Implementation and Performance Evaluation 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) Graduate School of Informatics and Engineering
University of Electro-Communications
Tokyo, Japan
2) Adaptive Communications Research Laboratories
Advanced Telecommunication Research Institute International
Kyoto, Japan
kato@net.lab.uec.ac.jp, chengshihan@net.lab.uec.ac.jp, ryo_yamamotog@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 in our previous papers. The proposed method takes an approach that, if an attacker cannot observe the data on *every* path, he/she 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 in more detail than our former papers.**

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

I. INTRODUCTION

This paper is an extension of our previous paper [1], which is presented in an IARIA conference.

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) mobile network, it is expected that multiple communication paths provided by multiple network operators are commonly involved [2]. In this case, mobile terminals will have more than two interfaces.

However, the conventional TCP establishes a connection between single IP addresses at individual ends, and so it cannot utilize multiple interfaces in one end at the same time. In order to cope with this issue, MPTCP [3] is being introduced in several operating systems, such as Linux, Apple OS/iOS [4] and Android [5]. MPTCP is an extension of the conventional TCP. It combines multiple TCP flows into one data stream called an MPTCP connection, and provides the same programing interface with the socket interface. So, existing TCP applications can use MPTCP as if they were working over conventional TCP.

MPTCP is defined by three Request for Comments (RFC) documents by the Internet Engineering Task Force. RFC 6182 [6] outlines architecture guidelines. RFC 6824 [7] 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 [8] 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 generally 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 [9][10]. 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/she cannot observe the traffic on *any* path [11]. 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 [12]-[15], 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 our previous paper [1] that is the origin of this paper, we showed how to implement the proposed method over the Linux operating system. We used the kernel debugging mechanism called *JProbe*, in order to avoid the modification of the Linux kernel as much as possible. The previous paper

also showed the results of implementation focusing on how the proposed method works over off-the shelf personal computers and access point, but the descriptions on performance evaluation was limited.

In this paper, we describe the proposed method and its implementation in more detail. We also show another behavior of the proposed method and the results of performance evaluation in detail.

The rest of this paper is organized as follows. Section II explains the overview and the security issue of MPTCP [10]. Section III describes the proposed method. Section IV shows how to implement the proposed method within the MPTCP software in the Linux operating system. Section V gives the behavior of the proposed method and the results of the performance evaluation. In the end, Section VI concludes this paper.

II. Overview and Security Issues of MPTCP

A. MPTCP connections and subflows

As described in Figure 1, the MPTCP module is located on top of TCP. As described above, MPTCP is designed so that the conventional applications do not need to care about the existence of MPTCP. MPTCP establishes an *MPTCP connection* associated with two or more regular TCP connections called *subflows*. The management and data transfer over an MPTCP connection is done by newly introduced TCP options for MPTCP operation.

Figure 2 shows an example of MPTCP connection establishment where host A with two network interfaces invokes this sequence for host B with one network interface. In the beginning, host A sends a SYN segment to host B with a *Multipath Capable (MP_CAPABLE)* TCP option. This option indicates that an initiator supports the MPTCP functions and requests to use them in this TCP connection. It contains host A's *Key* (64 bits) used by this MPTCP connection. Then, host B replies a SYN+ACK segment with MP_CAPABLE option with host B's Key. This reply means that host B accepts the use of MPTCP functions. In the end, host A sends an ACK segment with MP_CAPABLE option including both A's and B's Keys. Through this three-way handshake procedure, the first subflow and the MPTCP connection are established. Here, it should be mentioned that these "Keys" are not keys in a cryptographic sense. They are a sort of random numbers assigned for individual MPTCP connections. As described below, they are used for generating the Hash-based Message Authentication Code (HMAC), but MPTCP does not provide any mechanisms to protect them from attackers' accessing while transfer.

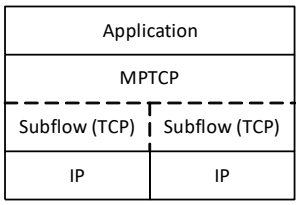


Figure 1. Layer structure of MPTCP.

