École Polytechnique Fédérale de Lausanne

Klark - Mining specification from Blob Dissectors

by Kamila Babayeva and Marc Egli

Master Project Report

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Project Advisor

Ahmad Hazimeh
Project Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

June 10, 2022

Abstract

In this report we present Klark - a first step to automate the extraction of fuzzing specifications for network protocols. We use Klee to symbolically execute the Wireshark dissector of the protocol we want to generate the specifications for. With the resulting constraints of every path, we compute sample packets and analyze them with Wireshark to verify their correctness. Additionally, we extract information about individual fields in each packet type. The fields and the constraints can be used to generate full fuzzing specifications for network protocols in the future.

Chapter 1

Introduction

Nowadays it is hard to imagine the world without the Internet. The amount of users and devices joining the Internet grows steadily. According to Statista [10], in April 2022 the number of Internet users reached 5 billion, which is 63 percent of the world population. Even though the Internet is extensively used, the fundamentals of the Internet - network protocols - are created by humans who are naturally prone to make errors. It is essential to verify these network protocols for correctness because errors in their design and implementation can lead to devastating security vulnerabilities [1]. One such example can be the Heartbleed [14] vulnerability in OpenSSL that allowed an attacker to read the memory of the affected system over the Internet. Another example can be an integer overflow in OpenSSH [5] that allowed an adversary to remotely execute arbitrary code on affected servers.

Checking the network protocols for correctness is a challenging task: 1) the amount of protocols is extremely huge, 2) the design and thus the implementation is complex, 3) the protocol state-space is large, and 4) protocol specifications are mostly human-readable only, and rarely machine-readable. Considering these challenges, the manual analysis and testing would be exhausting. Therefore, developers tend to use state-of-the-art techniques such as fuzzing, symbolic execution, and static code analysis to detect security vulnerabilities in network protocols. Fuzzing is a popular technique to test the protocol implementation due to its straightforwardness. AFLNet [8], a coverage-guided grey-box fuzzer, uses real packets as a seed for future packet mutation and server response code as feedback. However, there is no guarantee of the complete

exploration of the protocol since some server response codes can stay unexplored. Moreover, AFLNet tests only the server side of the protocol. Compared to AFLNet, StateFuzzer grey-box fuzzer [7] can fuzz both client-side and server-side thus increasing the exploration coverage. A format-specific fuzzer FormatFuzzer [2] uses binary templates (a format specification used by the 010 Editor [13]) to generate valid inputs for fuzzing. With the provided specification, the results showed that FormatFuzzer increases coverage in the program under test. However, FormatFuzzer cannot be used for protocol fuzzing since the number of available 010 Editor binary templates for protocols is small. There were also attempts to analyze the protocols with symbolic execution. Symbolic execution of protocol binary software [15] produces successful results in finding vulnerabilities. However, the input packet structure was manually symbolized based on chosen packet type from Request for comments [4] (RFC).

The main limitation of the approaches listed above is the lack of network protocol specifications. If complete protocol specifications are available, the fuzzing would be more precise and the exploration coverage would be more complete.

This semester project aims to overcome this limitation by automatic extraction of fuzzing protocol specifications. We present a small tool called Klark[3] that uses Klee symbolic execution engine on the source code of Wireshark dissectors to extract protocol specifications. With Klark we were able to extract protocol specifications and generate different packet types for such protocols as ARP, LPD, VTP and FCCT. For the validity of our results, we generated real packet captures with resultant different packets for ARP and LPD protocols. These captures were successfully dissected by Wireshark. Even though we achieved successful results in extracting valid protocol specifications, there are challenges we encountered during the process: path explosion, mocking Wireshark API calls, extraction of detailed packet fields data, partial analysis of ternary statement inside Klee and others.

This report is built as follows. Chapter 2 explains the choice of Wireshark dissectors and Klee symbolic engine to automatically extract the protocol specifications. Chapter 3 provides a background for Wireshark and Klee to understand the modification needed for Klark. Chapter 4 describes the challenges and the design of Klark. Chapter 5 evaluates the results. Chapter 6 presents encountered limitations and discusses the future work. Chapter 7 concludes the project.

This semester project was done by Kamila Babayeva and Marc Egli. Kamila has started the research by analyzing the previous work in the extraction of protocol specifications, searching for existing protocol implementation and symbolic execution engines. Marc joined the project with appropriate Klee modifications to symbolically execute Wireshark dissectors. Later, Marc did the core part of Klark by mocking Wireshark dissectors to make them eligible for symbolic execution. Kamila finished Klee modifications to extract protocol fields information. Then both worked on solving resultant constraints to generate valid packets. Kamila generated sample captures to validate the results.

Chapter 2

Approach

The goal of this semester project is to automate fuzzing specifications for network protocols. This chapter reasons about the choice of the tools for our solution. We discuss why Wireshark dissectors have been chosen to extract protocol specifications from and why Klee was chosen as a symbolic execution engine to analyze Wireshark dissectors.

2.1 Protocol Specifications

Several sources exist where the protocol specifications can be extracted from 1) Request for Comments (RFC), 2) binary templates with protocol specifications, 3) dissectors source code from network analysis tools. There are pros and cons to each idea.

RFC. The first idea of a collection of protocol specifications would be official Internet Standards documented by RFC. The proposed design of protocols is described in RFC in the form of text and graphs. One way to extract specifications would be manual processing, but this approach is tedious and error-prone. Another option would be to process the text automatically. There was an attempt [6] to use Natural language processing to extract finite state machines from RFC, but the results are not precise.

Binary templates. Following successful results of fuzzing with FormatFuzzer [2] based on binary templates specifications, another idea would be to use FormatFuzzer but with protocol

binary templates in 010 Editors format. But there are no binary templates for network protocols available online.

