RIBDetector: an RFC-guided Inconsistency Bug Detecting Approach for Protocol Implementations

Jingting Chen[1,2], Feng Li[1,2,*], Mingjie Xu[1], Jianhua Zhou[1,2], Wei Huo[1,2]
[1]Institute of Information Engineering, Chinese Academy of Sciences, China
[2]School of Cyber Security,University of Chinese Academy of Sciences, China

***Abstract*—The implementations of network protocols must comply with rules described in their Request For Comments (RFC) Standards. Developers' misunderstanding or negligence of RFCs may bring in inconsistency bugs, which could further cause incorrect behaviors, interoperability issues, or critical security implications. Detecting such bugs is difficult as they usually result in silent erroneous effect. Prior work on RFC-directed inconsistency bug detection usually deal with a certain protocol or ad-hoc properties in RFCs. In this paper, we present RIBDetector, an approach focusing on statically and efficiently locating inconsistency bugs that could be triggered by hand-crafted network packets in protocol implementations. Given an implementation, its corresponding RFCs and a user-provided configuration file, our approach automatically extracts rules about packet format, state transition and error handling from RFCs into a uniform format which dictates condition checks that must be performed before taking particular operations. Then we leverage common programming conventions to identify corresponding locations of the conditions and operations in implementations and use a light-weight predominator-based algorithm to detect violations of RFC rules. We implemented a prototype of RIBDetector and demonstrated its efficacy by applying it on 14 implementations of 5 network protocols. For implementations varying in size from 1.5 to 141.3 KLOC, RIBDetector consumes 17.57 seconds on average to finish its analysis. We have detected 23 new inconsistency bugs, 6 of which are confirmed and fixed by developers.**

***Index Terms*—Network protocol, RFC, inconsistency bug**

I. Introduction

The implementations of network protocols must comply with properties described in their Request For Comments (RFC) Standards. For instance, RFC 2328 [1], the standard RFC of the Open Shortest Path First (OSPF) Protocol, dictates the formats of Link State Advertisement (LSA) messages, the correct operational behaviors when receiving specific types of network packets or events, and the actions to be taken when errors occur. Developers' misunderstanding or negligence of RFCs may bring in bugs during implementations. These bugs may further cause incorrect behaviors, interoperability issues, or critical security implications. They are referred to as inconsistency bugs or semantic bugs in previous literatures [2].

Consider the vulnerability CVE-2017-3224 in the OSPF implementations of Quagga where it doesn't prematurely age an LSA instance even if the instance has the MaxSequenceNumber as stipulated in RFC 2328. By exploiting this vulnerability, an attacker can erase or alter the routing tables of routers within the routing domain, causing denial of service or re-routing of traffic on the network. Inconsistency bugs in protocol implementations like CVE-2017-3224 may not always display externally erroneous effect like crash, thus they are difficult to detect automatically by applying well-known techniques like fuzzing.

Recent approaches, such as RFCcert [3] and TCP-fuzz [4], use rules extracted from RFC documents to improve the efficacy of differential testing and fuzzing. Although these approaches have successfully found quite a few real inconsistency bugs in SSL/TLS implementations and TCP stacks, respectively, it is difficult to apply them to deal with other protocol implementations, due to the richness of RFC descriptions and the diversity of implementations. Besides, some consistency bugs can only be triggered under specified options. Dynamic approaches may have to exhaustively test all possible options in order to find these bugs, which is time consuming. For efficiency, these approaches only consider ad-hoc rules during fuzzing, thus they may omit some crucial inconsistency bugs that could be exploited by attackers.

To overcome these shortages, we describe RIBDetector, an approach focusing on statically and efficiently locating inconsistency bugs that could be exploited by attackers via hand-crafted network packets in protocol implementations. Our investigation (cf. II-A) shows that protocol implementations non-compliant to RFC properties about packet decoding or processing may bring in incomplete validation check or improper response to network received packets, allowing attackers to construct elaborate crafted packets which can lead to deny of service, information leak and other security issues. This enable RIBDetector to restrict rule extraction and detection to three major types of packet-related RFC rules, including *packet format* rules, *state transition* rules and *error handling* rules (cf. II-B).

RFC descriptions for the three types of rules, as illustrated in Table I, usually contain either domain-specific nouns, such as the names of packets or states, or comparative phrases that involve constant values. Furthermore, they often correspond to condition checks as shown in Table I in source code implementations. Based on these findings, we employ a keyword-based strategy to identify target statements in RFC for rule extraction and leverage NLP to extract the three types of rules into a uniform format. To deal with the differential writing styles of RFC documents, we require users to provide a configuration file ($\sim$20 lines) for each protocol.

Comparing to dynamic approaches, our research is meant

* corresponding author

TABLE I: Examples of RFC rule descriptions and their implementations

Rule Type	Corresponding RFC description	Implementation (-/+: code before/after fixing inconsistency)

Packet Format	Example 1. *(from RFC2132, DHCP)*
	IP Address Lease Time:
	The code for this option is 51, and its length is 4.
	
	Rule 1: *chk_bf(length == 4, use(option))*
		// BusyBox 1.32 udhcp/dhcpc.c
		void udhcpc_main(char **argv) {
		  optionptr = packet→options ;
		  if optionptr[0] == DHCP_LEASE_TIME {
		+ if (optionptr[1] != 4) {
		        memcpy(&lease_seconds, optionptr+2, 4); }}

	Example 2: *(from RFC2328, OSPF)*
	12.1.5. Advertising Router:
	This field specifies the OSPF Router ID of the LSA's originator. For router-LSAs, this field is identical to the Link State ID field.
	
	Rule 2: *chk_bf((LS type == router-LSAs && Advertise Router==Link State ID), use(Advertise Router))*
		// Frrouting 7.5 ospfd/ospf_packet.c
		void ospf_ls_upd (struct ospf_lsa *lsa){
		+ if(&lsa→data→id != &lsa→data→adv_router){
		+ if (lsa→data→type == OSPF_ROUTER_LSA){
		+ DISCARD_LSA(lsa, 0); }}
		  current = ospf_lsa_lookup_by_header (oi→area,lsa→data);
		  }

State Transition	Example 3. *(from RFC4271, BGP)*
	OpenConfirm: If the local system receives a KEEPALIVE message (KeepAliveMsg(Event 26)), the local system:
	  - restarts the HoldTimer and
	  - changes its state to **Established**
	
	Rule 3:*chk_bf((state == OpenConfirm && event == KeepAliveMsg), set(state == Established))*
		// Bird 2.0.8 proto/pakcets.c
		void bgp_rx_update(struct bgp_conn *conn){
		  if (conn→state == BS_OPENCONFIRM){
		    bgp_conn_enter_established_state(conn);
		      return;
		    }
		  }

Error Handling	Example 4. *(from RFC 6286, BGP)*
	If the **BGP Identifier** field of the OPEN message is zero, or if it *is the same as* the BGP Identifier of the local BGP speaker and the message is from an internal peer, then the **Error Subcode** *is set to* "**Bad BGP Identifier**".
	
