Journal on Internet of Things
DOI: 10.32604/jiot.2021.014921
Article

Tech Science Press

Research on the Key Techniques of TCP Protocol Normalization for Mimic Defense Architecture

Mingxing Zhu, Yansong Wang, Ruyun Zhang, Tianning Zhang, Heyuan Li, Hanguang Luo and Shunbin Li[*]

Zhejiang Lab, Hangzhou, 310012, China
[*]Corresponding Author: Shunbin Li. Email: lishunbin@zhejianglab.com
Received: 25 May 2021; Accepted: 16 July 2021

Abstract: The Mimic Defense (MD) is an endogenous security technology with the core technique of Dynamic Heterogeneous Redundancy (DHR) architecture. It can effectively resist unknown vulnerabilities, backdoors, and other security threats by schedule strategy, negative feedback control, and other mechanisms. To solve the problem that Cyber Mimic Defense devices difficulty of supporting the TCP protocol. This paper proposes a TCP protocol normalization scheme for DHR architecture. Theoretical analysis and experimental results show that this scheme can realize the support of DHR-based network devices to TCP protocol without affecting the security of mimicry defense architecture.

Keywords: Mimic defense; TCP protocol; normalization

1 Introduction

Currently, the commonly used security defense strategies such as anti-virus and anti-trojan software, and vulnerability patches are a kind of "mend the fold after the sheep have been stolen" –like modes, which can only provide containment defenses against known security issues. However, unknown vulnerabilities and backdoors in networks and devices are ubiquitous and cannot be exhaustive. It is difficult to cope with the unknown vulnerabilities and backdoors defense issues by using traditional security defense methods. Mimic defense technology is based on dynamic heterogeneous redundancy (DHR) [1–3] architecture, which randomly extracts several elements from the pool of equivalent heterogeneous redundant executors aperiodically to form the service set. And it can perform defensive or repairable initialization and cleaning operations on heterogeneous redundant executors, so that the defensive system has the uncertainty of the apparent structural representation, thereby effectively resisting security attacks against system vulnerabilities and backdoors.

With the continuous improvement of Cyber Mimic Defense (CMD) theory, the corresponding network devices (hereinafter referred to as CMD devices) have appeared one after another. The representatives are the mimic defense Web server designed in [4], the router mimic defense architecture based on DHR proposed in [5], the endogenous security architecture of Ethernet switches based on mimic defense in [6], and the mimic security processor architecture for the industrial control field [7]. At present, the types and applications scenarios of network devices that support mimic defense functions are gradually increasing, and the expanded business scenarios also put forward higher requirements for the types and quality of communication protocols that CMD devices need to support.

Transmission Control Protocol (TCP) [8–10] is a connection-oriented, reliable, byte stream-based transmission layer communication protocol. In order to ensure the communication reliability, two communication devices or applications should establish a TCP connection before adopting the TCP protocol, and prepare a corresponding sequence number for each interactive data message to ensure that the data can be received in order at the receiving end. To make the CMD device support the TCP protocol, in a complete TCP protocol process (from connection establishment to disconnection), each executor in the CMD device needs to execute TCP sessions independently, and only output a unique TCP connection. However, TCP protocol involves the generation and use of random parameters. To ensure security, it is necessary to avoid information exchange and synchronization between the heterogeneous redundant executor. Such operation makes it impossible to synchronize the data messages generated by the TCP protocol running independently in each executor and the random parameters cannot be unified, which further makes it difficult to judge the TPC messages output by the heterogeneous executors. In order to resolve the above problems, Wei et al. [7] proposed a method of generating a uniform random number by using an external random number generator to make each redundant executor obtaining uniformed procedure parameters. However, although such method can solve the problem of the unification of random parameters in the redundant executors, it destroys the criterion that there is no clear synchronization information for each redundant executors under the mimic defense architecture, and introduces security risks to the mimic defense systems.

Aiming at the above mentioned TCP communication normalization and synchronization of the mimic defense systems, we propose a TCP connection normalization implementation scheme for dynamic heterogeneous redundant architecture. The proposed method effectively solves the problem of data coordination between redundant executor by matching, synchronizing and normalizing TCP data messages of each heterogeneous executor of system, and finally realizes TCP communication based on the mimic defense architecture.

2 Preliminaries

2.1 TCP Protocol

The TCP protocol realizes the establishment of a connection through the interaction of three message segments. This process is called a three-way handshake, as shown in Fig. 1. First, the client initiates a connection and sends a SYN message to the server, where seq = x represents a sequence number with a random value of x. After the server receives the SYN message from the client, it will return an ACK confirmation message, where seq = y is the random sequence number generated by the server, and ack = x + 1 is the confirmation sequence number calculated according to the client sequence number. Finally, the client returns confirmation messages seq = x + 1 and ack = y + 1, and the TCP connection is established.

