Fuzzing an Industrial Proprietary Protocol

Eduard Baranov[0000−0002−7357−705X], Axel Legay[0000−0003−2287−8925], and Martin Vivian[0009−0009−5736−4394]

INGI, ICTEAM, UCLouvain, Belgium

Abstract. For many proprietary systems source code and documentation are not available which makes them hard to test leaving only black-box approaches. In this work, we present an experience of fuzzing a protocol for drone control and the developed tool BinFuzz. BinFuzz is a man-in-the-middle stateful black-box protocol fuzzer. Listening to real communication as a man-in-the-middle, the fuzzer reconstructs states of the protocol as well as detects message types and their variable fields. The collected knowledge is used during the fuzzing to improve the quality of the generated inputs. For the application, we first test BinFuzz on an FTP protocol and then use it to fuzz the protocol for drone control.

Keywords: Protocol Fuzzing · Vulnerability Detection

1 Introduction

Software systems are moving towards distributed setup. Parts of a system can be placed in different locations or a system operation might utilize services provided by other companies. The Internet of Things is just one example. What makes the separate parts a single system is their interaction. In a distributed setup the interactions require one or several communication protocols for information transfer over a network. While many systems use one of the standard well-known protocols, proprietary protocols are common as well. In the latter case, documentation is often not up-to-date, not available, or does not exist at all.

One of the important aspects of system development is ensuring its correctness. Fuzzing is a well-known testing approach applicable to a black-box scenario where the knowledge of system internals is not required. The core idea of fuzzing is to generate a large number of inputs to the system and to monitor its behavior for unexpected results or crashes. A large number of fuzzing tools have been developed over the years, both in industry and in academia. A survey [12] lists about 50 black-box fuzzers that either appeared at a major conference or have many GitHub stars. Many systems have benefited from fuzzing and, according to Google, the number of bugs uncovered with fuzzers is immense [15, 3] - only in Chromium tens of thousands of bugs have been found with fuzzing. However, the majority of the fuzzers focus on a single application rather than on communication protocols and interactions between different systems or parts of a system.

In this work, we have a task to test an industrial proprietary protocol for drone control. Only few details of this protocol have been given to us while source

code and documentation were unavailable. In addition, traffic analysis uncovered several particularities of this protocol such as a lack of textual interpretation of the messages and no evident order of directions in a message sequence. As a result, none of the existing tools we tried to apply were capable of fuzzing the protocol. Therefore, we designed a new tool BinFuzz - a stateful man-in-the-middle (MITM) fuzzer that can be applied to a wide range of protocols including proprietary ones and, in particular, to the protocol for drone control. It combines a set of ideas for a more targeted input generation and can be used to test binary (non-text-based) protocols. BinFuzz operates in two phases: first it listens to the real communications and reconstructs an approximate protocol structure in the form of a Finite State Machine, and in the second stage it intercepts and mutates messages in the communication channel.

In this paper, we present challenges encountered in industrial protocol testing and present our solutions. We describe the ideas behind the BinFuzz tool and show its application to two protocols. First, we fuzz a faulty implementation of an FTP protocol to ensure that the tool is capable of finding bugs. Then we show the fuzzing of the protocol for drone control.

The rest of the paper is organized as follows. In Section 2 we list the challenges common to protocol fuzzing and particular to the protocol for drone control. In Section 3 we present the architecture of BinFuzz and how protocol learning and fuzzing are working. Section 4 shows the application of BinFuzz to two protocols and discusses the limitations. Related work is presented in Section 5. Section 6 concludes the paper.

2 Challenges

There are several challenges specific to protocol fuzzing in a black-box scenario. First of all, non-trivial protocols are stateful, and therefore the set of expected inputs changes over time during the execution and depends on the current state of the protocol. The fuzzer must be capable of reaching different states and selecting relevant inputs. Another challenge is to generate relevant inputs: message structures are usually fixed and might include a large number of different fields. If a detailed protocol specification is available, it is not complex to generate a valid input, however, this is rarely the case for proprietary protocols. In the latter case, the structure derivation is a complex task. For example, messages in the drone control protocol have 15 fields of different lengths in the header part. Moreover, valid messages may include information from previous messages, e.g. session ID is selected during the first steps of the communication and has to be included in all consequent messages.

An approach to overcome the aforementioned challenges has been proposed in [7]. The idea is to place a fuzzer between two communicating parties as MITM. First of all, this position allows the fuzzer to gather knowledge about a protocol by intercepting messages. Secondly, the fuzzer can modify messages in the network: instead of generating fuzzing inputs from scratch, it mutates real messages that are expected by the receiver. As a result, fuzzing can be

performed without deriving full knowledge of the protocol, for example, session ID is included in the intercepted message and is included in the fuzzed input unless the corresponding field has been mutated. Several MITM fuzzers have been developed in recent years, however, they have their limitations. Some of them are stateless (e.g. [13, 20]), and others support only text-based protocols (e.g. [7]) limiting their applicability. A common assumption in MITM fuzzers is an alternation of message direction, i.e. messages are always grouped in pairs with the first message always sent from the first communication party and the second message coming from another party.

The protocol for drone control that we have tested has a set of particularities adding additional challenges. First of all, messages do not have a text-based interpretation. Secondly, the protocol sends hundreds of messages per second adding a performance requirement to a fuzzer. Finally, it does not follow a request-response message sequence: it is common for the protocol to send multiple messages in one direction and then transfer a different number of messages in the opposite direction. In addition to that, the TCP protocol periodically transfers several messages in a single packet further complicating message ordering.

