Journal of Surveillance, Security and Safety

Original Article

Open Access

Split keys for station-to-station (STS) protocols

Gizem Akman[1,2], Mohamed Taoufiq Damir[1,2], Philip Ginzboorg[3], Sampo Sovio[3], Valtteri Niemi[1,2]

[1]Department of Computer Science, University of Helsinki, Helsinki 00014, Finland.
[2]Helsinki Institute for Information Technology (HIIT), Espoo 02150, Finland.
[3]Huawei Technologies Oy, Helsinki 00180, Finland.

Correspondence to: Gizem Akman, Department of Computer Science, University of Helsinki, Helsinki, 00014, Finland. E-mail: gizem.akman@helsinki.fi

How to cite this article: Akman G, Damir MT, Ginzboorg P, Sovio S, Niemi V. Split keys for station-to-station (STS) protocols. *J Surveill Secur Saf* 2023;4:62-93. http://dx.doi.org/10.20517/jsss.2023.16

Received: 31 May 2023 **First Decision:** 25 Aug 2023 **Revised:** 8 Sep 2023 **Accepted:** 14 Sep 2023 **Published:** 22 Sep 2023

Academic Editor: Josef Pieprzyk **Copy Editor:** Yanbin Bai **Production Editor:** Yanbin Bai

Abstract

Aim: For authentication and key agreement, it is advisable to reduce the risks of key exposure and provide an additional level of control over key usage. This can be achieved by splitting the secret key across several devices, requiring their cooperation to use the key effectively.

Methods: We have studied the split-key setting in the context of the station-to-station with key derivation function (STS-KDF) protocol – a well-known two-party authenticated key agreement protocol based on the Diffie-Hellman key exchange and digital signatures – and developed it further. We use the methods of design science, modeling, and formal verification.

Results: First, we have found a new reflection attack against the STS-KDF protocol for scenarios where several entities share the same private key. We designed a modification of that protocol, called STS-KDF with certificate binding (STS-KDF-CB), that includes measures against this attack and enhances user privacy. Second, we designed the STS-KDF-CB with the key encapsulation mechanism (KEM) protocol, where KEM is used instead of the Diffie-Hellman key exchange and digital signatures. Third, we designed split-key variants of the STS-KDF-CB and STS-KDF-CB with KEM protocols. The security properties of the STS-KDF protocol, the STS-KDF-CB protocols, and their split-key variants were formally verified using the ProVerif tool.

Conclusion: We have increased security and privacy for authentication and key agreement by developing new variants of the STS-KDF protocol. In addition, we have STS-KDF variants for the split key setting. Future work includes implementation of the protocols and extension to the case where one of the split-key devices provides attestation for the other.

Keywords: Authenticated key agreement, station-to-station, split key, formal verification, ProVerif

1 INTRODUCTION

We are concerned with the usage and storage of cryptographic keys in situations where the keys are vulnerable to exposure, either because they are not protected by the platform security of the underlying hardware or because an attacker may get physical access to the device using the keys.

An example of such a situation is the usage and storage of cryptographic keys in a household network, which typically includes a heterogeneous set of IoT devices that have different levels of platform security. Constrained IoT devices lack sophisticated platform security features. On the other hand, smartphone and personal computer platforms include hardware-based process isolation and secure storage. In addition, these sophisticated devices may also have a hardware-based root of trust, such as a trusted platform module (TPM)[1] and embedded secure element (eSE)[2]. Another example is when the cryptographic keys are installed in a device, e.g., a wireless base station, or stored in a data center; these keys are vulnerable to physical attacks.

One way to address this issue is to split the key between two or more devices, such that both devices must cooperate when using that key. This mitigates the issue because compromising two or more devices is, in general, harder than compromising a single device. Moreover, the split provides an additional level of control over the key usage, which makes attacks, such as device cloning, more difficult. In the above examples, the key could be split between the constrained IoT device and a smartphone, between the wireless base station and an entity in a more secure location (e.g., in the core network), or between two different computers in a data center.

The station-to-station (STS) protocol is an authenticated key agreement (AKA) protocol with key confirmation that combines the Diffie-Hellman (DH) key agreement [3] and signature-based authentication of the two parties[4]. Blake and Menezes[5] have shown that STS is vulnerable to Unknown Key Share (UKS) attacks and proposed a variant, STS with Key Derivation Function (STS-KDF), that prevents these attacks. Jackson *et al.* have shown using the Tamarin Prover that STS-KDF satisfies the following security properties: key secrecy, identity agreement, and strong session agreement[6]. We chose to study the STS-KDF protocol in the split-key scenario because it is a well-studied and well-known protocol, supports an abstract two-party setting, and has been formally verified. In particular, the STS-KDF protocol is formally shown secure, assuming only minimal properties from the underlying signature scheme.

While designing the split-key variant of the STS-KDF protocol, we have found a new reflection attack on the STS-KDF. This attack is applicable in scenarios where, for privacy reasons, several STS-KDF endpoints use the same key pair.

The key encapsulation mechanism (KEM) is an efficient and provably secure public encryption scheme that generates encrypted random keys by combining asymmetric and symmetric encryption techniques[7,8]. KEM is an umbrella term for several protocols, such as Elliptic Curve Integrated Encryption Scheme-based KEM (ECIES-KEM), RSA-KEM, etc[9]. In the DH key agreement, the shared key is constructed based on the public parameters of the two parties. With KEM, the shared key is generated by one party and sent to the other, encrypted with the public key of the receiver. The security of the DH key agreement is based on the hardness of the discrete logarithm problem. In contrast, KEM is more flexible regarding the used public-key cryptosystem and the protocol design. It is possible to construct a KEM from almost any public key primitive[8]. For example, the security of KEM could be based on the hardness of a discrete logarithm problem or another hard computational problem, e.g., the hardness of decoding a general linear code. For these reasons, KEM has become a popular cryptographic primitive, and there are studies to adopt KEM into TLS 1.3. Furthermore, secure KEMs, i.e., indistinguishability under chosen ciphertext attack (IND-CCA) KEMs, can be constructed from weaker public key encryption schemes by applying, e.g., the Fujisaki-Okamoto transform[10]. IND-CCA is the de facto security property for modern public key encryption schemes. Moreover, there exist split-key KEMs[11]. Considering the advantages of KEM presented above, we have designed a variant of the STS-KDF protocol that uses KEM (instead of the DH key agreement and signatures) and a split-key variant of the STS-KDF with KEM.

Contributions:

1. A new reflection attack on the STS-KDF protocol in situations where two parties have the same key pair but different identities.
2. A modification of the STS-KDF protocol: Privacy-Enhanced STS-KDF-CB protocol that mitigates the attack and, in addition, protects the privacy of the user identity. This protocol can be seen as a variant of SIGMA-I protocol[12].
3. A further variant of the protocol: Privacy-Enhanced STS-KDF-CB with KEM, that uses Key Encapsulation Mechanism (KEM)[8] instead of Diffie–Hellman key exchange and signatures.
4. The "split-key" variants for the Privacy-Enhanced STS-KDF-CB protocol, with and without KEM, where the asymmetric key pair of one or both of the parties is split between several devices.
5. Automated analysis of protocol models using ProVerif tool[13] and proofs of the security properties of the above protocols.

The rest of the paper is organized as follows. Section 2 contains the background information on our study and related work. The system and adversary models are defined in Section 3. The reflection attack against the STS-KDF protocol is explained in Section 4. In Section 5, we present the privacy-enhanced variants of STS-KDF that mitigate the reflection attack and their adaptations to the split-key scenarios. The formal verification of the protocols is described in Section 6. The paper ends with the analysis of the performance of the protocols in Section 7 and the discussion in Section 8.

2 BACKGROUND AND RELATED WORK

In this section, we review the following topics: formal verification tools, secret splitting for authentication, STS protocol, Authenticated Key Exchange (AKE), KEM, and Authenticated Encryption (AE).

2.1 Formal verification tools for security protocols

The application of formal methods for the analysis of cryptographic protocol means using automated formal analysis tools to determine whether an attacker can prevent the protocol from achieving its security objectives[14]. Formal methods have proven valuable when developing critical systems, such as protocols, where safety or security is important. In fact, various protocols were believed to be secure for years, and then formal verification showed later that it was not the case. One classic example is the Needham-Shroeder protocol[15], which was developed in 1978 and later proven to be insecure by Lowe[16] in 1996.

ProVerif[13] and Tamarin-Prover[17] are state-of-the-art tools used for formally verifying and analyzing security protocols. The Dolev-Yao adversary model is utilized in both tools.

ProVerif uses applied pi-calculus as a formal language, translates the protocol into a set of Horn clauses, and considers an unbounded number of sessions and an unbounded message space for the protocol analysis. The tool can detect attacks. If a protocol property cannot be proven, ProVerif attempts to construct an execution trace that contradicts that property.

Tamarin-Prover offers comprehensive support for modeling and analyzing security protocols. The specification of protocols and adversaries employs an expressive language based on multiset rewriting rules. These rules define a labeled transition system, where the state includes a symbolic representation of the knowledge of the adversary, the messages on the network, freshly generated values, and the state of the protocol. The adversary and protocol interaction involves updating network messages and generating new ones.

2.2 Secret splitting for authentication

The internet standard IETF RFC 5026[18] introduces a split scenario in a model where the entity that provides service is different from the entity that authenticates and authorizes the user, e.g., the network access provider differs from the mobility service authorizer.

Several authentication schemes exist, e.g., biometric verification, text passwords, public key infrastructure, and symmetric-key-based authentication techniques. Multi-tier authentication schemes are more secure than single sign-on, considering layered defense; however, multi-tier authentication schemes increase the complexity of user experience[19]. To reduce the computational overhead in a system with multiple sources of trust, Choi *et al.* present a new multi-source authentication scheme called Split-Join One-Way Key Chain (SOKC) that stores the parts of the keys in the source nodes[20].

Shah *et al.*[21] and Wang *et al.*[22] present a multi-factor authentication with the secret-splitting concept of biometrics, including exclusive-or operations, DH key exchange algorithms, and encryption algorithms. The proposed approaches split the biometric data into two, encrypt both parts, and store one on a smart card and another on a server. The proposed solutions are secure against authentication factor attacks, network attacks, and interior attacks from the card-issuing organization.

Choi *et al.*[23] suggest a solution for a certificate-based authentication system using a password and a secondary source, i.e., an honest-but-curious server or another mobile device. In this setup, the user has a secret key to authenticate himself to a controller (a website or an application). To prevent the danger of compromising the device of the user, some secret values are utilized, each of which is used to encrypt their secret key. The secret values are stored in the secondary sources. When the user wants to sign into the controller, he retrieves the secret value from the secondary sources to decrypt the secret key. Therefore, the user can authenticate himself to the controller only with the contribution of secondary sources.

2.3 Station-to-station protocol

The STS protocol was proposed by Diffie *et al.*[4] in 1992 to obtain mutual entity authentication and mutual explicit key authentication[24]. STS is an AKA based on the DH key exchange protocol and authenticated signatures. Diffie *et al.* provide variants of the STS protocol that have explicit key confirmation by using a symmetric-key encryption scheme and a message authentication code (MAC)[4]. Those variants are called STS-ENC and STS-MAC, respectively, in Blake-Wilson *et al.*[5].

Blake-Wilson *et al.* show that STS-ENC and STS-MAC are vulnerable to UKS attacks[5]. To prevent these attacks, the authors propose to apply the Key Derivation Function (KDF) on the shared DH key. This variant was later named as STS-KDF by Jackson *et al.*[6], who also provide formal verification of the variants of the STS protocol by using the Tamarin Prover. They show that STS-KDF is the only variant that satisfies all the following security requirements: key secrecy, identity agreement, and strong session agreement (injective agreement) while using signatures that are secure under the Existential Unforgeability under an Adaptive Chosen Message Attack (EUF-CMA) model.

Next, we describe the STS-KDF protocol. Let Alice and Bob be two parties that use the protocol to agree on a shared key. Each party has a public and a secret key for signature verification and signing, respectively: (pk_a, sk_a) is the public and secret key pair of Alice, and (pk_b, sk_b) is the public and secret key pair of Bob. The parties authenticate the public key of each other through the certificates, where $cert_a$ includes pk_a, the identity id_a of Alice, and possibly other information; $cert_b$ includes pk_b, the identity id_b of Bob, and possibly other information. The certificates are signed by the Certificate Authority (CA). The steps of an STS-KDF protocol run are shown in the listing Protocol 1 and in Figure 1.

Protocol 1 : STS-KDF

Setup: $Alice$ or Bob choose a safely large prime p and a generator g $(mod\ p)$, where p and g are public.
The Protocol: (cf. Figure 1.)

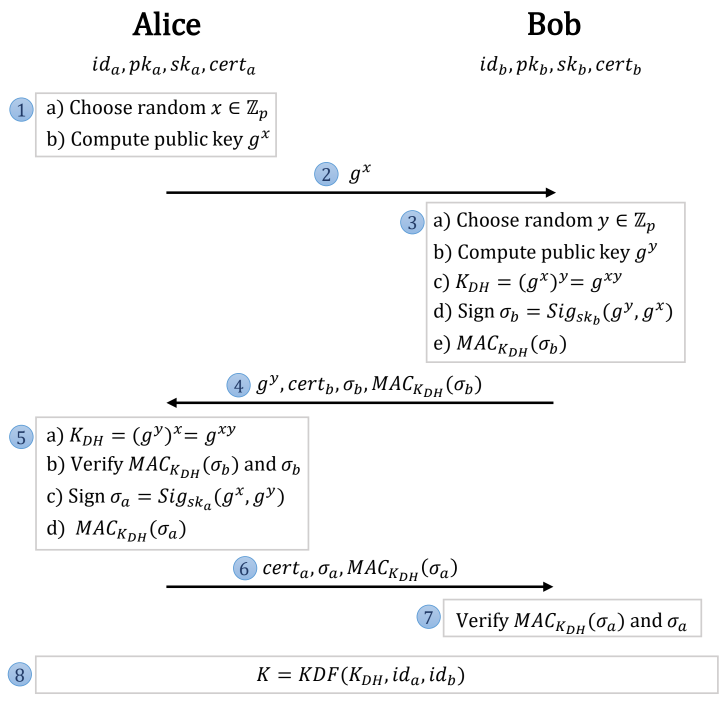
1. (a) Alice chooses random $x \in \mathbb{Z}_p$.
   (b) Alice computes her public DH key: g^x.
2. Alice sends g^x to Bob.
3. (a) Bob chooses random $y \in \mathbb{Z}_p$.
   (b) Bob computes his public DH key: g^y.
   (c) Bob computes the shared DH key: $K_{DH} = (g^x)^y = g^{xy}$.
   (d) Bob signs his public DH key, along with the public DH key of Alice, by using his secret signing key: $\sigma_b = Sig_{sk_b}(g^y, g^x)$.
   (e) Bob also computes the MAC of the signature by using the shared DH key: $MAC_{K_{DH}}(\sigma_b)$.
4. Bob sends g^y, $cert_b$, σ_b, and $MAC_{K_{DH}}(\sigma_b)$ to Alice.
5. (a) Alice computes the shared DH key: $K_{DH} = (g^y)^x = g^{xy}$.
   (b) Alice verifies $MAC_{K_{DH}}(\sigma_b)$, the certificate $cert_b$, and the signature σ_b.
   (c) Alice signs her public DH key, along with the public DH key of Bob, by using her secret signing key: $\sigma_a = Sig_{sk_a}(g^x, g^y)$.
   (d) Alice also computes the MAC of the signature by using the shared DH key: $MAC_{K_{DH}}(\sigma_a)$.
6. Alice sends $cert_a$, σ_a, and $MAC_{K_{DH}}(\sigma_a)$ to Bob.
7. Bob verifies $MAC_{K_{DH}}(\sigma_a)$, the certificate $cert_a$, and the signature σ_a.
8. Alice and Bob compute the shared session key with the shared DH key and identities of Alice and Bob: $K = KDF(K_{DH}, id_a, id_b)$.

