PAPER Special Section on Formal Approaches

Verification of LINE Encryption Version 1.0 Using ProVerif*

Cheng SHI[†], *Nonmember and* **Kazuki YONEYAMA**[†a)], *Member*

SUMMARY LINE is currently the most popular messaging service in Japan. Communications using LINE are protected by the original encryption scheme, called LINE Encryption, and specifications of the client-to-server transport encryption protocol and the client-to-client message end-to-end encryption protocol are published by the Technical Whitepaper. Though a spoofing attack (i.e., a malicious client makes another client misunderstand the identity of the peer) and a reply attack (i.e., a message in a session is sent again in another session by a man-in-the-middle adversary, and the receiver accepts these messages) to the end-to-end protocol have been shown, no formal security analysis of these protocols is known. In this paper, we show a formal verification result of secrecy of application data and authenticity for protocols of LINE Encryption (Version 1.0) by using the automated security verification tool ProVerif. Especially, since it is claimed that the transport protocol satisfies forward secrecy (i.e., even if the static private key is leaked, security of application data is guaranteed), we verify forward secrecy for client's data and for server's data of the transport protocol, and we find an attack to break secrecy of client's application data. Moreover, we find the spoofing attack and the reply attack, which are reported in previous papers.

key words: *formal methods, end-to-end encryption, LINE Encryption, ProVerif*

1. Introduction

1.1 Background

With the development of network communications technology, more and more people use messaging services to communicate. LINE is currently the most popular messaging service in Japan. Thus, if there is security vulnerability in LINE, widespread incidents may be caused due to its popularity. Hence, it is required that the security of LINE is rigorously analyzed.

In order to ensure the security of LINE, communications in LINE are protected by a dedicated encrypted communication scheme, called LINE Encryption, and specifications of the client-to-server transport encryption protocol (TEP) and the client-to-client message end-to-end encryption protocol (E2EEP) are published by the Technical Whitepaper [2]. In [2], some informal security analyses of the TEP and the E2EEP are shown, and it is claimed that the TEP satisfies forward secrecy [3] such that "In the event that a private key is leaked, messages that were encrypted before the leak are protected if the communication supports forward secrecy" [4]. Since there are two kinds of messages (client's application data encrypted by temporary key $\mathbf{key}_{\mathrm{temp}}$ and server's application data encrypted by forward secure key $\mathbf{key}_{\mathrm{FS}}$) in the TEP, we need to consider two kinds of forward secrecy (i.e., forward secrecy for client's data and forward secrecy for server's data).

On the other hand, Espinoza et al. [5] showed a reply attack against the E2EEP. A man-in-the-middle (MTM) adversary can send an encrypted message in an old session as the message in the new session without changing the content of the message. Though the adversary cannot know the content of the message, the receiver client accepts these two messages as valid. Isobe and Minematsu [6] showed a spoofing attack against the E2EEP. A malicious client C_3 intercepts the E2EEP between clients C_1 and C_2, and impersonates C_1 to C_2. Hence, since another unknown attack may exist, security of LINE Encryption is still unclear.

On the other hand, since it is difficult to analyze all attacks by hands, such as the replay attack or the spoofing attack, automated security verification methods by using formal methods have been studied to formally verify the security of cryptographic protocols.

1.2 Contribution

In this paper, we give the first formal verification result of the security of LINE Encryption (Version 1.0) by using the automated security verification tool ProVerif [7]. Specifically, for the TEP, we verify forward secrecy of both client's application data and server's application data, and server authenticity. For the E2EEP, we verify secrecy of application data and authenticity. We obtain the following verification results:

For the TEP: We find an attack to break forward secrecy for client's data, but we cannot find attacks for forward secrecy for server's data and server authenticity.

For the E2EEP: We find the spoofing attack and the replay attack, but we cannot find attacks for secrecy of application data.

Thus, our verification result captures all known attacks to LINE Encryption, and points out the attack to forward secrecy for client's data, which is not formally reported. Therefore, our result clarifies that the automated verification tool is useful to verify the security of messaging protocols.

Manuscript received September 18, 2018.
Manuscript revised February 8, 2019.
Manuscript publicized April 24, 2019.

[†]The authors are with Ibaraki University, Hitachi-shi, 316–8511 Japan.

*This paper is the full version of the extended abstract appeared in [1].

a) E-mail: kazuki.yoneyama.sec@vc.ibaraki.ac.jp

DOI: 10.1587/transinf.2018FOP0001

1.3 Organization of Paper

In Sect. 2, we recall protocols of the TEP and the E2EEP, and roughly explain ProVerif. In Sect. 3, we give our formalization of the TEP and the E2EEP by using the ProVerif language. In Sect. 4, we show LINE's claimed security for the TEP and the E2EEP. In Sect. 5, we give our verification result and found attacks.

2. Preliminaries

2.1 Client-to-Server Transport Encryption Protocol (TEP)

The client and server exchange the following messages in order to establish the transport key used to protect application data. A message sequence chart is given in Fig. 1.

2.1.1 Static Keys

In order to guarantee that clients only connect to legitimate the LINE servers, the TEP uses static ECDH and ECDSA [8] key pairs. The LINE servers securely store the private part of each pair, while the corresponding public keys are embedded in LINE client applications.

- ECDH key pair for key exchange: ($\mathbf{static}_{\mathrm{private}}$, $\mathbf{static}_{\mathrm{public}}$)
- ECDSA key pair for server identity verification: ($\mathbf{sign}_{\mathrm{private}}$,$\mathbf{sign}_{\mathrm{public}}$)

2.1.2 Client Hello

The client generates a temporary transport key and initialization vector by the static ECDH key exchange, and encrypts application data.

1. Generate an initial ephemeral ECDH key ($\mathbf{c\_init}_{\mathrm{private}}$, $\mathbf{c\_init}_{\mathrm{public}}$) and a client nonce $\mathbf{c}_{\mathrm{nonce}}$.
2. Derive a temporary transport key and initialization vector (IV) using the server's static key and the initial ephemeral key generated in Step 1 (where HKDF is the HMAC-based extract-and-expand key derivation function [9]) as follows.