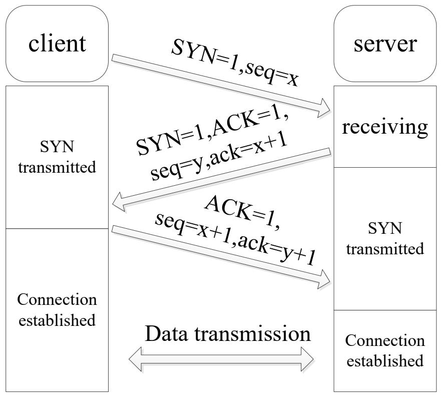


Figure 1: The three-way handshake process in TCP protocol

The data message transmission process of the TCP protocol is shown in Fig. 2. The data message is composed of TCP header and data. The seq = m represents the sequence number of the first byte of the data, and length = L represents the total length of the data. The receiving end returns an acknowledgement message ack = m + L after receiving the data message, indicating that the receiving end has received the corresponding data, and notifying the transmitting end that the next data is sent from the m + L-th byte.

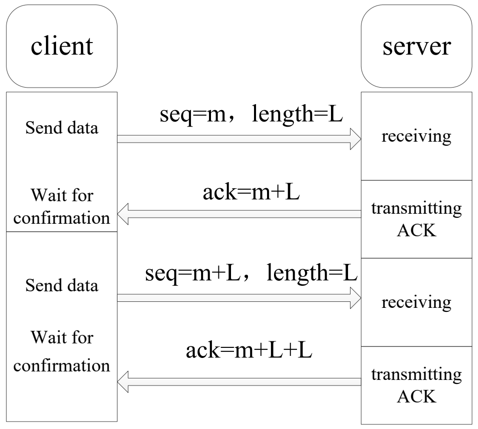


Figure 2: The data transmission process in TCP protocol

After the data transmission is completed, four-way wavehand is required to disconnect the TCP connection. Fig. 3 shows the four-way wavehand process initiated by the client (also can be initiated by the server of by both parties). First, the client sends a connection release message FIN, where the sequence number is seq = u. The server responds with ACK after receiving the FIN message, where the sequence number is seq = v, and the confirmation sequence number is ack = u + 1, then the server sends a connection release message FIN, where the sequence number is seq = w, and the confirmation sequence number is ack = u + 1. Finally, the client returns an acknowledgement message, where seq = u + 1, ack = w + 1, and the TCP connection ends.

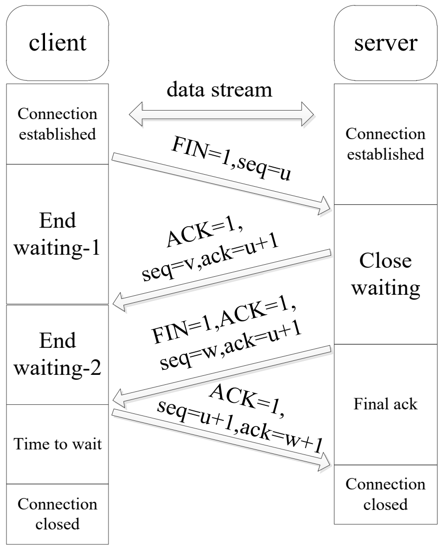


Figure 3: The four-way wavehand process in TCP protocol

2.2 Mimic Defense

Mimic defense [11–15] is a new type of active defense technology based on heterogeneous redundant structure. It uses mechanisms such as heterogeneous redundancy, strategic scheduling and negative feedback control to make the system possess endogenous features of active defense. The typical structure of DHR is shown in Fig. 4. In the DHR architecture, an odd number of executors are selected from the heterogeneous component set (E1 to Em) to form a heterogeneous equivalent executor set (A1 to An). The heterogeneous equivalent executors independently process the same information distributed by the input agent, and respectively output the equivalent processing results to the voting module. In the following, the voting module judges the output of the executor set through multi-mode decision algorithm, and uses the decision result as the final unified output. Finally, the voting module feeds back the decision result of the executor set to the scheduling module, and the scheduling module dynamically adjusts the heterogeneous equivalent executor set according to the corresponding strategy, including switching and cleaning of the executors.

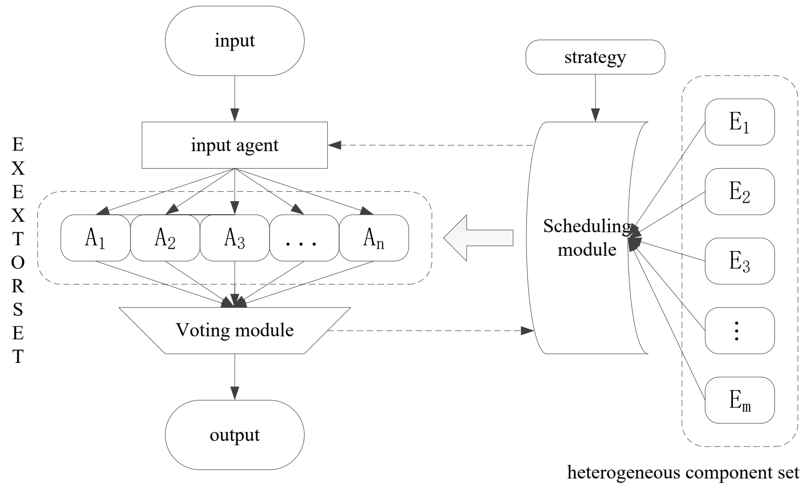


Figure 4: The diagram of DHR architecture

3 The Proposed Scheme

From the above description of TCP protocol and DHR architecture, it can be seen that the random parameters introduced in the TCP protocol make the equivalent output of the heterogeneous redundant executors inconsistent, resulting in the voting module being unable to judge the output of executors. In turn, the output of CMD devices cannot be unified. Aiming at the problem that the TCP connection of each executor of the CMD device is difficult to be displayed uniformly, we design a implementation scheme that normalizes the output of multiple TCP connections of redundant executor. Our proposed scheme includes matching and synchronization of TCP messages and the normalization of random parameters. The internal functional block diagram of CMD device is shown in Fig. 5, in which the matching, synchronization and normalization of TCP messages are all completed by the input/output agent module.

