Implementation of Eavesdropping Protection Method over MPTCP Using Data Scrambling and Path Dispersion

Toshihiko Kato[1)2)], Shihan Cheng[1)], Ryo Yamamoto[1)], Satoshi Ohzahata[1)] and Nobuo Suzuki[2)]
1) University of Electro-Communications, Tokyo, Japan
2) Advanced Telecommunication Research Institute International, Kyoto, Japan
e-mail: kato@net.lab.uec.ac.jp, chengshihan@net.lab.uec.ac.jp, ryo_yamamoto@net.lab.uec.ac.jp, ohzahata@net.lab.uec.ac.jp, nu-suzuki@atr.jp

***Abstract*—In order to utilize multiple communication interfaces installed mobile terminals, Multipath Transmission Control Protocol (MPTCP) has been introduced recently. It can establish an MPTCP connection that transmits data segments over the multiple interfaces, such as 4G and Wireless Local Area Network (WLAN), in parallel. However, it is possible that some interfaces are connected to untrusted networks and that data transferred over them is observed in an unauthorized way. In order to avoid this situation, we proposed a method to improve privacy against eavesdropping using the data dispersion by exploiting the multipath nature of MPTCP. The proposed method takes an approach that, if an attacker cannot observe the data on *every* path, he cannot observe the traffic on *any* path. The fundamental techniques of this method is a per-byte data scrambling and path dispersion. In this paper, we present the result of implementing the proposed method within the Linux operating system and its performance evaluation.**

Keywords- Multipath TCP; Eavesdropping; Data Dispersion; Data Scrambling.

I. INTRODUCTION

Recent mobile terminals are equipped with multiple interfaces. For example, most smart phones have interfaces for 4G Long Term Evolution (LTE) and WLAN. In the next generation (5G) network, it is studied that multiple communication paths provided multiple network operators are commonly involved [1]. In this case, mobile terminals will have more than two interfaces.

However, the traditional TCP establishes a connection between a single IP address at one end, and so it cannot utilize multiple interfaces at the same time. In order to cope with this issue, MPTCP [2] is being introduced in several operating systems, such as Linux, Apple OS/iOS [3] and Android [4]. MPTCP is an extension of TCP. Conventional TCP applications can use MPTCP as if they were working over traditional TCP and are provided with multiple byte streams through different interfaces.

MPTCP is defined in three Request for Comments (RFC) documents by the Internet Engineering Task Force. RFC 6182 [5] outlines architecture guidelines. RFC 6824 [6] presents the details of extensions to support multipath operation, including the maintenance of an MPTCP connection and subflows (TCP connections associated with an MPTCP connection), and the data transfer over an MPTCP connection. RFC 6356 [7] presents a congestion control algorithm that couples the congestion control algorithms running on different subflows.

When a mobile terminal uses multiple paths, some of them may be unsafe such that an attacker is able to observe data over them in an unauthorized way. For example, a WLAN interface is connected to a public WLAN access point, data transferred over this WLAN may be disposed to other nodes connected to it. One way to prevent the eavesdropping is the Transport Layer Security (TLS). Although TLS can be applied to various applications including web access, e-mail, and ftp, however, it requires at least one end to maintain a public key certificate, and so it will not be used in some kind of communication, such as private server access and peer to peer communication.

As an alternative scheme, we proposed a method to improve confidentiality against eavesdropping by exploiting the multipath nature of MPTCP [8][9]. Even if an unsafe WLAN path is used, another path may be safe, such as LTE supported by a trusted network operator. So, the proposed method is based on an idea that, if an attacker cannot observe the data on *every* path, he cannot observe the traffic on *any* path [10]. In order to realize this idea, we adopted a byte based data scrambling for data segments sent over multiple subflows. This mixes up data to avoid its recognition through illegal monitoring over an unsafe path. Although there are some proposals to use multiple TCP connections to protect eavesdropping [11]-[14], all of them depend on the encryption techniques. The proposed method is dependent on the exclusive OR (XOR) calculation that is much lighter in terms of processing overhead.

In this paper, we show the result of implementation of the proposed method and the result of performance evaluation. We adopted a kernel debugging mechanism in the Linux operating system so as to modify the Linux kernel as least as possible. We conducted performance evaluation through Ethernet and WLAN using off-the-shelf PCs and access point.

