Proposal and Study on Implementation of Data Eavesdropping Protection Method over Multipath TCP Communication 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@is.uec.ac.jp, chengshihan@net.is.uec.ac.jp, ryo_yamamotog@is.uec.ac.jp,
ohzahata@is.uec.ac.jp, nu-suzuki@atr.jp

***Abstract*— Recent mobile terminals have multiple interfaces, such as 4G and wireless local area network (WLAN). In order to use those interfaces at the same time, multipath transmission control protocol (MPTCP) is introduced in several operating systems. 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, in our previous paper, a new method to improve privacy against eavesdropping using the data dispersion by exploiting the multipath nature of MPTCP. One feature of the proposed method is to realize that an attacker cannot observe data on any path, even if he observes traffic over only a part of paths. Another feature is to use data scrambling instead of ciphering. In this paper, we present the design of this method and the results of performance evaluation. Besides, we discuss how to implement it inside the Linux operating system kernel, using a kernel debugging mechanism called JProbe.**

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

I. INTRODUCTION

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

Recently, mobile terminals with multiple interfaces have come to be widely used. For example, most smart phones are equipped with 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 [2]. In this case, mobile terminals will have more than two interfaces at the same time.

In order for applications to use multiple interfaces effectively, MPTCP [3] is being introduced in several operating systems, such as Linux, Apple OS/iOS [4] and Android [5]. MPTCP is an extension of TCP, and provides multiple byte streams through different interfaces. It is designed so as for conventional TCP applications to use MPTCP as if they were working over traditional TCP.

MPTCP is specified in three request for comments (RFC) documents provided by the Internet Engineering Task Force. RFC 6182 [6] outlines architecture guidelines for developing MPTCP protocols, by discussing the high level design decisions on selecting the protocol functions from multiple candidates. RFC 6824 [7] presents the details of extensions to the traditional TCP to support multipath operation. It defines the MPTCP control information realized as new TCP options, and the MPTCP protocol procedures for the initiation and association of subflows (TCP connections related with an MPTCP connection), the data transfer and acknowledgment over multiple subflows, and the closing 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 interfaces, i.e., 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 without any encryption at the WLAN level, data transferred over this WLAN may be disposed to other nodes connected to it. In order to prevent this eavesdropping, the transport layer security (TLS) is used to provide communication security. Although TLS can be applied to various applications including web access, e-mail and ftp, however, it is widely used only with HTTP, and some applications like VoIP cannot use TLS.

In order to avoid this eavesdropping, we proposed another approach to improve privacy against eavesdropping by exploiting the multipath nature of MPTCP. We called this approach a *not-every-not-any protection*, in our previous paper [1]. 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 such that if an attacker cannot observe the data on *every* path, he cannot observe the traffic on *any* path [9]. The feature of the proposed method is to adopt the not-every-not-any protection, and to use the data scrambling instead of ciphering.

In this paper, we present the proposed method in detail, and show the results of processing overhead of the data scrambling/descrambling in the proposed method, and the conventional encryption/decryption methods. We also discuss a study on how the proposed method is implemented in the MPTCP program inside the Linux operating system kernel.

The rest of this paper is organized as follows. Section II explains the overview [10] and the security issues of MPTCP. Section III describes the design of the proposed method protecting against eavesdropping. Section IV gives the performance evaluation on the processing overhead of the proposal method and other ciphering methods. Basically, the content in Sections II through IV comes from our previous paper [1]. Section V shows a study on how to implement the proposed method inside the Linux operating system kernel. 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. 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.

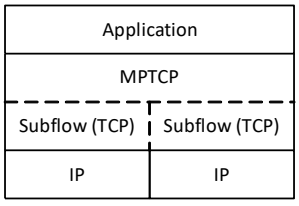


Figure 1. Layer structure of MPTCP.

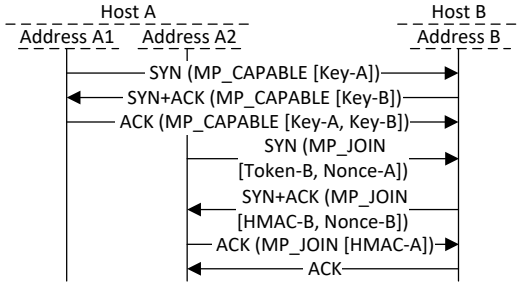


Figure 2. Example of MPTCP connection establishment.