Dissectors. Last but not least, such traffic analysis tools as Wireshark[16], Zeek[18] and Suricata[12] have their own protocol parsers - dissectors - implemented inside. These dissectors contain information about the protocol fields, the correct way to parse protocols and the dependencies between them. The dissectors also keep track of sessions in stateful protocols.

As we can see, the most feasible choice out of all is to use dissectors implemented inside the network analyzer. Dissectors are available for most of the protocols and are already in a machine code. Zeek, Suricata, and Wireshark are all open-source tools, i.e. dissector's source code is publicly accessible. For our research, we chose Wireshark dissectors among all network traffic analyzers: 1) it dissects on the packet level (compared to Zeek), 2) the dissection is straightforward and source code is written in C, and 3) it is a most commonly used tool.

2.2 Symbolic Execution Engine

After choosing Wireshark dissectors as a source to extract specifications from, we have to analyze their source code written in C language and generate all the possible packet types with appropriate constraints on each packet field. The best technique that suits our goal is symbolic execution. The idea is to represent the input packet data as a symbolic value and run symbolic execution on a dissector to constrain our symbolic input data.

To perform the required symbolic execution, we chose Klee Symbolic Execution Engine based on the following reasons. First Klee analyzes LLVM bitcode. Wireshark dissectors are written and C and can be compiled to LLVM bitcode. Second Klee is an open-source thus allowing us to modify its source code as it was needed in our project. Finally, Klee has proved its effectiveness in other research papers and it has a stable developers' support compared to other tools.

Chapter 3

Background

This chapter lays the knowledge foundation for Wireshark and Klee necessary to understand the design and the evaluation of Klark. In the first part, we present Wireshark and its dissection internals, followed by an overview of the dissection example in Wireshark's official dissector tutorial. In the second part, we provide an overview of Klee and briefly explain how it operates.

3.1 Wireshark : Network Packet Analyzer

3.1.1 Overview

Wireshark is a network packet analyzer that operates on emitted and received packets on a specific network interface. A packet, at its most simple representation, is a stream of bytes carried over the network. Wireshark analyzes this packet, internally dissects it for the protocols used in each network layer to send this packet, and provides visualization of the packet dissection for the user in its graphical interface. 3.1 is an example how a packet is dissected by Wireshark and the information presented in the graphical interface.

Internal Wireshark dissection. Wireshark dissection follows the layering in the current network architecture. The dissection starts in the Physical Layer and continues up to the Application Layer. A complete dissection goes as follows. When the packet enters Wireshark, the dissection always starts from the Frame dissector where basic packet info such as packet time and the

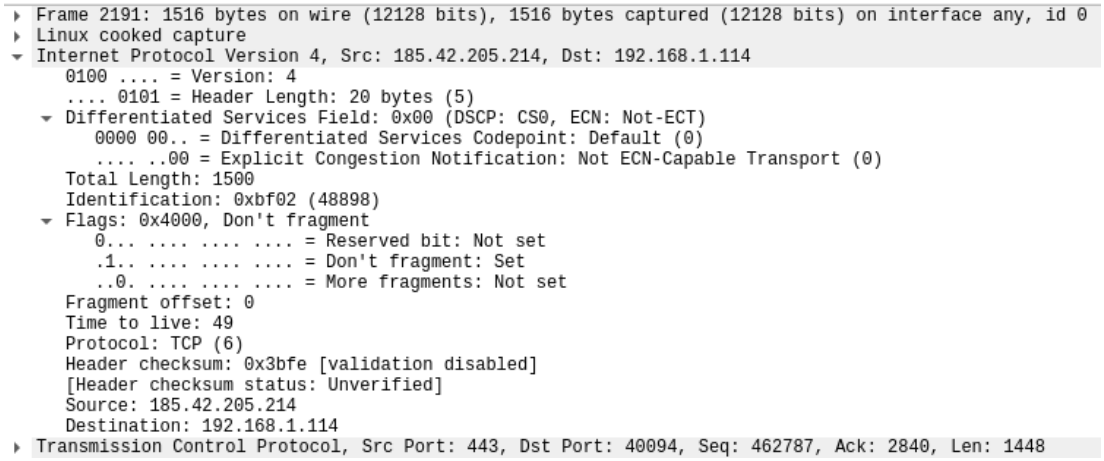


Figure 3.1: Wireshark dissection visualisation for a packet sent over TCP.

number is recorded. Then Wireshark detects the next protocol encapsulated in the packet. If it exists, the remaining bytes from the previous dissector will be passed to the next dissector. Such dissection cycle continues until the Application Layer is reached and all bytes of the packet have been consumed. During those multiple dissections, all information about dissection and protocol fields is collected and stored in a tree structure. If Wireshark struggles to detect the next dissector, Wireshark sends the remaining bytes to the Data dissector.

Alongside the dissection chain, Wireshark keeps track of field information from lower layers that then can be used to make decisions in dissectors operating at a higher layer (e.g using the destination port to distinguish between a request or a response packet). Wireshark also keeps track of context between packets to handle fragmentation correctly.

Wireshark visualisation dissection From a visualisation point of view, a packet in Wireshark is a tree root node that contains different sub-trees. A new sub-tree is created for each protocol (sample schema in Figure 3.2). There can also be sub-tress inside each protocol based on protocol specifications. As we can see in Figure 3.1 IP dissector has a sub-tree for the `Flags` category.

