

Article

A Novel Network Protocol Syntax Extracting Method for Grammar-Based Fuzzing

Huashan Li ^{1,2,*}, Lei Zhang ^{1,2,*}, Dawei Zhao ^{1,2,*} , Lijuan Xu ^{1,2}, Xin Li ^{1,2}, Shumian Yang ^{1,2} and Xiaohui Han ³

¹ Key Laboratory of Computing Power Network and Information Security, Ministry of Education, Shandong Computer Science Center (National Supercomputer Center in Jinan), Qilu University of Technology (Shandong Academy of Sciences), Jinan 250014, China; lhs5137@163.com (H.L.); xulj@sdas.org (L.X.); lixin@sdas.org (X.L.); yangshm@sdas.org (S.Y.)

² Shandong Provincial Key Laboratory of Computer Networks, Shandong Fundamental Research Center for Computer Science, Jinan 250014, China

³ Quancheng Provincial Laboratory, Jinan 250014, China; hanxh@sdas.org

* Correspondence: zhanglei@sdas.org (L.Z.); zhaodw@sdas.org (D.Z.)

Abstract: Network protocol syntax information plays a crucial role in grammar-based fuzzing. Current network protocol syntax extraction methods are less versatile, inefficient, and the extracted information is not comprehensive. This paper proposes a novel method for extracting syntax information, which innovatively extracts network protocol syntax from Wireshark protocol dissector files. The extracted syntax information includes packet types of the protocol, the constituent fields of each packet type, and detailed attributes of each field. Based on this method, an automated system for network protocol syntax information extraction was developed. The experiment was conducted with this system on a variety of protocols including DCCP, DNP3.0, Modbus TCP, and S7COMM. The experimental results show that compared with the current methods, our method has a better performance in terms of efficiency and versatility and at the same time ensures the comprehensiveness and accuracy of the extracted syntax information.

Keywords: network security; Wireshark; fuzzing; network protocol; protocol syntax extraction



Citation: Li, H.; Zhang, L.; Zhao, D.; Xu, L.; Li, X.; Yang, S.; Han, X.

A Novel Network Protocol Syntax Extracting Method for Grammar-Based Fuzzing. *Appl. Sci.* **2024**, *14*, 2409. <https://doi.org/10.3390/app14062409>

Academic Editor: Christos Bouras

Received: 29 January 2024

Revised: 10 March 2024

Accepted: 11 March 2024

Published: 13 March 2024



Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Network protocols are a set of rules, standards, or conventions established for data exchange in a computer network. Network protocols consist of three elements, namely semantics, syntax, and timing, wherein the syntax is the structure and format of the exchanging data and control information [1]. It plays an important role in the process of functionality/security testing of network communication software or hardware equipment [2,3]. For example, when conducting fuzzing on network communication software, network protocol syntax information ensures the validity of generated test cases, thereby enhancing fuzzing efficiency [4–6]. However, the complexity and diversity of network protocols present significant challenges to understand and process them thoroughly. Consequently, there is an urgent need to rapidly and comprehensively extract syntax information from a wide array of different network protocols. This has emerged as a critical research area within the realms of network security and software testing [7,8].

Currently, methods for extracting network protocol syntax information can be divided into two categories, manual method and automated methods. The manual method usually relies on experts to manually extract protocol syntax information from protocol specifications, it has the advantage of high accuracy. Experts can accurately define key features of protocol syntax based on domain knowledge [9]. However, this method has a low degree of automation, the extraction process is usually time-consuming and labor-intensive. At the same time, the manual method has poor versatility and makes it difficult to process plenty of different protocols. Automated methods include methods based on software reverse

engineering, methods based on traffic analysis, and artificial intelligence methods based on protocol specifications [10]. The software reverse engineering method extracts the syntax information of the network protocol by analyzing the binary data flow of the protocol processing software [11–13]. This method is usually applied to the analysis of unknown protocols, and researchers need to have perfect knowledge and skills in assembly language and binary analysis [14]. Traffic-analysis-based methods utilize network communication data flows to obtain network protocol syntax knowledge, but there are challenges in improving information comprehensiveness. Besides, encrypted traffic makes data unreadable thus further limiting the analysis of protocol syntax [15–17]. Artificial intelligence methods use machine learning or natural language processing technology to extract structured information from text documents by learning protocol grammar rules [18–20]. By using special pre-made protocol specification data sets for model training, the machine learning model can learn the underlying patterns of protocol syntax and achieve a certain degree of automation and generality. However, the accuracy of this method is affected by many factors, including the quality of the training data, the selection of features, and the complexity of the model. Especially when facing new or unknown protocols, the applicability of this method is low. In addition, obtaining high-quality training data is also one of the challenges faced by artificial intelligence methods [21].

Considering the diversity and complexity of network protocols, the manual method definitely has difficulty in extracting protocol syntax information arbitrarily and quickly. At the same time, although artificial intelligence methods have made certain progress in dealing with these problems, they usually have an unsatisfactory performance in result completeness and they are often designed for specific protocol formats, which limits their versatility. In this context, designing an automated and universal method for extracting network protocol syntax information has become an important and challenging research issue. The designated method not only needs to improve the efficiency of protocol syntax information extraction, but also should maintain high versatility and accuracy to ensure that it is applicable to a wider range of protocols [22,23].

To address these challenges, we propose a new automated network protocol syntax information extraction method, the main idea of the method is to extract detailed network protocol syntax information by parsing the protocol's Wireshark dissector file. The reason for proposing this approach is based on:

- Wireshark is an open-source protocol analysis tool that supports a wide variety of protocols. From mainstream Internet protocols and emerging Internet of Things (IoT) protocols in IT (Information Technology) fields to Industrial Network protocols in OT (Operation Technology) fields, Wireshark offers wide coverage across various communication domains.
- Wireshark plug-ins, in this paper, mainly refer to protocol dissector files written in C language under certain writing specifications. These files usually contain detailed parsing logic of related protocols, based on this parsing logic we can acquire abundant syntax information including field information, data format, and the possible values of a certain field, etc.

The contributions of this work are as follows:

- We propose a network protocol syntax extraction method based on Wireshark network protocol dissector file. The method includes single-layer protocol field information extraction, multi-layer protocol hierarchical relationship extraction, packet type acquisition based on basic path collection and field dependency judgment. It has high efficiency and can extract more comprehensive syntax information. Besides Byte-level field information, the method can extract Bit-level information for some fields.
- We designed a network protocol syntax extraction system based on the method. The system consists of a pre-processing module and a main parsing module. The system can extract the syntax information of a specified protocol together with its

associated protocols by only one process. Our code is open source at GitHub platform https://github.com/QingJiuYS/Syntax_extraction, accessed on 10 March 2024.

The rest of this paper is structured as follows: Section 2 introduces and provides insights into related work. Section 3 elucidates the core components of the proposed method. Section 4 presents a comprehensive exposition of the method's intricate design and systematic implementation. Section 5 presents the experimental results and comparisons with other methods, and Section 6 encapsulates the paper with a succinct conclusion.