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.

C. Security issues on MPTCP and related work

Some new security issues emerge by the introduction of MPTCP [8]. One is a new threat that an attacker splits malicious data over multiple paths. Traditional signature-based intrusion detection systems (IDSs) suppose that they can monitor all packets of a given flow. If a target system uses

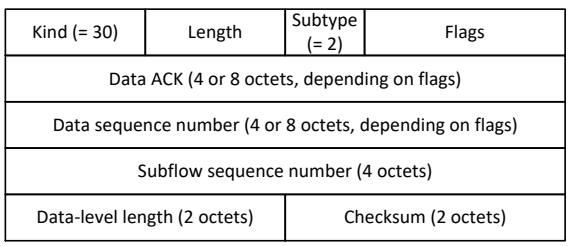


Figure 3. Data Sequence Signal option.

MPTCP and an attacker sends signatures over different subflows, IDSs cannot detect them. Ma et al. [11] proposed a new approach for this problem, where each IDS locally scans and processes its monitored traffic, and all IDSs share asynchronously a global state of string matching automaton.

Another issue is related to MPTCP and privacy. MPTCP has a potential to provide improved privacy against attackers who are able to observe or interfere with subflow traffic along a subset of paths. Dispersing traffic over multiple paths makes it less likely that attackers will get access to all of the data. Pearce and Zeadally [9] suggested the concept of the not-every-not-any protection and introduced some ideas including sending cryptographic signing details using multiple paths and applying cryptographic chaining, such as cipher block chaining (CBC), across multiple paths.

There have been several proposals on the data dispersion over multiple paths. Yang and Papavassiliou [12] provided a method to analyze the security performance when a virtual connection takes multiple disjoint paths to the destination, and 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 propose 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. PROPOSAL

A. Requirements and possible approaches

The following are the requirements for designing a not-any-not-every protection method protecting eavesdropping.

- The method needs to cope with two way data exchanges within one MPTCP connection.
- The length of exchanged data should not be expanded.
- Even if there are any bytes with known values, such as fixed bytes in an application protocol header, the method provides protection from information leakage.
- The method does not introduce any new overheads into MPTCP as much as possible.
- The method does not change the behaviors of MPTCP as much as possible.

In designing the proposed method, we have considered the following possible candidates.

(1) Secret sharing method

The secret sharing method is to divide data *D* into *n* pieces in such a way that *D* is easily reconstructed from any *k* pieces, but even complete knowledge of *k* -1 pieces reveals absolutely no information about *D* [16]. Shamir [16] gave an example method based on polynomial interpolation. It is possible to apply the idea of secret sharing to data transfer. Zhao et al. [17] proposed an efficient anonymous message submission protocol based on secret sharing and a symmetric key cryptosystem. It aggregates messages of multiple members into a message vector such that a member knows only his own position in the submission sequence.

Figure 4 shows an idea of applying secret sharing to the eavesdropping protection. It supposes the case that $n = 2$ and $k = 2$. Pieces D_1 and D_2 are generated from an original data and transferred through different paths. An attacker can access only D_2 over an untrusted path, and so he cannot obtain the original data. In this approach, however, the amount of transferred data is increased, twice in this example.

(2) Network coding

The second candidate is the network coding [18]. In this framework, the exclusive OR (XOR) is calculated among multiple packets and the result is transferred instead of packets themselves. Ahlswede et al. [18] mentioned that by employing coding at network nodes, which they referred to as network coding, it is possible to save bandwidth in general. Li et al. [19] proposed a network coding based multipath TCP (NC-MPTCP), which uses the mix of regular subflows, delivering original data, and network coding subflows, which deliver linear combinations of original data. NC-MPTCP

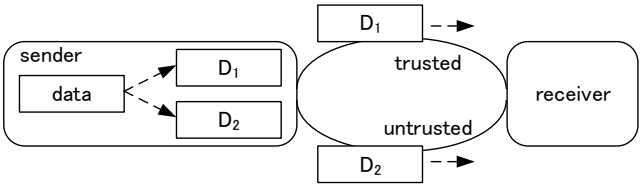


Figure 4. Secret sharing based approach.