3.1.2 Simple Dissector example

To understand in more detail how the dissection works, we followed Wireshark's tutorial to create a Foo dissector that dissects a packet in the Application Layer. According to our specifications,

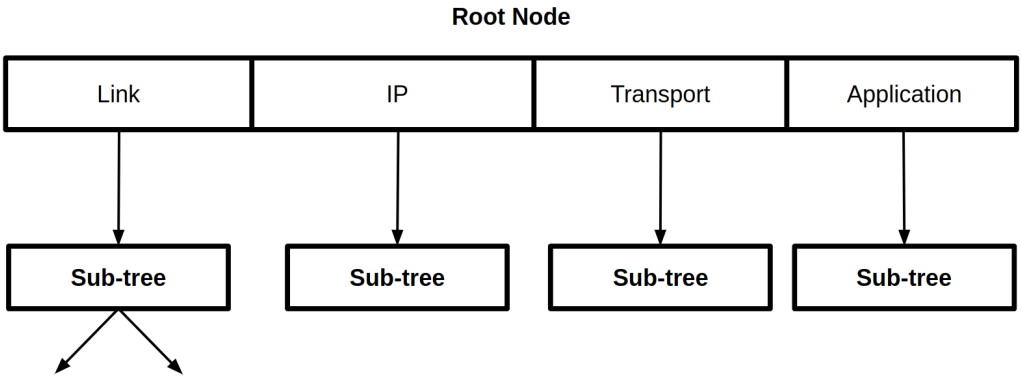


Figure 3.2: Tree packet dissection.

this protocol runs over UDP on port 1234. The code for the dissector is available **here**.

There are several important blocks one dissector can be divided into 1) register protocol handle, 2) record protocol fields, 3) dissect packet data according to protocol specifications. We will introduce the examples of each block based on the Foo dissector.

Block register protocol handle. A dissector has to register itself in Wireshark's internal system such that the lower layer dissectors can find the higher layer dissectors and delegate the remaining packet data for dissection. Figure 3.3 shows an example of the Foo dissector registering. Since the Foo dissector is located on the Application Layer, Wireshark internal system detects it on the Transport Layer protocol (in our case UDP) by Foo port number 1234. If detected, Wireshark calls the Foo dissector on the remaining data.

```
1 void
2 proto_reg_handoff_foo(void)
3 {
4     static dissector_handle_t foo_handle;
5     foo_handle = create_dissector_handle(dissect_foo, proto_foo);
6     dissector_add_uint("udp.port", FOO_PORT, foo_handle);
7 }
```

Figure 3.3: Foo protocol registration in Wireshark internal system.

```
...
static int hf_foo_pdu_type = -1;
static int hf_foo_flags = -1;
...

void
proto_register_foo(void) {

    static hf_register_info hf[] = {

    {&hf_foo_pdu_type,
        {"FOO PDU Type", "foo.type",
        FT_UINT8, BASE_DEC,
        NULL, 0x0,
        NULL, HFILL}},

    {&hf_foo_flags,
        {"FOO PDU Flags", "foo.flags",
        FT_UINT8, BASE_HEX,
        NULL, 0x0,
        NULL, HFILL}},
    ...
    };
}
```

Figure 3.4: Field Registration

Block record protocol fields. Every protocol dissector has to register the fields it will extract from the raw packet data. All the fields are recorded in an array `static hf_register_info hf[]`. Each array element holds information about the field: Name, Type, Size and a Bit Mask that is used to reference only a subset of bits in a byte. For simplification, the dissector references to the fields by their indexes in the array. Those indexes are registered in the beginning, and then set during the dissector run. An example of packet fields in the Foo dissector can be shown in Figure 3.4.

Block dissect packet based on protocol specifications. The actual dissection of the packets happens in a `dissect_<protocol_name>` function. Figure 3.5 shows an example of such a function in Foo dissector. From line 13 to line 23, different fields of the packet are dissected and added to the Foo dissector sub-tree via `proto_tree_add_item` function. As it can be seen, the dissector iterates through the packet starting from offset 0. Every time a field is detected, the offset is incremented based on the field length.

```
1
2  int dissect_foo(tvbuff *tvb, packet_info *pinfo, proto_tree *tree _U_, void *data _U_)
       {
3      gint offset = 0;
4      col_set_str(pinfo->cinfo, COL_PROTOCOL, "FOO");
5      col_clear(pinfo->cinfo,COL_INFO);
6
7      // Foo does not encapsulate other protocol data, we consume all the date  from 0
       to -1
8      proto_item *ti = proto_tree_add_item(tree, proto_foo, tvb, 0, -1, ENC_NA);
9      // Adds a child node to the protocol tree.
10     proto_tree *foo_tree = proto_item_add_subtree(ti, ett_foo);
11
12     // 1 byte, starting at 0, format is Big Endian.
13     proto_tree_add_item(foo_tree, hf_foo_pdu_type, tvb, offset, 1, ENC_BIG_ENDIAN);
14     offset += 1;
15     proto_tree_add_item(foo_tree, hf_foo_flags, tvb, offset, 1, ENC_BIG_ENDIAN);
16     offset += 1;
17
18     ti = proto_tree_add_item_ret_uint(foo_tree, hf_foo_sequenceno, tvb, offset, 2,
       ENC_BIG_ENDIAN, &sequenceno);
19
20     if(sequenceno == 0){
21       expert_add_info(pinfo, ti, &ei_foo_seqn_zero);
22     }
23
24     return tvb_captured_length(tvb);
25 }
```

Figure 3.5: Dissection in Foo dissector.

3.2 Klee: Dynamic Symbolic Execution Engine

Klee is a dynamic symbolic execution (DSE) engine running on top of the LLVM compiler. DSE employs concrete execution alongside symbolic execution to collect path constraints and explore new ones. Concrete execution is used on instructions that do not depend on symbolic variables. Thus, this behaviour is the same as normal code execution. In addition, Klee supports concrete execution of external calls as system calls and libc library functions.