2.4 Authenticated key exchange

Authenticated Key Exchange (AKE) aims to provide two communicating parties some assurance that they know the identities of each other and share a secret key only known to them[4]. The shared secret key can be used further to provide privacy, data integrity, or both.

Krawczyk[12] proposes several authenticated DH key exchange protocols under the name SIGMA. These protocols adapt the sign-and-MAC approach using DH key exchange and have been used in Internet Key Exchange (IKE) protocols[25]. The author begins by studying STS protocols and builds the SIGMA protocols upon them. The basic form of SIGMA is similar to STS-KDF. Still, the MAC computation differs from the STS-KDF in that only the certificate or identifier is used as an input instead of the signature, while the MAC key is derived from the DH key. Another variation of SIGMA includes encryption of all the messages except for the public key to provide identity protection.

Krzywiecki *et al.*[26] designed an AKE protocol for wearable devices by improving the SIGMA protocol of Krawczyk[12]. Krzywiecki *et al.*[26] aim to prevent impersonation attacks by splitting the signing key into two signing modules and renewing the partial keys. They claim that the impersonation attack would be successful



Alice
Bob
$id_a, pk_a, sk_a, cert_a$
$id_b, pk_b, sk_b, cert_b$
1 a) Choose random $x \in \mathbb{Z}_p$ b) Compute public key g^x
2 g^x
3 a) Choose random $y \in \mathbb{Z}_p$ b) Compute public key g^y c) $K_{DH} = (g^x)^y = g^{xy}$ d) Sign $\sigma_b = Sig_{sk_b}(g^y, g^x)$ e) $MAC_{K_{DH}}(\sigma_b)$
4 $g^y, cert_b, \sigma_b, MAC_{K_{DH}}(\sigma_b)$
5 a) $K_{DH} = (g^y)^x = g^{xy}$ b) Verify $MAC_{K_{DH}}(\sigma_b)$ and σ_b c) Sign $\sigma_a = Sig_{sk_a}(g^x, g^y)$ d) $MAC_{K_{DH}}(\sigma_a)$
6 $cert_a, \sigma_a, MAC_{K_{DH}}(\sigma_a)$
7 Verify $MAC_{K_{DH}}(\sigma_a)$ and σ_a
8 $K = KDF(K_{DH}, id_a, id_b)$

Figure 1. STS-KDF protocol. STS-KDF: station-to-station with key derivation function.

only if the attacker captures both partial keys.

2.5 Key encapsulation mechanism

A KEM is a public encryption scheme that produces a shared key that can be used for symmetric encryption. The notion of KEMs was introduced by Cramer and Shoup[7] to build an efficient hybrid encryption scheme with an unrestricted message space.

KEM: The KEM consists of three algorithms[7,8]:

1. **Key Generation** is a probabilistic, polynomial-time key generation algorithm that takes a security parameter k as input and generates a pair of public and secret keys (PK, SK):
   $(PK, SK) \longleftarrow KEM\_KeyGen(1^k)$.
2. **Key Encapsulation** is a probabilistic, polynomial-time encryption algorithm that takes a public key PK as input and generates a cipher text C and a KEM key K:
   $(C, K) \longleftarrow KEM\_Encaps(PK)$.
3. **Key Decapsulation** is a deterministic, polynomial-time decryption algorithm that takes a ciphertext C and the secret key SK as input and recovers the KEM key K:
   $K \longleftarrow KEM\_Decaps(C, SK)$.

The KEM key K, the output of the functions KEM_Encaps and KEM_Decaps, is shared between communicating parties.

Split KEM: Next, we consider KEM in a secret-splitting scenario. Ebina *et al.* introduced a distributed KEM decapsulation, where the secret key SK is split into n parts, SK_1, SK_2, ..., SK_n, and the encapsulated key K is reconstructed from t out of n potential partial decapsulation results, $K_1, K_2, \cdots, K_n$[11]. The Split KEM below is an adaptation of their construction for our scenario, where $t = n = 2$. For the purposes of formal analysis, we have combined some of the seven algorithms in Section 2.3 of Ebina *et al.*[11] to form the following set of five.

1. **Split Key Generation** is a probabilistic, polynomial-time key generation algorithm that takes a security parameter k as input and outputs public encapsulation key PK, two decapsulation key shares SK_1, SK_2, and a verification key vk:
   $(PK, (SK_1, SK_2), vk) \longleftarrow SplitKEM\_KeyGen(1^k)$.
2. **Split Key Encapsulation** is a probabilistic, polynomial-time encryption algorithm that takes a public key PK as input and outputs a ciphertext C and a session key K:
   $(C, K) \longleftarrow SplitKEM\_Encaps(PK)$.
3. **Split Key Decapsulation** is a deterministic, polynomial-time decapsulation algorithm that takes decapsulation key shares SK_1 and SK_2 and the ciphertext C as input and outputs session key shares K_1 and K_2, respectively:
   $K_1 \longleftarrow SplitKEM\_Decaps(C, SK_1)$.
   $K_2 \longleftarrow SplitKEM\_Decaps(C, SK_2)$.
4. **Split Key Reconstruction** is a deterministic, polynomial-time key generation algorithm that takes the two session key shares K_1, K_2 as input and outputs the session key K:
   $K \longleftarrow SplitKEM\_Recons(K_1, K_2)$.
5. **Split Key Share Verification** is a deterministic, polynomial-time algorithm that takes PK, C, vk, and a key share K_i as input and outputs 0 or 1:
   $0/1 \longleftarrow SplitKEM\_Verif(PK, C, K_i, vk)$ for $i = 1, 2$.
   We say that K_i is a *valid* share if $SplitKEM\_Verif(PK, C, K_i, vk) = 1$.

The outputs, K_1 and K_2, of the function $SplitKEM\_Decaps$, are reconstructed using the function $SplitKEM\_Recons$ to obtain the shared key K, which is identical to the second output of the function $SplitKEM\_Encaps$.

2.6 Authenticated encryption

The communication between two parties over a network requires two main security goals: privacy and authentication. Achieving both privacy and authentication at the same time is called authenticated encryption (AE). The three basic ways to achieve AE are (1) MAC-then-Encrypt; (2) Encrypt-then-MAC; and (3) Encrypt-and-MAC[27].

Since some application settings require associated data, which should be authenticated without encryption, in addition to the authenticated and encrypted data, the notion of AE with associated data (AEAD) is introduced. The AEAD was first formalized by Rogaway *et al.* in 2002 using the generic composition of a nonce-based, privacy-only encryption scheme and a pseudorandom function[28].

The National Institute of Standards and Technology (NIST) initiated a standardization process for lightweight cryptography to select one or more schemes for AEAD in 2015 and announced the decision in 2023. The permutation-based AEAD and a hashing scheme named ASCON are chosen to be standardized[29].

3 SYSTEM MODEL

An Authentication and Key Agreement (AKA) protocol typically takes place between two parties. The aim is to establish a key where both parties contribute to deriving a shared secret key. In addition, the protocol should provide authentication, confirming that both entities are who they claim to be. Informally, an AKA protocol should mimic a face-to-face key agreement; a few security properties can capture such a requirement.

We aim to present an AKA protocol that can also be used in split-key settings. The rest of this section covers the split-key scenarios, adversary models, and security properties that we are interested in.

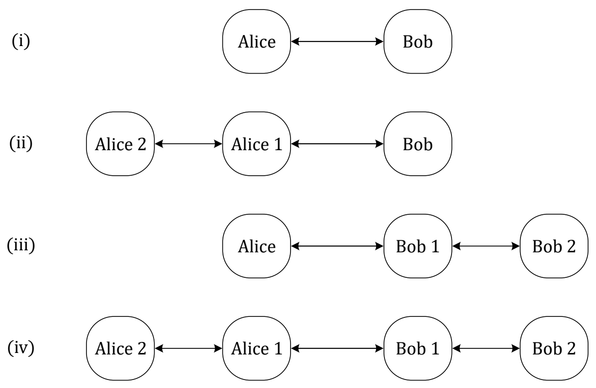


Figure 2. Cases of the system model: (i) no split key; (ii) Alice's keys are split between Alice1-Alice2; (iii) Bob's keys are split between Bob1-Bob2; (iv) both Alice's and Bob's keys are split between Alice1-Alice2 and Bob1-Bob2, respectively.

3.1 Split-key scenarios

We model the scenario where the secret keys of the users could be split with, e.g., an assistant device. In our model, the users are called Alice and Bob. Alice splits her key into two pieces and distributes each piece in two devices, Alice1 and Alice2, where Alice2 is the assistant device. Alice requires the full cooperation of both Alice1 and Alice2 to use her key, e.g., for digital signature. Similarly, Bob splits his key between Bob1 and Bob2. The following four cases of the split key scenario are illustrated in Figure 2.

(i) No Split Key (Alice-Bob);
(ii) Split Key for Alice (Alice2-Alice1-Bob);
(iii) Split Key for Bob (Alice-Bob1-Bob2);
(iv) Split Key for both Alice and Bob (Alice2-Alice1-Bob1-Bob2).

Note that our secret splitting can be seen as a special case of threshold secret sharing where the threshold t equals the number of shares n and $n = 2$. However, in our setting, the secret is not reconstructed from the shares, which is in contrast with the traditional notion of secret sharing introduced in[30]. See also[31] for different types of secret sharing.

3.2 Adversary model

As described in Section 3.1, the system model has two parties, say, Alice and Bob, that communicate over an unsecured channel. When the split key scenarios are considered, the number of parties in the model can be three or four.

We assume that the outsiders are Dolev-Yao adversaries; the attacker (a) can see any message that is sent to the network, (b) is a legitimate user of the network and can start a conversation with other users, and (c) has an opportunity to be a receiver of any initiator of the protocol[32]. In addition to this definition, any message that is sent through the network by legitimate users is assumed to be either created or passed on by the attacker[14]. In brief, the attacker, namely a Dolev-Yao adversary, "can compose messages, replay them, or decipher them if she knows the right keys, but cannot otherwise crack encrypted messages"[33].

We assume perfect cryptographic primitives, i.e., attackers can decrypt secrets only if they possess the corresponding keys, and hash functions are assumed to be one-way and collision-resistant. Moreover, we consider a more powerful adversary following the extended Canetti–Krawczyk (eCK) model[34]. In particular, the adversary is allowed to compromise some parties, e.g., by physical access or malware in a device[14]. Furthermore, we consider the possibility of the key reveals.

3.3 Security properties

In the following, we give an overview of the main security properties provided by our proposed AKA protocols. For simplicity, we state the security properties in the case of an AKA protocol between two parties, say A and B.

Secrecy: One of the main goals of an AKA protocol is to agree on a cryptographic session key between the participating parties. The protocol is said to achieve session key secrecy if the shared key between A and B at the end of the protocol is not revealed to any malicious third party.

Mutual Authentication: Mutual authentication in an AKA protocol between two parties ensures that both parties are who they claim to be. In other words, mutual authentication is achieved if both sides correctly verify each other's identity.

Forward Secrecy: An AKA protocol satisfying forward secrecy ensures that the compromise of a long-term private key does not compromise past session keys resulting from past AKA executions.

Resistance to Unknown Key Share (UKS) attack: An AKA protocol, say between two parties A and B, is said to be vulnerable to a UKS attack if, by the end of the protocol, A believes that she shares the session key with B, which is the case, while B ends up believing that he shares the key with a party (e.g., the attacker) that is different from A.

Resistance to Key Compromise Impersonation (KCI) attack: A key compromise impersonation (KCI) attack against an AKA protocol is successful if the attacker manages to impersonate a party by compromising a private key.

Identity Confidentiality: An AKA protocol is said to provide identity confidentiality if the protocol protects the parties' privacy, especially their identities.

4 REFLECTION ATTACK AGAINST STATION-TO-STATION WITH KEY DERIVATION FUNCTION

We consider scenarios where the same secret and public key pairs are assigned to all members in a group, e.g., to protect user privacy. For example, members of a group can be devices belonging to the same production batch. One such scenario with the use of the same attestation key for protecting user privacy is recommended by W3C API specification for Web Authentication[35] and FIDO Alliance[36]. Based on these recommendations, the Google-certified attestation key pairs for Android hardware-backed keystore are deployed to devices in batches of a minimum of 10^4 devices per key[37]. Also, a key pair used for device attestation could be unique at the level of a device model rather than at the level of a single device[38].

In these scenarios, it is conceivable that two devices, having the same key pair but different identities, would like to establish a secure communication channel based on that key pair. For example, an Android phone would like to check if it is connected to another genuine Android phone, even in the case where the attestation keys of the two devices are identical. Using STS-KDF for this purpose has the advantage of perfect forward secrecy – if the key pair is compromised in one of the devices, past communications will not be compromised. However, we have found the following reflection attack on STS-KDF in this setting.

Assume Alice and Bob have the same secret key for signing, sk; therefore, the same public key, pk. Recall that although we use the terms Alice and Bob, these can be two machines communicating with each other. The certificates of Alice and Bob include pk and the identity id_a or id_b, respectively. The steps of the attack are listed below and depicted in Figure 3.