2. Related Work

Snake_nlp [18] is an artificial intelligence extraction method based on protocol specifications. This method uses an NLP algorithm based on zero-shot learning to extract network protocol syntax information from RFC documents, with an accuracy of 0.82. Reference [24] is also an artificial intelligence extraction method based on protocol specifications. The author proposes a data-driven method for extracting finite state machines from protocol specification RFC documents. In paper [25], the author proposes a method of feature extraction and recognition using convolutional neural networks to extract syntax information of unknown protocols. This type of artificial intelligence method relies on a large amount of high-quality training data and is often designed for a specific format of the protocol specification document. For example, Snake_nlp [18] takes advantage of the feature in the RFC document that the protocol field is set as a title. This method has the problems of incomplete information extraction and insufficient accuracy. On the other hand, in MATT Security [26], the author monitors the MQTT protocol communication traffic and extracts syntax information such as message type, field structure, field position, and format of the protocol from the actual transmission data packets. Paper [27] also extracts field information and field boundary information in the data packet through real-time capture of data packets and a simple tag-based technology. This approach often relies on real-time capture of communication data, and for encrypted or compressed communication traffic, the accuracy and completeness of information extraction suffer. Furthermore, ProsegDL [28] is a software reverse engineering-based method that combines an image semantic segmentation model and a Siamese network to achieve format extraction within acceptable running time. In paper [29], the authors propose a new protocol reverse engineering method that uses the Continuous Sequence Pattern (CSP) algorithm to extract protocol specifications. NetPlier [30] is a protocol reverse engineering technique based on probabilistic network tracing that models the inherent uncertainty of the problem by introducing random variables to represent the likelihood of each field representing a packet type. In paper [31], the author conducts a comprehensive review of the research status and technological development of network protocol reverse engineering tools. Although this type of protocol reverse method can extract information such as protocol format and field boundaries, the inference of the protocol format is often not accurate enough.

In summary, existing research has greatly promoted the development of the field of network protocol syntax information extraction but there is still no universal method that can meet the requirements of network protocol-related testing, which needs quick and comprehensive protocol syntax information extraction, and can be adapted to most existing protocols. For this reason, we propose the protocol syntax information extraction method based on the Wireshark protocol dissector file, which can extract protocol syntax information quickly, comprehensively, and accurately, and is applicable to all Wireshark-supported protocols.

3. Methods

The overview of Wireshark's function blocks is shown in Figure 1. The Wireshark framework mainly consists of GUI, Core, Epan, Wiretap, Capture, and Dumpcap, wherein Epan (Enhanced Packet ANalyzer) is a packet analysis engine and the core of the framework, responsible for the key tasks of parsing packets and extracting protocol information. Epan provides a series of APIs, including Protocol Trees, Dissectors, and Dissector-Plugins.

Protocol Tree is used to store the parsing information of a single data packet. Dissectors are responsible for parsing data packets of a specific protocol. Dissector files are located in the epan/dissectors directory (<https://github.com/wireshark/wireshark/tree/master/>, accessed on 10 March 2024). Each dissector can parse data packets of a specific protocol and fill the parsing results into the protocol tree. Through Dissector-Plugins, Wireshark can flexibly extend its parsing capabilities, allowing users to add new protocol dissectors or improve existing dissectors to better adapt to the changing network communication environment.

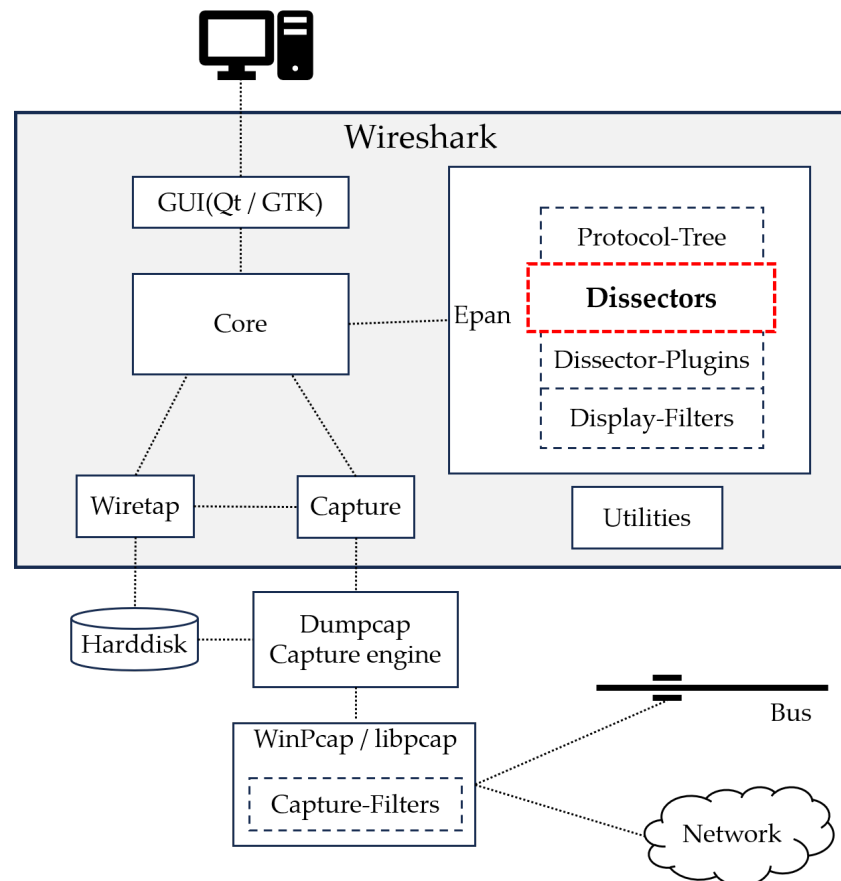


Figure 1. Overview of Wireshark's function blocks (https://www.wireshark.org/docs/wsdg_html_chunked/ChWorksOverview.html, accessed on 10 March 2024).

The dissector framework consists of three parts: “proto_register_PROTOABBREV”, “proto_reg_handoff_PROTOABBREV” and “dissect_PROTOABBREV”, as shown in Figure 2.

Wherein, in “proto_register_PROTOABBREV”, protocol names are registered through the “proto_register_protocol” function; the global dissector handle is created through the “new_create_dissector_handle” function and associated with the protocol name, the dissector is registered to the dissector table through the “register_dissector_table” function; all field information of the protocol is registered through the “proto_register_field_array” function and maintained in the structure array “hf_register_info”.

In “proto_reg_handoff_PROTOABBREV”, the local dissector handle is created through the “new_create_dissector_handle” function and associated with the protocol name; the dissector is registered to Wireshark through the “dissect_add_uint” function, and associate the protocol's port with the dissector handle.