3.2.1 States

Each path in Klee is represented by an `ExecutionState` containing the current constraints of the explored path. When reaching a branch, an `ExecutionState` is split into two new states, both containing one additional constraint corresponding to the constraint added by the branch and its negation. Klee can use multiple heuristics to perform a faster search. For example, a depth-first search heuristic which first picks the state that has the largest depth or a random state search heuristic which picks a random state every time.

3.2.2 Complete and partial paths

Klee keeps track of completed and partially completed paths during the execution. Klee does not analyze all the paths in parallel. When it reaches the condition, it internally creates two new states. It chooses one state to explore, and other state is set in the queue for future exploration. Thus, if exiting Klee before its termination, there will be states in the queue. Thus, completed paths are paths that have reached termination point in the program while partially completed paths are new unexplored paths set in the queue.

3.2.3 Built-in functions

Klee exposes a set of built-in functions that can be directly called from the target. These functions are used to operate the symbolic values. For example, `klee_make_symbolic` function allows to make a variable symbolic. `klee_assume` constrains the set of possible values a symbolic variable can take. With `klee_stack_trace` a full log of the current state of the stack can be obtained.

Finally, `klee_print_expr` returns the expression of a variable that may depend on a symbolic variable

3.2.4 SMT Solver

A SMT solver such as Z3[17] or STP[11] is necessary to only explore valid paths based on the solution of the constraints in the current state. Such solvers verify the satisfiability of the path to make sure that all paths are reachable. Moreover, Klee uses the solvers to generate sample solution for each explored path.

Chapter 4

Design

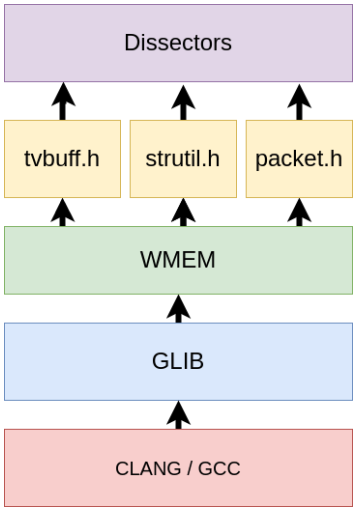
This chapter exposes the design of our system. In the first part, we will present the challenges encountered during the process of building Klark. In the second part, we present its current design. We will explain how native Klee works on Wireshark dissector, and the modifications needed to extract detailed information about protocol fields in the packet. Then we will explain Wireshark's mock implementation needed to compile a packet dissector to LLVM bitcode and successfully execute Klee on it. Finally, we will explain how we solved the collected constraints for each packet type and generated sample packet captures to prove our work.

4.1 Challenges

4.1.1 Compiling Wireshark

Klee performs its symbolic execution on LLVM bitcode. Therefore, to symbolically execute a packet dissector in Wireshark we first have to compile it into LLVM bitcode. This step already brings the first design choice as Wireshark source code is large and complex, and contains a lot of internal and external dependencies. We tried to compile all of Wireshark into LLVM bitcode using the already existing build system. The advantage of this is that we rely on the native Wireshark build, thus the correctness of the symbolic execution is guaranteed. However, this was not achieved because of the machine-generated build code Wireshark is relying on.

Figure 4.1: Simplified Wireshark Dissector dependencies.



4.1.2 Low-level Wireshark Mocking

Since compiling Wireshark into LLVM bitcode was not successful, we tried to mock the functions needed to compile the subset of dissectors we selected. A low-level mock has the goal to not change high-level functions but to provide the necessary low-level environment to compile them such as the libraries they dependent on. We therefore planned to build a simplified Wireshark version using this method.

Wireshark functionalities are built on top of a custom memory management framework (Wmem) which itself is built on top of GLib-2.0. Therefore, we tried to only modify the functions we needed to call our custom Klee built-in function.To compile the rest of the needed Wireshark source code that could depend on Wireshark's custom memory management framework we decided to compile `GLib-2.0.so` into LLVM bitcode and statically link the library when running Klee. This implementation indeed solved the missing dependencies but added new crashes inside GLib-2.0 because of inlined assembly code not supported by Klee. This meant we had to patch out all the inlined assembly that is only used for testing purposes. Considering that, we decided to not pursue this idea. Our next idea was to implement a more complete mock that would modify functions on a higher level and therefore remove the need for the custom memory management framework which would be replaced with standard C heap memory operations.

4.1.3 Protocol Field Extraction

We run Klee on Wireshark dissectors to extract all possible paths and their constraints. Since Klee generates a complete path when the program terminates, each path in Wireshark dissectors represents a separate packet. Thus, we extract all possible packet types and their constraints. In addition, we also want to extract detailed information about the individual protocol fields in the packet. Unfortunately, this information can not be obtained by using a native Klee version.. We therefore, modified Klee allowing us to collect the field information during the symbolic execution.

4.1.4 Packet generation with buffer length

When symbolically executing dissectors, we introduce two symbolized values: buffer and buffer length. After Klee was run on the dissector and the constraints for each path were collected, Klee uses Z3 to generate a sample test for each path (ktest in Klee vocabulary). However, ktest does not consider the correlation between the symbolized buffer and buffer length. It is an important step because first some branch conditions depend on the buffer length and second it is better to preserve the original buffer length.