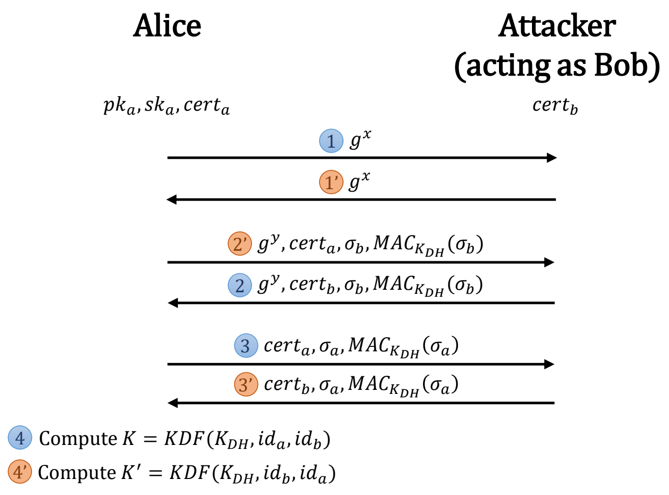

Alice
Attacker (acting as Bob)
$pk_a, sk_a, cert_a$
$cert_b$
1 g^x
1' g^x
2' $g^y, cert_a, \sigma_b, MAC_{K_{DH}}(\sigma_b)$
2 $g^y, cert_b, \sigma_b, MAC_{K_{DH}}(\sigma_b)$
3 $cert_a, \sigma_a, MAC_{K_{DH}}(\sigma_a)$
3' $cert_b, \sigma_a, MAC_{K_{DH}}(\sigma_a)$
4 Compute $K = KDF(K_{DH}, id_a, id_b)$
4' Compute $K' = KDF(K_{DH}, id_b, id_a)$

Figure 3. Reflection attack against STS-KDF.

1. One session of communication (Session 1) starts with Alice sending her public DH key, g^x, (Step 2 of Figure 1) to Bob. The attacker captures this message.

1'. Another session (Session 2) starts with the attacker sending g^x to Alice (Step 2 of Figure 1). Note that in Session 2, the attacker is the initiator, and Alice is the responder, whereas Alice was the initiator in Session 1.

2'. Alice (acting as the responder) chooses another secret y and computes public DH key g^y and shared DH key g^{xy}. Alice signs (g^y, g^x) with her secret key sk and computes MAC with the shared DH key (Step 3 of Figure 1). Alice sends the signature, public DH key g^y, her certificate, and MAC to the attacker (Step 4 of Figure 1).

2. The attacker gets the signature, public DH key g^y, and MAC and sends them, along with the certificate of Bob (Step 4 of Figure 1), to Alice in Session 1. Note that signing keys are the same; therefore, the public key in the certificate of Bob can be used to verify the signature. Also, g^x is the initiator's key, and g^y is the responder's key in both sessions.

3. Alice verifies the MAC and signature she received in Step 2, computes the shared DH key, signs (g^x, g^y), and computes MAC of the new signature using the shared DH key (Step 5 of Figure 1). Alice sends her certificate, signature, and MAC to the attacker (Step 6 of Figure 1).

3'. This time, the attacker captures the signature and MAC and sends them, along with the certificate of Bob, to Alice in Session 2 (Step 6 of Figure 1).

4. Alice derives the key K.

4'. After verifying the signature and the MAC, Alice (acting as the responder) derives the key K'.

As a result of this attack, Alice agrees – allegedly with Bob – on two identical session keys, K and K'. However, Bob is not aware of these agreements. The attacker authenticates himself as Bob to Alice. Even though the attacker does not get to know what the shared DH key K_{DH} or the shared session keys K and K' are, he has deceived Alice into believing that she has successfully completed both sessions of the protocol.

Depending on the purpose of the key usage, the reflection attack can cause trouble to Alice and Bob. For example, Alice might be in control of a group of devices, including Bob's device, and give commands to entities in the group. Since Alice thinks she has agreed on a key with Bob, she would send commands and requests to Bob, which would be encrypted with the shared key. Being unaware of the key establishment, Bob would not follow the commands and respond to the requests. For example, the request from Alice could require an immediate response to protect the security of the system, such as destroying data or sending information to another entity, and an unresponsive Bob would put the system in danger. Moreover, if the attacker reflects Alice's messages to Bob back to Alice, then Alice could think that these messages were sent for her, and she might destroy or send her own data.

The attack can be realized independently from the type of signature used in the protocol. Thus, the classification of signature schemes in the paper of Jackson *et al.*[6] does not matter for the reflection attack.

Please note that the reflection attack against an STS-KDF protocol also works in the case where each party sharing the same signing key would have individual splits for that same key. Alice's signing key, sk, is split between Alice1 and Alice2, and the public key that corresponds to sk stays the same. After Alice1 and Alice2 have done the split key signature with their partial keys, the resultant signature would be the same as in the case it would be signed directly with sk. Note also that in communication between Alice and Bob, Bob would not be able to distinguish whether Alice is using the split key or not.

In the next section, we will present protocols that can mitigate this reflection attack.

5 PROTOCOL DESCRIPTIONS

Below, we present four new variants of STS-KDF and their adaptations for the split-key scenarios (see Section 3.1). Our goal is to add the following security properties to the basic STS-KDF (described in Section 2.3): (1) privacy enhancements such that outsiders cannot learn the identities of the parties, (2) protection against the reflection attack described in Section 4, and (3) measures against single-point failures by splitting keys with assistant devices. First, in Section 5.1, we introduce two protocols that provide properties (1) and (2). Then, in Section 5.2, we extend these protocols to the split key case, providing property (3).

5.1 New variants of station-to-station protocols

We will now describe two protocols for the "no split-key" setting, depicted as the case (i) in Figure 2: the Privacy-Enhanced STS-KDF-CB protocol and the Privacy-Enhanced STS-KDF-CB protocol with KEM that uses KEM instead of Diffie–Hellman key exchange and signature.

Next, we present Privacy-Enhanced STS-KDF-CB. In this protocol, the DH Key Exchange and the signature scheme are used as in the generic STS-KDF. Second, we adopt the KEM to the STS-KDF-CB protocol to provide flexibility in implementing different cryptosystems.

5.1.1 Privacy-enhanced STS-KDF certificate-binding (STS-KDF-CB) protocol

The Privacy-Enhanced STS-KDF-CB protocol protects against the reflection attack, and the parties can be anonymous toward outsiders because certificates and signatures are encrypted before transmission.

Protocol 2 : Privacy-Enhanced STS-KDF-CB

Setup: Alice and Bob choose a safely large prime p and a generator $g \ (mod \ p)$, where p and g are public.
The Protocol: (cf. Figure 4).

1. (a) Alice chooses random $x \in \mathbb{Z}_p$.
   (b) Alice computes her public DH key: g^x.
2. Alice sends g^x to Bob.
3. (a) Bob chooses random $y \in \mathbb{Z}_p$.
   (b) Bob computes his public DH key: g^y.
   (c) Bob computes the shared DH key: $K_{DH} = (g^x)^y = g^{xy}$.
   (d) Bob signs his public DH key along with the public DH key of Alice by using his secret signing key: $\sigma_b = Sig_{sk_b}(g^y, g^x)$.
   (e) Bob applies AEAD on his signature and certificate: $enc_1 = AEAD_{K_{DH}}(\sigma_b, cert_b)$. It should be noted that AEAD allows the receiver to verify the integrity of encrypted and unencrypted information in the message by, e.g., Encrypt-then-MAC[39].
4. Bob sends g^y and enc_1 to Alice.
5. (a) Alice computes the shared DH key: $K_{DH} = (g^y)^x = g^{xy}$.
   (b) Alice then decrypts the enc_1 to get the signature and the certificate of Bob: $(\sigma_b, cert_b) = DEC_{K_{DH}}(enc_1)$.
   (c) Alice verifies the certificate $cert_b$, the fact that $cert_b \neq cert_a$, and the signature σ_b.
   (d) Alice signs her public DH key, along with the public DH key of Bob, by using her secret signing key: $\sigma_a = Sig_{sk_a}(g^x, g^y)$.
   (e) Alice applies AEAD on her signature and certificate: $enc_2 = AEAD_{K_{DH}}(\sigma_a, cert_a)$.
6. Alice sends enc_2 to Bob.
7. (a) Bob decrypts the enc_2 to get the signature and the certificate of Alice: $(\sigma_a, cert_a) = DEC_{K_{DH}}(enc_2)$.
   (b) Bob verifies the certificate $cert_a$, the fact that $cert_a \neq cert_b$, and the signature σ_a.
8. Alice and Bob derive the session key K from the DH key and their certificates: $K = KDF(K_{DH}, cert_a, cert_b)$. Please note that other parameters could also be added to diversify the key, such as session ID.

5.1.2 *Privacy-enhanced STS-KDF-CB protocol with KEM*

Next, we present the Privacy-Enhanced STS-KDF-CB Protocol with KEM, where we replace both the DH key exchange and the signature scheme with KEM. The certified public keys of the parties are used for the KEM instead of the signature scheme.

In this adaptation, we assume that Alice already knows who she is communicating with; therefore, she has the certificate of Bob when she starts the protocol. Only Alice needs to encrypt her certificate and send it to Bob to prove herself.

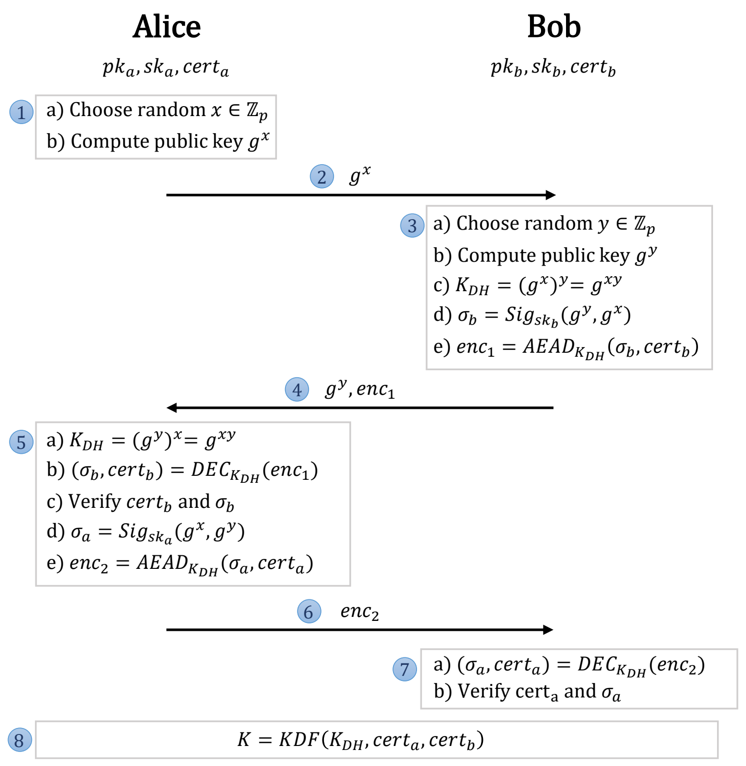

Alice
Bob
$pk_a, sk_a, cert_a$
$pk_b, sk_b, cert_b$
1 a) Choose random $x \in \mathbb{Z}_p$
b) Compute public key g^x
2 g^x
3 a) Choose random $y \in \mathbb{Z}_p$
b) Compute public key g^y
c) $K_{DH} = (g^x)^y = g^{xy}$
d) $\sigma_b = Sig_{sk_b}(g^y, g^x)$
e) $enc_1 = AEAD_{K_{DH}}(\sigma_b, cert_b)$
4 g^y, enc_1
5 a) $K_{DH} = (g^y)^x = g^{xy}$
b) $(\sigma_b, cert_b) = DEC_{K_{DH}}(enc_1)$
c) Verify $cert_b$ and σ_b
d) $\sigma_a = Sig_{sk_a}(g^x, g^y)$
e) $enc_2 = AEAD_{K_{DH}}(\sigma_a, cert_a)$
6 enc_2
7 a) $(\sigma_a, cert_a) = DEC_{K_{DH}}(enc_2)$
b) Verify $\mathrm{cert_a}$ and σ_a
8 $K = KDF(K_{DH}, cert_a, cert_b)$

Figure 4. Privacy-Enhanced STS-KDF Certificate-Binding (STS-KDF-CB) protocol. STS-KDF: station-to-station with key derivation function.

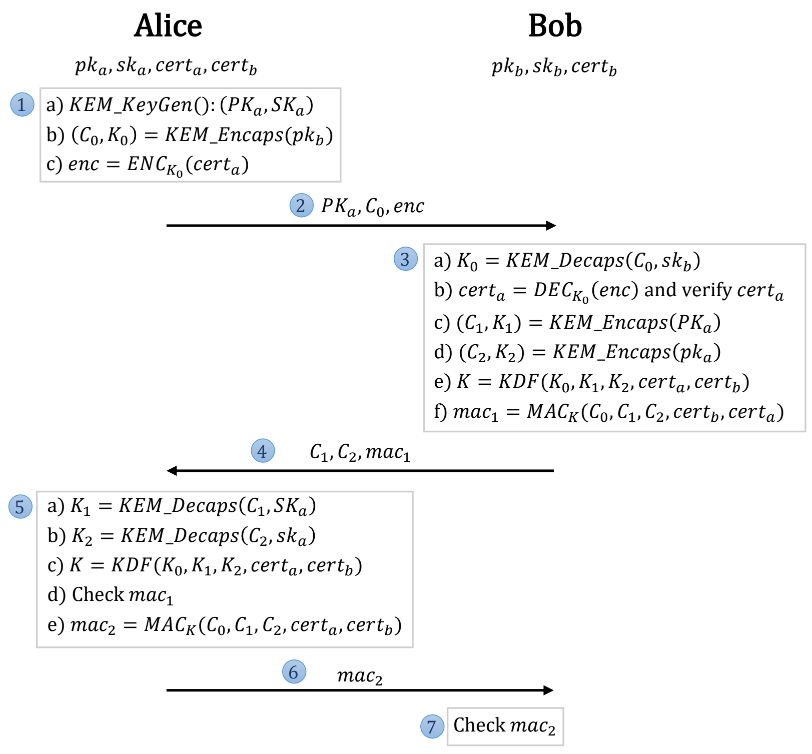

Alice
Bob
$pk_a, sk_a, cert_a, cert_b$
$pk_b, sk_b, cert_b$
1 a) $KEM\_KeyGen(): (PK_a, SK_a)$
b) $(C_0, K_0) = KEM\_Encaps(pk_b)$
c) $enc = ENC_{K_0}(cert_a)$
2 PK_a, C_0, enc
3 a) $K_0 = KEM\_Decaps(C_0, sk_b)$
b) $cert_a = DEC_{K_0}(enc)$ and verify $cert_a$
c) $(C_1, K_1) = KEM\_Encaps(PK_a)$
d) $(C_2, K_2) = KEM\_Encaps(pk_a)$
e) $K = KDF(K_0, K_1, K_2, cert_a, cert_b)$
f) $mac_1 = MAC_K(C_0, C_1, C_2, cert_b, cert_a)$
4 C_1, C_2, mac_1
5 a) $K_1 = KEM\_Decaps(C_1, SK_a)$
b) $K_2 = KEM\_Decaps(C_2, sk_a)$
c) $K = KDF(K_0, K_1, K_2, cert_a, cert_b)$
d) Check mac_1
e) $mac_2 = MAC_K(C_0, C_1, C_2, cert_a, cert_b)$
6 mac_2
7 Check mac_2

Figure 5. Privacy-Enhanced STS-KDF-CB protocol with KEM.