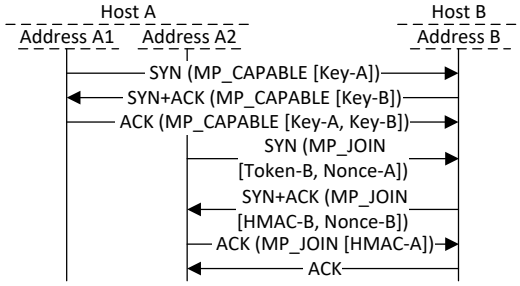


Figure 2. Example of MPTCP connection establishment.

Next, host A tries to establish the second subflow through another network interface. In the first SYN segment in this try, another TCP option called a *Join Connection (MP_JOIN)* option is used. An MP_JOIN option contains the receiver's *Token* (32 bits) and the sender's *Nonce* (random number, 32 bit). A Token is an information to identify the MPTCP connection to be joined. It is obtained by taking the most significant 32 bits from the SHA-1 hash value for the receiver's Key (host B's Key in this example). Then, host B replies a SYN+ACK segment with MP_JOIN option. In this case, MP_JOIN option contains the random number of host B and the most significant 64 bits of the HMAC value. An HMAC value is calculated for the nonces generated by hosts A and B using the Keys of A and B. In the third ACK segment, host A sends an MP_JOIN option containing host A's full HMAC value (160 bits). In the end, host B acknowledges the third ACK segment. Using these sequence, the newly established subflow is associated with the MPTCP connection.

B. Data transfer

An MPTCP implementation will take one input data stream from an application, and split it into one or more subflows, with sufficient control information to allow it to be reassembled and delivered to the receiver side application reliably and in order. The MPTCP connection maintains the *data sequence number* independent of the subflow level sequence numbers. The data and ACK segments may contain a *Data Sequence Signal (DSS)* option depicted in Figure 3.

The data sequence number and data ACK is 4 or 8 byte long, depending on the flags in the option. The number is assigned on a byte-by-byte basis similarly with the TCP sequence number. The value of data sequence number is the number assigned to the first byte conveyed in that TCP segment. The data sequence number, subflow sequence number (relative value) and data-level length define the mapping between the MPTCP connection level and the subflow level. The data ACK is analogous to the behavior of the standard TCP cumulative ACK. It specifies the next data sequence number a receiver expects to receive.

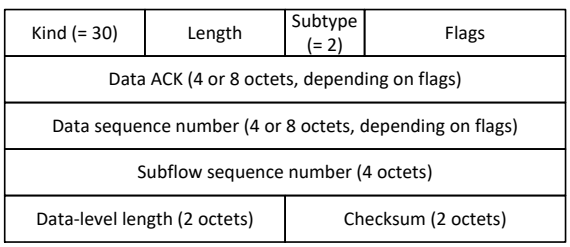


Figure 3. Data Sequence Signal option.

C. Eavesdropping protection over multiple TCP connections

As we mentioned in Section I, there are several proposals on the data dispersion over multiple paths. Yang and Papavassiliou [12] showed a method to analyze the security performance when a virtual connection takes multiple disjoint paths to the destination, and proposed a traffic dispersion scheme to minimize the information leakage when some of the intermediate routers are attacked. Nacher et al. [13] tried to determine the optimal trade-off between traffic dispersion and TCP performance over mobile ad-hoc networks to reduce the chances of successful eavesdropping while maintaining acceptable throughput. These two studies use multiple TCP connections by their own coordination methods instead of MPTCP. Gurtov and Polishchuk [14] used host identity protocol (HIP), which locates between IP and TCP to provide multiple paths, and proposed how to spread traffic over them. Apiecionek et al. [15] proposed a way to use MPTCP for more secure data transfer. After data are encrypted, they are divided into blocks, mixed in the predetermined random sequence, and then transferred through multiple MPTCP subflows. A receiver rearranges received blocks in right order and decrypts them.

All of those proposals aim at just spreading data packets over multiple paths, and do not consider the coordination over multiple paths. If the transferred data are encrypted before dispersion, it can be said that they are coordinated by the encryption procedure, but the coordination is not realized by the dispersion schemes. In contrast with them, our proposal adopts an approach to improve privacy by coordinating data over multiple paths through data scrambling not encryption.