4.2 Klark Design

4.2.1 Native Klee: Collect Packet Constraints

As stated before, we want to collect the constraints on the symbolic input packet data and thus generate different packet types of the protocol. To prove this concept, we implemented a small dissector similar to a real Wireshark dissector and set simple constraints on the packet data as displayed in 4.2. We initialized the packet data (`input->real_data`) as a symbolic byte array. Thus, all the branches depending on the packet data will create a new path with unique constraints. Based on the example in Figure 4.2, Klee has generated 3 paths in total. Each path contains constraints of `input->real_data`. However, these constraints do not provide any details about the packet field types, size and offset as required in our work. Thus, we need to modify

```
1  if(input->real_data[0] >  127) {
2
3    if(input->real_data[0] == 255){
4        proto_tree_add_item(tree, 0, input, offset, 1, 0);
5        offset += 1;
6
7    }else{
8        proto_tree_add_item(tree, 0, input, offset, 1, 0);
9        offset += 1;
10
11   }
12 } else {
13   int size = input->real_data[0];
14   proto_tree_add_item(tree, 1, input, offset, 1, 0);
15   offset += 1;
16   proto_tree_add_item(tree, 2, input, offset, size, 0);
17   offset += size;
18   proto_tree_add_item(tree, 3, input, offset, 1, 0);
19 }
20
```

Figure 4.2: A custom look-alike Wireshark dissector.

Klee to collect the required details.

4.2.2 Modified Klee: Collect Packet fields

To generate fuzzing specifications we need precise information about the fields of a packet type generated by a set of path constraints. As stated before, Klee does not provide any details about the packet fields in generated constraints. One way to solve this problem is to modify Klee internally so that it collects all packet field information in the current path.

After a detailed analysis of Wireshark dissectors, we noticed that field details, i.e. type, size and offset, are passed in `proto_tree_add_item` function which is used to set field in the sub-tree. The size and the offsets of the fields are either fixed or dependent on another value in the packet. As an example, in Figure 4.2 the size of the field in line 14 is fixed while the size of the field in line 16 depends on the value of the first byte in the packet data (line 13).

Thus, the solution to collect these details from `proto_tree_add_item` is to add a new Klee built-in function `klee_proto_tree_add_item` which will be executed when the original `proto_tree_add_item` function in the dissector is reached. `klee_proto_tree_add_item` function stores the field type and the symbolic expressions of the size and offset for each execution

state. When Klee finishes its run, it outputs original path constraints together with packet field information to a file.

The collection of field information cannot be done in the code under study (the dissector itself) since Klee is the only program aware of the number of active states and their content in Klark.

4.2.3 Wireshark Mocking

In the current state of the mock, we redefine calls to the memory management API with standard C memory management calls (e.g `malloc, calloc`). We also simplify types such as `tvbuff` containing the packet data. Additionally, we add memory access checks and in case of an out-of-bound, we exit the program. This serves two purposes. First, the execution state is terminated and will not lead to a crash. Second, it allows Klee to continue the exploration of other valid paths without continuing the exploration of an invalid path. In the mock, we also support the global field array in each dissector that contains references to protocol field structures. Simplifications have been done compared to the initial Wireshark code. For example, we do not support packet fragmentation or nested dissector.

The mock still uses Wireshark source code to conform to the type definitions, constants and macros it defines. However, functions needed for the correct behaviour of the dissector are redefined. There are 3 ways how we redefine the functions. First, the function can be redefined to do nothing, for example, all user-interface functions are mocked in doing nothing. Second, the function is not modified if it is implementing a common operation such as extracting a 64-bit integer from the `tvbuff`. Last, the function can be simplified. For example, `proto_register_field_array` is redefined to initialize the field array in the mock. This implementation is simpler than the actual function in the source code that performs several other background operations that can be ignored.

4.2.4 Wrapping

The execution of the dissector is done by creating a wrapper that initializes and symbolizes appropriate variables and calls the required dissector. The packet data buffer is initialized,

symbolized and then put in the `tvbuff` struct. This struct is later passed as an argument to the dissector call. The length of the buffer is also symbolized as the packet length can be a factor that determines a packet type. The `proto_tree` struct which represents the main tree in the dissector is also initialized to simply provide a none NULL reference. Once everything is correctly initialized we call `proto_register_dissector` which registers all fields in the global field array declared in the mock. Finally, we can dissect the buffer by calling `proto_dissect_dissector`.

4.2.5 Packet generation

As stated in the challenges, Klee generates the test for sample packet data without taking into account the previously symbolized length. Therefore, we created a Python script in which we generate sample tests using Klee collected constraints and Z3 Solver. In our script, we manually extract a buffer length and apply it to restrict the buffer. With this solution, we can generate packet types that depend on buffer length during the dissection. In addition, as a good practice, we preserve the original buffer length.

Once the sample packet data was generated, we use Scapy[9] packet manipulation program to imitate real packets in the network. Since protocols operate on different network layers, they have their unique specifications. Thus, each protocol required a manual approach to generate sample packets.

Chapter 5

Evaluation

This section evaluates the results produced by Klark. First we evaluate results for all Klark supported protocols. Then we discuss in details the real packet captures generated for ARP and LPD protocols.

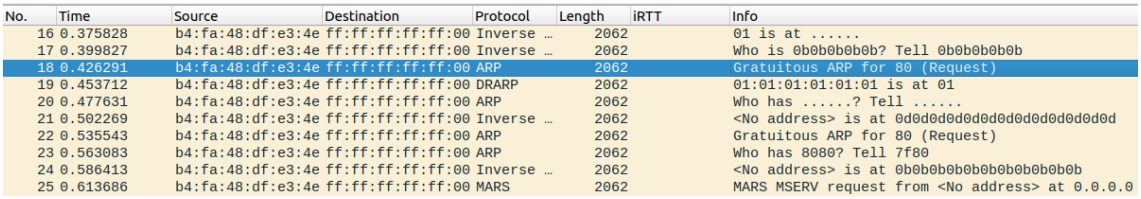
5.1 Overall Results

To confirm Klark's functionality, first we verified the correctness of small protocols such as `ECHO` or `LPD` one. Once we assessed our results, we targeted larger and more common protocol dissectors such as `ARP`, `IP`, `VTP` and `FCCT`. Dissectors are not modified in any way but each of them needs a corresponding wrapper that is initializing the environment.

