A Formal Security Analysis of Session Resumption Across Hostnames

Kai Gellert and Tobias Handirk

Bergische Universität Wuppertal, Germany
{kai.gellert, tobias.handirk}@uni-wuppertal.de

Abstract. The TLS 1.3 session resumption handshakes enables a client and a server to resume a previous connection via a shared secret, which was established during a previous session. In practice, this is often done via *session tickets*, where the server provides a "self-encrypted" ticket containing the shared secret to its clients. A client may resume its session by sending the ticket to the server, which allows the server to retrieve the shared secret stored within the ticket.
Usually, a ticket is only accepted by the server that issued the ticket. However, in practice, servers that share the same hostname, often share the same key material for ticket encryption. The concept of a server accepting a ticket, which was issued by a different server, is known as session resumption across hostnames (SRAH). In 2020, Sy *et al.* showed in an empirical analysis that, by using SRAH, the time to load a webpage can be reduced by up to 31% when visiting the page for the very first time. Despite its performance advantages, the TLS 1.3 specification currently discourages the use of SRAH.
In this work, we formally investigate which security guarantees can be achieved when using SRAH. To this end, we provide the first formalization of SRAH and analyze its security in the multi-stage key exchange model (Dowling *et al.*; JoC 2021), which proved useful in previous analyses of TLS handshakes. We find that an adversary can break authentication if clients do not specify the intended receiver of their first protocol message. However, if the intended receiver is specified by the client, we prove that SRAH is secure in the multi-stage key exchange model.

1 Introduction

If two parties want to securely establish a common key over an insecure channel, they typically execute a key exchange protocol. The most used key exchange protocol is the Transport Layer Security (TLS) protocol, whose most recent version, TLS 1.3, was standardized in 2018 [17]. The TLS 1.3 protocol provides two variants of key establishment: (i) the full handshake where two users can establish a fresh key, and (ii) the resumption handshake (also called pre-shared

This is the full version of a paper that appears at ESORICS 2021 – 26th European Symposium on Research in Computer Security. Supported by the German Research Foundation (DFG), project JA 2445/2-1.

key handshake) where two users may derive a new key from preexisting key material (e.g., key material from an earlier session, or key material that was established out-of-band).

One of the main motivations to use the resumption handshake instead of the full handshake, is a reduction of computational complexity. That is, the resumption handshake does not require the expensive verification of signatures to authenticate the server. Instead authentication is provided implicitly by knowledge of the *pre-shared key* (PSK). In fact, according to Cloudflare in 2017[1], 40% of handshakes are users resuming a previous connection, which further illustrates the resumption handshakes form one of the cornerstones of secure connection establishments over the Internet.

Session Resumption Across Hostnames. In practice, a client rarely only establishes one connection to a server, but rather has to open multiple connections in order to retrieve additional data distributed across multiple servers, requiring several *full* handshakes. These additional handshakes often slow down the TLS connection establishment and are desirable to avoid.

An interesting observation is that not all of the additional handshakes request data from external services but some request data from the same content provider, only under a different hostname. For example, a user could request the web page *www.webpage.com* and could be required to load additional content from subdomains such as *assets.webpage.com*. Even though the subdomain may share the certificate of the original domain, a full handshake would need to be executed. Naturally, it would be interesting to investigate whether connection establishment could be accelerated if such handshakes could rely on a resumption-based handshake rather than a full handshake. This approach is called *session resumption across hostnames* (SRAH).

In 2020, Sy *et al.* [18] conducted a study investigating the potential performance improvement when using SRAH. They found that 59% of the (on average) 20 full TLS handshakes required to retrieve a website can be converted into handshakes based on resumption across hostnames. According to them, this would reduce 44% of the computational complexity and accelerate the connection establishment by 31%.

SRAH in TLS 1.3. The typical session resumption can be described as follows: After a client and a server have completed a full TLS 1.3 handshake, both parties derive a PSK which can be used in future resumption handshakes. Typically, the server does not want to keep track of each of those PSKs for each user and encrypts the PSK under a symmetric key only known to the server. This ciphertext is called *ticket* and sent to the client at the end of the original full handshake. Note that this enables the server to delete the PSK as it can always retrieve it from the issued ticket. That is, when the client sends back the ticket to the server, both parties can take the PSK as basis to derive a new session key.

The above approach can be extended to capture the conceptual approach of SRAH as well. To this end, the client would not only send the ticket back

[1] See `https://blog.cloudflare.com/introducing-0-rtt/`.

to the server, but the client would also indicate which hostname it wants to establish a connection with. This indication can be given in form of a Server Name Indication (SNI) value chosen by the client, providing leverage to choose for which server (sharing the same symmetric ticket encryption key) a ticket should be used. The TLS 1.3 standard specifies in Section 4.2.11:

> *"In TLS 1.3, the SNI value is always explicitly specified in the resumption handshake, and there is no need for the server to associate an SNI value with the ticket. Clients, however, SHOULD store the SNI with the PSK to fulfill the requirements of Section 4.6.1."*

We remark that we find the first part of this quote worrying, as the standard does not indicate that the SNI value *must* be set in the resumption handshake. While we agree that the SNI value is not necessary if connections are only resumed with the server that issued a ticket, this formulation also opens room for interpretation how the SNI value should be used when considering SRAH. We are concerned that this ambiguity might lead to wrong conclusions when implementing TLS 1.3.

Furthermore, the standard states in Section 4.6.1:

> *"Clients MUST only resume if the new SNI value is valid for the server certificate presented in the original session and SHOULD only resume if the SNI value matches the one used in the original session.* [...] *Normally, there is no reason to expect that different servers covered by a single certificate would be able to accept each other's tickets;* [...] *If such an indication is provided (externally or by any other means)*[2]*, clients MAY resume with a different SNI value."*

We can observe that the latter half of the above excerpt allows to use SRAH without need to change the standard. However, the standard does not yet elaborate what consequence a resumption across hostnames could have and how those consequences should be dealt with. A potential reason for this might be that the security of SRAH has never been formally analyzed and we currently do not understand its advantages and disadvantages well enough.

Our Contributions. In this work, we formally investigate the security of SRAH. We summarized our contributions as follows:

- We give the first formal definition of secure SRAH, as an abstraction of the construction that can be used in TLS 1.3. This approach enables us to carefully investigate what security for SRAH means and how we can achieve it, without being overwhelmed by the complexity of protocols such as TLS 1.3.

[2] We remark that this indication can either be provided by the server via the `subjectAltName` field in a server's certificate, or via an extension providing this information within the `ClientHello` message of the original full handshake as recommended by Sy *et al.* [18].

– We show security in the most recent version of multi-stage key exchange model [7], which has been proven useful in many analyses of TLS handshakes [3, 6, 8, 11]. Specifically, we show that a misinterpretation of the TLS 1.3 standard leads to an attack on authentication when two servers share the same certificate and symmetric ticket key. Furthermore, we provide an SRAH protocol which constitutes an abstraction of the TLS 1.3 resumption protocol and prove its security.

Related Work. The security of the TLS 1.3 resumption handshake has been analyzed in many previous works. Most notably are the works by Dowling *et al.* [7], and Drucker and Gueron [10] who analyzed the security of the standardized handshake, and Arfaoui *et al.* [1] who analyzed privacy aspects of the resumption handshake. Furthermore, Aviram *et al.* [2, 3] proposed an improvement of the resumption handshake, which achieves forward security for the 0-RTT variant of the handshake. Note, however, that none of the related works consider the case of SRAH.

2 Preliminaries

Notation. We denote the security parameter as λ. For some $n \in \mathbb{N}$ we write $[n] = \{1, \ldots, n\}$, respectively $[n]_0 = \{0, \ldots, n\}$, for the set of integers ranging from 1, respectively 0, to n. For two bit strings s, t let $s \,\|\, t$ be the concatenation of s and t. By $x \xleftarrow{\$} \mathcal{S}$ we indicate sampling x uniformly at random from the set $\mathcal{S}$. We write $y \leftarrow \mathcal{A}(x)$ or $\mathcal{A}(x) \to y$, respectively $y \xleftarrow{\$} \mathcal{A}(x)$ or $\mathcal{A}(x) \xrightarrow{\$} y$, for some algorithm $\mathcal{A}$ that on the input x deterministically, respectively probabilistically, outputs y.

Building Blocks. In this work, we use standard definitions for hash functions and their collision resistance, MACs and their strong unforgeability. We recap the definitions and security of these building blocks in the following.

Definition 1. *A hash function* $\mathsf{H} : \{0,1\}^* \to \{0,1\}^\lambda$ *maps an input* $x \in \{0,1\}^*$ *of arbitrary length to an output* $y \in \{0,1\}^\lambda$ *of fixed length* $\lambda \in \mathbb{N}$. *A hash function* H *provides* collision resistance *if we cannot efficiently construct an efficient adversary* $\mathcal{A}$ *for which the advantage*

$$\mathsf{Adv}^{\mathsf{collision}}_{\mathsf{H},\mathcal{A}} := \Pr[(x, x') \xleftarrow{\$} \mathcal{A}(1^\lambda) : x \neq x' \text{ and } \mathsf{H}(x) = \mathsf{H}(x')]$$

is not a negligible function in λ.

Definition 2. *A* message authentication code MAC *is a tuple of three PPT algorithms* $\mathsf{MAC} = (\mathsf{MAC.Gen}, \mathsf{MAC.Tag}, \mathsf{MAC.Vrfy})$ *with the following properties:*

$\mathsf{MAC.Gen}(1^\lambda)$**.** *On input of a security parameter* λ *the algorithm outputs a key* $k \in \{0,1\}^\lambda$.

$\mathsf{MAC.Tag}(k, m)$**.** *On input of a key* $k \in \{0,1\}^\lambda$ *and a message* $m \in \{0,1\}^*$ *the algorithm outputs a tag* $\tau \in \{0,1\}^*$.

$\mathsf{MAC.Vrfy}(k, m, \tau)$**.** *On input of a key* $k \in \{0,1\}^\lambda$*, a message* $m \in \{0,1\}^*$*, and a tag* $\tau \in \{0,1\}^*$ *the algorithm outputs 1 if the given tag is valid, or 0 otherwise.*

We say $\mathsf{MAC} = (\mathsf{MAC.Gen}, \mathsf{MAC.Tag}, \mathsf{MAC.Vrfy})$ *is correct if for all* $\lambda \in \mathbb{N}$*, all* $k \xleftarrow{\$} \mathsf{MAC.Gen}(1^\lambda)$*, all* $m \in \{0,1\}^*$ *it holds that* $\mathsf{MAC.Vrfy}(k, m, \mathsf{MAC.Tag}(k, m)) = 1$.

We define the security of a MAC with the notion of *strong existential unforgeability under chosen message attacks* (sEUF-CMA). Consider the following game $\mathsf{G}^{\mathsf{sEUF\text{-}CMA}}_{\mathsf{MAC},\mathcal{A}}(\lambda)$ played between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$:

1. $\mathcal{C}$ draws a key $k \xleftarrow{\$} \{0,1\}^\lambda$ uniformly at random and sets $\mathcal{Q} = \emptyset$.
2. $\mathcal{A}$ is given oracle access to $\mathsf{MAC.Tag}(k, \cdot)$. For each oracle query m by $\mathcal{A}$ set $\mathcal{Q} = \mathcal{Q} \cup \{(m, \tau)\}$ where τ is the corresponding oracle response.
3. At some point $\mathcal{A}$ outputs a pair (m, τ).