III. Proposed Method

A. Motivation

The first point we considered in designing our proposal was utilize multiple paths supported by MPTCP. So we picked up the secret sharing method [16], which produces some number of pieces from data in a way that a specific number of pieces are required for reconstructing the original data. By using the secret sharing, it is possible to divide data into multiple subflows. However, in this approach, it is required to duplicate the original data or at least increase the amount of sending data. Besides, this approach requires some cryptographic calculation that uses a lot of CPU power.

Next approach we considered is the network coding approach [17], where a packet is XORed with the following packet and the first packet and the XOR result are sent via different paths [18]. This approach does not require any cryptographic calculation and so the processing overhead is low. However, if the packet length is different, unnecessary padding may be necessary.

The third approach is applying the mode of operation, such as Cipher Block Chaining (CBC) and Output Feedback (OFB), used in block ciphering [19]. The block cipher defines only how to encrypt or decrypt a fixed length bits (block). The mode of operation defines how to apply this operation to data longer than a block. CBR and OFB introduce a chaining between blocks such that a block is combined with the preceding block by XOR calculation. The application of the mode of operation without any encryption to data dispersion is considered as a block level data scrambling. So, if the length of packet is not integral multiple of block length, unnecessary padding will be required again.

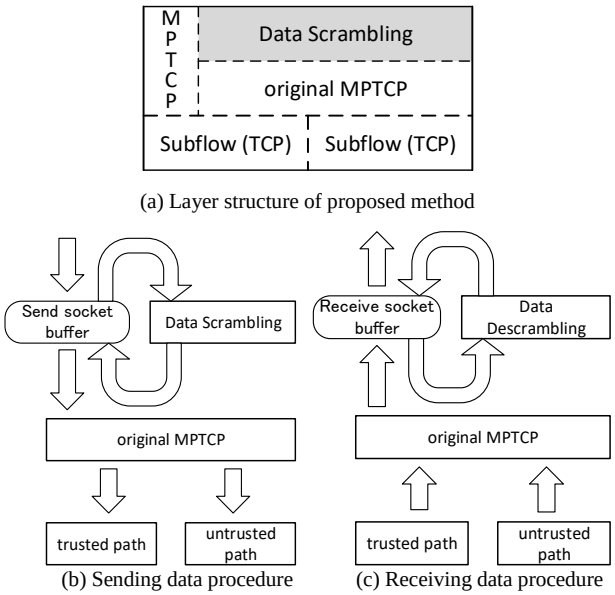
Based on these considerations, we picked up a byte level scrambling which is described in the following subsection.

B. Detailed procedure

Figure 4 shows the overview of the proposed method. As shown in Figure 4(a), we introduce a data scrambling function within MPTCP and on top of the original MPTCP. When an MPTCP communication is started, the use of data scrambling is negotiated. It may be done using a flag bit in MP_CAPABLE TCP option.

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



(a) Layer structure of proposed method

(b) Sending data procedure (c) Receiving data procedure

Figure 4. Overview of proposed method [9].

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 6 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.

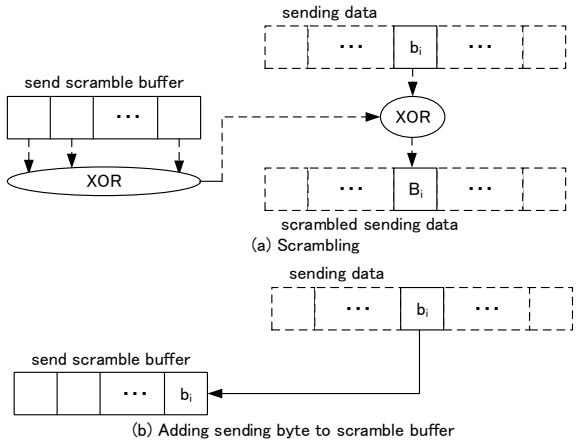


Figure 5. Processing of data scrambling [9].