   $\mathbf{len}_{\mathrm{key}}$=16
   $\mathbf{len}_{\mathrm{iv}}$=16
   $\mathbf{share}_{\mathrm{temp}}$=ECDH($\mathbf{c\_init}_{\mathrm{private}}$,$\mathbf{static}_{\mathrm{public}}$)
   $\mathbf{MS}_{\mathrm{temp}}$=HKDF$_{\mathrm{ex}}$($\mathbf{c_init}_{\mathrm{public}}\|\mathbf{c}_{\mathrm{nonce}}$,$\mathbf{share}_{\mathrm{temp}}$)[†]
   $\mathbf{keyiv}_{\mathrm{temp}}$=HKDF$_{\mathrm{exp}}$($\mathbf{MS}_{\mathrm{temp}}$,"legy temp key", $\mathbf{len}_{\mathrm{key}}+\mathbf{len}_{\mathrm{iv}}$)
   $\mathbf{key}_{\mathrm{temp}}$=$\mathbf{keyiv}_{\mathrm{temp}}$[0:15]
   $\mathbf{iv}_{\mathrm{temp}}$=$\mathbf{keyiv}_{\mathrm{temp}}$[16:31]
   ($\mathbf{keyiv}_{\mathrm{temp}}$[0:15] and $\mathbf{keyiv}_{\mathrm{temp}}$[16:31] mean the high-order 16 bytes and the low-order 16 bytes of $\mathbf{keyiv}_{\mathrm{temp}}$, respectively.)

3. Generate an ephemeral ECDH client handshake key ($\mathbf{c}_{\mathrm{private}}$,$\mathbf{c}_{\mathrm{public}}$).
4. $\mathbf{c}_{\mathrm{public}}$ and application data $\mathbf{appdata}_{\mathrm{client}}$ are encrypted with $\mathbf{key}_{\mathrm{temp}}$ and the client nonce $\mathbf{c}_{\mathrm{nonce}}$ using the AES-GCM [10] AEAD cipher as the ciphertext $\mathbf{data}_{\mathrm{enc}}$. The nonce is calculated by combining a client/server marker **marker**, a sequence number $\mathbf{num}_{\mathrm{seq}}$, and $\mathbf{iv}_{\mathrm{temp}}$ obtained in the handshake process.
5. Send the version $\mathbf{static}_{\mathrm{keyversion}}$, client's initial ephemeral key $\mathbf{c\_init}_{\mathrm{public}}$[††], client nonce $\mathbf{c}_{\mathrm{nonce}}$ and encrypted data $\mathbf{data}_{\mathrm{enc}}$ to the server.

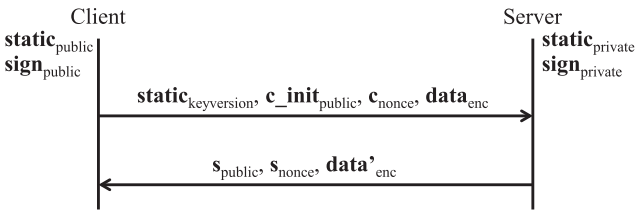

Fig. 1 Message sequence chart for TEP

2.1.3 Server Hello

The server generates the temporary transport key and initialization vector by the static ECDH key exchange, and decrypts the ciphertext from the client. Also, the server generates a forward-secure transport key and initialization vector by the ephemeral ECDH key exchange, and encrypts application data.

1. Calculate the temporary transport key $\mathbf{key}_{\mathrm{temp}}$ and IV $\mathbf{iv}_{\mathrm{temp}}$ using the server's static ECDH key $\mathbf{static}_{\mathrm{private}}$ and the client's initial ephemeral key $\mathbf{c\_init}_{\mathrm{public}}$ as follows.

   $\mathbf{share}_{\mathrm{temp}}$=ECDH($\mathbf{static}_{\mathrm{private}}$,$\mathbf{c\_init}_{\mathrm{public}}$)
   $\mathbf{MS}_{\mathrm{temp}}$=HKDF$_{\mathrm{ex}}$($\mathbf{c_init}_{\mathrm{public}}\|\mathbf{c}_{\mathrm{nonce}}$,$\mathbf{share}_{\mathrm{temp}}$)
   $\mathbf{keyiv}_{\mathrm{temp}}$=HKDF$_{\mathrm{exp}}$($\mathbf{MS}_{\mathrm{temp}}$,"legy temp key", $\mathbf{len}_{\mathrm{key}}+\mathbf{len}_{\mathrm{iv}}$)
   $\mathbf{key}_{\mathrm{temp}}$=$\mathbf{keyiv}_{\mathrm{temp}}$[0:15]
   $\mathbf{iv}_{\mathrm{temp}}$=$\mathbf{keyiv}_{\mathrm{temp}}$[16:31]

2. Decrypt $\mathbf{data}_{\mathrm{enc}}$ by using $\mathbf{key}_{\mathrm{temp}}$ and $\mathbf{iv}_{\mathrm{temp}}$, and obtain application data $\mathbf{appdata}_{\mathrm{client}}$ and $\mathbf{c}_{\mathrm{public}}$.
3. Generate an ephemeral key pair ($\mathbf{s}_{\mathrm{private}}$,$\mathbf{s}_{\mathrm{public}}$) and a server nonce $\mathbf{s}_{\mathrm{nonce}}$.
4. Derive the forward-secure (FS) transport key $\mathbf{key}_{\mathrm{FS}}$ and IV $\mathbf{iv}_{\mathrm{FS}}$ as follows.

   $\mathbf{len}_{\mathrm{key}}$=16
   $\mathbf{len}_{\mathrm{iv}}$=16
   $\mathbf{share}_{\mathrm{FS}}$=ECDH($\mathbf{s}_{\mathrm{private}}$,$\mathbf{c}_{\mathrm{public}}$)

[†]In [2], it is described as $\mathbf{MS}_{\mathrm{temp}}$ = HKDF$_{\mathrm{ex}}$($\mathbf{c}_{\mathrm{public}}\|\mathbf{c}_{\mathrm{nonce}}$, $\mathbf{share}_{\mathrm{temp}}$). However, it is a typo. The authors confirmed the typo to the LINE Security Team.

[††]In [2], it is described as to send $\mathbf{c}_{\mathrm{public}}$. However, it is a typo. The authors confirmed the typo to the LINE Security Team.