Definition 3. *We say that* $\mathcal{A}$ *wins the game, denoted by* $\mathsf{G}^{\mathsf{sEUF\text{-}CMA}}_{\mathsf{MAC},\mathcal{A}}(\lambda) = 1$*, if and only if* $(m, \tau) \notin \mathcal{Q}$ *and* $\mathsf{MAC.Vrfy}(k, m, \tau) = 1$*. A message authentication code* MAC *is* strongly existentially unforgeable under adaptive chosen message attacks *if for all PPT adversaries* $\mathcal{A}$ *the advantage*

$$\mathsf{Adv}^{\mathsf{sEUF\text{-}CMA}}_{\mathsf{MAC},\mathcal{A}}(\lambda) := \Pr[\mathsf{G}^{\mathsf{sEUF\text{-}CMA}}_{\mathsf{MAC},\mathcal{A}}(\lambda) = 1]$$

is a negligible function in the security parameter λ*.*

Definition 4. *Let* $f : \{0,1\}^* \times \{0,1\}^{i(\lambda)} \to \{0,1\}^{o(\lambda)}$ *be an efficient function. We call* f pseudorandom *if for all PPT adversaries* $\mathcal{A}$ *the advantage*

$$\mathsf{Adv}^{\mathsf{PRF\text{-}sec}}_{f,\mathcal{A}}(\lambda) := \left|\Pr[\mathcal{A}^{f(k,\cdot)}(1^\lambda) = 1] - \Pr[\mathcal{A}^{g(\cdot)}(1^\lambda) = 1]\right|$$

is negligible in λ*, where* $k \xleftarrow{\$} \{0,1\}^\lambda$*, and* g *is chosen randomly from the set of all functions mapping* $\{0,1\}^{i(\lambda)} \to \{0,1\}^{o(\lambda)}$*. Additionally we say* f *achieves dual PRF security if the advantage*

$$\mathsf{Adv}^{\mathsf{dual\text{-}PRF\text{-}sec}}_{f,\mathcal{A}}(\lambda) := \mathsf{Adv}^{\mathsf{PRF\text{-}sec}}_{f^{\mathsf{swap}},\mathcal{A}}(\lambda)$$

is a negligible function in the security parameter λ*, where* $f^{\mathsf{swap}}(k, l) := f(l, k)$*.*

HMAC-based Key Derivation. The *HMAC-based key derivation function* (HKDF) [14,15] is a key derivation scheme and is used, e.g., in TLS 1.3 [17]. The HKDF scheme is based on the HMAC construction [4,16] and follows the *extract-then-expand* paradigm, i.e., from some given source key material first a pseudorandom key of fixed length is extracted, which is then expanded to a pseudorandom key of the desired length. Formally, $\mathsf{HKDF.Extract}(\mathsf{salt}, \mathsf{src})$ given a (potentially fixed) salt salt and a source key material src outputs a pseudorandom key prk. $\mathsf{HKDF.Expand}(\mathsf{prk}, \mathsf{ctxt})$ given a pseudorandom key prk and a (potentially empty)

context info ctxt outputs a new pseudorandom key.[3] In our security analysis we rely on the assumption that both HKDF.Extract and HKDF.Expand are pseudorandom functions [15].

PRF-ODH Assumption. An important security assumption in the analysis of TLS 1.2 and 1.3 is the PRF-ODH assumption first introduced by Jager *et al.* [13]. Brendel *et al.* [5] analyzed and generalized the PRF-ODH assumption into several variants. In this work we will need the dual-snPRF-ODH assumption.

Definition 5. *Let $\lambda \in \mathbb{N}$, $\mathbb{G}$ be a cyclic group of prime order q with generator g and $\mathsf{PRF} : \{0,1\}^* \times \mathbb{G} \to \{0,1\}^\lambda$ be a pseudorandom function keyed with the second input. Consider the following game $\mathsf{G}^{\mathsf{dual\text{-}snPRF\text{-}ODH}}_{\mathsf{PRF},\mathbb{G},\mathcal{A}}(\lambda)$ played between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$:*

1. *$\mathcal{C}$ samples $b \xleftarrow{\$} \{0,1\}$, $u, v \xleftarrow{\$} \mathbb{Z}_q$ and outputs $\mathbb{G}, g, g^u, g^v$ to $\mathcal{A}$. $\mathcal{A}$ then outputs a challenge label $x^\star$.*
2. *$\mathcal{C}$ computes $y_0^\star = \mathsf{PRF}(x^\star, g^{uv})$ and samples $y_1^\star \xleftarrow{\$} \{0,1\}^\lambda$ uniformly at random and outputs $y_b^\star$ to $\mathcal{A}$.*
3. *$\mathcal{A}$ may query a pair (x, S). If $S \notin \mathbb{G}$ or $(x, S) = (x^\star, g^v)$, $\mathcal{C}$ returns $\perp$. Otherwise $\mathcal{C}$ returns $y \leftarrow \mathsf{PRF}(x, S^u)$.*
4. *At some point $\mathcal{A}$ outputs a guess $b' \in \{0,1\}$.*

We say that $\mathcal{A}$ wins the game $\mathsf{G}^{\mathsf{dual\text{-}snPRF\text{-}ODH}}_{\mathsf{PRF},\mathbb{G},\mathcal{A}}(\lambda) = 1$ if $b = b'$. A pseudorandom function PRF is secure under the dual-snPRF-ODH assumption if for PPT adversaries $\mathcal{A}$ the advantage

$$\mathsf{Adv}^{\mathsf{dual\text{-}snPRF\text{-}ODH}}_{\mathsf{PRF},\mathbb{G},\mathcal{A}}(\lambda) := \Pr[\mathsf{G}^{\mathsf{dual\text{-}snPRF\text{-}ODH}}_{\mathsf{PRF},\mathbb{G},\mathcal{A}}(\lambda) = 1] - \frac{1}{2}$$

is a negligible function in the security parameter λ.

2.1 Multi-Stage Key Exchange

We briefly recap the multi-stage key exchange (MSKE) model in its pre-shared secret variant from [7], which has been used in previous analyses of resumption handshakes in TLS 1.3 [3, 7–9, 12].

The following description of the model is taken verbatim from [7], except for the following minor changes. We modified the NewSecret query such that a pre-shared secret can be shared between more than two users with the restriction that only a single user uses the pre-shared secret in the initiator role. As a consequence we modified the maps storing all pre-shared secrets as well as the input to the NewSession and Corrupt query to properly identify the pre-shared secret to be used in the session, resp. the pre-shared secret to corrupt.

[3] Formally, HKDF.Expand is given an additional input L. This third parameter determines the length of the output pseudorandom key. For simplicity we omit this parameter and assume that $L = \lambda$ unless stated otherwise.

Preliminaries. In the MSKE model, properties are separated into *protocol-specific* and *session-specific* properties. The protocol-specific properties are defined by a vector $(\mathsf{M}, \mathsf{AUTH}, \mathsf{USE}, \mathsf{FS}, \mathsf{REPLAY})$ denoting the following:

- $\mathsf{M} \in \mathbb{N}$: the number of stages (i.e., the number of keys derived).
- $\mathsf{AUTH} \subseteq \{((u_1, m_1,) \ldots, (u_{\mathsf{M}}, m_{\mathsf{M}}))|u_j, m_j \in \{1, \ldots, \mathsf{M}, \infty\}\}$: a set of vectors of pairs, where each vector encodes a supported scheme for authentication and authentication upgrades. The i-th entry (u_i, m_i) of a vector indicates that the session key of stage i initially is unauthenticated, then becomes unilaterally authenticated at stage u_i, and finally reaches mutual authentication at stage m_i. Entries in each pair must be non-decreasing, and $u_i = \infty$ or $m_i = \infty$ denotes that unilateral, respectively mutual, authentication is never reached for stage i.
- $\mathsf{USE} \in \{\mathsf{internal}, \mathsf{external}\}^{\mathsf{M}}$: the usage indicator for each stage. We denote with USE_i the usage of the key of stage i. An internal key is used within the key exchange protocol (but possibly also outside of it), while an external key must not be used within the protocol.
- $\mathsf{FS} \in \{1, \ldots, \mathsf{M}, \infty\}$: the stage $j = \mathsf{FS}$ from which on keys are forward secure (or ∞ in case of no forward security).
- $\mathsf{REPLAY} \in \{\mathsf{replayable}, \mathsf{nonreplayable}\}^{\mathsf{M}}$: the replayability indicator for each stage. We denote with REPLAY_i whether the stage i is replayable meaning that an adversary is able to force identical session identifiers and keys in this stage.

We denote the set of users by $\mathcal{U}$, where each user is uniquely identified by some $U \in \mathcal{U}$. Protocol sessions are uniquely identified by a *label* $\mathsf{label} \in \mathsf{LABELS} = (\mathcal{U} \times \mathcal{U} \times \mathbb{N})$, where $\mathsf{label} = (U, V, k)$ indicates the k-th local session of the *session owner* U with V as the intended communication partner. Each session holds an identifier $\mathsf{pssid} \in \{0,1\}^*$ for the pre-shared secret $\mathsf{pss} \in \mathcal{P}$ (for some pre-shared secret space $\mathcal{P}$) used in the session. All pre-shared secrets are stored in maps $\mathsf{pss}_{U,\mathcal{V}} : \{0,1\}^* \to \mathcal{P}$ mapping pre-shared secret identifiers to the corresponding pre-shared secrets pss. A pre-shared secret pss stored in a map $\mathsf{pss}_{U,\mathcal{V}}$ is shared between the participants U and all $V \in \mathcal{V} \subseteq \mathcal{U}$ where U uses pss only in the initiator role and all $V \in \mathcal{V}$ use pss only in the responder role.

All sessions are stored in the *session list* $\mathsf{List}_{\mathsf{S}}$ with each entry holding the following information:

- $\mathsf{label} \in \mathsf{LABELS}$: the unique session label.
- $\mathsf{id} \in \mathcal{U}$: the session owner.
- $\mathsf{pid} \in \mathcal{U}$: the intended communication partner.
- $\mathsf{role} \in \{\mathsf{initiator}, \mathsf{responder}\}$: the role of the session owner.
- $\mathsf{auth} \in \mathsf{AUTH}$: the intended authentication type vector, where auth_i indicates the authentication level pair for stage i, and $\mathsf{auth}_{i,j}$ its j-th entry.
- $\mathsf{pssid} \in (\{0,1\}^* \cup \{\perp\})$: the identifier of the pre-shared secret to be used in this session.

- $\mathsf{st}_{\mathsf{exec}} \in (\mathsf{RUNNING} \cup \mathsf{ACCEPTED} \cup \mathsf{REJECTED})$: the state of execution, where $\mathsf{RUNNING} = \{\mathsf{running}_i \mid i \in \mathbb{N}_0\}$, $\mathsf{ACCEPTED} = \{\mathsf{accepted}_i \mid i \in \mathbb{N}\}$, $\mathsf{REJECTED} = \{\mathsf{rejected}_i \mid i \in \mathbb{N}\}$. $\mathsf{st}_{\mathsf{exec}}$ is initialized to $\mathsf{running}_0$. It is set to $\mathsf{accepted}_i$ when the i-th key is accepted, set to $\mathsf{running}_i$ when the protocol continues after accepting the i-th key, and set to $\mathsf{rejected}_i$ if the session rejects the i-th key (we assume a session does not continue after rejecting a key in any stage).
- $\mathsf{stage} \in [\mathsf{M}]_0$: the current stage. stage is initialized to 0 and set to i when $\mathsf{st}_{\mathsf{exec}}$ is set to $\mathsf{accepted}_i$ or $\mathsf{rejected}_i$.
- $\mathsf{sid} \in (\{0,1\}^* \cup \{\bot\})^{\mathsf{M}}$: sid_i indicates the session identifier of stage i, that is set once when $\mathsf{st}_{\mathsf{exec}}$ is set to $\mathsf{accepted}_i$. sid is initialized to $(\bot, \ldots, \bot)$.
- $\mathsf{cid} \in (\{0,1\}^* \cup \{\bot\})^{\mathsf{M}}$: cid_i indicates the contributive identifier of stage i, that may be set multiple times until $\mathsf{st}_{\mathsf{exec}}$ is set to $\mathsf{accepted}_i$. cid is initialized to $(\bot, \ldots, \bot)$.
- $\mathsf{key} \in (\{0,1\}^* \cup \{\bot\})^{\mathsf{M}}$: key_i indicates the established session key of stage i. key is initialized to $(\bot, \ldots, \bot)$.
- $\mathsf{st}_{\mathsf{key}} \in \{\mathsf{fresh}, \mathsf{revealed}\}^{\mathsf{M}}$: $\mathsf{st}_{\mathsf{key},i}$ indicates the state of the session key in stage i. $\mathsf{st}_{\mathsf{key}}$ is initialized to $(\mathsf{fresh}, \ldots, \mathsf{fresh})$.
- $\mathsf{tested} \in \{\mathsf{true}, \mathsf{false}\}^{\mathsf{M}}$: tested_i indicates whether the session key of stage i has been tested. tested is initialized to $(\mathsf{false}, \ldots, \mathsf{false})$.
- $\mathsf{corrupted} \in \{0, \ldots, \mathsf{M}, \infty\}$: indicates the stage the session was in when a $\mathsf{Corrupt}$ query for the pre-shared secret used in the session was issued. $\mathsf{corrupted}$ may be set to 0, indicating that the pre-shared secret was corrupted before the session started, and to ∞, indicating that the pre-shared secret is not corrupted. $\mathsf{corrupted}$ is initialized to ∞.

Whenever an incomplete tuple $(\mathsf{label}, \mathsf{id}, \mathsf{pid}, \mathsf{role}, \mathsf{auth}, k, \mathsf{pss}, \mathsf{pssid})$ is added to List_S the missing entries are initialized as described above. As a shorthand notation we use $\mathsf{label}.\mathsf{sid}$ for the entry sid of the tuple with the unique label in List_S. Two distinct sessions label and label' are defined to be *partnered* in stage i if $\mathsf{label}.\mathsf{sid}_i = \mathsf{label}'.\mathsf{sid}_i \neq \bot$. For correctness two sessions having a non-tampered joint execution are required to be partnered in all stages upon acceptance.

Upgradeable Authentication. The model captures that the authentication level of some stage may increase when some later stage is accepted. However, the authentication level of some stage i can only be increased in some later stage j if the pre-shared secret is not corrupted when stage j accepts. Otherwise, the adversary may be able to impersonate one participant of the session while it is unauthenticated and post-authenticate after corrupting the participant. To exclude this trivial attack the model introduces a *rectified authentication level*.