In “dissect_PROTOABBREV”, this is where the protocol data packet is actually parsed. Field information is added to the protocol tree through the “proto_tree_add_item” function.

```

static int dissect_PROTOABBREV(tvbuff_t *tvb, packet_info *pinfo, proto_tree *tree, void *data _U_)
{
    .....
    expert_ti = proto_tree_add_item(PROTOABBREV_tree, hf_PROTOABBREV_FIELDABBREV, tvb, offset, len, ENC_x
    offset += len;
    .....
    data_handle = find_dissector("data");
    .....
}

void proto_register_PROTOABBREV(void)
{
    .....
    static hf_register_info hf[] = {
        { &hf_PROTOABBREV_FIELDABBREV,
          { "FIELDNAME", "PROTOABBREV.FIELDABBREV",
            FIELDTYPE, FIELDDISPLAY, FIELDCONVERT, BITMASK,
            "FIELDDDESCR", HFILL }
        }
    };
    .....
    proto_PROTOABBREV = proto_register_protocol("PROTONAME", "PROTOSHORTNAME", "PROTOABBREV");
    PROTOABBREV_handle = register_dissector("PROTONAME", dissect_PROTOABBREV, proto_PROTOABBREV);
    PROTOABBREV_dissector_table = register_dissector_table("PROTOABBREV.prot_id", "PROTOABBREV protocol id
    proto_register_field_array(proto_PROTOABBREV, hf, array_length(hf));
    .....
}

void proto_reg_handoff_PROTOABBREV(void)
{
    .....
    if (!initialized) {
        .....
        PROTOABBREV_handle = new_create_dissector_handle(dissect_PROTOABBREV, proto_PROTOABBREV);
        .....
        dissector_add_uint("tcp.port", currentPort, PROTOABBREV_handle);
    }
}

```

Figure 2. Wireshark protocol dissector framework. (<https://github.com/wireshark/wireshark/tree/master/>, accessed on 10 March 2024).

Through the above analysis, we propose a new protocol syntax information extraction method. The main idea of the method is to extract protocol syntax information by parsing the Wireshark protocol dissector file. It can be mainly divided into the following four extraction tasks:

- Single-layer protocol field information extraction.
- Multi-layer protocol hierarchical relationship extraction.
- Packet type extraction based on the basic path set.
- Field dependency extraction.

3.1. Single-Layer Protocol Field Information Extraction

Single-layer protocol field information extraction refers to extracting all fields' attribute information owned by a protocol from the protocol dissector file. Attributes information includes field name, offset, length, type, optional value, etc. There are three main ways to obtain fields' attributes information:

- From static arrays. Each protocol dissector file defines one or several static structure arrays called "header fields" in which the names, identifiers, data types, lengths, and optional values information for all fields are stored, we call these fields "standard fields". In the subsequent parsing process, the standard fields are used to judge the validity of the extracted per packet type fields and at the same time can give supplement information to the valid fields.
- From the buffer reading function. By analyzing the packet buffer read function, the variable name, offset, and length information of each field is extracted.
- From the protocol tree. By tracing the process of the protocol tree generating in the main parsing function, the identifier, offset, length, variable name, and byte order of each field is extracted.

Table 1 lists the attribute information included in protocol fields and the corresponding obtaining ways, wherein "Static Array" means that it can be extracted from the structure array "header fields", "Buffer Read" means that it can be obtained from the buffer reading function, and "Proto Tree" means that it can be extracted from the generation process of the protocol tree.

Table 1. The attribute information of the field, and where it can be extracted. “✓” means the attribute can be extracted.

Attribute	Static Array	Buffer Read	Proto Tree
name	✓	-	-
identifier	✓	-	✓
optional value	✓	-	-
data type	✓	✓	-
length	-	✓	✓
bit-length	-	✓	-
variable name	-	✓	-
endian	-	✓	✓
offset	-	✓	✓
dependencies	-	-	✓

3.2. Multi-Layer Protocol Hierarchical Relationship Extraction

Wireshark maintains multiple dissector tables dynamically to establish relationships between different protocol dissectors. In this process, lower-layer protocols first create a sub-table within a dissector table, and then upper-layer protocols add entries to this sub-table. For instance, during initialization, the TCP dissector creates a “TCP port” sub-table within the “Integer Tables” dissector table, and other upper-layer protocols subsequently populate this “TCP port” sub-table with corresponding entries. These entries are typically associated with specific TCP port numbers, so when the TCP dissector processes packets with special TCP port numbers, it searches and invokes the appropriate upper-layer protocol dissectors based on these port numbers within the “TCP port” sub-table. Additionally, lower-layer dissectors can explicitly specify upper-layer dissectors by invoking specific functions like “find_dissector()”. According to these two rules, the hierarchical relationship between different protocols can be deduced and extracted by analyzing all Wireshark protocol dissector files.

3.3. Packet Type Extraction Based on Basic Path Set

In this step, the program structure and control flow of the Wireshark protocol dissector file are analyzed to get the basic path set of the whole file, then the basic path set is used to extract the packet types of the protocol. In contrast to traditional methods that focus on functions, methods, or data structures as the basic units of analysis, we center around the entire file as the fundamental unit to parse the basic path set. Each basic path represents a unique execution path in the protocol parsing process, encompassing various protocol parsing scenarios and packet structures. Therefore, each path in the basic path set is regarded as a parsing path of one packet type, and all constituent fields of the packet type can be extracted from the parsing path, thus all packet types and their corresponding field information can be obtained by parsing the whole basic path set. During the extraction process, we track the generation of the protocol tree along the parsing paths, allowing for the accurate extraction of attributes for each field, including offset, length, variable name, and byte order. Furthermore, these field attributes are further refined by associating them with extracted standard fields. The extraction process is described in detail in Section 4.

3.4. Field Dependency Extraction

Dependencies between fields are important in building valid packets, which often means that the value of one field is affected by the value or attribute of another field. For example, in the Modbus TCP protocol, the value of the header field “Length” is determined by the actual length of the “Data” field. Wireshark protocol dissector files encompass a series of field parsing rules, including the definition of inter-field dependencies. During the parsing process, it is imperative to analyze the execution conditions and iteration counts of the program to ascertain these dependencies, facilitating the addition of appropriate attributes to pertinent fields.

4. Design and Implementation

Based on the method mentioned above, we implemented a protocol syntax information extraction system. The main function of this system is to automatically extract the syntax information of a specified protocol and its related protocols through one process and give the results in a structured format. The system consists of a pre-processing module and a main parsing module.

The pre-processing module is responsible for the preliminary processing of all Wireshark protocol dissector files, including extracting all protocol names and protocol dissector information, and generating protocol tables and parser tables for subsequent retrieval of target protocols and related dissectors. At the same time, the pre-processing module can extract the hierarchical relationship of multi-layer protocols and support the generation of multi-layer protocols for complete network data packets. The main-parsing module conducts an in-depth analysis of the protocol to be processed and extracts all packet types of that protocol, all field information that makes up each packet type, and the dependencies between fields. Figure 3 describes the overall architecture of the system, and each step in the figure is described in detail below.