	Rule 4:*chk_bf(BGP Identifier == 0 || BGP Identifier == Local BGP Identifier), set(Error Subcode, Bad BGP identifier))*
		// OpenBGPD 6.9 bgpd/session.c
		int parse_open(struct peer *peer) {
		- if (ntohl(bgpid) == 0) {
		+ if (ntohl(bgpid) == 0 || (!peer→conf.ebgp &&
		          bgpid == conf→bgpid)){
		  session_notification(ERR_OPEN, ERR_OPEN_BGPID);
		  }}

to provide developers explicit inconsistency information, i.e. the location of inconsistency bugs in the source code implementations, in order to help them fix the bugs. To achieve this goal, our main challenge during detection is to identify the corresponding code for each RFC rule described in natural language, and check semantically equivalence between them. Our key intuition is that different implementations of the same protocol must follow common program conventions in order to obtain the expected functionality and interoperability. These conventions can help us to identify key structures implemented for packet parsing or processing, such as structures used to store the receiving packets or the current state of an FSM. Specifically, there are two major conventions that can be used to help identify structures used to store packets and states: 1) developers should always decode the network received packets based on the length and offset of each packet field described in RFC; 2) the current state of a finite state machine(FSM) should be checked before performing state transition and updated after that. Structures found based on these conventions may further help locate codes that access or check their types of variables as specified in the extracted RFC rules. The code locations form the analysis scope of violation detection.

Given a protocol implementation, its corresponding RFC document and a user-provided configuration file, RIBDetector automatically extracts rules for three types of rules mentioned above along with meta-info about packets, FSM states/events and error subcodes from RFC. Each rule is represented in such a uniform format $chk\_bf(Cond, Op)$, which indicates the condition check $Cond$ that must be performed before taking a packet field accessing, state transition or error notification operation Op. RIBDetector then uses meta-info to locate key structures in the implementations, including those designed for storing different types of network packets or the current state of an FSM, and consequently identifies the location of each rule-specified operation in the implementation. Finally, a light-weight predominator-based algorithm is performed to detect if the location is properly protected by the condition prescribed in the corresponding rule.

The contributions of our approach are summarized as follow:

1) We revise a uniform format for rules about package format, state transition and error handling extracted from RFCs, so that a general algorithm can be used to detect rule violations in protocol implementations.
2) We design RIBDetector, an approach which automatically identifies locations in source code implementations that should comply with RFC rules, and detects inconsistency bugs based on a light-weight predominator-based algorithm.
3) We implement a prototype of RIBDetector which supports C/C++ network protocol implementations and demonstrate its efficacy by applying it on the latest versions of 14 protocol implementations, discovering 23 inconsistency bugs, 6 of which have been confirmed and fixed by developers.

The rest of the paper is organized as follows. Section II describes the scope of our work and illustrates our approach using a real-world example. Section III presents the design and implementation of RIBDetector. Section IV evaluates its effectiveness and efficiency followed by a discussion of its limitations. Section V reviews the related work and Section VI concludes the paper.

II. Overview

A. A Motivating Example

We have investigated the triggering conditions for vulnerabilities in 3 popular network protocols, including OSPF, BGP and DHCP. The vulnerabilities are collected from the CVE (common vulnerabilities and exposures) vulnerability database [5], using its provided default search engine. For each protocol, we queried the database using the protocol name along with each of the following keywords: ”packet”, ”message” and the names of packet fields. The keyword-based search returned hundreds of results. We manually went through each returned result to ensure its triggering conditions while removing duplicate results. According to our statistics, almost 76% of the vulnerabilities discovered in the under-considering protocols can be triggered by elaborate crafted network packets. These vulnerabilities, once triggered, may consequently lead to denial of service or silent incorrect behaviors such as re-routing of network traffic. Nevertheless, most of the vulnerabilities would be avoided if their corresponding implementations complied with the respective RFC descriptions.

Figure 1 gives an example that illustrates a typical vulnerability caused by programmers' negligence of a packet format description in RFC. The vulnerability is known as CVE-2018-20679, which exists in the udhcp implementation of BusyBox before 1.30.0. In function *udhcpc_main*, the *DHCP_LEASE_TIME* field of the *packet* received at line 4 is copy to *lease_seconds* at line 13, without checking the length of the field. This vulnerability allows an attacker to inject a crafted DHCP packet whose *DHCP_LEASE_TIME* field length is less than its legal size. Successful exploitation could cause an out-of-bounds read at line 14, which may leak sensitive socket information from the stack.

As previously mentioned, this vulnerability would be avoided if the implementation strictly followed RFC 2131 [6], the standard RFC of DHCP, which stipulates both in graphics and text that *for Lease Time option, its length is 4* in Sec 9.2, page 25. The RFC document also defines 255 possible options in the DHCP packet header and describes how these options should be handled. In order to detect this vulnerability, dynamic approaches such as fuzzing or RFC-directed differential testing may have to exhaustively test all possible options. Besides, inconsistencies found by these approaches require users to manually locate the corresponding code.

B. Scope of this work

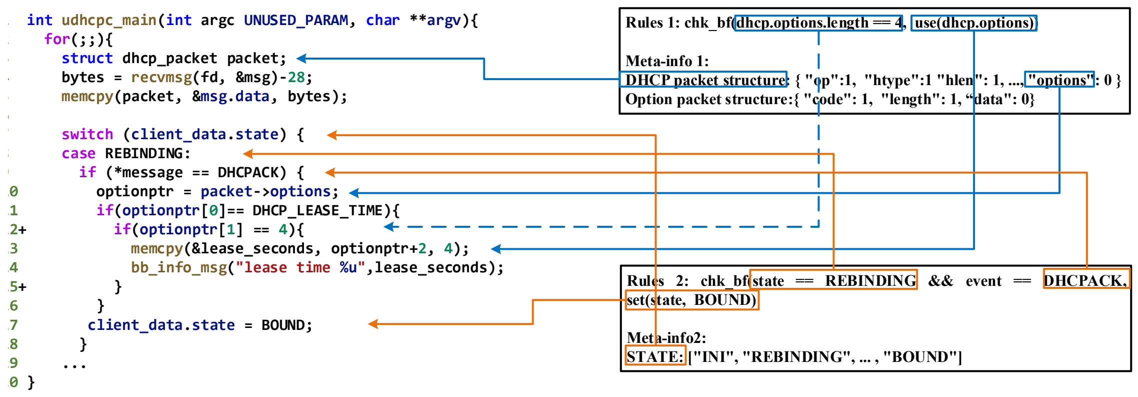
The RIBDetector approach presented in our research is meant to efficiently locate the root cause of inconsistency bugs that could be triggered by hand-crafted network packets in popular protocol implementations. For this purpose, we define such a 4-tuple threat model: $T = \{A, O, I, G\}$, where A stands for a set of actors, including a malicious packet generator (e.g., a script file) and a victim protocol service, O is a set of operations the adversary can perform, I is the information at his disposal and G includes his goal(s). In our research, the adversary's goal G is to disrupt the victim protocol service, either standalone or running on a network device, using the malicious packet generator, which is capable of generating and sending elaborate crafted packets, leading to unexpected behaviors such as DoS, information leak and other security or functionality issues; his information set I contains the inconsistency bugs in the victim's protocol implementations, especially those located in packet decoding or processing codes.

Table I illustrates three major types of packet-related rules described in RFC documents that RIBDetector considered during inconsistency bug detection. **Packet format rules** stipulate the size and value of each packet field, or how a field associates with others. **State transition rules** indicate the state transitions a network protocol should take in response to different sequence of network packets or events. **Error handling rules** describe the processing logic when an error is encountered. Recent approaches usually handle packet format rules as well, since their violations can bring in incomplete validation check of network packets (Example 1&2 in Table I), which may further cause security implications. However, they usually neglect the other two types of rules. Thus, they may omit inconsistencies involving illegal state transitions or improper responses to crafted packets (Example 3&4 in Table I). These inconsistencies may also lead to crucial issues.

C. Our approach

Motivated by the threat model, our research extracts all three types of rules mentioned above from RFC documents and detects their violations in source code implementations. Our main challenge is to identify the corresponding implementation for each RFC rule described in natural language, and check semantically equivalence between them. Natural language is expressive and admits multiple ways to express a single idea. Even for the same description, developers may adopt different structures or functions during implementation. Moreover, they may replace the RFC specified properties with functional equivalence implementations in order to obtain better performance and scalability.