The rectified authentication level $\mathsf{rect\_auth}_i$ of some stage i in a session currently in stage stage with the intended authentication vector auth_i and the corruption indicator $\mathsf{corrupted}$ is defined as follows:

$$\mathsf{rect\_auth}_i := \begin{cases} \mathsf{mutual} & \text{if } \mathsf{stage} \geq \mathsf{auth}_{i,2} \text{ and } \mathsf{corrupted} \geq \mathsf{auth}_{i,2} \\ \mathsf{unilateral} & \text{if } \mathsf{stage} \geq \mathsf{auth}_{i,1} \text{ and } \mathsf{corrupted} \geq \mathsf{auth}_{i,1} \\ \mathsf{unauth} & \text{otherwise} \end{cases}$$

Intuitively, this captures that the authentication level of stage i is upgraded when stage $\mathsf{auth}_{i,1}$, respectively $\mathsf{auth}_{i,2}$, is reached only if the session is not corrupted before reaching these stages.

Adversary Model. We consider a probabilistic polynomial-time adversary $\mathcal{A}$ that controls the communication between all participants and can intercept, inject, and drop messages. We use a flag lost to capture actions by $\mathcal{A}$ where it trivially loses, e.g., revealing and testing the session key in partnered sessions. The flag lost is initialized to false. The adversary is given access to the following queries:

- $\mathsf{NewSecret}(U, \mathcal{V}, \mathsf{pssid})$: Generates a pre-shared secret pss with identifier pssid. pss is shared between the user U and all users $V \in \mathcal{V} \subseteq \mathcal{U}$ with $\mathcal{V} \neq \emptyset$. U uses pss in the $\mathsf{initiator}$ role and all $V \in \mathcal{V}$ use pss in the $\mathsf{responder}$ role. If the value $\mathsf{pss}_{U,\mathcal{V}}(\mathsf{pssid})$ is already defined, return $\perp$. Otherwise, sample pss uniformly at random from the pre-shared secret space $\mathcal{P}$ and define $\mathsf{pss}_{U,\mathcal{V}}(\mathsf{pssid}) := \mathsf{pss}$.
- $\mathsf{NewSession}(U, V, \mathcal{V}, \mathsf{role}, \mathsf{auth}, \mathsf{pssid})$: Creates a new session with a unique new label label with U as the session owner in the role role, V as the intended communication partner, and aiming at authentication type auth. pssid indicates the pre-shared secret to be used in the session. If $\mathsf{role} = \mathsf{initiator}$, it must hold that $V \in \mathcal{V}$ and the session then uses $\mathsf{pss}_{U,\mathcal{V}}(\mathsf{pssid})$. If $\mathsf{role} = \mathsf{responder}$, it must hold that $U \in \mathcal{V}$ and the session then uses $\mathsf{pss}_{V,\mathcal{V}}(\mathsf{pssid})$. Add $(\mathsf{label}, U, V, \mathsf{role}, \mathsf{auth}, \mathsf{pssid})$ to List_S. If label is corrupted, set $\mathsf{corrupted}$ to 0. Return label.
- $\mathsf{Send}(\mathsf{label}, m)$: Sends a message m to the session with label label. If no session with label label exists in List_S, return $\perp$. Otherwise, run the protocol on behalf of the session owner U on input of the message m and return the response as well as the updated state of execution $\mathsf{label.st}_\mathsf{exec}$. If $\mathsf{label.role} = \mathsf{initiator}$ and $m = \mathsf{init}$ the protocol is initiated without any input message.
  If, during the protocol execution, $\mathsf{label.st}_\mathsf{exec}$ is changed to $\mathsf{accepted}_i$ for some stage $i \in [\mathsf{M}]$, the protocol execution is suspended and $\mathsf{accepted}_i$ is returned to the adversary. In order to let the protocol execution resume and receive the next protocol message and execution state, the adversary may send a special message $m = \mathsf{continue}$ to the session.
  If, during the protocol execution, $\mathsf{label.st}_\mathsf{exec}$ is changed to $\mathsf{accepted}_i$ for some $i \in [\mathsf{M}]$ and there exists a session $\mathsf{label}' \neq \mathsf{label}$ partnered in stage i, i.e., $\mathsf{label.sid}_i = \mathsf{label}'.\mathsf{sid}_i$, in List_S with $\mathsf{label}'.\mathsf{tested}_i = \mathsf{true}$, then $\mathsf{label.tested}_i$ is set to true as well. Moreover, if $\mathsf{USE}_i = \mathsf{internal}$, $\mathsf{label.key}_i$ is set to $\mathsf{label}'.\mathsf{key}_i$. This ensures that, if the partnered session has been tested before, subsequent Test queries for the session are answered accordingly.
  If, during protocol execution, $\mathsf{label.st}_\mathsf{exec}$ is changed to $\mathsf{accepted}_i$ for some $i \in [\mathsf{M}]$ and the session label is corrupted, set $\mathsf{label.st}_{\mathsf{key},i}$ to $\mathsf{revealed}$.
- $\mathsf{Reveal}(\mathsf{label}, i)$: Reveals the session key $\mathsf{label.key}_i$ of the session with label label in stage i. If no session with label label exists in List_S or $\mathsf{label.stage} < i$, return $\perp$. Otherwise, set $\mathsf{label.st}_{\mathsf{key},i}$ to $\mathsf{revealed}$ and return $\mathsf{label.key}_i$ to the adversary.

- $\mathsf{Corrupt}(U, \mathcal{V}, \mathsf{pssid})$: Provides the pre-shared secret $\mathsf{pss}_{U,\mathcal{V}}(\mathsf{pssid})$ to the adversary. Add the global pre-shared secret identifier $(U, \mathcal{V}, \mathsf{pssid})$ to the set of corrupted entities $\mathcal{C}$. For all sessions label with $\mathsf{label}.(\mathsf{role}, \mathsf{id}, \mathsf{pid}, \mathsf{pssid}) \in \{(\mathsf{initiator}, U, V, \mathsf{pssid}), (\mathsf{responder}, V, U, \mathsf{pssid})\}$ and $\mathsf{label}.\mathsf{corrupted} \neq \infty$ for some $V \in \mathcal{V}$, set $\mathsf{label}.\mathsf{corrupted} \coloneqq \mathsf{label}.\mathsf{stage}$. For stage-$j$ forward secrecy, in any such session label, set $\mathsf{st}_{\mathsf{key},i}$ to $\mathsf{revealed}$ for all $i < j$ and all $i > \mathsf{label}.\mathsf{stage}$. For the non-forward secure case, in any such session label set $\mathsf{st}_{\mathsf{key},i}$ to $\mathsf{revealed}$ for all $i \in [\mathsf{M}]$.
- $\mathsf{Test}(\mathsf{label}, i)$: Tests the session key of stage i of the session with label label. The Test oracle stores a bit b_{test}, which is fixed throughout the game.
  If no session with label label exists in List_S or if $\mathsf{label}.\mathsf{st}_{\mathsf{exec}} \neq \mathsf{accepted}_i$ or if $\mathsf{label}.\mathsf{tested}_i = \mathsf{true}$, return $\perp$. If $\mathsf{USE}_i = \mathsf{internal}$ and there exists a session $\mathsf{label}' \neq \mathsf{label}$ partnered in stage i, i.e., $\mathsf{label}.\mathsf{sid}_i = \mathsf{label}'.\mathsf{sid}_i$, in List_S with $\mathsf{label}'.\mathsf{st}_{\mathsf{exec}} \neq \mathsf{accepted}$, set lost to true. If $\mathsf{label}.\mathsf{rect\_auth}_i = \mathsf{unilateral}$ and $\mathsf{label}.\mathsf{role} = \mathsf{responder}$ or if $\mathsf{label}.\mathsf{rect\_auth}_i = \mathsf{unauth}$, but there exists no session $\mathsf{label}' \neq \mathsf{label}$ in List_S with $\mathsf{label}.\mathsf{cid}_i = \mathsf{label}'.\mathsf{cid}_i$, then the flag lost is set to true. Otherwise, set $\mathsf{label}.\mathsf{tested}_i$ to true. This ensures that unilaterally authenticated stages in the responder's session and unauthenticated stages can only be tested if an honest contributive partner exists.
  If $b_{\mathsf{test}} = 0$, a key $K \xleftarrow{\$} \mathcal{S}$ is drawn uniformly at random from the session key space $\mathcal{S}$. If $b_{\mathsf{test}} = 1$, set K to the real session key $\mathsf{label}.\mathsf{key}_i$. If $\mathsf{USE}_i = \mathsf{internal}$, $\mathsf{label}.\mathsf{key}_i$ is set to K.
  If there exists a partnered session $\mathsf{label}' \neq \mathsf{label}$ in stage i, i.e., $\mathsf{label}.\mathsf{sid}_i = \mathsf{label}'.\mathsf{sid}_i$, in List_S with $\mathsf{label}.\mathsf{st}_{\mathsf{exec}} = \mathsf{label}'.\mathsf{st}_{\mathsf{exec}} = \mathsf{accepted}_i$, then $\mathsf{label}.\mathsf{tested}_i$ is set to true, and if $\mathsf{USE}_i = \mathsf{internal}$, $\mathsf{label}'.\mathsf{key}_i$ is set to $\mathsf{label}.\mathsf{key}_i$ as well.
  Return K.

The security of an MSKE protocol is modeled via two games played between a challenger and an adversary. The first game modeling Match security ensures the soundness of session identifiers, while the second game modeling $\mathsf{Multi\text{-}Stage}$ security ensures the classical key indistinguishability as well as further security properties like forward security and replay protection.

Match Security. The notion of Match security ensures that session identifiers properly identify partnered sessions. The Match security game $\mathsf{G}^{\mathsf{Match}}_{\mathsf{MSKE},\mathcal{A}}(\lambda)$ is defined as follows.

Definition 6 (Match security). *Let* MSKE *be a multi-stage key exchange protocol with properties* $(\mathsf{M}, \mathsf{AUTH}, \mathsf{USE}, \mathsf{FS}, \mathsf{REPLAY})$ *and* $\mathcal{A}$ *a PPT adversary interacting with* MSKE *in the following game* $\mathsf{G}^{\mathsf{Match}}_{\mathsf{MSKE},\mathcal{A}}(\lambda)$*:*

1. *The adversary* $\mathcal{A}$ *is given access to the queries* $\mathsf{NewSecret}$*,* $\mathsf{NewSession}$*,* Send*,* Reveal*,* $\mathsf{Corrupt}$*,* Test*.*
2. *At some point* $\mathcal{A}$ *stops without any output.*

We say that $\mathcal{A}$ *wins the game, denoted by* $\mathsf{G}^{\mathsf{Match}}_{\mathsf{MSKE},\mathcal{A}}(\lambda) = 1$*, if at least one of the following conditions holds:*

1. *There exist two distinct labels* $\mathsf{label}, \mathsf{label}' \in \mathsf{List}_\mathsf{S}$ *with* $\mathsf{label.sid}_i = \mathsf{label}'.\mathsf{sid}_i \neq \bot$ *for some stage* $i \in [\mathsf{M}]$*, but* $\mathsf{label.key}_i \neq \mathsf{label}'.\mathsf{key}_i$*. (Different session keys in some stage of partnered sessions.)*
2. *There exist two distinct labels* $\mathsf{label}, \mathsf{label}' \in \mathsf{List}_\mathsf{S}$ *with* $\mathsf{label.sid}_i = \mathsf{label}'.\mathsf{sid}_i \neq \bot$ *for some stage* $i \in [\mathsf{M}]$*, but* $\mathsf{label.role} = \mathsf{label}'.\mathsf{role}$ *and* $\mathsf{REPLAY}_i = \mathsf{nonreplayable}$ *or* $\mathsf{label.role} = \mathsf{label}'.\mathsf{role} = \mathsf{initiator}$ *and* $\mathsf{REPLAY}_i = \mathsf{replayable}$*. (Non-opposite roles of partnered sessions in non-replayable stage.)*
3. *There exist two distinct labels* $\mathsf{label}, \mathsf{label}' \in \mathsf{List}_\mathsf{S}$ *with* $\mathsf{label.sid}_i = \mathsf{label}'.\mathsf{sid}_i \neq \bot$ *for some stage* $i \in [\mathsf{M}]$*, but* $\mathsf{label.auth}_i \neq \mathsf{label}'.\mathsf{auth}_i$*. (Different authentication types in some stage of partnered sessions.)*
4. *There exist two distinct labels* $\mathsf{label}, \mathsf{label}' \in \mathsf{List}_\mathsf{S}$ *with* $\mathsf{label.sid}_i = \mathsf{label}'.\mathsf{sid}_i \neq \bot$ *for some stage* $i \in [\mathsf{M}]$*, but* $\mathsf{label.cid}_i \neq \mathsf{label}'.\mathsf{cid}_i$ *or* $\mathsf{label.cid}_i = \mathsf{label}'.\mathsf{cid}_i = \bot$*. (Different or unset contributive identifiers in some stage of partnered sessions.)*
5. *There exist two distinct labels* $\mathsf{label}, \mathsf{label}' \in \mathsf{List}_\mathsf{S}$ *with* $\mathsf{label.sid}_i = \mathsf{label}'.\mathsf{sid}_i \neq \bot$ *and* $\mathsf{label.sid}_j = \mathsf{label}'.\mathsf{sid}_j \neq \bot$ *for some stages* $i, j \in [\mathsf{M}]$ *with* $j \leq i$*, with* $\mathsf{label.role} = \mathsf{initiator}$*, and* $\mathsf{label}'.\mathsf{role} = \mathsf{responder}$ *such that*
   - $\mathsf{label.auth}_{j,1} \leq i$ *(unilateral authentication), but* $\mathsf{label.pid} \neq \mathsf{label}'.\mathsf{id}$*, or*
   - $\mathsf{label.auth}_{j,2} \leq i$ *(mutual authentication), but* $\mathsf{label.id} \neq \mathsf{label}'.\mathsf{pid}$.