$\mathbf{MS}_{\mathrm{FS}}$=HKDF$_{\mathrm{ex}}$($\mathbf{c}_{\mathrm{nonce}}$||$\mathbf{s}_{\mathrm{nonce}}$,$\mathbf{share}_{\mathrm{FS}}$)
$\mathbf{keyiv}_{\mathrm{FS}}$=HKDF$_{\mathrm{exp}}$($\mathbf{MS}_{\mathrm{FS}}$,"legy temp key",
$\mathbf{len}_{\mathrm{key}}$+$\mathbf{len}_{\mathrm{iv}}$)
$\mathbf{key}_{\mathrm{FS}}$=$\mathbf{keyiv}_{\mathrm{FS}}$[0:15]
$\mathbf{iv}_{\mathrm{FS}}$=$\mathbf{keyiv}_{\mathrm{FS}}$[16:31]

5. Generate and sign the handshake state using the server's static signing key as follows.

   $\mathbf{state}$=SHA256($\mathbf{c}_{\mathrm{public}}$||$\mathbf{c}_{\mathrm{nonce}}$||$\mathbf{s}_{\mathrm{public}}$||$\mathbf{s}_{\mathrm{nonce}}$)
   $\mathbf{state}_{\mathrm{sign}}$=ECDSA$_{\mathrm{sign}}$($\mathbf{state}$,$\mathbf{sign}_{\mathrm{private}}$)

6. Application data $\mathbf{appdata}_{server}$ is encrypted with $\mathbf{key}_{\mathrm{FS}}$ and the nonce $\mathbf{s}_{\mathrm{nonce}}$ using the AES-GCM AEAD cipher as the ciphertext $\mathbf{data}'_{\mathrm{enc}}$. The nonce is calculated by combining a client/server marker **marker**, a sequence number $\mathbf{num}_{\mathrm{seq}}$, and the $\mathbf{iv}_{\mathrm{FS}}$ obtained in the handshake process.
7. Send the ephemeral key $\mathbf{s}_{\mathrm{public}}$, server nonce $\mathbf{s}_{\mathrm{nonce}}$ and encrypted data $\mathbf{data}'_{\mathrm{enc}}$ to the client.

2.1.4 Client Finish

The client generates the forward-secure transport key and initialization vector by the ephemeral ECDH key exchange.

1. Verify the handshake signature $\mathbf{state}_{\mathrm{sign}}$. If it is valid, proceed to the next step. If not, abort the connection.
2. Derive $\mathbf{key}_{\mathrm{FS}}$ and $\mathbf{iv}_{\mathrm{FS}}$ as follows.

   $\mathbf{share}_{\mathrm{FS}}$=ECDH($\mathbf{c}_{\mathrm{private}}$,$\mathbf{s}_{\mathrm{public}}$)
   $\mathbf{MS}_{\mathrm{FS}}$=HKDF$_{\mathrm{ex}}$($\mathbf{c}_{\mathrm{nonce}}$||$\mathbf{s}_{\mathrm{nonce}}$,$\mathbf{share}_{\mathrm{FS}}$)
   $\mathbf{keyiv}_{\mathrm{FS}}$=HKDF$_{\mathrm{exp}}$($\mathbf{MS}_{\mathrm{FS}}$,"legy temp key",
   $\mathbf{len}_{\mathrm{key}}$+$\mathbf{len}_{\mathrm{iv}}$)
   $\mathbf{key}_{\mathrm{FS}}$=$\mathbf{keyiv}_{\mathrm{FS}}$[0:15]
   $\mathbf{iv}_{\mathrm{FS}}$=$\mathbf{keyiv}_{\mathrm{FS}}$[16:31]

2.2 Message End-to-End Encryption (E2EEP)

Two clients exchange the following messages in order to send a message without revealing it to others. A message sequence chart is given in Fig. 2.

2.2.1 Client-to-Client Key Exchange

In order to be able to exchange encrypted messages, clients must share a common cryptographic secret. When a LINE client wishes to send a message, it first retrieves the current public key of the recipient. Next, the client passes its own private key and the recipient's public key to the ECDH algorithm in order to generate a shared secret as follows.

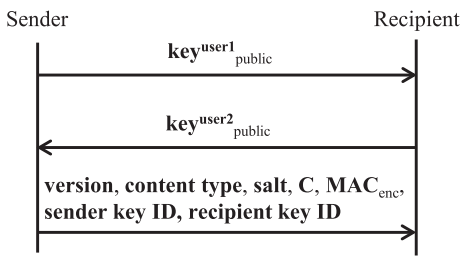

Fig. 2 Message sequence chart for E2EEP

Shared Secret
=ECDH$_{\mathrm{curve25519}}$($\mathbf{key}^{\mathbf{user1}}{}_{\mathrm{private}}$,$\mathbf{key}^{\mathbf{user2}}{}_{\mathrm{public}}$)
=ECDH$_{\mathrm{curve25519}}$($\mathbf{key}^{\mathbf{user2}}{}_{\mathrm{private}}$,$\mathbf{key}^{\mathbf{user1}}{}_{\mathrm{public}}$)

2.2.2 Message Encryption

The sender client encrypts a message with a unique encryption key and IV, and sends the encrypted message to the recipient client.

1. The sender derive the encryption key $\mathbf{Key}_{\mathrm{encrypt}}$ and IV $\mathbf{IV}_{\mathrm{encrypt}}$ from the shared secret calculated in the above process, and a randomly generate **salt** as follows.

   $\mathbf{Key}_{\mathrm{encrypt}}$=SHA256(**Shared Secret**||**salt**||"Key")
   $\mathbf{IV}_{\mathrm{pre}}$=SHA256(**Shared Secret**||**salt**||"IV")
   $\mathbf{IV}_{\mathrm{encrypt}}$=$\mathbf{IV}_{\mathrm{pre}}$[0:15] $\oplus$ $\mathbf{IV}_{\mathrm{pre}}$[16:31]

2. The generated key and IV are used to encrypt the message payload **M** using AES in CBC block mode.
3. The sender calculates a message authentication code (MAC) of the ciphertext **C** (where AESECB is the AES in ECB mode) as follows.