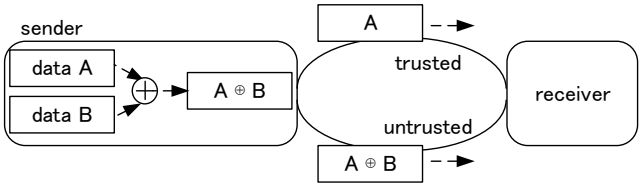


Figure 5. Network coding based approach.

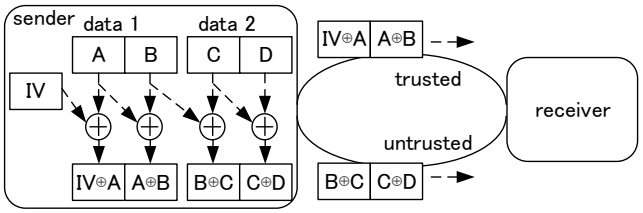


Figure 6. Block ciphering based approach.

achieves higher goodput compared to MPTCP in the presence of different subflow qualities.

Figure 5 shows an idea of applying network coding to the eavesdropping protection. Using data A and B, their XOR (A ⊕ B) is calculated. Through a trusted path, an original data A is transferred, and through an untrusted path, A ⊕ B is transferred. Since an attacker observes only A ⊕ B, he cannot obtain data B without knowledge of data A. This idea can be said a *packet level data scrambling*. Although it can provide the not-every-not-any protection, it introduces an additional overhead due to the variable length packets, and an additional control in MPTCP, such as sending XOR data only over an untrusted path.

(3) Mode of operation in block ciphering

The third candidate is the mode of operation, such as CBC and output feedback (OFB), used in block ciphering [20]. The block cipher defines only how to encrypt or decrypt a fixed length bits (block). A 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.

Figure 6 shows an idea of applying mode of operation to the eavesdropping protection. Data to be sent (data 1 and 2) are divided into blocks (A through D). The first block is XORed with the initialization vector (IV), and the following blocks are XORed with their preceding blocks. The XORed results are transferred via different paths. In the example, an attacker can only observe B ⊕ C and C ⊕ D, and does not know block B, which is transferred through a trusted path. So, he cannot obtain C and D any more. This idea can be said a *block level data scrambling*. Although it can provide the not-every-not-any protection, it introduces an additional data overhead because the length of packets is not integral multiple of block length in general.

According to those considerations, we select a byte stream based data scrambling approach described below that avoids the issues of the approaches described so far.

B. Detailed design of proposed method

As shown in Figure 7, 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.

Figure 8 shows an overview of data scrambling. In the data sending side, an application sends data to MPTCP. It is stored in the send socket buffer, and the data scrambling module scrambles it in a byte-by-byte basis. The result is stored in the send socket buffer again. The data in this buffer is transferred reliably by MPTCP. While sending data, MPTCP tries to send the first packet over an MPTCP connection via a subflow that uses a trusted path. After that, the data transfer by MPTCP is performed according to its native scheduler. We suppose that the distinction of trusted or untrusted path can be done by the IP address of interfaces. In the data receiving side, data is transferred through MPTCP without any losses, transmission errors, nor duplications. The received in-sequence data is stored in the receive socket buffer.

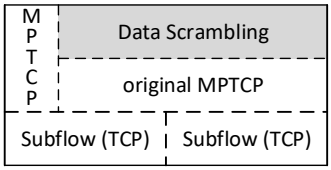
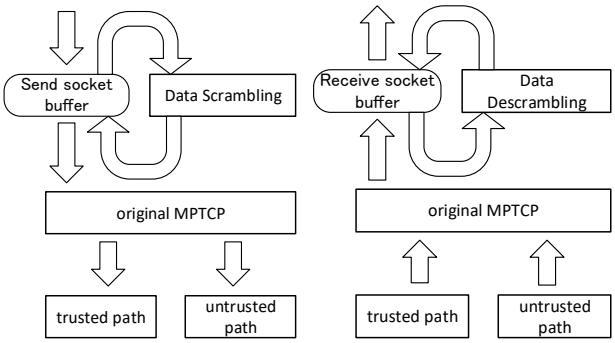


Figure 7. Layer structure of MPTCP with data scrambling.