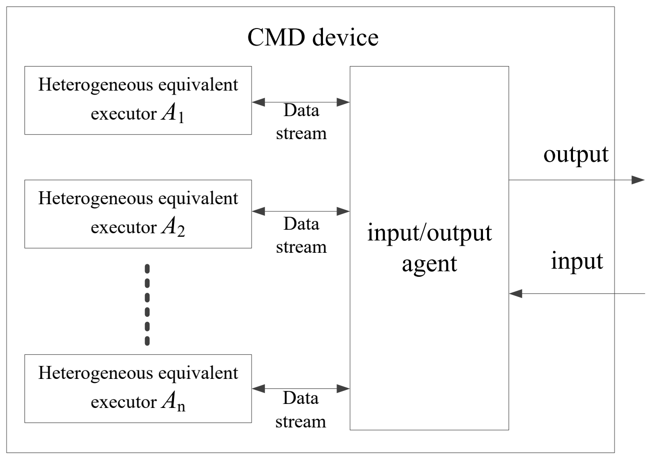

CMD device
Heterogeneous equivalent executor A_1
Data stream
Heterogeneous equivalent executor A_2
Data stream
Heterogeneous equivalent executor A_n
Data stream
input/output agent
output
input

Figure 5: The internal functional block diagram of CMD device

3.1 Message Matching and Synchronization

To achieve the normalization of redundant equivalent executors' TCP messages, it is necessary to correctly distinguish the equivalent TCP messages generated by each executor. Due to differences in the performance and operating mechanism of each executor under the DHR architecture, the equivalent TCP messages of each executor under the same TCP connection will be out of sync and order when they reach the input/output agent. In turn, the normalization operation fails.

To achieve the matching and synchronization of TCP messages, all TCP messages entering the input/output agent need to be buffered before the normalization operation. By comparing the source IP address, destination IP address, destination port number and other elements of the TCP header of the buffered message, the matching and synchronization of equivalent TCP messages of different executor are realized. Fig. 6 describes the schematic diagram of the principle of TCP message matching and synchronization. As shown in Fig. 6, we allocate k buffer queues in the buffer area of the input/output agent, and divide the corresponding data storage memory for n executors in each buffer queue. The data storage of each executor can be further divided into a tuple matching domain and a TCP message domain. The tuple matching domain stores the hash values of matching elements like source IP address, destination IP address, and destination port number of the TCP message. The corresponding TCP message is stored in the message domain. When the input/output agent receives a TCP message of a certain executor, it first extracts the matching element group in the TCP header (i.e., source IP address, destination IP address, destination port number, etc.). Then the input/output agent performs a hash operation, and sequentially compares the hash value with the hash values of matching tuples of other executors in the non-empty buffer queue. If there is an equal matching tuple in a buffer queue, the agent stores the hash value and the corresponding TCP message in the corresponding position of the buffer queue. Otherwise, it saves the hash value and the corresponding TCP message to the corresponding position of the empty buffer queue, and wait for synchronization and matching with the message of other executors. If there is an equal matching tuple in a buffer queue, the hash value and the corresponding TCP message are stored in the buffer queue. When the buffer space corresponding to all the executors in a certain buffer queue is empty, it indicates that the matching synchronization process for the same TCP message is completed, and then the normalization phase is entered.

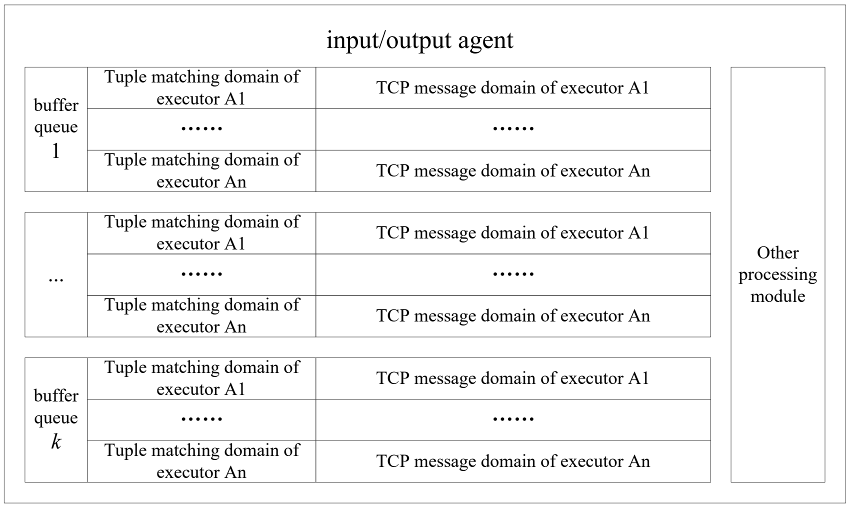

input/output agent
buffer queue 1
Tuple matching domain of executor A1
TCP message domain of executor A1
……
……
Tuple matching domain of executor An
TCP message domain of executor An
...
Tuple matching domain of executor A1
TCP message domain of executor A1
……
……
Tuple matching domain of executor An
TCP message domain of executor An
buffer queue *k*
Tuple matching domain of executor A1
TCP message domain of executor A1
……
……
Tuple matching domain of executor An
TCP message domain of executor An
Other processing module