The rest of this paper is organized as follows. Section II explains the details of the proposed method. Section III shows how to implement the proposed method within the MPTCP software in the Linux operating system. Section IV gives the results of the performance evaluation. In the end, Section V concludes this paper.

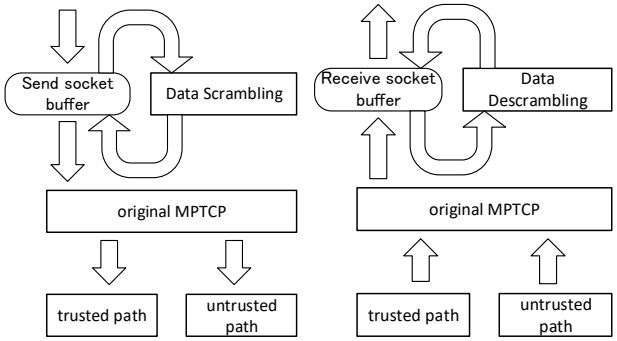
II. DETAILS OF PROPOSED METHOD

Figure 1 shows the overview of the proposed method. When an application sends data, it is stored in the send socket buffer in the beginning. The proposed method scrambles the data by calculating XOR of a byte with its preceding 64 bytes in the sending byte stream. Then, the scrambled data is sent

 ISBN: 978-1-61208-661-3

through multiple subflows associated with the MPTCP connection. Since some data segments are transmitted through trusted subflows, an attacker monitoring only a part of data segments cannot obtain all of sent data and so cannot descramble any of them. When receiving data segments, they are reordered in the receive socket buffer by MPTCP. The proposed method descrambles them in a byte-by-byte basis just before an application reads the received data.

Figure 2 shows the details of data scrambling. In order to realize this scrambling, the data scrambling module maintains



(a) Sending data (b) Receiving data

Figure 1. Overview of proposed method [8].

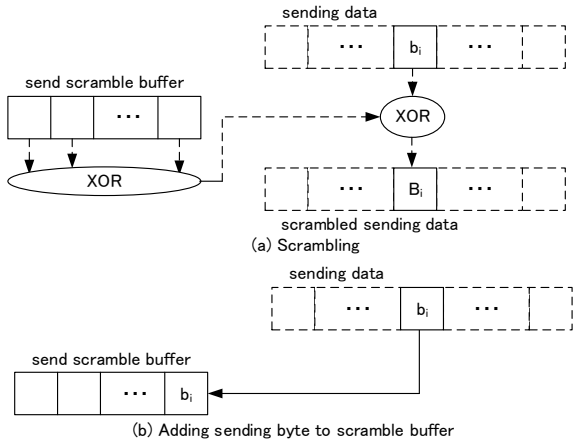


Figure 2. Processing of data scrambling [8].

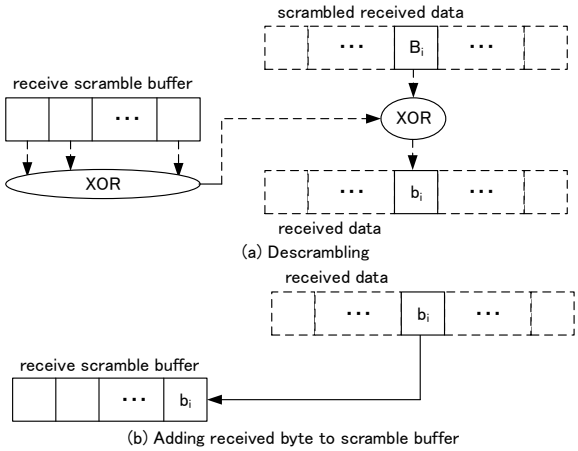


Figure 3. Processing of data descrambling [8].

the *send scrambling buffer*, whose length is 64 bytes. It is a shift buffer and its initial value is HMAC of the key of this side, with higher bytes set to zero. The key used here is one of the MPTCP parameters, exchanged in the first stage of MPTCP connection establishment. When a data comes from an application, each byte (b_i in the figure) is XORed with the result of XOR of all the bytes in the send scrambling buffer. The obtained byte (B_i) is the corresponding sending byte. After calculating the sending byte, the original byte (b_i) is added to the send scramble buffer, forcing out the oldest (highest) byte from the buffer. The send scrambling buffer holds recent 64 original bytes given from an application. By using 64 byte buffer, the access to the original data is protected even if there are well-known byte patterns (up to 63 bytes) in application protocol data.