(a) Sending data (b) Receiving data

Figure 8. Overview of data scrambling processing.

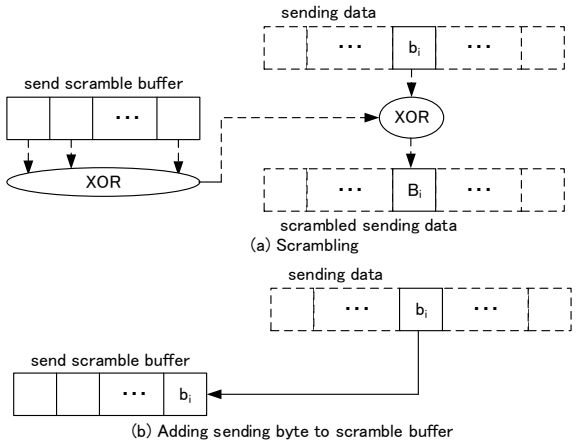


Figure 9. Procedure of data scrambling.

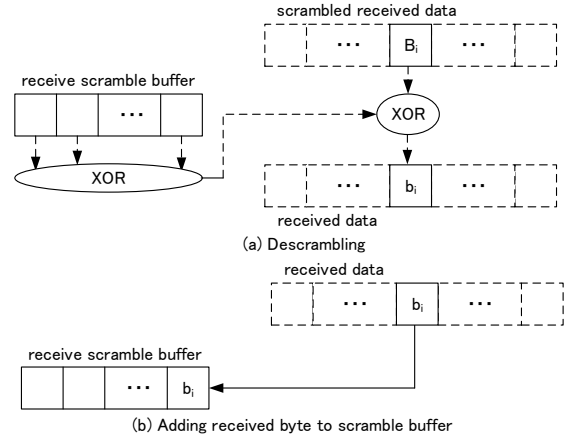


Figure 10. Procedure of data descrambling.

After that, the data descrambling module is invoked to restore the scrambled data to the original one.

Figure 9 shows the details of data scrambling. As described above, the scrambling is performed in a byte-by-

byte basis. More specifically, one byte being sent is XORed with its preceding 64 bytes. 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 the Key of this side. Since the length of the Key is 8 bytes, the higher bytes in the send scrambling buffer is filled by zero. 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 10 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 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.

C. Discussions

We need to discuss here about the security scheme of the proposed method. The proposed method does not use the data ciphering, and so it does not protect eavesdropping in a strict sense. It depends on the difficulty of unauthorized data access over networks provided by trusted operators. That is, the intrusion model is that an attacker can access only untrusted networks, such as public access point based WLANs, but he/she cannot access to trusted networks.

We also need to point out that the proposed method gives a small modification to MPTCP. It uses the HMAC value of sender side Key as an initial value of XORing, which means that no additional vulnerabilities are introduced for the initialization vector setting. Besides, as for the dependency between multiple paths that a byte cannot obtained only after the precedence bytes are received, it is intrinsic to MPTCP and is not a defect of the proposed method itself.

Another feature of the proposed method is that it does not introduce any additional control information at all. It just performs XORing a sending byte with bytes in the send scramble buffer. Even if data sending and data receiving are interleaved, the sending byte stream is focused and XORed beyond data receiving. The fact that data is delivered in sequence is assured by MPTCP, because the scrambling is done before data sending, and the descrambling is done after data receiving according to MPTCP.

IV. Performance Evaluation

In this section, we evaluate the processing overhead of the proposed method. In addition, we evaluate the overhead of commonly used cryptographic methods for the purpose of comparison. We adopt the data encryption standard (DES) [21], the triple data encryption algorithm (TDEA) [21], and the advanced encryption standard (AES) [22].

DES is a block based ciphering algorithm standardized by the National Institute of Standards and Technology (NIST). It is designed to encipher and decipher of blocks of data consisting of 64 bits (8 bytes) under control of a 64 bit (8 byte) key. Currently, it has been withdrawn as a standard ciphering method, but the TDEA, a compound operation of DES encryption and decryption operations, can be used as one of cipher suites in TLS.