Figure 6: The schematic diagram of TCP message matching

3.2 Random Parameter Normalization

After matching and synchronizing the equivalent TCP message of each executor, it is necessary to normalize the random parameters in the TCP header. Because the proposed method has similar normalization processing principles for each phase of the TCP protocol, only the normalization process of the TCP three-way handshake (as shown in Fig. 1) is described in detail, and the description of normalization process of data transmission and four-way wavehand will not be repeated.

The TCP protocol normalization process involves many random parameters. In the following, we will take the sequence number and confirmation sequence number as examples to introduce the normalization process in detail. The normalization principle of other random parameters is similar. In addition, for the convenience of description, we use subscripts to distinguish the message sent or received by each executor. For example, seq1 represents the sequence number of the TCP message sent by the executor A1.

Referring the TCP three-way handshake process in Fig. 1 and the internal functional block diagram in Fig. 5, taking the client as a CMD device as an example, the detailed normalization steps for TCP three-way handshake are as follows.

Step 1: The heterogeneous equivalent executors (A1 to An) respectively send TCP connection request messages to the input and output agents, and the corresponding random parameters in TCP message header connected to the requests are from [SYN1 = 1, seq1 = x1] to [SYNn = 1, seqn = xn].

Step 2: The input/output agent extracts the corresponding sequence number seq1 = x1 to seqn = xn after receiving the message from each heterogeneous executor, and calculates the unified message sequence number seq = x based on x1 to xn. x can be obtained by simple logical operations such as AND, OR, XOR, etc. For example, $x = x1 \oplus x2 \oplus \ldots \oplus xn$ ($\oplus$ stands for XOR operation). In the following, seq = x is used as the unified sequence number of TCP message to generate a normalized TCP session request (i.e., [SYN = 1, seq = x]) which is sent to the remote server.

Step 3: Based on the TCP session request, the remote server returns an acknowledgement message, i.e., [seq = y, ack = x + 1].

Step 4: After the input/output agent receives the confirmation message returned from the remote server, it extracts the confirmation sequence number ack = x + 1. Then through the inverse logical operation of Step 2, the input/output agent restores the confirmation sequence number of the corresponding heterogeneous equivalent executor. For example, if the logical operation used in Step 2 is XOR, the confirmation sequence number of executor A1 is $ack1 = ack \oplus seq2 \oplus seq3 \oplus \ldots \oplus seqn$. Finally, the input/output agent reorganizes all confirmation messages [seq1 = y, ack1] to [seqn = y, ackn] according to calculated confirmation sequence numbers, and send them to the corresponding heterogeneous equivalent executors.

Step 5: After the heterogeneous equivalent executors receive the confirmation messages, they generate the corresponding confirmation messages [seq1 = ack1, ack1 = y + 1] to [seqn = ackn, ackn = y + 1], and then send them to the input/output agent.

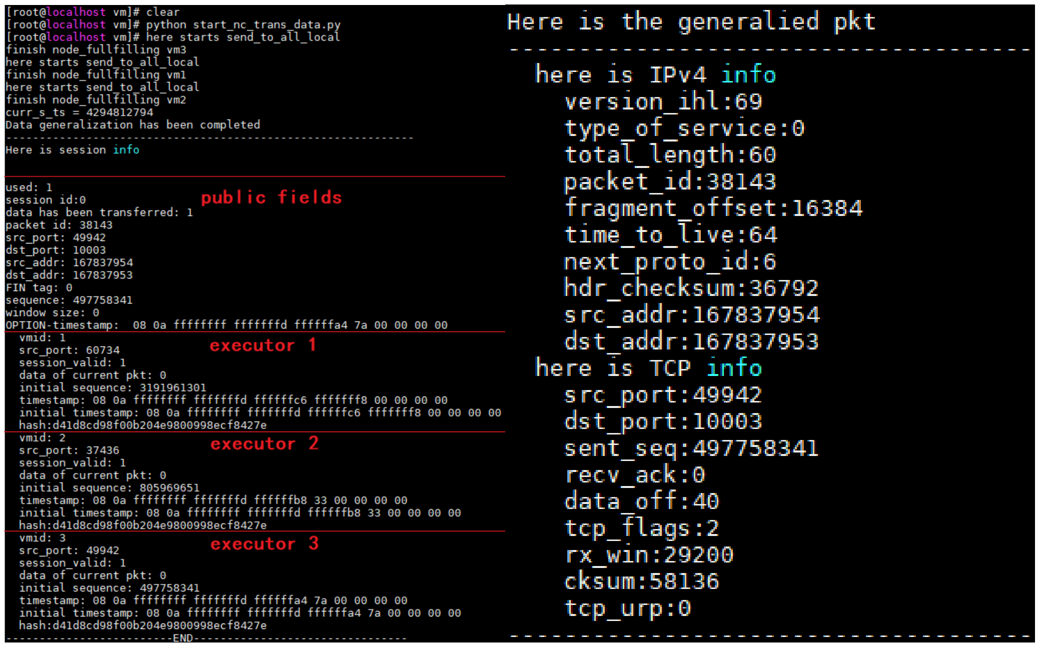
Step 6: After the input/output agent receives the confirmation message returned by each heterogeneous equivalent executor, it generates an external normalized message [seq = x + 1, ack = y + 1], and then sends it to the remote server. The TCP three-way handshake normalization process ends.