Figure 3 shows the details of data descrambling, which is similar with data scrambling. The data scrambling module also maintains the receive scramble buffer whose length is 64 bytes. Its initial value is HMAC of the key of the remote side. When an in-sequence data is stored in the receive socket buffer, a byte (B_i that is scrambled) is applied to XOR calculation with the XOR result of all the bytes in the receive scramble buffer. The result is the descrambled byte (b_i), which is added to the receive scramble buffer.

By using the byte-wise scrambling and descrambling, the proposed method does not increase the length of exchanged data at all. The separate send and receive control enables two way data exchanges to be handled independently. Moreover the proposed method introduces only a few modification to the original MPTCP.

III. IMPLEMENTATION

A. Use of Kernel Probes

Since MPTCP is implemented inside the Linux operating system, the proposed method also needs to be realized by modifying operating system kernel. However, modifying an operating system kernel is a hard task, and so we decided to use a debugging mechanism for the Linux kernel, called kernel probes [15].

Among kernel probes methods, we use a way called "JProbe" [9]. JProbe is used to get access to a kernel function's arguments at runtime. It introduces a JProbe handler with the same prototype as that of the function whose arguments are to be accessed. When the probed function is executed, the control is first transferred to the user-defined JProbe handler. After the user-defined handler returns, the control is transferred to the original function [15].

In order to make this mechanism work, a user needs to prepare the following;

- registering the entry by `struct jprobe` and
- defining the init and exit modules by functions `register_jprobe()` and `unregister_jprobe()` [16].

In the Linux kernel, function `tcp_sendmsg()` is called when an application sends data to MPTPCP (actually TCP, too) [17]. As stated in Section II, the scrambling will be done at the beginning of this function. So, we define a JProbe

handler for function `tcp_sendmsg()` for scrambling data to be transferred.

In order for an application to read received data, it calls function `tcp_recvmsg()` in MPTCP. In contrast to data scrambling, the descrambling procedure needs to be done at the end of this function. So, we introduce a dummy kernel function and export its symbol just before the returning points of function `tcp_recvmsg()`. We then define a JProbe handler for descrambling in this dummy function.

By adopting this approach, we can program and debug scrambling/descrambling independently of the Linux kernel itself.

B. Modification of Linux opeating system

We modified the source code of the Linux operating system in the following way. We believe that this is a very slight modification that requires to us to rebuild the kernel only once.

- *Introduce a dummy function in* `tcp_recvmsg()`.

As described above, we defined a dummy function named `dummy_recvmsg()`. It is defined in the source file "`net/ipv4/tcp.c`" as shown in Figure 4. It is a function just returning and inserted before function `tcp_recvmsg()` releases the socket control. The prototype declaration is done in the source file "`include/net/tcp.h`".

- *Maintain control variables within socket data structure.*

In order to perform the scrambling/descrambling, the control variables, such as a scramble buffer, need to be installed within the Linux kernel. The TCP software in the kernel uses a socket data structure to maintain internal control data on an individual TCP / MPTCP connection [17]. This is controlled by the following variable, as shown in Figure 4.

```
struct tcp_sock *tp = tcp_sk(sk);
```

This structure includes the MPTCP related parameters, such as keys and tokens. The parameters are packed in an element given blow.

```
int tcp_recvmsg(struct sock *sk, struct msghdr *msg,
    size_t len, int nonblock,int flags, int *addr_len) {
  struct tcp_sock *tp = tcp_sk(sk);
  . . . .
  dummy_recvmsg(sk, msg, len, nonblack, flags, addr_len);
  release_sock(sk);
  return copied;
  . . . .
} // dummy_recvmsg() inserted
EXPORT_SYMBOL(tcp_recvmsg);

void dummy_recvmsg(struct sock *sk, struct msghdr *msg,
     size_t len, int nonblock, int flags, int *addr_len)
{
  return;
} // Defining dummy_recvmsg()
EXPORT_SYMBOL(dummy_recvmsg);
```

Figure 4. Dummy function in `tcp_recvmsg()`.

```
struct mptcp_cb {
  . . . .
  unsigned char sScrBuf[64], rScrBuf[64];
  unsigned char sXor, rXor;
  int sIndex, rIndex, sNotFirst, rNotFirst;
};
```

Figure 5. Control variables for data scrambling/descrambling.

```
struct mptcp_cb *mpcb;
```

So, we added the control variables for data scrambling in this data structure. Figure 5 shows the control variables. The details of those variables are given in the following.