AES is another block based ciphering algorithm newly 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 paper, we used publicly available source programs for DES and AES [23] distributed by PJC, a Japanese software company. They are written in C language. As for the DES algorithm, we prepared 160 blocks ($8 \times 160 = 1280$ bytes) and performed encryption and decryption for those blocks with the electronic codebook (ECB) mode. That is, each block is just encrypted and decrypted independently from other blocks. As for the TDEA algorithm, each of 160 blocks is encrypted or decrypted three times according to the DES algorithm with independent three keys. As for the AES algorithm, we prepared 80 blocks ($16 \times 80 = 1280$ bytes) and used keys with 128, 192 and 256 bit length (AES-128, AES-192 and AES-256). We also used the ECB mode here. It should be mentioned that we suppose 1280 byte long message to be transferred.

As for the proposed method, we introduced two kinds of implementations. One is a straightforward implementation, where the proposed method described in the previous section is programmed in C language as they are. The following are the summary of the straightforward implementation.

- The send/receive scramble buffers are realized by an array of unsigned char type.
- When a byte is scrambled or descrambled, the exclusive OR of all bytes in the scramble buffer is calculated.
- When a byte is scrambled or descrambled, it is added to the scramble buffer by shifting all bytes in the buffer.

The other is a revised implementation, where unnecessary data copying nor exclusive OR calculation are avoided. The following are the summary of the revised implementation.

- The send/receive scramble buffers are realized as ring buffers, by an array of unsigned char type (`sScrBuf[]` and `rScrBuf[]`). In order to avoid unnecessary data copying, the last element (newest element) in the ring buffer is maintained by an index parameter (`sIndex` or `rIndex`).
- The exclusive OR calculation for all bytes in the scramble buffer is performed just once in the beginning.

This result is maintained by a static variable `sXor` or `rXor`.

- When a byte is to be scrambled or descrambled, the static variable (`sXor` or `rXor`) is overwritten by the exclusive OR of the oldest element in the scramble buffer, `sXor` (or `rXor`) and the new byte.
- When a byte is to be scrambled or descrambled, it is added to the scramble buffer just by moving the index parameter (`sIndex` or `rIndex`).

By use of these two implementations, we executed the data scrambling and descrambling for a message with length of 1280 bytes.

We evaluated the performance of those seven methods (DES, TDEA, AES-128, AES-192, AES-256, the proposed method by straightforward implementation, and the proposed method by revised implementation). Table I shows the specification of personal computer used for the evaluation. It is a laptop computer manufactured by Lenovo over which the Linux operating system is installed. We measured the processing time of the encryption and decryption, or the scrambling and descrambling for a message with 1280 byte length. We used Linux `time` command for 10,000 iterations, and calculated the processing time for one operation.

Table II gives the performance results. The encryption and decryption of the DES and AES-128 algorithms require around 2.2 or 2.3 msec. The AES-192 and AES-256 algorithms requires a little more time. The TDEA algorithm requires around 6.7 msec, which is about three time of the DES algorithm. We need to say that we also evaluated the performance of the DES and AES with cipher block chaining (CBC) mode, and obtained the result that the processing time is almost the same with ECB mode.

On the other hand, the straightforward implementation of the proposed method requires around 1 msec. This is smaller than the cryptographic approaches, but the improvement is not large. However, the revised implementation of the proposed method decreases the processing time largely, to around 0.04 msec. It is less than 1/60 compared with the DES and AES algorithms. Although the implementation of DES and AES algorithms is a publicly accessible software, which may be optimized adequately, the obtained results are considered to show that the proposed method is able to decrease the processing overhead of ciphering operations and to provide some level of security against the eavesdropping over untrusted paths in MPTCP communications.

TABLE I. SPECIFICATION OF PC USED IN EVALUATION.

model	lenovo ThinkPad E430
CPU	Intel Core i5-3230M CPU × 4
clock	2.60GHz
memory size	3.7 Gbytes
kernel	ubuntu 16.04 LTS

TABLE II. PROCESSING TIME OF 1280 BYTE MESSAGE.