3 BinFuzz

The protocol for drone control due to its specifics cannot be fuzzed with existing tools. Therefore, for this work, we have developed a new tool BinFuzz. BinFuzz is a Man-in-the-Middle black-box fuzzer that can be applied to a large specter of protocols. A high-level architecture of the fuzzer is shown in Figure 1. Its execution has two phases. At the first phase BinFuzz works as a passive listener: it records all the messages passing in the network in order to obtain some knowledge about the protocol. Alternatively, it can use previously collected communication traces. The communications are processed in order to extract different message types and to build a finite state machine (FSM) approximating the protocol behavior. The second step is active fuzzing: intercepted messages are mutated according to their type and current FSM state and are forwarded to the receiver. Throughout the fuzzing process, the FSM state is updated based on the messages going through the network. While it is possible to modify all the messages, we opted out to modify messages in one direction only, as it provides better control of the protocol state change. In the remainder of the paper, we call the communicating parties the client and the server, and the tool fuzzes messages from the client to the server.

BinFuzz has been forked from AutoFuzz [7] and implemented in Java. The tool is placed as a proxy server: a client has to be configured to communicate with the tool and the tool gets a server address as input. In the remainder of the section, we provide details about different parts of the tool.

3.1 Protocol Reconstruction Phase

During the first phase, shown on the left of Figure 1, BinFuzz attempts to learn a protocol by listening to the network traffic or by scanning previously collected

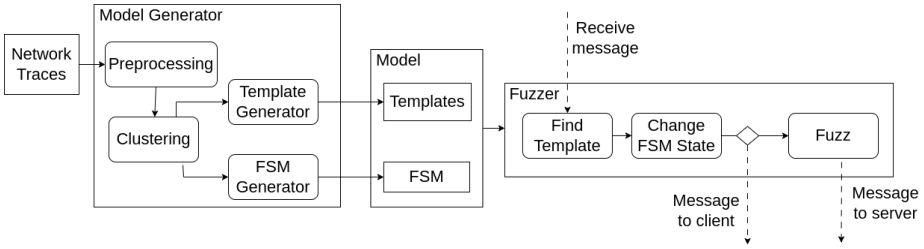


Fig. 1. BinFuzz Architecture.

data. The input to this phase is a set of traces; each trace is a sequence of messages and their directions. The output of this phase is a model of the protocol consisting of two parts: an FSM and a set of templates representing different message types. FSM transitions are labeled with templates.

The first step of model construction is message clustering. Messages to and from the server are clustered separately. At this step, all the messages in each direction from all the traces are taken and grouped according to their contents. The order of messages in the traces is not considered during the clustering. Messages are represented as hex strings and the similarity distance is based on the number of positions in which the messages have different values. Our approach is not limited to a particular clustering algorithm, in the implementation we used a Birch Leaf hierarchical clustering algorithm [31]. Hierarchy allows the user to adjust the number of clusters, as the number of message types is unknown in the black-box scenario, plus the ability of Birch to cluster incrementally is useful if the traces are not collected in advance.

In order to improve clustering results we use additional preprocessing. Some of the message fields can add noise to the clustering as their values may seem random. Examples of such fields are a millisecond part of a timestamp or a checksum. Moreover, these fields are usually located in a message header and likely in the same position in all message types. We use the Chi-Squared test to find positions in messages where the distribution of values across the whole message set is close to uniform. Such positions are ignored in the distance computation. The second heuristic is based on the idea that the message type is often encoded in its header or in the first bytes, for example, an FTP message starts with a command. Therefore, BinFuzz has the option to cluster according to the first n bytes of messages. With protocol knowledge, this bound can be set precisely, in black-box cases different values of n can be tried out.

Resulted clusters are used to build message templates and an FSM. The messages from traces are mapped to their respective clusters. Traces consisting of cluster sequences are used to construct an FSM. Transitions between states are labeled with cluster IDs. Note that messages to and from a server have been clustered separately, therefore the distinction between messages in different directions is maintained in the FSM. We use an open-source tool FlexFringe [28] to generate an FSM from the traces.

For each cluster, BinFuzz builds a template representing a message type. A template is a hex string with two special symbols: '-' and '+' representing variable parts of the message type. We call positions in the template with special symbols as a variable part of the template and all positions with standard hex symbols (i.e. from 0 to f) as a fixed part. Any message of the type corresponding to the template can have any values in the variable part while it is expected to coincide with the template's fixed part. For example, in FTP protocol a message type "USER" would have a template "5553455220------" where "5553455220" is a fixed part "USER " in hex and "------" is a variable part. Any message of this type is expected to start with "USER " followed by any symbols.

For the fuzzing, we consider two levels of variability for the variable part: strongly variable (denoted with "+") and weakly (denoted with "-") variable. The idea behind the separation is similar to the one for preprocessing described above. Strongly variable part positions have a close to uniform distribution of values in the cluster. An exhaustive fuzzing of fields such as milliseconds in timestamps or checksums rarely poses interest - all possible values can normally appear in real messages. During the fuzzing, the focus is given to the weakly variable parts.

The construction of templates is straightforward: if all messages in the cluster have the same value in a particular position then it is a fixed part, otherwise, we run the Chi-Squared test to distinguish between strongly and weakly variable parts. Note that while it is expected that positions labeled during the preprocessing become highly variable parts, the construction of templates has to recheck these positions. An example of a beginning of a template from the drone control protocol is "fe--0-0000++++++++++++++++00...". The first byte "fe" indicates the beginning of a message and is the same for all messages. The next two bytes indicate message length; while messages could have different lengths, the distribution of values is not uniform. The part consisting of plus symbols corresponds to a checksum, the Chi-Squared test does not show a significant difference from the uniform distribution.

