. The extracted blocks may acquire certain semantics (for example, a block specifying the packet length).

Most modern parsing systems use the concepts of a packet, protocol, and data block. A data model corresponding to such an approach (hereinafter, the *basic model*) is shown in Fig. 4.

Each data block stores a pointer to a *parent* block (parent_block) and a list of pointers to *children* blocks (children_blocks). A parent block is a block containing a given block, while a child block is a block contained in a given one. This enables block nesting, so a network packet (data buffer) can be associated with a corresponding parse *tree*. Note that the size of a block is a sum of sizes of all its child blocks (if any).

Each block has a *size*, *offset* relative to the parent block (offset_in_parent), and *type*. The type provides a parser (parse_routine) that is suitable for analyzing the data in a given block. In addition, the type has a flag (proto) indicating (if set) that the block of this type is a header of a network protocol. This enables block grouping and allows emphasizing protocol headers when viewing the structure of the parsed packets.

Each block also has an additional attribute: either *name* (string) or index (integer). The indexed blocks describe network packets and array elements, while the named blocks describe fields of protocol headers.

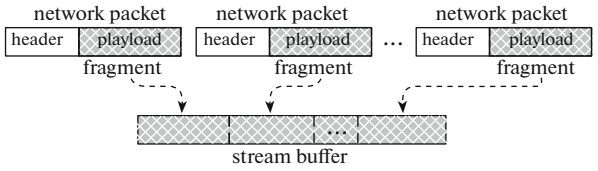

Fig. 5. Adding the data from several fragments into the stream buffer.

In the case of the basic model, the data are processed as follows. A data buffer (in the offline mode, a network trace) is inputted to the parsing system. Then, a proper parser is selected according to the type of the data in the buffer. The parser extracts the blocks and, if necessary, calls other parsers for them. The parsing system yields a certain tree representation of the buffer with each extracted block corresponding to the node of the tree. The system allows the analyst to view the data in each block.

The basic model has the following limitations.

1. Each parser has to determine directly which parser should be called next. The system thus loses its flexibility: when adding new parsers , the existing ones have to be modified so as to make them capable of using these new parsers. In fact, this means that, in order to provide the ability of adding new parsers, the developer has to open the source code of all already implemented parsers. None of the commercial analyzers allow this.

2. The tree representation cannot be applied to the packets containing encrypted or compressed data. Meanwhile, the share of such (modified) data circulating through the networks steadily increases.

3. To describe the *assembly of sessions*, more than one buffer containig all packets is required. In this case, a *session* is a reassembled (completely or partially) transmission unit of the streaming protocol that is sent (or received) through more than one network packet. In certain analyzers (for example, Snort and Bro Network Security Monitor), the sessions are reassembled by plug-in modules but without subsequent parsing. Yet, analyzing the reassembled sessions makes it possible to restore high-level data of the application protocols, thereby clarifying the logic of the interaction between network nodes.

It should also be noted that most analyzers are oriented either to the long-term operation in a statistics gathering mode without exhaustive parsing and saving the data structure or to the short-term operation in a data accumulation mode with subsequent parsing and viewing the data structure.

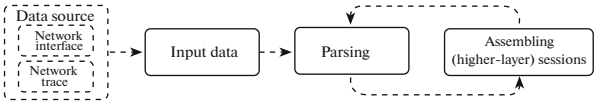

Fig. 6. Data processing scheme.

4. FORMAL DESCRIPTION OF THE DATA REPRESENTATION MODEL

Hereinafter, we refer to the proposed model of network data as the extended model (in contrast to the basic model).

In the extended model, the parsing process consists in extracting logically continuous data blocks. Essentially, such a block is a generalization of a field in the basic model: in the extended model, a field is a block with particular characteristics. When analyzing the block, it may be required to process some of its parts individually. To do so, a new block is created that is a child block relative to the original one. The concepts of the parent and children for the blocks are similar to those for the fields. Each block has a size (positive number). While parsing, each block is made to correspond to a particular type. This (parsing) type sets a parser suitable for analyzing data in a given block. Each type has a unique name.

To describe the assembly of sessions, the concept of a *data buffer* is introduced. The data from the blocks can be added to the end of the buffer by copying. The size of the buffer is always a sum of the sizes of the data blocks added into it. The content of the buffer is analyzed (by parsers) just like the data of the blocks. It is essential to distinguish between the blocks that own the buffer (*streams*) and the blocks that do not (*fragments*). When it is required to assemble a session (particularly, when parsing TCP packets [14]), a stream is created and the necessary data of fragments are then added into its buffer (see Fig. 5). Note that the stream can be created or supplemented with any parsing function.