   $\mathbf{MAC}_{\mathrm{plain}}$=SHA256(**C**)
   $\mathbf{MAC}_{\mathrm{enc}}$=AESECB($\mathbf{Key}_{\mathrm{encrypt}}$,$\mathbf{MAC}_{\mathrm{plain}}$[0:15]
   $\oplus\mathbf{MAC}_{\mathrm{plain}}$[16:31])

4. **version**, **content type**, **salt**, **C**, $\mathbf{MAC}_{\mathrm{enc}}$, **sender key ID** and **recipient key ID** are included in the message sent to the recipient.
5. The recipient derives the symmetric encryption key $\mathbf{Key}_{\mathrm{encrypt}}$, and IV $\mathbf{IV}_{\mathrm{encrypt}}$ as described above.
6. The recipient calculates the MAC $\mathbf{MAC}'_{\mathrm{enc}}$ of the received cipher text, and compares it with the MAC $\mathbf{MAC}_{\mathrm{enc}}$ value included in the message. If they match, the contents of the message are decrypted and displayed. Otherwise, the message will be discarded.

2.3 ProVerif

ProVerif is a model checking tool that performs automated security verification. To verify a security requirement of a protocol by ProVerif, we must formalize the cyptographic primitives, the protocol specification and the security requirement as input to ProVerif. Here, we briefly explain how to formalize these by using an example of a symmetric key encryption. For the detail of ProVerif, please see [7].

1. Define communication paths and cryptographic primitives
   - Type designates types representing keys, random

numbers, etc.

```
type key (*types of private keys
    in symmetric key encryption*)
type coins (*type of random
    numbers*)
```

- Free name defines channel name.

```
free c:channel (*Public
    communication channel*)
free c:channel [private] (*Secret
     communication channel*)
```

- Constructors defines cryptographic primitives such as encryption functions, etc.

```
fun senc(bitstring,key):bitstring
     (*function of symmetric key
    encryption*)
```

- Destructor specifies conditions of functions.

```
reduc forall m:bitstring,k:key;
    sdec(senc(m,k),k)=m. (*
    decryption condition of
    symmetric key encryption*)
```

2. Define participant behaviors within the cryptographic protocol

```
out(c,A) (*send the message A to
    channel c*)
in(c,B) (*receive the message B
    from channel c*)
```

3. Define public information, confidential information held in advance by each participant, and give it as input to participants.

```
new r:coins (*generate a random number*)
((!clientA)|(!serverB)) (*parallel execution
    of client and server*)
```

A variety of properties can be analyzed by ProVerif, such as the correspondence assertions (i.e., whether event B occurred before the event A occurred), the reachability (i.e., whether a specific event occurred), and the observation equivalence (i.e., whether is able to analyze two processes that perform different computations but have the same run result).

3. Formalization of LINE Encryption

In this section, we give our formalization of LINE Encryption in ProVerif.

3.1 Formalization of TEP

3.1.1 Rules for Signature

Here, we define types of signature key and verification key required for digital signature, furthermore, the function `spk` that creates a verification key from the signing key. When a plaintext (bitstring) and a signing key are inputted, the function `sign` generates a signature. When a signature and a verification key are inputted, if the verification result is correct, define the `checksign` outputs true.

```
type signpublickey. (*verification key *)
type signprivatekey. (*signing key *)
fun spk(signprivatekey):signpublickey. (*
    generate a verification key *)
fun sign(bitstring,signprivatekey):bitstring.
     (*generate a signature *)
reduc forall m:bitstring,sprikey:
    signprivatekey;
checksign(sign(m,sprikey),spk(sprikey),m)=
    true. (*verify signature *)
```

3.1.2 Rules for ECDH Key Exchange

Here, we define types of the generator, exponents and the base point, and functions `Ggen` to convert the base point to the generator and `sca` to compute the scalar multiplication, and commutativity by `equation`.

```
type G. (*type of generator *)
type scalar. (*type of scalar *)
type basis. (*type of base points *)
fun Ggen(basis):G. (*convert base point to
    generator *)
fun sca(scalar,G):G. (*scalar multiplication
    *)
equation forall a:scalar, b:scalar, P:basis;
sca(a,sca(b,Ggen(P)))=sca(b,sca(a,Ggen(P))).
    (*commutativity*)
```

3.1.3 Ruless for XOR

Here, we define the function `xor` to compute the XOR of two inputs, and the property of XOR by four `equation`[†].

```
fun xor(bitstring,bitstring):bitstring.
equation forall x:bitstring, y:bitstring; xor
    (xor(x,y),y)=x.
equation forall x:bitstring, y:bitstring; xor
    (y,xor(x,x))=y.
equation forall x:bitstring, y:bitstring; xor
    (xor(x,y),xor(x,x))=xor(x,y).
equation forall x:bitstring, y:bitstring; xor
    (xor(x,y),xor(y,y))=xor(x,y).
```

3.1.4 Parameter

Here, we define key type, random number type, version type, and fixed values and word “legy temp key” as `const`.

```
type key.
type iv.
```

[†]As being described in [7], associativity cannot be handled by ProVerif, which prevents the modeling of primitives such as XOR, because associativity as ($xor(x, y) = xor(y, x)$) generates an infinite number of rewrite rules, so in this case ProVerif does not terminate. Thus, we formalize the XOR by limited rules as previous works.

```
type coins.
type version.
const legy:bitstring[data].
const len:bitstring[data].
const num:bitstring[data].
const marker:bitstring[data].
```

3.1.5 Rules for AEAD

Here, we define the function senc to encrypt a plaintext and the function sdec to decrypt a ciphertext, and the relationship between senc and sdec by reduc.

```
fun senc(key,coins,bitstring):bitstring. (*
    encryption function *)
reduc forall x:bitstring, k:key, r:coins;
sdec(k,r,senc(k,r,x))=x.(*decryption function
     *)
```

3.1.6 Declaration of Channel and Secret

Here, we define the channel and secret information by free.

```
free c:channel. (*the channel *)
free appdata1:bitstring[private]. (*the
    client's secret information *)
free appdata2:bitstring[private]. (*the
    server's secret information *)
```

3.1.7 Type-Converting Functions

Here, we define implicit functions to convert types for type adjustments of inputs of functions.

```
fun HKDFex(bitstring,G):bitstring. (*HKDFex*)
fun HKDFexp(bitstring,bitstring,bitstring):
    bitstring. (*HKDFexp*)
fun Ggen3(bitstring):key. (*convert bitstring
     type to key type *)
fun Ggen4(bitstring):iv. (*convert bitstring
    type to iv type *)
fun Hash(bitstring):bitstring. (*hash
    function *)
fun Ggen6(bitstring):coins. (*convert
    bitstring type to randam type *)
fun Ggen7(iv):bitstring. (*convert iv type to
     bitstring type *)
```