3.2 Fuzzing Phase

The second phase of BinFuzz shown in the right part of Figure 1 is fuzzing. The tool continues to listen to network traffic and attempts to follow the protocol updating the state of the FSM accordingly. For this purpose, it searches for a corresponding template (i.e. cluster) for each message and then for an outgoing transition from the current FSM state labeled with the cluster. The metric for the template search is the proportion of the fixed part of the template coinciding with the message, the variable part of the template is ignored during the template search. If no template is found, the state is left unchanged.

In addition to the FSM state update, BinFuzz intercepts messages from the client, mutates them, and the server receives a modified version. For the mutations, we use the variable part of templates. A mutation function takes as an input an original message and a template and outputs a mutated message. We have implemented several mutation functions in our tool including:

- Random replace: with a predefined probability a value from the variable part of the template is changed to a new random value. We distinguish two probabilities for the mutation, for strongly and weakly variable parts.
- Random group replace: same as above but a group of several sequential hex digits is replaced. The maximal size of the group is a predefined parameter.
- 0/f replace: it replaces a value (or a group) with '0'(s) or 'f'(s).
- Random insert: it adds a random group of values at a random place of the message including at the beginning and the end of the message.
- Random delete: it deletes several values at a random place.
- Duplicate: it selects a group of hex digits and inserts several copies of the group at the end of the group. An alternative version duplicates the whole message in order to test the replayability of messages.

Other mutation functions can be easily added to BinFuzz. The mutated message is sent to the server. Logging at the server and client sides would allow the user to check if anything goes wrong. BinFuzz logs all the messages, their templates, and mutated versions allowing a reconstruction of the traces leading to an issue.

4 Protocol Fuzzing

As a first test to ensure that BinFuzz is capable of finding bugs, we used a faulty implementation of an FTP server. After that, we fuzzed the proprietary protocol for drone control. In this section, we present the fuzzing results.

4.1 FTP Protocol

FTP protocol [19], having dozens of implementations, is a classic case study for protocol fuzzers. Messages are text-based and have a simple structure: client messages start with a command of 3 or 4 symbols, followed by space and command arguments, and server replies start with a 3-digit code, followed by space and some text. Messages end with a new line ("/r/n" in the case of Windows).

Nevertheless, despite being quite simple, some of the implementations have vulnerabilities. We used a publicly available FTP server "Open and Compact FTP Server"[1] version 1.2 that is known to have several vulnerabilities. The goal of this test was to check that BinFuzz is capable of finding them.

We collected 166 communication traces using a Python script that represented a client: first, it attempts to connect to the server using one of the given credentials and then sends several commands randomly selected from a list. The collected traces have been given to BinFuzz as an input. During the first phase, BinFuzz builds a set of templates and an FSM of the protocol. Knowing that FTP commands are identified by at most 4 symbols and the server replies by 3 symbols, we set a header size to 8 (each byte is represented by 2 hex digits).

Examples of generated templates are shown in Table 1, where each row contains a cluster ID, a message direction, a template, and its length (the

[1] https://sourceforge.net/projects/open-ftpd/files/open-ftpd/

ID	Dir	Size	Template
1000	S	24	5553455220-+-+-+-+-+0d0a
1000	S	26	5553455220-----+-+-+-+0d0a
1001	S	26	5041535320-+-+-+-+-+-+0d0a
1001	S	28	5041535320++++++++++++++0d0a
1002	S	30	4d4b4420++++++++++++++++++0d0a
...	...	...	...
1007	S	8	51554954
1008	S	42	5245545220-+-+-+-+---+-+---+-+2e7478740d0a
1009	S	42	44454c4520-+---+-------+-+-+-+2e7478740d0a
1010	S	12	4c4953540d0a
0	C	360	3232302d202a2a2a2a2a2a2a2a2a2a2a2a2a2a2a2a...
1	C	78	353330204c6f67696e206f722050617373776f7264...
1	C	54	353330204c6f67696e20696e636f72726563742e20...
...	...	...	...
5	C	86	3535302022-+-+-+-+---+-+---+-+2e7478742220...
6	C	110	323530204368616e67656420746f20646972656374...
6	C	68	323530204368616e67656420746f20646972656374...
7	C	126	313530204f70656e696e672062696e617279206d6f...
7	C	102	313530204f70656e696e672042696e617279206d6f...
8	C	50	323236205472616e736665727420436f6d706c6574...

Table 1. A subset of templates for the FTP protocol. "+" and "-" symbols represent variable positions of the template. "S" represents messages sent to the server, "C" is for replies to the client. Several clusters with the same ID represent several versions of the cluster template for different message lengths.

number of hex digits). For clusters corresponding to client requests, the templates start with 6 or 8 hex digits followed by "20" representing a space symbol. This part represents a command and a space symbol. For example, cluster 1000 has templates for the "USER" command, cluster 1001 - for "PASS", and cluster 1007 is "QUIT". The next positions in the templates have either '-' or '+' representing variable parts of the template. These parts are fuzzed during the second phase. The substring "2e747874" in several clusters corresponds to ".txt" since all of the file operations in the collected traces used text files. All server replies start with a reply code followed by "20". Most of them are fixed messages, therefore almost all templates have no variable part. The only exception is the "file unavailable" message that includes the file name. All templates end with "/r/n" symbols.

