The Effect of Using Weak Random Source on the State Synchronization of the Signal Protocol

DOI: https://doi.org/10.54654/isj.v1i21.1026

Trieu Quang Phong, Pham Duc Hung, Ngo Phuong Tuan

***Abstract*— Signal is an end-to-end security protocol used in instant messaging applications such as Facebook Messenger, Whatsapp, Zalo, etc. One of the outstanding features of the Signal protocol is its support for establishing end-to-end secure channels and conducting communications asynchronously, i.e., the sender can send messages securely even if the receiver is offline. This property is provided by maintaining synchronized states on each side. In our article, we will analyze the risk of desynchronizing states between parties in the Signal protocol if at least one of those parties in this protocol uses a weak random source.**

***Tóm tắt*— Signal là một giao thức bảo mật đầu cuối được sử dụng trong các ứng dụng nhắn tin nhanh như Facebook Messenger, Whatsapp, Zalo,… Một trong những tính năng nổi bật của giao thức Signal đó là hỗ trợ thiết lập kênh an toàn và nhắn tin bảo mật đầu cuối theo cách dị bộ, nghĩa là các tin nhắn vẫn có thể được gửi an toàn ngay cả khi người nhận ở trạng thái ngoại tuyến. Tính năng này được đảm bảo thông qua việc duy trì các trạng thái đồng bộ ở mỗi bên. Trong bài viết dưới đây, chúng tôi sẽ phân tích nguy cơ gây mất đồng bộ trạng thái ở các bên trong giao thức Signal nếu ít nhất một trong các bên trong giao thức này sử dụng nguồn ngẫu nhiên yếu.**

Keywords*— *Signal protocol, DR Algorithm, Message key, Chain key, DHRatchet public key.

Từ khóa*— *Giao thức Signal, Thuật toán DR, Khóa tin nhắn, Khóa chuỗi, Khóa công khai DHRatchet.

This manuscript is received on March 21, 2024. It is commented on May 24, 2024 and is accepted on June 20, 2024 by the first reviewer. It is commented on June 10, 2024 and is accepted on June 20, 2024 by the second reviewer.

I. INTRODUCTION

Nowadays, "end-to-end encryption" has become a popular concept in practical applications [1, [2]. The Signal protocol is a cryptographic protocol that provides end-to-end encryption for instant messaging. This protocol has been applied as a core component in some widespread applications such as WhatsApp [5], Google [6], Facebook Messenger and Skype, with billions of users.

Historically, the first secure protocol for instant messaging was proposed in 2004 by Nikita Borisov and his co-workers, which is known as the OTR (Off-the-Record Messaging) protocol [1, [2]. An important feature that OTR provides is key freshness by using the Diffie-Hellman ratcheting technique (denoted by DHRatchet), where the user establishes a new ephemeral Diffie–Hellman (DH) shared secret to protect each outgoing and incoming message. Although the OTR protocol has relatively limited adoption, its ratcheting technique is used in modern security protocols.

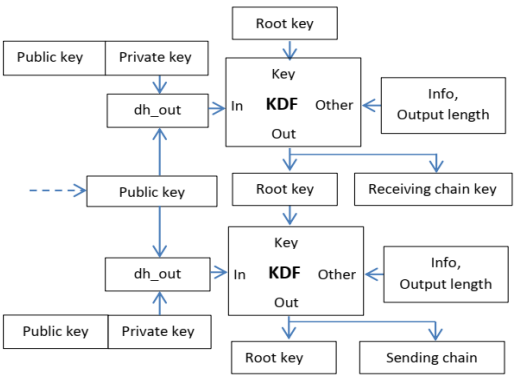

Figure 1. Block diagram of the DHRatchet Algorithm

The above technique also appears in the design of the Signal protocol. Specifically, by combining the efficiency of the idea of OTR's DHRatchet technique and a symmetric ratchet technique (denoted by SymRatchet) we obtain the core algorithm of the Signal protocol known as the Double Ratchet (DR) algorithm. According to this algorithm, three following types of keys are maintained and updated throughout the process of exchanging secret information between parties:

- *Root key:* A 32-byte value that is used to create a new chain key via the key derivation function in the DHRatchet algorithm (see Figure 1).