In this paper, we address this key challenge based on **two observations**. **First**, all three types of rules have the same characteristics both in their RFC descriptions and the corresponding implementations. Specifically, a sentence describing such a rule always satisfies one of the following conditions: 1) it contains at least two specific keywords of the corresponding protocol, e.g., name of a packet field, state or error code; 2) it contains one specific keyword along with a modal keyword or comparative keyword. Table II illustrates the specific keywords, comparative keywords and modal keywords used in RFC 2328. As shown in Table I, these rules are implemented in source code as condition checks before accessing fields, performing state transitions or sending error notifications. Based on this observation, RIBDetector identifies target sentences for rule extraction based on the three types of keywords, and represents the extracted rules in such a



```
int udhcpc_main(int argc UNUSED_PARAM, char **argv){
  for(;;){
    struct dhcp_packet packet;
    bytes = recvmsg(fd, &msg)-28;
    memcpy(packet, &msg.data, bytes);

    switch (client_data.state) {
    case REBINDING:
      if (*message == DHCPACK) {
        optionptr = packet->options;
        if(optionptr[0]== DHCP_LEASE_TIME){
          if(optionptr[1] == 4){
            memcpy(&lease_seconds, optionptr+2, 4);
            bb_info_msg("lease time %u",lease_seconds);
          }
        }
       client_data.state = BOUND;
      }
    ...
}
```

Rules 1: chk_bf(dhcp.options.length == 4, use(dhcp.options))

Meta-info 1:
DHCP packet structure: { "op":1, "htype":1 "hlen": 1, ..., "options": 0 }
Option packet structure:{ "code": 1, "length": 1, “data": 0}

Rules 2: chk_bf(state == REBINDING && event == DHCPACK, set(state, BOUND))

Meta-info2:
STATE: ["INI", "REBINDING", ... , "BOUND"]

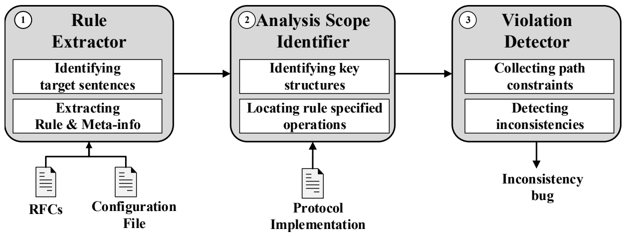
Fig. 1: Motivating Example

uniform format $chk\_bf(Cond, Op)$ which dictates condition checks ($Cond$) that must be performed before taking particular operations (Op). It then performs a general rule violation detecting process for the extracted rules. **Second**, there are common programming conventions that should be followed by different implementations of the same protocol. Otherwise, the implementations may fail to achieve the expected functionality or loss interoperability with others. For instance, developers may define their own packet structures which may not have the same field name or field order as those described in RFCs. However, they should always decode the network received packets based on the length and offset of each field specified by RFC. Similarly, developers may adopt different FSM implementations, but they should always check the current state of an FSM before state transition and update the state after that. RIBDetector employs these conventions to identify key structures used in implementations for packet parsing or processing. Such key structures include those designed for storing different types of packets or the current state of an FSM. They are further used to search for code that access or check their types of variables, and help locate the RFC-specified operations and conditions in the source code.

TABLE II: Examples of Keywords in RFC 2328

Type	Examples of Keywords
Specific Keyword	*(Packet fields)* LS age, Options, LS type, Link State ID, Advertising Router, ...
	(FSM States) Down, Attempt, Init, ExStart, Exchange, ...
Comparative Keyword	is identical to, is the same as, is less than, at least, greater than, ...
Modal Keyword	MUST, MUST NOT, SHOULD, ...

A workflow of RIBDetector is illustrated in Figure 2. It consists of three major steps. At the **first** step, its Rule Extractor identifies target sentences and extracts rules about packet format, state transition and error handling from RFCs based on a user-provided configuration file. Meta-info recording field offsets and orders in a network received packet, FSM states/events and error subcodes is extracted as well. At the **second** step, Analysis Scope Identifier employs the meta-info to identify key structures used for packet decoding or processing in a given protocol implementation. It then uses the structures to locate RFC-specified operations, such as packet field accessing, state transition or error notification, in the same implementation. These locations form the analysis scope of violation detection. **Finally**, a predominator-based rule violation detecting algorithm is used to check if each location in the analysis scope is properly guarded by condition prescribed in the respective rule, and report inconsistency bugs if it is not.



① Rule Extractor
Identifying target sentences
Extracting Rule & Meta-info
② Analysis Scope Identifier
Identifying key structures
Locating rule specified operations
③ Violation Detector
Collecting path constraints
Detecting inconsistencies
RFCs
Configuration File
Protocol Implementation
Inconsistency bug

Fig. 2: Architecture of RIBDetector

We use the vulnerability illustrated in Figure 1 to demonstrate how our approach works. Suppose we have already extracted the rules and meta-info shown in Figure 1 from RFC based on built-in keywords and a pre-written configuration file. To investigate if the example violates Rule 1, we need to first identify source code locations that access the *options* field of a DHCP packet, and then check if each of them can only be performed when condition (dhcp_packet.options.length == 4) is *True*. Meta-info 1 is used to identify the structure type of DHCP packet in the implementation. In this example, the optimal solution we are looking for is the one that share the same field sizes and orders with those described in Meta-info 1, so that a received DHCP packet can be straightly copied into the structure at line 5. Nested structures are flattened for proper comparison, and *dhcp_packet* turns out to be the only candidate. According to Meta-info 1, the offset of the *options* field in *dhcp_packet* structure (after flattening) is 280 bits. Therefore, variable *optionptr* defined at line 10 is the one corresponding to the field, and its use point at line 13 is consequently considered as the corresponding operation specified in Rule 1. Since line 13 is not guarded by the expected condition, an inconsistency bug is reported. Rule 2 dictates that a state transition should be performed from *REBINDING* to *BOUND* when receiving an *DHCPACK* event. In Figure 1, *client_data.state* is checked at line 7 and updated at line 17. Other switch-case blocks in the same udhcp revision also check and update variables coming from the same structure offset as client_data.state. Therefore, these variables, including client_data.state, are considered as variables used to store the current state and line 17 is regarded as one of the state transition operations. No violation is reported for Rule 2 as line 17 is properly guarded by the rule-specified conditions (lines 7&9).

III. The RIBDetector Approach

We have designed and implemented RIBDetector via three sub-analysis (Figure 2): Rule Extractor, Analysis Scope Identifier and Violation Detector.

A. Rule Extractor

Rule Extractor takes RFC documents along with a configuration file as input, and extracts rules and meta-info from the documents. Previous work extracts rules from RFCs according to the modal keywords (such as MUST, SHOULD) specified by RFC 2119. However, most existing RFCs are not written in strict accordance with RFC 2119. In other words, the three types of rules considered by RIBDetector may not be described using modal keywords. Based on the observations given in Section II-C, we consider a sentence in RFC as a rule extraction target if it satisfies one of the following conditions: 1) it contains at least two specific keywords of the corresponding protocol; 2) it contains one specific keyword along with a modal keyword or comparative keyword. Rules extracted from RFCs are represented as $chk\_bf(Cond, Op)$, where *Op* refers to a packet field accessing, state transition or error notification operation, and *Cond* indicates the conditions that must be checked before taking the operation.