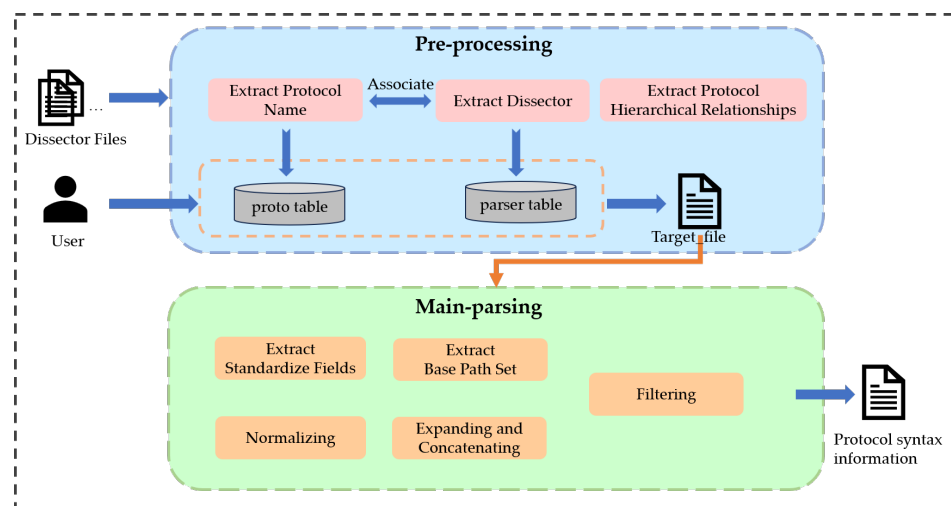


Figure 3. Overall system architecture.

4.1. Pre-Processing Module

4.1.1. Protocol Name and Dissector Extraction

The extraction system first investigates each Wireshark dissector file to locate the protocol registration function according to Wireshark's protocol registration mechanism and then extracts the protocol name and protocol handle from it. The protocol name, protocol handle, and associated protocol dissector file name are stored in a data structure called "protocol table". Subsequently, the extraction system locates all dissector registration functions and extracts all dissector names, each dissector's main parsing function and related protocol handle from the dissector file, and then the dissector information is associated with the related protocol according to the protocol handle. The dissector information and its associated protocol name are stored in a data structure called "parser table". The purpose of the association is that in some dissector files, there may be several protocol-dissector pairs, for example, in the "packet-mbtcp.c" dissector file, there are several dissectors including "mbtcp" and "mbrtu" that are associated with "Modbus TCP" and "Modbus RTU" protocol each. By establishing this association, each protocol and its corresponding dissector can be accurately found by accessing the protocol table.

4.1.2. Multi-Layer Protocol Hierarchical Relationship Extraction

A network communicating packet can be divided into seven layers according to the OSI (Open System Interconnection) model. To construct a complete packet that can be used

for network protocol fuzzing, the syntax information of all concatenated protocols on top of the transport layer (TCP, e.g.) is needed. This means that we need to accurately identify and extract all protocol layer relationships from the transport layer to the application layer. For example, the S7 Communication packet contains three protocols above TCP: TPKT, COTP, and S7Comm. Correctly identifying and extracting the multi-layer protocol hierarchical relationships can ensure the comprehensiveness of extracted protocol syntax information.

The Multi-layer protocol hierarchical relationship extraction process is executed as follows: Firstly, if a dissector “Da” creates a dissector table “Ta” by calling a protocol registration function, the protocol “Pa” that is related to dissector “Da” is considered a lower-layer protocol, afterward, when another dissector “Db” adds entries to dissector table “Ta”, the dissector Db’s associated protocol “Pb” is considered an upper-layer protocol of “Pa”. Secondly, if a dissector “Dc” explicitly calls an external dissector “Dd” by calling “find_dissector()” function, the called dissector “Dd” is the upper-level dissector of “Dc” and its corresponding protocol “Pd” is the upper-level protocol of “Pc”. In this way, multi-layer relationships of protocols are extracted and written into the parser table. Afterward, the system will list all supported protocols for the user to select, when the user selects a special protocol, all associated protocols are processed in sequence according to the hierarchical relationship stored in the parser table.

4.2. Main-Parsing Module

First, the system searches the parser table obtained in pre-processing stage to determine whether there is a multi-layer relationship in the current protocol. Based on the stored protocol relationships, the associated protocols’ names, each protocol’s associated dissector name and related dissector file name are acquired. Next, each protocol dissector file is analyzed in detail. The main parsing process is illustrated in Figure 4, which can be divided into three parts, wherein part (a) is extracting the standard fields of the protocol and locating the main parsing function, part (b) is normalizing the main parsing function, and part (c) is extracting the packet types and the constituent field information for each packet type from the normalized main parsing function.

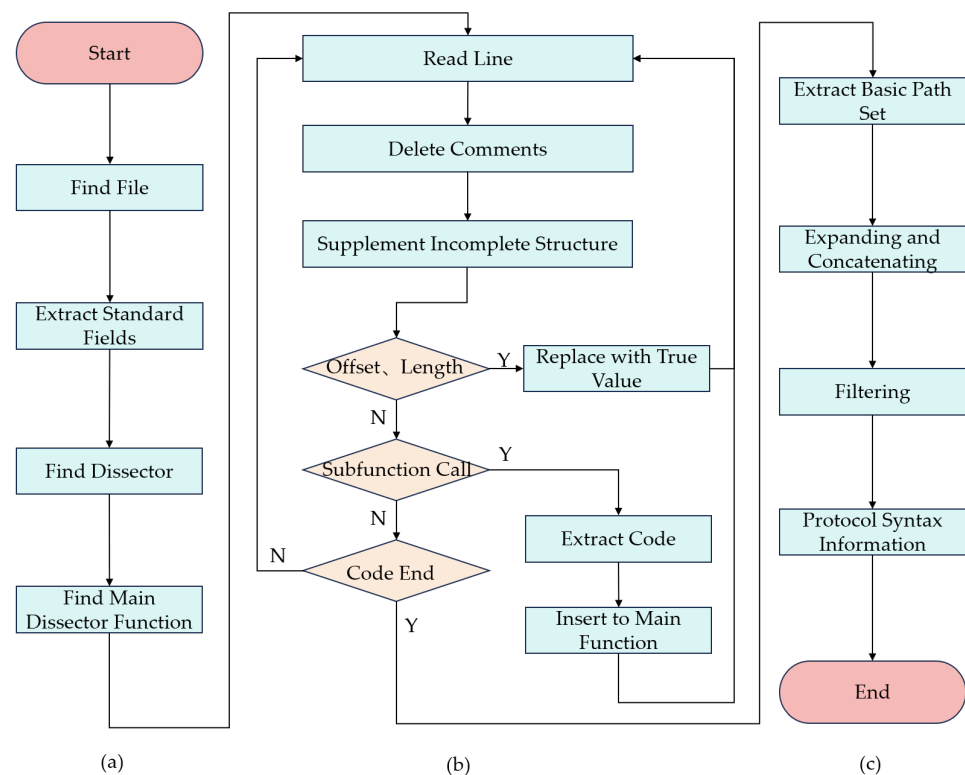


Figure 4. Flowchart of the main parsing module. Where (a–c) represent its three components.

4.2.1. Standard Fields Extraction

In this step, the parsing process will extract all fields owned by the target protocol, which are defined in the static data structure “hf_register_info” in the protocol registration function. We call the field information extracted in this step which contains the name, identifier, data type, length, and optional value information of the fields as standard fields. In the subsequent parsing process, standard fields are used to verify the extracted per packet type field information and supplement them.