DES	TDEA	AES-128	AES-192	AES-256	Proposed (straight)	Proposed (revised)
2.24 msec	6.69 msec	2.29 msec	2.80 msec	3.40 msec	0.950 msec	0.0352 msec

V. STUDY ON IMPLEMENTATION

A. How to modify Linux operating system

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 hard task, and so we decided to use a debugging mechanism for the Linux kernel, called kernel probes [24].

The following are cited from [24]. *A kernel probe is a set of handlers placed on a certain instruction address. There are two types of probes in the kernel as of now, called "KProbes" and "JProbes." A KProbe is defined by a pre-handler and a post-handler. When a KProbe is installed at a particular instruction and that instruction is executed, the pre-handler is executed just before the execution of the probed instruction. Similarly, the post-handler is executed just after the execution of the probed instruction. JProbes are used to get access to a kernel function's arguments at runtime. A JProbe is defined by 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, followed by the transfer of execution to the original function.*

Figure 11 shows a schematic explanation of JProbe. We assume that there is function `a_func()` inside the Linux kernel, whose symbol is exported. A user may define JProbe handler `ja_func()` whose arguments are exactly the same as `a_func()`. When the Linux kernel is going to call `a_func()`, `ja_func()` is executed in the beginning of `a_func()`. When `ja_func()` returns, the kernel executes `a_func()`. 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()` [25].

A famous example of JProbe is tcpprobe [26] used to collect TCP sender internal information such as a TCP congestion window value.

B. Design principles

We adopted the following design principles to implement the proposed method inside the Linux kernel.

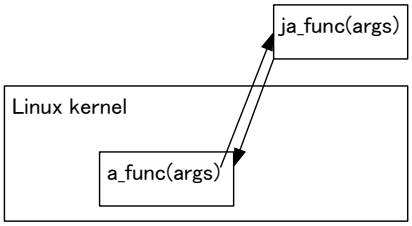


Figure 11. Schematic explanation of JProbe.

- *Use the JProbe mechanism as much as possible.*
  In the Linux kernel, function `tcp_sendmsg()` is called when a user process tries to send data to MPTPCP (actually TCP, too) [27]. So, we define a JProbe handler for this function in order to scramble data to be transferred. On the other hand, function `tcp_recvmsg()` is called when a user process is going to receive data from MPTCP. In this case, however, the descrambling procedure needs to be done in the end of this function. So, we introduce a dummy kernel function and export its symbol. We then introduce a JProbe handler for descrambling. By adopting this approach, we can program and debug scrambling/descrambling independently of the Linux kernel itself.
- *Maintain control variables within socket data structure.*
  In order to perform the scrambling/descrambling, the control variables described in the previous section, such as `sScrBuf[64]` and `sXor`, 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 [27]. So, we add the control variables for data scrambling to this data structure. Although the kernel modification and rebuild are required, we believe that to insert some variables is an easy task and therefore frequent debugging and rebuilding are not necessary.

C. Detailed design

(1) How to insert control variables in socket data structure

As described above, function `tcp_sendmsg()` is called at data sending. Its prototype in Ubuntu 16.04 LTS is;

```
tcp_sendmsg(struct sock *sk, struct
            msghdr *msg, size_t size).
```

Here, `struct sock` is a type of socket data structure. In the beginning of this function, `sk` is converted to type `struct tcp_sock` in the following way.

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

`struct tcp_sock` is a type of TCP socket data structure maintaining TCP related members like `rcv_nxt`, which is sequence number of a byte to be expected to receive next, and `snd_nxt`, which is sequence number of a byte to be sent next. It also includes a control information on MPTCP like

```
struct mptcp_cb *mpcb;.
```

Structure `struct mptcp_cb` includes MPTCP related information including keys and tokens like `__u64 mptcp_loc_key;` (local key) and `__u32 mptcp_loc_token;` (local token). Based on these considerations, we decided to insert control variables for data scrambling within structure `struct mptcp_cb` in a way shown in Figure 12. When `tcp_sendmsg()` is called, we can access to these variables in a way like `tp->mpcb->sXor`. It should be noted that `sFirst` and `rFirst` indicate whether the scrambling and descrambling is performed at first or not in this MPTCP connection, respectively.

```
struct mptcp_cb {
 . . . .
  unsigned char sScrBuf[64], rScrBuf[64];
  unsigned char sXor, rXor;
  int sIndex, rIndex, sFirst=1, rFirst=1;
};
```

Figure 12. Control variables for data scrambling.