Protocol 3 : Privacy-Enhanced STS-KDF-CB with KEM

Setup: $Alice$ and Bob apply $KEM\_KeyGen()$ to generate their public-secret key pairs, pk_a-sk_a and pk_b-sk_b, respectively. $Alice$ and Bob have certificates for their public keys, $cert_a$ and $cert_b$.
The Protocol: (cf. Figure 5.)

1. (a) Alice generates the key pair (PK_a, SK_a) with the $KEM\_KeyGen()$ function. Note that the newly generated PK_a and SK_a are not the same as certified pk_a and sk_a.
   (b) Alice knows and has verified the certificate of Bob before starting the protocol, so she knows his public key pk_b. By using this public key, Alice applies the encapsulation function and gets a ciphertext and a session key: $(C_0, K_0) = KEM\_Encaps(pk_b)$.
   (c) Alice then encrypts her certificate by using the session key derived above: $enc = ENC_{K_0}(cert_a)$.
2. Alice sends PK_a, C_0, and enc to Bob.
3. (a) Bob applies the decapsulation function on the ciphertext C_0 and his secret key sk_b: $K_0 = KEM\_Decaps(C_0, sk_b)$.
   (b) Bob decrypts enc to get the certificate of Alice: $cert_a = DEC_{K_0}(enc)$. Bob also verifies that the certificate of Alice is not the same as his certificate, i.e., $cert_a \neq cert_b$.
   (c) Bob applies the encapsulation function on the public key PK_a to get the session key and the ciphertext: $(C_1, K_1) = KEM\_Encaps(PK_a)$.
   (d) Bob also applies the encapsulation function on the certified public key pk_a to get another session key and the ciphertext: $(C_2, K_2) = KEM\_Encaps(pk_a)$.
   (e) Bob then derives the shared session key from the session keys: $K = KDF(K_0, K_1, K_2, cert_a, cert_b)$.
   (f) Bob computes the MAC of the ciphertexts and the certificates using the derived key K: $mac_1 = MAC_K(C_0, C_1, C_2, cert_b, cert_a)$.
4. Bob sends the ciphertexts C_1, C_2, and mac_1 to Alice.
5. (a) Alice applies the decapsulation function on the ciphertext C_1 and her secret key SK_a to get the session key: $K_1 = KEM\_Decaps(C_1, SK_a)$.
   (b) Alice then applies the decapsulation function on the ciphertext C_2 and her secret key sk_a to get the other session key: $K_2 = KEM\_Decaps(C_2, sk_a)$.
   (c) Alice then derives the shared session key: $K = KDF(K_0, K_1, K_2, cert_a, cert_b)$.
   (d) Alice verifies the mac_1.
   (e) Alice computes the MAC of the ciphertexts and the certificates using the shared session key K: $mac_2 = MAC_K(C_0, C_1, C_2, cert_a, cert_b)$.
6. Alice sends mac_2 to Bob.
7. Bob verifies the mac_2.

Please note that replacing the DH key exchange in STS-KDF with a KEM can be done as follows. Using the same notations as in Figure 1, a KEM version of STS-KDF is achieved by minor changes at Alice's and Bob's sides. On Alice's side, it is enough to replace generating a random x by generating a pair of KEM public-secret keys (PK, SK), and then Alice sends PK instead of g^x. At Bob's side, generating a random y is replaced by encapsulation using the received PK, and finally, Bob sends the ciphertext resulting from encapsulation instead of sending g^y. On both sides, the DH key is replaced by the session key produced by the KEM. Such a KEM variant of STS-KDF, however, would inherit the security properties of the original protocol. In particular, it would be vulnerable to the attack described in Section 4.

Similar modifications can be done for the STS-KDF-CB protocol in Figure 4, with the additional change of using the KEM key instead of the DH key in Step 8. This variant would protect against the reflection attack of Section 4.

Protocol 4 : Split-Key Privacy-Enhanced STS-KDF-CB

Setup:

i. Alice and Bob choose a safely large prime p and a generator g $(mod\ p)$, where p and g are public.

ii. Alice1 and Alice2 generate a public key pk_a for signature verification and split the secret signing key into the shares, sk_{a_1} and sk_{a_2}, respectively. Similarly, Bob1 and Bob2 generate a public key pk_b for signature verification and split the secret signing key into the shares, sk_{b_1} and sk_{b_2}, respectively. A threshold signature scheme can be used to construct signatures from these shares[40].

iii. Alice1 gets a certificate for her public verification key pk_a, and Bob1 gets a certificate for his public verification key pk_b.

The Protocol: (cf. Figure 6.)

1. (a) Alice1 chooses random $x \in \mathbb{Z}_p$.
   (b) Alice2 computes her public DH key: g^x.
2. Alice1 sends g^x to Bob1.
3. (a) Bob1 chooses random $y \in \mathbb{Z}_p$.
   (b) Bob1 computes his public DH key: g^y.
   (c) Bob1 computes the shared DH key: $K_{DH} = (g^x)^y = g^{xy}$.
4. Bob1 sends the public DH keys, (g^y, g^x) to Bob2.
5. Bob2 signs the DH keys with his secret key sk_{b_2}: $\sigma_{b_2} = Sig_{sk_{b_2}}(g^y, g^x)$.
6. Bob2 sends the signature σ_{b_2} to Bob1.
7. (a) Bob1 also signs the DH keys with his secret key sk_{b_1} with the contribution of Bob2 to get the signature to send to Alice: $\sigma_b = Sig_{sk_{b_1}}(\sigma_{b_2}, g^y, g^x)$.
   (b) Bob applies AEAD on his signature and certificate: $enc_1 = AEAD_{K_{DH}}(\sigma_b, cert_b)$.
8. Bob1 sends g^y and enc_1 to Alice1.
9. (a) Alice1 computes the shared DH key: $K_{DH} = (g^y)^x = g^{xy}$.
   (b) Alice1 then decrypts the enc_1 by using the derived key to get the signature and the certificate of Bob: $(\sigma_b, cert_b) = DEC_{K_{DH}}(enc_1)$.
   (c) Alice1 verifies the certificate, i.e., $cert_b \neq cert_a$ and the signature σ_b.
10. Alice1 sends the public DH keys, (g^x, g^y) to Alice2.
11. Alice2 signs the DH keys with her secret key sk_{a_2}: $\sigma_{a_2} = Sig_{sk_{a_2}}(g^x, g^y)$.
12. Alice2 sends the signature σ_{a_2} to Alice1.
13. (a) Alice1 also signs the DH keys with her secret key sk_{a_1} with the contribution of Alice2 to get the signature to send to Bob: $\sigma_a = Sig_{sk_{a_1}}(\sigma_{a_2}, g^x, g^y)$.
    (b) Alice1 applies AEAD on her signature and certificate: $enc_2 = AEAD_{K_{DH}}(\sigma_a, cert_a)$.
14. Alice1 sends σ_a, enc_2, and mac_2 to Bob1.
15. (a) Bob decrypts the enc_2 to get the signature and the certificate of Alice: $(\sigma_a, cert_a) = DEC_{K_{DH}}(enc_2)$.
    (b) Bob verifies the certificate, i.e., $cert_a \neq cert_b$ and the signature σ_a.
16. Alice and Bob derive the session key K from the shared DH key and their certificates: $K = KDF(K_{DH}, cert_a, cert_b)$.

5.2 Split-key settings for station-to-station protocol

In this section, we adopt the privacy-enhanced variants of STS-KDF-CB to the setting in case (iv) of Figure 2, where both of the parties of the protocol have split keys. By splitting the keys, we aim to protect the security of the key if the device is compromised since compromising two or more devices is harder than a single device.

In these split-key protocols, neither Alice nor Bob is aware that the other party has a split key. The protocols

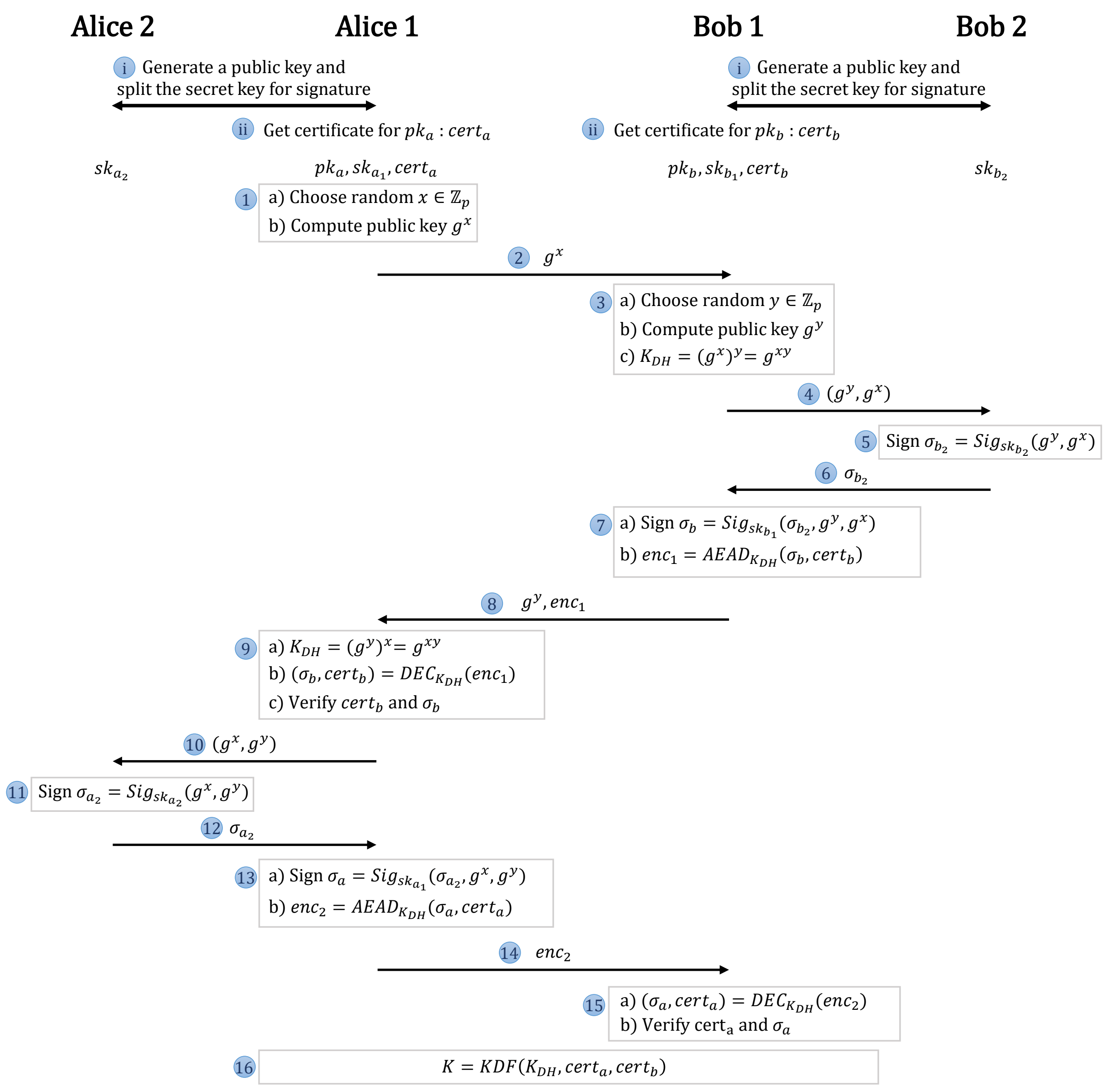

Alice 2
Alice 1
Bob 1
Bob 2
i Generate a public key and split the secret key for signature
ii Get certificate for pk_a : $cert_a$
ii Get certificate for pk_b : $cert_b$
sk_{a_2}
$pk_a, sk_{a_1}, cert_a$
$pk_b, sk_{b_1}, cert_b$
sk_{b_2}
1 a) Choose random $x \in \mathbb{Z}_p$ b) Compute public key g^x
2 g^x
3 a) Choose random $y \in \mathbb{Z}_p$ b) Compute public key g^y c) $K_{DH} = (g^x)^y = g^{xy}$
4 (g^y, g^x)
5 Sign $\sigma_{b_2} = Sig_{sk_{b_2}}(g^y, g^x)$
6 σ_{b_2}
7 a) Sign $\sigma_b = Sig_{sk_{b_1}}(\sigma_{b_2}, g^y, g^x)$ b) $enc_1 = AEAD_{K_{DH}}(\sigma_b, cert_b)$
8 g^y, enc_1
9 a) $K_{DH} = (g^y)^x = g^{xy}$ b) $(\sigma_b, cert_b) = DEC_{K_{DH}}(enc_1)$ c) Verify $cert_b$ and σ_b
10 (g^x, g^y)
11 Sign $\sigma_{a_2} = Sig_{sk_{a_2}}(g^x, g^y)$
12 σ_{a_2}
13 a) Sign $\sigma_a = Sig_{sk_{a_1}}(\sigma_{a_2}, g^x, g^y)$ b) $enc_2 = AEAD_{K_{DH}}(\sigma_a, cert_a)$
14 enc_2
15 a) $(\sigma_a, cert_a) = DEC_{K_{DH}}(enc_2)$ b) Verify $cert_a$ and σ_a
16 $K = KDF(K_{DH}, cert_a, cert_b)$

Figure 6. Split-Key Privacy-Enhanced STS-KDF-CB protocol.

for the hybrid cases (ii) or (iii) in Figure 2 can be constructed by combining a split-key protocol from this section with the corresponding protocol from Section 5.1 above.