   (Different authenticated partner or different key identifiers in mutual authentication)
6. *There exist two (not necessarily distinct) labels* $\mathsf{label}, \mathsf{label}' \in \mathsf{List}_\mathsf{S}$ *with* $\mathsf{label.sid}_i = \mathsf{label}'.\mathsf{sid}_j \neq \bot$ *for some stages* $i, j \in [\mathsf{M}]$ *with* $i \neq j$*. (Different stages share the same session identifier.)*
7. *There exist three pairwise distinct labels* $\mathsf{label}, \mathsf{label}', \mathsf{label}'' \in \mathsf{List}_\mathsf{S}$ *with* $\mathsf{label.sid}_i = \mathsf{label}'.\mathsf{sid}_i = \mathsf{label}''.\mathsf{sid}_i \neq \bot$ *for some stage* $i \in [\mathsf{M}]$ *with* $\mathsf{REPLAY}_i = \mathsf{nonreplayable}$*. (More than two sessions share the same session identifier in a non-replayable stage.)*

We say MSKE *is* Match*-secure if for all PPT adversaries* $\mathcal{A}$ *the advantage*

$$\mathsf{Adv}^{\mathsf{Match}}_{\mathsf{MSKE},\mathcal{A}}(\lambda) := \Pr[\mathsf{G}^{\mathsf{Match}}_{\mathsf{MSKE},\mathcal{A}}(\lambda) = 1]$$

is a negligible function in the security parameter λ*.*

Multi-Stage Security. The notion of Multi-Stage security ensures that established session keys are indistinguishable from randomly sampled keys.

Definition 7 (Multi-Stage **security).** *Let* MSKE *be a multi-stage key exchange protocol with key space* $\mathcal{S}$ *and properties* $(\mathsf{M}, \mathsf{AUTH}, \mathsf{USE}, \mathsf{FS}, \mathsf{REPLAY})$ *and* $\mathcal{A}$ *a PPT adversary interacting with* MSKE *in the following game* $\mathsf{G}^{\mathsf{Multi\text{-}Stage}}_{\mathsf{MSKE},\mathcal{A}}(\lambda)$*:*

1. *The challenger samples the bit* $b_{\mathsf{test}} \xleftarrow{\$} \{0,1\}$ *and sets the flag* lost *to* false.
2. *The adversary* $\mathcal{A}$ *is given access the the queries* NewSecret, NewSession, Send, Reveal, Corrupt, Test*. Note that such queries may set* lost *to* true*.*

3. *At some point $\mathcal{A}$ stops and ouputs a guess b. The challenger sets the flag* lost *to* true *if there exist two (not necessarily distinct) labels* $\mathsf{label}, \mathsf{label}' \in \mathsf{List}_\mathsf{S}$ *and some stage* $i \in [\mathsf{M}]$ *with* $\mathsf{label}.\mathsf{sid}_i = \mathsf{label}'.\mathsf{sid}_i$, $\mathsf{label}.\mathsf{st}_{\mathsf{key},i} = \mathsf{revealed}$, *and* $\mathsf{label}'.\mathsf{tested}_i = \mathsf{true}$, *i.e., the adversary has tested and revealed the key of some stage in a single session or in two partnered sessions.*

We say that $\mathcal{A}$ wins the game, denoted by $\mathsf{G}^{\mathsf{Multi\text{-}Stage}}_{\mathsf{MSKE},\mathcal{A}}(\lambda) = 1$, *if* $b = b_{\mathsf{test}}$ *and* $\mathsf{lost} = \mathsf{false}$. *We say* MSKE *with properties* $(\mathsf{M}, \mathsf{AUTH}, \mathsf{USE}, \mathsf{FS}, \mathsf{REPLAY})$ *is* Multi-Stage*-secure if* MSKE *is* Match*-secure and for all PPT adversaries $\mathcal{A}$ the advantage*

$$\mathsf{Adv}^{\mathsf{Multi\text{-}Stage}}_{\mathsf{MSKE},\mathcal{A}}(\lambda) := \Pr[\mathsf{G}^{\mathsf{Multi\text{-}Stage}}_{\mathsf{MSKE},\mathcal{A}}(\lambda) = 1] - \frac{1}{2}$$

is a negligible function in the security parameter λ.

3 Breaking the Security of Session Resumption Across Hostnames in TLS 1.3

In this section we show that session resumption across hostnames in TLS 1.3 is not a secure MSKE protocol if clients do not include the SNI value of the intended receiver in the ClientHello. As mentioned before, the TLS 1.3 specification [17] does not precisely state that resumption handshakes require an SNI value to be sent by the client. We therefore assume that in practice this may not always be seen as an absolute requirement for implementations and it may happen that clients do not specify an SNI value in their ClientHello.

We will exploit this lack of an SNI value to break the authentication of TLS 1.3 if two servers share the same certificate and symmetric ticket encryption key. Our attack targets a client C that wants to perform SRAH with some server S_A. The core idea of the attack is for the adversary to reroute any messages from C to a different server S_B that is able to execute the session resumption. This results in S_B implicitly authenticating itself by its ability to accept the session ticket while from the view of C it was S_A that implicitly authenticated itself.

3.1 Modeling TLS 1.3 Session Resumption as an MSKE Protocol

We begin by formally modeling the TLS 1.3 session resumption as an MSKE protocol. We consider both the PSK-only as well as the PSK-(EC)DHE handshake variants of TLS 1.3. We illustrate both variants in Figure 1. The TLS 1.3 session resumption protocol proceeds as follows. The client's first message ClientHello contains the protocol version, a random nonce, and a list of supported cryptographic algorithms. Moreover, it contains a `pre_shared_key` extension that indicates the identity of the PSK used in this handshake. In the PSK-(EC)DHE handshake it additionally contains the client's Diffie–Hellman share in the `key_share` extension. Additionally, the client may use the `server_name` extension to specify the SNI value of the intended receiver.

Furthermore, from the PSK the client derives the early secret ES. The early secret ES is then expanded into the binder key bk and the early traffic secret ets. The binder key bk is used to derive fk_0, which is used to compute the Fin_0 message. The Fin_0 message constitutes a MAC tag over the ClientHello and thus ensures its integrity and authenticity. As the last step in stage 1, the early traffic key $\mathsf{tk}_{\mathsf{ets}}$ is derived from the early traffic secret ets and set as the stage's session key. The client may use $\mathsf{tk}_{\mathsf{ets}}$ to encrypt early application data and send them to the server. However, note that this is not part of the handshake protocol itself.

Stage 2 only consists of deriving the early exporter master secret EEMS from the early secret ES. In stage 3 the server sends a ServerHello in response to the ClientHello. It contains the protocol version, a random nonce, the selected cryptographic algorithms, and the `pre_shared_key` extension confirming the identity of the PSK. In the PSK-(EC)DHE handshake, it additionally contains the server's Diffie–Hellman share in the `key_share` extension. Other optional extensions are sent separately in an EncryptedExtensions message, although we omit that message for simplicity.

Moreover, the early secret ES is expanded into the derived early secret dES. Next, the handshake secret HS is computed: in the PSK-only handshake it is derived only from dES, while in the PSK-(EC)DHE handshake it is derived from dES and the shared Diffie–Hellman secret. The handshake secret HS is expanded into the client handshake traffic secret chts, respectively server handshake traffic secret shts. The handshake traffic secrets are then used to compute the stage's session keys $\mathsf{tk}_{\mathsf{chts}}$ and $\mathsf{tk}_{\mathsf{shts}}$ from chts, respectively shts.

In stage 4 the server sends the Fin_S message, which constitutes a MAC over all previously exchanged messages. It is computed with the server finished key fk_S derived from the server handshake traffic secret shts and encrypted with the server handshake traffic key $\mathsf{tk}_{\mathsf{shts}}$. Moreover, the handshake secret HS is expanded into the derived handshake secret dHS, from which the master secret MS is computed. The master secret MS is then used to derive the client, respectively server, application traffic secret cats, respectively sats. As the last step in stage 4, the client, respectively server, application traffic key $\mathsf{tk}_{\mathsf{cats}}$, respectively $\mathsf{tk}_{\mathsf{sats}}$, are computed from cats, respectively sats. The stage's session keys $\mathsf{tk}_{\mathsf{cats}}$ and $\mathsf{tk}_{\mathsf{sats}}$ are used to encrypt any exchanged application data. Again, note that this is not actually a part of the handshake protocol itself.

Stage 5 only consists of deriving the exporter master secret EMS from the master secret MS. In stage 6 the client sends the Fin_C message, which again constitutes a MAC over all previously exchanged messages. It is computed with the client finished key fk_C derived from the client handshake traffic secret chts and encrypted with the client handshake traffic key $\mathsf{tk}_{\mathsf{chts}}$. The last step of the protocol consists of deriving the resumption master secret RMS from the master secret MS.

For completeness we provide an exact listing of the messages over which the MACs are computed in Table 1. In Figure 1 for readability we separated the Fin_0 message from the ClientHello, although according to the standard it is contained in the ClientHello as the *PSK binder* (cf. [17, § 4.2.11.2]). Moreover, note that

both participants possess a pre-shared key PSK. The value PSK is derived from the RMS computed in the previous handshake and a random nonce, which is provided by the server via the $\mathsf{NewSessionTicket}$ message. The $\mathsf{NewSessionTicket}$ message is encrypted under $\mathsf{tk}_{\mathsf{sats}}$ and additionally contains an opaque label `ticket`, which is used as the PSK identifier in the `pre_shared_key` extension.

Similar to previous security analyses of TLS 1.3 [7, 9, 12], we capture neither the $\mathsf{NewSessionTicket}$ message nor the derivation of PSK. In order to simplify the analysis the previous works treat $\mathsf{tk}_{\mathsf{sats}}$ as an external key that can be used in an arbitrary symmetric protocol. To capture the transmission of the $\mathsf{NewSessionTicket}$ message, $\mathsf{tk}_{\mathsf{sats}}$ would have to be treated as an internal key. Instead it is assumed that some out-of-band mechanism is used to establish a mapping between PSK identifiers and the PSK values. We follow the same approach and assume the same out-of-band mapping.

Table 1: List of the hash values and the transcript messages used to compute them in the TLS 1.3 session resumption handshake.

Hash value	**Messages**
H_1	$\mathsf{ClientHello}$
H_2	$\mathsf{ClientHello}$, Fin_0
H_3	$\mathsf{ClientHello}$, Fin_0, $\mathsf{ServerHello}$
H_4	$\mathsf{ClientHello}$, Fin_0, $\mathsf{ServerHello}$, $\mathsf{EncryptedExtensions}$
H_5	$\mathsf{ClientHello}$, Fin_0, $\mathsf{ServerHello}$, $\mathsf{EncryptedExtensions}$, Fin_S
H_6	$\mathsf{ClientHello}$, Fin_0, $\mathsf{ServerHello}$, $\mathsf{EncryptedExtensions}$, Fin_S, Fin_C

Protocol properties. The protocol properties of the TLS 1.3 PSK-only and PSK-(EC)DHE handshakes in the MSKE model are as follows:

- $\mathsf{M} = 6$. Both handshake variants comprise 6 stages deriving the keys $\mathsf{tk}_{\mathsf{ets}}$, EEMS, $\mathsf{tk}_{\mathsf{chts}}/\mathsf{tk}_{\mathsf{shts}}$, $\mathsf{tk}_{\mathsf{cats}}/\mathsf{tk}_{\mathsf{sats}}$, EMS, and RMS.
- The authentication properties AUTH are different for each handshake variant. We explain the reasoning for the difference in detail in Remark 8.
  - for PSK-only $\mathsf{AUTH} = \{((1,1),(2,2),(3,3),(4,4),(5,5),(6,6))\}$: all keys are immediately mutually authenticated.
  - for PSK-(EC)DHE $\mathsf{AUTH} = \{((1,1),(2,2),(4,4),(4,4),(5,5),(6,6))\}$: the keys $\mathsf{tk}_{\mathsf{chts}}$ and $\mathsf{tk}_{\mathsf{shts}}$ are at first unauthenticated and reach mutual authentication in stage 4, while all other keys are immediately mutually authenticated.
- The forward security is different for the two handshake variants:
  - for PSK-only $\mathsf{FS} = \infty$. The PSK-only variant does not provide forward security.
  - for PSK-(EC)DHE $\mathsf{FS} = 3$. The PSK-(EC)DHE variant provides forward security for all stages $i \geq 3$.