It is noted that all reassembled TCP messages in the normalization process need to recalculate the checksum of the TCP message header. Through the above normalization processing steps, it can be ensured that each heterogeneous equivalent executor of the CMD device independently completes the three-way handshake process, and only outputs a unified TCP session connection. The data transmission phase and the four-way wavehand phase of TCP session can be normalized in a similar way. In the data transmission phase, TCP supports sliding window mechanism. Due to the differences in the sending and receiving performance between different heterogeneous equivalent executors, the normalized window size is set to the minimal size among all executors.

4 Experimental Results

In order to verify the TCP normalization function of CMD device, we use virtualization technology to build a set of simulation environment on the server. The server CPU adopts Intel Xeon processor. The operating system is CentOS 7.6 with the necessary components such as qemu-kvm-3.0.90, libvirt-4.5.0 and dpdk-stable-18.11.2. On the server, four virtual machines are built based on qemu-kvm+libvirt, of which three virtual machines simulate three heterogeneous equivalent executors in the CMD device, and the other virtual machine simulates a remote PC. All virtual machines are installed with CentOS 7.6 minimum system. Each virtual machine is allocated 2 vCPU, 2 GB memory and 64 GB virtual hard disk, and 1 virtual network card is allocated to connect to the TCP normalization module. All virtual network cards of the executors are configured with the same IP address (10.1.1.2/24) and MAC address, and the IP address of the remote PC virtual network card is 10.1.1.1/24. The TCP normalization module is based on DPDK (dpdk-18.11.2) design, and each virtual machine (including the executor's virtual machine and the remote PC's virtual machine) uses a vhost-user interface as the network connection interface, which corresponds to the network card in the virtual machine.

The simulation elaborately shows the normalization process of the TCP three-way handshake. Fig. 7(a) shows a part of the cache content in the first handshake message sent by the CMD device as a client, and some random parameters to be normalized are stored in the public field. Among them, the normalized sequence number, source port number, time stamp and other random parameters are equal to the corresponding value of the 3rd executor. The key parameters of the first handshake TCP message header sent by the CMD device after normalization are shown in Fig. 7(b), where the window size rx_win is set to the smallest window size among the three executors.



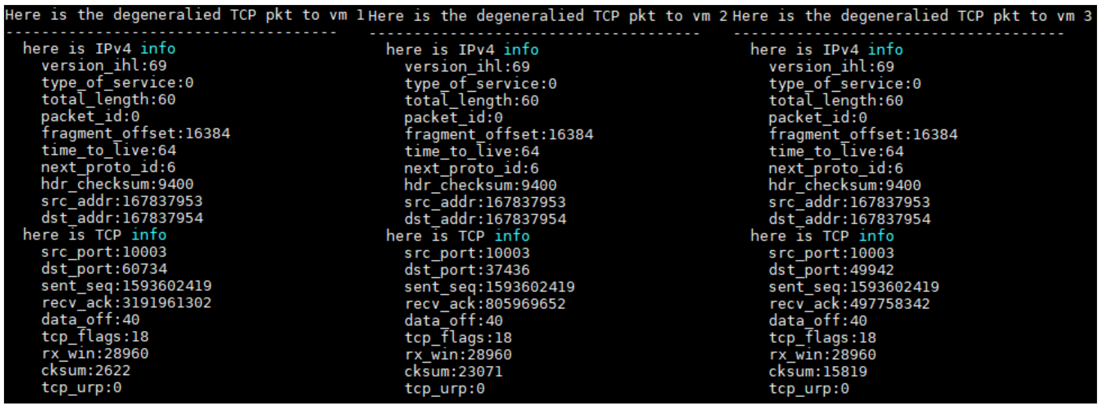
```
[root@localhost vm]# clear
[root@localhost vm]# python start_nc_trans_data.py
[root@localhost vm]# here starts send_to_all_local
finish node_fullfilling vm3
here starts send_to_all_local
finish node_fullfilling vm1
here starts send_to_all_local
finish node_fullfilling vm2
curr_s_ts = 4294812794
Data generalization has been completed
-------------------------------------------------------------
Here is session info

used: 1
session id:0
data has been transferred: 1
packet id: 38143
src_port: 49942
dst_port: 10003
src_addr: 167837954
dst_addr: 167837953
FIN tag: 0
sequence: 497758341
window size: 0
OPTION-timestamp:  08 0a ffffffff fffffffd ffffffa4 7a 00 00 00 00
  vmid: 1
  src_port: 60734
  session_valid: 1
  data of current pkt: 0
  initial sequence: 3191961301
  timestamp: 08 0a ffffffff fffffffd ffffffc6 fffffff8 00 00 00 00
  initial timestamp: 08 0a ffffffff fffffffd ffffffc6 fffffff8 00 00 00 00
  hash:d41d8cd98f00b204e9800998ecf8427e
  vmid: 2
  src_port: 37436
  session_valid: 1
  data of current pkt: 0
  initial sequence: 805969651
  timestamp: 08 0a ffffffff fffffffd ffffffb8 33 00 00 00 00
  initial timestamp: 08 0a ffffffff fffffffd ffffffb8 33 00 00 00 00
  hash:d41d8cd98f00b204e9800998ecf8427e
  vmid: 3
  src_port: 49942
  session_valid: 1
  data of current pkt: 0
  initial sequence: 497758341
  timestamp: 08 0a ffffffff fffffffd ffffffa4 7a 00 00 00 00
  initial timestamp: 08 0a ffffffff fffffffd ffffffa4 7a 00 00 00 00
  hash:d41d8cd98f00b204e9800998ecf8427e
-------------------------END-------------------------------
```

```
Here is the generalied pkt
--------------------------------------
  here is IPv4 info
    version_ihl:69
    type_of_service:0
    total_length:60
    packet_id:38143
    fragment_offset:16384
    time_to_live:64
    next_proto_id:6
    hdr_checksum:36792
    src_addr:167837954
    dst_addr:167837953
  here is TCP info
    src_port:49942
    dst_port:10003
    sent_seq:497758341
    recv_ack:0
    data_off:40
    tcp_flags:2
    rx_win:29200
    cksum:58136
    tcp_urp:0
--------------------------------------
```