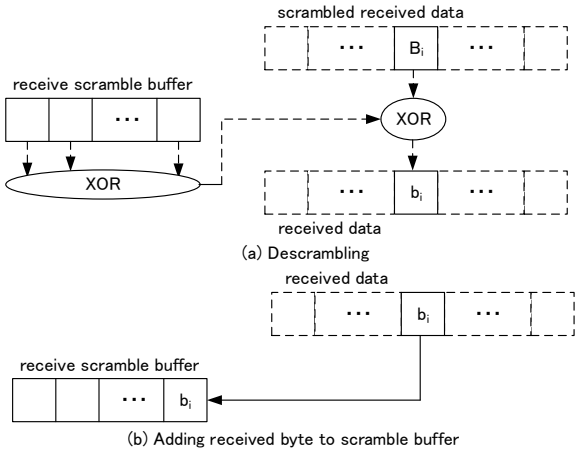


Figure 6. Processing of data descrambling [9].

IV. 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 [20].

Among kernel probes methods, we use a way called "JProbe" [21]. 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 [20].

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()` [21].

In the Linux kernel, function `tcp_sendmsg()` is called when an application sends data to MPTPCP (actually TCP, too) [22]. 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 7. It is a function just returning and inserted before function `tcp_recvmsg()` releases the socket control. Since this function is very simple, "`noinline`" indication pragma needs to be specified. 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 [22]. 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 below.

```
struct mptcp_cb *mpcb;
```

So, we added the control variables for data scrambling in this data structure. Figure 8 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 9 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.

```
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, copied);
  release_sock(sk);
  return copied;
  . . . .
} // dummy_recvmsg() inserted
EXPORT_SYMBOL(tcp_recvmsg);

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

Figure 7. Dummy function in `tcp_recvmsg()`.

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

Figure 8. Control variables for data scrambling/descrambling.

```
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 9. JProbe hander definition for `tcp_sendmsg()`.

(2) Flowchart of data scrambling

The data scrambling procedure is implemented in `jtcp_sendmsg()`. Figure 10 shows the flowchart for this 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 9. The flowchart of descrambling procedure is shown in Figure 11. This is similar with the flowchart shown in Figure 10. In the first part of the flowchart, it should

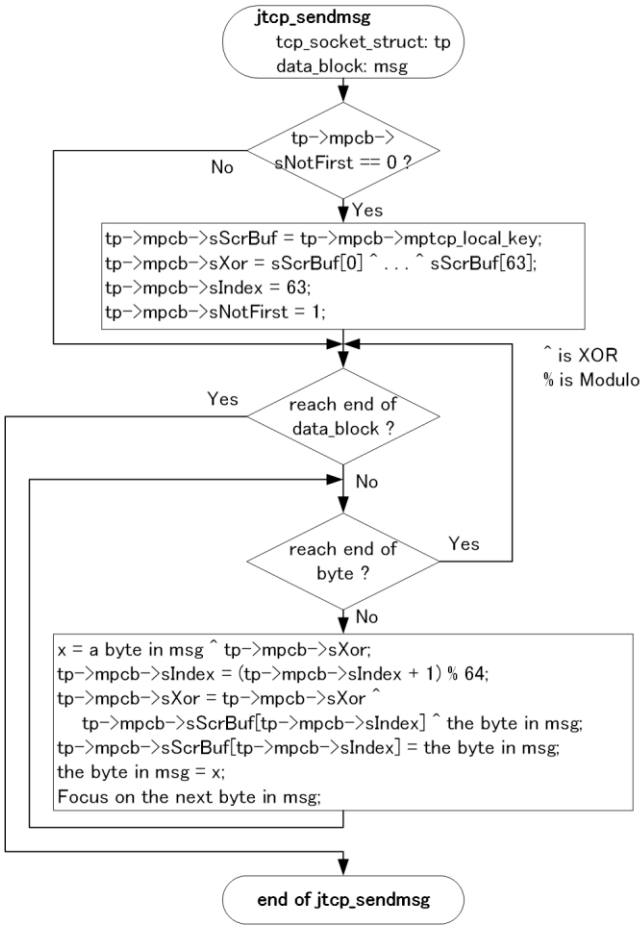


Figure 10. Flowchart of data scrambling.