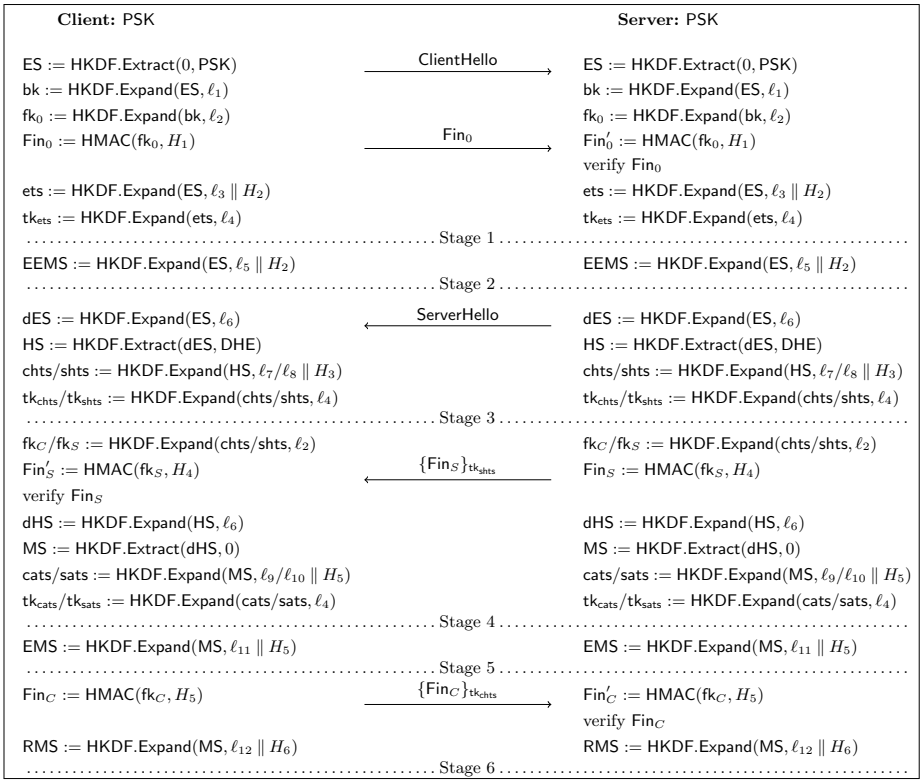

Fig. 1: The TLS 1.3 session resumption protocol. The client and server possess a pre-shared secret PSK. A dotted line indicates the key of the stage noted at that line being accepted. The values $\ell_1, \ldots, \ell_{11}$ are publicly known string labels.The values $H_1, \ldots, H_6$ are hash values computed from the transcript of the communication. We provide the exact listing of messages for each hash value in Table 1. In the PSK-only variant DHE is set to 0 and in the PSK-(EC)DHE variant DHE is the shared (elliptic curve) Diffie–Hellman key. $\{m\}_k$ denotes a message m being encrypted with key k.

- $\mathsf{USE} = (\mathsf{external}, \mathsf{external}, \mathsf{internal}, \mathsf{external}, \mathsf{external}, \mathsf{external})$. The handshake traffic keys $\mathsf{tk}_{\mathsf{chts}}$ and $\mathsf{tk}_{\mathsf{shts}}$ are used internally to encrypt some handshake messages. All other keys are used externally.
- $\mathsf{REPLAY} = (\mathsf{replayable}, \mathsf{replayable}, \mathsf{nonreplayable}, \mathsf{nonreplayable}, \mathsf{nonreplayable}, \mathsf{nonreplayable})$. The stages 1 and 2 with the keys $\mathsf{tk}_{\mathsf{ets}}$ and EEMS are replayable. All other stages are non-replayable.

Remark 8. In the PSK-EC(DHE) handshake the keys $\mathsf{tk}_{\mathsf{chts}}$ and $\mathsf{tk}_{\mathsf{shts}}$ of stage 3 only reach mutual authentication in stage 4 due to the following attack. An adversary can intercept the $\mathsf{ServerHello}$, replace the server's Diffie–Hellman share with its own share, and then send the modified $\mathsf{ServerHello}$ to the client. After corrupting the pre-shared secret the adversary can then compute $\mathsf{tk}_{\mathsf{chts}}$ and $\mathsf{tk}_{\mathsf{shts}}$ from the pre-shared secret, the client's Diffie–Hellman share and its own Diffie–Hellman share. Note that the adversary cannot execute a similar attack by replacing the client's Diffie–Hellman share instead of the server's share since in contrast to the Fin_S message, which is sent separately from the $\mathsf{ServerHello}$, the Fin_0 message is included in the $\mathsf{ClientHello}$.

To prevent this attack we at first treat $\mathsf{tk}_{\mathsf{chts}}$ and $\mathsf{tk}_{\mathsf{shts}}$ as unauthenticated and as mutually authenticated once the server authenticated its Diffie–Hellman share in stage 4. An alternative approach is to prohibit the adversary from corrupting the pre-shared secret during stage 3 by treating stage 3 as non-forward secure. However, this would be a counterintuitive solution as both parties contributed ephemeral key material to the key of stage 3. Thus, the straightforward choice is to treat $\mathsf{tk}_{\mathsf{chts}}$ and $\mathsf{tk}_{\mathsf{shts}}$ as unauthenticated in stage 3.

Session and Contributive Identifiers. We define the session identifiers of each stage to comprise all handshake messages sent up to that stage. In order to have different session identifiers for each stage, we add a label string if in some stage no handshake message is sent. Formally, the session identifiers are defined as follows:

$$\mathsf{sid}_1 = (\mathsf{ClientHello}, \mathsf{Fin}_0) \qquad \mathsf{sid}_2 = (\mathsf{sid}_1, \text{“EEMS”}) \qquad \mathsf{sid}_3 = (\mathsf{sid}_2, \mathsf{ServerHello})$$
$$\mathsf{sid}_4 = (\mathsf{sid}_3, \mathsf{Fin}_S) \qquad \mathsf{sid}_5 = (\mathsf{sid}_4, \text{“EMS”}) \qquad \mathsf{sid}_6 = (\mathsf{sid}_5, \mathsf{Fin}_C)$$

For all stages $i \in \{1, 2, 4, 5, 6\}$ we set $\mathsf{cid}_i = \mathsf{sid}_i$ when sid_i is set. For stage 3 on sending (resp. receiving), the $\mathsf{ClientHello}$, the client (resp. server) sets $\mathsf{cid}_3 = (\mathsf{ClientHello}, \mathsf{Fin}_0)$. Similarly, on sending (resp. receiving) the $\mathsf{ServerHello}$, the server (resp. client) extends cid_3 to $(\mathsf{ClientHello}, \mathsf{Fin}_0, \text{“EEMS”}, \mathsf{ServerHello})$.

3.2 The Attack

We now proceed to describe the attack breaking the security of SRAH in TLS 1.3 in the MSKE model. To this end, we give a sequence of query calls that allows us to win the Match security game $\mathsf{G}^{\mathsf{Match}}_{\mathsf{TLS\text{-}1.3\text{-}SRAH},\mathcal{A}}$ with a probability of 1, proving that SRAH in TLS 1.3 is not a Match-secure protocol. The attack is applicable to both the PSK-only handshake as well as the PSK-(EC)DHE handshake.

Theorem 9. *If the client does not include the SNI value of the intended communication partner in the* ClientHello*, the TLS 1.3 PSK-only and the TLS 1.3 PSK-(EC)DHE handshakes with SRAH are not* Match*-secure and we can construct an adversary $\mathcal{A}$ with the advantage*

$$\mathsf{Adv}^{\mathsf{Match}}_{\mathsf{TLS\text{-}1.3\text{-}SRAH},\mathcal{A}}(\lambda) = 1.$$

Proof. We construct $\mathcal{A}$ as follows. Let $\mathsf{pssid} \in \{0,1\}^*$ be a pre-shared key identifier, $C, S, S' \in \mathcal{U}$ be participant identities, and $\mathsf{auth} \in \{((1,1),(2,2),(3,3),(4,4),(5,5),(6,6)),((1,1),(2,2),(4,4),(4,4),(5,5),(6,6))\}$ depending on the handshake variant. We proceed as follows:

1. $\mathsf{NewSecret}(C, \{S, S'\}, \mathsf{pssid})$. We begin by generating a new pre-shared secret with the identifier pssid shared between C, S, and S'.
2. $\mathsf{NewSession}(C, S, \{S, S'\}, \mathsf{initiator}, \mathsf{auth}, \mathsf{pssid})$. We create a new session with the label $\mathsf{label} = (C, S, 1)$ for the client C with the server S as the intended communication partner, where the session is resumed based on the pre-shared secret we created in step 1.
3. $\mathsf{NewSession}(S', C, \{S, S'\}, \mathsf{responder}, \mathsf{auth}, \mathsf{pssid})$. We then create a new session with the label $\mathsf{label}' = (S', C, 1)$ for the server S' with the client C as the intended communication partner, where the session is resumed based on the pre-shared secret we created in step 1.
4. $\mathsf{Send}((C, S, 1), \mathsf{init})$. We send the special message init to the client C, which initiates the protocol. After C accepts the key of stage 1, it sets $\mathsf{sid}_1 = (\mathsf{ClientHello}, \mathsf{Fin}_0)$ and suspends the execution we send the special message $\mathsf{continue}$ to C to obtain the messages $\mathsf{ClientHello}$ and Fin_0.
5. $\mathsf{Send}((S', C, 1), (\mathsf{ClientHello}, \mathsf{Fin}_0))$. We send the messages $\mathsf{ClientHello}$ and Fin_0 to the server S'. Since we did not modify either of the messages, the Fin_0 message will successfully be verified. Therefore, S' accepts the key of stage 1, sets $\mathsf{sid}_1 = (\mathsf{ClientHello}, \mathsf{Fin}_0)$ and suspends the protocol execution.

The above sequence of query calls results in the game $\mathsf{G}^{\mathsf{Match}}_{\mathsf{TLS\text{-}1.3\text{-}SRAH},\mathcal{A}}$ being won with probability 1. The session label is partnered with the session label' and we have $\mathsf{label.auth}_{1,1} = 1$ but $\mathsf{label.pid} \neq \mathsf{label}'.\mathsf{id}$. Observe that this fulfills condition 5 in $\mathsf{G}^{\mathsf{Match}}_{\mathsf{TLS\text{-}1.3\text{-}SRAH},\mathcal{A}}$ and we therefore have

$$\mathsf{Adv}^{\mathsf{Match}}_{\mathsf{TLS\text{-}1.3\text{-}SRAH},\mathcal{A}}(\lambda) = 1.$$

4 Secure SRAH Protocols

In this section we formally define secure SRAH protocols, which will resemble an abstraction of the TLS 1.3 session resumption. While in TLS 1.3 multiple keys are derived for many different purposes (e.g., partial encryption of the handshake, or multiple keys for application-based encryption), we start with a reduced approach to SRAH, which generates two keys only: one “temporary” key

to send (optional) data within the first protocol message[4] and one potentially stronger "main" key as result of the handshake. Reducing the complexity of the protocol in comparison to TLS 1.3 will later allow us to achieve a simpler security proof compared to the security proofs of the full TLS 1.3 protocol by Dowling *et al.* [7].

Definition 10. *A* session resumption across hostnames *protocol with key space* $\mathcal{S}$ *is a tuple of five PPT algorithms* $\mathsf{SRAH} = (\mathsf{KeyGen}, \mathsf{TicketGen}, \mathsf{SessionRes}^{\mathsf{client}}_{\mathsf{init}}, \mathsf{SessionRes}^{\mathsf{server}}_{\mathsf{refresh}}, \mathsf{SessionRes}^{\mathsf{client}}_{\mathsf{refresh}})$.