Current version of Klark supports such protocols as `FCCT`, `VTP`, `ARP`, `LPD` and `ECHO`. We ran Klee on each of these dissectors for 30 seconds and collected the statistics shown in Figure 5.1. During that period, Klee terminated for LPD, FCCT and ECHO dissectors because they are relatively simple. ARP and VTP dissectors are more complex and thus Klee manages to complete some of the paths (Column 3) and set partially completed paths in the queue for future exploration (Column 4).

Dissector	Termination	Completed paths	Partially completed paths
ARP	False	144	1801
LPD	True	6	0
VTP	False	502	969
FCCT	True	61	0
ECHO	True	1	0

Figure 5.1: Dissector Symbolic execution results after 30 seconds.



No.	Time	Source	Destination	Protocol	Length	iRTT	Info
16	0.375828	b4:fa:48:df:e3:4e	ff:ff:ff:ff:ff:00	Inverse …	2062		01 is at .....
17	0.399827	b4:fa:48:df:e3:4e	ff:ff:ff:ff:ff:00	Inverse …	2062		Who is 0b0b0b0b0b? Tell 0b0b0b0b0b
18	0.426291	b4:fa:48:df:e3:4e	ff:ff:ff:ff:ff:00	ARP	2062		Gratuitous ARP for 80 (Request)
19	0.453712	b4:fa:48:df:e3:4e	ff:ff:ff:ff:ff:00	DRARP	2062		01:01:01:01:01:01 is at 01
20	0.477631	b4:fa:48:df:e3:4e	ff:ff:ff:ff:ff:00	ARP	2062		Who has ......? Tell .....
21	0.502269	b4:fa:48:df:e3:4e	ff:ff:ff:ff:ff:00	Inverse …	2062		<No address> is at 0d0d0d0d0d0d0d0d0d0d0d0d0d
22	0.535543	b4:fa:48:df:e3:4e	ff:ff:ff:ff:ff:00	ARP	2062		Gratuitous ARP for 80 (Request)
23	0.563083	b4:fa:48:df:e3:4e	ff:ff:ff:ff:ff:00	ARP	2062		Who has 8080? Tell 7f80
24	0.586413	b4:fa:48:df:e3:4e	ff:ff:ff:ff:ff:00	Inverse …	2062		<No address> is at 0b0b0b0b0b0b0b0b0b0b0b
25	0.613686	b4:fa:48:df:e3:4e	ff:ff:ff:ff:ff:00	MARS	2062		MARS MSERV request from <No address> at 0.0.0.0

Figure 5.2: Generated ARP packets.

5.2 Detailed Result Analysis

Klark is able to generate real packet data from the constraints obtained by executing our modified version of Klee on a dissector. In order to verify the packet data, we build a sample packet capture using generated packet data. We run Wireshark to check that it can correctly dissect the packets and to verify that the packets have different types and fields.

5.2.1 ARP

ARP is one of the backbone protocols in the current network architecture. The main functionality of ARP is to resolve the MAC address of a device given its IP address.

Figure 5.2 shows a sample of packets generated from the constraints. Each packet was generated with a different set of constraints and thus represents a different path in the dissector. We can recognize unique packets from dynamic reverse ARP packets to ARP Response and requests. As one may notice, there are some packets that have the same description. However, the difference in those packets lies inside in their fields as compared in Figure 5.3. Both packets are of a `Gratuitous ARP for 80 (Request)` type, but they differ in the protocol they try to resolve the MAC address for. The first packet resolves the MAC address of a device with a field of Unknown

```
Hardware type: Ethernet (1)
Protocol type: Unknown (0x0000)
Hardware size: 15
Protocol size: 1
Opcode: request (1)
[Is gratuitous: True]
Sender hardware address: 808080808080808080808080808080
Sender protocol address: 80
Target hardware address: 000000808080808080808080808080
Target protocol address: 80
```

(a) Packet No. 18

```
Hardware type: Ethernet (1)
Protocol type: IPv4 (0x0800)
Hardware size: 7
Protocol size: 1
Opcode: request (1)
[Is gratuitous: True]
Sender hardware address: 07070707070707
Sender protocol address: 80
Target hardware address: 80070707070707
Target protocol address: 80
```

(b) Packet No. 22

Figure 5.3: Fields of two similar ARP packets.

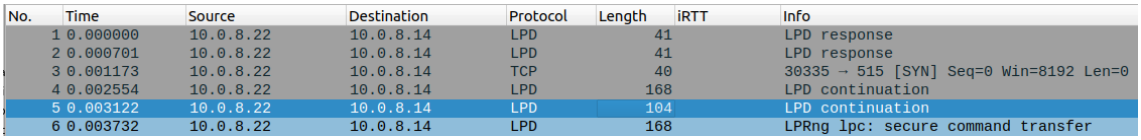
```
Protocol fields [
    //Concrete Field type, Offset and Size
    FieldType(w32 0)
    Offset(w32 0)
    Size(w32 2)

    ...
    //Symbolic Field type, Offset and Size
    FieldType(Select w32 (And (Or (Eq 2048
                                   (Extract w16 0 (Or w32 (Shl w32 (ZExt w32 (ZExt w16
     (Read w8 2 buf))) ... )
    Offset(Add w32 8 (ZExt w32 (Read w8 4 buf)))
    Size(ZExt w32 (Read w8 5 buf))
]
```

Figure 5.4: Fields of a generated ARP packet.

protocol whereas the second packet resolves the MAC address against an IPv4 address.