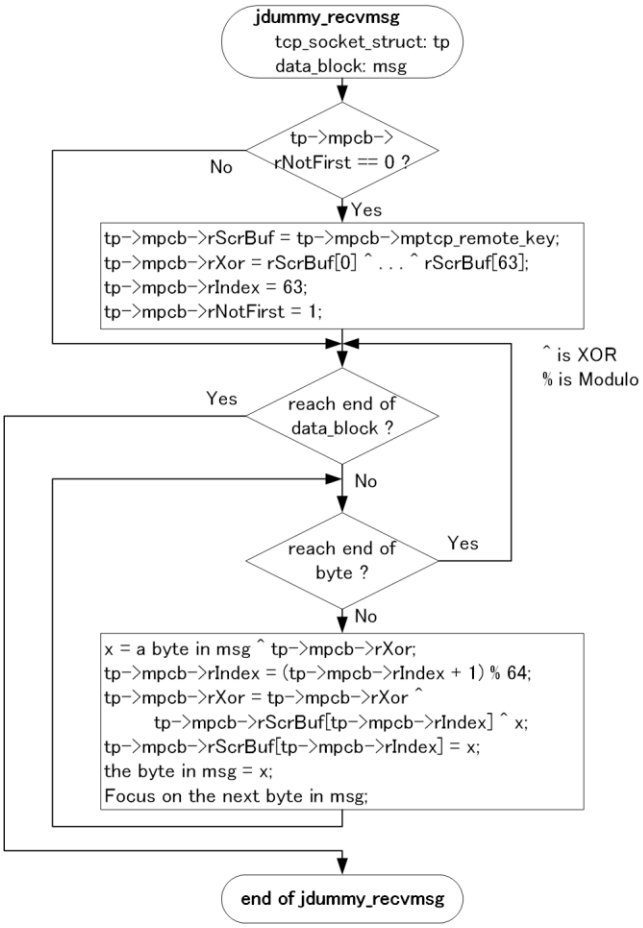


Figure 11. Flowchart of data descrambling.

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]`.

V. Experiment and Performance Evaluation

A. Experimental settings

We implemented the proposed method over the Linux operating system (Ubuntu 16.04 LTS/14.04 LTS with MPTCP support). We evaluated it in the experimental configuration shown in Figure 12. Two note PCs are used as a client and a server. They are connected with each other via a hub/access point using Ethernet and WLAN. Ethernet is 100base-T and WLAN is IEEE 802.11g with 2.4 GHz. The client uses both Ethernet and WLAN, and the server uses only 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. Table I shows the specification of nodes. It should be noted that the model for the client and server nodes is a little old and the processing power of CPU is low. The model of the access point is Buffalo Air Station G54. The proposed method is implemented in the Linux operating system running over the client and server nodes. In the attacker node, Wireshark (and tshark) is executed in order to monitor data transferred over WLAN.

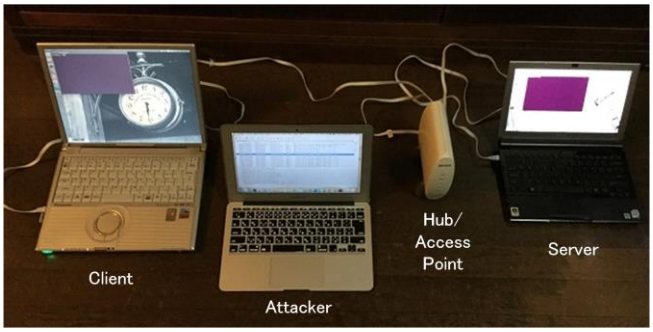


Figure 12. Experiment configuration.

TABLE I. SPECIFICATION OF NODES USED IN EXPERIMENT.

Node	Model	CPU	Operating system
Client	Panasonic Let's note CF-Y4	Intel Pentium M 1.50 GHz	ubuntu 16.04 LTS
Server	Sony Vaio PCG-4P7N	Intel Core 2 Duo U7700 1.33 GHz x 2	ubuntu 14.04 LTS
Attacker	Apple MacBook Air	Intel Core i5 1.7 GHz	macOS High Sierra

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 another evaluation, we used a simple file transfer, which we implemented, where a specified file is transferred from the server to the client.
- In the attacker node, the Wireshark network analyzer is invoked for monitoring a WLAN interface with both the promiscuous mode the monitor mode set to effective. When we use tshark at the attacker node, which is a command line interface version of Wireshark, the "`-I`" option is used for capturing all WLAN frames and the "`-Y`" option is used for applying a display filter defined similarly with Wireshark.