(a) (b)

Figure 7: The cache data in the first handshake message of CMD device

The confirmation package returned by the remote server is restored by the input/output agent and distributed to each executor. The key parameters of the restored message of each executor are shown in Fig. 8. The timestamp and destination port number corresponding to each executor are obtained from the cache and restored. The method for restoring the corresponding confirmation sequence number of each executor is that the confirmation sequence number returned by the remote server + the normalized sequence number – the sequence number of each executor. Fig. 9 shows the partial data of the confirmation message returned by the CMD device. Herein, Fig. 9(a) shows the relevant parameters before normalization, and Fig. 9(b) shows the relevant parameters after normalization.

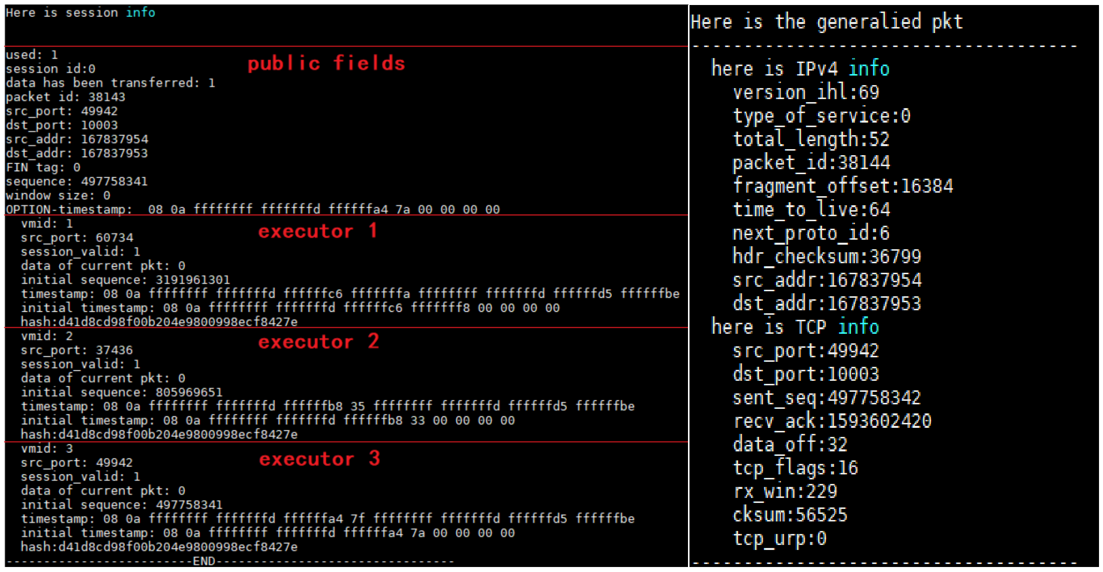


```
Here is the degeneralied TCP pkt to vm 1
-------------------------------------
 here is IPv4 info
   version_ihl:69
   type_of_service:0
   total_length:60
   packet_id:0
   fragment_offset:16384
   time_to_live:64
   next_proto_id:6
   hdr_checksum:9400
   src_addr:167837953
   dst_addr:167837954
 here is TCP info
   src_port:10003
   dst_port:60734
   sent_seq:1593602419
   recv_ack:3191961302
   data_off:40
   tcp_flags:18
   rx_win:28960
   cksum:2622
   tcp_urp:0
```

```
Here is the degeneralied TCP pkt to vm 2
-------------------------------------
 here is IPv4 info
   version_ihl:69
   type_of_service:0
   total_length:60
   packet_id:0
   fragment_offset:16384
   time_to_live:64
   next_proto_id:6
   hdr_checksum:9400
   src_addr:167837953
   dst_addr:167837954
 here is TCP info
   src_port:10003
   dst_port:37436
   sent_seq:1593602419
   recv_ack:805969652
   data_off:40
   tcp_flags:18
   rx_win:28960
   cksum:23071
   tcp_urp:0
```

```
Here is the degeneralied TCP pkt to vm 3
-------------------------------------
 here is IPv4 info
   version_ihl:69
   type_of_service:0
   total_length:60
   packet_id:0
   fragment_offset:16384
   time_to_live:64
   next_proto_id:6
   hdr_checksum:9400
   src_addr:167837953
   dst_addr:167837954
 here is TCP info
   src_port:10003
   dst_port:49942
   sent_seq:1593602419
   recv_ack:497758342
   data_off:40
   tcp_flags:18
   rx_win:28960
   cksum:15819
   tcp_urp:0
```