(2) How to implement scrambling

(2-1) Overview

Figure 13 shows an overview program structure to implement scrambling using JProbe handler `jtcp_sendmsg()`. As described in the previous subsection, `jtcp_sendmsg()` is declared so as to have the same arguments as `tcp_sendmsg()`, as shown in part (i) in the figure. Part (ii) in the figure shows a data structure registering an entry point of the JProbe hander and its related symbol name. Parts (iii) and (iv) are the initialization and exit functions, respectively.

We need to explain about the second argument of `jtcp_sendmsg()`. Structure `struct msghdr` has a linked data structure maintaining one or more members, each of which is expressed by structure `struct iovec`, including pointer to data (`iov_vase`) and its length (`iov_length`). `iov_for_each()` is a macro for traversing individual members, and can be used as a for statement in C language.

```
int jtcp_sendmsg(struct sock *sk,
        struct msghdr *msg, size_t size) {
  struct tcp_sock *tp = tcp_sk(sk);
  struct iov_iter iter;
  struct iovec iov;

  if(tp->mpcb->sFirst) scramble_init(tp);
  iov_for_each(iov, iter, msg->msg_iter) {
    scramble(tp, iov.iov_base, iov.iov_len);
  }
  jprobe_return();
  return 0;
} // (i) JProbe handler

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

static __init int jtcp_sendmsg_init(void) {
  ret= register_jprobe(&tcp_sendmsg_jprobe);
  if (ret < 0) return -1;
  return 0;
}  // (iii) Init function
module_init(jtcp_sendmsg_init);

static __exit void jtcp_sendmsg_exit(void) {
  unregister_jprobe(&tcp_sendmsg_jprobe);
}  // (iv) Exit function
module_exit(jtcp_sendmsg_exit);
```

Figure 13. Overview on how to implement scrambling in `tcp_sendmsg()`.

In function jtcp_sendmsg(), control variable tp->mpcb->sFirst is checked in the beginning and, if it is 1, function scramble_init() is called. After that, function scramble() is called for each data contained in msg.

(2-2) Detailed design for scrambling

Figure 14 shows an example of program code for functions scramble_init() and scramble(). scramble_init() is called with an argument tp, which is a pointer to struct tcp_sock data structure. By functions memset() and memcpy(), the send scramble buffer tp->mpcb->sScrBuf[64] is initialized so as to contain the local Key in this MPTCP connection. Then, the XOR result for all bytes in the send scramble buffer is stored in control variable tp->mpcb->sXor. After that, the index parameter indicating the end of the send scramble buffer is settled and tp->mpcb->sFirst is reset.

Function scramble() performs the scrambling procedure for data pointed by argument data whose length is len. In this function, each byte in data is XORed with tp->mpcb->sXor, and index parameter tp->mpcb->sIndex is shifted by one. Then, tp->mpcb->sXor is updated by XORing itself, a byte that is stored in the newly indexed position, and a byte being scrambled (original byte). After that, the original byte is stored in a newly indexed position in the send scramble buffer. In the end, data is changed by the XORed byte for sending.