3.1.8 Verification of Forward Secrecy

We use reachability to verify forward secrecy. The attacker is passive, but he/she can obtain all static private keys of the server. Forward secrecy for client's (resp. server's) data require that the attacker cannot reach client's (resp. server's) application data. If the protocol has forward secrecy for client's (resp. server's) data, attacker(appdata1) (resp. attacker(appdata2)) will not happen. In other words, appdata1 (resp. appdata2) cannot be obtained by the passive attacker.

```
set attacker = passive.
query attacker(appdata1).
query attacker(appdata2).
```

3.1.9 Verification of Server Authenticity

Using the correspondence assertions, we can verify authenticity for clients. We define event Client1 corresponding to encrypting Client's secret message, and event Server1 corresponding to decrypting the secret message. If Server1 occurs, then Client1 must occur before Server1. It corresponds to client authenticity. Also, we define event Server2 corresponding to accepting digital signature verification, and event Client2 corresponding to completion of the session. If Client2 occurs, then Server2 must occur before Client2. It corresponds to server authenticity.

```
event Client1(key,coins).
event Server1(key,coins).
event Client2(key,iv).
event Server2(key,iv).
query x:key, n:coins;
event(Server1(x,n))==>event(Client1(x,n)).
query x:key, n:iv;
event(Client2(x,n))==>event(Server2(x,n)).
```

3.1.10 Client Subprocess

Here, we define client's actions.

```
let Client(ver:version,J:basis,stapu:G,spuk:
    signpublickey)=
new cinitpr:scalar;
new cnon:coins; (*Generate random number of
    client*)
let cinitpu=sca(cinitpr,Ggen(J))in
let sharedtemp=sca(cinitpr,stapu)in
let MStemp=HKDFex((cinitpu,cnon),sharedtemp)
    in (*MStemp*)
let keyivtemp=HKDFexp(MStemp,legy,len)in
let keytemp=Ggen3(breakf(keyivtemp))in (*
    keytemp*)
let ivtemp=Ggen4(breakb(keyivtemp))in (*
    ivtemp*)
let nonce=Ggen6(xor((marker,num),Ggen7(ivtemp
    )))in
new cpr:scalar;
let cpu=sca(cpr,Ggen(J))in
let dataenc=senc(keytemp,nonce,(cpu,appdata1)
    )in (*dataenc*)
event Client1(keytemp,nonce);
out(c,(ver,cinitpu,cnon,dataenc));
in(c,(spu':G,srand:coins,statesign':bitstring
    ,dataenc1':bitstring));
let statesign1=Hash((cpu,cnon,spu',srand))in
if checksign(statesign',spuk,statesign1)then
    (*Detect signature *)
let sharedFS'=sca(cpr,spu')in
let MSFS'=HKDFex((cnon,srand),sharedFS')in
let keyivFS'=HKDFexp(MSFS',legy,len)in
let keyFS'=Ggen3(breakf(keyivFS'))in
let ivFS'=Ggen4(breakb(keyivFS'))in
event Client2(keyFS',ivFS').
```

3.1.11 Server Subprocess

Here, we define server's actions.

```
let Server(stapr:scalar,stapu:G,J:basis,sprk:
    signprivatekey)=
in(c,(sver:version,cinitpu':G,crand:coins,
    dataenc':bitstring));
let sharedtemp'=sca(stapr,cinitpu')in (*
    shardtemp*)
let MStemp'=HKDFex((cinitpu',crand),
    sharedtemp')in (*MStemp*)
let keyivtemp'=HKDFexp(MStemp',legy,len)in (*
    keyivtemp*)
let keytemp'=Ggen3(breakf(keyivtemp'))in (*
    keytemp*)
let ivtemp'=Ggen4(breakb(keyivtemp'))in (*
    ivtemp*)
let nonce1=Ggen6(xor((marker,num),Ggen7(
    ivtemp')))in
let (cpu':G,appdata1':bitstring)=sdec(keytemp
    ',nonce1,dataenc')in (*cpu&app data*)
new snon:coins; (*Generate random number of
    server*)
new spr:scalar;
let sharedFS=sca(spr,cpu')in
let MSFS=HKDFex((crand,snon),sharedFS)in (*
    MSFS*)
let keyivFS=HKDFexp(MSFS,legy,len)in
let keyFS=Ggen3(breakf(keyivFS))in (*keyFS*)
let ivFS=Ggen4(breakb(keyivFS))in (*ivFS*)
let spu=sca(spr,Ggen(J))in
let state=Hash((cpu',crand,spu,snon))in (*
    state*)
let statesign=sign(state,sprk)in (*signature
    *)
let nonce2=Ggen6(xor((marker,num),Ggen7(ivFS)
    ))in
let dataenc1=senc(keyFS,nonce2,appdata2)in (*
    dataenc*)
event Server1(keytemp',nonce1);
event Server2(keyFS,ivFS);
out(c,(spu,snon,statesign,dataenc1)).
```

3.1.12 Main Process for Verifying Forward Secrecy

Here, we define the version `ver`, base point `J`, the ECDH private key `stapr`, and the signing key `sprk`. The ECDH public key `stapu` and public key of signature `spuk` are exposed in channel `c`. The client subprocess and the server subprocess are executed in parallel in phase 0. In order to verify forward secrecy, the ECDH private key `stapr` and the signing key `sprk` will be exposed in phase 1.

```
process
new ver:version;
new J:basis;
new stapr:scalar;
new sprk:signprivatekey;
let stapu=sca(stapr,Ggen(J))in out(c,stapu);
let spuk=spk(sprk)in out(c,spuk);
(((!Client(ver,J,stapu,spuk))|(!Server(stapr,
    stapu,J,sprk)))|phase 1;out(c,(stapr,
    sprk)))
```

3.2 Formalization of E2EEP

3.2.1 Rules for ECDH Key Exchange

The process is described in Sect. 3.1, and we omit it.