It should also be noted that the data of a fragment are localized by an *offset value* relative to the beginning of the data in a parent block. When a parent block is a stream, the offset is done relative to the beginning of the buffer of this stream.

Session assembling sequentially raises the data representation level (in terms of the OSI model): once assembling is complete, the session is parsed with the possibility of reassembling the sessions for higher-layer protocols (see Fig. 6).

To make the parsing system capable of handling modified (e.g., encrypted) data, the concept of a *substitutional* fragment is introduced (the other fragments are hereinafter referred to as *simple*). Encrypted data cannot be parsed directly: a preliminary decryption is required (in this case, the analyst may know the decryption algorithms and the values of the encryp-

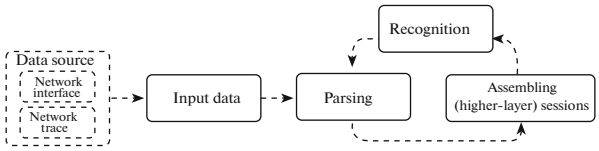

Fig. 7. Extended data processing scheme.

tion keys used). The decrypted data are placed into a preliminary created buffer associated with the fragment that describes the encrypted data. Thus, the fragment can have an additional memory buffer, and it is the data of this (substitutional) buffer that are subjected to parsing. The proposed approach can be generalized to any modification of data that allows for inverse transformation. The child blocks of the substitutional fragment have their offset values relative to the beginning of the data in the corresponding substitutional buffer. Note that, in the general case, the size of the substitutional buffer is by no means related with the size of the *original* data in the fragment. It should also be noted that, to describe the substitutional fragments within the tree structure, a distinctive feature needs to be introduced for the nodes that own the buffer (such nodes are hereinafter referred to as *data sources*). Then, a simple fragment is characterized by the offset relative to the "nearest" (when passing to the parent) data source.

To implement universal parsers (that do not require code modifications when adding new types), the concept of a *recognizer* is introduced. A recognizer is a function that, based on the block data and, perhaps, some additional information, identifies the type of a given block. At the beginning of parsing, the type of the block is either known or not. In the latter case, the type of the block is identified by the recognizers. The parsing system allows the analyst to register new recognizers and ensures the correct use of the already registered ones.

Such recognition is mostly required for the fields in protocol headers that contain data of various types. For instance, the field "payload" in the header of the IPv4 protocol can contain data of other protocols such as TCP, UDP [15], etc. In this case, the type is determined by the value of the field "protocol" of the same header. More formally, the type of a fragment is identified using the data of its parent. Such recognizers are called *field recognizers*. To register a field recognizer in the system, it is required to specify the name of the corresponding field and the type of the parent block. The field recognizer is used only for the field with a specified name provided that the corresponding parent block has a specified type.

To identify the type of assembled sessions, *stream recognizers* are used. As input data, a stream recognizer uses only data of the corresponding buffer. In addition, the *assembly* type of the stream can be used for

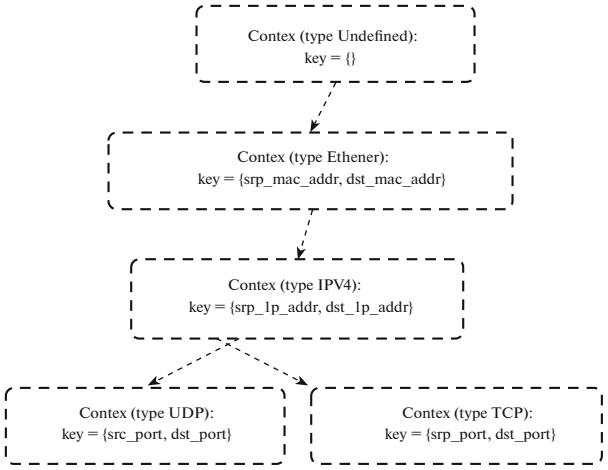

Fig. 8. Example of a context tree.

recognition. An assembly type is determined based on the type of the data blocks added into the stream buffer. The assembly type allows one to narrow (if necessary) the set of recognizers applied to the stream. Such recognizers are referred to as *stream recognizers with known assembly type*. A stream recognizer that does not use the assembly type can also be applied to identifying the type of a fragment; such recognizers are referred to as *stream recognizers with unknown assembly type*.

The recognizers are an integral part of the system (see Fig. 7); their primary purpose is to provide the mutual independence between the parsers (and, therefore, the parsing modules): when adding new types into the system, it is sufficient to register the corresponding recognizer. In the opposite case where the recognition functionality is concentrated in the parsers, it would require to modify their code. Note also that recognizers can be located in any parsing module.