Figure 3 illustrates how we identify target sentences and extract rules from RFC 2328. A fragment of the configuration file used for rule extraction is also presented. Users can modify the configuration file to specify the starting/ending section of rule extraction, and the writing format of specific keywords in the given RFC. In RFC 2328, specific keywords like "*Advertise_Router*" appear in either (sub)section captions or graphical representations after Section 12. In Figure 3, the configuration file provides a regular expression used to extract these keywords. During target sentence identification, pronouns like "*this field*" are automatically replaced with the specific keyword found in the same paragraphs or (sub)sections. Consider the second sentence in Example 2 of Table I. Figure 3 shows how the sentence is transformed into a uniform rule format via part-of-speech analysis and built-in regular regression replacing rules (e.g. replacing comparative keyword "*is identical to*" with "==").

Rule Extractor also considers rules hidden in graphical representation. The graphical representation of a packet structure describes the fields involved and their bit-width. For instance, in RFC 2328, the mark "|" indicates the beginning or end of a field, and the number of repeated "+-" between two "|" marks indicates the bit-width of a field. For the graphical representation shown in page 204 of RFC 2328, 8 packet format type of rules can be extracted from the figure, each corresponding to a field.

There are also rules that are not explicitly described in RFC but should be followed by developers during implementation. Intuitively, a size of packet must be no less than the total size of its mandatory fields, nor should it be less than the value of any length field. Revisit the packet structure described in page 204 of RFC 2328. It contains 8 fields with a total size of 20 bit-width. Besides, its last field is a length field. Therefore, two additional packet format type of rules should be followed during implementation:1) *chk_bf(len(ospf packet) >= 20, use(LSA Header))*; 2) *chk_bf(length <= len(OSPF packet), use(length))*.

Meta-info is extracted along with RFC rules. As illustrated in Figure 1, meta-info mainly contains two kinds of information: 1) the bit-width and offset of each mandatory field in various types of packets the protocol may receive; 2) names (and values) of FSM states, events or error subcodes. The meta-info will be used in following steps to help identify key structures in implementations whose types of variables appear in RFC-specified conditions and operations.

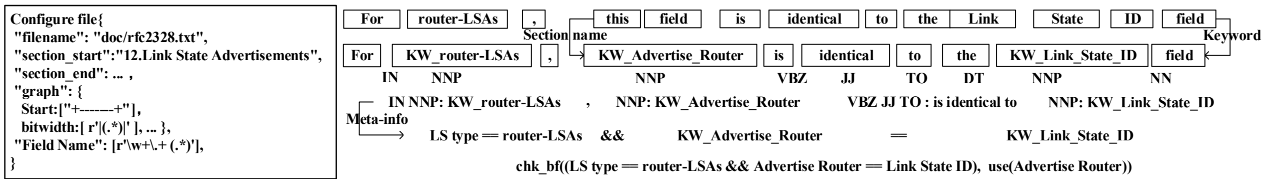
B. Analysis Scope Identifier

Analysis Scope Identifier takes a source code implementation, the former extracted RFC rules and meta-info as input, and identifies locations in the implementation that correspond to RFC-specified operations. These locations make up the analysis scope of violation detection.

As previously mentioned, developers may adopt different implementations for the same RFC description. In order to identify the RFC-specified operations in the given implementation, our Analysis Scope Identifier leverages meta-info extracted by Rule Extractor to help locate key structures (especially those used to store packets or states) whose types of variables are specified to be accessed or checked by RFC rules.

Figure 4 illustrates two typical implementations of packet structures and their corresponding decoding logic. As we can see, if a packet structure shares the same field sizes and orders with RFC specifications, the decoding logic can straightly copy or assign the network received packet to the structure (as shown in Figure 1 line 5, and Figure 4a line 12); Otherwise, the decoding logic should extract each field from the network received packet based on its length and offset specified in the corresponding RFC (as shown Figure 4b).

Based on this finding, we employ the following heuristic strategies to identify packet structures in a given implementa-



```
Configure file{
 "filename": "doc/rfc2328.txt",
 "section_start":"12.Link State Advertisements",
 "section_end": ... ,
 "graph": {
  Start:["+-------+"],
  bitwidth:[ r'|(.*)|' ], ... },
 "Field Name": [r'\w+\.+ (.*)'],
}
```

Fig. 3: An example of RFC rule extraction

tion. Let *pkt_meta* denote meta-info recording the field orders and their sizes in a certain type of packet (denoted by *mpkt*) whose fields appear in at least one of the extracted RFC rules. Analysis Scope Identifier searches for its corresponding packet structure in the given implementation as follow: 1) it collects all structures from the implementation and filters out those whose size and field number are smaller than those recorded in *pkt_meta*. 2) For the remaining structures, those sharing the same mandatory field sizes and orders with *mpkt* is inspected first. If a structure is used for type casting or as the destination parameter type of any memory copy function, it is added to the candidate set of *mpkt*. 3) If the candidate set is empty, continue to process structures that do not follow the field order of *mpkt* but containing all mandatory fields. If a structure whose type of variables are most frequently used as the base address of memory store statements, and the set of the offset values collected from these statements is equivalent to the set of field offsets recorded in *pkt_meta*, then the structure is added to the candidate set. If there is more than one candidate for *mpkt*, we require user efforts to identify the correct structure. Once the packet structure of *mpkt* is found, code accessing the rule-specified field is added to the analysis scope of the following step.

```
struct lsa_header{
  uint16_t ls_age;
  uint8_t options;
  uint8_t type;
  struct in_addr id;
  struct in_addr adv_router;
  uint32_t ls_seqnum;
  uint16_t checksum;
  uint16_t length;
};
void ospf_ls_upd(struct stream *s){
  struct lsa_header *lsah = (struct lsa_header*)
          (s);
  ...
}
```

(a) Code snippet of Frrouting 7.5-OSPF

```
void bgp_rx_open(byte *pkt, uint len){
  asn = get_u16(pkt+20);
  hold = get_u16(pkt+22);
  id = get_u32(pkt+24);
}
```

(b) Code snippet of Bird 2.0.8-BGP

Fig. 4: Packet structures in different implementations

Figure 5 illustrates two typical implementations of state transition logic and their corresponding structures. As we can see, state transitions can be implemented as either a global constant array indexing by the current state and a newly received event (as shown in Figure 5a), or a series of switch-case blocks scattering in different functions. In both scenarios, developers would maintain a structure to store the current state of an FSM. The structure would be accessed before state transition and updated after that. Such a structure or variable can be used as the clue to locate state transition code snippets.

We employ the following heuristic strategies to identify state transition operations in a given implementation. Let *fsm_meta* denote meta-info recording all states and events contained in an FSM of the corresponding protocol. Analysis Scope Identifier searches for its corresponding structures and logic in the given implementation as follow: 1) it collects all one-/two-dimension global arrays and filters out those whose element number of each dimension is smaller than the state number in *fsm_meta*. 2) For each remaining array, search for its use points and record all variables used as array index. If such a use point is indexed by a constant value or if it is control dependent on conditions comparing each array index variable with certain constant values, record the constant values related to each dimension of the array respectively. If any of the constant value sets is smaller than the number of states recorded in *fsm_meta*, the array will also be filtered. 3) For the remaining arrays, the one that has the closest element numbers and constant value sets as those recorded for states and events in *fsm_meta* is added into the candidate set of *fsm_meta*. Its use points are identified as state transition operations. Variables recorded at the use points are considered as either a current state variable or a received event variable according to the sets of constant values each of them is compared with. 4) If the candidate set is empty, continue to process all switch-case blocks and filter out those that satisfy at least one of the following conditions: variable used as the switch condition is not updated in the block, no common function is called by each case branch except for the default branch. 5) The remaining blocks are classified by variables used as their switch conditions: those sharing the same structure type of variables are divided into the same set. For each set, we collect constant values that the switch variable is compared with and the common functions called by each case branch. The one with the closest constant value set as the set of states recorded in *fsm_meta* and the largest number of common functions is considered as the set of state transition code snippets. Among these code snippets, all definition points for the variables used in the switch condition and the callsites of common functions where the variable is passed as parameters are considered as state transition operations, and the variables used as switch conditions is considered as those storing the current states of the under-considering FSM.