- `sScrBuf[64]` and `rScrBuf[64]`: the send and receive scramble buffers, used as ring buffers.
- `sXor` and `rXor`: the results of calculation of XOR for all the bytes in the send and receive scramble buffers.
- `sIndex` and `rIndex`: the index of the last (newest) element in `sScrBuf[64]` and `rScrBuf[64]`.
- `sNotFirst` and `rNotFirst`: the flags indicating whether the scrambling and descrambling are invoked for the first time in the MPTCP connection, or not.

C. Implementation of scrambling

(1) Framework of JProbe handler

Figure 6 shows the framework of JProbe hander defined for `tcp_sendmsg()`. Function `jtcp_sendmsg()` is a main body of the JProbe hander. The arguments need to be exactly the same with the hooked kernel function `tcp_sendmsg()`, and it calls `jprobe_return()` just before its returning. Data structure `struct jprobe mptcp_jprobe` specifies its details.

Function `mptcp_scramble_init()` is the initialization function invoked when the relevant kernel module is inserted. In the beginning, it confirm that the hander has the same prototype with the hooked function. Then it defines the entry point and registers the JProbe handler. Function `mptcp_scramble_exit()` is called when the relevant kernel module is removed. It removes the entry point and unregisters the hander from the kernel.

(2) Flowchart of data scrambling

The data scrambling procedure is implemented in `jtcp_sendmsg()`. Figure 7 shows the flowchart for this

```
static const char procname[] = "mptcp_scramble"
int jtcp_sendmsg(struct sock *sk, struct msghdr *msg,
       size_t size) {
  struct tcp_sock *tp = tcp_sk(sk);
  . . .
  jprobe_return();
  return 0;
} // (i) JProbe handler

static struct jprobe mptcp_jprobe = {
  .kp = {.symbol_name = "tcp_sendmsg",},
        .entry = jtcp_sendmsg,
}; // (ii) Register entry

static __init int mptcp_scramble_init(void) {
  int ret = -ENOMEM;
  BUILD_BUG_ON(__same_type(tcp_sendmsg, jtcp_sendmsg) == 0);
  if(!proc_create(procname, S_IRUSR, init_net.proc_net, 0))
    return ret;
  ret = register_jprobe(&mptcp_jprobe);
  if (ret) {
    remove_proc_entry(procname, init_net.proc._net);
    retrun ret;
  }
  return 0;
}  // (iii) Init function
module_init(mptcp_scramble_init);

static __exit void mptcp_scramble_exit(void) {
  remove_proc_entry(procname, init_net.proc._net);
  unregister_jprobe(&mptcp_jprobe);
}  // (iv) Exit function
module_exit(mptcp_scramble_exit);
```

Figure 6. JProbe hander definition for `tcp_sendmsg()`.