A generated FSM for the FTP protocol is shown in Figure 2. State 0 corresponds to the beginning of the protocol, and its outgoing transition is the first message which is a 220 "hello" message from the server. From the next state, there is an outgoing transition labeled with cluster 1000 - a USER command followed by either a reply "331" or "530". In the former case password is sent and it is either accepted or rejected. After the login procedure, the client can send any command and receive one of the possible responses. The FSM is almost bipartite: client requests (colored in blue) are followed by server replies (colored in red). The only exception is state 17, since during the data transfer the server

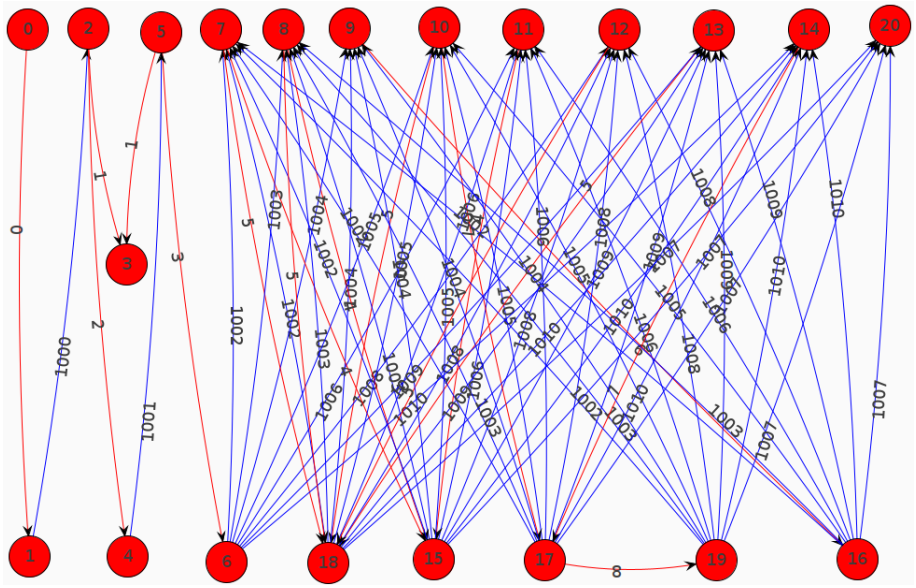


Fig. 2. FSM generated by BinFuzz for the FTP protocol. Blue transitions correspond to client messages and red transitions correspond to server replies.

sends two consecutive messages to open and close the data connection. There are 2 sink states without any outgoing transitions 3 and 20 representing wrong username or password and quit, respectively.

During the second phase, client commands have been mutated with different mutation functions. Multiple vulnerabilities have been quickly uncovered. Among them: the insertion of "/r", "/n" or a space symbol in the middle of the parameters could crash the server; the removal of the whole template variable part (i.e. sending a command without a parameter) could also result in the crash. These results show the effectiveness of the tool.

4.2 Proprietary Drone Control Protocol

We used BinFuzz to search for vulnerabilities in a proprietary protocol of an industrial company that is used to control unmanned aerial vehicles (UAVs). Limited knowledge of the protocol has been given that does not include either source code or documentation, therefore only the black-box approach is possible.

Protocol Description Overall, the protocol involves three entities: a drone, a base station, and a server which are shown in Figure 3. The drone runs software on an Ubuntu OS that includes components for network communication, navigation, control of the drone's modules such as engines, cameras, etc., and software for logging and telemetry. For the testing, we have been given a virtual machine with an installed drone emulator.

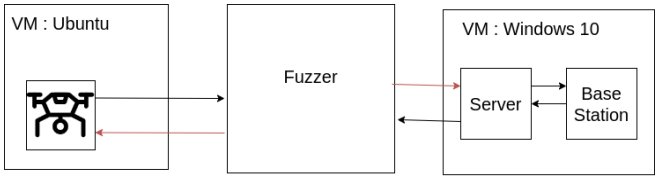


Fig. 3. Drone control system fuzzing setup.

The base station serves as the primary interface through which users interact with the drone. A user can view the drone's position on a map, its telemetry data, such as speed and battery charge, and issue various commands such as take-off, landing, fly to a point, and other control operations. The base station is implemented as software running on a Windows 10 operating system.

The server is an intermediary component that transfers commands between the base station and the drone. It runs on Windows and in addition to the message relaying, the server is responsible for authentication and logging. In our case study, it is hosted on the same machine as the base station.

At the initialization step, both the base station and the drone establish a connection to the server that is further used to transfer the messages. There are multiple types of messages sent by the protocol including command messages issued by a user via the base station, telemetry messages with real-time information about the drone status, logging messages, and keep-alive messages to check the connection and to restart it in case of problems.

All messages have an identical structure. Each message has a start byte "fe" followed by a header, a payload, and ends with a byte "0a". Headers contain essential information such as message length, checksum, timestamp, and a service ID that is expected to receive the message. All fields of the header have fixed positions in the message. The payload comprises serialized objects.

As discussed in Section 2, the protocol has several specifics that make the fuzzing more complex. There is no general text-based interpretation of the messages. The messages are often sent in groups and we were not given any knowledge allowing us to find a relation of a message with one of the previous ones, i.e.. it is not possible to find request-response pairs. In addition, the protocol sends hundreds of messages every second, thus searching for related messages based on their timestamp is not possible either. For example, by attempting to decode text from messages, it is possible to find a supposedly logging message with a user command, yet the message transferring this command to the drone cannot be detected.