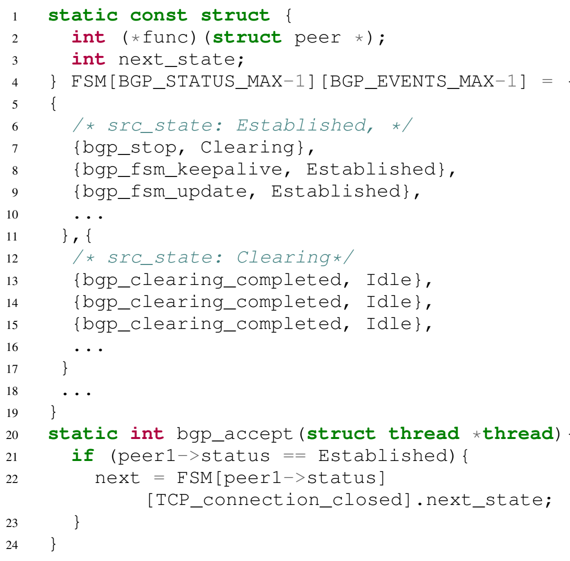
After locating the state transition operations, we extract the implemented FSM and checked its equivalence with RFC specifications. However, there are always gaps between implementations and specifications. Specifically, developers may divide a certain RFC state into multiple substates or replace a set of RFC events that trigger the same state transition with one aggregate event, in order to obtain clearer code logic or better performance. Therefore, we should further map each source code state/event to its corresponding RFC state/event before performing violation detection.

Our intuition is that developers will not deliberately violate RFC, thus their modifications on FSMs should be functional equivalent to those specified in RFC. Based on this assumption, we leverage a greedy strategy to find corresponding RFC states for as many source code states as possible by comparing their incoming/outgoing transitions numbers. Transitions triggered by different events between the same pair of states are counted once. For source code states that can not be mapped to any RFC state, we inspect if it can be merged with other states. Specifically, the source and destination of a state transition (denoted by *t*) can be merged if they satisfy the following conditions: the source state has no other outgoing edge except for *t*, and the destination has no other incoming edge except for *t*. In Figure 5a, the Frrouting implementation of BGP brings in a new state *Clearing* which is not described in RFC. The new state can be merged with *Idle* as the two states satisfy the above conditions. Identifying structures and operations for error handling rules are similar to that of state transition rules. It is not described in this paper due to space limitation.

C. Violation Detector

Violation Detector detects inconsistency bugs based on the RFC rules and the analysis scope evaluated by former steps. For each location corresponding to an RFC-specified operation in the under-considering protocol implementation, we need to detect if 1) the condition specified in the same RFC rule is checked on each possible path from entry to the location, and 2) the condition is $True$ on each path. If either of the condition is not satisfied, an inconsistency bug should be reported.

We leverages a predominator-based light-weight algorithm to detect such violations. In the beginning, Control Flow Graph (CFG) and Dominator Tree are constructed for each function in the implementation. Dominator tree represents the dominating relationship between CFG nodes. Node *n1* dominates node *n2* if all paths from entry to node *n2* includes node *n1* [7]. For each under-considering source code location (denoted by *op_loc*), we collect all definition points for variables used by the corresponding RFC-specified condition, based on key structures identified by former steps. Violation Detector then applies a reach-definition analysis to find condition checks in the implementation using any of the variables. For each of the conditions (denoted by *cond*), we enumerate all possible paths from the nearest predominator of *op_loc*, which is also the node in CFG that *cond* is control dependent on, to *op_loc* and collect constraints on each path. If the conjunction of the constraints and the negation of RFC-specified condition is satisfiable, it means that there is at least one path in the implementation which does not comply with the under-considering RFC rule; thus, a consistency bug should be reported at *op_loc*. In order to fix an inconsistency bug, developers can add the required condition check at the program point right before *op_loc*.


```
static const struct {
  int (*func)(struct peer *);
  int next_state;
} FSM[BGP_STATUS_MAX-1][BGP_EVENTS_MAX-1] = {
{
  /* src_state: Established, */
  {bgp_stop, Clearing},
  {bgp_fsm_keepalive, Established},
  {bgp_fsm_update, Established},
  ...
 },{
  /* src_state: Clearing*/
  {bgp_clearing_completed, Idle},
  {bgp_clearing_completed, Idle},
  {bgp_clearing_completed, Idle},
  ...
 }
 ...
}
static int bgp_accept(struct thread *thread){
  if (peer1->status == Established){
    next = FSM[peer1->status]
        [TCP_connection_closed].next_state;
  }
}
```
(a) Code snippet of Frrouting 7.5-BGP

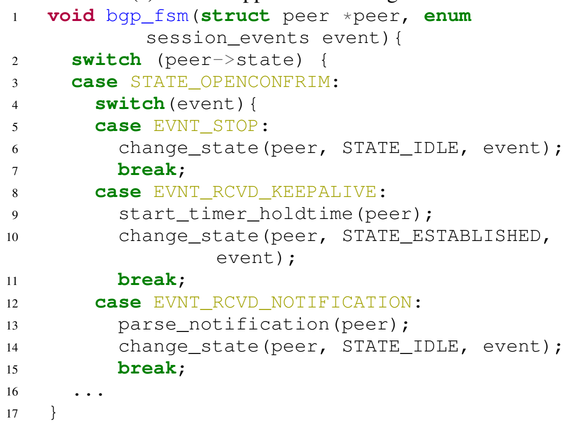

```
void bgp_fsm(struct peer *peer, enum
        session_events event){
  switch (peer->state) {
  case STATE_OPENCONFRIM:
    switch(event){
    case EVNT_STOP:
      change_state(peer, STATE_IDLE, event);
      break;
    case EVNT_RCVD_KEEPALIVE:
      start_timer_holdtime(peer);
      change_state(peer, STATE_ESTABLISHED,
          event);
      break;
    case EVNT_RCVD_NOTIFICATION:
      parse_notification(peer);
      change_state(peer, STATE_IDLE, event);
      break;
  ...
}
```
(b) Code snippet of OpenBSD 6.8-BGP

Fig. 5: State transition logic in different implementations

IV. Evaluation

We have implemented RIBDetector based on NLTK [8], LLVM [9] and z3 solver [10]. Our implementation is about 2.1 KLOC of Python code and 4.9 KLOC of C++ code. In this section, we will introduce the experimental setup, an overall evaluation of effectiveness and efficiency followed by detailed studies of new inconsistency bugs found by RIBDetector.

A. Setup

We applied RIBDetector to several implementations of 5 network protocols. Border Gateway Protocol (BGP) is an inter-Autonomous System routing protocol that exchanges network reachability information and provides communications between two autonomous systems. Open Shortest Path First (OSPF) is a link-state routing protocol designed to be run internal to a single Autonomous System. Routing Information Protocol (RIP) is a dynamic routing protocol which uses hop count as a routing metric to find the best path between the source and the destination network. Enhanced Interior Gateway Routing Protocol (EIGRP) is an advanced dynamic routing protocol that is used for routing decisions and configuration on the routers. Dynamic Host Configuration Protocol(DHCP) is a client/server protocol that allows hosts to obtain required TCP/IP configuration information from a DHCP server.