3.2.2 Rules for AES-CBC

Here, we define the function `senccbc` to encrypt a plaintext and the function `sdeccbc` to decrypt a ciphertext, and the relationship between `senccbc` and `sdeccbc` by `reduc`.

```
type key.
type iv. (*type of keys *)
fun senccbc(bitstring,key,iv):bitstring. (*
    encryption function *)
reduc forall m:bitstring, k:key, i:iv ;
    sdeccbc(senccbc(m,k,i),k,i)=m. (*
    decryption function *)
```

3.2.3 Rules for AES-ECB

Here, we define the function `sencecb` to encrypt a plaintext and the function `sdececb` to decrypt a ciphertext, and the relationship between `sencecb` and `sdececb` by `reduc`.

```
fun sencecb(bitstring,key):bitstring. (*
    encryption function *)
reduc forall m:bitstring, k:key; sdececb(
    sencecb(m,k),k)=m. (*decryption function
     *)
```

3.2.4 Parameter

Here, we define random number type, version type and fixed words “Key” and “IV” by `const`.

```
type coins.
type version.
type ID.
const Key:bitstring[data].
const IV:bitstring[data].
```

3.2.5 Rules for XOR

The process is described in Sect. 3.1, and we omit it.

3.2.6 Declarationof Channel and Secret

Here, we define the channel and secret information.

```
free c:channel. (*channel *)
free M:bitstring[private]. (*client's secret
    information *)
```

3.2.7 Hash Function and Type-Converting Functions

Here, we define the hash function and implicit functions to convert types for type adjustments of inputs of functions.

```
fun Hash(bitstring):bitstring. (*hash
    function *)
fun Ggen1(G,coins,bitstring):bitstring. (*G,
    coins, bitstring to bitstring type *)
fun Ggen2(bitstring):key. (*convert bitstring
     type to key type *)
fun Ggen3(bitstring):iv. (*convert bitstring
    type to iv type *)
```

3.2.8 Verification of Secrecy and Authenticity for Replay Attack

We use the correspondence assertions, we can verify authenticity for clients. We define event `Client1` which Client1 encrypts Client1's secret message, and event `Client2` which Client2 decrypts Client 1's secret message. If `Client2` occurs, then `Client1` must occur only once before `Client2`. In order to verify replay attack, we use injective correspondence assertions `inj-event` to capture the one-to-one relationship. If it is a non-one-to-one relationship, it may happen that `Client2` is executed twice or more, but `Client1` is executed only once. In other words, it corresponds to a replay attack.

```
query attacker(M).
event Client1(key,iv).
event Client2(key,iv).
query x:key, i:iv;
inj-event(Client2(x,i))==>inj-event(Client1(x
    ,i)).
```

3.2.9 Client1 Subprocess for Replay Attack

Here, we define client1's actions.

```
let Client1(ver:version,c1cpr:scalar,c1cpu:G,
    c2cpu:G,J:basis,c1id:ID,c2id:ID)=
in(c,(c2cpu':G));
if c2cpu=c2cpu' then
new salt:coins;
let SharedSecret=sca(c1cpr,c2cpu')in
let Ken=Ggen2(Hash((SharedSecret,salt,Key)))
    in
let IVpre=Hash((SharedSecret,salt,IV))in
let IVen=Ggen3(xor(breakf(IVpre),breakb(IVpre
    )))in
event Client1(Ken,IVen);
let C=senccbc(M,Ken,IVen)in
let MACp=Hash(C)in
let MACe=sencecb(xor(breakf(MACp),breakb(MACp
    )),Ken)in
out(c,(ver,salt,C,MACe,c1id,c2id)).
```

3.2.10 Client2 Subprocess for Replay Attack

Here, we define client2's actions.

```
let Client2(ver:version,c2cpr:scalar,c2cpu:G,
    c1cpu:G,J:basis,c2id:ID)=
out(c,(c2cpu));
in(c,(ver':version,ran:coins,C':bitstring,mac
    :bitstring,id1:ID,id2:ID));
if c2id=id2 then
let SharedSecret'=sca(c2cpr,c1cpu)in
let Ken'=Ggen2(Hash((SharedSecret',ran,Key)))
    in
let MACp'=Hash(C')in
let MACe'=sencecb(xor(breakf(MACp'),breakb(
    MACp')),Ken')in
if mac=MACe' then
let IVpre'=Hash((SharedSecret',ran,IV))in
let IVen'=Ggen3(xor(breakf(IVpre'),breakb(
    IVpre')))in
let M'=sdeccbc(C',Ken',IVen')in
event Client2(Ken',IVen').
```

3.2.11 Main Process for Replay Attack

Here, we define the version `ver`, base point `J`, Client1's ECDH private key `c1cpr` and client2's ECDH private key `c2cpr`. The Client1's ECDH public key `c1cpu` and client2's ECDH public key c2cpu are exposed in channel `c`. Client1 subprocess and Client2 subprocess are executed in parallel.

```
process
new ver:version;
new J:basis;
new c1cpr:scalar;
new c2cpr:scalar;
new c1id:ID;
new c2id:ID;
let c1cpu=sca(c1cpr,Ggen(J))in out(c,c1cpu);
let c2cpu=sca(c2cpr,Ggen(J))in out(c,c2cpu);
((!!Client1(ver,c1cpr,c1cpu,c2cpu,J,c1id,c2id
    ))|(!Client2(ver,c2cpr,c2cpu,c1cpu,J,
    c2id)))
```

3.2.12 Verification of Secrecy and Authenticity for Spoofing Attack

Using the correspondence assertions, we can verify authenticity for clients. Event `accept`, which the client1 believes that it has accepted to run the protocol with the client2 and the correct sender key ID. Event `term`, which the client2 believes that it has terminated a protocol run with the client1 with the correct sender key ID. When a spoofing attack occurs, a fake sender key ID is used by an attacker and the correspondence assertions cannot be correct execution.

```
query attacker(M).
event accept(ID).
event term(ID).
query i:ID;
event(term(i))==>event(accept(i)).
```

3.2.13 Client1 Subprocess for Spoofing Attack

Here, we define client1's actions.

```
let Client1(ver:version,c1cpr:scalar,c1cpu:G,
    c2cpu:G,J:basis,c1id:ID)=
in(c,(c2cpu':G,c2id':ID));
if c2cpu=c2cpu' then
event accept(c1id);
new salt:coins;
let SharedSecret=sca(c1cpr,c2cpu')in
let Ken=Ggen2(Hash((SharedSecret,salt,Key)))
    in
```

```
let IVpre=Hash((SharedSecret,salt,IV))in
let IVen=Ggen3(xor(breakf(IVpre),breakb(IVpre
    )))in
let C=senccbc(M,Ken,IVen)in
let MACp=Hash(C)in
let MACe=sencecb(xor(breakf(MACp),breakb(MACp
    )),Ken)in
out(c,(ver,salt,C,MACe,c1id,c2id')).
```