$\mathsf{KeyGen}(1^\lambda) \to k$**.** *On input of a security parameter* λ *the algorithm outputs a long–term key* k*.*

$\mathsf{TicketGen}(s) \to t$**.** *On input of a secret* s *the algorithm outputs a session ticket* t*.*

$\mathsf{SessionRes}^{\mathsf{client}}_{\mathsf{init}}(s, t, j) \to (s_{\mathsf{tmp}}, m_{\mathbf{C}})$**.** *On input of a secret* $s \in \mathcal{S}$*, a ticket* $t \in \{0,1\}^*$ *and a server identifier* j *the algorithm outputs a temporary secret* s_{tmp} *and a message* m_C*.*

$\mathsf{SessionRes}^{\mathsf{server}}_{\mathsf{refresh}}(k, m_{\mathbf{C}}) \to (s_{\mathsf{tmp}}, s_{\mathsf{main}}, m_{\mathbf{S}})$**.** *On input of a long-term key* k *and a message* m_C *the algorithm outputs a temporary secret* s_{tmp}*, a secret* s_{main}*, and a message* m_S*.*

$\mathsf{SessionRes}^{\mathsf{client}}_{\mathsf{refresh}}(m_{\mathbf{S}}) \to s_{\mathsf{main}}$**.** *On input of a message* m_S *the algorithm outputs a secret* s_{main}*.*

We say a session resumption across hostnames protocol is correct if for all $k \xleftarrow{\$} \mathsf{KeyGen}(1^\lambda)$*, all* $s \xleftarrow{\$} \mathcal{S}$*, and all* $t \xleftarrow{\$} \mathsf{TicketGen}(k, s)$ *it holds that* $s_{\mathsf{tmp}} = s'_{\mathsf{tmp}}$ *and* $s_{\mathsf{main}} = s'_{\mathsf{main}}$*, where* $(s_{\mathsf{tmp}}, m_C) \xleftarrow{\$} \mathsf{SessionRes}^{\mathsf{client}}_{\mathsf{init}}(s, t, j)$*,* $(s'_{\mathsf{tmp}}, s'_{\mathsf{main}}, m_S) \xleftarrow{\$} \mathsf{SessionRes}^{\mathsf{server}}_{\mathsf{refresh}}(k, m_C)$*, and* $s_{\mathsf{main}} \xleftarrow{\$} \mathsf{SessionRes}^{\mathsf{client}}_{\mathsf{refresh}}(m_S)$*.*

We use the modified version of the MSKE security model described in section 2.1 to define the security of SRAH protocols.

Definition 11. *We say a SRAH protocol is* secure *if it is* Multi-Stage*-secure.*

Using an SRAH protocol. The flow of an SRAH protocol is shown in Figure 2.

4.1 Constructing Secure SRAH Protocols

In this section we show a construction of a secure SRAH protocol. Our construction essentially resembles an abstraction of the TLS 1.3 session resumption protocol with the addition of a mandatory server identifier in the client's first message, which identifies the intended recipient of the message. By adding the server identifier we prevent an adversary from applying the attack described in Section 3.2.

[4] This captures the optional zero round-trip time feature of TLS 1.3 resumption handshakes, where a client may send encrypted early data with its first flight of messages. Note that due to the lack of interaction, this often comes at the cost of forward security for this message.

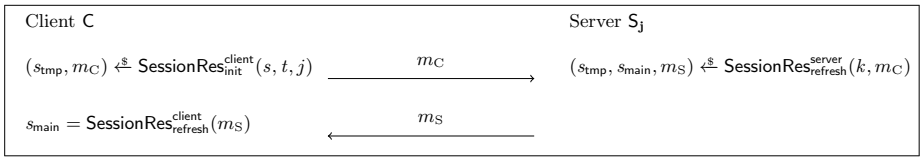

Fig. 2: Execution of an SRAH protocol between a client C_i and a server S_j where $t = \mathsf{TicketGen}(k, s)$, computed by a server $\mathsf{S}_\ell \neq \mathsf{S}_j$.

Recall that the main idea of the attack in Section 3.2 is to reroute the messages sent by the client to a different server than the intended receiver. This is unnoticed by both the client and the server receiving the message since the client's messages do not include any information on the intended recipient. However, if the message m_C contains the identifier of the intended receiver and an adversary reroutes it to a different server, that server recognizes that it is not the intended recipient of the message and aborts the handshake. Moreover, we use a MAC to prevent the adversary from modifying the server identifier.

With the `server_name` extension TLS 1.3 already provides a mechanism to indicate the intended receiver of a ClientHello. We recommend to make the `server_name` extension mandatory in session resumption handshakes to prevent the attack as described intuitively above. In the following we formally prove that this change prevents the attack by showing that our construction, which abstracts TLS 1.3 session resumption with a mandatory `server_name` extension, is a secure MSKE protocol.

Definition 12. *Let $\Pi = (\Pi.\mathsf{KeyGen}, \Pi.\mathsf{Enc}, \Pi.\mathsf{Dec})$ be a symmetric encryption scheme, HMAC be the HMAC construction, HKDF the HKDF scheme, $\mathbb{G}$ a cyclic group of prime order p, and g a generator of $\mathbb{G}$. We construct a SRAH protocol $\mathsf{SRAH} = (\mathsf{KeyGen}, \mathsf{TicketGen}, \mathsf{SessionRes}^{\mathsf{client}}_{\mathsf{init}}, \mathsf{SessionRes}^{\mathsf{server}}_{\mathsf{refresh}}, \mathsf{SessionRes}^{\mathsf{client}}_{\mathsf{refresh}})$ as follows.*

$\mathsf{KeyGen}(1^\lambda)$**.** *Returns $k \xleftarrow{\$} \Pi.\mathsf{KeyGen}(1^\lambda)$.*
$\mathsf{TicketGen}(k, s)$**.** *Returns $t \xleftarrow{\$} \Pi.\mathsf{Enc}(k, s)$.*
$\mathsf{SessionRes}^{\mathsf{client}}_{\mathsf{init}}(s, t, \mathsf{ID})$**.** *Samples $x \xleftarrow{\$} \mathbb{Z}_p^*, r_C \xleftarrow{\$} \{0,1\}^\lambda$, computes*

$$\begin{aligned} s_{\mathsf{tmp}} &\coloneqq \mathsf{HKDF.Extract}(\text{“}tmp\text{”}, s), \\ s_{\mathsf{MAC}} &\coloneqq \mathsf{HKDF.Extract}(\text{“}MAC\text{”}, s), \\ \tau_C &\coloneqq \mathsf{HMAC}(s_{\mathsf{MAC}}, g^x \parallel r_C \parallel j \parallel t), \end{aligned}$$

and returns (s_{tmp}, m_C), where $m_C \coloneqq (g^x, r_C, j, \tau_C, t)$.
$\mathsf{SessionRes}^{\mathsf{server}}_{\mathsf{refresh}}(k, m_{\mathbf{C}})$**.** *Parses m_C as $(g^x, r_C, j', \tau_C, t)$. If j' is not the identifier of the executing server, returns $\perp$. Otherwise computes*

$$\begin{aligned} s &= \Pi.\mathsf{Dec}(k, t), \\ s_{\mathsf{MAC}} &= \mathsf{HKDF.Extract}(\text{“}MAC\text{”}, s). \end{aligned}$$

If $\mathsf{HMAC}(s_{\mathsf{MAC}}, g^x \parallel r_C \parallel j' \parallel t) \neq \tau_C$*, returns* $\perp$*. Otherwise samples* $y \xleftarrow{\$} \mathbb{Z}_p^*, r_S \xleftarrow{\$} \{0,1\}^\lambda$*, computes*

$$s_{\mathsf{tmp}} \coloneqq \mathsf{HKDF.Extract}(\text{“}tmp\text{”}, s),$$
$$s_{\mathsf{main}} \coloneqq \mathsf{HKDF.Extract}(s, g^{xy}),$$
$$\tau_S \coloneqq \mathsf{HMAC}(s_{\mathsf{MAC}}, m_C \parallel g^y \parallel r_S)$$

and returns $(s_{\mathsf{tmp}}, s_{\mathsf{main}}, m_S)$*, where* $m_S \coloneqq (g^y, r_S, \tau_S)$*.*

$\mathsf{SessionRes}^{\mathsf{client}}_{\mathsf{refresh}}(m_{\mathbf{S}})$**.** *Parses* m_S *as* (g^y, r_S, τ_S)*. If* $\mathsf{HMAC}(s_{\mathsf{MAC}}, m_C \parallel g^y \parallel r_S) \neq \tau_S$*, returns* $\perp$*. Otherwise computes*

$$s_{\mathsf{main}} \coloneqq \mathsf{HKDF.Extract}(s, g^{xy})$$

and returns s_{main}*.*

We set the following protocol-specific properties for the MSKE model:

- $\mathsf{M} = 2$*: The number of stages is equal to two, where* s_{tmp} *is the key of stage 1, and* s_{main} *the key of stage 2.*
- $\mathsf{AUTH} = \{((1,1),(2,2))\}$*: All keys are mutually authenticated.*
- $\mathsf{USE} = (\mathsf{external}, \mathsf{external})$*: Both keys are used only externally.*
- $\mathsf{FS} = 2$*: The main key is forward secure, while the temporary key is not forward secure.*
- $\mathsf{REPLAY} = (\mathsf{replayable}, \mathsf{nonreplayable})$*: stage 1 is replayable while stage 2 is non-replayable.*

Moreover, we define the session identifiers of both stages to comprise all messages sent up to that stage. Formally, the session identifiers are defined as $\mathsf{sid}_1 = (m_C)$ *and* $\mathsf{sid}_2 = (m_C, m_S)$*. For stage 1 we set the contributive identifier* $\mathsf{cid}_1 = \mathsf{sid}_1$ *when* sid_i *is set. On sending (resp. receiving) the message* m_C*, the client (resp. server) sets* $\mathsf{cid}_2 = (m_C)$*. Similarly, on sending (resp. receiving) the message* m_S*, the server (resp. client) sets* $\mathsf{cid}_2 = (m_C, m_S)$*.*

Theorem 13. *The above construction* SRAH *is* Match*-secure with properties* $(\mathsf{M}, \mathsf{AUTH}, \mathsf{USE}, \mathsf{FS}, \mathsf{REPLAY})$ *given above. For each PPT adversary* $\mathcal{A}$*, there exists an algorithm* $\mathcal{B}$ *such that*

$$\mathsf{Adv}^{\mathsf{Match}}_{\mathsf{SRAH},\mathcal{A}}(\lambda) \leq \mathsf{Adv}^{\mathsf{collision}}_{\mathsf{HMAC},\mathcal{B}}(\lambda) + \frac{n_p^2}{|\mathcal{P}|} + n_s^2 \cdot \frac{1}{p} + 2^{-\lambda}.$$

Proof. We need to show that the seven conditions for Match-security hold:

1. *Sessions with the same session identifier for some stage hold the same key at that stage.* All keys derived in the protocol SRAH are uniquely determined by the messages the pre-shared secret and the Diffie–Hellman secret. Furthermore, all session identifiers contain all messages exchanged up to that stage. The message m_C contains the pre-shared secret identifier and the client's Diffie–Hellman share. The message m_S contains the server's Diffie–Hellman

share. Therefore, two sessions with the same session identifier agree on the pre-shared secret as well as the Diffie-Hellman secret used to derive all session keys. Thus, sessions with the same session identifier in some stage hold the same key in that stage.

2. *Sessions with the same session identifier for some stage have opposite roles, except for potential multiple responders in replayable stages.* Since stage 1 is replayable we only consider stage 2. As shown in condition 7 in a nonreplayable stage there are at most two sessions with the same session identifier. As a session does not accept a message intended for the opposite role, the two session having the same session identifier must have opposite roles.
3. *Sessions with the same session identifier for some stage agree on that stage's authentication level.* This trivially holds since the authentication vector is fixed to $((1,1),(2,2))$.
4. *Sessions with the same session identifier for some stage share the same contributive identifier at that stage.* This trivially holds for stage 1 as $\mathsf{cid}_1 = \mathsf{sid}_1$. Once cid_2 is final, i.e., when stage 2 is accepted and sid_2 is set, $\mathsf{cid}_2 = \mathsf{sid}_2$ holds as well.
5. *Sessions are partnered with the intended (authenticated) participant and, for mutual authentication, share the same key identifier.* The message m_{C}, which contains pssid, τ_C and the intended communication partner, is part of the session identifier in both stages. This first ensures that the initiator is partnered with its intended partner since any other participant aborts the handshake if it receives m_{C}. Additionally, this ensures that two partnered session always agree on pssid. As τ_C is computed from pss through a series of HKDF and HMAC invocations, two sessions agreeing on τ_C implies that the session also agree on pss if we consider HMAC as an unkeyed collision resistant hash function. Since pss is chosen uniformly at random in the NewSecret query, we can bound the probability of pre-shared secrets colliding with the negligible birthday bound $n_p^2/|\mathcal{P}|$ where $\mathcal{P}$ is the pre-shared secret space and n_p is the number of pre-shared secrets. Thus, from the client's and the server's perspective pss must originate from the same call to the NewSecret query, which to the server uniquely identifies its communication partner as its intended partner.
6. *Session identifiers do not match across different stages.* This trivially holds due to the session identifiers being unique for each stage.
7. *At most two sessions have the same session identifier at any nonreplayable stage.* As the first stage is replayable, we only consider the second stage. The session identifier of stage 2 contains the messages m_{C} and m_{S}, which both contain a Diffie–Hellman share and a nonce. In order for three session to have the same session identifier, two sessions would need to draw the same Diffie–Hellman share and nonce and then both be partnered with some third session. The probability of this can be upper-bounded with the birthday bound by $n_s^2 \cdot 1/p \cdot 2^{-\lambda}$, where n_s is the maximum number of sessions, p is the group order, and λ the length of the nonce.