- *Chain key:* A 32-byte value that is used to create a new message key via the key derivation function (Figure 2). Each call to that function is considered as a SymRatchet operation. Note that each party will maintain two types of chain keys named as the sending chain key (for encrypting outgoing messages) and the receiving chain key (for decrypting incoming messages).

- *Message key:* A 32-byte value that is used (at most) once by the sender and receiver to execute the message encryption on the sender side and the message decryption on the receiver side.

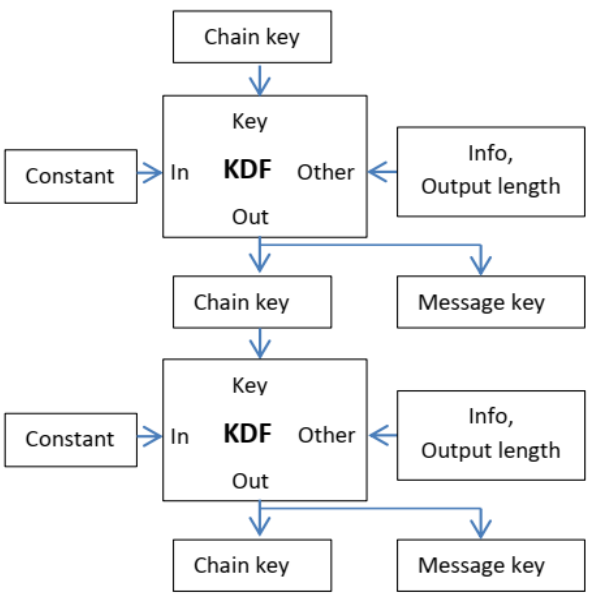

Figure 2. Block diagram of the SymRatchet Algorithm

The Signal protocol includes three main steps as follows for a communication between two parties Alice and Bob:

• ***Registration:*** At installation (and periodically afterwards), each party independently registers an identity and a prekey bundle (containing some identity key, signed prekey, and a set of one-time prekey) with a key distribution server S. These keys will then be used to establish sessions via the X3DH protocol (see [8]).

• ***Session establishment:*** The session initiator (Alice) sends a request and get back a prekey bundle of the receiver (Bob) from the server S to establish a X3DH session key and then use it to setup a long-lived communication session.

• ***Secure message exchange:*** After establishing a secure communication session, participants Alice and Bob can exchange their secure messages in one of the following ways:

– *Synchronous messaging:* This operation is executed when Alice sends a message to Bob (or vice versa) immediately after received at least on "new" message from Bob. In this case, Alice will need to perform one DHRatchet operation to update its root key and chain keys, and then derive its message key from the sending chain key via the SymRatchet algorithm.

– *Asynchronous messaging:* This operation is executed when Alice sends a message to Bob (or vice versa) but has not received any new message from Bob. In this case, Alice will need to perform one SymRatchet operation to derive a message key from its recent sending chain key.

According to the above process, each message sent by Alice or Bob is encrypted with a new message key, which strengthens the forward and backward security. Another advantage of the Signal protocol is that it allows parties to establish an end-to-end secure messaging channel asynchronously. Note that, in order to handles lost or out-of-order messages the Signal protocol maintains a list of skipped

message keys for encrypting the messages arrived late.

The above properties of the Signal protocol have been analyzed and proven in several recent works [11], [12], [13],... However, those results are all evaluated after modifying the Signal protocol. For example, in [13], the list of skipped message keys will be indexed through an epoch counter (which counts the order of DHRatchet steps performed at each party) rather than being indexed via the DHRatchet public key in the incoming message (according to the original description of the Signal protocol).

In this article, we will show that the above difference will be significant in cases where the sender uses weak random sources. Specifically, in those cases, we will show the risks of losing state synchronization on the original Signal protocol while the modification variant of J. Alwen and his co-workers shown in [13] still ensures synchronization of states.

The structure of our article includes five parts. In particular, Part I presents an overview of the Signal protocol. Part II presents a description of this protocol. Part III presents some risks of losing synchronization between parties in the Signal protocol if a weak random source on one side is used. Part IV presents some discussions on the above problem of losing synchronization. Finally, the conclusions are presented in Part V.