5.2.1 Split-key privacy-enhanced STS-KDF-CB protocol

Next, we present the split-key variant of the Privacy-Enhanced STS-KDF-CB Protocol.

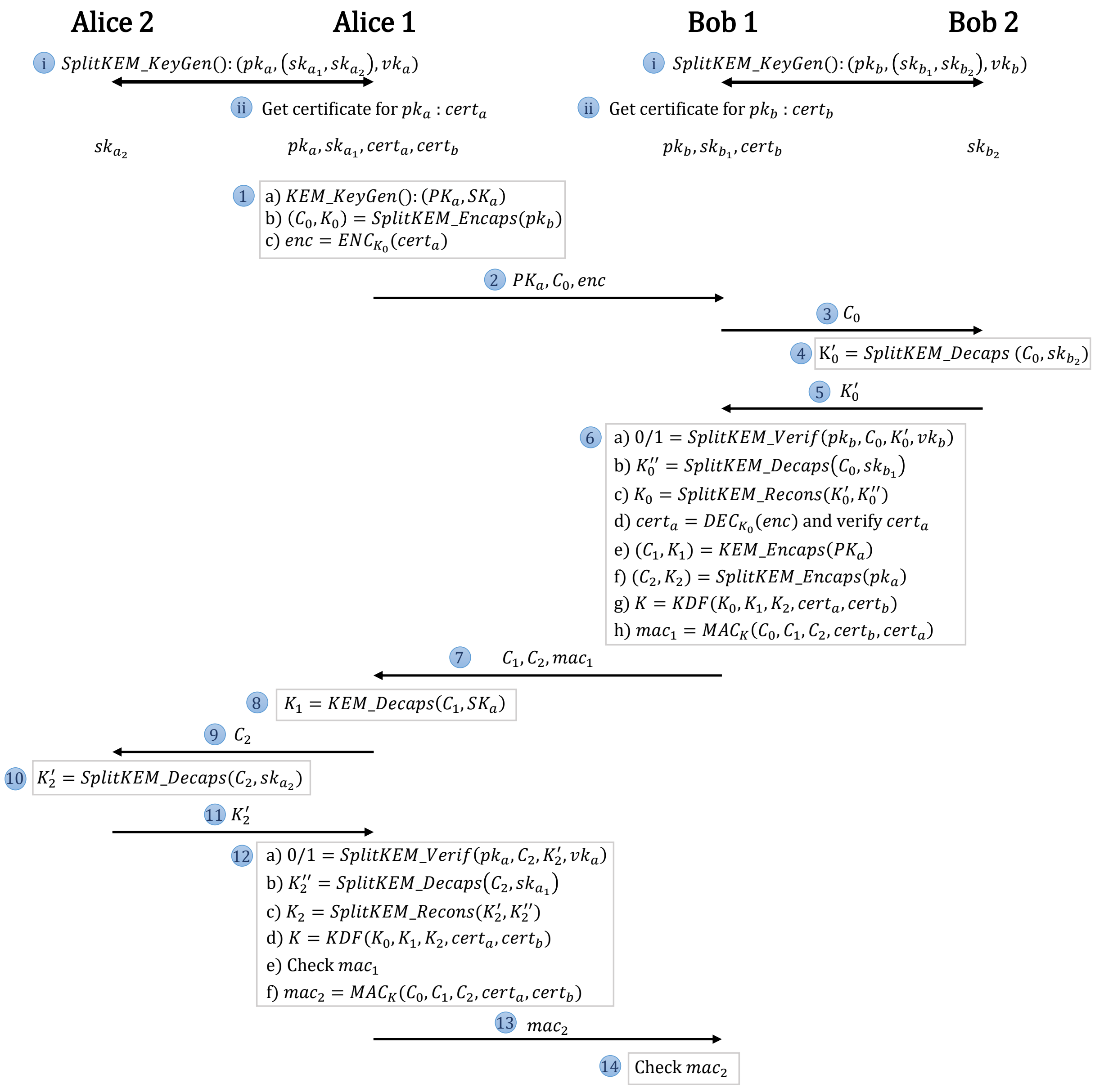

Alice 2
Alice 1
Bob 1
Bob 2
i $SplitKEM\_KeyGen(): (pk_a, (sk_{a_1}, sk_{a_2}), vk_a)$
i $SplitKEM\_KeyGen(): (pk_b, (sk_{b_1}, sk_{b_2}), vk_b)$
ii Get certificate for $pk_a : cert_a$
ii Get certificate for $pk_b : cert_b$
sk_{a_2}
$pk_a, sk_{a_1}, cert_a, cert_b$
$pk_b, sk_{b_1}, cert_b$
sk_{b_2}
1 a) $KEM\_KeyGen(): (PK_a, SK_a)$ b) $(C_0, K_0) = SplitKEM\_Encaps(pk_b)$ c) $enc = ENC_{K_0}(cert_a)$
2 PK_a, C_0, enc
3 C_0
4 $K_0' = SplitKEM\_Decaps\ (C_0, sk_{b_2})$
5 K_0'
6 a) $0/1 = SplitKEM\_Verif(pk_b, C_0, K_0', vk_b)$ b) $K_0'' = SplitKEM\_Decaps(C_0, sk_{b_1})$ c) $K_0 = SplitKEM\_Recons(K_0', K_0'')$ d) $cert_a = DEC_{K_0}(enc)$ and verify $cert_a$ e) $(C_1, K_1) = KEM\_Encaps(PK_a)$ f) $(C_2, K_2) = SplitKEM\_Encaps(pk_a)$ g) $K = KDF(K_0, K_1, K_2, cert_a, cert_b)$ h) $mac_1 = MAC_K(C_0, C_1, C_2, cert_b, cert_a)$
7 C_1, C_2, mac_1
8 $K_1 = KEM\_Decaps(C_1, SK_a)$
9 C_2
10 $K_2' = SplitKEM\_Decaps(C_2, sk_{a_2})$
11 K_2'
12 a) $0/1 = SplitKEM\_Verif(pk_a, C_2, K_2', vk_a)$ b) $K_2'' = SplitKEM\_Decaps(C_2, sk_{a_1})$ c) $K_2 = SplitKEM\_Recons(K_2', K_2'')$ d) $K = KDF(K_0, K_1, K_2, cert_a, cert_b)$ e) Check mac_1 f) $mac_2 = MAC_K(C_0, C_1, C_2, cert_a, cert_b)$
13 mac_2
14 Check mac_2

Figure 7. Split-Key Privacy-Enhanced STS-KDF-CB Protocol with KEM.

5.2.2 *Split-key privacy-enhanced STS-KDF-CB with KEM*

Next, we present how we adopt the split-Key KEM construction (see Section 2.5) to the privacy-enhanced STS-KDF-CB with KEM.

Protocol 5 : Split-Key Privacy-Enhanced STS-KDF-CB with KEM

Setup:

i. Alice1 and Alice2 apply $SplitKEM\_KeyGen()$ to generate a public key pk_a, split secret keys, sk_{a_1} and sk_{a_2}, respectively, and a verification key vk_a. Similarly, Bob1 and Bob2 apply $SplitKEM\_KeyGen()$ to generate a public key pk_b and split secret keys, sk_{b_1} and sk_{b_2}, respectively, and a verification key vk_a.

ii. Alice1 gets a certificate for her public key pk_a, and Bob1 gets a certificate for his public key pk_b.

The Protocol: (cf. Figure 7.)

1. (a) Alice1 generates key pair (PK_a, SK_a) with $KEM\_KeyGen()$ function. Note that the newly generated PK_a and SK_a are not the same as certified pk_a and sk_a.
   (b) Alice1 has the certificate of Bob before starting the protocol, so she knows his public key pk_b. By using this public key, Alice1 applies the split encapsulation function and gets a ciphertext and a session key: $(C_0, K_0) = SplitKEM\_Encaps(pk_b)$.
   (c) Alice1 then encrypts her certificate by using the session key derived above: $enc = ENC_{K_0}(cert_a)$.
2. Alice1 sends PK_a, C_0, and enc to Bob1.
3. Bob1 sends C_0 to Bob2.
4. Bob2 applies the split decapsulation function on the ciphertext C_0 and his share of the secret key sk_{b_2} to get part of the session key: $K_0' = SplitKEM\_Decaps(C_0, sk_{b_2})$.
5. Bob2 sends the key part K_0' to Bob1.
6. (a) Bob1 first verifies that the key part he received from Bob2 is derived by using the correct secret key: $SplitKEM\_Verif(pk_b, C_0, K_0', vk_b) = True/False$.
   (b) Bob1 then applies the split decapsulation function on the ciphertext C_0 and his share of the secret key sk_{b_1}: $K_0'' = SplitKEM\_Decaps(C_0, sk_{b_1})$.
   (c) Bob1 applies the split reconstruction function on the key parts K_0' and K_0'': $K_0 = SplitKEM\_Recons(K_0', K_0'')$.
   (d) Bob1 decrypts enc to get the certificate of Alice: $cert_a = DEC_{K_0}(enc)$. Bob also verifies that the certificate of Alice is not the same as his own certificate, i.e., $cert_a \neq cert_b$.
   (e) Bob1 applies the encapsulation function on the public key PK_a to get another session key and the ciphertext: $(C_1, K_1) = KEM\_Encaps(PK_a)$.
   (f) Bob1 also applies the split encapsulation function on the certified public key pk_a to get one more session key and the ciphertext: $(C_2, K_2) = SplitKEM\_Encaps(pk_a)$.
   (g) Bob1 then derives the shared session key from the derived keys above and the certificates: $K = KDF(K_0, K_1, K_2, cert_a, cert_b)$.
   (h) Bob1 computes the MAC of the ciphertexts and the certificates using the derived key K: $mac_1 = MAC_K(C_0, C_1, C_2, cert_b, cert_a)$.
7. Bob1 sends the ciphertexts C_1, C_2, and mac_1 to Alice1.
8. Alice1 applies the decapsulation function on the ciphertext C_1 and her secret key SK_a to get the session key: $K_1 = SplitKEM\_Decaps(C_1, SK_a)$.
9. Alice1 sends C_2 to Alice2.
10. Alice2 applies the split decapsulation function on the ciphertext C_2 and her share of the secret key sk_{a_2} to get part of the session key: $K_2' = SplitKEM\_Decaps(C_2, sk_{a_2})$.
11. Alice2 sends the key part K_2' to Alice1.
12. (a) Alice1 first verifies that the key part she received from Alice2 is derived by using the correct secret key: $SplitKEM\_Verif(pk_a, C_2, K_2', vk_a) = True/False$.
    (b) Alice1 then applies the split decapsulation function on the ciphertext C_2 and her share of the secret key sk_{a_1}: $K_2'' = SplitKEM\_Decaps(C_2, sk_{a_1})$.
    (c) Alice1 applies the split reconstruction function on the key parts K_2' and K_2'': $K_2 = SplitKEM\_Recons(K_2', K_2'')$.
    (d) Alice1 derives the shared session key from the derived keys above and the certificates: $K = KDF(K_0, K_1, K_2, cert_a, cert_b)$.
    (e) Alice1 verifies the mac_1.
    (f) Alice1 computes the MAC of the ciphertexts and her certificate with the certificate of Bob using the shared session key K: $mac_2 = MAC_K(C_0, C_1, C_2, cert_a, cert_b)$.
13. Alice1 sends mac_2 to Bob1.
14. Bob1 verifies the mac_2.

6 FORMAL VERIFICATION

STS-KDF was formally verified by Jackson *et al.*[6] using the Tamarin Prover. They have proved that this protocol satisfies key secrecy, identity agreement, and strong session agreement (injective agreement) while using the EUF-CMA signature.

We have used the ProVerif tool[13] to model STS-KDF and the protocols that we have presented in Section 5 and prove their security properties. In addition, we have modeled the possibility of sharing the same private key between different parties in order to capture the reflection attack, introduced in Section 4, for STS-KDF. Then, we have shown that the Privacy-Enhanced STS-KDF-CB protocols are not vulnerable to this reflection attack.

We will now explain how we constructed the formal models of the protocols and their security properties and present the verification results. The source codes of the formal verifications of the protocols are publicly available on GitHub[41].

6.1 Modeling the protocols

The ProVerif model is constructed in three parts: declarations, process macros, and main process.

The cryptographic primitives are formalized using *declarations*, which are a finite set of types, free names, and constructors (functions) [13]. We have used the following primitives, for which examples of constructions are given in the ProVerif manual[13]: symmetric encryption, MAC, DH Key Exchange, digital signature, and AEAD. In addition, we have modeled the following primitives, some of which have been used in our earlier work.