The proposed modifications make it possible to overcome the above-mentioned limitations of the basic model. However, a new problem arises that concerns session assembling: when adding new data, the corresponding session must be uniquely identified. As an example, let us consider the IPv4 protocol used for transferring data between network nodes with unique IP addresses. This uniqueness, however, holds only within a particular IP subnetwork, while network traffic generally represents lots of such subnetworks. Moreover, packets of one IP subnetwork can be nested into packets of another IP subnetwork. Thus, a pair of IP addresses is insufficient to identify an IP session within a global network. When transferring data via the TCP protocol, a pair of ports—sender and receiver—need to be given; i.e., to identify a TCP session, it is required to specify the values of these ports. In summary,

• different protocols use different concepts of a session; i.e., there are different *types of sessions*;

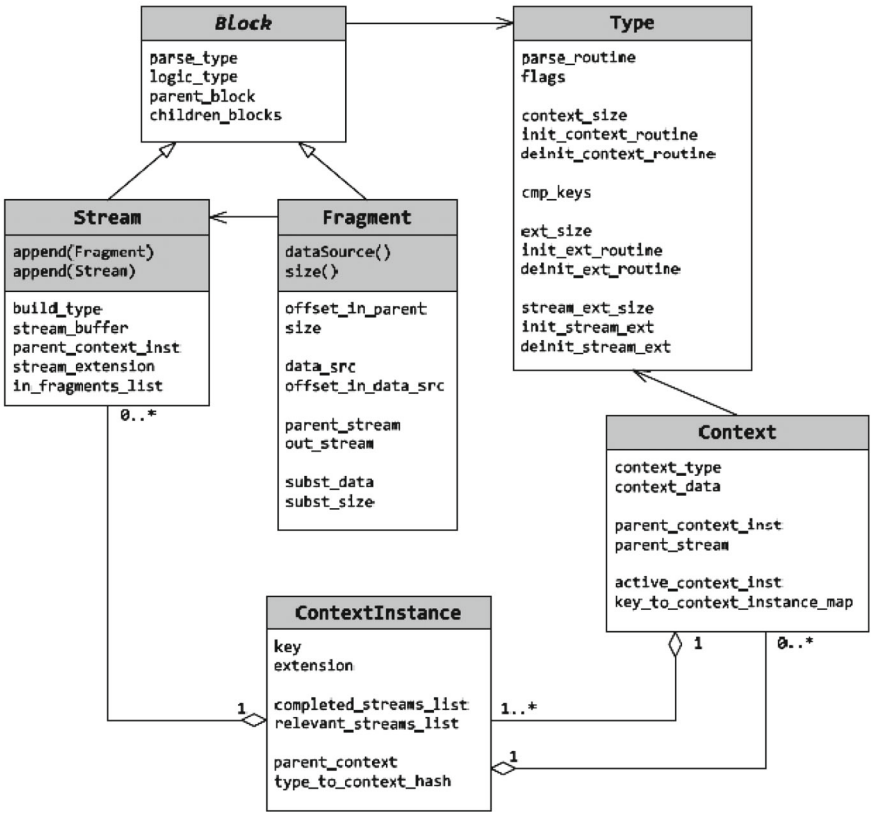

Fig. 9. Extended model of data.

• each network protocol determines a set of identification characteristics (a key) for its sessions;

• the identification characteristics of a session (determined by a protocol) are unique within a particular *context*.

For an IP session, such a context is a certain IP subnetwork, while, for a TCP session, it is an IP session in an IP subnetwork. Note that the higher is the OSI layer of a protocol, the larger is the set of identification characteristics for the session of this protocol in the global network. Particularly, for a TCP session, this set includes an IP session and its characteristics. Thus, there is a tree of contexts where each node represents the set of characteristics for the corresponding session; this set contains all characteristics of the parent node and at least one extra characteristic (see Fig. 8). For the root of the tree, this set is empty.

Each node of the context tree is associated with a set of *context instances*. A context instance is a context for which the values of all identification characteristics are known.

The concept of a stream, which is introduced above, fully describes a session with the assembly type of a stream (by definition) being the type of the session. The stream is assembled (and subsequently stored) within the context instance.

To describe the contexts, we extend the concept of a type (see above) with a feature that indicates the necessity of creating a context. If a type has this feature, then, before parsing the block of this type, the system kernel *switches* the context (particularly, before parsing the header of the IPv4 protocol). At every instant of analysis, exactly one context is *active*. Switching consists in changing the active context: if there is no desired context by the time of switching, then it is created. We refer to the type of the block that requires creating a context for its parsing as a *context type*.