Trace Collection BinFuzz inputs is a set of traces containing message sequences. BinFuzz has been placed on the network between the drone and the server and recorded communications. At the beginning of each trace, we restarted the drone and the server. During the recording, a user issued several commands to the drone:

ID	Dir	Size	Template
0	D	152	fe4c000000+++++++-+++++++-00000000000000101000008000...
1	D	152	fe4c000000+++++++++++++++00000000000000101000008000...
2	D	136	fe44000000+-++++-++-++++-+00000000010001050000008000...
3	D	960	fee0010000+++++++++++++++00000000530001010000002000...
...	...	...	...
1000	S	152	fe4c000000++-+++++++-+++++00000000000000101000008000...
1001	S	152	fe4c000000+++-+++++++-++++00000000000000101000008000...
1002	S	136	fe44000000+++++++++++++++00000000630000090000002000...
1003	S	136	fe44000000+-++++-++-++++-+00000000010000090000008000...
1004	S	1180	fe--0-0000+++++++++++++++00000000630001010000002000...
1004	S	482	fe--0-0000+++++++++++++++00000000630001010000002000...
1004	S	3096	fe--0-0000+++++++++++++++00000000630001010000002000...
1005	S	634	fe----0000+++++--++++++--+00000000--0-010100000-000...
...	...	...	...
1006	S	934	fe--010000+++++++++++++++00000000630001010000002000...
1007	S	1392	fe--0-0000-++++++--++++++-00000000630001010000002000...
1007	S	484	fe--0-0000-++++++--++++++-00000000630001010000002000...
1007	S	452	fe--0-0000-++++++--++++++-00000000630001010000002000...

Table 2. A subset of templates for drone protocol messages. "D" represents messages sent to the drone, and "S" is for messages to the base station.

in some traces, there are only 3-4 commands such as start or stop motors, while other traces contain longer lists of commands involving a flight to a destination. In total, we recorded 28 communication traces. The number of messages varies between 1500 and 5000, the total number of messages is 93000.

A quick exploration of messages showed the following statistics. The drone has been sending messages 3 times more often than the base station. Among all the messages there are only 104 different message lengths. 68% of messages are exactly 480 bytes long, 15% of messages have the length of either 68 or 76 bytes, 12% are longer than 6000 bytes, and the remaining messages have lengths between 170 and 750 bytes.

During the exploration of messages, we found out that some of the messages have been packed into a single TCP packet, moreover, a single packet could contain up to 8 messages. While it is possible to use these packed messages in the model construction, their utilization deteriorates the clustering results and the following FSM construction. Therefore, in order to obtain a better model of the protocol, we used the knowledge about the position of a message length field. This knowledge has been used only for the separation of messages: after the interception of a TCP packet a proxy server splits its payload into messages based on the message lengths. All other fuzzing steps have been performed in a black-box manner.

BinFuzz Fuzzing The first phase is protocol learning from communication traces. It starts with message clustering. Its results are affected by two parameters: header size i.e. length of a message part used in clustering and the number of

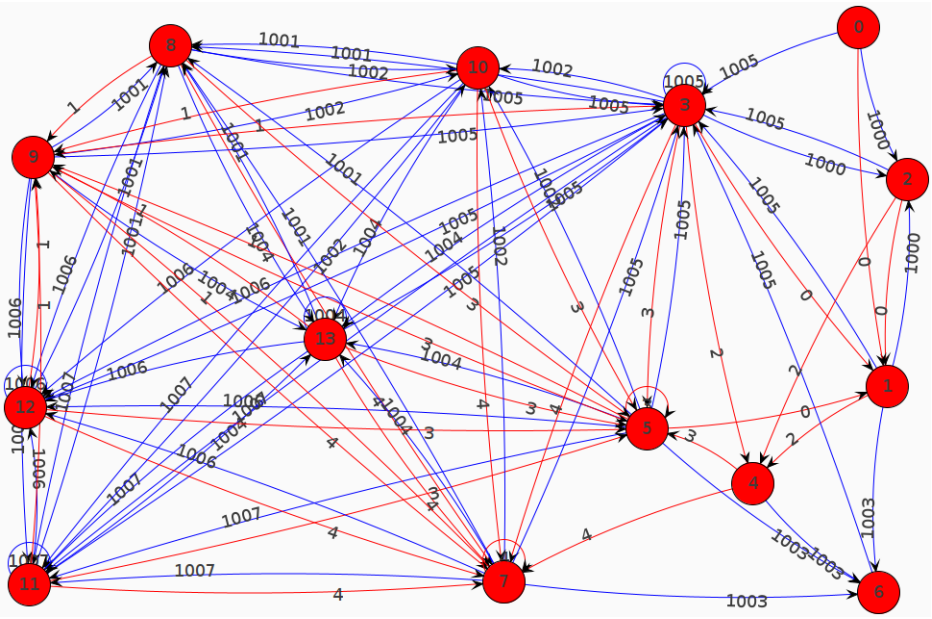


Fig. 4. FSM for the drone control protocol.

clusters. Note that the header size parameter does not have to be the actual size of a message header since this value is unknown due to black-box fuzzing, but an expected length to encounter enough information to distinguish different message types. We used a trial-and-error approach to find a useful parameter value. For the drone protocol, setting up a small header size below 100 bytes resulted in unusable clusters: they either have identical templates (and therefore cannot be distinguished during fuzzing) or templates with almost empty variable parts, while 125 bytes or 250 hex digits yielded several distinct clusters with templates having both fixed and variable parts. For the number of clusters, the parameter is chosen to avoid, on one hand, clusters with identical templates and, on the other hand, clusters with few messages; for the drone protocol, it is achieved with 7 and 8 clusters for messages to and from the drone, respectively.