B. *Behaviors of proposed method*

Figure 13 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 14 shows a monitoring result by the attacker over the WLAN link when the data scrambling is performed.

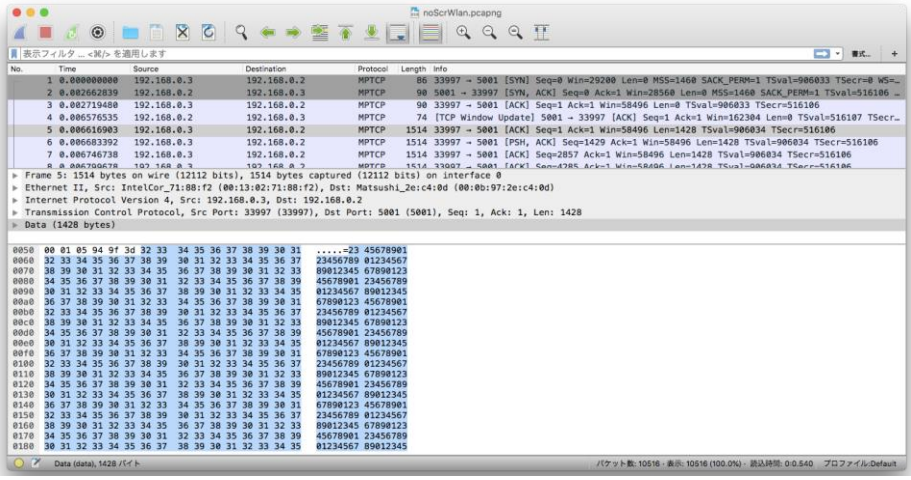


Figure 13. Capturing result of iperf when no scrambling is performed.

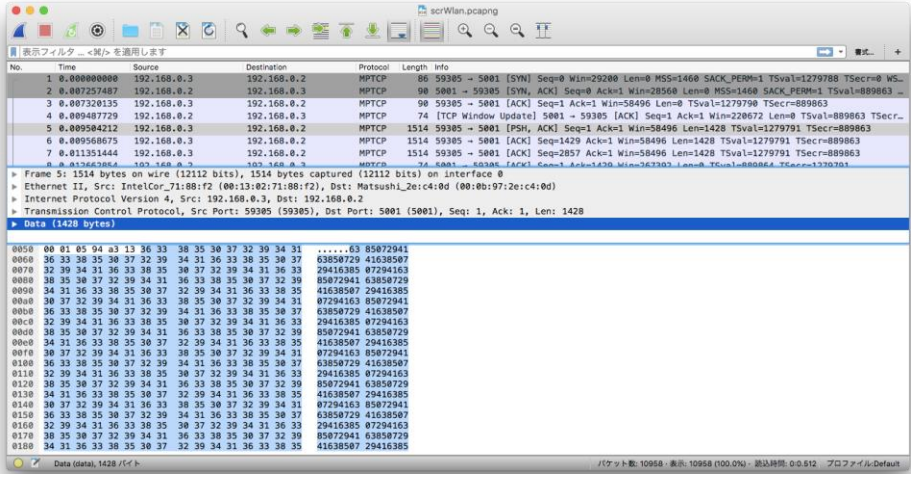


Figure 14. Capturing result of iperf when scrambling is performed.

This figure shows the monitoring result for the first data segment over the WLAN link, which is the same with Figure 13. 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.

Figures 15 through 17 show result of the attacker's monitoring of file transfer over WLAN. Figure 15 is a display image at the client node when it is receiving a text file containing the text of our previous paper. It is the part of references in the paper. Figure 16 shows a monitoring result by the attacker when no data scrambling is performed. This figure is a result with tshark command with "-x" option that requests to display of data part in TCP segments. As shown in this figure, the content of file can be obtained by the attacker. Figure 17 shows the monitoring result by the attacker for the same part of file given in Figure 16. Since the content is scrambled, the attacker cannot recognize the content of the file.