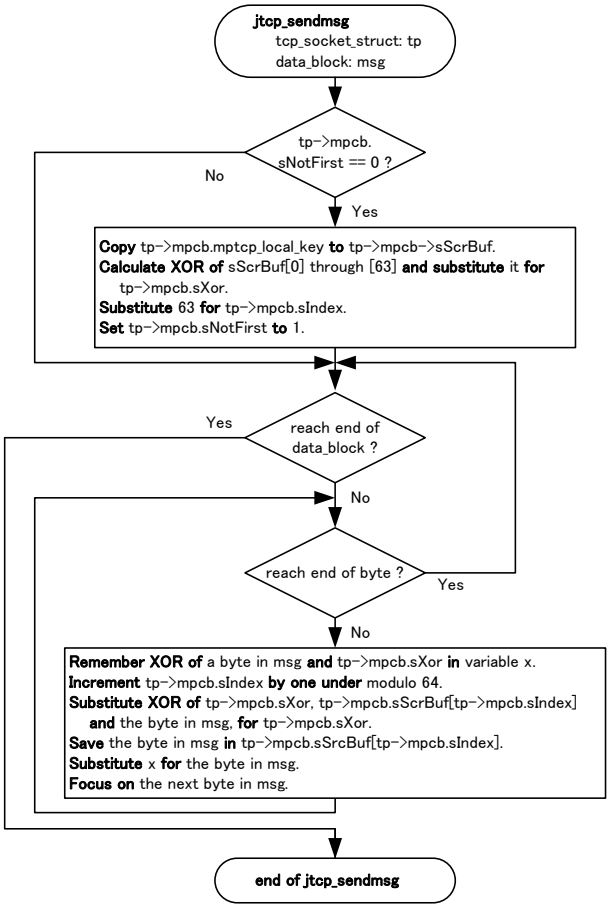


Figure 7. Flowchart of data scrambling.

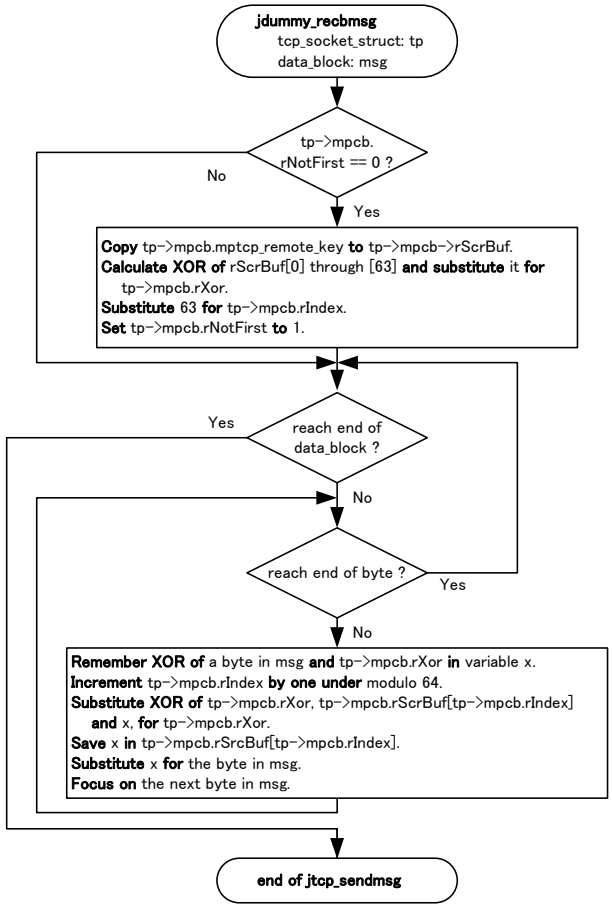


Figure 8. Flowchart of data descrambling.

procedure. When `jtcp_sendmsg()` is called, it is checked whether this function is invoked for the first time or not. If it is the first invocation over a specific MPTCP connection, `sScrBuf[]` is initialized to the value of the local key maintained in the `struct mptcp_cb` structure. Then, XOR of all the bytes in `sScrBuf[]` is calculated and saved in `sXor`, and `sIndex` is set to 63.

The argument containing data (`msg`) is a list of data blocks, and so individual blocks are handled sequentially. For each data block, a byte-by-byte basis calculation is performed in the following way. First, the XOR of the focused byte and `sXor` is saved in temporal variable `x`. Then, `sIndex` is advanced by one under modulo 64. Thirdly, the XOR of `sXor`, `sScrBuf[sIndex]` and the original byte are calculated and saved in sXor. It should be noted that the value in `sScrBuf[sIndex]` at this stage is the oldest value in the send scramble buffer. Fourthly, the original byte is stored in `sScrBuf[sIndex]`,which means that the send scramble buffer is updated. At last, the byte in the message block is replaced by the value of `x`.

D. Implementation of descrambling

The data descrambling is implemented similarly with scrambling. We developed the JProbe handler for function `dummy_recvmsg()` in the same way with the approach given in Figure 6. The flowchart of descrambling procedure is shown in Figure 8. This is similar with the flowchart shown in Figure 7. In the first part of the flowchart, it should be noted that `rScrBuf[]` is set to the remote key, which is the local key in the sender side. In this case, the data block is a descrambled data. Therefore, in the byte-by-byte basis part, the original value (`x` in the figure) is used to calculate `rXor` and is stored in `rSrcBuf[rIndex]`.