Less formally, the contexts and context instances are meant for grouping the blocks of the same nesting level. In fact, this refers to the problem of traffic classification (for more detail about traffic classification methods see [16]). The grouping criterion is provided by the context type and is given by the function that, having analyzed the content of a block, classifies it as belonging to a particular group (each instance within a certain context describes one such group).

A stream is assembled within the context instance where it has been created, which implies that the

blocks belonging to different context instances cannot become parts of the same stream.

Figure 9 shows the object model that implements all the features described above. Note that, according to the type ("flags" attribute), a block is either a *structure* (similar to the "struct" in C) or a *sequence*. Therefore, in the first case, the child blocks are regarded as fields and, in the second case, as *sequence elements*. Each field is characterized by its name, while each sequence element by its index. Thus, the extended model completely covers the functionality of the basic one.

It should also be noted that, within a context instance, the streams are identified using a key (attribute "stream_extension"); the size (attribute "stream_ext_size") and structure of the key are given by the type of the corresponding context.

5. CONCLUSIONS

In this paper, an object model of data for both offline and online analyses has been proposed. Unlike the basic model used by most modern network traffic analyzers, this extended model supports session assembling and modified data processing, as well as allows one to develop completely independent parsing modules. The Wireshark analyzer operates on the basis of a similar model, but it has a significant disadvantage: session assembling is completely shifted on to the developer. At the kernel level, the Wireshark has no analogs for the concepts of a stream, context, and context instance, so they need to be created independently for each parsing module (if session assembling is required). As a result, the parser code proves to be overloaded with the session assembling logic. In contrast, the proposed model introduces the abstractions for session assembling at the kernel level, which makes it possible to considerably simplify (and shorten) the parser code, as well as to avoid unnecessary errors.

ACKNOWLEDGMENTS

This work was supported by the Russian Foundation for Basic Research, project no. 15-07-07652 A.

REFERENCES

1. Tsankov, P., Dashti, M.T., and Basin, D., SECFUZZ: Fuzz-testing security protocols, *Proc. 7th Int. Workshop on Automation of Software Test (AST),* 2012, pp. 1–7.
2. Pakulin, N.V., Shnitman, V.Z., and Nikeshin, A.V., Automation of correspondence testing for telecommunication protocols, *Tr. Inst. Sistemnogo Program. Ross. Akad. Nauk*, 2014, vol. **26**, no. 1, pp. 109–148.
3. Scarfone, K. and Mell, P., *Guide to Intrusion Detection and Prevention Systems (IDPS),* Gaithersburg: Natl. Inst. Stand. Technol., 2007.
4. Markin, Yu.V. and Sanarov, A.S., Survey of modern network traffic analyzers, *Preprint of Inst. for Syst. Program. Russ. Acad. Sci.,* Moscow, 2014, no. 27.
5. Recommendation MSE-T Y.2770: Requirements for In-Depth Analysis of Packets in Next-Generation Networks v. 1.0, 2012.
6. Snort: Network Intrusion Detection and Prevention System. http://www.snort.org.
7. Wireshark: Network Protocol Analyzer. http://www.wireshark.org.
8. The Bro Network Security Monitor. http://www.bro.org.
9. Internet protocol, IETF RFC 791, Information Sciences Institute, 1981.
10. Dierks, T. and Rescorla, E., The transport layer security (TLS) protocol v. 1.2, IETF RFC 5246, 2008.
11. Pakulin, N.V., Shnitman, V.Z., and Nikeshin, A.V., Development of a test set for verifying the implementations of the TLS security protocol, *Tr. Inst. Sistemnogo Program. Ross. Akad. Nauk*, 2012, vol. **23**, pp. 387–404.
12. Nikeshin, A.V., Pakulin, N.V., and Shnitman, V.Z., Testing the implementations of the TLS client, *Tr. Inst. Sistemnogo Program. Ross. Akad. Nauk*, 2015, vol. **27**, no. 2, pp. 145–160.
13. Ylonen, T. and Lonvick, C., The secure shell (SSH) protocol architecture, IETF RFC 4251, 2006.
14. Transmission control protocol, IETF RFC 791, Information Sciences Institute, 1981.
15. Postel, J., User datagram protocol, IETF RFC 768, 1980.
16. Risso, F., Baldini, A., Baldi, M., Monclus, P., and Morandi, O., Lightweight, payload-based traffic classification: An experimental evaluation, *Proc. IEEE Int. Conf. on Communications (ICC),* Beijing, 2008, pp. 5869–5875.

Translated by Yu. Kornienko