Figure 8: The key cache data of the remote confirmation package restored by the executors



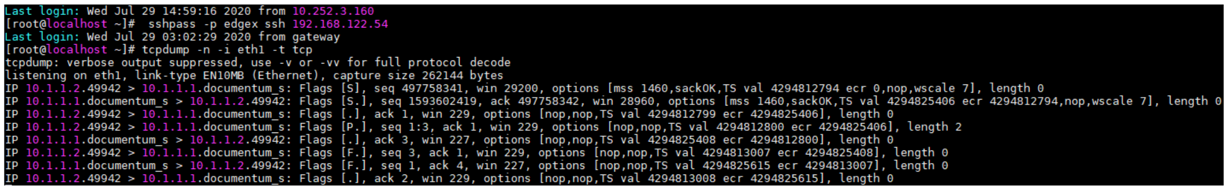
```
Here is session info

used: 1
session id:0
data has been transferred: 1
packet id: 38143
src_port: 49942
dst_port: 10003
src_addr: 167837954
dst_addr: 167837953
FIN tag: 0
sequence: 497758341
window size: 0
OPTION-timestamp:  08 0a ffffffff fffffffd ffffffa4 7a 00 00 00 00
  vmid: 1
  src_port: 60734
  session_valid: 1
  data of current pkt: 0
  initial sequence: 3191961301
  timestamp: 08 0a ffffffff fffffffd ffffffc6 fffffffa ffffffff fffffffd ffffffd5 ffffffbe
  initial timestamp: 08 0a ffffffff fffffffd ffffffc6 fffffff8 00 00 00 00
  hash:d41d8cd98f00b204e9800998ecf8427e
  vmid: 2
  src_port: 37436
  session_valid: 1
  data of current pkt: 0
  initial sequence: 805969651
  timestamp: 08 0a ffffffff fffffffd ffffffb8 35 ffffffff fffffffd ffffffd5 ffffffbe
  initial timestamp: 08 0a ffffffff fffffffd ffffffb8 33 00 00 00 00
  hash:d41d8cd98f00b204e9800998ecf8427e
  vmid: 3
  src_port: 49942
  session_valid: 1
  data of current pkt: 0
  initial sequence: 497758341
  timestamp: 08 0a ffffffff fffffffd ffffffa4 7f ffffffff fffffffd ffffffd5 ffffffbe
  initial timestamp: 08 0a ffffffff fffffffd ffffffa4 7a 00 00 00 00
  hash:d41d8cd98f00b204e9800998ecf8427e
-------------------------END--------------------------------
```

```
Here is the generalied pkt
-------------------------------------
 here is IPv4 info
   version_ihl:69
   type_of_service:0
   total_length:52
   packet_id:38144
   fragment_offset:16384
   time_to_live:64
   next_proto_id:6
   hdr_checksum:36799
   src_addr:167837954
   dst_addr:167837953
 here is TCP info
   src_port:49942
   dst_port:10003
   sent_seq:497758342
   recv_ack:1593602420
   data_off:32
   tcp_flags:16
   rx_win:229
   cksum:56525
   tcp_urp:0
-------------------------------------
```

(a) (b)