II. THE SIGNAL DESCRIPTION

In this section, we will recall the process of sending and receiving the encrypted messages of the Signal protocol as described in [9].

A.The External Functions

In order to perform the DR algorithm, it is need to define the following functions:

- GENERATE_DH(): Returns a new DH key pair. In this function, we need to use a source of randomness to ensure the freshness of key pairs. Therefore, if the source of randomness is weak, the probability that two DH key pairs created by GENERATE_DH() (at different times) are the same is significant.

- DH(dh_pair, dh_pub): Returns an output from Diffie-Hellman computation between the private key from Diffie-Hellman key pair dh_pair and the Diffie-Hellman public key dh_pub. If this function rejects some invalid public key, it may return an exception to stop the process.

- KDF: An underlying key derivation function, such as HKDF [14].

- KDF_RK(rk, dh_out): Returns 64-byte keys (a 32-byte root key and a 32-byte chain key) as the output of applying KDF with a 32-byte key rk to dh_out, some constant string info and output length 64. For instance,

KDF_RK(rk, dh_out)=KDF(rk,dh_out,info,64)

- KDF_CK(ck): Returns a pair of 64-byte keys (a 32-byte chain key and a 32-byte message key) as the ouput of applying KDF with a 32-byte key ck to some constant const, some constant string info and output length 64. For instance,

KDF_CK(ck)=KDF(ck,const,info,64)

- ENCRYPT(mk, m, ad): Returns an AEAD encryption of m with the message key mk. The associate data ad is authenticated but is not a part of the output ciphertext.

- DECRYPT(mk, c, ad): Returns the AEAD decryption of c by using the message key mk and the associate data ad. If the authentication fails, this function ouputs an exception to stop the process.

- HEADER(dh_pair, pn, n): Makes a message header which includes the *DHRatchet* public key from *dh_pair*, the number of message keys in previous (sending) chain *pn*, and the message's number in recent (sending) chain *n.*

- CONCAT(ad, h): Encodes a header h into a parseable byte sequence and then prepends the byte sequence ad.

Besides, the Signal protocol uses a MAX_SKIP value to specify the maximum of skipped message key in a receiving chain.

B. The State Variables

The following state variables are maintained by each party:

- DHs: the sending DHRatchet key pair
- DHr: the receiving DHRatchet public key
- RK: the root key with 32-byte length
- CKs, CKr: the 32-byte chain keys for sending and receiving.
- Ns, Nr: the message numbers for sending and receiving.
- PN: the message number in the previous sending chain.
- MKSKIPPED: Dictionary of skipped message keys, indexed by the receiving DHRatchet public key and message number.

C. The Initialization Process

After both parties have agreed on a 32-byte shared secret key SK and Bob's DHRatchet public key bob_dh_public_key by using the X3DH protocol, Alice in the initiator role will call *RatchetInitAlice()* to initialize its state stA:

```
def RatchetInitAlice(stA, SK,bob_dh_public_key):
    stA.DHs = GENERATE_DH()
    stA.DHr = bob_dh_public_key
    stA.RK, st.CKs = KDF_RK(SK, DH(st.DHs, st.DHr))
    stA.CKr = None
    stA.Ns = 0
    stA.Nr = 0
    stA.PN = 0
    stA.MKSKIPPED = {}
```

and Bob in the responder role will call *RatchetInitBob()* to initialize its state stB:

```
def RatchetInitBob(stB, SK, bob_dh_key_pair):
    stB.DHs = bob_dh_key_pair
    stB.DHr = None
    stB.RK = SK
    stB.CKs = None
    stB.CKr = None
    stB.Ns = 0
    stB.Nr = 0
    stB.PN = 0
    stB.MKSKIPPED = {}
```

D. Encrypting Messages

In [9], the function RatchetEncrypt() is defined to encrypt messages. This function takes the sender's state st, a plaintext m, an associate data ad (which is a concatenation of sender's public identity key and receiver's public identity key) as its input, and then runs the following steps:

```
def RatchetEncrypt(st, m, ad):
1. st.CKs, mk = KDF_CK(st.CKs)
2. h = HEADER(st.DHs, st.PN, st.Ns)
3. st.Ns += 1
4. return h, ENCRYPT(mk, m, CONCAT(ad, h))
```