IV. EXPERIMENT

We implemented the proposed method over the Linux operating system (Ubuntu 16.04 LTS). We evaluated it in the experimental configuration shown in Figure 9. Two Panasonic Let's note PCs are used as a client and a server. The processor types are Intel UPU U1300 with 1.06GHz and Intel Pentium M with 1.50 GHz. The client PC is connected with an access point (Buffalo Air Station G54) through WLAN and Ethernet. On the other hand, the server PC is connected with the access point through Ethernet. We used 802.11g with 2.4 GHz as WLAN and 100base-T as Ethernet. The WLAN interface does not use any encryption. We suppose that the Ethernet link is a trusted network and the WLAN link without any encryption is an untrusted network. A MacBook Air with macOS High Sierra is used as an attacker. It runs Wireshark to capture packets sent over WLAN.

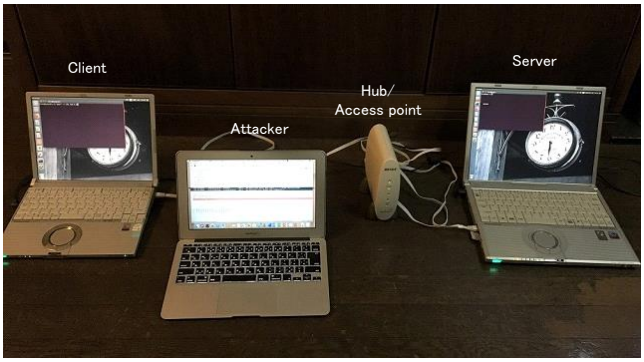


Figure 9. Experiment configuration.

The network setting is as follows.

- Since the access point works as a bridge, the client and the server are connected to the same subnetwork, 192.168.0.0/24.
- The Ethernet and WLAN interfaces in the client are assigned with IP addresses 192.160.0.1 and 192.168.0.3, respectively. The Ethernet interface in the server is assigned with IP address 192.168.0.2. The ESSID of the WLAN is "MPTCP-AP."
- In order to use two interfaces at the client, the IP routing tables are set for individual interfaces, by use of the ip command in the following way (for the Ethernet interface `enp4s1`).
  - `ip rule add from 192.168.0.1 table 1`
  - `ip route add 192.168.0.0/24 dev enp4s1 scope link table 1`
- The JProbe handlers for `jtcp_sendmsg()` and `jdummy_recvmsg()` are built as kernel modules. They are inserted and removed using `insmod` and `rmmod` Linux commands without rebooting the system.
- In the experiment, we used `iperf` for sending data from the client to the server, using Ethernet and WLAN.
- In the attacker, the Wireshark network analyzer is invoked for monitoring a WLAN interface with the monitor mode set to effective.

Figure 10 shows a result of the attacker's monitoring of iperf communication over WLAN in the conventional communication. In the iperf communication, an ASCII digit sequence "0123456789" is sent repeatedly. If the attacker can monitor the WLAN, the content is disposed as shown in this figure. Figure 11 shows a monitoring result by the attacker over the WLAN link when the data scrambling is performed. This figure shows the monitoring result for the first data segment over the WLAN link, which is the same with Figure 10. The original data is a repetition of "0123456789" but the data is scrambled in the result here. So, it can be said that the attacker cannot understand the content, even the WLAN link is not encrypted.

As for the throughput of iperf communication, we executed ten times evaluation runs. The results are as follows.
Without scrambling: 89.92 Mbps average, 1.19 Mbps STD.
With scrambling: 86.04 Mbps average, 1.69 Mbps STD.
Since the processor types used in the experiment are rather old, the processing of scrambling and descrambling provided some overhead. But we believe that the throughput reduction is small.

V. CONCLUSIONS

This paper described the results of implementation and evaluation of a method to improve privacy against eavesdropping over MPTCP communications, which we proposed in the previous papers. The proposed method here is based on the not-every-not-any protection principle, that is, *if an attacker cannot observe the data over trusted path such as an LTE network, he cannot observe the traffic on any path*. Specifically, the proposed method uses the byte oriented data scrambling and the data dispersion over multiple paths.