3.2.14 Client2 Subprocess for Spoofing Attack

Here, we define client2's actions.

```
let Client2(ver:version,c2cpr:scalar,c2cpu:G,
    c1cpu:G,J:basis,c2id:ID)=
out(c,(c2cpu,c2id));
in(c,(ver':version,ran:coins,C':bitstring,mac
    :bitstring,id1:ID,id2:ID));
if c2id=id2 then
let SharedSecret'=sca(c2cpr,c1cpu)in
let Ken'=Ggen2(Hash((SharedSecret',ran,Key)))
    in
let MACp'=Hash(C')in
let MACe'=sencecb(xor(breakf(MACp'),breakb(
    MACp')),Ken')in
if mac=MACe' then
let IVpre'=Hash((SharedSecret',ran,IV))in
let IVen'=Ggen3(xor(breakf(IVpre'),breakb(
    IVpre')))in
let M'=sdeccbc(C',Ken',IVen')in
event term(id1).
```

3.2.15 Main Process for Spoofing Attack

Here, we define the version `ver`, base point `J`, Client1's ECDH private key `c1cpr`, client2's ECDH private key `c2cpr`, the sender key ID `c1id` and the recipient key ID `c2id`. The Client1's ECDH public key `c1cpu`, client2's ECDH public key `c2cpu` , `c1id` and `c2id` are exposed in channel `c`. Client1 subprocess and Client2 subprocess are executed in parallel.

```
process
new ver:version;
new J:basis;
new c1cpr:scalar;
new c2cpr:scalar;
new c1id:ID;
new c2id:ID;
out(c,c2id);
out(c,c1id);
let c1cpu=sca(c1cpr,Ggen(J))in out(c,c1cpu);
let c2cpu=sca(c2cpr,Ggen(J))in out(c,c2cpu);
((!Client1(ver,c1cpr,c1cpu,c2cpu,J,c1id))|(!
    Client2(ver,c2cpr,c2cpu,c1cpu,J,c2id)))
```

4. LINE's Claimed Security

Here, we recall the claimed security for the TEP and the E2EEP in the Technical Whitepaper [2] and the Status Report [4].

4.1 Claimed Security for TEP

In [2], the claim about security of the TEP is as follows:

> Messaging traffic between LINE clients and our servers is protected with forward-secure encryption, and both text messages and media streams in VoIP calls are end-to-end encrypted.

Though it just says that 'traffic is encrypted', it seems claims secrecy of both application data of clients and the server.

In [4], the claim about security of the TEP is as follows:

> In the event that LINE server's secret key is leaked, messages that were encrypted before the leak are protected.

It claims forward secrecy of application data, but it is ambiguous because which application data must be protected.

To summarize, the following security is claimed for the TEP.

- secrecy of both application data of clients and the server
- forward secrecy of application data

There is no security claim about authenticity.

4.2 Claimed Security for E2EEP

In [2], the claim about security of the E2EEP is as follows:

> LINE messages are locally encrypted on each client device before being sent to LINE's messaging server, and can only be decrypted by their intended recipient.

It claims secrecy of application data of clients against the server.

In [4], the claim about security of the E2EEP is as follows:

> Messages are encrypted on the client side before they are sent, and the content cannot be decrypted, even on LINE's servers. Forward secrecy in Letter Sealing[†] (if a user's device's secret key is leaked) is not supported.

The first sentence claims secrecy of application data of clients against the server. The second sentence claims that the E2EEP does not satisfy forward secrecy of application data.

To summarize, the following security is claimed for the E2EEP.

- secrecy of application data of clients against the server

There is no security claim about authenticity.

5. Verification Results

In this section, we show the verification results. To summarize, we obtain the following verification results as in Table 1 for the TEP and Table 2 for the E2EEP. In tables, we also describe LINE's security claim shown in Sect. 4.

[†]Letter Sealing is an implementation of the E2EEP.

Table 1 Verification results for TEP

	Secrecy of app. data		Forward secrecy of app. data		Server authenticity
	client's data	server's data	client's data	server's data	
LINE's claim	✓	✓	✓?	✓?	-
Our result	✓	✓	×	✓	✓

Table 2 Verification results for E2EEP

	Secrecy of message	Authenticity	
		Reply attack	Spoofing attack
LINE's claim	✓	-	-
Our result	✓	×	×

✓ means that no attack is found, × means that an attack is found, ✓? means that security may be claimed, and - means that there is no security claim.

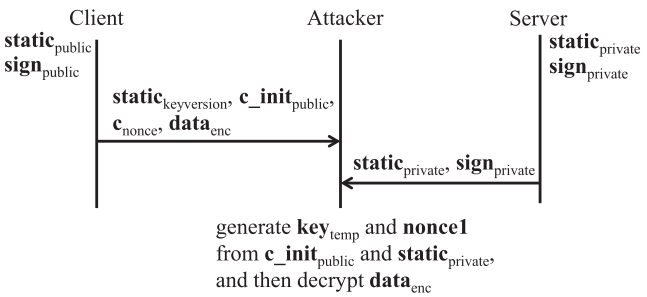

Fig. 3 Attack procedure of breaking forward secrecy

Next, we describe the found attack against forward secrecy for client's data of the TEP, and the found reply attack and spoofing attack for the E2EEP.

5.1 Attack to Break Forward Secrecy for Client's Data of TEP

1. An attacker who monitors channel **c** can know the **version**, initial ephemeral ECDH public key $\mathbf{c\_init}_{\text{public}}$, clint nonce $\mathbf{c}_{\text{nonce}}$ and ciphertext $\mathbf{data}_{\text{enc}}$, which the client sends to the server.
2. The attacker obtains the static private keys $\mathbf{static}_{\text{private}}$ and $\mathbf{sign}_{\text{private}}$.
3. By using $\mathbf{c\_init}_{\text{public}}$ and $\mathbf{static}_{\text{private}}$, the attacker generates temporary transport key $\mathbf{key}_{\text{temp}}$ and initialization vector(IV) $\mathbf{iv}_{\text{temp}}$.
4. By using $\mathbf{iv}_{\text{temp}}$ the attacker generates **nonce1**.
5. By using **nonce1** and $\mathbf{key}_{\text{temp}}$, the attacker decrypts $\mathbf{appdata}_{\text{client}}$.