E. Decrypting Messages

In [9], the function RatchetDecrypt() is defined to decrypt ciphertext c with header h in the incoming message. By using the receiver's state st and the associate data ad (which is a concatenation of sender's public identity key and receiver's public identity key), this function does as the following:

```
def RatchetDecrypt(st, h, c, ad):
1. m = TrySkippedMessageKeys(st, h, c, ad)
2. if m != None:
    2.1. return m
3. if h.dh != st.DHr:
    3.1. SkipMessageKeys(st, h.pn)
    3.2. DHRatchet(st, h)
4. SkipMessageKeys(st, h.n)
5. st.CKr, mk = KDF_CK(st.CKr)
6. st.Nr += 1
7. return DECRYPT(mk, c,CONCAT(ad, h))

def TrySkippedMessageKeys(st, h, c, ad):
1. if (h.dh, h.n) in st.MKSKIPPED:
  1.1. mk = st.MKSKIPPED[h.dh, h.n]
  1.2. del st.MKSKIPPED[h.dh, h.n]
  1.3. return DECRYPT(mk, c, CONCAT(ad, h))
2. else:
  2.1. return None

def SkipMessageKeys(st, until):
1. if st.Nr + MAX_SKIP < until:
  1.1. raise Error()
```

```
2. if st.CKr != None:
  2.1. while st.Nr < until:
      2.1.1. st.CKr, mk = KDF_CK(st.CKr)
      2.1.2. st.MKSKIPPED[st.DHr, st.Nr] = mk
      2.1.3. st.Nr += 1

def DHRatchet(st, h):
1. st.PN = st.Ns
2. st.Ns = 0
3. st.Nr = 0
4. st.DHr = h.dh
5. st.RK, st.CKr = KDF_RK(st.RK, DH(st.DHs, st.DHr))
6. st.DHs = GENERATE_DH()
st.RK, st.CKs = KDF_RK(st.RK, DH(st.DHs, st.DHr))
```

Remark 1: In the open source libsignal-protocol-C [7], the implementation of RatchetDecrypt() is different from the one described above, including:

- Step 3 in RatchetDecrypt() above will be changed into the first step. Note that, with such a modification, we will temporarily obtain h.dh = st.DHr in the following steps, and
- The call TrySkippedMessageKeys() will check whether (h.dh, h.n) was ever in the st.MKSKIPPED list instead of just determining whether it currently exists in that list. This comes from the facts: (1) if TrySkippedMessageKeys() is called, we will have h.dh = st.DHr because of adjusting Step 3 into the first step in RatchetDecrypt(); and (2) if st.Nr > h.n then (h.dh, h.n) must have been in the st.MKSKIPPED list since (h.dh, st.Nr) = (st.DHr, st.Nr) is included in the st.MKSKIPPED list only if the indexes (st.DHr, until) with until < st.Nr were previously included in the st.MKSKIPPED list. In the case that (h.dh, h.n) was in the st.MKSKIPPED list but now has been removed, a warning is issued about the incoming message having been previously processed and stops the decryption process. This modification is to prevent the risk of increased computation on the receiving side due to an attacker replaying a validly encrypted message many times.

III. The Risk of Losing Synchronization from Weak Random Source

A.The Risk 1

In this section, we will present the first risk that a valid message generated by the sender Alice (according to the encryption algorithm RatchetEncrypt() in Section II.0) may not be accepted by the receiver Bob (according to the decryption algorithm RatchetDecrypt() in Section II.0), if the random source used by sender A is weak (i.e, there is a high probability that the public key $\mathrm{DHs_A}$ generated by Alice after a DHRatchet step is the same as the DHRatchet public key sent previously in the same communication with Bob). Specifically, we have the following statement.