Theorem 14. *The above construction* SRAH *is* Multi-Stage*-secure with the properties* (M, AUTH, USE, FS, REPLAY) *given above. For each efficient adversary* $\mathcal{A}$*,*

there exist efficient algorithms $\mathcal{B}_1, \ldots, \mathcal{B}_8$ *such that*

$$\mathsf{Adv}^{\mathsf{Multi\text{-}Stage}}_{\mathcal{A},\mathsf{SRAH}}(\lambda) \leq 2n_s \cdot \Big(n_p \cdot \Big(\mathsf{Adv}^{\mathsf{CPA}}_{\Pi,\mathcal{B}_1} + \mathsf{Adv}^{\mathsf{dual\text{-}PRF\text{-}sec}}_{\mathsf{HKDF.Extract},\mathcal{B}_2}(\lambda) + \mathsf{Adv}^{\mathsf{sEUF\text{-}CMA}}_{\mathsf{HMAC},\mathcal{B}_3}(\lambda) \Big) + n_p \cdot n_s \cdot \Big(\mathsf{Adv}^{\mathsf{CPA}}_{\Pi,\mathcal{B}_4}(\lambda) + \mathsf{Adv}^{\mathsf{dual\text{-}PRF\text{-}sec}}_{\mathsf{HKDF.Extract},\mathcal{B}_5}(\lambda) + \mathsf{Adv}^{\mathsf{sEUF\text{-}CMA}}_{\mathsf{HMAC},\mathcal{B}_6}(\lambda) + \mathsf{Adv}^{\mathsf{dual\text{-}PRF\text{-}sec}}_{\mathsf{HKDF.Extract},\mathcal{B}_7}(\lambda) \Big) + n_s \cdot \mathsf{Adv}^{\mathsf{dual\text{-}snPRF\text{-}ODH}}_{\mathsf{HKDF.Extract},\mathcal{B}_8}(\lambda) \Big)$$

where n_s *is the maximum number of sessions and* n_p *is the maximum number of pre-shared secrets.*

Proof. We conduct the proof as a sequence of games played between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$. We start with the original security game $\mathsf{G}^{\mathsf{Multi\text{-}Stage}}_{\mathcal{A},\mathsf{SRAH}}$ and transform it over a sequence of hybrid arguments to a game where the adverary's advantage is at most 0. Let Adv_i denote the adversary's advantage in the i-th game.

Game 0. We start with Game 0, which is the original security game $\mathsf{G}^{\mathsf{Multi\text{-}Stage}}_{\mathcal{A},\mathsf{SRAH}}$. By definition we have

$$\mathsf{Adv}_0(\lambda) = \mathsf{Adv}^{\mathsf{Multi\text{-}Stage}}_{\mathcal{A},\mathsf{SRAH}}(\lambda).$$

Game 1. This game is identical to Game 0 except we only allow $\mathcal{A}$ to issue a single Test query. As shown by Dowling *et al.* [7, Lemma A.1] the adversary's advantage is reduced by a factor of at most $\mathsf{M} \cdot n_s$ where n_s is the number of sessions and we can implicitly guess which session $\mathsf{label} = (U, V, k)$ is tested by $\mathcal{A}$. Thus, we have

$$\mathsf{Adv}_1(\lambda) \geq \frac{1}{2n_s} \cdot \mathsf{Adv}_0(\lambda).$$

In the following we will distinguish between three cases:

A. the adversary $\mathcal{A}$ tests a non-forward secure stage, i.e., $\mathcal{A}$ issues $\mathsf{Test}(\mathsf{label}, 1)$
B. the adversary $\mathcal{A}$ tests a forward secure stage, i.e., $\mathcal{A}$ issues $\mathsf{Test}(\mathsf{label}, 2)$, and the tested session label has no honest contributive partner in stage 2, i.e., there exists no session $\mathsf{label}' \neq \mathsf{label}$ with $\mathsf{label}'.\mathsf{cid}_2 = \mathsf{label}.\mathsf{cid}_2$
C. the adversary $\mathcal{A}$ tests a forward secure stage, i.e., $\mathcal{A}$ issues $\mathsf{Test}(\mathsf{label}, 2)$, and the tested session label has an honest contributive partner in stage 2, i.e., there exists a session $\mathsf{label}' \neq \mathsf{label}$ with $\mathsf{label}'.\mathsf{cid}_2 = \mathsf{label}.\mathsf{cid}_2$

Hence, we have

$$\mathsf{Adv}_1(\lambda) \leq \mathsf{Adv}^{\mathrm{NFS}}_1(\lambda) + \mathsf{Adv}^{\text{FS-without-partner}}_1(\lambda) + \mathsf{Adv}^{\text{FS-with-partner}}_1(\lambda).$$

Case A: Test in non-forward secure stage. We begin with the case where the adversary $\mathcal{A}$ tests a non-forward secure stage, i.e., $\mathcal{A}$ issues $\mathsf{Test}(\mathsf{label}, 1)$.

Game A.0. This game is identical to Game 1 except the adversary is restricted to testing in stage 1. Therefore, we have

$$\mathsf{Adv}_1^{\mathrm{NFS}}(\lambda) = \mathsf{Adv}_{A.0}(\lambda).$$

Game A.1. This game is identical to Game A.0 except we guess the pre-shared secret pss used in the tested session and abort the game if our guess was wrong. This reduces the adversary's advantage by a factor of at most n_p, where n_p is the number pre-shared secrets:

$$\mathsf{Adv}_{A.1}(\lambda) \geq \frac{1}{n_p} \cdot \mathsf{Adv}_{A.0}(\lambda).$$

Game A.2. This game is identical to Game A.1 except we guess the contributive partner session of the session label and abort the game if our guess was wrong. This reduces the adversary's advantage by a factor of at most n_s, where n_s is the number pre-shared secrets:

$$\mathsf{Adv}_{A.2}(\lambda) \geq \frac{1}{n_s} \cdot \mathsf{Adv}_{A.1}(\lambda).$$

Game A.3. This game is identical to Game A.2 except in any session using the pre-shared secret guessed in Game A.2 we replace the pre-shared secret with a uniformly random value $\widetilde{\mathsf{pss}} \xleftarrow{\$} \mathcal{P}$. We will now show that we can use any adversary $\mathcal{A}$ that is able to distinguish Game A.2 from Game A.3 to construct an adversary $\mathcal{B}_1$ against the CPA security of Π. The adversary $\mathcal{B}_1$ behaves exactly like the challenger in Game A.2 except for generating the pre-shared secret used in the session label. When the adversary $\mathcal{A}$ calls the $\mathsf{NewSecret}$ query to create the pre-shared secret used in the session label $\mathcal{B}_1$ first draws uniformly at random $\mathsf{pss} \xleftarrow{\$} \mathcal{P}$ and sets it as the freshly generated pre-shared secret. It then additionally draws $\widetilde{\mathsf{pss}} \xleftarrow{\$} \mathcal{P}$, outputs pss and $\widetilde{\mathsf{pss}}$ to its challenger, and outputs the received challenge ciphertext as the pre-shared secret identifier pssid. Note that $\mathcal{B}_1$ is able to simulate all queries by the adversary $\mathcal{A}$ independent of the challenge chiphertext. Moreover, if the challenger encrypts pss as the challenge, we then perfectly simulate Game A.2. If the challenger encrypts $\widetilde{\mathsf{pss}}$, the pre-shared secret used in the session label is a uniformly random value that is independent from the pre-shared secret identifier sent resp. received by the session label. Thus, in that case we perfectly simulate Game A.3. Therefore, if $\mathcal{A}$ is able to distinguish between Game A.2 and Game A.3, $\mathcal{B}_1$ is able to break the CPA security of Π and we have

$$|\mathsf{Adv}_{A.3}(\lambda) - \mathsf{Adv}_{A.2}(\lambda)| \leq \mathsf{Adv}_{\mathsf{CPA}}^{\Pi,\mathcal{B}_1}(\lambda).$$

Game A.4. This game is identical to Game A.3 except we replace the derivation of s_{MAC} in any session using the pre-shared secret $\widetilde{\mathsf{pss}}$ with a uniformly random value $\widetilde{s_{\mathsf{MAC}}} \xleftarrow{\$} \{0,1\}^*$. We will now show that we can use any adversary $\mathcal{A}$ that is able to distinguish Game A.3 from Game A.4 to construct an adversary $\mathcal{B}_2$

against the PRF-sec security of HKDF.Extract. The adversary $\mathcal{B}_2$ behaves exactly like the challenger in Game A.3 except that it uses its PRF oracle to derive s_{MAC}. $\mathcal{B}_2$ perfectly simulates Game A.3 if its PRF oracle computes HKDF.Extract and perfectly simulates Game A.4 if its PRF oracle is a random function and we have

$$|\mathsf{Adv}_{A.4}(\lambda) - \mathsf{Adv}_{A.3}(\lambda)| \leq \mathsf{Adv}^{\mathsf{HKDF.Extract},\mathcal{B}_2}_{\mathsf{PRF\text{-}sec}}(\lambda).$$

Game A.5. In this game we want to ensure that although the pre-shared secret used in the session label may be shared with multiple participants, none of these participants can accept in stage 1 if they are not the intended partner of the session label. To this end, we abort the game and raise an event $\mathsf{abort}_{A.5}$ if a session $\mathsf{label}' \neq \mathsf{label}$ with $\mathsf{label.pssid} = \mathsf{label}'.\mathsf{pssid}$ and $\mathsf{label.pid} \neq \mathsf{label}'.\mathsf{id}$ accepts in stage 1. Clearly, we have

$$\mathsf{Adv}_{A.5}(\lambda) = \mathsf{Adv}_{A.4}(\lambda) - \Pr[\mathsf{abort}_{A.5}].$$

We will now show that we can construct an adversary $\mathcal{B}_3$ that breaks the sEUF-CMA security of MAC with probability at least $\Pr[\mathsf{abort}_{A.5}]$. The adversary $\mathcal{B}_3$ behaves exactly like the challenger in Game A.4 except that it uses its MAC oracle to compute the MAC value for the session label and the corresponding session of the intended partner.

If $\mathsf{label.role} = \mathsf{responder}$, there exists only a single participant that uses the same pre-shared key as label in the initiator role, which is then the intended partner of the session label. Therefore, the event $\mathsf{abort}_{A.5}$ cannot occur if $\mathsf{label.role} = \mathsf{responder}$. If $\mathsf{label.role} = \mathsf{initiator}$ and the event $\mathsf{abort}_{A.5}$ occurs, this implies that some session $\mathsf{label}' \neq \mathsf{label}$ with $\mathsf{label.pid} \neq \mathsf{label}'.\mathsf{id}$ that uses the pre-shared key $\widetilde{\mathsf{pss}}$ received a message containing a valid MAC value. Since Match security guarantees that label can only be partnered with its intended partner, label' cannot be partnered with label. Since all messages output resp. received by a session are contained in the session's session identifier, the message received by label' must be different from the message output by label and thus it constitutes a valid MAC forgery. The adversary $\mathcal{B}_3$ hence outputs the message received by label' and the MAC value contained in it as a forgery to its challenger. This proves our claim and we have

$$\mathsf{Adv}_{A.5}(\lambda) \geq \mathsf{Adv}_{A.4}(\lambda) - \mathsf{Adv}^{\mathsf{MAC},\mathcal{B}_3}_{\mathsf{sEUF\text{-}CMA}}(\lambda).$$

Game A.6. This game is identical to Game A.5 except we replace the derivation of s_{tmp} in any session using the pre-shared secret $\widetilde{\mathsf{pss}}$ with a uniformly random value $\widetilde{s_{\mathsf{tmp}}} \xleftarrow{\$} \mathcal{S}$. We will now show that we can use any adversary $\mathcal{A}$ that is able to distinguish Game A.5 from Game A.6 to construct an adversary $\mathcal{B}_4$ against the PRF-sec security of HKDF.Extract. The adversary $\mathcal{B}_4$ behaves exactly like the challenger in Game 0 except that it uses its PRF oracle to derive s_{tmp}. Hence, $\mathcal{B}_4$ perfectly simulates Game A.5 if its PRF oracle computes HKDF.Extract and perfectly simulates Game A.6 if its PRF oracle is a random function and we have

$$|\mathsf{Adv}_{A.6}(\lambda) - \mathsf{Adv}_{A.5}(\lambda)| \leq \mathsf{Adv}^{\mathsf{HKDF.Extract},\mathcal{B}_4}_{\mathsf{PRF\text{-}sec}}(\lambda).$$

In Game A.6 the value $\widetilde{s_{\mathsf{tmp}}}$ of the tested session label is chosen uniformly at random. Hence, from the view of $\mathcal{A}$, Game A.6 is independent of the bit b_{test}, which results in

$$\mathsf{Adv}_{A.6}(\lambda) = 0.$$

Case B: Test without contributive partner. Next we consider the case where the adversary $\mathcal{A}$ tests a session label that has no honest contributive partner. In particular, we consider the case that label is an initiator session that has no contributive partner in stage 2, or that label is a responder session that has no contributive partner in stage 1.