4.2.2. Normalizing

The parsing process will normalize the code structure of each dissector’s main parsing function and save the results to a standardized file. The purpose of this step is to optimize the code structure so that all protocol dissector codes comply with the same standards, thereby avoiding errors caused by format issues during the parsing process. The normalizing rules are as follows:

- Delete all comments.
- Merge statements that are displayed on multiple lines.
- All “if/else if/else” statements must have a complete structure. For structures lacking else statements, add an empty else statement to it.
- All conditional and loop statements must be enclosed with a pair of curly braces.
- All sub-function call statements are replaced with the sub-function body.
- The offset and length-related variables are replaced with their real values.

4.2.3. Basic Path Set Extraction

The main purpose of this step is to get all the packet types of the target protocol. The idea is that we get the basic path set of the dissector code, and then treat each basic path as a parsing path for a packet type, thereby obtaining all data packet types of the protocol. In order to achieve this goal, we propose a new basic block division method, which takes the normalized file as a basic processing unit and then divides the file into four types of basic blocks and assigns a unique type value to each of them. By maintaining a stack structure to indicate the current under-processing basic block, and executing different numbers of “push” and “pop” operations according to different block type values, all the packet types and the corresponding field information of each packet type can be obtained by scanning the normalized protocol dissector file only once. The four basic block types are defined as follows:

- Ordinary basic block: A code block composed of consecutive statements that contain sequential execution. This block is also the place that actually saves field information. In particular, the “do...while”, “while”, and “for” loop structures are treated as ordinary basic blocks that only contain the “data” field. They are only traversed once during execution, and the iteration number of the loop is treated as the “data” field length, the loop-control-variable’s change value in each iteration is regarded as the data type of the “data” field.
- Branch basic block: A code block that contains a branch structure. These branch structures can be conditional structures, such as “if/else if/else” structures, or selection structures, such as “switch” structures. Typically, these structures are used to differentiate packet types and determine dependencies between fields. For example, for the switch structure, we extract its variable expressions as dependent fields and each constant expression as the value of the dependent field, and add this dependency relationship to the subsequently extracted fields, indicating that the subsequent fields are specific fields under the special values of the dependent fields.
- Branch item basic block: The code block of each branch in the branch structure, including “if” basic block, “else if” basic block, “else” basic block, “case” basic block, and “default” basic block. Each branch item basic block represents a packet type. Wherein, the “if” basic block is the default first basic block in the “if/else if/else”

branch basic block. In particular, we consider “function” basic blocks as branch item basic blocks.

- End basic block: The end basic block includes the return basic block which is added when encountering a “return” statement and the function end basic block which is added when there is a function end. These basic blocks identify the exit of the function and are used to guide the subsequent concatenating process.

Figure 5 is a piece of Modbus TCP dissector code, we use it to illustrate the basic block partitioning process. Lines “3–7” of the code form an ordinary basic block, lines “8–28” form a branch basic block, lines “8–25” and lines “26–28” form two branch item basic blocks, each of line “17” and line “29” is a function return basic block, line “30” is a function end basic block.

```

1  dissect_mbtcp_pdu_common(tvbuff_t *tvb, packet_info *pinfo, proto_tree *tree, int proto, range_t *ports)
2  {
3      transaction_id = tvb_get_ntohs(tvb, 0);
4      protocol_id = tvb_get_ntohs(tvb, 2);
5      len = tvb_get_ntohs(tvb, 4);
6      unit_id = tvb_get_guint8(tvb, 6);
7      function_code = tvb_get_guint8(tvb, 7) & 0x7F;
8      if (tvb_reported_length_remaining(tvb, offset) > 0)
9      {
10         call_dissector_with_data(modbus_handle, next_tvb, pinfo, tree, &modbus_data);
11         dissect_modbus(tvbuff_t * tvb, packet_info * pinfo, proto_tree * tree, void *data)
12         {
13             len = tvb_captured_length(tvb);
14             if (len == 0)
15             {
16                 return 0;
17             }
18             else
19             {
20                 function_code = tvb_get_guint8(tvb, offset) & 0x7F;
21                 proto_tree_add_item(modbus_tree, hf_modbus_functioncode, tvb, offset, 1, ENC_BIG_ENDIAN);
22             }
23         }
24     }
25     else
26     {
27         return tvb_captured_length(tvb);
28     }
29 }
30
```

Figure 5. Sample code for basic block partitioning.

The normalized parsing code is sequentially traversed, each basic block encountered is allocated a separate list structure to store the basic block type and the field information extracted from that basic block, and the list is pushed onto the stack. Pop the list out of the stack at the end of its traversal. If there are nested basic blocks, such as switch basic block and case basic block, the child basic block (case basic block) is stored as an element in the list structure of the parent basic block (switch basic block), in this way a hierarchy between basic blocks is established. The basic block types and corresponding stack operations are listed in Table 2, where “value” represents the type value of the basic block, and the values in “Push” and “Pop” columns represent the number of the corresponding operations that should be performed.

Subsequently, the parsing process will parse each line of a basic block, extract the field’s variable name, offset, data type, endian, and the bit-level information of length if it finds a buffer reading function, and maintains this information in a list. Next, the parsing process will extract the field’s identifier, offset, length, dependencies, variable name, and endian information by tracing the generation process of the protocol tree, and then complete the field’s variable name and bit-level information of length based on the matching of variable name with the fields extracted from the buffer reading function. Finally, the parsing process will complete the field’s name, data type, and optional value information by matching the field’s identifier with standard fields.

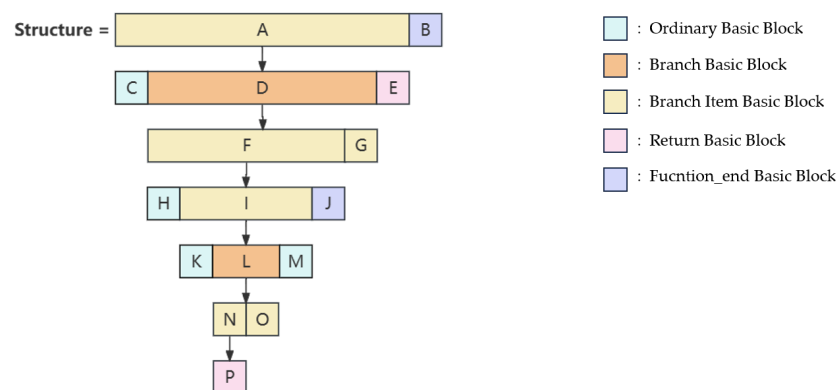
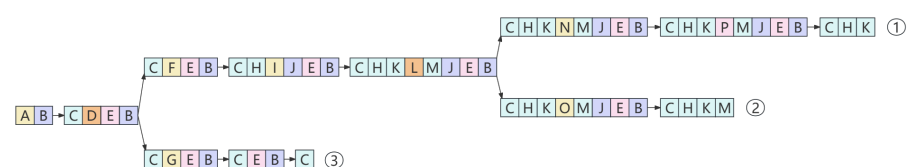
Table 2. Detailed division of basic blocks and corresponding stack operations.