(3) How to implement descrambling

As mentioned in the previous subsection, the descramble procedure needs to be implemented at the end of function tcp_recvmsg(). In order to realize the descrambling procedure by the JProbe mechanism, we introduced dummy function dummy_recvmsg() just before returning from tcp_recvmsg(). This is shown in Figure 15. For this function, JProbe handler jdummy_recvmsg() is implemented in a similar way with the scrambling procedure.

```
void scramble_init(struct tcp_sock *tp) {
  int i;
  unsigned char x;

  memset(tp->mpcb->sScrBuf, 0, 64);
  memcpy(&tp->mpcb->sScrBuf[56],
         &tp->mpcb.mptcp_loc_key, 8);
  for(i=0,x=0;i<64;i++)
    x = x ^ tp->mpcb->sScrBuf[i];
  tp->mpcb->sXor = x;
  tp->mpcb->sIndex = 63;
  tp->mpcb->sFirst = 0;
  return;
}

void scramble(struct tcp_sock *tp,
              unsigned char *data, size_t len) {
  int i;
  unsigned char x;

  for(i=0;i<len;i++) {
    x = data[i] ^ tp->mpcb->sXor;
    tp->mpcb->sIndex = (tp->mpcb->sIndex+1)%64;
    tp->mpcb->sXor = tp->mpcb->sXor
         ^ tp->mpcb->sScrBuf[tp->mpcb->sIndex]
         ^ data[i];
    tp->mpcb->sScrBuf[tp->mpcb->sIndex] = data[i];
    data[i] = x;
  }
  return;
}
```

Figure 14. Program code for scrambling.

```
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);
  . . . .
  release_sock(sk);
  dummy_recvmsg(sk, msg, len, nonblack, flags, addr_len);
  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 15. JProbe handler for data descrambling.

VI. CONCLUSIONS

This paper proposes a new method to improve privacy against eavesdropping over MPTCP communications, which has become popular among recent mobile terminals. Recent mobile terminals have multiple communication interfaces, some of which are connected to trusted network operators (e.g., LTE interfaces), and some of which may be connected to untrusted network, such as public WLAN hot spots. The proposed method here is based on the not-every-not-any protection principle, where, *if an attacker cannot observe the data on every path, he cannot observe the traffic on any path*. We designed a detailed procedure by following the byte oriented data scrambling in order to avoid unnecessary data length expansion.

We evaluated the processing overhead of the DES, TDEA and AES encryption/decryption and that of data scrambling in the proposed method. The result showed that the optimized implementation of our method requires only less than 1/60 processing time compared with the cryptographic approaches. Although the proposed method is a practical solution, as described above, the processing capability of mobile terminals is still low, and so our proposal is considered to be useful to increase the security against eavesdropping over untrusted mobile communication networks.

Moreover, we discussed how to implement the proposed method in the Linux operating system. We explained about a kernel debugging mechanism called JProbes, which is used to implement tcpprobe. We showed how to make program codes, especially focusing on how to realize the control parameters in the socket data structure and on how to realize the scrambling and descrambling procedures in the JProbe handlers.

We are currently implementing the proposed method on top of MPTCP software in the Linux operating system. We will continue this implementation and conduct the performance evaluation over real networks. Moreover, the proposed method can only prevent eavesdropping, and cannot ensure the integrity of transferred data. We need to improve our method in this aspect.

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, "Protecting Eavesdropping over Multipath TCP Communication Based on Not-Every-Not-Any Protection," in Proc. SECURWARE 2017, pp. 82-87, Sep. 2017.

[2] NGNM Alliance, "5G White Paper," https://www.ngmn.org/5g-white-paper/5g-white-paper.html, [retrieved: May 2018].

[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: May 2018].

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

[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] 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.

[10] T. Kato, M. Tenjin, R. Yamamoto, S. Ohzahata, and H. Shinbo, "Microscopic Approach for Experimental Analysis of Multipath TCP Throughput under Insufficient Send/Receive Socket Buffers," in Proc. 15th ICWI 2016, pp. 191-199, Oct. 2016.

[11] J. Ma, F. Le, A. Russo, and J. Lobo, "Detecting Distributed Signature-based Intrusion: The Case of Multi-Path Routing Attacks," in Proc. 2015 INFOCOM, pp. 558-566, Apr. 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] X. Zhao, L. Li, G. Xue, and G. Silva, "Efficient Anonymous Message Submission," in Proc. INFOCOM 2012, pp. 2228-2236, Mar. 2012.

[18] 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.

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

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

[21] Federal Information Processing Standards Publication 46-3, "Annoucing the Data Encryption Standard," Oct. 1999.

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

[23] PJC, "Distribution of Sample Program / Source / Software (in Japanese)," http://free.pjc.co.jp/index.html, [retrieved: May 2018].

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

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

[26] Linux Foundation Wiki, "TCP Probe," https://wiki.linuxfoundation.org/networking/tcpprobe, [retrieved: May 2018].

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