In the implementation of the proposed method, we took an approach to avoid the modification of the Linux kernel as much as possible. The modification is as follows. The control

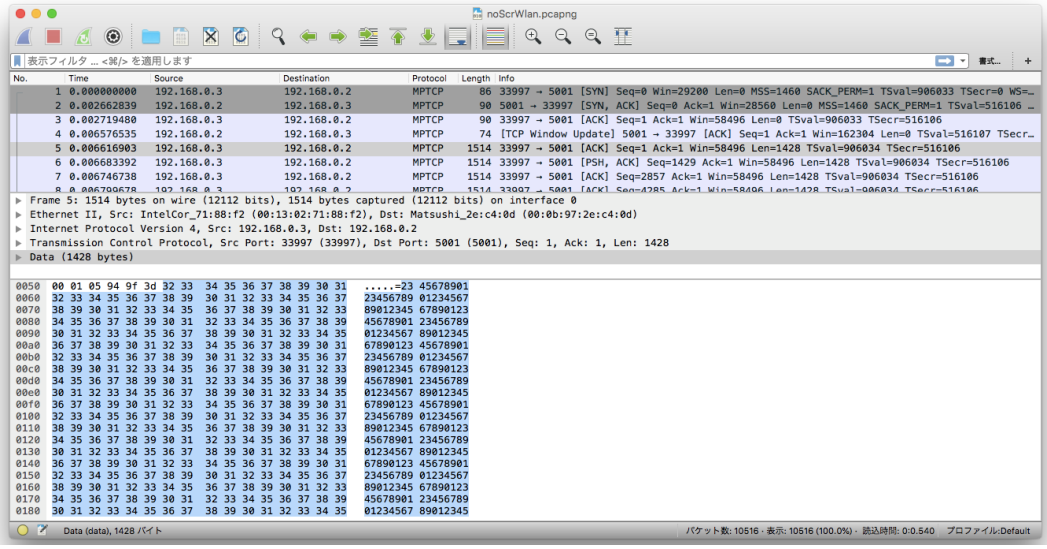


Figure 10. Capturing result when no scrambling is performed.

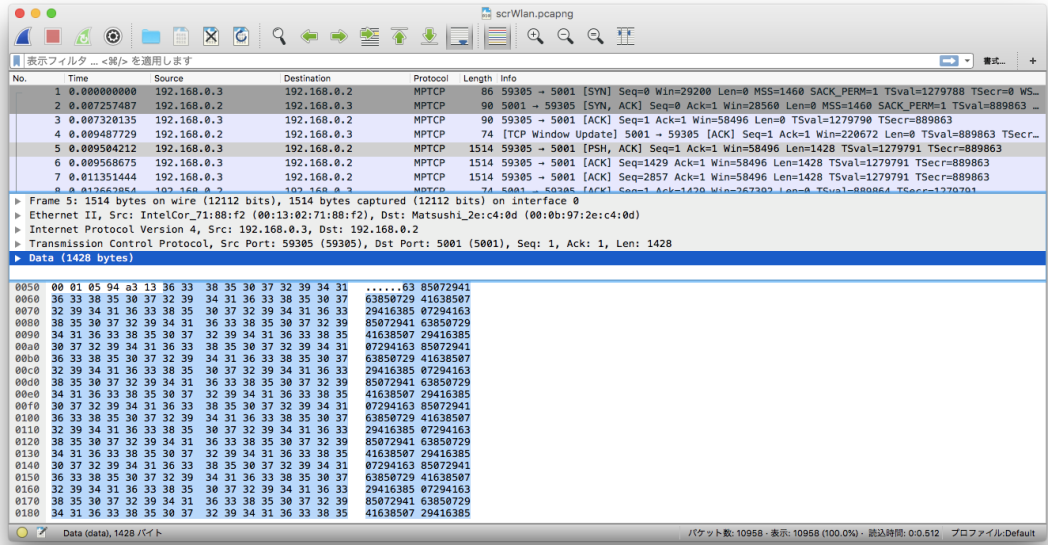


Figure 11. Capturing result when scrambling is performed.

parameters are inserted in the socket data structure, and the dummy function for the last part of `tcp_recvmsg()` function. The main part of scrambling and descrambling is implemented by use of the kernel debugging routine called JProbe handler, which is independent of the kernel.