For each cluster, a template has been generated, some of them are shown in Table 2. An FSM generated by BinFuzz is shown in Figure 4. Transitions are labeled with cluster IDs and the states they are connecting. Red transitions are for messages to the drone and the blue ones are for the messages from the drone. Unlike the FTP example, message direction does not alter after every single message, and it is not possible to separate states into sending messages only to or from the drone, respectively, but the majority of states have outgoing transitions for both message directions. Clustering and FSM construction have been run on a laptop with Intel Core i7-8650U CPU. Clustering of 93000 messages takes several seconds and the FSM construction takes about 3 minutes. However, the RAM

consumption of the FlexFringe tool was above 20Gb thus limiting the number of messages to be used for the model reconstruction.

During the second phase, we fuzzed messages in both directions, first messages sent to the drone, and later we switched to to fuzz messages towards the base station. The templates and the FSM have been reused in the second part. Independently from the fuzzing direction, all messages are intercepted and classified to one of the clusters in order to maintain the FSM state.

During the fuzzing, we kept the connection between the drone and the base station and periodically issued control commands. For the mutation, we tried all the mutation functions described in Section 3. We set up a 20% probability for messages to be passed unmodified. This way we can ensure that the protocol could periodically progress even if most of the fuzzed messages would be rejected. In addition, BinFuzz allows the user to stop and later restart fuzzing at any moment. Periodic pauses allowed us to check the recovery procedures of the protocol: if significant mutations of the messages prevent protocol execution, during the pause it is possible to validate that the protocol can resume its normal execution. BinFuzz logs intercepted messages, their clusters, and modified messages from the fuzzer. The drone software also writes the log including commands issued by the user and errors appearing in the application.

Fuzzing Results We tested various mutation functions and observed their impact on the drone's behavior. The first finding is that neither the drone nor the base station checks the integrity of the messages. The message header includes a checksum, yet randomly modified messages have been processed. This discovery simplified the protocol fuzzing as various message modifications could be applied without an automatic message drop by a receiver. On the other hand, random inserts at the beginning or end of messages have no detectable effect on the protocol execution, from which we assume that the first and the last bytes of the message are checked and everything outside of the message is ignored.

Fuzzing with random insert and random delete shows that start and end bytes are checked as well as message length. Modification of the message size without changing the message length field leads to the message drop. The logs show that the drone considers itself disconnected from the base station and starts a recovery procedure returning him to the home location.

Random replacements yield more interesting results. The messages are not immediately dropped due to the wrong length or lack of start or end byte. However, the majority of commands are not executed. Even with a relatively low probability of mutation (each position in the weakly (strongly) variable part of templates can be mutated with 5% (1%) probability resulting in the expected number of mutated hex digits or groups of digits equal to 2.8) approximately 80% of the commands fail to execute. Contrary to the "disconnected" scenario, with random replacement the drone continues the flight. In its log, there are entries "Control Lost" and "Connection reestablished" appearing every second. This leads to an issue reproducible on the emulator, that even being unable to execute any control command and acknowledging it the drone continues to maintain a

connection and wait for the commands without returning to its home location. It is possible to maintain the drone in this state until its batteries are depleted.

Fuzzing messages towards the base station yields similar results. Messages' beginning and end bytes as well as message length are checked, but not the checksum. The data received from the drone is not checked either: the user interface of the base station sometimes shows drone speed being faster than the speed of light or instant teleportation to a remote location. However, this issue is not protocol-related, and, potentially, not important for the base station software.

Overall, the fuzzing showed the lack of message integrity checks on the protocol level, though the integrity checks on the TCP level, to some extent, cover this issue. In addition, the drone might need some tuning for the detection of cases when recovery procedures are required.

4.3 Limitations and Potential Extensions

The utilization of the MITM fuzzer allows the tester to use messages close to real ones without any knowledge of a protocol and its message structure. However, some of the mechanisms can prevent easy mutations. Any integrity checks of messages such as checksums would fail after a random mutation. In our use case, these checks have not been implemented, which simplified the fuzzing procedure. On the plus side, templates generated from real messages can provide insight into checksum field positions as such positions are expected to be highly variable. The beginnings of drone protocol templates have two groups of highly variable positions (the first group can be seen in Table 2): one of them is a checksum position, while the exploration of bytes next to the second one showed that it was a part of the timestamp. Having the position guessed it is possible to recompute and update the checksum after the mutation.

A related complication for which it is harder to find a workaround is encryption. In the case of encrypted messages, it is hard to build relevant templates for the encrypted parts of messages. If such messages include an unencrypted header, it might be possible to use this header for fuzzing: to build a model of the protocol and perform fuzzing, yet only random mutations of the encrypted part are possible. To the best of our knowledge, only one approach to deal with encryption exists: provide knowledge about the encryption algorithm and keys to the fuzzer, which would decrypt a message and later re-encrypt the mutated version. This approach has been implemented in SecFuzz [27].

There are some limitations in our implementation that we have encountered during protocol fuzzing. Currently, the number of clusters for messages has to be selected by the user. We consider some heuristics to provide an automated selection. To measure the clustering quality we need to check the generated templates. If there are templates that have (almost) no variable part, then there is a high chance of having too many clusters and their number shall be reduced (unless there is a constant message in the protocol that appears in multiple traces). Another heuristic is to check if there are almost identical templates and to merge them. If some templates have almost no fixed part but a big weakly variable part, the tool could attempt to split such templates into smaller clusters.