Game B.0. This game is identical to Game 1 except that the adversary is restricted to issuing a query $\mathsf{Test}(\mathsf{label}, i)$ where no session $\mathsf{label}' \neq \mathsf{label}$ exists with $\mathsf{label}'.\mathsf{cid}_i = \mathsf{label}.\mathsf{cid}_i$ and where either $i = 1$ and $\mathsf{label}.\mathsf{role} = \mathsf{responder}$ or $i = 2$ and $\mathsf{label}.\mathsf{role} = \mathsf{initiator}$ Thus, we have

$$\mathsf{Adv}_1^{\text{FS-without-partner}}(\lambda) = \mathsf{Adv}_{B.0}(\lambda).$$

Game B.1. This game is identical to Game B.0 except we guess the pre-shared secret pss used in the tested session and abort the game if our guess was wrong. This reduces the adversary's advantage by a factor of at most n_p, where n_p is the number pre-shared secrets:

$$\mathsf{Adv}_{B.1}(\lambda) \geq \frac{1}{n_p} \cdot \mathsf{Adv}_{B.0}(\lambda).$$

Game B.2. This game is identical to Game B.1 except we abort the game and raise an event $\mathsf{abort}_{B.2}$ if $\mathsf{label}.\mathsf{role} = \mathsf{responder}$ and label accepts in stage 1 without a contributive partner or $\mathsf{label}.\mathsf{role} = \mathsf{initiator}$ and label accepts in stage 2 without a contributive partner. Therefore, we have

$$|\mathsf{Adv}_{B.2}(\lambda) - \mathsf{Adv}_{B.1}(\lambda)| \ \leq \Pr[\mathsf{abort}_{B.2}].$$

Since the game aborts if the session label accepts in stage i and $\mathcal{A}$ is restricted to issuing a query $\mathsf{Test}(\mathsf{label}, i)$ we have $\mathsf{Adv}_{B.2}(\lambda) = 0$. In the following we will now proceed to bound $\Pr[\mathsf{abort}_{B.2}]$.

Note that in Game B.2 the adversary $\mathcal{A}$ cannot issue a Corrupt query for the pre-shared secret pss used by the tested session label. This is implied from the following facts. First, if $\mathcal{A}$ tests a responder session, it cannot issue a Corrupt query since stage 1 is not forward secure. Second, if $\mathcal{A}$ tests an initiator sessions and corrupts pss before label accepts stage 2, $\mathsf{label}.\mathsf{rect\_auth}_2$ will be set to unauth and $\mathcal{A}$ then loses the game when it issues $\mathsf{Test}(\mathsf{label}, 2)$. Third, the game is aborted when label accepts in stage i. Thus, $\mathcal{A}$ can neither issue the Corrupt query prior to label accepting in stage i nor after label accepted in stage i. Since $\mathcal{A}$ cannot corrupt pss it is guaranteed that pss remains an unknown random value for $\mathcal{A}$.

Game B.3. This game is identical to Game B.2 except in any session using the pre-shared secret guessed in Game B.1 we replace the pre-shared secret with a uniformly random value $\widetilde{\mathsf{pss}} \xleftarrow{\$} \mathcal{P}$. As shown before, we can construct an adversary $\mathcal{B}_5$ that breaks the CPA security of Π using an adversary $\mathcal{A}$ that is able to distinguish Game B.3 and Game B.2 and we have

$$|\mathsf{Adv}_{B.3}(\lambda) - \mathsf{Adv}_{B.2}(\lambda)| \leq \mathsf{Adv}^{\Pi,\mathcal{B}_5}_{\mathsf{CPA}}(\lambda).$$

Game B.4. This game is identical to Game B.3 except we replace the derivation of s_{MAC} in any session using the pre-shared secret $\widetilde{\mathsf{pss}}$ with a uniformly random value $\widetilde{s_{\mathsf{MAC}}} \xleftarrow{\$} \{0,1\}^*$. As shown before, any adversary that is able to distinguish Game B.3 from Game B.4 can be used to construct an adversary $\mathcal{B}_6$ against the PRF-sec security of HKDF.Extract and we have

$$\Pr[\mathsf{abort}_{B.3}] \leq \Pr[\mathsf{abort}_{B.4}] + \mathsf{Adv}^{\mathsf{HKDF.Extract},\mathcal{B}_6}_{\mathsf{PRF\text{-}sec}}(\lambda).$$

As the final step we will now show that we can use any adversary $\mathcal{A}$ that triggers the event $\mathsf{abort}_{B.4}$ to construct an attacker $\mathcal{B}_7$ against the sEUF-CMA security of MAC. $\mathcal{B}_7$ behaves exactly like the challenger in Game B.4 except that it uses its MAC oracle to compute the MAC value for the session label and the corresponding session of the intended partner. Note that since $\widetilde{s_{\mathsf{MAC}}}$ is a uniformly random value this is a sound simulation of Game B.4 for $\mathcal{A}$. When the event $\mathsf{abort}_{B.4}$ is triggered the session label must have received some message m_{C} or m_{S} containing a valid MAC. holds some contributive identifier cid_i, which consists of either m_{C} or m_{C} and m_{S} depending on the role of label. Since there exists no session label' holding the same contributive identifier as label, at least one message contained in cid_i cannot be the output of an honest session. However, the session label only accepts in stage 2 if it received a message containing a valid MAC value. Thus, $\mathcal{B}_7$ outputs the MAC value received by the session label as a forgery to its sEUF-CMA challenger and we have

$$\Pr[\mathsf{abort}_{B.4}] \leq \mathsf{Adv}^{\mathsf{MAC},\mathcal{B}_7}_{\mathsf{sEUF\text{-}CMA}}(\lambda).$$

Case C: Test in forward secure stage with contributive partner. In the last case the adversary $\mathcal{A}$ tests a forward secure stage, i.e., $\mathcal{A}$ issues $\mathsf{Test}(\mathsf{label}, 2)$, and the tested session label has an honest contributive partner in stage 2, i.e., there exists a session $\mathsf{label}' \neq \mathsf{label}$ with $\mathsf{label}'.\mathsf{cid}_2 = \mathsf{label}.\mathsf{cid}_2$.

Game C.0. This game is identical to Game 1 except that the adversary is restricted to testing a session in stage 2 with a contributive partner. Thus, we have

$$\mathsf{Adv}^{\text{FS-with-partner}}_1(\lambda) = \mathsf{Adv}_{C.0}(\lambda).$$

Game C.1. This game is identical to Game C.0 except we guess as session label' uniformly at random and abort the game if the guessed session is not the contributive partner of the session label in stage 2 and we have

$$\mathsf{Adv}_{C.1}(\lambda) \geq \frac{1}{n_s} \cdot \mathsf{Adv}_{C.0}(\lambda).$$

Game C.2. In this game we want to replace the derived main secret in the session label with a uniformly random value. First, note that if the tested session label is a client session, we have $\mathsf{label.cid}_2 = \mathsf{label}'.\mathsf{cid}_2 = (m_\mathrm{C}, m_\mathrm{S})$ when label accepts in stage 2. Thus, we are guaranteed that label and label' use the same Diffie–Hellman shares g^x and g^y to derive s_{main}. However, if the tested session label is a server session, we only have $\mathsf{label.cid}_2 = \mathsf{label}'.\mathsf{cid}_2 = (m_\mathrm{C})$ when label accepts in stage 2. The adversary may then replace the Diffie–Hellman share g^y of the session label with some other Diffie–Hellman share $g^{y'}$ in the message m_S. Note that the MAC value in m_S cannot prevent this modification since the adversary $\mathcal{A}$ may corrupt pss and is therefore able to compute a valid MAC value over the modified Diffie–Hellman share. Hence, we must be able to compute the Diffie–Hellman value $g^{xy'}$ for an arbitrary $g^{y'} \neq g^y$ and for this reason we model the security of $\mathsf{HKDF.Extract}$ using the $\mathsf{PRF\text{-}ODH}$ assumption.

Formally, this game is identical to Game C.1 except we replace the value s_{main} with a uniformly random value $\widetilde{s_{\mathsf{main}}} \xleftarrow{\$} \mathcal{S}$. We will now show that we can use any adversary $\mathcal{A}$ that is able to distinguish Game C.1 from Game C.2 to construct an adversary $\mathcal{B}_8$ against the $\mathsf{PRF\text{-}ODH}$ security of $\mathsf{HKDF.Extract}$. The adversary $\mathcal{B}_8$ behaves exactly like the challenger in Game 0 except that at the start of the game it outputs pss to its $\mathsf{PRF\text{-}ODH}$ challenger and then uses the obtained Diffie–Hellman shares g^x and g^y as the Diffie–Hellman shares for the sessions label and label'. Moreover, $\mathcal{B}_8$ uses the obtained PRF challenge as the main secret for the session label and for label' if label' becomes partnered with label. If the session label' receives a Diffie–Hellman share $g^{y'} \neq g^y$, $\mathcal{B}_8$ uses its ODH_u query to derive the main secret from $g^{xy'}$. This results in $\mathcal{B}_8$ perfectly simulating Game C.1 if its oracle ODH_u computes $\mathsf{HKDF.Extract}$ and perfectly simulates Game C.2 if ODH_u is a random function and we have

$$|\mathsf{Adv}_{C.2}(\lambda) - \mathsf{Adv}_{C.1}(\lambda)| \leq \mathsf{Adv}^{\mathsf{HKDF.Extract},\mathcal{B}_8}_{\mathsf{PRF\text{-}ODH}}(\lambda).$$

In Game C.2 the value $\widetilde{s_{\mathsf{main}}}$ of the tested session label is chosen uniformly at random. Therefore, from the view of $\mathcal{A}$, Game C.2 is independent of the bit b_{test}, which results in

$$\mathsf{Adv}_{C.2}(\lambda) = 0.$$

References

1. Ghada Arfaoui, Xavier Bultel, Pierre-Alain Fouque, Adina Nedelcu, and Cristina Onete. The privacy of the TLS 1.3 protocol. *PoPETs*, 2019(4):190–210, October 2019.
2. Nimrod Aviram, Kai Gellert, and Tibor Jager. Session resumption protocols and efficient forward security for TLS 1.3 0-RTT. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages 117–150. Springer, Heidelberg, May 2019.
3. Nimrod Aviram, Kai Gellert, and Tibor Jager. Session resumption protocols and efficient forward security for TLS 1.3 0-RTT. *Journal of Cryptology*, 34(3):1–57, 2021.

4. Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In Neal Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 1–15. Springer, Heidelberg, August 1996.
5. Jacqueline Brendel, Marc Fischlin, Felix Günther, and Christian Janson. PRF-ODH: Relations, instantiations, and impossibility results. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 651–681. Springer, Heidelberg, August 2017.
6. Denis Diemert and Tibor Jager. On the tight security of TLS 1.3: Theoretically sound cryptographic parameters for real-world deployments. *Journal of Cryptology*, 34(3):1–57, 2021.
7. Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the tls 1.3 handshake protocol. Cryptology ePrint Archive, Report 2020/1044, 2020. `https://eprint.iacr.org/2020/1044`.
8. Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 1197–1210. ACM Press, October 2015.
9. Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 draft-10 full and pre-shared key handshake protocol. Cryptology ePrint Archive, Report 2016/081, 2016. `http://eprint.iacr.org/2016/081`.
10. Nir Drucker and Shay Gueron. Selfie: reflections on TLS 1.3 with PSK. *Journal of Cryptology*, 34(3):1–18, 2021.
11. Marc Fischlin and Felix Günther. Replay attacks on zero round-trip time: The case of the TLS 1.3 handshake candidates. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*, pages 60–75. IEEE, 2017.
12. Marc Fischlin and Felix Günther. Replay attacks on zero round-trip time: The case of the TLS 1.3 handshake candidates. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 60–75. IEEE, 2017.
13. Tibor Jager, Florian Kohlar, Sven Schäge, and Jörg Schwenk. On the security of TLS-DHE in the standard model. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 273–293. Springer, Heidelberg, August 2012.
14. H. Krawczyk and P. Eronen. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869, IETF, May 2010.
15. Hugo Krawczyk. Cryptographic extraction and key derivation: The HKDF scheme. In Tal Rabin, editor, *CRYPTO 2010*, volume 6223 of *LNCS*, pages 631–648. Springer, Heidelberg, August 2010.
16. Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-hashing for message authentication. IETF Internet Request for Comments 2104, February 1997.
17. E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, IETF, August 2018.
18. Erik Sy, Moritz Moennich, Tobias Mueller, Hannes Federrath, and Mathias Fischer. Enhanced performance for the encrypted web through tls resumption across hostnames. In *Proceedings of the 15th International Conference on Availability, Reliability and Security*, ARES '20, New York, NY, USA, 2020. Association for Computing Machinery.