C. Throughput evaluation

In order to evaluation the performance of the proposed method, we measured file transfer throughput for the original MPTCP (without data scrambling nor encryption), the proposed method, and the MPTCP data transfer with encryption and decryption using Advanced Encryption Standard (AES) [23]. AES is a block based ciphering algorithm standardized by NIST in 2001. It is a symmetric block cipher that can process data blocks of 128 bits (16 bytes), using cipher keys with lengths of 128, 192, and 256 bits (16 bytes, 24 bytes, and 32 bytes, respectively). In this experiment, we used 16 byte key with the CBC mode. For the implementation of AES based file transfer, we used a publicly available source programs for AES [24] distributed by PJC, a

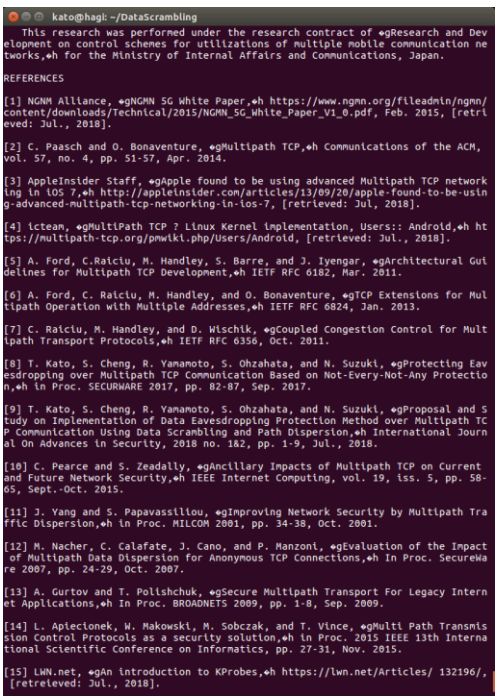


Figure 15. Client display image of file receiving.

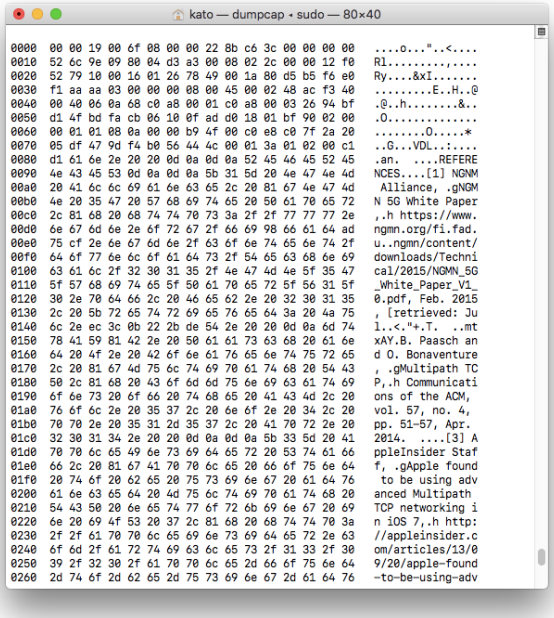


Figure 16. Capturing result of file transfer without scrambling.

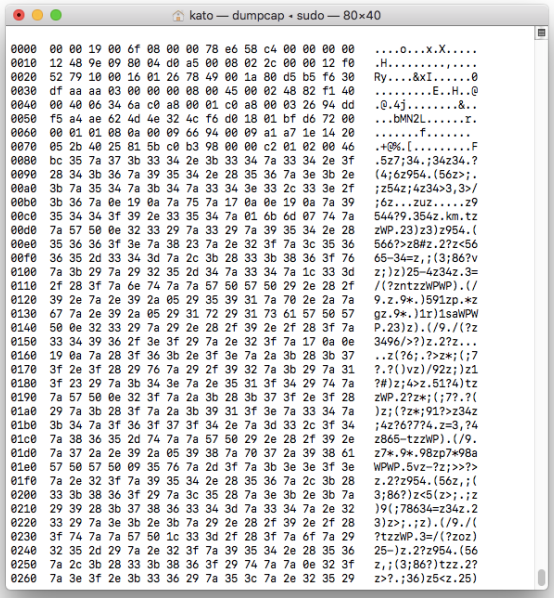


Figure 17. Capturing result of file transfer with scrambling.

Japanese software company. In the throughput measurement, data is transferred from the server node to the client node in the configuration given in Figure 12. We measured the throughput by changing the size of file transferred.