Through the experiment, we confirmed that the data transferred over unencrypted WLAN link cannot be recognized when the data scrambling is performed. As for the performance, the throughput of the scrambled communication is just a little smaller than the conventional communication exposed to unauthorized access.

ACKNOWLEDGMENT

This research was performed under the research contract of "Research and Development on control schemes for utilizations of multiple mobile communication networks," for the Ministry of Internal Affairs and Communications, Japan.

REFERENCES

[1] NGNM Alliance, "NGMN 5G White Paper," https://www.ngmn.org/fileadmin/ngmn/content/downloads/Technical/2015/NGMN_5G_White_Paper_V1_0.pdf, Feb. 2015, [retrieved: Jul., 2018].

[2] C. Paasch and O. Bonaventure, "Multipath TCP," Communications of the ACM, vol. 57, no. 4, pp. 51-57, Apr. 2014.

[3] AppleInsider Staff, "Apple found to be using advanced Multipath TCP networking in iOS 7," http://appleinsider.com/articles/13/09/20/apple-found-to-be-using-advanced-multipath-tcp-networking-in-ios-7, [retrieved: Jul, 2018].

[4] icteam, "MultiPath TCP – Linux Kernel implementation, Users:: Android," https://multipath-tcp.org/pmwiki.php/Users/Android, [retrieved: Jul., 2018].

[5] A. Ford, C.Raiciu, M. Handley, S. Barre, and J. Iyengar, "Architectural Guidelines for Multipath TCP Development," IETF RFC 6182, Mar. 2011.

[6] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses," IETF RFC 6824, Jan. 2013.

[7] C. Raiciu, M. Handley, and D. Wischik, "Coupled Congestion Control for Multipath Transport Protocols," IETF RFC 6356, Oct. 2011.

[8] T. Kato, S. Cheng, R. Yamamoto, S. Ohzahata, and N. Suzuki, "Protecting Eavesdropping over Multipath TCP Communication Based on Not-Every-Not-Any Protection," in Proc. SECURWARE 2017, pp. 82-87, Sep. 2017.

[9] T. Kato, S. Cheng, R. Yamamoto, S. Ohzahata, and N. Suzuki, "Proposal and Study on Implementation of Data Eavesdropping Protection Method over Multipath TCP Communication Using Data Scrambling and Path Dispersion," International Journal On Advances in Security, 2018 no. 1&2, pp. 1-9, Jul., 2018.

[10] C. Pearce and S. Zeadally, "Ancillary Impacts of Multipath TCP on Current and Future Network Security," IEEE Internet Computing, vol. 19, iss. 5, pp. 58-65, Sept.-Oct. 2015.

[11] J. Yang and S. Papavassiliou, "Improving Network Security by Multipath Traffic Dispersion," in Proc. MILCOM 2001, pp. 34-38, Oct. 2001.

[12] M. Nacher, C. Calafate, J. Cano, and P. Manzoni, "Evaluation of the Impact of Multipath Data Dispersion for Anonymous TCP Connections," In Proc. SecureWare 2007, pp. 24-29, Oct. 2007.

[13] A. Gurtov and T. Polishchuk, "Secure Multipath Transport For Legacy Internet Applications," In Proc. BROADNETS 2009, pp. 1-8, Sep. 2009.

[14] L. Apiecionek, W. Makowski, M. Sobczak, and T. Vince, "Multi Path Transmission Control Protocols as a security solution," in Proc. 2015 IEEE 13th International Scientific Conference on Informatics, pp. 27-31, Nov. 2015.

[15] LWN.net, "An introduction to KProbes," https://lwn.net/Articles/132196/, [retreieved: Jul., 2018].

[16] GitHubGist, "jprobes example: dzeban / jprobe_etn_io.c," https://gist.github.com/dzeban/a19c711d6b6b1d72e594, [retreieved: Jul., 2018].

[17] S. Seth and M. Venkatesulu, "TCP/IP Architecture, Desgn, and Implementation in Linux," John Wiley & Sons, 2009.

 ISBN: 978-1-61208-661-3