TABLE III: Five Network Protocols under Testing

Protocol	RFC Documents	Implementation
BGP	RFC 4271, RFC 6608 [11] and RFC 4486 [12]	Bird 2.0.8 [13]
		OpenBSD 6.8 [14]
		Frrouting 7.5 [15]
OSPF	RFC 2328	Bird 2.0.8
		Frrouting 7.5
		OpenBSD 6.8
EIGRP	RFC 7868 [16]	Frrouting 7.5
		OpenBSD 6.8
RIP	RFC 2435 [17], RFC 4822 [18]	Bird 2.0.8
		Frrouting 7.5
		OpenBSD 6.8
DHCP	RFC 2131 [19], RFC 2132 [6]	ISC DHCP 4.4.2 [20]
		BusyBox 1.34.0 [21]
		DHCPCD 9.4.0 [22]

We obtained a total of 14 implementations of these protocols shown in Table III. For our demonstration, we consider the latest versions of these implementations. We also evaluate RIBDetector on existing vulnerabilities. All of our evaluations are running on an Intel(R) Xeon(R) Gold 5218 processor with 64GB of memory.

B. Effectiveness

Rules Extracted from RFCs. Table IV statistics rules extracted from the RFC Standards of each protocol shown in Table III. Overall, RIBDetector extracts 395 rules from 9 RFC documents, which covers all rule extraction targets that satisfy the conditions described in Section III-A. RIP has the least rule number as it is the only stateless protocol among the 5 protocols. Thus, its Standard RFC does not involve description of state transition property like others. DHCP has a sophisticated packet structure that involves a series of options, making its packet format rule set significantly larger than those of other protocols.

Structures and Operations identified for Violation Detection. Table V summarizes the number of key structures and rule-specified operations located by Analysis Scope Identifier for each implementation. For key structures, Columns *Found* and *Missed* record the numbers of structures we successfully found and those we failed to locate, respectively. Data in parentheses represent the number of false positives. For Bird 2.0.8-OSPF and OpenBSD 6.8-OSPF, each of the missed structures is implemented as two individual structures. For Bird 2.0.8-BGP, DHCPD 9.4.0 and ISC DHCP 4.4.2, no corresponding structures or variables are defined for FSM events. These two scenarios are not supported by our heuristic strategies yet. For rule-specified operations, Columns *Found* lists the number of operations whose corresponding implementations are found. RFC rules whose operations are not found in respective implementations are further divided into two categories: those failed to be located by RIBDetector (Column *Missed*) due to the missed key structures, and those remain unimplemented in the source code (Column *UnImpl*). For rules whose operations are not implemented, RIBDetector generates no report for packet format rules, but will consider the unimplemented state transition or error handling rules as inconsistencies because they could bring in functionality or interoperability issues.

TABLE IV: Statistics of Rules Extracted from RFCs

Protocol	Rules	Packet Fmt	State Trans	Err Handling
BGP	**146**	17	98	31
OSPF	**116**	17	99	-
RIP	**6**	6	-	-
EIGRP	**35**	9	26	-
DHCP	**92**	73	19	-
Total / %	**395**	30.9%	61.3%	7.8%

TABLE V: Structures and Operations Identified based on RFC Rules and Meta-info

Implementation	Key Structures		Operation		
	Found(FP)	Missed	Found	Missed	UnImpl
Bird 2.0.8-BGP	7	**1**	37	**98**	22
Frrouting 7.5-BGP	8 (1)	0	138	0	8
OpenBSD 6.8-BGP	7	0	131	0	13
Bird 2.0.8-OSPF	7	**4**	97	**4**	0
Frrouting 7.5-OSPF	15 (4)	0	111	0	5
OpenBSD 6.8-OSPF	10	**1**	115	**1**	0
Bird 2.0.8-RIP	3	0	6	0	2
Frrouting 7.5-RIP	3	0	4	0	2
OpenBSD 6.8-RIP	3	0	6	0	0
Frrouting 7.5-EIGRP	15	0	30	0	5
OpenBSD 6.8-EIGRP	15	0	33	0	2
DHCPCD 9.4.0	4	**1**	19	**19**	54
ISC DHCP 4.4.2	4	**1**	51	**19**	15
BusyBox 1.34.0	4	0	60	0	32

Inconsistency Bugs found in latest implementations. RIBDetector has successfully found 23 inconsistency bugs (listed in Table VI) in the latest versions of 7 routing protocol implementations given in Table III. In Column *New Bugs*, data separated by '/' in parentheses represent the numbers of new bugs caused by violating packet format rules, state transition rules and error handling rules, respectively, in each implementation. By carefully inspecting the newly discovered bugs, we found that 5 (*) of them may lead to security implications, others may cause interoperability (†) or functionality issues. We have reported majority of the bugs to their original developers, and 6 of them have already been confirmed and patched. Further discussion about these newly discovered inconsistency bugs will be given in section IV-D.

There are also 21 false positives reported by RIBDetector (Data presented in parentheses in Column *Reports*). Half of them are due to the conservation of static analysis. Others are because Bird further implements RFC 7606 [23], which revises the error subcodes defined for BGP *Update* messages in RFC 4271. The revised RFC withdraws 9 error subcodes, leading to 9 unimplemented error handling operators. As mentioned above, RIBDetector reports them as inconsistency bugs. These false positive can be eliminated by applying the revised RFC for RIBDetector.

TABLE VI: New Bugs Detected by RIBDetector

Implementation	**Reports(FP)**	**New Bugs**
Bird 2.0.8-BGP	19(12)	**7** (1 / 1* / 5†)
Frrouting 7.5-BGP	8(3)	**5** (0 / 1 / 4†)
OpenBSD 6.8-BGP	5(2)	**3** (0 / 1† / 2†)
Bird 2.0.8-OSPF	2(2)	0 (0 / 0 / -)
OpenBSD 6.8-OSPF	1(0)	**1** (1 / 0 / -)
Frrouting 7.5-RIP	2(0)	**2** (2 / - / -)
OpenBSD 6.8-EIGRP	1(0)	**1** (1 / 0 / -)
Frrouting 7.5-EIGRP	6(2)	**4** (4* / 0 / -)
Total	44(21)	**23**(8 / 3 / 11)

Finding Existing Vulnerabilities. We evaluate RIBDetector using existing vulnerabilities reported in the past 5 years. The vulnerabilities are selected from CVE database using keywords "packet/message", "state machine" or specific keywords extracted from RFCs of respective protocols. An existing vulnerability is selected for further investigation if its corresponding patch is available and the patch contains at least one of the specific keywords. In the end, a total of 11 existing vulnerabilities have been selected, 3 for BGP and 9 for DHCP.

RIBDetector has detected 10 out of the 12 vulnerabilities. For the two remaining vulnerabilities in BusyBox implementation, we failed to identify the field accessing operations whose condition checks specified by RFC are violated as the corresponding field offsets are obtained via complex calculations.

C. *Efficiency*

Table VII reports time and memory usage in detecting the latest version of 14 implementations using RIBDetector. Column 3 gives the time spent on rule extraction. In practice, rule extraction is a one-time effort as the rules and meta-info can be reused by different implementations of the same protocol. Column 4 gives the time for identifying analysis scope, including the time spent on generating LLVM IR and CFGs. Column 5 gives the time for detecting rule violations. For protocol implementations varying in size from 1.5 to 141.3 KLOC, RIBDetector consumes 17.57 seconds on average to finish its analysis with 0.4 GB maximum memory usage.

D. *Case Studies of Inconsistency Bugs*

We now describe the new inconsistency bugs discovered by RIBDetector in details. Due to space limitation, we only give details for representatives of bugs with security implications.