Split-Key Digital Signatures: In addition to the digital signatures, we have added a function for reconstructing the split secret keys. We then defined split signing functions for the parties in the protocol who want to collectively create digital signatures:

```
fun splitKey(sskey,sskey):sskey. (* sk_a1 and sk_a2 => sk_a *)
fun splitSign2(G,G,sskey):bitstring. (*Alice2 signs*)
fun splitSign1(G,G,bitstring,sskey):bitstring. (*Alice1 signs*)
```

The equation below defines that the result of the reconstruction of split signatures should be the same as the result of the message signed with the reconstructed key. This equation is important, especially in scenarios where one party splits her key while the other party does not, as described in cases (ii) and (iii) of the split key scenarios in Section 3.1.

```
equation forall a:G,b:G,k1:sskey,k2:sskey;
splitSign1(a,b,splitSign2(a,b,k2),k1)=sign(a,b,splitKey(k1,k2)).
```

Key Encapsulation Mechanism (KEM): The KEM is modeled using the following five functions: key generation, encapsulation, KEM key derivation, ciphertext generation, and decapsulation of the KEM key. It is important to recall that the encapsulation algorithm is a probabilistic algorithm with two outputs, the ciphertext and the KEM key; thus, we model the two outputs with two separate functions: KEMkey and KEMCipher, taking randomized public encryption as input, i.e., output of Encaps function:

```
fun pk(skey):pkey.
fun Encaps(bitstring,pkey):bitstring.
fun KEMkey(bitstring):key.
fun KEMCipher(bitstring):bitstring.
fun DecapsKey(bitstring,skey):key.
```

The equation below ensures that the key generated during encapsulation (`KEMkey` function) is the same as the key generated by decapsulation (`DecapsKey` function).

```
    equation forall sk:skey, r:bitstring;
    DecapsKey(KEMCipher(Encaps(r,pk(sk))),sk)=KEMkey(Encaps(r,pk(sk))).
```

Split Key Encapsulation Mechanism (Split KEM): In addition to the functions that are defined for KEM, we added functions for split key generation, key share verification, and reconstruction of the key shares, which are the outputs of split decapsulation:

```
 fun splitpk(skey,skey):pkey.
 fun vk(pkey,skey,skey):vkey.
 fun SplitEncaps(bitstring,pkey):bitstring.
 fun SplitKEMkey(bitstring):key.
 fun SplitKEMCipher(bitstring):bitstring.
 fun SplitDecaps(bitstring,skey):kshare.
 fun ReconstKey(kshare,kshare):key.
```

The reduction below ensures the verification of valid splits. The equation, in the end, ensures that the key generated during encapsulation (`SplitKEMkey` function) is the same as the key generated after reconstructing the key shares that are the outputs of decapsulation (`ReconstKey` function).

```
reduc forall sk1:skey, sk2:skey, r:bitstring;
SplitVerif(splitpk(sk1,sk2),
SplitKEMCipher(SplitEncaps(r,splitpk(sk1,sk2))),
SplitDecaps(SplitKEMCipher(SplitEncaps(r,splitpk(sk1,sk2))),sk2),
vk(splitpk(sk1,sk2),sk1,sk2))= true.

equation forall sk1:skey, sk2:skey, r:bitstring;
ReconstKey(SplitDecaps(SplitKEMCipher
(SplitEncaps(r,splitpk(sk1,sk2))),sk1),
SplitDecaps(SplitKEMCipher(SplitEncaps(r,splitpk(sk1,sk2))),sk2))
=SplitKEMkey(SplitEncaps(r,splitpk(sk1,sk2))).
```

The process macro consists of sub-processes such that different sub-processes are defined for different parties in the protocol in order to ease the development: Privacy-Enhanced STS-KDF-CB Protocol (Section 5.1.1) and Privacy-Enhanced STS-KDF-CB with KEM Protocol (Section 5.1.2), have two parties, say Alice and Bob. Split-Key Privacy-Enhanced STS-KDF-CB Protocol (Section 5.2.1) and Split-Key Privacy-Enhanced STS-KDF-CB with KEM Protocol (Section 5.2.2) have two additional parties, say Alice2 and Bob2. In ProVerif models, we created a sub-process for each party.

Finally, the protocol is encoded as a *main process* using the process macros. Next, we explain the main processes of the protocols of Section 5. We model the CA by simply generating secret and public sk_{CA} and pk_{CA} in the main process, where pk_{CA} is output in the public channel that we call `internet`. Then, to construct a certificate for a user with identity, say Alice, holding a pair of public/private keys $(pk_1, sk1)$, we use the function `makecert(Alice,`pk_1`,`sk_1`).`

Protocol 1: STS-KDF: The main process creates secret-public signing and verification keys for CA, Alice, and Bob.

```
new skCA:sskey; let pkCA=spk(skCA) in out(internet,pkCA);
new Alice:host; new sk1:sskey; let pk1=spk(sk1) in
let cert1=makecert(Alice,pk1,skCA) in out(internet,cert1);
new Bob:host; new sk2:sskey; let pk2=spk(sk2) in
let cert2=makecert(Bob,pk2,skCA) in out(internet,cert2);

    !processAlice(Alice,cert1,sk1,pkCA) | !processBob(Bob,cert2,sk2,pkCA)
```

Protocol 2: Privacy-Enhanced STS-KDF-CB: The main process of Protocol 2 has the same main process as Protocol 1.

Protocol 3: Privacy-Enhanced STS-KDF-CB with KEM: The main process of Protocol 2 differs from Protocol 1 only when creating secret and public keys: `new sk1:skey; let pk1=pk(sk1)` for Alice and `new sk2:skey; let pk2=pk(sk2)` for Bob. Note that, in this protocol, keys for asymmetric encryption are derived instead of signature keys.

Protocol 4: Split-Key Privacy-Enhanced STS-KDF-CB: This protocol has two more parties participating, and the main process is prepared accordingly:

```
new skCA:sskey; let pkCA=spk(skCA) in out(internet,(pkCA));
new Alice1:host; new sk1:sskey;
new Alice2:host; new sk2:sskey;
let cert1=makecert(Alice1,spk(splitKey(sk1,sk2)),skCA)

in out(internet,(cert1));
new Bob1:host; new sk3:sskey;
new Bob2:host; new sk4:sskey;
let cert3=makecert(id3,spk(splitKey(sk3,sk4)),skCA)

in out(internet,(cert3));
!processAlice(Alice1,cert1,sk1,pkCA) | !processAlice2(Alice2,sk2)
| !processBob(Bob1,cert3,sk3,pkCA)| !processBob2(Bob2,sk4)
```

Protocol 5: Split-Key Privacy-Enhanced STS-KDF-CB with KEM: The setup of the main process of Protocol 5 is similar to Protocol 4, except for generating split keys for asymmetric encryption. In addition, in this protocol, a verification key for Split-KEM is generated for Alice1 and Bob1: `let vk1=vk(pk1,sk1,sk2) in` and `let vk3=vk(pk3,sk3,sk4) in`, respectively.

6.2 Queries

In ProVerif, `events` are defined while modeling the protocol to record incidents, e.g., key derivation and signature verification. These events are used to check if the protocol runs without errors by checking the reachability of events and construct correspondence assertions by capturing the relationships between events[13].

A query is a statement about a security property that is wanted to verify. It is used to express a specific security property or behavior that is expected by a protocol or system to satisfy.

Next, we list the queries that we have constructed to verify the security properties. Note that the protocols with KEM do not have queries for DH key K_{DH}.

Secrecy: We want to prove the secrecy of the key that is derived during the protocol run. Therefore, we constructed the following queries:

```
(*Secrecy of K*)
query a:host,b:host,k:key; event(acceptKeyAlice(a,b,k))&&attacker(k)
==> false.
query a:host,b:host,k:key; event(acceptKeyBob(a,b,k))&&attacker(k)
==> false.
```

Mutual Authentication: In ProVerif, the correspondence is defined for two events, `e2` and `e1`, such that the query for `event(e2(M))==>event(e1(N))` means if an event `e2(M)` has been executed, then event `e1(N)` has been previously executed. The in-built functionality, injective correspondence, of ProVerif additionally checks if the number of occurrences of `e1(N)` is greater than or equal to the number of occurrences of `e2(M)`, in addition to the correspondence queries[13]. Mutual authentication can be proved by proving injective agreement based on the shared keys:

```
(*Mutual Authentication of K_DH*)
query a:host,b:host,k:G;
inj-event(verifB_DH(a,b,k)) ==> inj-event(hasKeyAlice_DH(a,k)).
query a:host,b:host,k:G;
inj-event(verifA_DH(a,b,k)) ==> inj-event(hasKeyBob_DH(b,k)).

(*Mutual Authentication of K*)
query a:host,b:host,k:key;
inj-event(acceptKeyBob(a,b,k)) ==> inj-event(acceptKeyAlice(a,b,k)).
query a:host,b:host,k:key;
inj-event(acceptKeyAlice(a,b,k)) ==> inj-event(acceptKeyBob(a,b,k)).
```

Forward Secrecy: To prove forward secrecy we use *phases*[13]. We leak the long-term secret keys after the initial run of the protocol (phase 1) and show the secrecy property of the session that took place before the leak `(phase 0)`. For this end, we use: `phase 1; out(internet,(sk1,sk2,skCA))`.

Resistance to Unknown Key Share (UKS) attack: In order to construct queries for proving the resistance to UKS attacks, we use the events that are used to prove mutual authentication so that if these events occur for the same key, then the identities of the parties should be same. We check the UKS attack resistance for the keys K_{DH} and K, and identities $Alice$ and Bob.

```
(*Unknown Key Share (UKS) attack K_DH*)
query a:host,b:host,c:host,d:host,k:G;
event(verifB_DH(a,b,k)) && event(hasKeyAlice_DH(c,k)) ==> a=c.
query a:host,b:host,c:host,d:host,k:G;
event(verifA_DH(a,b,k)) && event(hasKeyBob_DH(c,k)) ==> b=c.

(*Unknown Key Share (UKS) attack K*)
query a:host,b:host,c:host,d:host,k:key;
event(acceptKeyBob(a,b,k)) && event(acceptKeyAlice(c,d,k))
==> a=c && b=d.
query a:host,b:host,c:host,d:host,k:key;
```

Table 1. Summary of ProVerif results. Protocol 1: STS-KDF, Protocol 2: Privacy-Enhanced STS-KDF-CB, Protocol 3: Privacy-Enhanced STS-KDF-CB with KEM, Protocol 4: Split-Key Privacy-Enhanced STS-KDF-CB, and Protocol 5: Split-Key Privacy-Enhanced STS-KDF-CB with KEM. N/A: not applicable. true*: true if only the key of the assistant device is leaked

Security Properties	Protocol 1	Protocol 2	Protocol 3	Protocol 4	Protocol 5
Secrecy of K	true	true	true	true	true
Mutual Authentication on K_{DH}	true	true	N/A	true	N/A
Mutual Authentication on K	false	false	true	false	true
Forward Secrecy	true	true	true	true	true
Resistance to UKS Attack on K_{DH}	true	true	N/A	true	N/A
Resistance to UKS Attack on K	true	true	true	true	true
Resistance to KCI attack	false	false	false	true*	true*
Resistance to the Reflection Attack	false	true	true	true	true

```
event(acceptKeyAlice(a,b,k)) && event(acceptKeyBob(c,d,k))
==> a=c && b=d.
```

Resistance to Key Compromise Impersonation (KCI) attack: We want to verify that the protocol is resistant to KCI attacks by leaking the secret keys in the main process. Consequently, if the secrecy and correspondence queries remain true, then we conclude that the protocol is resistant to KCI attacks:

```
new Alice:host; new sk1:sskey; let pk1=spk(sk1) in
let cert1=makecert(Alice,pk1,skCA) in out(internet,(cert1,sk1));
new Bob:host; new sk2:sskey; let pk2=spk(sk2) in
let cert2=makecert(Bob,pk2,skCA) in out(internet,(cert2,sk2));
```

When KCI resistance is checked, these lines should be replaced with the corresponding lines in the main process.

Identity Confidentiality: The identity confidentiality of the users against outsiders can be verified by the notion of observational equivalence, see[13] Section 4.3.2. When we check the observational equivalence of Bob, we ask ProVerif to choose an identity for Bob while we assign a new identity to Alice. Note that `Bob1` and `Bob2` are public names. These are the identity assignments for Alice and Bob to check observational equivalence of Bob:

```
new Alice:host;
let Bob=choice[Bob1,Bob2] in
```

Note that the certificates, and, therefore, the identities, are already public. While creating certificates, we send the certificates that are made for Alice, Bob1, and Bob2 to the public network. However, we also create another certificate for chosen Bob, which should be the same as the certificate of either Bob1 or Bob2, and we send this certificate to Bob via a private channel. In addition, in our protocols, the certificates are sent in an encrypted form.

6.3 Formal verification results

After introducing how to construct the model for formally verifying the protocols, we now explain the results of the formal verification. We will start with the security properties of the STS-KDF protocol (Protocol 1) and then continue with the security properties of the other protocols when they differ from those of the STS-KDF protocol. A summary of the formal verification results is given in Table 1 and Table 2.

The identity confidentiality results vary depending on users. Table 2 presents the ProVerif results for identity confidentiality of user identities.

Table 2. Summary of ProVerif results for identity confidentiality of user identities. true: true with implicit rejection**

Protocols	Alice	Bob
Protocol 1: STS-KDF	false	false
Protocol 2: Privacy-Enhanced STS-KDF-CB	true	false
Protocol 3: Privacy-Enhanced STS-KDF-CB with KEM	true	true**
Protocol 4: Split-Key Privacy-Enhanced STS-KDF-CB	true	false
Protocol 5: Split-Key Privacy-Enhanced STS-KDF-CB with KEM	true	true**

The following is the explanation of security properties that we get from ProVerif results.

Secrecy of K: The attacker cannot capture the shared key K.

```
Query not (event(acceptKeyAlice(a_1,b_1,k)) && attacker_p1(k)) is true.
Query not (event(acceptKeyBob(a_1,b_1,k)) && attacker_p1(k)) is true.
```

Mutual authentication on K_{DH}: The injective correspondences on K_{DH} are true. This means that the mutual authentication between Alice and Bob is achieved for K_{DH}.

```
Query inj-event(verifBmac(a_1,b_1,k)) ==> inj-event(Amac(a_1,b_1,k))
is true.
Query inj-event(verifAmac(a_1,b_1,k)) ==> inj-event(Bmac(b_1,k))
is true.
```

Mutual authentication on K: The injective correspondences on K are false, which means that there is no mutual authentication between Alice and Bob for the key K. This is because the parties do not perform key confirmation on K. In this protocol, the master key is K_{DH}, and this key is confirmed in the protocol. In addition, the identities of the parties are also confirmed through the certificates. STS-KDF is a standalone protocol, and K will be confirmed indirectly later when it is used.

```
Query event(acceptKeyBob(a_1,b_1,k))
==> event(acceptKeyAlice(a_1,b_1,k)) is false.
Query event(acceptKeyAlice(a_1,b_1,k))
==> event(acceptKeyBob(a_1,b_1,k)) is false.
```

Forward Secrecy: The STS-KDF protocol has forward secrecy. The leak of the secret keys of Alice, Bob, and CA does not help the attacker recover the session keys derived during earlier sessions.

Resistance to Unknown Key Share Attack on K_{DH}: STS-KDF protocol is secure against the UKS attacks on K_{DH}.

```
Query event(verifBmac(a_1,b_1,k)) && event(Amac(c,d,k))
==> a_1 = c && b_1 = d is true.
Query event(verifAmac(a_1,b_1,k)) && event(Bmac(c,k))
==> b_1 = c is true.
```

Resistance to Unknown Key Share Attack on K: STS-KDF protocol is secure against the UKS attacks K.

```
Query event(acceptKeyBob(a_1,b_1,k)) && event(acceptKeyAlice(c,d,k))
==> a_1 = c && b_1 = d is true.
Query event(acceptKeyAlice(a_1,b_1,k)) && event(acceptKeyBob(c,d,k))
==> a_1 = c && b_1 = d is true.
```

Resistance to Key Compromise Impersonation (KCI) attack: When the secret keys of Alice and Bob are leaked at the beginning of the protocol, the secrecy of the keys and the mutual authentication between Alice and Bob on K_{DH} both return false. This means that the STS-KDF protocol is vulnerable to KCI attacks.

Identity Confidentiality: The certificates are sent in plaintext in the protocol; therefore, identity confidentiality is trivially not true for STS-KDF.