As stated before, having the constraints of a packet type is not enough to achieve our final goal which is the generation of fuzzing specifications. Indeed, such specifications require more information about the individual packet fields. Therefore, alongside the packet constraints we collected the information about individual fields inside the packet.

Figure 5.4 shows the example of collected fields inside one generated ARP packet. The first field is of type 0, starts at the offset 0 and has size of 2 bytes. From the field type we can then recover detailed information about this field to write a better specification. In this case field type 0 represents the Hardware type where the ARP packets are transmitted on. Furthermore, it might be that the the field type and/or length depends on the value of a previous field. Thus, the length can be expressed as an expression with respect to packet data. This dependency can only be recovered through the fields, not the constraints.



No.	Time	Source	Destination	Protocol	Length	iRTT	Info
1	0.000000	10.0.8.22	10.0.8.14	LPD	41		LPD response
2	0.000701	10.0.8.22	10.0.8.14	LPD	41		LPD response
3	0.001173	10.0.8.22	10.0.8.14	TCP	40		30335 → 515 [SYN] Seq=0 Win=8192 Len=0
4	0.002554	10.0.8.22	10.0.8.14	LPD	168		LPD continuation
5	0.003122	10.0.8.22	10.0.8.14	LPD	104		LPD continuation
6	0.003732	10.0.8.22	10.0.8.14	LPD	168		LPRng lpc: secure command transfer

Figure 5.5: Generated LPD packets.

5.2.2 LPD

The Line Print Daemon protocol is a simple printer protocol used to communicate with printers. The protocol is fairly simple with only a small set of available operations/packet types. To be precise, the dissector `packet-lpd.c` contains the following field types: `LPD response`, `LPD request`, `LPD client code`, `LPD printer option` and `LPD response code`. Klark generated 6 sample packets in total as displayed in Figure 5.5. It can be seen that packet 4 and packet 5 both have `LPD Continuation` info, but the data inside actually differs. The same situation happens with packet 1 and packet 2 with `LPD Response` info.

Chapter 6

Discussion

6.1 Limitations

6.1.1 Path explosion

Path explosion is one of the main problems in large and complex dissectors. This happens particularly when the dissector operates on the bit level instead of the byte level. As an example, there are 4 categories of packets in the `VTP` dissector (`packet-vtp.c`): `SUMARRY_ADVERT`, `SUBSET_ADVERT`, `ADVERT_REQUEST`, and `JOIN_MSG`. For each category, additional dissection is done. The `JOIN_MSG` branch contains a for-loop that iterates through each bit in a byte (6.1). For every bit, 2 new branches are created because of the if-statement inside. Thus, only for one byte there already exist 256 states in Klee's state manager unit. This dissection is done on every remaining byte in the packet that we assume of having a maximum length of 2048 bytes. This execution will never terminate.

Another example of path explosion is in the `IRC` dissector (`packet-irc.c`) where the end of the message is a `EOL` character. The number of states explodes here too because a new state will be created for every possible position of the `EOL`. Furthermore, other delimiters can also exist and pose the same problem as the `EOL`.

A solution to limit path explosion would be to target a specific subset of packet types and constrain the set of valid paths when we declare the packet data as symbolic. Additional constraints can be applied to the packet data in the dissector wrapper after it has been declared symbolic.

```
vlan_usage_bitmap = tvb_get_guint8(tvb, offset);
for (shift = 0; shift < 8; shift++) {
  if (vlan_usage_bitmap & (1<<7)) {
    proto_tree_add_uint(vtp_pruning_tree, hf_vtp_pruning_active_vid,
    tvb, offset, 1, pruning_vlan_id);
  }
  pruning_vlan_id += 1;
  vlan_usage_bitmap <<= 1;
}
```

Figure 6.1: `packet-vtp.c` dissector code

6.1.2 LLVM IR

We can observe that in the `ECHO` dissector only a single path is explored. However, this protocol supports two packet types each with a different first field: `Request` and `Response`. The exceptional case here is that the decision of the packet type in the dissector is made with a ternary statement on the variable `request`. Unfortunately, ternary statements are not detected by Klee as in the LLVM IR they are represented through a `select` statement. This is a limitation of Klee - an IR-Based Symbolic Execution engine that is known to generate under-constrained outputs.

6.1.3 Wireshark Mock Correctness

Klark relies on the mock of Wireshark to explore different paths in the dissector and to find constraints for different packet types. However, the mocking of Wireshark requires a lot of manual work and can be prone to errors. Wireshark is also a big project with many dependencies and interactions between different sub-modules that need to be mocked correctly.

6.2 Further Improvements

Klark is the first system in the community that automatically extracts fuzzing specifications. Although Klark managed to successfully analyze a number of protocols, there are a lot of challenges and limitations faced. In this section we try to address the improvements that can be done in the future.

6.2.1 Extended Mocking

Klark should be improved to support a bigger range of protocols by extending the mock functionalities. A more complete mock can only be beneficial for Klark as it would also enforce the correctness of depending sub-modules. For example, the `IP` dissector is partially supported. The IP option fields are registered as separate dissectors and called indirectly. However, nested dissector calls are not yet supported.

6.2.2 Limit Path explosion

We encountered several cases of state explosion which could be solved by using loop summarization. The mock already already provides partial function summaries but loops in the dissector itself are still a problem. Loop summarization could be used to provide a summary of the loop statement such that the number of paths does not grow exponentially.