Type	Value	Basic Block	Push	Pop
Ordinary Basic Block	0	ordinary	1	1
		for	1	1
		while/do while	1	1
Branch Basic Block	1	(if/else if/else)	2	-
		switch	1	1
Branch Item Basic Block	2	function	1	1
		if	-	1
		else if	1	1
		case	1	1
		default	1	1
End Basic Block	3	else	1	2
	4	return	-	-
	5	function_end	-	-

4.2.4. Expanding and Concatenating

The basic path set of the protocol dissector file obtained in the previous step is a multi-level nested list structure containing all basic blocks. Take Figure 6 sample code's basic path set as an example, as shown in Figure 7, each letter represents a basic block, and the colors identify different types of basic blocks. Arrows indicate that a basic block contains sub-basic blocks, and the numbers indicate the three basic paths obtained. For example, basic block A contains sub-basic blocks C, D, and E. The multi-level nested structure is then expanded to generate each basic path that will eventually generate each packet type. Each basic block has the following three expansion methods according to its type:

- Branch basic block: Select each of its sub-basic blocks to concatenate with the previous and next basic blocks.
- End Basic Block: Concatenate the basic block itself with the preceding and following basic blocks.
- Other basic blocks: Concatenate all sub-basic blocks with the basic blocks before and after them in sequence.

**Figure 6.** Basic path set structure.**Figure 7.** Expanding and concatenating of basic blocks.

For each expanded basic path, delete all basic blocks between the return basic block and the first function-ending basic block after it, this is because return represents an exit of the function and may be located at some place other than the end of the function body, then the field information acquired between this return statement and the end of the function body is invalid information for the basic path that contains this return statement, so these invalid information should be deleted. This process can be seen in the first path in Figure 7, basic blocks between “P” and “J” (P, M, J), and basic blocks between “E” and “B” (E, B) are deleted.

Repeating this process until each basic path contains only ordinary basic blocks. Each ordinary basic block stores the component field information of the current packet type. The elements of all ordinary basic blocks in each basic path are merged and only the field information is retained, and finally, the complete packet type is obtained.

4.2.5. Filtering

The final step is to filter the packet types to eliminate duplicate packet types and invalid packet types. Duplication of packet types may occur because the same path may occur in the branch structure. For example, in the “if/else if/else” structure, if no field information is extracted under the “if”, “else if” or “else” branch, that is, the branch does not generate a new packet type, then it will be expanding into three basic paths which will result in basic path duplication. In addition, the validity of the packet types is judged based on the dependencies of the extracted fields. If the dependencies of a certain field in the packet type are inconsistent or the dependencies between multiple fields are inconsistent, it means that the packet type is the wrong packet type and should be filtered.

5. Experiment

In this section, we use the designed protocol syntax information extraction system to conduct syntax information extraction experiments on the Modbus TCP, Modbus RTU, DCCP, DNP3.0, and S7COMM protocols, and analyze the experimental results. Through various comparisons with manual methods, traffic-based methods, and artificial intelligence methods, the advantages of the system in extracting protocol syntax information are proven.

5.1. Evaluate

In order to verify the effectiveness of syntax information extraction, we used the accuracy and coverage rate as evaluation metrics, wherein the accuracy rate represents the proportion of syntax information that is correctly extracted to all syntax information extracted, the coverage rate represents the proportion of syntax information that is correctly extracted to all syntax information contained in the protocol. Including the accuracy of data packet type, the coverage of data packet type, the accuracy of total fields, and the coverage of total fields. Finally, we calculated the average of the accuracy and coverage of all protocols.

In order to compare the comprehensiveness of the extracted syntax information, we use field information, packet types, fields per packet type, and multi-layer protocol support as evaluation metrics.

In order to compare various methods as a whole, we use accuracy, processing time per protocol, applicability, and comprehensiveness as evaluation metrics.

5.2. Results

Table 3 shows the actual number of data packet types owned by Modbus TCP, Modbus RTU, DCCP, DNP3.0, and S7COMM protocols, the number of data packet types extracted using our method, the correct extracted data packet types number, and the calculated accuracy and coverage information. Although the accuracy of the DCCP protocol and S7COMM protocol is not high, the coverage is relatively high, which means that we can effectively extract most protocol packet types.

Table 3. Packet type extraction results.

Protocol	Actual	Extracted	Correct	Accuracy	Coverage
Modbus TCP	35	40	34	0.850	0.971
Modbus RTU	35	40	34	0.850	0.971
DCCP	12	22	11	0.500	0.917
DNP3.0	27	28	22	0.786	0.815
S7COMM	13	21	11	0.514	0.846
COTP	11	14	11	0.786	1.0
TPKT	1	1	1	1.0	1.0
Average				0.680	0.843

Table 4 shows the total number of fields owned by the protocols, the number of fields extracted using our method, the number of correctly extracted fields, and the corresponding accuracy and coverage information. In most cases, our method is able to extract most fields and maintain high accuracy.

Table 4. Total fields extraction results.

Protocol	Actual	Extracted	Correct	Accuracy	Coverage
Modbus TCP	28	26	25	0.962	0.893
Modbus RTU	26	24	23	0.958	0.885
DCCP	20	21	20	0.952	1.0
DNP3.0	17	16	15	0.938	0.882
S7COMM	19	18	16	0.889	0.842
COTP	16	7	16	0.941	1.0
TPKT	3	3	3	1.0	1.0
Average				0.944	0.922

Figure 8 is a part of syntax information output results which is acquired by using our method over the Modbus TCP protocol. The structure in line “2” represents a data packet type, which contains all the component fields of the data packet type “READ_COILS” and detailed attribute information of each field.

```

1  [
2  [
3      {
4          "paket_type": "READ_COILS"
5      },
6      {
7          "field_name": "Transaction Identifier",
8          "field_offset": 0,
9          "field_length": 2,
10         "field_type": "FT_UINT16",
11         "field_endian": "ENC_BIG_ENDIAN"
12     },
13     {
14         "field_name": "Protocol Identifier",
15         "field_offset": 2,
16         "field_length": 2,
17         "field_type": "FT_UINT16",
18         "field_endian": "ENC_BIG_ENDIAN"
19     },
20 > { ...
26 ]

```

Figure 8. Partial output results of Modbus TCP protocol.

Comparison and Discussion

Table 5 is the entire comparison results of our method, the manual method, the traffic-based method [15], and the NLP-based artificial intelligence method [18].

Table 5. Entire comparison of different methods.

	Accuracy	Processing Time	Applicability	Comprehensiveness
Manual	high	days	One Target Protocol	packet types, fields
Traffic-based	medium	minutes	Most Network Protocols	fields, field boundaries
NLP-based	0.82	minutes	RFC Protocols	fields, packet format
Our Method	0.94	seconds	RFC, IoT, ICS Protocols	packet types, fields, field dependencies

It can be seen from the results that our method not only maintains both the high-accuracy feature of the manual method and the high-automation feature of the artificial intelligence method, but also has better versatility and can extract more comprehensive protocol syntax information, wherein, in terms of accuracy, our method can extract the field information of the protocol with an average accuracy higher than 0.9, which is much higher than the 0.82 of the NLP-based method. In terms of processing time per protocol, our method can process in a shorter time, usually a few seconds, with very low resource consumption. In terms of applicability, our method can cover a wider range of protocols, including RFC protocols, IoT protocols, and industrial control protocols, and does not rely on real traffic.