Next, we highlight the differences in security properties between STS-KDF and the protocols developed in this paper.

Privacy-Enhanced STS-KDF-CB protocols (Protocols 2, 3, 4, and 5) provide identity confidentiality so that the identities of the parties are not captured by outsiders and passive attackers. However, if the attacker starts Protocols 2 and 4 by sending the first message containing his public key to Bob, then Bob derives the shared DH key (K_{DH}) using the attacker's public key. Even though Bob encrypts his certificate with K_{DH}, the attacker also derives the same key. Therefore, the attacker can decrypt the message in Step 4 of Figure 5 and obtain Bob's certificate. On the other hand, Protocols 3 and 5 can overcome this problem.

A limitation of our ProVerif model is that it cannot detect information exposure through error messages. However, the attacker could learn the identity of Bob with this kind of information exposure in Protocols 3 and 5. For example, the attacker replays Message (2) of Protocol 3 in Figure 5. If the ciphertext C_0 is computed with the correct pk_b, then Bob can execute the rest of (3) and send (4). Bob can understand in (7) that either the other party is not Alice or does not have the key. Therefore, the protocol fails at the end. If the ciphertext C_0 is computed with the wrong pk_b, then Bob cannot execute (3.a) and understands that this message is not intended for him. If Bob rejects explicitly by raising an *Error* at this step, the attacker would understand that this user is not the same Bob that the original Message (2) was sent to. The protocol could be enhanced in such a way that Bob rejects implicitly as follows. He sends an invalid ciphertext, which the attacker cannot distinguish from a successful reply to Message (2). The genuine Alice (who in the example sent the original Message (2)) would still understand the rejection because the ciphertext is invalid.

In Privacy-Enhanced STS-KDF-CB protocols, parties check the certificate of the other party so that they do not receive their own certificates. They do this check by executing `if certA<>certB then` line. When we apply observational equivalence, we choose from two identifiers, Bob1 and Bob2, but they keep using the same public-secret key pair. This is why there is no observation equivalence in Protocols 3 and 5. However, if we check `if Alice<>Bob && pkA<>pkB` by restricting that the public keys are also different, we can get observational equivalence.

Privacy-Enhanced STS-KDF-CB with KEM (Protocol 3) and Split-Key Privacy-Enhanced STS-KDF-CB with KEM (Protocol 5) protocols achieve mutual authentication for the key K.

```
Query inj-event(AliceFinished(a,b,k)) ==> inj-event(hasKeyBob(a,b,k))
is true.
Query inj-event(BobFinished(a,b,k)) ==> inj-event(AliceFinished(a,b,k))
is true.
```

Note that the queries related to K_{DH} are non-applicable for Protocols 3 and 5 since the DH key exchange is replaced by KEM; therefore, the key K_{DH} is not derived.

Considering Protocols 2 and 3, when the secret key of Alice is leaked at the beginning of the protocol, the secrecy of Bob's key and injective correspondence of Bob for Alice on K_{DH} return false. Similarly, if Bob's key is leaked, then the secrecy of Alice's key is compromised, and the injective correspondence of Bob for Alice on K_{DH} returns false. This means that if the key of one party is leaked, the secrecy and authentication of the other party is compromised.

Table 3. Computational cost of the protocols from Section 5

Protocols	Scalar Multiplications	Modular Inverses	Symmetric Key Operations
(1) STS-KDF	10	6	12
(2) STS-KDF-CB	14	10	16
(3) STS-KDF-CB-KEM	12	2	19
(4) Split STS-KDF-CB	16	12	18
(5) Split STS-KDF-CB-KEM	14	2	21

In a split-key setting (Protocols 4 and 5), a KCI attack is unsuccessful only if the secret keys of Alice2 and Bob2 are leaked. However, the cases for Alice1 and Bob1 are similar to those of Alice and Bob in Protocols 2 and 3.

6.4 Construction of reflection attack in ProVerif

In Section 4, we have explained that the reflection attack happens in a setting where the parties in the protocol have the same secret key but different identities and, therefore, different certificates. In order to capture the reflection attack in ProVerif, we have modified the protocol such that (i) only one secret key is created and assigned to the parties, (ii) the certificates are created with different identities but the same public key, and (iii) we allow that Alice and Bob can be both initiator and responder in the protocol.

In this setting, we could capture the reflection attack against STS-KDF in ProVerif. When the protocol model is executed, the mutual authentication for the DH key returns false. The traces captured from the query, `query a:host,b:host,k:G; inj-event(verifBmac(a,b,k))==> inj-event(Amac(a,b,k))`, represents the reflection attack.

ProVerif proved that the reflection attack is not successful when a similar setting is applied to Privacy-Enhanced STS-KDF-CB and Privacy-Enhanced STS-KDF-CB with KEM protocols.

7 ANALYSIS

In this section, we analyze the computational cost, communication overhead, and resource requirement of the protocols presented in Section 5. While doing the analysis, we have selected the following cryptosystems, which are currently popular, to improve the readability by presenting concrete results.

- **Diffie-Hellman Key Exchange:** Elliptic Curve Diffie-Hellman (ECDH) with the curve P-256.
- **Digital Signature Scheme:** Elliptic Curve Digital Signature Algorithm (ECDSA) with curve P-256.
- **Public Key Primitive for KEM:** Elliptic Curve Integrated Encryption Scheme (ECIES) with curve P-256.
- **Certificate:** X.509 certificate.
- **Symmetric Key Encryption:** Advanced Encryption Standard (AES-128).
- **Message Authentication Code:** HMAC-SHA-256.
- **Authenticated Encryption:** AES-128-HMAC-SHA-256.

Next, we present the computational cost of the protocols in Table 3. We gathered all the operations under three titles: (i) scalar multiplication for elliptic curves, (ii) modular inverse, and (iii) symmetric key operations, which include symmetric encryption, symmetric decryption, hash, key derivation function, and random number generation.

As we can see from Table 3, the proposed protocols do more computations than STS-KDF (1). This is because, for example, STS-KDF-CB (2) and Split STS-KDF-CB (4) have additional encryption and certificate verification compared to STS-KDF. We also observe that Protocols (3) and (5), where we use KEM instead of DH key exchange and digital signature, are computationally less complex than Protocols (2) and (4). Even though the number of symmetric key operations increased, the total number of computationally heavier scalar multiplication and modular inverse functions decreased.

Table 4. Communication overhead of the messages (in bytes) in the protocols from Section 5. A1: Alice1, A2: Alice2, B1: Bob1, B2: Bob2. The order of parties represents the direction of messages; e.g., A1-B1 means that the message is sent from Alice1 to Bob1

Protocols	A1-B1	B1-B2	B2-B1	B1-A1	A1-A2	A2-A1	A1-B1	Total
(1) STS-KDF	32	-	-	356	-	-	324	712
(2) STS-KDF-CB	32	-	-	356	-	-	324	712
(3) STS-KDF-CB-KEM	292	-	-	96	-	-	32	420
(4) Split STS-KDF-CB	32	64	64	224	64	64	192	968
(5) Split STS-KDF-CB-KEM	292	32	32	96	32	32	32	548

Table 5. Average computation time of STS-KDF-CB-KEM (in ms), referring to Figure 5

	Step 1	Step 3	Step 5	Step 7	Total
Time (ms)	4.82	9.42	3.09	0.19	17.52

Comparing protocols STS-KDF-CB (2) and STS-KDF-CB-KEM (3) with Split STS-KDF-CB (4) and Split STS-KDF-CB-KEM (5), respectively, we see that the additional cost of the split-key variant (4) is two extra signatures, while the additional cost of the split-key variant (5) is two extra decapsulations. In conclusion, the split-key Protocols (4) and (5) are not much more computationally heavier compared to Protocols (2) and (3).

Table 4 presents the communication overhead of the messages sent in the protocols. Please note that split key protocols have more messages.

According to the cryptosystem choices mentioned earlier, we assume that the public and secret keys in elliptic curves have a length of 32 bytes. Therefore, signatures are 64 bytes long. We estimate the certificate length to be 228 bytes; this includes a public key (32 bytes), signature (64 bytes), and other information (132 bytes). Recall that HMAC-SHA-256 creates a 32-byte output. AES-128 creates a ciphertext of the same size as the plaintext. In our case, the plaintext consists of a signature and certificate; therefore, the ciphertext has a length of 292 bytes. The authenticated encryption adds 32 bytes of MAC to the ciphertext. Thus, for example, we get 324 bytes of ciphertext in Protocols (2) and (4) in the message sent from Alice1 to Bob1. The message from Bob1 to Alice1 includes the ciphertext and the public key of Bob. Thus, the length of that message is 356 bytes.

Observe that the total communication overhead of Protocols (1) and (2) are the same. Therefore, we conclude that adding privacy does not increase the communication overhead.

Note that the first message of protocol (3) is larger than Protocols (1) and (2), but the other messages are much smaller. Thus, replacing DH key exchange and digital signature with KEM decreases the total communication overhead by around 40%.

The total communication overhead increases for split-key variant of STS-KDF-CB by around 35%, and for split-key variant STS-KDF-CB with KEM by around 30%.

We have implemented STS-KDF-CB with KEM in Python with Cryptography package version 41.0.3[42]. Measurements were taken with a computer with 3.5 GH dual-core Intel i7 and 16GB RAM. With references to Figure 5 for step numbers, we present the average computation time of the protocol in Table 5.

From Table 5, we can observe that an STS-KDF-CB Protocol with KEM runs in less than 20 ms, ignoring the communication time between the parties. As we can see in Table 3, splitting keys in KEM increases the number of computations slightly. Therefore, we can estimate that the Split STS-KDF-CB Protocol with KEM could take around 30 ms, again ignoring the communication time between the parties. The implementations of other protocols are left for future work.

8 DISCUSSION

The protocols, described in Section 5 and analyzed in Section 7, add several security properties to the basic STS-KDF. The system designer could decide which protocol to choose based on the desired properties. For example, STS-KDF-CB and STS-KDF with KEM protocols provide additional privacy and protection against reflection attack, while split-key protocol variants also provide measures against single-point failures by splitting keys with assistant devices. In the KEM variants of the protocols, there is flexibility in the choice of any type of KEM and Split-KEM.

As verified by ProVerif, the protocols we propose in Section 5 are privacy-enhanced, which is mainly because the certificate and the signature are encrypted. The certificates are encrypted to provide identity confidentiality to the user by concealing the identity against the passive attackers. The signatures are also encrypted because otherwise, the attacker could run a brute-force attack with a list of public keys to figure out who the sender of the message is. If identity confidentiality is not a concern, the encryption in the protocols could be removed.

In addition, an active attacker can learn the identity of Bob in Protocols 2 and 4 if the attacker starts the protocol. This can be prevented in Protocols 3 and 5 due to the assumption that Alice already knows Bob's certificate. However, Information Exposure Through an Error Message could reveal to the attacker that if the decapsulation in Step 3.a of Figure 5 fails, the attacker understands Bob does not own the public key that was used to create C_0. In order to overcome this issue, even though Bob knows that the message is not intended for him, he prepares a message (4) and sends it. Here, Bob can compute the correct C_1 so that if the attacker knows PK_a, he does not understand if Bob is real or if he understood the replay attack. Bob also includes an invalid ciphertext for C_2 and mac_1 in (4) that appears as the real values. This way, the attacker observes that Bob replies with (4) but cannot understand if it is a valid or invalid response. However, this improvement causes an increase in bandwidth, battery consumption, and the possibility of Denial of Service (DoS) attacks.

In our modification to the STS-KDF protocol, we are using certificates as inputs to the key derivation function instead of identities. One reason is that the same identity might occur in several certificates, e.g., for different devices controlled by the same user. Then, two devices with the same identity may still need to authenticate each other and exchange keys. Another reason for using certificates instead of identities is a case where identity is not relevant at all, while the certificate would technically act as a replacement for an identity.

As mentioned in the introduction, splitting the secret key between two or more devices has security benefits, such that those devices must cooperate when using the key. On the other hand, key usage and management are more complicated in the key-split setting. The usage of split keys is illustrated in Protocols 4 and 5 in Section 5. We will next discuss the different options for creating and certificating the long-term key pair pk_a, sk_a and splitting the secret key sk_a.

In the basic setting, where Alice is a single entity, the certification of the long-term key pair happens as follows. Alice generates a key pair pk_a, sk_a and requests a trusted Certification Authority (CA) to certify the public key pk_a. The certificate request typically includes data signed by the secret key sk_a, and the CA verifies that signature using the public key pk_a from the certificate request. This ensures that the sender of the certificate request has the secret key sk_a. The CA also verifies the identity of Alice before issuing the certificate. The manner of identity verification by the CA depends on the use case.

In some scenarios, the key pair generation can be done by the trusted CA instead of by Alice. For example, in a secure factory environment, the key pair and the certificate can be generated by the manufacturer's CA and directly injected into the device. In this scenario, the identity verification step can be omitted.

In the case where there are two entities, Alice1 and Alice2, one of them, say Alice1, generates a key pair pk_a, sk_a, splits the secret key into two shares sk_{a_1} and sk_{a_2} and requests a trusted Certification Authority (CA) to certify the public key pk_a. The signature in the certificate request is done using sk_a. The CA verifies that signature using the public key pk_a from the certificate request. The CA also verifies the identity of Alice1 before issuing the certificate. Alice1 then delivers pk_a, sk_{a_2} to Alice2 via a secure communication channel and deletes sk_a. Similarly to the basic setting, the key pair can be generated by the trusted CA instead of by Alice1. The secret key can be split by the trusted CA or by Alice1.

In another option, the key shares can be generated by Alice1 and Alice2 using a secure multi-party computation. As a result of this protocol Alice1 gets pk_a and sk_{a_1}; and Alice2 gets pk_a and sk_{a_2}. Alice1 and Alice2 cooperate to sign a certificate request for the public key pk_a, and Alice1 sends the request to CA. The CA verifies the signature using the public key pk_a from the certificate request. The CA also verifies the identity of Alice1 before issuing the certificate.

In summary, the key pair pk_a, sk_a can be generated by (i) one, (ii) both of the parties Alice1, Alice2, or (iii) the trusted CA. The splitting of the secret key sk_a can be done by one of the parties or by both of the parties using a multi-party computation protocol. In scenarios where the CA generates the key pair, it can also split the sk_a. A scenario where Alice2 generates the key pair and asks the CA to split sk_a does not seem to make sense, at least not for the expected use cases.

In a non-split-key case, it is critical to revoke $cert_a$ whenever sk_a gets compromised. In this case, Alice needs to obtain new $cert_a$ and sk_a to be functional again. However, this can be problematic, especially in situations where Alice does not support over-the-air updates or key sk_a is used to secure over-the-air updates.