Another limitation comes from comparing exact positions in the message when messages have different lengths. An extension of templates to support sequences of variable positions with non-constant length (e.g. instead of "----" with the length 4 use "-[4-8]" defining the length interval) might improve the generation of inputs. Finally, the FSM is constructed from real traces sending valid messages. Fuzzing might cause unexpected replies from the server that can potentially lead to an incorrect state. An interesting future direction for research is to build mechanisms for incorrect state detection and recovery.

5 Related Work

Fuzzing is a vulnerability discovery methodology [26] and its name was coined by Miller et al. in [16]. Over the years an immense number of fuzzers have been developed, a survey [12] mentioned more than 100 fuzzers. One of the most well-known fuzzer categorizations is the availability of the source code. White-box fuzzers (e.g. [6, 10, 18]) have full access to the source code, grey-box fuzzers (e.g. [30, 22]) have partial knowledge about the internals of the system under test, and black-box fuzzers (e.g. [1, 5]) can observe only inputs and outputs of the system. Another classification criterion is a type of system under test: many tools fuzz a single binary, and our use case was focused on the communication protocol.

Among the protocol fuzzers a big group consists of MITM fuzzers that can intercept and inject modified messages. This position provides an opportunity to obtain some information about the protocol from the exchanged messages. Moreover, real messages sent through the network can be used as a basis for input generation which has multiple benefits: (i) in case of insufficient knowledge about the message structure, mutations of a valid message would provide more meaningful inputs than ones generated from scratch; (ii) identifiers of the current message exchange such as session ID are present in the real message and, unless mutated, would be present in the fuzzer input. Protocol fuzzers can be classified into stateful and stateless. The latter processes each message separately reducing their effectiveness for complex protocols. ProxyFuzz [13] randomly mutates network traffic and, as a result, is inefficient for many protocols since the majority of packets become invalid. To avoid such inefficiency, many tools use templates for messages that allow smarter mutations. Polymorph [20] and mqtt_fuzzing [21] are stateless fuzzers that dissect messages with Tshark or Scapy. While these tools know message structures for a large number of protocols, their applicability to proprietary protocols cannot be guaranteed. SecFuzz [27] can perform fuzzing on encrypted messages, however it requires prior knowledge of the message structure, encryption keys, and encryption algorithm. PropFuzz[17] is another stateless fuzzer specialized in industrial protocols for Programmable Logic Controllers that use statistical computations to create message templates.

Stateful fuzzers use some kind of state automaton to model the protocol behavior. KiF [1] requires the automaton to be defined by the user. Restler [2] analyses API specification to infer the automaton. AutoFuzz [7] from which our fuzzer has been forked learns the automaton from network traffic. However, it

is limited to text-based protocols compatible with socks5, thus it is unsuitable for our case study. Pulsar [5] is another fuzzer that uses intercepted network messages to infer protocol states, yet their current implementation does not support non-textual protocols. In AutoFuzz and Pulsar, a single transition of a state automaton represents a pair of messages: one message sent from a first communication party and a return message from another. The drone control protocol we were testing cannot be modeled in such a manner as there is no evident dependency in the number of consequent messages in one direction. FitM [11] attempts to build the FSM and message templates during the fuzzing thus limiting its effectiveness in case of complex message structure.

An example of a non-MITM fuzzer is Mutiny Fuzzing Framework [25] which is a network fuzzer that operates by replaying mutated PCAPs. This stateless fuzzer is limited to replaying collected traces. Authors of [4] test several implementations of a TLS protocol. Users have to provide a list of abstractions of the protocol messages and the fuzzer attempts to rebuild a state machine by sending different messages from this list. Another TLS protocol fuzzer TLS-Attacker [24] also relies on prior knowledge of the message structure.

Another related group of fuzzers checks web applications. Examples of tools for web applications are WFuzz [14], ffuf [9], and WebFuzz [23]. Their input can be considered as protocol messages, and are generated from a string in which parts indicated with the word "FUZZ" are modifiable. However, they can learn neither a state machine nor a structure of the input.

Mutation testing measuring test suite quality can also be applied to protocols. DaMAT [29] modifies the exchanged data at runtime to detect the test suite shortcomings, however it requires knowledge of message structure. The approach can be used to measure fuzzing quality as proposed in [8].

6 Conclusion

Testing of proprietary protocols is a complex task as usually their source code and documentation are not available leaving the testers with black-box techniques. Fuzzing results primarily depend on input quality, for protocols it is important for generated messages to not be rejected immediately due to incorrect structure or an unexpected message type. For the fuzzing of the drone control protocol, many common assumptions are not valid and, therefore, we have built BinFuzz that works as a stateful MITM fuzzer. The preliminary construction of the FSM and message templates allows it to improve the fuzzing quality and likelihood of finding vulnerabilities. The application of the tool showed several issues in the protocol. There are multiple directions for future research and improvement of BinFuzz that include the application of advanced automated techniques for protocol reengineering, the addition of details to the templates, and the addition of new heuristics for the detection of message fields such as checksums.

Acknowledgements This work was supported by the Walloon Region of Belgium under the conventions n°8560 (DeepConstruct) and n°2110186 (Cyber Excellence).