Therefore, the attacker can obtain application data from the client by using the static private key of the server without interrupting the communication between the client and the server. It corresponds to breaking forward secrecy for client's data.

A procedure of the attack is given in Fig. 3.

5.2 Replay Attack to E2EEP

1. An attacker can know the static public information that the sender key ID **sender key ID**, ECDH public key $\mathbf{key}^{\mathbf{user1}}{}_{\text{public}}$, the recipient key ID **recipient key ID** and ECDH public key $\mathbf{key}^{\mathbf{user2}}{}_{\text{public}}$.
2. The attacker starts two sessions by initiating client2, and receives two ECDH public key $\mathbf{key}^{\mathbf{user2}}{}_{\text{public}}$.
3. The attacker sends the client2's ECDH public key $\mathbf{key}^{\mathbf{user2}}{}_{\text{public}}$ to client1.
4. Client1 returns **version**, **content type**, **salt**, **C**, **MAC**, **sender key ID** and **recipient key ID** to the attacker.
5. The attacker sends **version**, **content type**, **salt**, **C**, **MAC**, **sender key ID** and **recipient key ID** to client2 in the first session.
6. The attacker sends **version**, **content type**, **salt**, **C**, **MAC**, **sender key ID** and **recipient key ID** send to client2 in the second session.

Since **MAC** is valid both for the first session and the second session, client2 completes both sessions. It corresponds to the reply attack.

5.3 Spoofing Attack to E2EEP

1. An attacker can know the static information that the sender key ID **sender key ID**, ECDH public key $\mathbf{key}^{\mathbf{user1}}{}_{\text{public}}$, the recipient key ID **recipient key ID** and ECDH public key $\mathbf{key}^{\mathbf{user2}}{}_{\text{public}}$.
2. The attacker starts a session by initiating client2, and receives ECDH public key $\mathbf{key}^{\mathbf{user2}}{}_{\text{public}}$.
3. The attacker randomly generates a fake sender key ID **sender key ID'**.
4. The attacker sends client2's ECDH public key $\mathbf{key}^{\mathbf{user2}}{}_{\text{public}}$ and the recipient key ID **recipient key ID** to client1.
5. Client1 returns **version**, **content type**, **salt**, **C**, **MAC**, sender key ID **sender key ID** and recipient key ID **recipient key ID** to the attacker.
6. The attacker sends **version**, **content type**, **salt**, **C**, **MAC**, the fake sender key ID **sender key ID'** and recipient key ID **recipient key ID** to client2.

Since **MAC** does not depend on the sender key ID, client2 accepts **MAC** if the sender key ID is replaced. Hence, the attacker can impersonate the fake sender to client2. It corresponds to the spoofing attack.

6. Conclusion

We verified the security of LINE Encryption (Version 1.0), (i.e., the TEP and the E2EEP) by ProVerif. In LINE documents [2], [4], it is claimed that the TEP satisfies forward secrecy (i.e., even if the static private key is leaked, security of encrypted application data before leakage is guaranteed). However, it is not clear if both client's and server's data must be protected. We clarify actual forward secrecy of the TEP by showing an attack to break forward secrecy for client's data. In addition, we found a replay attack and a spoofing attack of the E2EEP.

Since all known attacks (the reply attack and the spoofing attack) and a new attack (breaking forward secrecy for client's data) are found, our result shows usefulness of ProVerif to verify security of messaging protocols like LINE. Finally, we note that the found attack for forward secrecy for client's data is not serious because the LINE security team says that client's data does not contain any sensitive information in the current implementation of LINE encryption. However, it may be potential vulnerability if an engineer use the TEP for another implementation.

References

[1] C. Shi and K. Yoneyama, "Verification of LINE Encryption Version 1.0 Using ProVerif," IWSEC 2018, pp.107–125, 2018.
[2] "LINE Encryption Overview (Ver.1.0)." https://scdn.line-apps.com/stf/linecorp/en/csr/line-encryption-whitepaper-ver1.0.pdf.
[3] W. Diffie, P.C. van Oorschot, and M.J. Wiener, "Authentication and Authenticated Key Exchanges," Des. Codes Cryptography, vol.2, no.2, pp.107–125, 1992.
[4] "LINE Encryption Status Report (2018.4.24)." https://linecorp.com/en/security/encryption_report.
[5] A.M. Espinoza, W.J. Tolley, J.R. Crandall, M. Crete-Nishihata, and A. Hilts, "Alice and Bob, who the FOCI are they?: Analysis of end-to-end encryption in the LINE messaging application," FOCI @ USENIX Security Symposium 2017, 2017.
[6] T. Isobe and K. Minematsu, "Breaking Message Integrity of an End-to-End Encryption Scheme of LINE," ESORICS (2) 2018, pp.249–268, 2018.
[7] B. Blanchet, B. Smyth, V. Cheval, and M. Sylvestre, "ProVerif 1.98." http://prosecco.gforge.inria.fr/personal/bblanche/proverif.
[8] "Public Key Cryptography for the Financial Services Industry, The Elliptic Curve Digital Signature Algorithm (ECDSA)," American National Standard X9.62-2005, 2005.
[9] H. Krawczyk and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)," RFC 5869, Internet Engineering Task Force. https://tools.ietf.org/html/rfc5869.
[10] "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) for Confidentiality and Authentication," NIST Special Publication 800-38D, 2007. https://csrc.nist.gov/publications/detail/sp/800-38d/final.

Cheng Shi received the B.E., degree from Civil Aviation University of China, Tianjin, China, in 2016. He is currently a graduate student at the University of Ibaraki, Ibaraki, Japan since 2018.

Kazuki Yoneyama received the B.E., M.E. and Ph.D. degrees from the University of Electro-Communications, Tokyo, Japan, in 2004, 2006 and 2008, respectively. He was a researcher of NTT Secure Platform Laboratories from 2009 to 2015. He is presently engaged in research on cryptography at the Ibaraki University, since 2015. He is a member of the International Association for Cryptologic Research (IACR), IPSJ and JSIAM.