In terms of syntax information comprehensiveness and accuracy, we compared our method with the most similar NLP-based method on the DCCP protocol, the results are shown in Table 6 and Figure 9. The Table 6 results show that our method can acquire more protocol syntax information, including packet types of the protocol and the component field information of each packet type. The information of each field includes the field name, offset, length, bit-level length information, data type, optional values, endian, and dependencies between fields. Moreover, our approach also provides support for multi-layer protocols. As shown in Figure 9a, although our method is slightly lower in the accuracy metric than the NLP-based method, it can acquire more comprehensive packet type information which not only includes the packet type names, but also the component fields of each packet type. As shown in Figure 9b, the total extracted fields accuracy and coverage of our method are all better than the NLP-based method.

Table 6. Extracted syntax information comprehensiveness comparison. Wherein, “✓” means it can be extracted, “×” means it cannot be extracted.

Syntax Information		NLP-Based	Our Method
Field Information	Name	✓	✓
	Length	Byte	Byte, Bit
	Offset	✓	✓
	Endian	×	✓
	Dependencies	×	✓
Packet Information	Type	✓	✓
	Fields per Packet Type	×	✓
Multi-layer Protocol Support		×	✓

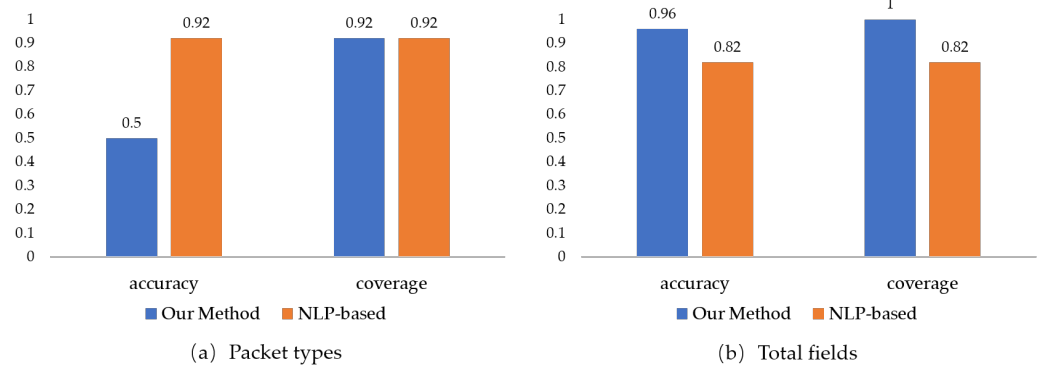


Figure 9. Comparison of extraction accuracy and coverage on DCCP protocol.

6. Conclusions and Future Work

In this work, we propose a novel method for extracting network protocol syntax information. The method can extract all packet types of the target protocol and all field information for each packet type by automatically analyzing the Wireshark protocol dissector file. We designed a protocol syntax information extraction system based on this method and applied it to the Modbus TCP, Modbus RTU, DCCP, DNP3.0, and S7COMM protocols. Experimental results show that our method can effectively extract most packet types of the protocol, the constituent fields of each packet type, and the dependencies between the fields. Using the extracted syntax information for network protocol fuzzing can increase the effectiveness of the test-case generating process, and improve the efficiency and path coverage depth of fuzzing. By comparing with manual methods, traffic-based methods, and artificial intelligence methods, our method has a better performance in terms of automation, process time per protocol, applicability, and cost; at the same time, the extracted protocol syntax information has more comprehensiveness.

Although our method has made some progress in some aspects, there are still some problems that need to be further solved. One of them is how to accurately determine the fixed value of a specific field, such as the “Protocol Identity” field in the Modbus TCP protocol, and the “Start Bytes” field in the DNP3.0 protocol, etc. Another challenge is determining the range of data associated with the checksum field. Additionally, we need to find a way to obtain the status information of the protocol. These issues are important directions for our next research. By solving these issues, we can further improve the accuracy and practicality of our method.