Figure 18 shows the measured throughput. When changing the transferred file size from 20 MB through 80 MB, the results are similar for all the sizes in every case. The

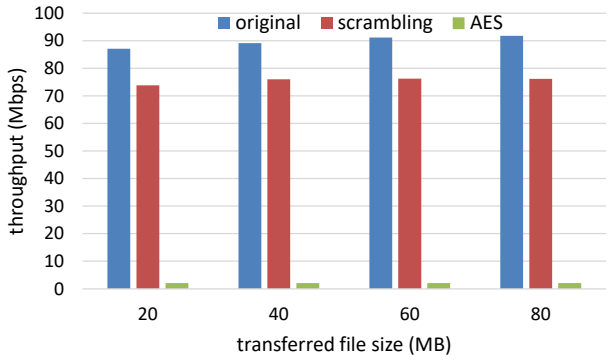


Figure 18. Measured throughput.

throughput of the original MPTCP is the highest among three and the result is around 90 Mbps. In the case of the proposed method, the measured throughput is around 75 Mbps, which is lower than the original MPTCP but high enough. In the case of AES encryption, the throughput goes down to 2 Mbps. Comparing the encryption and decryption, the decryption has higher overhead in our experiment, and it limits the performance of file transfer. It should be noted that the software we used for AES encryption may not be optimized and so the sophisticated AES program may improve the throughput. But, it can be said that the proposed method will require much less overhead than the encryption based method. Another thing to be mentioned is that the equipment we used in this experiment is rather old and so the processing power of CPU is not high. This is one factor that makes the throughput of the AES based method worse.

VI. Conclusions

This paper described the results of implementation and performance 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/she 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 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 original MPTCP communication exposed to unauthorized access. Moreover, the throughput of cryptographic method is degraded largely compared with the original MPTCP and the proposed method. For the terminals with low power CPU, the cryptographic approach will decrease the throughput and so the proposed method will be effective.

In the last of this paper, we need to say that the proposed method is a practical approach based on the assumption that the trusted path, for example, a path via a trusted network operator, is safe enough. That means that, if the trusted path is accessed in an unauthorized way, the data will be observed thoroughly. By owing to the safety in the trusted networks, the proposed method provides low overhead in the data protection.

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] T. Kato, S. Cheng, R. Yamamoto, S. Ohzahata, and N. Suzuki, “Implementation of Eavesdropping Protection Method over MPTCP Using Data Scrambling and Path Dispersion,” in Proc. SECURWARE 2018, pp. 108-113. Sep. 2018.

[2] 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: Feb., 2019].

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

[4] 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: Feb., 2019].

[5] icteam, “MultiPath TCP – Linux Kernel implementation, Users:: Android,” https://multipath-tcp.org/pmwiki.php/Users/Android, [retrieved: Feb., 2019].

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

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

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

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

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

[15] 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.

[16] A. Shamir, "How to share a secret," Communications of the ACM, vol. 22, no. 11, pp. 612-613, Nov. 1979.

[17] R. Ahlswede, N. Cai, S. Li, and R. Yeung, "Network Information Flow," IEEE Trans. Information Theory, vol. 46, no. 4, pp. 1204-1216, Jul. 2000.

[18] M. Li, A. Lukyanenko, and Y. Cui, "Network Coding Based Multipath TCP," in Proc. Global Internet Symposium 2012, pp. 25-30, Mar. 2012.

[19] ISO JTC 1/SC27, "ISO/IEC 10116: 2006 – Information technology – Security techniques – Modes of operation for an n-bit cipher," ISO Standards, 2006.

[20] LWN.net, "An introduction to KProbes," https://lwn.net/Articles/132196/, [retreieved: Feb., 2019].

[21] GitHubGist, "jprobes example: dzeban / jprobe_etn_io.c," https://gist.github.com/dzeban/a19c711d6b6b1d72e594, [retreieved: Feb., 2019].

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

[23] Federal Information Processing Standards Publication 197, "Anouncing the Advanced Encryption Standard (AES)," Nov. 2001.

[24] PJC, "Distribution of Sample Program / Source / Software (in Japanese)," http://free.pjc.co.jp/index.html, [retrieved: Feb., 2019].