References

1. Abdelnur, H.J., State, R., Festor, O.: Kif: a stateful sip fuzzer. In: Proceedings of the 1st international Conference on Principles, Systems and Applications of IP Telecommunications. pp. 47–56 (2007)
2. Atlidakis, V., Godefroid, P., Polishchuk, M.: Restler: Stateful rest api fuzzing. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). pp. 748–758. IEEE (2019)
3. Chang, O.: Taking the next step: Oss-fuzz in 2023 (2023), `https://security.googleblog.com/2023/02/taking-next-step-oss-fuzz-in-2023.html`
4. De Ruiter, J., Poll, E.: Protocol state fuzzing of {TLS} implementations. In: 24th USENIX Security Symposium (USENIX Security 15). pp. 193–206 (2015)
5. Gascon, H., Wressnegger, C., Yamaguchi, F., Arp, D., Rieck, K.: Pulsar: Stateful black-box fuzzing of proprietary network protocols. In: Security and Privacy in Communication Networks: 11th EAI International Conference, SecureComm 2015, Dallas, TX, USA, October 26-29, 2015, Proceedings 11. pp. 330–347. Springer (2015)
6. Godefroid, P., Levin, M.Y., Molnar, D.: Sage: whitebox fuzzing for security testing. Communications of the ACM **55**(3), 40–44 (2012)
7. Gorbunov, S., Rosenbloom, A.: Autofuzz: Automated network protocol fuzzing framework. Ijcsns **10**(8), 239 (2010)
8. Görz, P., Mathis, B., Hassler, K., Güler, E., Holz, T., Zeller, A., Gopinath, R.: Systematic assessment of fuzzers using mutation analysis. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 4535–4552 (2023)
9. Hoikkala, J.: ffuf - fuzz faster u fool. `https://github.com/ffuf/ffuf` (2021)
10. Kargén, U., Shahmehri, N.: Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. pp. 782–792 (2015)
11. Maier, D., Bittner, O., Munier, M., Beier, J.: Fitm: Binary-only coverage-guided fuzzing for stateful network protocols. In: Workshop on Binary Analysis Research (BAR). vol. 35 (2022)
12. Manès, V.J., Han, H., Han, C., Cha, S.K., Egele, M., Schwartz, E.J., Woo, M.: The art, science, and engineering of fuzzing: A survey. IEEE Transactions on Software Engineering **47**(11), 2312–2331 (2019)
13. Marcos, R.: Proxyfuzz – mitm network fuzzer in python. `https://www.darknet.org.uk/2007/06/proxyfuzz-mitm-network-fuzzer-in-python/` (2007)
14. Mendez, X.: Wfuzz - the web fuzzer. `https://github.com/xmendez/wfuzz` (2011)
15. Metzman, J., Arya, A., Szekeres, L.: Fuzzbench: Fuzzer benchmarking as a service (2020), `https://security.googleblog.com/2020/03/fuzzbench-fuzzer-benchmarking-as-service.html`
16. Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of unix utilities. Communications of the ACM **33**(12), 32–44 (1990)
17. Niedermaier, M., Fischer, F., von Bodisco, A.: Propfuzz—an it-security fuzzing framework for proprietary ics protocols. In: 2017 International conference on applied electronics (AE). pp. 1–4. IEEE (2017)
18. Peng, H., Shoshitaishvili, Y., Payer, M.: T-fuzz: fuzzing by program transformation. In: 2018 IEEE Symposium on Security and Privacy (SP). pp. 697–710. IEEE (2018)
19. Postel, J., Reynolds, J.: Rfc 959 - file transfer protocol (ftp). `https://www.ietf.org/rfc/rfc959.txt` (1985)

20. Ramos, S.H.: Polymorph: A real-time network packet manipulation framework. Exploit Database (2018)
21. Ramsauer, T.: mqtt_fuzzing. `https://github.com/ramsaut/mqtt_fuzzing` (2018)
22. Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., Bos, H.: Vuzzer: Application-aware evolutionary fuzzing. In: NDSS. vol. 17, pp. 1–14 (2017)
23. van Rooij, O., Charalambous, M.A., Kaizer, D., Papaevripides, M., Athanasopoulos, E.: webfuzz: Grey-box fuzzing for web applications. In: Computer Security–ESORICS 2021: 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4–8, 2021, Proceedings, Part I 26. pp. 152–172. Springer (2021)
24. Somorovsky, J.: Systematic fuzzing and testing of tls libraries. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 1492–1504 (2016)
25. Spadaro, J., Wyatt, L.: Mutiny fuzzing framework. `https://github.com/Cisco-Talos/mutiny-fuzzer` (2018)
26. Sutton, M., Greene, A., Amini, P.: Fuzzing: brute force vulnerability discovery. Pearson Education (2007)
27. Tsankov, P., Dashti, M.T., Basin, D.: Secfuzz: Fuzz-testing security protocols. In: 2012 7th International Workshop on Automation of Software Test (AST). pp. 1–7. IEEE (2012)
28. Verwer, S., Hammerschmidt, C.A.: Flexfringe: a passive automaton learning package. In: 2017 IEEE international conference on software maintenance and evolution (ICSME). pp. 638–642. IEEE (2017)
29. Viganò, E., Cornejo, O., Pastore, F., Briand, L.: Damat: A data-driven mutation analysis tool. In: 2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). pp. 165–169. IEEE (2023)
30. Zalewski, M.: American fuzzy lop. `http://lcamtuf.coredump.cx/afl/`
31. Zhang, T., Ramakrishnan, R., Livny, M.: Birch: A new data clustering algorithm and its applications. Data mining and knowledge discovery **1**, 141–182 (1997)