Statement 1: Assuming that Alice and Bob are conducting a communication session using the Signal protocol implemented with a secure encryption scheme and a secure key derivation function, then Bob cannot successfully decrypt the message sent from Alice if the following conditions occur:

(1) The current execution of the DHRatchet step and some previous execution of the DHRatchet step at Alice during communication with Bob produce the same sending DHRatchet public key $\mathrm{DHs_A}$.

(2) Alice has sent more than one message corresponding to the public key $\mathrm{DHs_A}$ to Bob in the previous execution. Assuming that there exists one of those messages is successfully processed by Bob, but the first one is lost.

(3) In the current execution, the first message before any subsequent messages corresponding to the public key $\mathrm{DHs_A}$ sent by Alice was received by Bob.

Proof. Note that since in the same communication secured by the Signal protocol, we always have that the sending DHRatchet public key of the sender will be equal to the receiving DHRatchet public key of the receiver, so we can also deduce that Bob has received the public key $\mathrm{DHr_B} = \mathrm{DHs_A}$ in some message sent by Alice in the past (according to the first and second conditions). By using again the second

condition, we can obtain that the message key corresponding to index $(\mathrm{DHs}_A, 0)$ has been saved in the stB.MKSKIPPED list of Bob and has not been deleted.

Furthermore, according to the third condition, now now Bob need to a encrypted message with (hA.dh,hA.n)=$(\mathrm{DHs}_A, 0)$ from Alice. Because Bob receives this message before any subsequent messages, Bob will decrypt it using the RatchetDecrypt() algorithm (in Section II.E). Hence, Bob will make a call TrySkippedMessageKeys(stB, hA, cA, ad) to check whether $(\mathrm{DHs}_A, 0)$ is in the list stB.MKSKIPPED or not. However, as pointed out above, this check is passed. Then, Bob will extract a message key mk_B corresponding to index $(\mathrm{DHs}_A, 0)$ in the list stB.MKSKIPPED and use it to decrypt cA. Note that the message key mk_B and the message key that is used by Alice to create the ciphertext cA are derived in different executions, so they are distinct with an overwhelming probability if applying a secure KDF function. It implies that Bob cannot successfully decrypt cA. Therefore, the call TrySkippedMessageKeys(stB, hA, cA, ad) will return an error and stop the decryption process on side Bob.

Thus, we have shown a case where there is a significant probability that a valid message sent from Alice will not be accepted by Bob if Alice uses a weak random source.

B. The Risk 2

In this section, we will continue to present another risk that causes a valid message to be rejected by the receiver. It is similar to in the previous section, this risk also comes from the fact that there is a significant probability the public key DHs_A generated by Alice after the current DHRatchet step is equal to the DHRatchet public key sent previously in the same communication with Bob if Alice use a weak random source. Note that the risk in this section will focus on implementing the RatchetDecrypt() function in [7], which was mentioned in Remark 1. In that case, RatchetDecrypt(stB, hA, cA, ad) would first require checking hA.dh = stB.DHr. If this condition occurs, (hA.dh, hA.n) is not in the stB.MKSKIPPED list and stB.Nr>hA.n, this algorithm will return an error (because the message has been previously processed) and stop the decryption process.

Indeed, we will show that the above risk will occur with a significant probability if there are two consecutive executions of the DHRatchet step at Alice during communication with Bob produce the same sending DHRatchet public key, assuming it is DHs_A. Specifically, we have the following statement.

Statement 2: Assuming that Alice and Bob are conducting a communication session using the Signal protocol (implemented with the libsignal-protocol-C library [5]), then Bob cannot successfully decrypt the message sent from Alice if the following conditions occur:

(1) The current execution of the DHRatchet step and the immediately previous execution of the DHRatchet step at Alice during communication with Bob produce the same sending DHRatchet public key DHs_A.

(2) Alice has sent more than one message corresponding to the public key DHs_A to Bob in the previous execution. Assuming that the first message and another of them are successfully processed by Bob.

(3) In the current execution, the first message corresponding to the public key DHs_A sent by Alice was received by Bob.