6.2.3 Generate Fuzzing specifications

Klark extracts the path constraints for every packet type. Additionally, the size, offset and field type index is extracted for every field of packet. Field type index is the index of the appropriate field type in the array storing all protocol fields. Current version of Klark does not access the actual field type. One way to solve it is to extract the global array with protocol fields from the wrapper and use it to access necessary fields later. Then we can create protocol specifications with the size, offset and the field type information.

6.2.4 In-breadth Fuzzing

Klark can be improved by generating multiple variations of the same packet to be used in in-breadth fuzzing. A trivial way of implementing this is to generate the first packet from the constraints and then add the constraints of the generated packet to the initial constraints. Therefore, the second packet generated will follow the initial constraints but it will also be different from the first one. This process can then be repeated until the constraints are not satisfiable anymore.

6.2.5 Extend Format Fuzzer

FormatFuzzer [2] transforms dissectors into fuzzing harnesses. This can be extended with the knowledge of the symbolic constraints Klark brings about the packet data. Additional fuzzing harnesses can be generated from not yet supported dissectors containing complex branch conditions from a transformation of the packet data.

Chapter 7

Conclusion

Network protocols are the core of the Internet making them responsible for the network functionality and even the safety of the users and devices on it. Correctness of network protocol design and implementation is commonly done with fuzzing. Even though individual research successfully fuzzed a number of protocols, fuzzing can benefit greatly from the precise protocol specifications to evaluate the tested system more effectively.

This project proposed to improve network protocol fuzzing by automating the extraction of protocol specifications. We designed and implemented a system Klark that uses Klee to symbolically execute Wireshark dissectors and collect packet data constraints and field details. The main phases of this project were (i) to analyze current community progress in automatic specification extraction, (ii) to mock Wireshark's environment to compile its dissectors to LLVM bitcode for symbolic execution with Klee, (iii) to modify Klee to collect packet fields data, (iv) perform symbolic execution of Wireshark dissectors using Klee and collect constraints/packet fields, and (v) generate sample captures.

Despite the limitations of symbolic execution and complexity of Wireshark dissectors, we were able to successfully generate valid packet data for such protocols as ARP, LPD, VTP and FCCT. Moreover, we successfully verified our results by imitating real packet capture with generated packet data for ARP and LPD protocols.

Automatic extraction of protocol specifications is a novel approach in the community. Even though the current state of Klark requires further improvements, its results can already be used

in practice: 1) generated packets can be used as a high-quality seed for fuzzing, and 2) partially extracted field information can already be used as specifications.

In future, we would like to support other network protocols thus creating more fuzzing specifications. We would like to generate multiple variants of the same packet. Last, we would also like to create a comprehensive dataset of protocol specifications to be used by the community.

Bibliography

[1] *Common Vulnerabilities and Exposures (CVE)*. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=protocol. CVE in protocols. Accessed: 2022-06-06.

[2] Rafael Dutra, Rahul Gopinath, and Andreas Zeller. *FormatFuzzer: Effective Fuzzing of Binary File Formats*. 2021. DOI: 10.48550/ARXIV.2109.11277. URL: https://arxiv.org/abs/2109.11277.

[3] Marc Egli and Kamila Babayeva. *Klark tool*. https://github.com/Marc-Egli/blob_dissector. Accessed: 2022-06-06.

[4] A. Huelsing, D. Butin, S. Gazdag, J. Rijneveld, and A. Mohaisen. *XMSS: eXtended Merkle Signature Scheme*. RFC 8391. RFC Editor, May 2018.

[5] *Integer overflow in OpenSSH*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-16905. Accessed: 2022-06-06.

[6] Maria Lenore Pacheco, Max von Hippel, Ben Weintraub, Dan Goldwasser, and Cristina Nita-Rotaru. "Automated Attack Synthesis by Extracting Finite State Machines from Protocol Specification Documents". In: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2022.

[7] Yan Pan, Wei Lin, Liang Jiao, and Yuefei Zhu. "Model-Based Grey-Box Fuzzing of Network Protocols". In: *Security and Communication Networks* 2022 (2022), p. 13.

[8] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. "AFLNET: A Greybox Fuzzer for Network Protocols". In: *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 2020, pp. 460–465. DOI: 10.1109/ICST46399.2020.00062.

[9] *Scapy Packet Manipulation Program*. https://scapy.net/. Accessed: 2022-06-06.

[10] Statista. *Global digital population as of April 2022*. https://www.statista.com/statistics/617136/digital-population-worldwide/. Accessed: 2022-06-06. 2022.

[11] *STP Theorem Prover*. https://stp.github.io/. Accessed: 2022-06-06.

[12] *Suricata Threat Detection Engine*. https://suricata.io/. Accessed: 2022-06-06.

[13] SweetScape Software. *010 Editor*. https://www.sweetscape.com/010editor/. Accessed: 2022-06-06.

[14] *The Heartbleed Bug*. https://heartbleed.com/. Accessed: 2022-06-06.

[15] Shameng Wen, Chao Feng, Qingkun Meng, Bin Zhang, Ligeng Wu, and Chaojing Tang. "Testing Network Protocol Binary Software with Selective Symbolic Execution". In: *2016 12th International Conference on Computational Intelligence and Security (CIS)*. 2016, pp. 318–322. DOI: 10.1109/CIS.2016.0078.

[16] *Wireshark network protocol analyzer*. https://www.wireshark.org/. Accessed: 2022-06-06.

[17] *Z3 Theorem Prover*. https://github.com/Z3Prover/z3. Accessed: 2022-06-06.

[18] *Zeek Network Security Monitoring Tool*. https://zeek.org/. Accessed: 2022-06-06.