Figure 9: The key data of the confirmation message returned by the CMD device

Fig. 10 shows the TCP protocol process recorded by the remote server. In this process, the CMD device (3 executors) actively initiates a TCP handshake connection to the remote PC. After the connection is successful, the executor sends a data message (set the sending character as "a"), and finally three executors disconnect the connection.



```
Last login: Wed Jul 29 14:59:16 2020 from 10.252.3.160
[root@localhost ~]#  sshpass -p edgex ssh 192.168.122.54
Last login: Wed Jul 29 03:02:29 2020 from gateway
[root@localhost ~]# tcpdump -n -i eth1 -t tcp
tcpdump: verbose output suppressed, use -v or -vv for full protocol decode
listening on eth1, link-type EN10MB (Ethernet), capture size 262144 bytes
IP 10.1.1.2.49942 > 10.1.1.1.documentum_s: Flags [S], seq 497758341, win 29200, options [mss 1460,sackOK,TS val 4294812794 ecr 0,nop,wscale 7], length 0
IP 10.1.1.1.documentum_s > 10.1.1.2.49942: Flags [S.], seq 1593602419, ack 497758342, win 28960, options [mss 1460,sackOK,TS val 4294825406 ecr 4294812794,nop,wscale 7], length 0
IP 10.1.1.2.49942 > 10.1.1.1.documentum_s: Flags [.], ack 1, win 229, options [nop,nop,TS val 4294812799 ecr 4294825406], length 0
IP 10.1.1.2.49942 > 10.1.1.1.documentum_s: Flags [P.], seq 1:3, ack 1, win 229, options [nop,nop,TS val 4294812800 ecr 4294825406], length 2
IP 10.1.1.1.documentum_s > 10.1.1.2.49942: Flags [.], ack 3, win 227, options [nop,nop,TS val 4294825408 ecr 4294812800], length 0
IP 10.1.1.2.49942 > 10.1.1.1.documentum_s: Flags [F.], seq 3, ack 1, win 229, options [nop,nop,TS val 4294813007 ecr 4294825408], length 0
IP 10.1.1.1.documentum_s > 10.1.1.2.49942: Flags [F.], seq 1, ack 4, win 227, options [nop,nop,TS val 4294825615 ecr 4294813007], length 0
IP 10.1.1.2.49942 > 10.1.1.1.documentum_s: Flags [.], ack 2, win 229, options [nop,nop,TS val 4294813008 ecr 4294825615], length 0
```

Figure 10: TCP protocol process recorded by remote server

Extensive simulation results show that the normalization method designed in this paper realizes TCP communication based on mimic defense, ensures that each executor in the DHR architecture runs the TCP protocol independently, and provides effective technical support for the CMD device to support connection-oriented network services.

5 Conclusion

In this paper, we propose a TCP protocol normalization method for mimic defense architecture. Through operations including matching synchronization and random parameter normalization, the proposed method can achieve that each heterogeneous equivalent executor under the DHR architecture runs the TCP protocol independently and output the unified results. The experimental results show that the proposed method can effectively implement TCP communication based on the mimic defense architecture, and cannot perceive the existence of the internal executor of the CMD device through the analysis of communication data packets, which provides attractive support for connection-oriented services based on the mimic defense network architecture solution.–

Funding Statement: This work was supported by the National Key Research and Development Project of China (Grant No. 2020YFB1804600), the Major Scientific Project of Zhejiang Lab (2018FD0ZX01).

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] H. Hu, F. Chen and Z. Wang, "Performance evaluation on DHR for cyberspace mimic defense," *Journal of Cyber Security*, vol. 1, pp. 40–51, 2016.

[2] W. Zhu, Y. Guo and B. Huang, "A mimic defense automaton model of dynamic heterogeneous redundancy structures," *Acta Electronica Sinica*, vol. 47, pp. 2025–2031, 2019.

[3] Y. Fan, W. Zhu and S. Ban, "Dynamic heterogeneous and redundancy data protection architecture," *Journal of Chinese Computer Systems*, vol. 40, pp. 1956–1961, 2019.

[4] Q. Tong, Z. Zhang, W. Zhang and J. Wu, "Design and implementation of mimic defense web server," *Journal of Software*, vol. 28, pp. 883–897, 2017.

[5] H. Ma, P. Yi, Y. Jiang and L. He, "Dynamic heterogeneous redundancy based router architecture with mimic defense," *Journal of Cyber Security*, vol. 2, pp. 29–42, 2017.

[6] K. Song, Q. Liu, S. Wei, W. Zhang and L. Tan, "Endogenous security architecture of Ethernet switch based on mimic defense," *Journal on Communications*, vol. 41, pp. 18–26, 2020.

[7] S. Wei, H. Yu, Z. Gu and X. Zhang, "Architecture of mimic security processor for industry control system," *Journal of Cyber Security*, vol. 2, pp. 54–73, 2017.

[8] J. Zhu, "TCP protocol outlined and three-way handshake principle of analytic," *Computer Knowledge and Technology*, vol. 5, pp. 1079–1080, 2009.

[9] K. Poduri and K. Nichols, "Simulation studies of increased initial TCP window size," CiteSeer^x, 1998.

[10] B. A. Forouzan and S. C. Fegan, *TCP/IP Protocol Suite*. McGraw-Hill Higher Education, 2002, pp. 1–64.

[11] H. Yu, X. Zhang and K. Song, "A device and method for ensuring consistent encryption behavior of redundant executables," CN 110176988, 2019.

[12] J. X. Wu, "Meaning and vision of mimic computing and mimic security defense," *Telecommunications Science*, vol. 30, no. 7, pp. 1–7, 2014.

[13] J. X. Wu, "Research on cyber mimic defense," *Journal of Cyber Security*, vol. 1, no. 4, pp. 1–10, 2016.

[14] J. X. Wu, *Cyberspace Mimic Defense*. Springer, 2019, pp. 1–25.

[15] J. X. Wu, *Cyberspace Mimic Defense (2nd)*. Springer, 2019, pp. 113–135.