Proof. According to the third condition, now Bob need to a encrypted message with (hA.dh,hA.n)=$(\mathrm{DHs}_A, 0)$ from Alice by using the implementation in the libsignal-protocol-C library [5]. Firstly, Bob has to check whether stB.DHr=hA.dh or not. However, because of the first condition, this check will be passed without triggering the DHRatchet step on the Bob side. Hence, the state stB remains unchanged. The process of decrypting cA will continue with a call to TrySkippedMessageKeys(stB,hA,cA,ad), which will need to check whether $(\mathrm{DHs}_A, 0)$ is in the previous stB.MKSKIPPED list.

According to the second condition, we have stB.Nr>0 and the message key

corresponding to the index $(\mathrm{DHs}_A, 0)$ has been removed from the list stB.MKSKIPPED. As discussed in Remark 1, since stB.Nr>0 and $(\mathrm{DHs}_A, 0)$ is not in the stB.MKSKIPPED list (according to the first condition in this case), an error will be returned to report "the message has been previously processed" and stop the decryption process.

Thus, we have shown another case that causes the receiver Bob rejects a valid message from the sender Alice. Although we need a strict condition that requiring two consecutive executions of DHRatchet on side Alice to produce the same sending public key, it can occur if a weak random source is used.

IV. DISSCUSION

Loss of state synchronization between participants using Signal protocol is an issue considered in [10]. There, de-synchronization of states can occur if the state of the sender or receiver device has been rolled back, or the state of the receiver device has been deleted. These situations may happen when delete the personal data on their device. To solve this problem, [10] provided the option of allowing the recipient to send a “retry request” to the receiver.

However, the risk we point out here does not follow the scenario in [10], meaning we still consider the parties to still operate normally. Furthermore, the use of a weak random source can cause the loss of state synchronization to occur more frequently than the cases given in [10]. Therefore, applying the solution of sending a “resend” request to the recipient as given in [10] may lead to overload on some participant who uses a weak random source.

Also note that the random source on the user's device may come from the device being injected with malicious code during use. Therefore, installing good randomness sources for user devices is not yet a complete solution to prevent the risks of de-synchronization pointed out in this article. Instead, we recommend further adoption of the epoch counter as in [13] to index message keys.

V. CONCLUSION

Currently, the Signal protocol has been widely applied in end-to-end security applications. In this protocol, messages will be protected with a new key, so maintaining synchronized states between parties is essential to ensure that a valid message can be successfully decrypted on the receiver side.

In this article, we have presented two risks of causing loss of state synchronization between parties in this protocol. These risks come from the observation that message keys have the same indies because of a weak source of randomness. As a result, the decryption process may decrypt a valid message with a message key that is different from what created the corresponding ciphertext. Consequently, the process of decrypting this message is unsuccessful.

We consider the risks according to the technical report [9] and the libsignal-protocol-C library [7] for the Signal protocol. As discussed in Part IV, these risks are non-trivial. Besides, to prevent these risks, message keys should be indexed by additionally using an epoch counter as in [13].

Furthermore, experimenting with the above risks to increase the significance of the theoretical analysis given in this article will be our next problem.

REFERENCES

[1] Khánh, T. V., & Vinh, N. T. (2020). Giải pháp bảo mật đầu cuối cho điện thoại di động. *Journal of Science and Technology on Information Security*, *9*(01), 37-48. https://doi.org/10.54654/isj.v9i01.41.

[2] Vinh, C. T., & Huong, P. V. (2023). Constructing a Model Combining Zalo and End-to-End Encryption for Application in Digital Transformation. *Journal of Science and Technology on Information Security*, *3*(20), 95-108. https://doi.org/10.54654/isj.v3i20.1012

[3] Nikita Borisov, Ian Goldberg, and Eric Brewer. “Off-the-record Communication, or, Why Not to Use PGP”. In: WPES. Washington DC, USA: ACM, 2004, pp. 77–84.

[4] Mario Di Raimondo, Rosario Gennaro, and Hugo Krawczyk. “Secure Off-the-record Messaging”. In: WPES. Alexandria, VA, USA: ACM, 2005, pp. 81–89.