Author Contributions: Conceptualization, H.L. and D.Z.; methodology, H.L. and L.Z.; software, L.X.; validation, X.L., S.Y. and X.H.; formal analysis, L.X.; investigation, H.L.; resources, L.Z. and D.Z.; data curation, H.L.; writing—original draft preparation, H.L.; writing—review and editing, L.Z.; visualization, S.Y. and X.H.; supervision, L.X.; project administration, X.L.; funding acquisition, D.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported in part by the National Key R&D Program of China (2023YFB3107305), in part by the Young Innovation Team of Colleges and Universities in Shandong Province (2021JK001), in part by the National Natural Science Foundation of China (62172244), in part by the Natural Science Foundation of Shandong Province (ZR2020YQ06 and ZR2021MF132), in part by the Innovation Ability Promotion Project for Small and Medium-sized Technology-based Enterprise of Shandong Province (2022TSGC2098, 2023TSGC0150), in part by Talent Research Project of Qilu University of Technology (Shandong Academy of Sciences) (2023RCKY145), in part by the Pilot Project for Integrated Innovation of Science, Education and Industry of Qilu University of Technology (Shandong Academy of Sciences) (2022JBZ01-01), in part by the Taishan Scholars Program (tsqn202211210), in part by the Key Research Project of Quancheng Laboratory (QCLZD202303), in part by the “20 New Universities” Project of Jinan City (202333023, 202333045).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The original contributions presented in the study are included in the article, further inquiries can be directed to the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Olsthoorn, M.; van Deursen, A.; Panichella, A. Generating highly-structured input data by combining search-based testing and grammar-based fuzzing. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, Melbourne, Australia, 21–25 September 2020; pp. 1224–1228.
2. Wondracek, G.; Comparetti, P.M.; Kruegel, C.; Kirda, E.; Anna, S.S.S. Automatic Network Protocol Analysis. In Proceedings of the NDSS, Citeseer, San Diego, CA, USA, 10–13 February 2008; Volume 8, pp. 1–14.
3. Zhu, X.; Wen, S.; Camtepe, S.; Xiang, Y. Fuzzing: A survey for roadmap. *ACM Comput. Surv. CSUR* **2022**, *54*, 1–36. [\[CrossRef\]](#)
4. She, D.; Shah, A.; Jana, S. Effective seed scheduling for fuzzing with graph centrality analysis. In Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP), IEEE, San Francisco, CA, USA, 23–25 May 2022; pp. 2194–2211.
5. Godefroid, P.; Kiezun, A.; Levin, M.Y. Grammar-based whitebox fuzzing. In Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, Tucson, AZ, USA, 7–13 June 2008; pp. 206–215.
6. Sargsyan, S.; Kurmangaleev, S.; Mehrabyan, M.; Mishechkin, M.; Ghukasyan, T.; Asryan, S. Grammar-based fuzzing. In Proceedings of the 2018 Ivannikov Memorial Workshop (IVMEM), IEEE, Yerevan, Armenia, 3–4 May 2018; pp. 32–35.
7. Xiao, M.M.; Yu, S.Z.; Wang, Y. Automatic network protocol automaton extraction. In Proceedings of the 2009 Third International Conference on Network and System Security, IEEE, Queensland, Australia, 19–21 October 2009; pp. 336–343.
8. Al Salem, H.; Song, J. A review on grammar-based fuzzing techniques. *Int. J. Comput. Sci. Secur.* **2019**, *13*, 114–123.
9. Caballero, J.; Yin, H.; Liang, Z.; Song, D. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In Proceedings of the 14th ACM Conference on Computer and Communications Security, Alexandria, VA, USA, 28–31 October 2007; pp. 317–329.
10. Gorbunov, S.; Rosenbloom, A. Autofuzz: Automated network protocol fuzzing framework. *Ijcsns* **2010**, *10*, 239.
11. Höschel, M.; Zeller, A. Mining input grammars from dynamic taints. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, Singapore, 3–6 September 2016; pp. 720–725.
12. Pan, F.; Wu, L.F.; Hong, Z.; Li, H.B.; Lai, H.G.; Zheng, C.H. Icefex: Protocol Format Extraction from IL-based Concolic Execution. *KSII Trans. Internet Inf. Syst.* **2013**, *7*, 576–599.
13. Blumbergs, B.; Vaarandi, R. Bbuzz: A bit-aware fuzzing framework for network protocol systematic reverse engineering and analysis. In Proceedings of the MILCOM 2017—2017 IEEE Military Communications Conference (MILCOM), IEEE, Baltimore, MD, USA, 23–25 October 2017; pp. 707–712.
14. Caballero, J.; Song, D. Automatic protocol reverse-engineering: Message format extraction and field semantics inference. *Comput. Netw.* **2013**, *57*, 451–474. [\[CrossRef\]](#)
15. Bytes, A.; Rajput, P.H.N.; Domanidis, C.; Maniatakis, M.; Zhou, J.; Tippenhauer, N.O. FieldFuzz: In Situ Blackbox Fuzzing of Proprietary Industrial Automation Runtimes via the Network. In Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses, Hong Kong, China, 16–18 October 2023; pp. 499–512.
16. Hu, Z.; Shi, J.; Huang, Y.; Xiong, J.; Bu, X. GANFuzz: A GAN-based industrial network protocol fuzzing framework. In Proceedings of the 15th ACM International Conference on Computing Frontiers, Ischia, Italy, 8–10 May 2018; pp. 138–145.
17. Han, X.; Wen, Q.; Zhang, Z. A mutation-based fuzz testing approach for network protocol vulnerability detection. In Proceedings of the 2012 2nd International Conference on Computer Science and Network Technology, IEEE, Changchun, China, 29–30 December 2012; pp. 1018–1022.
18. Jero, S.; Pacheco, M.L.; Goldwasser, D.; Nita-Rotaru, C. Leveraging textual specifications for grammar-based fuzzing of network protocols. In Proceedings of the AAAI Conference on Artificial Intelligence, Honolulu, HI, USA, 27 January–1 February 2019; Volume 33, pp. 9478–9483.
19. Rontti, T.; Juuso, A.M.; Takanen, A. Preventing DoS attacks in NGN networks with proactive specification-based fuzzing. *IEEE Commun. Mag.* **2012**, *50*, 164–170. [\[CrossRef\]](#)
20. Deng, Y.; Xia, C.S.; Peng, H.; Yang, C.; Zhang, L. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, Seattle, WA, USA, 17–21 July 2023; pp. 423–435.
21. Pradhan, S.; Ray, M.; Swain, S.K. Transition coverage based test case generation from state chart diagram. *J. King Saud Univ.-Comput. Inf. Sci.* **2022**, *34*, 993–1002. [\[CrossRef\]](#)
22. Ba, J.; Böhme, M.; Mirzamomen, Z.; Roychoudhury, A. Stateful greybox fuzzing. In Proceedings of the 31st USENIX Security Symposium (USENIX Security 22), Boston, MA, USA, 10–12 August 2022; pp. 3255–3272.
23. Sargsyan, S.; Hakobyan, J.; Mehrabyan, M.; Mkoyan, R.; Sahakyan, V.; Melkonyan, V.; Arutunian, M.; Fahradyan, A.; Avetisyan, A. Advanced Grammar-Based Fuzzing. In Proceedings of the 2022 Ivannikov Memorial Workshop (IVMEM), IEEE, Kazan, Russia, 23–24 September 2022; pp. 61–64.

24. Pacheco, M.L.; von Hippel, M.; Weintraub, B.; Goldwasser, D.; Nita-Rotaru, C. Automated attack synthesis by extracting finite state machines from protocol specification documents. In Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP), IEEE, San Francisco, CA, USA, 22–26 May 2022; pp. 51–68.
25. Wang, Y.; Bai, B.; Hei, X.; Zhu, L.; Ji, W. An unknown protocol syntax analysis method based on convolutional neural network. *Trans. Emerg. Telecommun. Technol.* **2021**, *32*, e3922. [[CrossRef](#)]
26. Hernández Ramos, S.; Villalba, M.T.; Lacuesta, R. Mqtt security: A novel fuzzing approach. *Wirel. Commun. Mob. Comput.* **2018**, *2018*, 8261746. [[CrossRef](#)]
27. Shapiro, R.; Bratus, S.; Rogers, E.; Smith, S. Identifying vulnerabilities in SCADA systems via fuzz-testing. In Proceedings of the Critical Infrastructure Protection V: 5th IFIP WG 11.10 International Conference on Critical Infrastructure Protection, ICCIP 2011, Hanover, NH, USA, 23–25 March 2011; Springer: Berlin/Heidelberg, Germany, 2011; pp. 57–72.
28. Zhao, S.; Wang, J.; Yang, S.; Zeng, Y.; Zhao, Z.; Zhu, H.; Sun, L. ProsegDL: Binary Protocol Format Extraction by Deep Learning-based Field Boundary Identification. In Proceedings of the 2022 IEEE 30th International Conference on Network Protocols (ICNP), IEEE, Lexington, KY, USA, 30 October–2 November 2022; pp. 1–12.
29. Goo, Y.H.; Shim, K.S.; Lee, M.S.; Kim, M.S. Protocol specification extraction based on contiguous sequential pattern algorithm. *IEEE Access* **2019**, *7*, 36057–36074. [[CrossRef](#)]
30. Ye, Y.; Zhang, Z.; Wang, F.; Zhang, X.; Xu, D. NetPlier: Probabilistic Network Protocol Reverse Engineering from Message Traces. In Proceedings of the NDSS, Online, 21–25 February 2021.
31. Duchêne, J.; Le Guernic, C.; Alata, E.; Nicomette, V.; Kaâniche, M. State of the art of network protocol reverse engineering tools. *J. Comput. Virol. Hacking Tech.* **2018**, *14*, 53–68. [[CrossRef](#)]

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.