Under certain circumstances in split-key cases, it is secure to continue the usage of original $cert_a$ even when one of the key shares, sk_{a_1} or sk_{a_2}, is compromised. This requires that the full key sk_a is in possession of a Trusted Party, such as CA, or the key is securely reconstructed from key shares. The original key can be re-split into new key shares[11], which are then securely communicated to Alice1 and Alice2.

Recall that the STS protocol is an AKA based on the DH key exchange protocol and authenticated signatures. Various modifications of the STS protocol were introduced to improve its security. The study by Jackson *et al.*[6] extensively analyzes the STS variants and concludes that STS-KDF is the only variant that satisfies key secrecy, identity agreement, and strong session agreement (injective agreement). The same study shows that STS-KDF is secure against UKS attacks and provides mutual authentication of shared DH key. We formally verified that STS-KDF has perfect forward secrecy. The protocols that we introduce in Section 5 mostly provide resistance to the KCI attack and mutual authentication of shared key K. In addition, the new variants we introduce provide resistance against the reflection attack that we describe in Section 4. Moreover, identity confidentiality is improved in our protocols compared to STS-KDF. The summary of the security properties of the protocols is presented in Table 1 and Table 2.

Shah *et al.*[21] and Wang *et al.*[22] use the secret-splitting concept for multi-factor authentication where they split, e.g., biometric data. The solutions presented in these two papers include servers for keeping one of the shares of the secret. However, our solution is independent of a Trusted Party.

Our STS-KDF-CB Protocol in Section 5.1.1 can be seen as a variant of SIGMA-I protocol[12]. Our protocol differs from the original SIGMA-I in using certificates, identities, and encryption. Our protocol is Certificate-Binding STS-KDF, which means that the shared session key includes the certificates of the participants. Recall that the same identity may be included in several certificates, e.g., with different capabilities. We also apply authenticated encryption on the signature and certificate, while SIGMA-I does not provide authenticated encryption because MAC is only applied to the identity of the user, not to the plaintext or ciphertext. Finally, we introduce the certificate verification upon arrival of the certificate such that the user checks that the received certificate does not belong to himself. By doing this, we aim to prevent replay attacks.

We emphasize that both the original STS and the SIGMA-I protocols use signatures for authentication. In our split and non-split STS-KDF-CB protocols with KEM, we achieve our desired security goals by using KEMs instead of signatures. We argue that KEM operations are typically computationally less costly than signing operations.

Krzywiecki *et al.*[26] present an AKE for wearable devices, such as a watch and a mobile phone, in which the key of a device is split between two signing modules in the same device. This is similar to our Split STS-KDF-CB Protocol in Section 5.2.1, but their setting has several limitations compared to ours. First, in our setting, the initiator and the responder do not have to know the public keys of each other, while Krzywiecki *et al.*[26] assume they do. Second, in our setting, the channel between the devices holding the key shares is public. In Krzywiecki *et al.*[26], the channel between the signing modules is internal to a device and is secure. Also, our protocol does not require human interaction to complete. The setting of Krzywiecki *et al.*[26] includes an out-of-band channel between the initiator and the responder, and human interaction is required to establish the connection at the end of the protocol. Our formal security proof does not make any assumption about the used cryptographic primitives, such as KEMs, signatures, and authenticated encryption. The solution of Krzywiecki *et al.* is based on signatures, and their security proof is specific to the Decisional DH problem and Schnorr split signatures.

Recall that the key management with split-key cases is more complicated since there are more parties involved. One limitation of our key-split setting is the following. If Alice1 is fully compromised, but the user does not notice it, then the key split does not help because Alice2 will cooperate with the attacker who controls Alice1. On the other hand, if the user can detect the compromise, they could turn off the device Alice2. This limitation could be alleviated if the communication between Alice1 and Alice2 requires explicit authorization from the user, e.g., by pressing a button on Alice2. However, we will not discuss this enhancement further in this paper.

Yet, another limitation is related to the KEM-based schemes, as introduced in Section 5.1.2 and Section 5.2.2. In these adaptations, Alice, the initiator, needs to have the certificate of Bob, the responder, before initiating the protocol. If Alice does not already have a certificate of Bob, she should use the privacy-enhanced STS-KDF-CB protocols, as explained in Section 5.1.1 and Section 5.2.1. However, in this case, identity confidentiality for Bob cannot be provided.

We do not discuss protection against side-channel attacks, and it is left for future work. Future work also includes the implementation of all protocols and extensions to the cases where one of the split-key devices is able to attest the correctness of the functionality of the other one.

9 DECLARATIONS

9.1 Authors' contributions

Conceptualization: Akman G, Damir MT, Ginzboorg P, Sovio S, Niemi V
Supervision: Ginzboorg P, Sovio S, Niemi V
Writing - original draft preparation: Akman G
Writing - review and editing:Damir MT, Ginzboorg P, Sovio S, Niemi V
Formal analysis: Akman G, Damir MT
Visualization: Akman G
All authors have read and agreed to the published version of the manuscript.

9.2 Availability of data and materials

The source code of our ProVerif Models is available in a GitHub repository[41].

9.3 Financial support and sponsorship

This research is a result of a funded collaboration project between the University of Helsinki and Huawei Technologies, Finland.

9.4 Conflicts of interest

All authors declared that there are no conflicts of interest.

9.5 Ethical approval and consent to participate

Not applicable.

9.6 Consent for publication

Not applicable.

9.7 Copyright

REFERENCES

1. Wooten D. Trusted platform module library. Trusted Computing Group (TCG); 2019. Available from:https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part1_Architecture_pub.pdf. [Last accessed on 18 Sep 2023]
2. GlobalPlatform. Secure Element Protection Profile. GlobalPlatform Technology; 2021. GPC_SPE_174. Available from: https://globalplatform.org/specs-library/secure-element-protection-profile/. [Last accessed on 18 Sep 2023]
3. Diffie W, Hellman M. New directions in cryptography. *IEEE Trans Inf Theory* 1976;22:644-54. DOI
4. Diffie W, Van Oorschot PC, Wiener MJ. Authentication and authenticated key exchanges. *Des Codes Cryptogr* 1992;2:107-25. DOI
5. Blake-Wilson S, Menezes A. Unknown key-share attacks on the station-to-station STS Protocol. In: Goos G, Hartmanis J, van Leeuwen J, editors. Lecture notes in computer science. Berlin, Heidelberg: Springer; 1999. p. 154-70. DOI
6. Jackson D, Cremers C, Cohn-Gordon K, Sasse R. Seems legit: automated analysis of subtle attacks on protocols that use signatures. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. London: ACM; 2019. p. 2165-80. DOI
7. Cramer R, Shoup V. Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack. *SIAM J Comput* 2003;33:167-226. DOI
8. Dent AW. A designer's guide to KEMs. In: Paterson KG, editor. Lecture notes in computer science. Berlin, Heidelberg: Springer; 2003. p. 133-51. DOI
9. Shoup V. A proposal for an ISO standard for public key encryption (version 2.1). Switzerland: IBM Zurich Research Lab; 2001. Available from: https://shoup.net/papers/iso-2_1.pdf. [Last accessed on 18 Sep 2023]
10. Fujisaki E, Okamoto T. Secure integration of asymmetric and symmetric encryption schemes. *J Cryptol* 2013;26:80-101. DOI
11. Ebina M, Mita J, Shikata J, Watanabe Y. Efficient threshold public key encryption from the computational bilinear Diffie-Hellman assumption. In: Proceedings of the 8th ACM on ASIA Public-Key Cryptography Workshop. Hong Kong: ACM; 2021. p. 23-32. DOI
12. Krawczyk H. SIGMA: the 'SIGn-and-MAc' approach to authenticated Diffie-Hellman and its use in the IKE protocols. In: Boneh D, editor. Lecture notes in computer science. Berlin, Heidelberg: Springer; 2003. p. 400-25. DOI
13. Blanchet B, Smyth B, Cheval V, Sylvestre M. ProVerif 2.04: automatic cryptographic protocol verifier. Available from:https://bblanche.gitlabpages.inria.fr/proverif/manual.pdf. [Last accessed on 18 Sep 2023]
14. van Tilborg HCA, Jajodia S, editors. Encyclopedia of cryptography and security. Boston: Springer; 2011. DOI
15. Needham RM, Schroeder MD. Using encryption for authentication in large networks of computers. *Commun ACM* 1978;21:993-99. DOI
16. Lowe G. Breaking and fixing the Needham-Schroeder Public-Key Protocol using FDR. In: Margaria T, Steffen B, editors. Lecture notes in computer science. Berlin, Heidelberg: Springer; 1996. p. 147-66. DOI
17. Basin D, Cremers C, Dreier J, et al. Tamarin-Prover Manual: Security Protocol Analysis in the Symbolic Model; 2022.
18. Giaretta G, Kempf J, Devarapalli V. RFC 5026. Mobile IPv6 bootstrapping in split scenario. DOI Available from:https://www.rfc-editor.org/info/rfc5026. [Last accessed on 18 Sep 2023]
19. Manzoor A, Shah MA, Khattak HA, Din IU, Khan MK. Multi-tier authentication schemes for fog computing: architecture, security perspective, and challenges. *Int J Commun Syst* 2022;35:e4033. DOI
20. Choi S, Sun K, Eom H. Chapter 17 - resource-efficient multi-source authentication utilizing split-join one-way key chain. In: Babak Akhgar, Hamid R. Arabnia, editors. Emerging trends in ICT security. Elsevier; 2014. p. 267-79. DOI
21. Shah RH, Salapurkar DP. A multifactor authentication system using secret splitting in the perspective of Cloud of Things. In: 2017

International Conference on Emerging Trends & Innovation in ICT (ICEI); 2017 Feb 03-05; Pune, India. IEEE; 2017. p. 1-4. DOI
22. Wang P, Ku CC, Wang TC. A new fingerprint authentication scheme based on secret-splitting for cloud computing security. In: Yang J, Norman Poh, editors. Recent application in biometrics. InTech; 2011. p. 183-96. DOI
23. Choi J, Cho J, Kim H, Hyun S. Towards secure and usable certificate-based authentication system using a secondary device for an industrial internet of things. *Appl Sci* 2020;10:1-16. DOI
24. Menezes AJ, Van Oorschot PC, Vanstone SA. Handbook of applied cryptography. CRC Press series on discrete mathematics and its applications. Boca Raton: CRC Press; 1997. Available from:https://cacr.uwaterloo.ca/hac/. [Last accessed on 18 Sep 2023]
25. Kaufman C, Hoffman P, Nir Y, Eronen P, Kivinen T. Internet key exchange protocol version 2 (IKEv2). RFC Editor; 2014. DOI
26. Krzywiecki L, Salin H. How to design authenticated key exchange for wearable devices: cryptanalysis of AKE for health monitoring and countermeasures via distinct sms with key split and refresh. In: Beresford AR, Patra A, Bellini E, editors. Lecture notes in computer science. Cham: Springer; 2022. p. 225-44. DOI
27. Black J. Authenticated encryption. In: Van Tilborg HCA, Jajodia S, editors. Encyclopedia of cryptography and security. 2nd ed. Boston, MA: Springer; 2011. p. 52-61. DOI
28. Rogaway P. Authenticated-encryption with associated-data. In: Proceedings of the 9th ACM conference on Computer and communications security. Washington: ACM; 2002. p. 98-107. DOI
29. Turan MS, McKay K, Chang D, et al. Status report on the final round of the NIST lightweight cryptography standardization process. 2023. DOI
30. Shamir A. How to share a secret. *Commun ACM* 1979;22:612-13. DOI
31. Ebri NA, Baek J, Yeun CY. Study on Secret Sharing Schemes (SSS) and their applications. In: 2011 International Conference for Internet Technology and Secured Transactions; 2011 Dec 11-14; Abu Dhabi, United Arab Emirates. IEEE; 2011. p. 40-5. Available from:https://ieeexplore.ieee.org/document/6148357. [Last accessed on 18 Sep 2023]
32. Dolev D, Yao A. On the security of public key protocols. *IEEE Trans Inf Theory* 1983;29:198-208. DOI
33. Halpern JY, Pucella R. Modeling adversaries in a logic for security protocol Analysis. In: Goos G, Hartmanis J, Van Leeuwen J, Abdallah AE, Ryan P, et al., editors. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer; 2003. p. 115-32. DOI
34. LaMacchia B, Lauter K, Mityagin A. Stronger security of authenticated key exchange. In: Susilo W, Liu JK, Mu Y, editors. PLecture notes in computer science. Berlin, Heidelberg: Springer; 2007. p. 1-16. DOI
35. W3C Web Authentication Working Group. Web authentication: an API for accessing public key credentials level 3. Available from:https://www.w3.org/TR/webauthn-3/. [Last accessed on 18 Sep 2023]
36. Baghdasaryan D, Balfanz D, Hill B, Hodges J, Yang K. FIDO UAF Protocol Specification. Available from:https://fidoalliance.org/specs/fido-uaf-v1.2-rd-20171128/fido-uaf-protocol-v1.2-rd-20171128.html. [Last accessed on 18 Sep 2023]
37. Android Bootcamp 2016: Android keystore attestation. Available from:https://source.android.com/static/docs/security/overview/reports/Android-Bootcamp-2016-Android-Keystore-Attestation.pdf. [Last accessed on 18 Sep 2023]
38. Powers A. FIDO TechNotes: The truth about attestation. Available from: https://fidoalliance.org/fido-technotes-the-truth-about-attestation/. [Last accessed on 18 Sep 2023]
39. McKay KA, Bassham L, Turan MS, Mouha N. Report on lightweight cryptography. Available from: https://nvlpubs.nist.gov/nistpubs/ir/2017/NIST.IR.8114.pdf. [Last accessed on 18 Sep 2023]
40. Chang TY. Threshold signatures: current status and key issues. *Int J Netw Secur* 2005;1:123-37. Available from: http://ijns.jalaxy.com.tw/download_paper.jsp?PaperID=IJNS-2005-07-28-1&PaperName=ijns-v1-n3/ijns-2005-v1-n3-p123-137.pdf. [Last accessed on 18 Sep 2023]
41. Akman G. ProVerif implementation of STS-KDF and split-key STS-KDF variants. Available from: https://github.com/gizem-akman/SplitKey-STS-KDF. [Last accessed on 18 Sep 2023]
42. Python Software Foundation. Cryptography 41.0.3. Available from: https://pypi.org/project/cryptography/. [Last accessed on 18 Sep 2023]