[5] "WhatsApp's Signal Protocol integration is now complete". *Signal*. Signal Blog. 2016. Archived from the original on 29 January 2021. Retrieved 5 April 2016.
[6] Bohn, Dieter (19 November 2020). "Google is rolling out end-to-end encryption for RCS in Android Messages beta". *The Verge*. Vox Media, Inc. Retrieved 28 November 2020.
[7] D. Konigsberg. "libsignal-protocol-c", 2020.https://github.com/signalapp/libsignal-protocol-c.
[8] T. Perrin and M. Marlinspike, "The X3DH Key Agreement Protocol," 2016. https://whispersystems.org/docs/specifications/x3dh/
[9] M. Marlinspike and T. Perrin. The double ratchet algorithm, 2016. https://whispersystems.org/docs/specifications/doubleratchet/doubleratchet.pdf.
[10] Marlinspike, M., & Perrin, T. (2017). The sesame algorithm: session management for asynchronous message encryption. *Revision*, *2*, 2017.
[11] Joseph Jaeger and Igors Stepanovs. Optimal channel security against fine-grained state compromise: The safety of messaging. In *CRYPTO 2018, Part I*, pages 33-62, 2018.
[12] Bertram Poettering and Paul Rösler. Asynchronous ratcheted key exchange. Cryptology ePrint Archive, Report 2018/296, 2018. https://eprint.iacr.org/2018/296.
[13] Alwen, J., Coretti, S., & Dodis, Y. (2019, April). The double ratchet: security notions, proofs, and modularization for the signal protocol. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (pp. 129-158). Cham: Springer International Publishing.
[14] Krawczyk, H., & Eronen, P. (2014). Hmac-based extract-and-expand key derivation function (hkdf)(May 2010). *URL: http://tools. ietf. org/html/rfc5869, accessed*, 11-29.

ABOUT THE AUTHOR

Trieu Quang Phong

Workplace: Institute of Cryptographic Science and Technology, Vietnam Government Information Security Commission

Email: phongtrieu53@gmail.com

Education: The BS in Department of Mathematics, Hanoi University of Science (2014). The PhD in Mathematics at Academy of Military Science and Technology – Ministry Of National Defence.

Recent research direction: The provable security for the signature schemes and the key exchange protocol.

Tên tác giả: **Triệu Quang Phong**

Cơ quan làm việc: Viện Khoa học - Công nghệ mật mã, Ban Cơ yếu Chính phủ

Email: phongtrieu53@gmail.com

Quá trình đào tạo: Cử nhân Khoa Toán, Đại học Khoa học Tự nhiên Hà Nội năm 2014. Tiến sỹ Toán học, Viện Khoa học và Công nghệ quân sự, Bộ Quốc phòng.

Hướng nghiên cứu hiện nay: Tính bảo mật có thể chứng minh được đối với sơ đồ chữ ký và giao thức trao đổi khóa.

Pham Duc Hung

Workplace: Institute of Cryptographic Science and Technology, Vietnam Government Information Security Commission.

Email: phamhungdg96@gmail.com

Education: The Information Technology Engineer, Military Technical Academy (2019).

Recent research direction: Information security

Tên tác giả: **Phạm Đức Hùng**

Cơ quan công tác: Viện Khoa học - Công nghệmật mã, Ban Cơ yếu Chính phủ

Email: phamhungdg96@gmail.com

Quá trình đào tạo: Kỹ sư Công nghệ thông tin, Học Viện Kỹ thuật Quân sự (2019).

Hướng nghiên cứu hiện nay: An toàn thông tin

Ngo Phuong Tuan

Workplace: Institute of Cryptographic Science and Technology, Vietnam Government Information Security Commission.

Email: ngophuongtuan841701@gmail.com

Education: The BS in Department of Cryptography, Academy of the Federal Security Service of the Russian Federation (2022).

Recent research direction: Provable security; Public key cryptography.

Tên tác giả: **Ngô Phương Tuấn**

Cơ quan công tác: Viện Khoa học - Công nghệ mật mã, Ban Cơ yếu Chính phủ

Email: ngophuongtuan841701@gmail.com

Quá trình đào tạo: Cử nhân Khoa Mật mã, Học viện An ninh Liên bang Nga (2022).

Hướng nghiên cứu hiện nay: Tính bảo mật có thể chứng minh được; Mật mã khoá công khai