Incomplete handling of connection collisions. RIBDetector finds a state transition rule violation in the latest BGP implementations of Bird. A TCP connection can be formed

TABLE VII: Time and Memory Usage of RIBDetector

Implementation	**#LOC**	**Ext(s)**	**Ident(s)**	**Det(s)**	**Total(s)**	**Mem(GB)**
Bird 2.0.8 -BGP	6.0K		8.85	23.09	32.67	0.19
Frrouting 7.5-BGP	89.2K	0.73	16.00	50.63	67.36	0.26
OpenBSD 6.8 -BGP	30.7K		3.67	16.61	21.02	0.13
Bird 2.0.8 -OSPF	9.1K		8.43	21.50	30.46	0.13
Frrouting 7.5-OSPF	42.4K	0.53	5.51	13.63	19.67	0.13
OpenBSD6.9-OSPF	14.4k		6.81	4.32	11.66	0.13
Bird 2.0.8-RIP	1.5K		7.82	18.08	26.16	0.13
Frrouting 7.5-RIP	7.8K	0.27	1.12	2.50	3.89	0.06
OpenBSD 6.8-RIP	6.8k		2.04	1.99	4.03	0.06
Frrouting 7.5-EIGRP	10.5K		1.34	3.19	4.95	0.06
OpenBSD6.9-EIGRP	10.6K	0.42	0.74	0.57	1.73	0.06
DHCPD 9.4.0	40.3K		1.48	1.59	3.37	0.1
ISC DHCP 4.4.2	141.3K	0.30	7.65	4.17	12.12	0.4
BusyBox 1.34.0	7.8K		3.01	3.35	6.66	0.1
AVG	29.8K	0.49	5.32	11.8	17.57	0.14

between a pair of BGP peers to exchange messages, and a state machine is instantiated for each connection. If a connection collision has occurred between a paired peers, only one of them should be retained. If a connection is determined to be the one that must be closed, an *OpenCollisionDump* event is signaled to its state machine. According to RFC 4271, if such an event is received in the *OpenSent* state, the state machine should drop the connection and change its state to Idle. RIBDetector extracts following rule from the above mentioned description: *chk_bf((state == OpenSent && event == OpenCollisionDump), set(state, Idle))*. As illustrated in Figure 6, when receiving an *OpenCollisionDump* event under *OpenSent*, the BGP implementation of Bird 2.0.8 performs no state transition, which violates the rule. When the hold timer of the connection expired, all connections between the same paired of peers would be closed. RIBDetector reports the violation as an inconsistency bug. An attacker can easily exploit this bug to break down connections by continuously establishing incompletely BGP connections that stay in *OpenSent* states, causing a denial of service.

```
static void bgp_rx_open(struct bgp_conn *conn,
  byte *pkt, uint len){
  switch (other->state){
  /* A rule violation of chk_bf((state ==
    opensent && event == opencollistiondump),
    set(state, Idle)) */
  case BS_OPENSENT:
  case BS_CLOSE:
    break;
  }
  ...
}
```

Fig. 6: An inconsistency bug detected by RIBDetector in Bird 2.0.8

Accessing TLV field without checking size. RIBDetector finds two rules violation in the latest EIGRP implementation of Frrouting. According to RFC 7868, an EIGRP packet contains a header, followed by a set of variable-length fields consisting of Type/Length/Value (TLV) triplets. The *Parameter* TLV of *HELLO* packet contains 7 fix-length fields with a total size of 12 bit-width. RIBDetector extracts a packet format rule for the *Parameter* TLV that the packet size should be greater than the total size of its 7 fields, i.e. *chk_bf(12 <= len(Hello packet), use(Parameter tlv))*. However, as illustrated in Figure 7, the EIGRP implementation of Frrouting 7.5 accesses the TLV field of a *HELLO* packet without checking its size, resulting in an out-of-bound memory access. This bug can cause security implication as unauthorized clients may obtain sensitive data beyond the TLV field. A similar violation of rule extracted for the *Version* TLV is also detected by RIBDetector in the same implementation.

```
static struct eigrp_neighbor *
eigrp_hello_parameter_decode(struct
eigrp_neighbor *nbr,struct eigrp_tlv_hdr_type
*tlv){
  /* A rule violation of chk_bf(len(12 <= Hello
  packet), use(parameter tlv))*/
  struct TLV_Parameter_Type *param = (struct
  TLV_Parameter_Type *)tlv;

  nbr->K1 = param->K1;
  nbr->K2 = param->K2;
  ...
  nbr->v_holddown = ntohs(param->hold_time);
}
```

Fig. 7: An inconsistency bug detected by RIBDetector in Frrouting 7.5

E. Limitations

RIBDetector has the following limitations. First, it requires manual efforts to kick off the detection: one needs to learn the writing style of an RFC document and writes a configuration file that describes the format of specified keywords and meta-info in the document. Although the writing styles of RFCs rarely changes across the protocols considered in our evaluations, they may not be generalized to other popular network protocols. Second, RIBDetector only considers packet-related RFC rules. In future work, we would like to deal with more sophisticated RFC rules and employ deep learning techniques to make our approach get rid of human efforts. Finally, like other static approaches, RIBDetector suffers from false positives. It can be extended with existing techniques like target-directed fuzzing to improve precision.

V. Related Work

In this section, we will give a brief summary of previous work closely related to RIBDetector.

Protocol analysis. Prior work on detecting protocol implementation bugs based on RFC focuses on either a certain protocol or ad-hoc properties. SymbexNet [24] presents an approach that combines symbolic execution and rule-based specifications to detect various types of flaws in network protocol implementations. It requires users to extract packet related rules from RFC in order to generate test packets for Zeroconf, DHCP and DNS protocols. CHORIN [25] extracts the implemented FSMs by symbolically exploring the protocol implementations of DHCP and TLS. It then uses an off-the-shelf model checker to detect whether the extracted FSMs violate the given temporal properties. The properties are also extracted from RFC manually. RFCcert [3] extracts rules of certificates according to the RFC standards of SSL/TLS. It then uses the obtained certificates to generate testcases and uses differential testing to finding bugs in SSL/TLS implementations. Sage [26] leverages NLP to extract packet rules from the RFC of ICMP and then uses them to generate code. The automatically generated ICMP code can interoperate perfectly with Linux implementations. RIBDetector extracts three types of packet-related RFC rules in order to detect inconsistency bugs that may bring in elaborate crafted packets, leading to security or functionality issues.

BaseSpec [27] performs comparative analysis of message structures implemented in baseband firmware and cellular specifications defined by 3GPP. MPInspector [28] evaluates the security of IoT Message Protocol implementations by combining model learning with formal analysis. In future work, we also plan to extend RIBDetector by dealing with specifications beyond RFCs and protocol implementations without source code.

Protocol Fuzzing. There is a large body of research on fuzzing protocol implementations [29]–[41]. AFLNet [34] takes a mutational approach and uses response codes as state-feedback to guide the fuzzing process of FTP and RSTP protocols. TLS-Attacker [33] presents a two-stage fuzzing approach to evaluate TLS server behaviors. It introduces cryptographic fuzzing for known vulnerabilities and systematically modifies message variables and flows to trigger bugs. The fuzzing approach is also proven effective for SSH [42], TCP [43] and OpenVPN [37]. Comparing to these dynamic approaches, RIBDetector is capable to identify the location of inconsistency bugs, especially those that may result in silent incorrect behaviors.

VI. Conclusion

We present RIBDetector, an approach focusing on efficiently locating inconsistency bugs that could be triggered by hand-crafted network packets in protocol implementations based on RFC. RIBDetector achieves its efficiency and effectiveness by extracting three types of packet-related rules into a uniform format and designing a general algorithm to detect their violations in implementations. Our evaluation shows that RIBDetector can finish its analysis in 17.57 seconds on average for 14 protocol implementations varying in size from 1.5 to 141.3 KLOC, finding 23 new inconsistency bugs, 6 of which have be confirmed and fixed by developers. RIBDetector is available at https://github.com/melissa-cjt/RIBDetector.

Acknowledgment

We thank anonymous reviewers for their comprehensive feedback. This research was partly supported by the Chinese National Natural Science Foundation (U1836209, 62032010, 61802394), the Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences, and Beijing Key Laboratory of Network security and Protection Technology.

References

[1] J. Moy, "Ospf version 2," https://www.rfc-editor.org/rfc/rfc2328/, 1998.

[2] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," vol. 19, no. 6, p. 1665–1705, 2014. [Online]. Available: https://doi.org/10.1007/s10664-013-9258-8

[3] C. Chen, C. Tian, Z. Duan, and L. Zhao, "Rfc-directed differential testing of certificate validation in ssl/tls implementations," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. Association for Computing Machinery, 2018, p. 859–870. [Online]. Available: https://doi.org/10.1145/3180155.3180226

[4] Y.-H. Zou, J.-J. Bai, J. Zhou, J. Tan, C. Qin, and S.-M. Hu, "Tcp-fuzz: Detecting memory and semantic bugs in TCP stacks with fuzzing," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 489–502. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/zou

[5] "Common vulnerabilities and exposures," https://cve.mitre.org/index.html/, 2021.

[6] R. D. S. Alexander, "Dhcp options and bootp vendor extensions," https://www.rfc-editor.org/rfc/rfc2131/, 1997.

[7] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[8] "Natural language processing with python," http://www.nltk.org/, 2009.

[9] "The llvm compiler infrastructure," https://llvm.org/, 2009.

[10] "The z3 theorem prover," https://github.com/Z3Prover/z3, 2020.

[11] M. C. J. Dong, "Subcodes for bgp finite state machine error," https://www.rfc-editor.org/rfc/rfc6608, 2012.

[12] F. T. E. Chen, Cisco Systems V. Gillet, "Subcodes for bgp cease notification message," https://www.rfc-editor.org/rfc/rfc4468, 2006.

[13] "The bird internet routing daemon," https://bird.network.cz/, 2021.

[14] "The openbsd project produces a free, multi-platform 4.4bsd-based unix-like operating system." http://www.openbsd.org/, 2021.

[15] "The frrouting protocol suite," https://frrouting.org/, 2021.

[16] S. M. D. Savage, J. Ng, "Cisco's enhanced interior gateway routing protocol (eigrp)," https://www.rfc-editor.org/rfc/rfc7868, 2016.

[17] G. Malkin, "Rip version 2," https://www.rfc-editor.org/rfc/rfc2453, 1998.

[18] M. F. R. Atkinson, "Ripv2 cryptographic authentication," https://www.rfc-editor.org/rfc/rfc4822, 2007.

[19] R. Droms, "Dynamic host configuration protocol," https://www.rfc-editor.org/rfc/rfc2131/, 1997.

[20] "Internet systems consortium(isc) dhcp," https://www.isc.org/dhcp/, 2021.

[21] "Busybox," https://www.busybox.net/, 2021.

[22] "Dhcpcd," https://roy.marples.name/projects/dhcpcd/, 2021.

[23] P. M. E. Chen, J. Scudder, "Revised error handling for bgp update messages," https://www.rfc-editor.org/rfc/rfc7606, 2015.

[24] J. Song, C. Cadar, and P. Pietzuch, "Symbexnet: Testing network protocol implementations with symbolic execution and rule-based specifications," *IEEE Transactions on Software Engineering*, vol. 40, no. 7, pp. 695–709, 2014.

[25] E. Hoque, O. Chowdhury, S. Chau, C. Nita-Rotaru, and N. Li, "Analyzing operational behavior of stateful protocol implementations for detecting semantic bugs," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '17)*, 2017, pp. 627–638. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/DSN.2017.36

[26] J. Yen, T. Lévai, Q. Ye, X. Ren, R. Govindan, and B. Raghavan, "Semi-automated protocol disambiguation and code generation," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, ser. SIGCOMM '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 272–286. [Online]. Available: https://doi.org/10.1145/3452296.3472910

[27] E. Kim, D. Kim, C. Park, I. Yun, and Y. Kim, "Basespec: Comparative analysis of baseband software and cellular specifications for l3 protocols," in *Network and Distributed System Security Symposium(NDSS '21)*, 2021.

[28] Q. Wang, S. Ji, Y. Tian, X. Zhang, B. Zhao, Y. Kan, Z. Lin, C. Lin, S. Deng, A. X. Liu, and R. Beyah, "Mpinspector: A systematic and automatic approach for evaluating the security of iot messaging protocols," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 2021, pp. 4205–4222. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/wang-qinying

[29] "Peach fuzzer," https://www.peach.tech/, 2020.

[30] "boofuzz: Network protocol fuzzing for humans," https://boofuzz.readthedocs.io/en/latest/, 2020.

[31] "bestorm black box testing," https://beyondsecurity.com/solutions/bestorm.html, 2020.

[32] P. Fiterau-Brostean, B. Jonsson, R. Merget, J. de Ruiter, K. Sagonas, and J. Somorovsky, "Analysis of DTLS implementations using protocol state fuzzing," in *29th USENIX Security Symposium (USENIX Security '20)*. USENIX Association, 2020, pp. 2523–2540. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/fiterau-brostean

[33] J. Somorovsky, "Systematic fuzzing and testing of tls libraries," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16. Association for Computing Machinery, 2016, p. 1492–1504. [Online]. Available: https://doi.org/10.1145/2976749.2978411

[34] V. T. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: A greybox fuzzer for network protocols," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST '20)*, 2020, pp. 460–465.

[35] S. Sivakorn, G. Argyros, K. Pei, A. D. Keromytis, and S. Jana, "Hvlearn: Automated black-box analysis of hostname verification in ssl/tls implementations," in *2017 IEEE Symposium on Security and Privacy (SP '17)*. IEEE, 2017, pp. 521–538.

[36] J. de Ruiter and E. Poll, "Protocol state fuzzing of TLS implementations," in *24th USENIX Security Symposium (USENIX Security '15)*. Washington, D.C.: USENIX Association, 2015, pp. 193–206. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter

[37] L. Daniel, E. Poll, and J. de Ruiter, "Inferring openvpn state machines using protocol state fuzzing," in *2018 IEEE European Symposium on Security and Privacy Workshops (EuroS PW '18)*, 2018, pp. 11–19.

[38] M. Tappler, B. K. Aichernig, and R. Bloem, "Model-based testing iot communication via active automata learning," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST '17)*, 2017, pp. 276–287.

[39] J. de Ruiter and E. Poll, "Protocol state fuzzing of TLS implementations," in *24th USENIX Security Symposium (USENIX Security '15)*. USENIX Association, 2015, pp. 193–206. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/de-ruiter

[40] H. Zhao, Z. Li, H. Wei, J. Shi, and Y. Huang, "Seqfuzzer: An industrial protocol fuzzing framework from a deep learning perspective," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST '19)*, 2019, pp. 59–67.

[41] V.-T. P. Roberto Natella, "Profuzzbench: A benchmark for stateful protocol fuzzing," 2021. [Online]. Available: https://arxiv.org/abs/2101.05102

[42] P. Fiterău-Broştean, T. Lenaerts, E. Poll, J. de Ruiter, F. Vaandrager, and P. Verleg, "Model learning and model checking of ssh implementations," ser. SPIN 2017. Association for Computing Machinery, 2017, p. 142–151. [Online]. Available: https://doi.org/10.1145/3092282.3092289

[43] P. Fiterău-Broştean, R. Janssen, and F. Vaandrager, "Combining model learning and model checking to analyze tcp implementations," in *Computer Aided Verification*. Springer International Publishing, 2016, pp. 454–471.